Confidentiality Policies and Their Extraction from Programs

Michael Carl Tschantz Jeannette M. Wing

Computer Science Department, Carnegie Mellon University
E-mail: {mtschant, wing}@cs.cmu.edu

Abstract

We examine a well known confidentiality requirement called noninterference *and argue that many systems do not meet this requirement despite maintaining the privacy of its users. We discuss a weaker requirement called* incident-insensitive noninterference *that captures why these systems maintain the privacy of its users while possibly not satisfying noninterference. We extend this requirement to depend on dynamic information in a novel way. Lastly, we present a method based on model checking to extract from program source code the dynamic incident-insensitive noninterference policy that the given program obeys.*

1 Introduction

Given a multi-user system, a user might wonder how it protects his privacy. Such a user would benefit from a summary of who else may use the system to access his information and under what conditions. We hope to develop a tool that automatically produces such a summery, or dynamic confidentiality policy, from the source code of the program controlling such a system. Before we may describe an approach to this problem, we must first consider what it means for a user's information to remain confidential.

Confidentiality Requirements. What must a system keep secret to maintain the privacy of its users? No single answer is correct for all systems: different balances of privacy and functionality result in systems with different confidentiality guarantees.

Consider a system with a high-level user H and a low-level user L, whom H does not trust. The user H desires that the system guarantees that the user L has no way of learning about the inputs of H to the system. This guarantee may be formalized as a *confidentiality assertion*. Such a formalization must make clear what exactly it means for the untrusted user L to learn about an input of H. Each different formalization of this concept corresponds to a different *confidentiality requirement*.

One of the most well known and earliest confidentiality requirements is *noninterference* as defined by Goguen and Meseguer [7] and later extended to nondeterministic systems by McCullough [20, 21]. Informally, the confidentiality assertion that the user H is noninterfering with the user L requires that the set of possible outputs seen by L is the same regardless of any inputs provided by H to the system. This requirement is so strong that the user L may not even know if H has provided any inputs to the system.

Such a strong requirement is often too stringent, that is, it places so much emphasis on privacy that it prevents some systems from achieving a reasonable level of functionality. In many realistic systems, allowing the user L to know that the user H has entered an input into the system is acceptable as long as L does not learn about the contents of the input. In Section 3, we provide examples of such systems and present a weakened form of noninterference that allows L to learn that H has provided inputs to the system while still protecting the contents of these inputs. We also formalize a weakened confidentiality requirement based on this observation that we call *incident-insensitive noninterference* since the user L is allowed to learn of the incident of the input. Likewise, we call the original noninterference requirement of Goguen and Meseguer *incident-sensitive noninterference*.

Dynamic Confidentiality Assertions. The confidentiality assertions described thus far have been static: they hold between two users regardless of their actions. Often static requirements cannot capture the confidentiality guarantee that a system should make to its users. For example, consider a system that stores emails for its users. The system should not allow a user to read any of the emails unless that user provides the correct password. To formally capture such a guarantee requires a *dynamic confidentiality assertion*, an assertion that some confidentiality requirement should hold between two users unless some condition that depends on dynamic information is met at runtime.

Along with noninterference, Goguen and Meseguer introduced a form of dynamic confidentiality assertion [7]. A dynamic assertion of their form declares that an input from a high-level user H should remain unknown to a low-level user L unless some predicate holds of the inputs that preceded the input in question. Since the dynamic assertion may only depend on the inputs that precede the input in question, we term their formulation *at-input-checking*.

In the password example above, the dynamic assertion should hold unless the user L enters the correct password, an event that might occur *after* the system has already received an email (from H). Since at-input-checking assertions may depend only on inputs received *before* such an email arrived, they cannot capture the needed assertion.

To fix this problem, we remove the requirement that the predicate of a dynamic assertion may only depend on inputs that precede the input in question. Under our formulation, a dynamic assertion will require that an input be protected until enough dynamic information is collected to rule otherwise. This information may come at anytime as long the input in question does not affect any outputs to the user L until it arrives. If such input never comes, the input will always be protected. We term this formulation *at-output-checking* since at the time of an output, all the inputs that have arrived may affect whether the output may depend on some previous input, rather than just those inputs that preceded the input in question. In Section 4, we formalize this new form of dynamic assertion.

Policy Extraction. A set of dynamic incident-insensitive noninterference (DIINI) assertions defines a DIINI policy. Given a DIINI policy, a programmer can take two different approaches to ensuring that a program obeys the policy. In the first approach, the programmer codes with the policy in mind and manually inserts any dynamic checks that the program must perform to ensure that the policy is obeyed. In the second approach, the programmer abstracts the policy enforcement mechanism from core application logic of the program and configures the program with an explicit representation of the policy.

While the first approach is usually easier to implement, the second approach has many advantages. Firstly, an organization with an explicit policy may apply that policy to multiple programs. Secondly, the decoupling of policy from application logic allows multiple organizations with differing confidentiality policies to use a single program since each organization may separately configure the program to enforce its policy. Thirdly, having a centralized policy facilitates reasoning about the policy and editing it.

To gain these advantages for legacy programs written using the first approach, the program maintainers should convert them to use a explicit policy as in the second approach. A tool that aggregates the manually inserted dynamic checks used to ensure that the program obeys the policy together into an explicit representation of this policy would ease this conversion, especially for large programs.

Many other uses for such a tool exist. A system administrator could examine the extracted policy by hand or use tools to answer queries about the policy. Furthermore, such a tool could verify that an extracted policy meets the requirements of a specified policy. Even in the absence of a formal specification, change-impact analysis is possible: given application code before and after some set of edits, one could compare the extracted policies to ensure that the program edits has introduced no new security holes.

In Section 5, we present an approach based on model checking for this policy extraction problem. Our approach tracks the flow of information through the program in a manner similar to type systems that track information flow [29]. However, our approach allows the same variable to carry both high- and low-level information without the low-level information being considered high-level preventing an overly conservative analysis. Furthermore, our approach attempts to rule out infeasible paths. While these features matter little in the context of writing a program with type analysis in mind, they become important in our primary use case of extracting policies from legacy code.

Road Map and Contributions. The order of this paper mirrors the development of this introduction: After handling some technical preliminaries in Section 2, we motivate and present incident-insensitive noninterference in Section 3. Then we present our formulation of dynamic confidentiality assertions in Section 4. With the notation of dynamic confidentiality policy fully formalized, we at last return to the original motivation of this work, automated policy extraction, in Section 5. Lastly, we cover related work.

The three main sections of this work each represent a separate contribution:

- Section 3 motives the need for incident-insensitive noninterference clarifies its relation to the original definition of noninterference. (Since a similar confidentiality requirement has appeared in the literature before [25], we do not consider the presentation of the requirement to be our contribution *per se*.)

- Section 4 motives the need for and presents a more general notation of dynamic confidentiality assertion, which allows for the expression of realistic policies.

- Section 5 provides an approach to automated policy extraction.

An additional contribution is that of unwinding conditions for incident-insensitive noninterference in both its static and dynamic form. Using unwinding conditions eases proving that a system satisfies a noninterference policy. We demonstrate their usefulness by employing them to prove the correctness of our approach to policy extraction.

2 The System Model

Automata. The input-output behavior of a system determines what confidentiality assertions it satisfies. Agents acting in various *security domains* create the inputs and receive the outputs. Each domain is a different entity or class of entities that might interact with the system. For example, the domains might be Top Secret, Secret, Classified, and Unclassified for modeling the flow of information between security classes in a military system and if the actual identity of the entity is irrelevant (only its security clearance matters). For modeling the use of different resources, the domains might be Hard Drive, Network, and User.

These domains interact with one another by using the system. We will model such a system as an automaton. Formally, an *system automaton* m consists of

- a set of inputs I,
- a set of outputs O such that $I \cap O = \emptyset$,
- a set of domains D,
- a function $\mathsf{dom} : A \to D$ that assigns to each action the domain that created or received it where the set of actions A is $I \cup O$,
- a set of states Q,
- a start state $q_0 \in Q$, and
- a transition relation $\to \subseteq Q \times A \times Q$.

We write $q_1 \overset{a}{\to} q_2$ if $\langle q_1, a, q_2 \rangle \in \to$. We write $q_1 \overset{\alpha}{\to} q_2$ for $\alpha \in A^*$ if either

- $\alpha = []$ and $q_1 = q_2$, or
- $\alpha = a{:}\alpha'$, $q_1 \overset{a}{\to} q_1'$, and $q_1' \overset{\alpha'}{\to} q_2$

where $[]$ is the empty sequence and $a{:}\alpha$ is the sequence formed by prepending a to α. (For example, $\mathsf{a}{:}\mathsf{b}{:}[] = [\mathsf{a}, \mathsf{b}]$.) Since we never have a list of lists, we abuse notation and also use : to append lists and to add elements to their end.

The above automaton model is asynchronous and nondeterministic, which greatly complicates proofs about them. We use asynchronous automata since programs often produce output for one user without producing it for other users. We require nondeterminism since we have the end goal of model checking in mind and model checking works over a nondeterministic abstraction of an actual program.

Behaviors. Let the set of behaviors of an automaton $m = \langle I, O, D, \mathsf{dom}, Q, q_0, \to \rangle$ be

$$\mathsf{behv}(m) = \{\, \alpha \in A^* \mid \exists q \in Q \text{ s.t. } q_0 \overset{\alpha}{\to} q \,\}$$

Each behavior represents one way in which the system might operate. Since each domain has control over its input actions, each domain may affect which behaviors the system can execute. Let $\iota \in I^*$ represent a sequence of inputs. If the system is subjected to the inputs of ι and no other inputs, then the system may only execute those behaviors that include all the inputs of ι in order and no other inputs.

To formalize this notion, let us first define the *restrict* function $\lfloor \cdot \rfloor_{\cdot} : A^* \times 2^A \to A^*$. The restrict function takes a sequence α and a subset A' of A and returns the sequence $\lfloor \alpha \rfloor_{A'}$ which only includes the elements of α that are in A'. Let $\lfloor \alpha \rfloor_{A'}$ be defined as follows:

$$\lfloor [] \rfloor_{A'} = []$$

$$\lfloor a{:}\alpha \rfloor_{A'} = \begin{cases} a{:}(\lfloor \alpha \rfloor_{A'}) & \text{if } a \in A' \\ \lfloor \alpha \rfloor_{A'} & \text{otherwise} \end{cases}$$

where $A' \in 2^A$. For example, $\lfloor [\mathsf{a}, \mathsf{c}, \mathsf{a}] \rfloor_{\{\mathsf{a},\mathsf{b}\}} = [\mathsf{a}, \mathsf{a}]$ and $\lfloor [\mathsf{c}, \mathsf{a}, \mathsf{b}, \mathsf{c}, \mathsf{a}] \rfloor_{\{\mathsf{a},\mathsf{b}\}} = [\mathsf{a}, \mathsf{b}, \mathsf{a}]$.

The set of behaviors that are possible given a sequence that provides all the inputs to the system m is given by $\mathsf{runs} : I^* \to 2^A$ where

$$\mathsf{runs}(\iota) = \{\, \alpha \in \mathsf{behv}(m) \mid \lfloor \alpha \rfloor_I = \iota \,\}$$

A domain d cannot observe all the actions of a system: d can only observe those actions a such that $\mathsf{dom}(a) = d$. Thus, if the system executes a behavior α, then the domain d only sees the sequence of actions $\lfloor \alpha \rfloor_{A^d}$ where $A^d = \{\, a \in A \mid \mathsf{dom}(a) = d \,\}$. If two behaviors α_1 and α_2 are such that $\lfloor \alpha_1 \rfloor_{A^d} = \lfloor \alpha_2 \rfloor_{A^d}$, then α_1 and α_2 provide the domain d with the same observations and thus look the same to domain d. In general, if a domain d sees the action sequence α, d will only be able to tell that some behavior α' such that $\lfloor \alpha' \rfloor_{A^d} = \alpha$ was executed; d will not know which one.

Let us raise $\lfloor \cdot \rfloor_{\cdot}$ to work over sets of sequences as follows: $\lfloor \{\alpha_1, \alpha_2, \ldots\} \rfloor_{A'} = \{\lfloor \alpha_1 \rfloor_{A'}, \lfloor \alpha_2 \rfloor_{A'}, \ldots\}$. Then, if for two input sequences ι_1 and ι_2 of a system m, $\lfloor \mathsf{runs}(\iota_1) \rfloor_{A^d} = \lfloor \mathsf{runs}(\iota_2) \rfloor_{A^d}$, then domain d cannot tell between when ι_1 or ι_2 is the input sequence to m. This provides an opportunity to prevent a domain from learning the inputs of another domain.

Adding Internal Transitions. A problem with the above automation model is that each transition either results in output or is the result of input. This limitation does not allow internal transitions. To allow internal transitions, we allow a distinguished action $\tau \notin A$ that represents an action that no domain can observe. If the users may deduce the execution of an internal transition (perhaps by timing), then this model is inappropriate.

Since users cannot observe internal transitions, they should not show up in the behaviors of a system and we must redefine behv with this in mind. Let $q \xrightarrow{a} q'$ iff $q \overset{a}{\to} q'$, or both $q \overset{\tau}{\to} q''$ and $q'' \xrightarrow{a} q'$ for some q'' Let

$$\mathsf{behv}(m) = \{\, \alpha \in A^* \mid \exists q \in Q \,\mathrm{s.t.}\, q_0 \xrightarrow{\alpha} q \,\}$$

where $\longrightarrow$ is raised to sequences α in the same manner as $\to$ was.

3 Noninterference

3.1 What is Confidentiality?

Consider the following simple program:

```
bool in = load("secret-file.db");
print(‘x’);
```

The first line reads in the contents of a secret file. The second line simply prints the character ‘x’ to the low-level user. If we model the reading of the secret file as receiving input from a high-level user, then this program fails to meet the requirements of incident-sensitive noninterference as defined by Goguen and Meseguer [7]. The reason is that the low-level user does not see the output ‘x’ unless the high-level user produces input, which allows the `load` statement to stop blocking and terminate. Thus, the low-level user has learned that the high-level user has interacted with the system. This violation occurs even though the low-level user clearly does not learn anything about the contents of `secret-file.db`. (We formalize this example in Appendix A.1.)

We believe that in many cases allowing the low-level user to know that the high-level user is interacting with the system is acceptable as long as the low-level user does not learn the contents of these interactions. Consider the following realistic examples:

- “Upon startup, a web server for online banking receives financial records from a secure database before answering any queries from users.”

  This web server violates incident-sensitive noninterference since if the web server answers the user’s queries with low-level outputs, the user will know that the server has consumed high-level input from the database. This violation holds even if the inputs consumed from the high-level database did not influence the server’s response to the low-level user. However, such a system maintains an acceptable level of confidentiality since the low-level user cannot learn what inputs the high-level database provided to the server and the low-level user learning that server has received high-level input only tells the low-level user that system is working correctly.

- “A student is applying for graduate school online. During the application process, both the student and the professors recommending him must enter information into the application database. Once the recommending professors have finished, the student receives a notice stating that the graduate school has received his recommendations. The applicant is not allowed access to his recommendation.”

  The low-level student only receives the notice if the professors have entered their high-level recommendations. Thus, by receiving the notice, the student learns that the system has consumed high-level inputs. This violates incident-sensitive noninterference even if the content of the high-level recommendation does not affect the content of the notice.

- “PhoneBook is a system produced by NS that organizes phone numbers for a law firm. While adding a new contact, PhoneBook reaches an error state. PhoneBook offers to send a bug report to NS stating only that the system failed to add a new contact. The law firm considers any personally identifiable information about its contacts to be private.”

  Since the error state was reached during the addition of contact information, the bug report indicates that the system was receiving high-level contact information. Thus, even if the bug report maintains the privacy of the contacts by not providing any information about them, the system will still violate incident-sensitive noninterference by sending the bug report to NS, which is low-level.

- “A physician uses a computer to record his interactions with patients. The physician enters into the computer both the treatment rendered and the fee charged (the physician negotiates the fee with each patient). The system should only allow the physician to access the treatment. However, the system provides the fee to his secretary for billing.”

  Since the low-level secretary receives a notice to bill a patient from the system, he knows that the physician has entered into the system a high-level input describing the treatment. This knowledge implies a violation of incident-sensitive noninterference even if the notice does not reveal any information about the treatment.

From these examples, it should be clear that often simply learning that some high-level input has taken place does not provide the low-level user with enough information to constitute a violation of the high-level user’s confidentiality. However, most confidentiality requirements (e.g., restrictiveness [20, 21, 22] and separability [23]) are *incident sensitive*: they prohibit low-level users from learning that any high-level input has taken place.

What we desire are *incident-insensitive* requirements, ones that allow low-level users to learn that high-level input has taken place while protecting the *contents* of these high-level inputs. Intuitively, a system obeys incident-insensitive noninterference if the content of inputs from a high-level user has no effect on the outputs that a low-level user sees. To make this slightly more formal, incident-insensitive noninterference requires that the set of possible outputs seen by a low-level user is the same regardless of the content of the inputs from high-level users. Note that the low-level user is, however, allowed to learn that the high-level user sent inputs to the system.

Incident-insensitive requirements have appeared in works on information-flow type systems (Sabelfeld and Myers provide a survey [29]). O'Neill et al. have proved that these type systems ensure that a program obeys an incident-insensitive requirement they simply call "noninterference" [25].

The rest of this section formalizes a slightly weaker form of O'Neill's noninterference. We delay describing how our formulation is weaker than O'Neill's until Section 6.

3.2 Noninterference Formalized

First we present policies in general. Then we present the statement of incident-sensitive noninterference as defined by McCullough for nondeterministic systems [20, 21]. After showing our definition for incident-insensitive noninterference, we compare the two.

Policies. For a system m, a *generic confidentiality policy* $\leadsto$ is an reflexive, transitive relation on D. We write $d_{\mathsf{f}} \not\leadsto d_{\mathsf{t}}$ iff $\neg(d_{\mathsf{f}} \leadsto d_{\mathsf{t}})$. If $d_{\mathsf{f}} \not\leadsto d_{\mathsf{t}}$, then information about d_{f} should not flow to d_{t}. A generic policy does not specify exactly what it means for information to flow. That is, a generic policy does not specify a confidentiality requirement.

Below, we formalize two confidentiality requirements that can give a generic policy meaning: incident-sensitive noninterference and incident-insensitive noninterference. Since these two requirements may be viewed as two different interpretations that one may assign to a generic policy, we represent policies of either type using $\leadsto$ as with generic policies and let the surrounding text make clear which type of policy it is.

Incident-Sensitive Noninterference. Let $\cong^{\leadsto,d}_{\text{IS}}$ be a relation on input sequences such that $[] \cong^{\leadsto,d}_{\text{IS}} []$, and $i_1{:}\iota_1 \cong^{\leadsto,d}_{\text{IS}} i_2{:}\iota_2$ iff

- $i_1 = i_2$ and $\iota_1 \cong^{\leadsto,d}_{\text{IS}} \iota_2$,
- $\mathsf{dom}(i_1) \not\leadsto d$ and $\iota_1 \cong^{\leadsto,d}_{\text{IS}} i_2{:}\iota_2$, or
- $\mathsf{dom}(i_2) \not\leadsto d$ and $i_1{:}\iota_1 \cong^{\leadsto,d}_{\text{IS}} \iota_2$.

A system m obeys $\leadsto$ as an incident-sensitive noninterference policy iff for all $d \in D$ and $\iota_1, \iota_2 \in I^*$,

$$\iota_1 \cong^{\leadsto,d}_{\text{IS}} \iota_2 \text{ implies } \lfloor \mathsf{runs}(\iota_1) \rfloor_{A^d} \subseteq \lfloor \mathsf{runs}(\iota_2) \rfloor_{A^d}$$

Intuitively, this definition says that if ι_1 has been received by the system and d should not be able to rule out the possibility that it was ι_2 that the system received, then there must exist no behavior of the system under ι_1 that is impossible under ι_2 from the perspective of d.

Incident-Insensitive Noninterference. Let $\cong^{\leadsto,d}_{\text{II}}$ be a relation on input sequences such that $[] \cong^{\leadsto,d}_{\text{II}} []$ and $i_1{:}\iota_1 \cong^{\leadsto,d}_{\text{II}} i_2{:}\iota_2$ iff

- $\mathsf{dom}(i_1) = \mathsf{dom}(i_2)$,
- $\mathsf{dom}(i_1) \leadsto d$ implies $i_1 = i_2$, and
- $\iota_1 \cong^{\leadsto,d}_{\text{II}} \iota_2$.

A system m obeys a policy $\leadsto$ as an incident-insensitive noninterference policy iff for all $d \in D$ and $\iota_1, \iota_2 \in I^*$,

$$\iota_1 \cong^{\leadsto,d}_{\text{II}} \iota_2 \text{ implies } \lfloor \mathsf{runs}(\iota_1) \rfloor_{A^d} \subseteq \lfloor \mathsf{runs}(\iota_2) \rfloor_{A^d}$$

Comparison. Note that for all d and $\leadsto$, both $\cong^{\leadsto,d}_{\text{IS}}$ and $\cong^{\leadsto,d}_{\text{II}}$ are equivalence relations. They are also alike in that if $\mathsf{dom}(i_1) \leadsto d$ and $\mathsf{dom}(i_2) \leadsto d$, both require that $i_1 = i_2$ for $i_1{:}\iota_1 \cong^{\leadsto,d}_{\text{II}} i_2{:}\iota_2$ or $i_1{:}\iota_1 \cong^{\leadsto,d}_{\text{IS}} i_2{:}\iota_2$ to hold. However, if $\mathsf{dom}(i_1) \not\leadsto d$, then $\cong^{\leadsto,d}_{\text{II}}$ still requires that $\mathsf{dom}(i_1) = \mathsf{dom}(i_2)$ whereas $\cong^{\leadsto,d}_{\text{IS}}$ makes no requirements at all and simply drops i_1 from consideration. This difference is the difference between incident-sensitive noninterference and incident-insensitive noninterference.

Since $\cong^{\leadsto,d}_{\text{II}}$ places more requirements on i_1 than $\cong^{\leadsto,d}_{\text{IS}}$, it should come as no surprise that $\iota_1 \cong^{\leadsto,d}_{\text{II}} \iota_2$ implies $\iota_1 \cong^{\leadsto,d}_{\text{IS}} \iota_2$ (see Lemma 3 in Appendix A.2). A direct result of this follows:

Theorem 1. *If a system obeys a generic policy $\leadsto$ as an incident-sensitive noninterference policy, then it will obey $\leadsto$ as an incident-insensitive noninterference policy; the converse is not true.*

Appendix A.2 provides a proof.

A specification may place both an incident-insensitive noninterference policy and an incident-sensitive noninterference policy on the same system. A specification might require that some users be incident-sensitively noninterfering with a second group of users and incident-insensitively noninterfering with a third group. The above theorem makes clear the relationship between these two policies.

3.3 Unwinding

Since noninterference is a global property, proving that a nontrivial system obeys a given policy is a daunting task. Thus, Goguen and Meseguer provided a property, the existence of an *unwinding relation*, to ease this task [8]. We provide such a property for incident-insensitive noninterference.

Let $q \xrightarrow[d]{a} q'$ iff

- $q \overset{a}{\to} q'$;
- $q \overset{\tau}{\to} q''$ and $q'' \xrightarrow[d]{a} q'$; or
- there exists $o \in O$ such that $\mathsf{dom}(o) \neq d$, $q \overset{o}{\to} q''$, and $q'' \xrightarrow[d]{a} q'$.

Informally, $q \xrightarrow[d]{a} q'$ means that the automaton can transition from q to q' by using only internal transitions, transitions that produce output for a domain other than d, and finally one transition using a.

Given a system automaton, let a *view partition* be a function from a domain to an equivalence relation on states. That is, a view partition is in $D \to 2^{Q \times Q}$. We will write $q_1 \overset{d}{\sim} q_2$ if for the domain d, the states q_1 and q_2 are within the relation.

Let a view partitioning for a program automaton m and policy $\rightsquigarrow$ be called an *incident-insensitive unwinding relation* if it satisfies the following *unwinding conditions*:

1. Local Respect: for all $d \in D$, $q, q'_1 \in Q$, and $i_1, i_2 \in I$, if $\mathsf{dom}(i_1) = \mathsf{dom}(i_2)$, $\mathsf{dom}(i_1) \not\rightsquigarrow d$ and $q \xrightarrow[d]{i_1} q'_1$, then there must exist $q'_2 \in Q$ such that $q \xrightarrow[d]{i_2} q'_2$ and $q'_1 \overset{d}{\sim} q'_2$.
2. Step Consistency: for all $d \in D$, $q_1, q'_1, q_2 \in Q$, and $i \in I$, if $q_1 \overset{d}{\sim} q_2$ and $q_1 \xrightarrow[d]{i} q'_1$, then there must exist $q'_2 \in Q$ such that $q_2 \xrightarrow[d]{i} q'_2$ and $q'_1 \overset{d}{\sim} q'_2$.
3. Output Consistency: for all $d \in D$, $q_1, q'_1, q_2 \in Q$, and $o \in O$, if $\mathsf{dom}(o) = d$, $q_1 \overset{d}{\sim} q_2$, and $q_1 \xrightarrow[d]{o} q'_1$, then there must exist $q'_2 \in Q$ such that $q_2 \xrightarrow[d]{o} q'_2$ and $q'_1 \overset{d}{\sim} q'_2$.

The above unwinding conditions are much more complex than the standard ones presented for incident-sensitive noninterference. However, incident-insensitivity is not blame: this actually stems from using asynchronous, nondeterministic automata for our system model instead of synchronous, deterministic automata.

Theorem 2. *If there exists an incident-insensitive unwinding relation for a incident-insensitive noninterference policy given an automaton, then that automaton obeys the policy.*

Appendix A.3 offers the proof.

4 Dynamic Policies

4.1 Motivation

Now we motivate the need for dynamic confidentiality assertions by relating in more detail the email server example from the introduction. As described before, the server should only allow access to the emails if the user supplies the correct password. The following program written in a C-like language enforces this requirement:

```
emails = load("mbox");
real_pw = load("password");
given_pw = read();
if(given_pw == real_pw)
  print(emails);
else
  print("wrong");
```

where the file `"mbox"` holds the emails and `"password"` holds the correct password.

To model this program, let the emails be represented by the domain e, the password by the domain p, and user by the domain u. Since the user u can gain access to the emails e by entering the correct password, the system does not obey any policy $\rightsquigarrow$ such that $\mathsf{e} \not\rightsquigarrow \mathsf{u}$. However, such a static policy fails to convey the design goal of only allowing the user access to the emails if he provides the correct password. We desire a policy that captures how supplying the correct password at runtime changes the allowed information flows.

To address such concerns, Goguen and Meseguer presented a dynamic version of incident-sensitive noninterference [7]. Informally, it allows an input from a high-level domain to be treated as insecure (accessible to the low-level domain) if the inputs that precede it satisfy some predicate. This allows the security of an input to depend on the inputs provided before it at runtime. Since all the information on which the security of an input may depend is present at the time that the input enters the system, we call their formulation *at-input-checking*.

The inability of at-input-checking to consider information that follows the input in question limits the expressiveness of at-input-checking. In the above example, the emails were the first input to the system. Since no input precedes the emails and the security of an input may only depend on those inputs that precede the input in question, the emails must either always be secure or always be insecure. This has the same problem as static policies: we cannot have the emails be secure in some behaviors of the system and insecure in others.

To fix this problem, we must allow the security of an input to depend on inputs that arrive after it. In this case, the security of the emails is undetermined until the user has entered his input. It may seem that such information comes too late: How can information from the future be used to determine the security of an input now? The answer is that the determination need not be made when the input has just arrived: as long as the input is treated as though it is secure until information becomes available indicating otherwise, this determination may be delayed.

To make use of this observation, we define a new version of dynamic policy that depends not only on the inputs that precede the input in question, but also those inputs that follow it. At the time of an output, whether that output may provide information about an input depends on all the inputs that precede that output, not just those the precede the input in question. Thus, we call our formulation *at-output-checking*.

4.2 Formalization

Dynamic Policies. Let a *generic dynamic policy* be a function from an input sequence to a static generic policy (a relation on domains). Give the set of inputs I and domains D, the set of possible generic dynamic policies is $I^* \to 2^{D \times D}$. Given a dynamic policy $\leadsto$ we write $d_\mathsf{f} \leadsto^{\iota} d_\mathsf{t}$ if ι is mapped to a policy that allows information to flow from d_f to d_t.

At-Input-Checking. To define dynamic incident-insensitive noninterference (DIINI) using at-input-checking, we must replace the relation $\cong_{\text{II}}$. Since the security of an input may only depend on the inputs that precede it, we define a new relation $\cong_{\text{DII}}$ that effectively forgets the inputs that follow the input currently in question. To achieve this, we define $\cong_{\text{DII}}$ to work from the end of input sequences to their front forgetting the inputs seen along the way.

Let $\iota_1{:}i_1 \cong_{\text{DII}}^{\leadsto,d} \iota_2{:}i_2$ iff $\mathsf{dom}(i_1) = \mathsf{dom}(i_2)$, $\mathsf{dom}(i_1) \leadsto^{\iota_1:i_1} d$ implies $i_1 = i_2$, and $\iota_1 \cong_{\text{DII}}^{\leadsto,d} \iota_2$. Also let $[] \cong_{\text{DII}}^{\leadsto,d} []$.

A system m obeys a DIINI policy $\leadsto$ using at-input-checking iff for all $d \in D$ and $\iota_1, \iota_2 \in I^*$,

$$\iota_1 \cong_{\text{DII}}^{\leadsto^{\iota_1},d} \iota_2 \text{ implies } \lfloor \mathsf{runs}(\iota_1) \rfloor_{A^d} \subseteq \lfloor \mathsf{runs}(\iota_2) \rfloor_{A^d}$$

At-Output-Checking. Since DIINI using at-output-checking does not need to forget any information, its definition is actually simpler. We provide the dynamic policy $\leadsto$ with the current input sequence ι_1 to obtain the static policy $\leadsto^{\iota_1}$ for use with $\cong_{\text{II}}$.

A system m obeys a DIINI policy $\leadsto$ using at-output-checking iff for all $d \in D$ and $\iota_1, \iota_2 \in I^*$,

$$\iota_1 \cong_{\text{II}}^{\leadsto^{\iota_1},d} \iota_2 \text{ implies } \lfloor \mathsf{runs}(\iota_1) \rfloor_{A^d} \subseteq \lfloor \mathsf{runs}(\iota_2) \rfloor_{A^d}$$

Discussion. Although the at-output-checking formulation allows us to formalize the email server policy, at-input-checking does have some advantages. Both use the input sequence on the left-hand side to produce a static policy. Given this sequence, the at-output-checking formulation selects one such static policy using the whole input sequence. The at-input-checking formulation, however, selects a new static policy with each recursive application. This allows the at-input-checking formulation more flexibility to treat each input of the sequence differently even if the inputs come from the same domain.

A related limitation of at-output-checking is its inability to capture *revocation*, the removal of a previously held access right. For example, revocation takes place if $d_\mathsf{f} \leadsto^{[i_1]} d_\mathsf{t}$ but $d_\mathsf{f} \not\leadsto^{[i_1,i_2]} d_\mathsf{t}$. Under the at-input-checking formulation, this would mean that d_f may access the input i_1 but not the input i_2. However, for a system to obey the policy under the at-output-checking formulation, the system must not produce output influenced by i_1 for d_f even if the output is produced before i_2 arrives. If the system did, it would lead to a violation of the policy once i_2 arrives. Thus, for a system to obey the above policy, it must actually also obey the policy where $d_\mathsf{f} \not\leadsto^{[i_1]} d_\mathsf{t}$ and $d_\mathsf{f} \not\leadsto^{[i_1,i_2]} d_\mathsf{t}$. For this reason, at-output-checking policies cannot express revocation.

We defined both of the above dynamic formulations to depend on input sequences and domains but not the states of the automaton, making them *input-based*. We view the states of an automaton to be implementation specific unlike the input-output behavior and domains of the system, which are at the specification level. Since policies should be at the specification level, we avoided referring to the states in the definition of a policy.

Henceforth, unless otherwise noted, all dynamic policies will be at-output-checking.

4.3 Dynamic Unwinding

Unlike policies that should be defined without reference to the states of an automaton, unwinding conditions must be. Thus, we need a version of dynamic policy that depends on the states instead of being input-based. Let a *generic state-based dynamic policy* $\rightsquigarrow$ be a function from a set of states to a relation on domains.

To give the unwinding conditions meaning with respect to an input-based policy, we must relate the input-based policy to a state-based policy. Let the state-based dynamic policy $\rightsquigarrow$ be a *safe approximation* of a input-based dynamic policy $\leadsto$ iff $d_{\mathsf{f}} \not\rightsquigarrow^{\iota} d_{\mathsf{t}}$, $\lfloor\alpha\rfloor_I = \iota$, and $q_0 \xrightarrow{\alpha} q$ implies $d_{\mathsf{f}} \not\rightsquigarrow^{q} d_{\mathsf{t}}$. We call $\rightsquigarrow$ *non-revoking* iff for all $\alpha \in A^*$, $q \xrightarrow{\alpha} q'$ and $d_{\mathsf{f}} \rightsquigarrow^{q} d_{\mathsf{t}}$ implies that $d_{\mathsf{f}} \rightsquigarrow^{q'} d_{\mathsf{t}}$.

Given a system automaton, let a *dynamic view partition* be a function from a pair of domains to an equivalence relation on states. That is, a view partition is in $D \times D \rightarrow 2^{Q \times Q}$. We will write $q_1 \overset{d_{\mathsf{t}}}{\underset{d_{\mathsf{f}}}{\sim}} q_2$ if for the pair of domains $\langle d_{\mathsf{t}}, d_{\mathsf{f}} \rangle$, the states q_1 and q_2 are within the relation. Intuitively, $q_1 \overset{d_{\mathsf{t}}}{\underset{d_{\mathsf{f}}}{\sim}} q_2$ means that the states q_1 and q_2 should look the same to d_{t} since they only differ by secret inputs from d_{f}.

Let a dynamic view partitioning $\cdot \overset{\cdot}{\underset{\cdot}{\sim}} \cdot$ for a program automaton m be called a *dynamic unwinding relation* for a state-based dynamic policy $\rightsquigarrow$ if $\sim$ satisfies the following *dynamic unwinding conditions*:

1. Local Respect: for all $d_{\mathsf{t}}, d_{\mathsf{f}} \in D$, $q, q'_1 \in Q$, and $i_1, i_2 \in I$ if $\mathsf{dom}(i_1) = \mathsf{dom}(i_2) = d_{\mathsf{f}}$, $q \xrightarrow[d_{\mathsf{t}}]{i_1} q'_1$, and $d_{\mathsf{f}} \not\rightsquigarrow^{q'_1} d_{\mathsf{t}}$, then there must exist $q'_2 \in Q$ such that $q \xrightarrow[d_{\mathsf{t}}]{i_2} q'_2$ and $q'_1 \overset{d_{\mathsf{t}}}{\underset{d_{\mathsf{f}}}{\sim}} q'_2$.

2. Step Consistency: for all $d_{\mathsf{t}}, d_{\mathsf{f}} \in D$, $q_1, q'_1, q_2 \in Q$, and $i \in I$, if $q_1 \overset{d_{\mathsf{t}}}{\underset{d_{\mathsf{f}}}{\sim}} q_2$, $q_1 \xrightarrow[d_{\mathsf{t}}]{i} q'_1$, and $d_{\mathsf{f}} \not\rightsquigarrow^{q'_1} d_{\mathsf{t}}$, then there must exist $q'_2 \in Q$ such that $q_2 \xrightarrow[d_{\mathsf{t}}]{i} q'_2$ and $q'_1 \overset{d_{\mathsf{t}}}{\underset{d_{\mathsf{f}}}{\sim}} q'_2$.

3. Output Consistency: for all $d_{\mathsf{t}}, d_{\mathsf{f}} \in D$, $q_1, q'_1, q_2 \in Q$, and $o \in O$ if $\mathsf{dom}(o) = d_{\mathsf{t}}$, $q_1 \overset{d_{\mathsf{t}}}{\underset{d_{\mathsf{f}}}{\sim}} q_2$, and $q_1 \xrightarrow[d_{\mathsf{t}}]{o} q'_1$, and $d_{\mathsf{f}} \not\rightsquigarrow^{q'_1} d_{\mathsf{t}}$, then there must exist $q'_2 \in Q$ such that $q_2 \xrightarrow[d_{\mathsf{t}}]{o} q'_2$ and $q'_1 \overset{d_{\mathsf{t}}}{\underset{d_{\mathsf{f}}}{\sim}} q'_2$.

As with static unwinding relations, the existence of a dynamic unwinding relation implies that the system obeys the policy:

Theorem 3. *For all automata m, if $\rightsquigarrow$ is a non-revoking safe approximation of the at-output-checking DIINI policy $\leadsto$ and there exists an unwinding relation for $\rightsquigarrow$ and m, then m obeys $\leadsto$.*

Appendix B provides the proof.

5 Automated Policy Extraction

Although using the dynamic unwinding conditions eases proving that a program obeys an DIINI policy, we really desire an automatic algorithm to check for obedience. Furthermore, as motived in the introduction, often one would like to know the most restrictive policy that a program obeys. Thus, we describe an approach for extracting from the source code of a program an approximation of the most restrictive policy obeyed by that program.

Our approach tracks the flow of information through the program in a manner similar to information-flow type systems [29, 25]. However, since our approach must work for legacy code designed without the analysis in mind, some of the limitations of these type systems render them unacceptable. For example, type systems will consider high-level any information stored in a variable that has ever stored high-level information even if the current information stored in the variable is low-level. Furthermore, type systems make no attempt to rule out infeasible paths.

Thus, we approach the problem with model checking. For each ordered pair of domains d_{f} and d_{t}, we will check for the property that the static incident-insensitive noninterference assertion $d_{\mathsf{f}} \not\rightsquigarrow d_{\mathsf{t}}$ is not violated by the program. The collection of all counterexamples to this property will form all the executions in which d_{t} gains access to information about d_{f}. From these, we construct a DIINI policy that the program obeys.

Our approach differs from standard model checking in that we need all of the counterexamples to the noninterference property, not just one. Furthermore, our approach differs in that the noninterference property is neither a safety nor liveness property and, thus, not expressible in any of the standard temporal logics used as property languages [23]. Like a safety property, noninterference requires that something does not happen: noninterference is not violated. However, unlike a safety property, to determine if noninterference is violated requires comparing two behaviors of the program. Thus, Terauchi and Aiken calls noninterference a *2-safety property* [33].

To address the first difference, we use an all-counterexample extension to standard model checking [15, 30]. To address the second difference, our approach constructs a model of the program that reifies this 2-safety property as a normal safety property. Before presenting this construction formally, we provide an example. In the example, and most of the rest of the section, we will only concern ourselves with extracting the dynamic conditions under which one given domain gains access to one other given domain. We discuss extending this approach to more than two domains in Section 5.4.

5.1 An Example

Consider the program from the email server example in Section 4.1. We would like to extract from this program the most restrictive DIINI policy that it obeys. For simplicity, we restrict our attention to only cases where the user (domain u) gains access to the emails (domain e). Thus, we will model check for the property that the static policy $\mathsf{e} \not\rightsquigarrow \mathsf{u}$ is obeyed.

The first step of our approach performs a property-reifying transformation to the program making the 2-safety property that $\mathsf{e} \not\rightsquigarrow \mathsf{u}$ is obeyed into a safety property. For each variable x, the transformation creates a shadow variable x' that tracks if x is independent of the value of all the inputs produced by e. The transformed program is

```
emails = load("mbox");
emails’ = false;
real_pw = load("password");
real_pw’ = true;
given_pw = read();
given_pw’ = true;
if(given_pw == real_pw)
  print(emails);
  print’(emails’ & given_pw’ & real_pw’);
else
  print("wrong");
  print’(given_pw’ & real_pw’);
```

where & is boolean AND. The variable `emails’` is the shadow variable for `emails`. It is set to `false` because `emails` depends on an input from e. `real_pw’`, the shadow variable of `real_pw`, is set to `true` since it is independent of e. Likewise with `given_pw’`.

`print’` is a special function that shadows calls to `print`. It allows us to reify that u has gained access to the inputs of e since whenever u does, `print’` is called with the value of `false`.

In the `then` branch of the `if` statement, `print’` is passed `emails’ & given_pw’ & real_pw’`. It is passed `emails` since the `print` statement it is shadowing, which precedes it, directly depends on the value of `emails`. It is passed `given_pw’` and `real_pw’` since by being in an `if` statement whose predicate depends on these values, the `print` statement indirectly depends them. These three shadow variables are conjoined since all three of them must be independent of e for the `print` statement to be independent of e.

The `print’` statement in the `else` branch only has `given_pw’ & real_pw’` since the `print` statement it is shadowing only depends (indirectly) on these values.

Checking for the safety property that “`print’` is never passed the value of `false`” yields a counterexample whenever `given_pw == real_pw`. This condition is only satisfied when the contents of `password` is equal to the user’s input. Thus, u only gains access to the input of e if the input of `password` equals the input of u. Therefore, the program obeys the policy $\rightsquigarrow$ where $\mathsf{e} \rightsquigarrow^{\iota} \mathsf{u}$ when the input sequence ι has the same second (`real_pw`) and third input (`given_pw`) and $\mathsf{e} \not\rightsquigarrow^{\iota} \mathsf{u}$ otherwise.

We can use the same method to extract the policy that governs access by u to the inputs of `password` (the domain p) by tracking how the value of the file `password` flows through the system instead of how the value of `mbox` does. One may see from the above transformed program, that both `print` statements depend on the value of `password`. Thus, the user always gets access to the input of p. Indeed, the user does learn if the password he has supplied as input is equal to value of `password` or not. In practice, this small bit of information is often negligible, a concept others have formalized [13, 18, 26], but we consider outside the scope of this paper.

Since our approach relies on the semantics of the analyzed language, we first present a simple language before formalizing our approach for that language.

5.2 The Language WhileIO

WhileIO is simple language with `while` loops, `if` statements, and operators for input and output. The syntax of WhileIO consists of statements S and expressions E:

$$\begin{aligned} \mathrm{S} &::= \mathrm{X}\texttt{:=}\mathrm{E} \mid \texttt{print(}\mathrm{E}\texttt{, }\mathrm{D}\texttt{)} \mid \texttt{read(}\mathrm{X}\texttt{, }\mathrm{D}\texttt{)} \mid \mathrm{S}\texttt{;}\mathrm{S} \\ &\quad \mid \texttt{if(}\mathrm{E}\texttt{)}\{\mathrm{S}\}\texttt{else}\{\mathrm{S}\} \mid \texttt{while(}\mathrm{E}\texttt{)}\{\mathrm{S}\} \\ \mathrm{E} &::= \mathrm{E}\texttt{+}\mathrm{E} \mid \mathrm{X} \mid \mathrm{D} \mid \mathrm{N} \end{aligned}$$

where X ranges over variable names, D over domains, and N over numbers. Statements always evaluate to void (written as •), and expressions always evaluate to a number. A program is just a single statement.

Table 1 gives the semantics of WhileIO. The judgment $\langle \Gamma, s\rangle \overset{a}{\hookrightarrow} \langle \Gamma', s'\rangle$ means that the statement s goes to s' while performing the action a and changing the store from Γ to Γ'. The store is a mapping from variables to numbers: $\Gamma : X \to N$. Let $\Gamma[x \mapsto v]$ be the store such that $\Gamma[x \mapsto v](y)$ is v if $x = y$ and is $\Gamma(y)$ if $x \neq y$. We extend stores to assign a number to expressions as follows: let $\Gamma(e_1\texttt{+}e_2)$ be $\Gamma(e_1) + \Gamma(e_2)$ and $\Gamma(n) = n$ for numbers n.

An *action* is an ordered triple: the first component is i if the action is an input and o if it is an output, the second component is the domain of the action, and the third component is the contents of the action. For example, $\langle \mathsf{i}, \mathsf{e}, \texttt{"Dear Bob..."}\rangle$ could be the input for the emails in the email server above example.

A program of WhileIO defines an automaton. The inputs I are those actions with i as the first component; the outputs O, those with o as the first component. dom projects the second component of an action. Each pair $\langle \Gamma, s\rangle$ defines a state. The transitions are provided by the judgment form $\hookrightarrow$: $\langle \Gamma, s\rangle \overset{a}{\to} \langle \Gamma', s'\rangle$ iff $\langle \Gamma, s\rangle \overset{a}{\hookrightarrow} \langle \Gamma', s'\rangle$. The initial state is $\langle \Gamma_0, s\rangle$ where s is the program and Γ_0 is the store that assigns zero to every variable. Given a program s let $\mathsf{autom}(s)$ represent this automaton.

A program s *obeys* a DIINI policy iff $\mathsf{autom}(s)$ obeys the DIINI policy as defined in Section 4.2.

$$\frac{}{\langle\Gamma, x{:=}e\rangle \overset{\tau}{\hookrightarrow} \langle\Gamma[x\mapsto\Gamma(e)], \bullet\rangle}$$

$$\frac{}{\langle\Gamma, \texttt{read(}x\texttt{, }d\texttt{)}\rangle \overset{\langle\mathsf{i},d,n\rangle}{\hookrightarrow} \langle\Gamma[x\mapsto n], \bullet\rangle}$$

$$\frac{n = \Gamma(e)}{\langle\Gamma, \texttt{print(}e\texttt{, }d\texttt{)}\rangle \overset{\langle\mathsf{o},d,n\rangle}{\hookrightarrow} \langle\Gamma, \bullet\rangle}$$

$$\frac{\langle\Gamma, s_1\rangle \overset{a}{\hookrightarrow} \langle\Gamma', s_1'\rangle}{\langle\Gamma, s_1\texttt{;}s_2\rangle \overset{a}{\hookrightarrow} \langle\Gamma', s_1'\texttt{;}s_2\rangle} \qquad \frac{\langle\Gamma, s_2\rangle \overset{a}{\hookrightarrow} \langle\Gamma', s_2'\rangle}{\langle\Gamma, \bullet\texttt{;}s_2\rangle \overset{a}{\hookrightarrow} \langle\Gamma', \bullet\texttt{;}s_2'\rangle}$$

$$\frac{}{\langle\Gamma, \bullet\texttt{;}\bullet\rangle \overset{\tau}{\hookrightarrow} \langle\Gamma, \bullet\rangle} \qquad \frac{\Gamma(e) = 0}{\langle\Gamma, \texttt{if(}e\texttt{)}\{s_1\}\{s_2\}\rangle \overset{\tau}{\hookrightarrow} \langle\Gamma, s_2\rangle}$$

$$\frac{\Gamma(e) \neq 0}{\langle\Gamma, \texttt{if(}e\texttt{)}\{s_1\}\{s_2\}\rangle \overset{\tau}{\hookrightarrow} \langle\Gamma, s_2\rangle}$$

$$\frac{}{\begin{array}{c}\langle\Gamma, \texttt{while(}e\texttt{)}\{s_1\}\rangle \overset{\tau}{\hookrightarrow} \\ \langle\Gamma, \texttt{if(}e\texttt{)}\{s_1\texttt{; while(}e\texttt{)}\{s_1\}\}\texttt{else}\{\bullet\}\rangle\end{array}}$$

Table 1. Semantics of WhileIO

5.3 Constructing the Model

Now we show how to convert a program of WhileIO to an automaton model. Rather than preform a source-to-source transformation as in the example of Section 5.1, we show how to reify the noninterference property directly in the model. Thus, the model contains some features that are unnecessary for simply modeling the behavior of the program. Strictly speaking, these extra features mean that the model is not a system automaton as defined in Section 2.

We present the model construction algorithm for finding the conditions under which the confidentiality assertion $d_{\mathsf{f}} \not\rightsquigarrow d_{\mathsf{t}}$ is violated for a fixed pair of domains d_{f} and d_{t} such that $d_{\mathsf{t}} \neq d_{\mathsf{f}}$. In the next section, we discuss dealing with more than two domains.

Let $\mathsf{model}(s) = \langle I, O, D, \mathsf{dom}, Q, q_0, \rightarrowtail\rangle$ be the model constructed for the program s. I, O, D, and dom come from the definition of action found in Section 5.2. The set of states Q is $(X \to N)\times L_s \times (X \to \{\mathsf{T},\mathsf{F}\})$ where L_s is a set of labels defined below. Each state $\langle\Gamma, \ell, \eta\rangle \in Q$ consists of a store Γ, a label ℓ, and an *independence predicate* η.

The set of labels L_s for atomic statements s holds just two labels: $\mathsf{pre}(s)$ and $\mathsf{post}(s)$, which represent the state right before executing s and the state right after. The set of labels for a compound statement s (an `if`, `while`, or `;` statement) results from adding $\mathsf{pre}(s)$ and $\mathsf{post}(s)$ to the disjoint union of the sets of labels for its sub-statements.

At a state $\langle\Gamma, \ell, \eta\rangle$, the independence predicate η, assigns to each variable x true if at that state the value of x is independent of the value of any input from the domain d_{f}. If x does depend on the value of an input from d_{f} or it is unclear if it does or not, then $\eta(x) = \mathsf{F}$. Let $\eta(e_1\texttt{+}e_2)$ be $\eta(e_1) \wedge \eta(e_2)$ and $\eta(n) = \mathsf{F}$ for $n \in N$.

The start state q_0 is $\langle\Gamma_0, \mathsf{pre}(s), \eta_{\mathsf{T}}\rangle$ where s is the program and η_{T} is the independence predicate that assigns true to all variables.

$\rightarrowtail$ is a transition relation from a state to a state under both an action and a boolean. $q \underset{\mathsf{T}}{\overset{a}{\rightarrowtail}} q'$ means that the model transitions from state q to state q' during the action a without providing any information about d_{f} to d_{t}. $q \underset{\mathsf{F}}{\overset{a}{\rightarrowtail}} q'$ means that the model transitions from q to q' during a while possibly providing information about d_{f} to d_{t}.

To define $\rightarrowtail$, we use a translation from a statement to a transition relation. We write $>s>$ for the translation of s. We write $q \underset{b}{\overset{a}{>s>}} q'$ if the state q transitions to q' under the action a and boolean b in the transition relation $>s>$. The value of $\rightarrowtail$ for the program s is $>s>$.

The translation $>s>$ is defined recursively on the structure of s. For each syntactic form that a statement can take, we provide all the cases in which $>s>$ holds: if $q \underset{b}{\overset{a}{>s>}} q'$ is not explicitly listed, then it does not hold (is not in the relation). (All variables are universally quantified.)

1. When s has the form $x{:=}e$:

$$\langle\Gamma, \mathsf{pre}(s), \eta\rangle \underset{\mathsf{T}}{\overset{\tau}{>s>}} \langle\Gamma[x\mapsto\Gamma(e)], \mathsf{post}(s), \eta[x\mapsto\eta(e)]\rangle$$

2. When s has the form `read(`x`, `d`)` with $d_{\mathsf{t}} \neq d \neq d_{\mathsf{f}}$:

$$\langle\Gamma, \mathsf{pre}(s), \eta\rangle \underset{\mathsf{T}}{\overset{\langle\mathsf{i},d,n\rangle}{>s>}} \langle\Gamma[x\mapsto n], \mathsf{post}(s), \eta[x\mapsto\mathsf{T}]\rangle$$

3. When s has the form `read(`x`, `d`)` with $d = d_{\mathsf{f}}$ or $d = d_{\mathsf{t}}$:

$$\langle\Gamma, \mathsf{pre}(s), \eta\rangle \underset{\mathsf{T}}{\overset{\langle\mathsf{i},d,n\rangle}{>s>}} \langle\Gamma[x\mapsto n], \mathsf{post}(s), \eta[x\mapsto\mathsf{F}]\rangle$$

4. When s has the form `print(`e`, `d`)` with $d \neq d_{\mathsf{t}}$:

$$\langle\Gamma, \mathsf{pre}(s), \eta\rangle \underset{\mathsf{T}}{\overset{\langle\mathsf{o},d,\Gamma(e)\rangle}{>s>}} \langle\Gamma, \mathsf{post}(s), \eta\rangle$$

5. When s has the form `print(`e`, `d_{t}`)`:

$$\langle\Gamma, \mathsf{pre}(s), \eta\rangle \underset{\eta(e)}{\overset{\langle\mathsf{o},d_{\mathsf{t}},\Gamma(e)\rangle}{>s>}} \langle\Gamma, \mathsf{post}(s), \eta\rangle$$

6. When s has the form s_1 `;` s_2:

$$\langle \Gamma, \mathsf{pre}(s), \eta\rangle >\overset{\tau}{\underset{\mathsf{T}}{s}}> \langle \Gamma, \mathsf{pre}(s_1), \eta\rangle$$

$$\langle \Gamma, \mathsf{post}(s_1), \eta\rangle >\overset{\tau}{\underset{\mathsf{T}}{s}}> \langle \Gamma, \mathsf{pre}(s_2), \eta\rangle$$

$$\langle \Gamma, \mathsf{post}(s_2), \eta\rangle >\overset{\tau}{\underset{\mathsf{T}}{s}}> \langle \Gamma, \mathsf{post}(s), \eta\rangle$$

$$q >\overset{a}{\underset{b}{s}}> q' \quad \text{if} \quad q >\overset{a}{\underset{b}{s_1}}> q' \text{ or } q >\overset{a}{\underset{b}{s_2}}> q'$$

7. When s has the form `if(`e`)` s_1 `else` s_2:

$$\langle \Gamma, \mathsf{pre}(s), \eta\rangle \ >\overset{\tau}{\underset{\eta(e)\vee w}{s}}> \ \langle \Gamma, \mathsf{pre}(s_j), \eta'\rangle$$

$$\langle \Gamma, \mathsf{post}(s_j), \eta\rangle >\overset{\tau}{\underset{\mathsf{T}}{s}}> \langle \Gamma, \mathsf{post}(s), \eta'\rangle$$

$$q >\overset{a}{\underset{b}{s}}> q' \quad \text{if} \quad q >\overset{a}{\underset{b}{s_j}}> q'$$

   where $j = 1$ if $\Gamma(e) \neq 0$ and $j = 2$ if $\Gamma(e) = 0$, and $\eta'(x) = \eta(x) \wedge (\eta(e) \vee x \notin \mathsf{def}(s_1) \cup \mathsf{def}(s_2))$ where $\mathsf{def}(s)$ is the set containing all variables defined (on the left-hand side of a `:=` statement or the variable in a `read` statement) in s, and w is false if s_1 or s_2 contain a `while` loop, a `read` statement, or a statement of the form `print(`e`,` d_{t}`)`.

8. When s has the form `while(`e`)`s_1 with $\Gamma(e) \neq 0$:

$$\langle \Gamma, \mathsf{pre}(s), \eta\rangle >\overset{\tau}{\underset{\eta(e)}{s}}> \langle \Gamma, \mathsf{pre}(s_1), \eta\rangle$$

$$\langle \Gamma, \mathsf{post}(s_1), \eta\rangle >\overset{\tau}{\underset{\eta(e)}{s}}> \langle \Gamma, \mathsf{pre}(s), \eta\rangle$$

$$q >\overset{a}{\underset{b}{s}}> q' \quad \text{if} \quad q >\overset{a}{\underset{b}{s_1}}> q'$$

9. When s has the form `while(`e`)`s_1 with $\Gamma(e) = 0$:

$$\langle \Gamma, \mathsf{pre}(s), \eta\rangle >\overset{\tau}{\underset{\eta(e)}{s}}> \langle \Gamma, \mathsf{post}(s), \eta\rangle$$

The transitions for `while` statements produce the boolean $\eta(e)$ despite producing no output since their termination or lack there of may affect the output seen by the user. `while` and `read` statements are treated specially in `if` statements for the same reason.

5.4 Using the Model

Once $\mathsf{model}(s)$ has been constructed, our approach uses it to create an approximation of the most restrictive DIINI policy that the program s obeys. First, our approach finds all reachable transitions of the form $q_1 \overset{o}{\underset{\mathsf{F}}{\rightarrowtail}} q_2$. These transitions indicate that the output o might provide the low-level user d_{t} with information about an input of d_{f}. Second, for each such transition, our approach finds each input sequence ι that leads to this transition. Third, for each such ι, $d_{\mathsf{f}} \rightsquigarrow^{\iota'} d_{\mathsf{t}}$ is added to the policy for every ι' that has ι as a prefix. After this process is complete, the resulting policy is returned with $d_{\mathsf{f}} \not\rightsquigarrow^{\iota} d_{\mathsf{t}}$ for all ι such that $d_{\mathsf{f}} \rightsquigarrow^{\iota} d_{\mathsf{t}}$ was not added to the policy. Let $\mathsf{policy}(\mathsf{model}(s))$ represent this policy. (We define $\mathsf{policy}(\mathsf{model}(s))$ more formally in Appendix C.)

Correctness of our approach may be stated as follows:

Theorem 4. *For every program s of WhileIO,* $\mathsf{autom}(s)$ *obeys the DIINI policy* $\mathsf{policy}(\mathsf{model}(s))$.

We prove this theorem in Appendix C.

To convert our approach to an actual algorithm, we must select a method for finding all the input sequences that lead to a transition of the form $q_1 \overset{o}{\underset{\mathsf{F}}{\rightarrowtail}} q_2$. Finding these sequences is equivalent to finding all the counterexamples to the property no such transition is reachable. While standard model checkers will stop after finding one counterexample to this property, algorithms exist for producing all the counterexamples. Jha and Wing [15] give an algorithm using a symbolic representation of the state space and a modified version of a standard iterative fixed-point algorithm [24]. Sheyner [30] gives another algorithm uses an explicit state representation.

For handling more than the two domains d_{f} and d_{t}, a tool can repeat the above approach for each ordered pair of domains. The transitive closure of the union of these policies provides a policy that the program obeys.

6 Related Work and Discussion

Assumptions. All the systems discussed in this paper have been *interactive*, that is, they receive input and produce output throughout their execution. A *batch-job* system only allows users to determine the contents of its memory at the beginning of its execution and to observe any changes at the end of its execution. Much of the work on type systems for enforcing confidentiality policies have been for batch-job systems [29].

We have assumed that the user may observe not only the outputs from a system but also the system consuming his inputs. Many systems actually buffer user inputs making it unclear when an input actually affects the state of the system. McCullough discusses some issues that arise from modeling systems that use buffers [21].

A system is *input-enabled* if it will always accept any input offered by a user. While most confidentiality requirements have been defined for input-enabled systems, we have not made this assumption.

We have modeled systems as asynchronous automata, which can provide output to one user without sending output to all the users. Most authors use synchronous automata, which must produce outputs to all users at regular intervals. (See [9] for a detailed comparison.) We believe our unwinding conditions to be the first for asynchronous automata.

We have assumed that the users cannot observe the termination of a system. This assumption makes our incident-insensitive noninterference requirement *termination-insensitive*. Others have considered program analysis for *termination-sensitive* confidentiality requirements [33].

Other Requirements. Incident-insensitive noninterference requires that if $\iota_1 \cong_{\mathrm{II}}^{\rightsquigarrow,d} \iota_2$, then any behavior of the system under ι_1 must also appear *possible* under ι_2 to the domain d. Thus, this formulation is called *possibilistic*. In some contexts, a system is unacceptable if the observations of d is likely to occur under ι_1 and unlikely under ι_2. Such concerns has led Gray and Syverson to define *probabilistic* noninterference, which requires the observation to be equiprobable under both ι_1 and ι_2 [14].

Nondeducibility on strategies requires that no matter how a high-level user interacts with a system, a low-level user will still not be able to learn anything about the high-level user's inputs [36]. The original formulation is incident-sensitive. O'Neill et al. created an incident-insensitive version to characterize formally the properties that information-flow type systems enforce for interactive systems [25]. We suspect that few if any modifications would be required to use our approach for extracting nondeducibility policies.

Even if two automata obey the same noninterference policy, their composition might not. McCullough has proposed requirements that ensure that the composition of two obeying automata will also obey a policy [20, 21]. Also, removing nondeterminism from an automaton that obeys a noninterference policy might result in one that does not. Others have studied conditions under which such refinement will not destroy the security of an automaton [16, 19, 1].

We have required that each policy $\rightsquigarrow$ be transitive. Intransitive policies model channel control, the requirement that information passes through a downgrading domain before reaching a domain of a lower level. Rusby defined the most commonly used formulation of intransitive noninterference [28]. However, Roscoe and Goldsmith [27] offer a competing formulation using CSP [11].

Whereas confidentiality requires that protected data does not become known to untrusted users, *integrity* requires that protected data does not become tainted or corrupted by untrusted users. By reversing the roles of the high- and low-level users of a system, integrity becomes confidentiality. Thus, our confidentiality requirements also define an integrity requirements.

Dynamic Unwinding. Leslie has also provided a set of dynamic unwinding conditions [17]. Rather than asynchronous, nondeterministic automata, she defines her unwinding conditions for synchronous, deterministic automata. Her conditions ensure that an intransitive incident-sensitive noninterference policy is obeyed while ours is for transitive incident-insensitive noninterference. Furthermore, hers is for at-input-checking dynamic policies rather than at-output-checking dynamic policies.

Related Tools. Although we are the first to propose using all-counterexamples model checking for policy extraction, others have used standard model checking for verifying that a given policy is obeyed. They observed that by composing a program with itself, one can obtain the two behaviors necessary to check the 2-safety property of noninterference [3, 2]. Later work improved this approach by using type theory to produce more efficient models [33, 35].

Program dependence graphs represent how inputs from different users interact [4, 5]. Thus, they reveal if a system obeys a noninterference policy [32]. Hammer et al. have extended this approach also to produce "witnesses" (counterexamples) in cases where the policy fails to hold [10]. These counterexamples could form the basis of an algorithm for dynamic policy extraction.

Just as a confidentiality policy may become buried within the code of a large program, the operating procedures of a business may also become hidden within large applications. Thus, others have created tools to extract these business rules from source code [12, 31]. These tools use program slicing [34] instead of model checking.

Once a policy is extracted from a program, the maintainer might want to update the program to accept the policy as a configuration parameter. This requires refactoring the code to use a centralized policy enforcement mechanism. Ganapathy et al. have developed tools to retrofit legacy code for this purpose [6].

7 Summary

Firstly, we have clarified the difference between incident-sensitive and incident-insensitive noninterference, two requirements often conflated as simply "noninterference". Secondly, we have introduced at-output-checking dynamic policies to express policies that at-input-checking dynamic policies cannot. Thirdly, we have presented an approach based on all-counterexamples model checking for the automated extraction of at-output-checking dynamic incident-insensitive noninterference policies from program source code.

References

[1] R. Alur, P. Cerny, and S. Zdancewic. Preserving secrecy under refinement. In *Proceeding of 33rd International Colloquium on Automata, Languages and Programming*, 1996.

[2] G. Barthe, P. R. D'Argenio, and T. Rezk. Secure information flow by self-composition. In *CSFW '04: Proceedings of the 17th IEEE Computer Security Foundations Workshop (CSFW'04)*, page 100, Washington, DC, USA, 2004. IEEE Computer Society.

[3] Á. Darvas, R. H ahnle, and D. Sands. A theorem proving approach to analysis of secure information. In D. Hutter and M. Ullmann, editors, *Security in Pervasive Computing*, Lecture Notes in Computer Science, pages 193–209. Springer, Apr. 2005. Appeared at the Workshop on Issues in The Theory of Security in 2003 but was not formally published at that time.

[4] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. In *Proceedings of the 6th Colloquium on International Symposium on Programming*, pages 125–132, London, UK, 1984. Springer-Verlag.

[5] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst.*, 9(3):319–349, 1987.

[6] V. Ganapathy, T. Jaeger, and S. Jha. Retrofitting legacy code for authorization policy enforcement. In *Proceedings of the 2006 IEEE Symposium on Security and Privacy*, pages 214–229, Los Alamitos, CA, USA, May 2006. IEEE Computer Society.

[7] J. A. Goguen and J. Meseguer. Security policies and security models. In *IEEE Symposium on Security and Privacy*, page 11. IEEE, 1982.

[8] J. A. Goguen and J. Meseguer. Unwinding and inference control. In *Proc. of IEEE Symp. on Security and Privacy*, pages 75–86, Los Alamitos, CA, USA, 1984. IEEE Computer Society.

[9] J. Halpern and K. O'Neill. Secrecy in multiagent systems. In *Proc. 15th IEEE Computer Security Foundations Workshop*, pages 32–46, 2002.

[10] C. Hammer, J. Krinke, and G. Snelting. Information flow control for java based on path conditions in dependence graphs. In *Proceedings of IEEE International Symposium on Secure Software Engineering*, pages 87–96, Arlington, VA, March 2006.

[11] C. A. R. Hoare. *Communicating sequential processes*. Prentice-Hall, 1985.

[12] H. Huang, W. T. Tsai, S. Bhattacharya, X. P. Chen, Y. Wang, and J. Sun. Business rule extraction from legacy code. In *COMPSAC '96 - 20th Computer Software and Applications Conference*, pages 162–167, Los Alamitos, CA, USA, 1996. IEEE Computer Society.

[13] I. James W. Gray. Toward a mathematical foundation for information flow security," research in security and privacy. In *IEEE Computer Society Symposium on Research in Security and Privacy*, pages 21–34, Oakland, CA, May 1991. IEEE.

[14] I. James W. Gray and P. F. Syverson. A logical approach to multilevel security of probabilistic systems. *Distrib. Comput.*, 11(2):73–90, 1998.

[15] S. Jha and J. M. Wing. Survivability analysis of networked systems. In *ICSE '01: Proceedings of the 23rd International Conference on Software Engineering*, pages 307–317, Washington, DC, USA, 2001. IEEE Computer Society.

[16] J. J urjens. Secrecy-preserving refinement. In *FME '01: Proceedings of the International Symposium of Formal Methods Europe on Formal Methods for Increasing Software Productivity*, pages 135–152, London, UK, 2001. Springer-Verlag.

[17] R. Leslie. Dynamic intransitive noninterference. In *Proc. IEEE International Symposium on Secure Software Engineering*, Washington, D.C., 2006.

[18] G. Lowe. Quantifying information flow. In *CSFW '02: Proceedings of the 15th IEEE Computer Security Foundations Workshop (CSFW'02)*, page 18, Washington, DC, USA, 2002. IEEE Computer Society.

[19] H. Mantel. Preserving information flow properties under refinement. In *SP '01: Proceedings of the 2001 IEEE Symposium on Security and Privacy*, page 78, Washington, DC, USA, 2001. IEEE Computer Society.

[20] D. McCullough. Specifications for multi-level security and a hook-up property. In *IEEE Symposium on Security and Privacy*, pages 161–166, 1987.

[21] D. McCullough. Noninterference and the composability of security properties. *sp*, 00:177, 1988.

[22] D. McCullough. A hookup theorem for multilevel security. *IEEE Transactions on Software Engineering*, 16(6):563–568, 1990.

[23] J. McLean. A general theory of composition for trace sets closed under selective interleaving functions. In *SP '94: Proceedings of the 1994 IEEE Symposium on Security and Privacy*, page 79, Washington, DC, USA, 1994. IEEE Computer Society.

[24] K. L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, Norwell, MA, USA, 1993.

[25] K. R. O'Neill, M. R. Clarkson, and S. Chong. Information-flow security for interactive programs. In *CSFW '06: Proceedings of the 19th IEEE Workshop on Computer Security Foundations*, pages 190–201, Washington, DC, USA, 2006. IEEE Computer Society.

[26] A. D. Pierro, C. Hankin, and H. Wiklicky. Approximate non-interference. In *CSFW '02: Proceedings of the 15th IEEE Computer Security Foundations Workshop (CSFW'02)*, page 3, Washington, DC, USA, 2002. IEEE Computer Society.

[27] A. W. Roscoe and M. H. Goldsmith. What is intransitive noninterference? In *CSFW '99: Proceedings of the 1999 IEEE Computer Security Foundations Workshop*, page 228, Washington, DC, USA, 1999. IEEE Computer Society.

[28] J. Rushby. Noninterference, transitivity, and channel-control security policies. Technical report, SRI, dec 1992.

[29] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, Jan. 2003.

[30] O. Sheyner. *Scenario Graphs and Attack Graphs*. PhD thesis, Carnegie Mellon University, 2004.

[31] H. M. Sneed. Extracting business logic from existing cobol programs as a basis for redevelopment. In *Proceedings. 9th International Workshop on Program Comprehension*, pages

167–175, Los Alamitos, CA, USA, 2001. IEEE Computer Society.

[32] G. Snelting, T. Robschink, and J. Krinke. Efficient path conditions in dependence graphs for software safety analysis. *ACM Trans. Softw. Eng. Methodol.*, 15(4):410–457, 2006.

[33] T. Terauchi and A. Aiken. Secure information flow as a safety problem. In *The Proceedings of the 12th International Static Analysis Symposium*, pages 352–367. Springer Berlin / Heidelberg, September 2005.

[34] F. Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3(3):121–189, 1995.

[35] H. Unno, N. Kobayashi, and A. Yonezawa. Combining type-based analysis and model checking for finding counterexamples against non-interference. In *PLAS '06: Proceedings of the 2006 workshop on Programming languages and analysis for security*, pages 17–26, New York, NY, USA, 2006. ACM Press.

[36] J. T. Wittbold and D. M. Johnson. Information flow in nondeterministic systems. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 144–161, Los Alamitos, CA, USA, 1990. IEEE Computer Society.

A Proofs about Static Noninterference

A.1 Formalization of Example

We formalize the example found in Section 3.1. We model this program as the system exs where

- $I_{\mathsf{exs}} = \{\mathsf{T}, \mathsf{F}\}$,
- $O_{\mathsf{exs}} = \{\mathsf{x}, \mathsf{y}, \mathsf{z}\}$,
- $D_{\mathsf{exs}} = \{\mathsf{H}, \mathsf{L}\}$,
- $\mathsf{dom}_{\mathsf{exs}}(\mathsf{T}) = \mathsf{dom}_{\mathsf{exs}}(\mathsf{F}) = \mathsf{H}$
  $\mathsf{dom}_{\mathsf{exs}}(\mathsf{x}) = \mathsf{dom}_{\mathsf{exs}}(\mathsf{y}) = \mathsf{dom}_{\mathsf{exs}}(\mathsf{z}) = \mathsf{L}$,
- $Q_{\mathsf{exs}} = \{q_0, q_1, q_2\}$,
- and the transition $\underset{\mathsf{exs}}{\rightarrow}$ is such that $q_0 \overset{\mathsf{T}}{\rightarrow} q_1$, $q_0 \overset{\mathsf{F}}{\rightarrow} q_1$, and $q_1 \overset{\mathsf{x}}{\rightarrow} q_2$

where q_0 is the start state. The system only accepts input from the domain H and only produces output for the domain L. It only has two behaviors $[\mathsf{T}, \mathsf{x}]$ and $[\mathsf{F}, \mathsf{x}]$. Each consumes an input from the domain H and then produces the output x for the domain L.

The desire of the system designer is to protect the confidentiality of the domain H from the domain L. So let $\rightsquigarrow_{\mathsf{exp}}$ be a policy such that $\mathsf{H} \not\rightsquigarrow_{\mathsf{exp}} \mathsf{L}$ and $\mathsf{L} \rightsquigarrow_{\mathsf{exp}} \mathsf{H}$. This policy makes H a high-level domain and L a low-level domain.

Lemma 1. *The system* exs *fails to obey* $\rightsquigarrow_{\mathsf{exp}}$ *as incident-sensitive noninterference policy.*

Proof. Both $[]$ and $[\mathsf{T}]$ are in I^* and $[] \cong_{\mathrm{IS}}^{\mathsf{exp},\mathsf{L}} [\mathsf{T}]$ since $\mathsf{dom}_{\mathsf{exs}}(\mathsf{T}) = \mathsf{H}$ and $\mathsf{H} \not\rightsquigarrow_{\mathsf{exp}} \mathsf{L}$. Thus, it should be the case that $\lfloor \mathsf{runs}_{\mathsf{exs}}([]) \rfloor_{A^{\mathsf{L}}} = \lfloor \mathsf{runs}_{\mathsf{exs}}([\mathsf{T}]) \rfloor_{A^{\mathsf{L}}}$. However, $\lfloor \mathsf{runs}_{\mathsf{exs}}([]) \rfloor_{A^{\mathsf{L}}} = \lfloor \{\} \rfloor_{A^{\mathsf{L}}} = \{\}$ whereas $\lfloor \mathsf{runs}_{\mathsf{exs}}([\mathsf{T}]) \rfloor_{A^{\mathsf{L}}} = \lfloor \{[\mathsf{T}, \mathsf{x}]\} \rfloor_{A^{\mathsf{L}}} = \{[\mathsf{x}]\}$. □

Lemma 2. *The system* exs *does obey* $\rightsquigarrow_{\mathsf{exp}}$ *as incident-insensitive noninterference policy.*

Proof. For H, $\iota_1 \cong_{\mathrm{II}}^{\rightsquigarrow_{\mathsf{exp}},\mathsf{H}} \iota_2$ only if $\iota_1 = \iota_2$ for all $\iota_1, \iota_2 \in I^*$ since $\mathsf{dom}_{\mathsf{exs}}(i) \rightsquigarrow d$ for all $d \in D_{\mathsf{exs}}$ and $i \in I_{\mathsf{exs}}$. Thus, clearly, $\iota_1 \cong_{\mathrm{II}}^{\rightsquigarrow_{\mathsf{exp}},\mathsf{H}} \iota_2$ implies that $\lfloor \mathsf{runs}_{\mathsf{exs}}(\iota_1) \rfloor_{A^{\mathsf{H}}_{\mathsf{exs}}} = \lfloor \mathsf{runs}_{\mathsf{exs}}(\iota_2) \rfloor_{A^{\mathsf{H}}_{\mathsf{exs}}}$.

For L, consider the following two cases:

1. $\iota_1 \in \{[\mathsf{T}], [\mathsf{F}]\}$. Then $\iota_1 \cong_{\mathrm{II}}^{\rightsquigarrow_{\mathsf{exp}},\mathsf{L}} \iota_2$ if and only if $\iota_2 \in \{[\mathsf{T}], [\mathsf{F}]\}$ since no other input sequences of length one exists and $\mathsf{dom}_{\mathsf{exs}}(\mathsf{T}) = \mathsf{dom}_{\mathsf{exs}}(\mathsf{F}) = \mathsf{H}$ and $\mathsf{H} \not\rightsquigarrow_{\mathsf{exp}} \mathsf{L}$. Note

   $$\begin{aligned}\lfloor \mathsf{runs}_{\mathsf{exs}}([\mathsf{T}]) \rfloor_{A^{\mathsf{L}}_{\mathsf{exs}}} &= \lfloor \{[\mathsf{T}, \mathsf{x}]\} \rfloor_{A^{\mathsf{L}}_{\mathsf{exs}}} \\ &= \{\lfloor [\mathsf{T}, \mathsf{x}] \rfloor_{A^{\mathsf{L}}_{\mathsf{exs}}}\} = \{[\mathsf{x}]\} = \{\lfloor [\mathsf{F}, \mathsf{x}] \rfloor_{A^{\mathsf{L}}_{\mathsf{exs}}}\} \\ &= \lfloor \{[\mathsf{F}, \mathsf{x}]\} \rfloor_{A^{\mathsf{L}}_{\mathsf{exs}}} = \lfloor \mathsf{runs}_{\mathsf{exs}}([\mathsf{F}]) \rfloor_{A^{\mathsf{L}}_{\mathsf{exs}}}\end{aligned}$$

   since T and F are not in $A^{\mathsf{L}}_{\mathsf{exs}}$. Thus, $\lfloor \mathsf{runs}_{\mathsf{exs}}(\iota_1) \rfloor_{A^{\mathsf{L}}_{\mathsf{exs}}} = \lfloor \mathsf{runs}_{\mathsf{exs}}(\iota_2) \rfloor_{A^{\mathsf{L}}_{\mathsf{exs}}}$ if $\iota_1 \cong_{\mathrm{II}}^{\rightsquigarrow_{\mathsf{exp}},\mathsf{L}} \iota_2$.

2. $\iota_1 \notin \{[\mathsf{T}], [\mathsf{F}]\}$. Then, as explained above, $\iota_2 \notin \{[\mathsf{T}], [\mathsf{F}]\}$ since $\iota_1 \cong_{\mathrm{II}}^{\rightsquigarrow_{\mathsf{exp}},\mathsf{L}} \iota_2$. Thus,

   $$\lfloor \mathsf{runs}_{\mathsf{exs}}(\iota_1) \rfloor_{A^{\mathsf{L}}_{\mathsf{exs}}} = \lfloor \{\} \rfloor_{A^{\mathsf{L}}_{\mathsf{exs}}} = \lfloor \mathsf{runs}_{\mathsf{exs}}(\iota_2) \rfloor_{A^{\mathsf{L}}_{\mathsf{exs}}}$$

   since no behavior of exs includes neither the input sequence $[\mathsf{T}]$ nor the input sequence $[\mathsf{F}]$.

□

A.2 Proof of Theorem 1

Lemma 3. *For a system m, for all $d \in D$ and $\alpha_1, \alpha_2 \in I^*$,*

$$\alpha_1 \cong_{\mathrm{II}}^{\rightsquigarrow,d} \alpha_2 \textit{ implies } \alpha_1 \cong_{\mathrm{IS}}^{\rightsquigarrow,d} \alpha_2$$

Proof. Proof by induction over the length of α_1. Note that if $\alpha_1 \cong_{\mathrm{II}}^{\rightsquigarrow,d} \alpha_2$, then $|\alpha_1| = |\alpha_2|$.

Base Case: $|\alpha_1| = 0$ and $\alpha_1 = []$. Then α_2 must be $[]$. Thus, $\alpha_1 \cong_{\mathrm{IS}}^{\rightsquigarrow,d} \alpha_2$ since $[] \cong_{\mathrm{IS}}^{\rightsquigarrow,d} []$.

Inductive Case: $|\alpha_1| = n > 0$. Here we may assume that $\alpha_1 = a_1{:}\alpha_1'$ for some $a_1 \in A$ and $\alpha_1' \in A^*$, $\alpha_2 = a_2{:}\alpha_2'$ for some $a_2 \in A$ and $\alpha_2' \in A^*$, and that $\alpha_1' \cong_{\mathrm{II}}^{\rightsquigarrow,d} \alpha_2'$ implies that $\alpha_1' \cong_{\mathrm{IS}}^{\rightsquigarrow,d} \alpha_2'$. We must show that $a_1{:}\alpha_1' \cong_{\mathrm{II}}^{\rightsquigarrow,d} a_2{:}\alpha_2'$ implies that $a_1{:}\alpha_1' \cong_{\mathrm{IS}}^{\rightsquigarrow,d} a_2{:}\alpha_2'$

Assume $\alpha_1 \cong_{\mathrm{II}}^{\rightsquigarrow,d} \alpha_2$. Then $\mathsf{dom}(a_1) = \mathsf{dom}(a_2)$, $\mathsf{dom}(a_1) \rightsquigarrow d$ implies that $a_1 = a_2$, and $\alpha_1' \cong_{\mathrm{II}}^{\rightsquigarrow,d} \alpha_2'$. Thus, $\alpha_1' \cong_{\mathrm{IS}}^{\rightsquigarrow,d} \alpha_2'$. Consider the following two cases

1. $\mathsf{dom}(a_1) \not\rightsquigarrow d$. In this case, $a_2 = a_1$. Since $\alpha'_1 \cong_{\text{IS}}^{\rightsquigarrow,d} \alpha'_2$, $a_1{:}\alpha'_1 \cong_{\text{IS}}^{\rightsquigarrow,d} a_2{:}\alpha'_2$.

2. $\mathsf{dom}(a_1) \not\rightsquigarrow d$. In this case, $\mathsf{dom}(a_2) \not\rightsquigarrow d$ also since $\mathsf{dom}(a_1) = \mathsf{dom}(a_2)$. Thus, $\alpha'_1 \cong_{\text{IS}}^{\rightsquigarrow,d} \alpha'_2$ implies that $a_1{:}\alpha'_1 \cong_{\text{IS}}^{\rightsquigarrow,d} \alpha'_2$, which implies that $\alpha'_1 \cong_{\text{IS}}^{\rightsquigarrow,d} \alpha'_2$ implies that $a_1{:}\alpha'_1 \cong_{\text{IS}}^{\rightsquigarrow,d} a_2{:}\alpha'_2$.

Thus, either way, $a_1{:}\alpha'_1 \cong_{\text{IS}}^{\rightsquigarrow,d} a_2{:}\alpha'_2$. □

Now the proof of Theorem 1.

Proof. Assume that a system m obeys $\rightsquigarrow$ as a noninterference policy. This implies that if $\alpha_1 \cong_{\text{IS}}^{\rightsquigarrow,d} \alpha_2$, then $\lfloor\mathsf{runs}(\alpha_1)\rfloor_{A_d} = \lfloor\mathsf{runs}(\alpha_2)\rfloor_{A_d}$. By Lemma 3, if $\alpha_1 \cong_{\text{II}}^{\rightsquigarrow,d} \alpha_2$, then $\alpha_1 \cong_{\text{IS}}^{\rightsquigarrow,d} \alpha_2$. Thus, if $\alpha_1 \cong_{\text{II}}^{\rightsquigarrow,d} \alpha_2$, then $\lfloor\mathsf{runs}(\alpha_1)\rfloor_{A_d} = \lfloor\mathsf{runs}(\alpha_2)\rfloor_{A_d}$. This means that m obeys $\rightsquigarrow$ as an incident-insensitive noninterference policy.

Lemmas 1 and 2 show that the converse is not true. □

A.3 Proof of Theorem 2

In each of the following lemmas let $\cdot \overset{\cdot}{\sim} \cdot$ be an unwinding relation for the automaton $m = \langle I, O, D, \mathsf{dom}, Q, q_0, \rightarrow\rangle$ and policy $\rightsquigarrow$.

Lemma 4 (Step Respect)**.** *For all $d \in D$, $i_1, i_2 \in I$, and $q_1, q'_1, q_2 \in Q$, if $\mathsf{dom}(i_1) = \mathsf{dom}(i_2)$, $\mathsf{dom}(i_1) \not\rightsquigarrow d$, $q_1 \overset{d}{\sim} q_2$, and $q_1 \xrightarrow[d]{i_1} q'_1$, then there must exist $q'_2 \in Q$ such that $q_2 \xrightarrow[d]{i_2} q'_2$ and $q'_1 \overset{d}{\sim} q'_2$.*

Proof. By Step Consistency, there must exist a q''_2 such that $q_2 \xrightarrow[d]{i_1} q''_2$ and $q'_1 \overset{d}{\sim} q''_2$. By Local Respect, there must exist a q'_2 such that $q_2 \xrightarrow[d]{i_1} q'_2$ and $q''_2 \overset{d}{\sim} q'_2$. By the transitivity of $\cdot \overset{d}{\sim} \cdot$, $q'_1 \overset{d}{\sim} q'_2$. □

Each of the next five lemmas proves almost the same statement for a more complicated set of behaviors than the last.

Lemma 5. *For all $d \in D$, $o \in O$, $\sigma_1 \in O^*$, and $q_1, q'_1, q_2 \in Q$, if $\mathsf{dom}(o) = d$, $\lfloor\sigma_1\rfloor_{A^d} = []$, $q_1 \overset{d}{\sim} q_2$, and $q_1 \xrightarrow{\sigma_1:o} q'_1$, then there must exist $\sigma_2 \in O^*$ and $q'_2 \in Q$ such that $q_2 \xrightarrow{\sigma_2:o} q'_2$, $q'_1 \overset{d}{\sim} q'_2$, and $\lfloor\sigma_2\rfloor_{A^d} = []$.*

Proof. Since $\lfloor\sigma_1\rfloor_{A^d} = []$, $q_1 \xrightarrow{\sigma_1:o} q'_1$ implies that $q_1 \xrightarrow[d]{o} q'_1$. Thus, by Output Consistency, there must exist q'_2 such that $q_2 \xrightarrow[d]{o} q'_2$ and $q'_1 \overset{d}{\sim} q'_2$. $q_2 \xrightarrow[d]{o} q'_2$ implies that there exists σ_2 such that $q_2 \xrightarrow{\sigma_2:o} q'_2$ and $\lfloor\sigma_2\rfloor_{A^d} = []$. □

Lemma 6. *For all $d \in D$, $i_1, i_2 \in I$, $\sigma_1 \in O^*$, and $q_1, q'_1, q_2 \in Q$, if $\lfloor\sigma_1\rfloor_{A^d} = []$, $q_1 \overset{d}{\sim} q_2$, $q_1 \xrightarrow{\sigma_1:i_1} q'_1$, $\mathsf{dom}(i_1) = \mathsf{dom}(i_2)$, and $\mathsf{dom}(i_1) \rightsquigarrow d$ implies $i_1 = i_2$, then there must exist $\sigma_2 \in O^*$ and $q'_2 \in Q$ such that $q_2 \xrightarrow{\sigma_2:i_2} q'_2$, $q'_1 \overset{d}{\sim} q'_2$, and $\lfloor\sigma_2\rfloor_{A^d} = []$.*

Proof. Since $\lfloor\sigma_1\rfloor_{A^d} = []$, $q_1 \xrightarrow{\sigma_1:i_1} q'_1$ implies that $q_1 \xrightarrow[d]{i_1} q'_1$. Consider the following two cases:

- $\mathsf{dom}(i_1) \rightsquigarrow d$. In this case $i_1 = i_2$. Thus, by Step Consistency, there must exist q'_2 such that $q_2 \xrightarrow[d]{i_1} q'_2$ and $q'_1 \overset{d}{\sim} q'_2$.
- $\mathsf{dom}(i_1) \not\rightsquigarrow d$. In this case, Step Respect (Lemma 4) implies the same thing.

In either case, $q_2 \xrightarrow[d]{i} q'_2$ implies that there exists σ_2 such that $q_2 \xrightarrow{\sigma_2:i} q'_2$ and $\lfloor\sigma_2\rfloor_{A^d} = []$. □

Lemma 7. *For all $d \in D$, $o \in O$, $\sigma_1, \sigma_d \in O^*$, and $q_1, q'_1, q_2 \in Q$, if $\mathsf{dom}(o) = d$, $q_1 \overset{d}{\sim} q_2$, $q_1 \xrightarrow{\sigma_1:o} q'_1$, and $\sigma_d = \lfloor\sigma_1\rfloor_{A^d}$, then there must exist $\sigma_2 \in O^*$ and $q'_2 \in Q$ such that $q_2 \xrightarrow{\sigma_2:o} q'_2$, $q'_1 \overset{d}{\sim} q'_2$, and $\lfloor\sigma_2\rfloor_{A^d} = \sigma_d$.*

Proof. Proof by induction over the structure of σ_d.

Case: $\sigma_d = []$. The result follows directly from Lemma 5.

Case: $\sigma_d = o'{:}\sigma''_d$. In this case, σ_1 must have the form $\sigma'_1{:}o'{:}\sigma''_1$ where $\lfloor\sigma'_1\rfloor_{A^d} = []$ and $\lfloor\sigma''_1\rfloor_{A^d} = \sigma''_d$. Since $q_1 \xrightarrow{\sigma_1:o} q'_1$, there must exist q''_2 such that $q_1 \xrightarrow{\sigma'_1:o'} q''_1 \xrightarrow{\sigma''_1:o} q'_1$.

By Lemma 5, there must exist $\sigma'_2 \in O^*$ and $q''_2 \in Q$ such that $q_2 \xrightarrow{\sigma'_2:o'} q''_2$, $q''_1 \overset{d}{\sim} q''_2$, and $\lfloor\sigma'_2\rfloor_{A^d} = []$. By the inductive hypothesis, there must exist $\sigma''_2 \in O^*$ and $q'_2 \in Q$ such that $q''_2 \xrightarrow{\sigma''_2:o} q'_2$, $q'_1 \overset{d}{\sim} q'_2$, and $\lfloor\sigma''_2\rfloor_{A^d} = \sigma''_d$.

Let $\sigma_2 = \sigma'_2{:}o'{:}\sigma''_2$. $q_2 \xrightarrow{\sigma'_2:o'} q''_2$ and $q''_2 \xrightarrow{\sigma''_2:o} q'_2$ implies $q_2 \xrightarrow{\sigma_2:o} q'_2$. $\lfloor\sigma'_2\rfloor_{A^d} = []$ and $\lfloor\sigma''_2\rfloor_{A^d} = \sigma''_d$ implies $\lfloor\sigma_2\rfloor_{A^d} = \lfloor\sigma'_2{:}o'{:}\sigma''_2\rfloor_{A^d} = o'{:}\sigma''_d = \sigma_d$. □

Lemma 8. *For all $d \in D$, $i_1, i_2 \in I$, $\sigma_1 \in O^*$, and $q_1, q'_1, q_2 \in Q$, if $q_1 \overset{d}{\sim} q_2$, $q_1 \xrightarrow{\sigma_1:i_1} q'_1$, $\mathsf{dom}(i_1) = \mathsf{dom}(i_2)$, and $\mathsf{dom}(i_1) \rightsquigarrow d$ implies $i_1 = i_2$, then there must exist $\sigma_2 \in O^*$ and $q'_2 \in Q$ such that $q_2 \xrightarrow{\sigma_2:i_2} q'_2$, $q'_1 \overset{d}{\sim} q'_2$, and $\lfloor\sigma_1{:}i_1\rfloor_{A^d} = \lfloor\sigma_2{:}i_2\rfloor_{A^d}$.*

Proof. In the case where $\lfloor\sigma_1\rfloor_{A^d} = []$, the result follows directly from Lemma 6.

Otherwise, σ_1 has the form $\sigma'_1{:}o{:}\sigma''_1$ where $\sigma'_1, \sigma''_2 \in O^*$, $o \in O$, $\mathsf{dom}(o) = d$, and $\lfloor\sigma''_1\rfloor_{A^d} = []$. Since $q_1 \xrightarrow{\sigma_1:i_1} q'_1$, there must exist $q''_1 \in Q$ such that $q_1 \xrightarrow{\sigma'_1:o_1} q''_1 \xrightarrow{\sigma''_1:i_1} q'_1$.

By Lemma 7, there must exist $\sigma'_2 \in O^*$ and $q''_2 \in Q$ such that $q_2 \xrightarrow{\sigma'_2:o} q''_2$, $q''_1 \overset{d}{\sim} q''_2$, and $\lfloor \sigma'_1{:}o \rfloor_{A^d} = \lfloor \sigma'_2{:}o \rfloor_{A^d}$. Since $q''_1 \overset{d}{\sim} q''_2$,$q''_1 \xrightarrow{\sigma''_1:i_1} q'_1$, and $\lfloor \sigma''_1 \rfloor_{A^d} = []$, Lemma 6 implies that there exists $\sigma''_1 \in O$ and $q'_2 inQ$ such that $q''_2 \xrightarrow{\sigma''_2:i_2} q'_2$, $q'_1 \overset{d}{\sim} q'_2$, and $\lfloor \sigma''_2 \rfloor_{A^d} = []$.

Let $\sigma_2 = \sigma'_2{:}o{:}\sigma''_2$. Since $q_2 \xrightarrow{\sigma'_2:o} q''_2 \xrightarrow{\sigma''_2:i_2} q'_2$, $q_2 \xrightarrow{\sigma_2:i_2} q'_2$ where $q'_1 \overset{d}{\sim} q'_2$. From $\lfloor \sigma'_1{:}o \rfloor_{A^d} = \lfloor \sigma'_2{:}o \rfloor_{A^d}$, $\lfloor \sigma''_1 \rfloor_{A^d} = [] = \lfloor \sigma''_2 \rfloor_{A^d}$, and the fact that $\mathsf{dom}(i_1) = d$ implies $i_1 = i_2$ ($\rightsquigarrow$ is reflexive), it follows that $\lfloor \sigma_1{:}i_1 \rfloor_{A^d} = \lfloor \sigma_2{:}i_2 \rfloor_{A_2}$. □

Lemma 9. *For all $d \in D$, $q_1, q_2, q'_1 \in Q$, $\iota_1, \iota_2 \in I^*$, $i_1, i_2 \in I$, and $\alpha_1 \in A^*$, if $\iota_1{:}i_1 \cong^{\rightsquigarrow,d}_{\mathrm{II}} \iota_2{:}i_2$, $\lfloor \alpha_1 \rfloor_I = \iota_1$, $q_1 \overset{d}{\sim} q_2$, and $q_1 \xrightarrow{\alpha_1:i_1} q'_1$, then there exists $\alpha_2 \in A^*$ and $q'_2 \in Q$ such that $q'_1 \overset{d}{\sim} q'_2$, $q_2 \xrightarrow{\alpha_2:i_2} q'_2$, $\lfloor \alpha_2 \rfloor_I = \iota_2$, and $\lfloor \alpha_1{:}i_1 \rfloor_{A^d} = \lfloor \alpha_2{:}i_2 \rfloor_{A^d}$.*

Proof. Proof by induction over the structure of ι_1.

Case: $\iota_1 = []$. Since $\iota_1{:}i_1 \cong^{\rightsquigarrow,d}_{\mathrm{II}} \iota_2{:}i_2$, ι_2 must be [], $\mathsf{dom}(i_1) = \mathsf{dom}(i_2)$, and $\mathsf{dom}(i_1) \rightsquigarrow d$ implies $i_1 = i_2$. Since $\lfloor \alpha_1 \rfloor_I = [i_1]$, there must exist $\sigma_1 \in O^*$ such that $\alpha_1 = \sigma_1{:}i_1$. Thus, Lemma 8 implies that there must exist $\sigma_2 \in O^*$ and $q'_2 \in Q$ such that $q_2 \xrightarrow{\sigma_2:i_2} q'_2$, $q'_1 \overset{d}{\sim} q'_2$, and $\lfloor \sigma_1{:}i_1 \rfloor_{A^d} = \lfloor \sigma_2{:}i_2 \rfloor_{A^d}$. Since $\lfloor \sigma_2 \rfloor_I = []$, $\lfloor \sigma_2{:}i_2 \rfloor_I = \iota_2{:}i_2$. Thus, the result holds with $\alpha_2 = \sigma_2$.

Case: $\iota_1 = i'_1{:}\iota'_1$. Since $\iota_1 \cong^{\rightsquigarrow,d}_{\mathrm{II}} \iota_2$, there must exist $i'_2 \in I$ and $\iota'_2 \in I^*$ such that $\iota_2 = i'_2{:}\iota'_2$, $\iota'_1 \cong^{\rightsquigarrow,d}_{\mathrm{II}} \iota'_2$, $\mathsf{dom}(i'_1) = \mathsf{dom}(i'_2)$, and $\mathsf{dom}(i'_1) \rightsquigarrow d$ implies $i'_1 = i'_2$. Since $\lfloor \alpha_1 \rfloor_I = \iota_1 = i'_1{:}\iota'_1$, there must exist $\sigma_1 \in O^*$ and $\alpha'_1 \in A^*$ such that $\alpha_1 = \sigma_1{:}i'_1{:}\alpha'_1$, $\lfloor \alpha'_1 \rfloor_I = \iota'_1$, and $q_1 \xrightarrow{\sigma_1:i'_1} q''_1 \xrightarrow{\alpha'_1:i_1} q'_1$.

Since $\mathsf{dom}(i'_1) = \mathsf{dom}(i'_2)$, $\mathsf{dom}(i'_1) \rightsquigarrow d$ implies $i'_1 = i'_2$, $q_1 \overset{d}{\sim} q_2$, and $q_1 \xrightarrow{\sigma_1:i'_1} q''_1$, Lemma 8 implies that there exists $q''_2 \in Q$ and $\sigma_2 \in O^*$ such that $q''_1 \overset{d}{\sim} q''_2$, $q_2 \xrightarrow{\sigma_2:i'_2} q''_2$ and $\lfloor \sigma_1{:}i'_1 \rfloor_{A^d} = \lfloor \sigma_2{:}i'_2 \rfloor_{A^d}$.

Since $\iota'_1{:}i_1 \cong^{\rightsquigarrow,d}_{\mathrm{II}} \iota'_2{:}i_2$, $\lfloor \alpha'_1 \rfloor_I = \iota'_1$, $q''_1 \overset{d}{\sim} q''_2$, and $q''_1 \xrightarrow{\alpha'_1:i_1} q'_1$, the inductive hypothesis implies that there must exist $\alpha'_2 \in A^*$ and $q'_2 \in Q$ such that $q'_1 \overset{d}{\sim} q'_2$, $q''_2 \xrightarrow{\alpha'_2:i_2} q'_2$, $\lfloor \alpha'_2 \rfloor_I = \iota'_2$, and $\lfloor \alpha'_1{:}i_1 \rfloor_{A^d} = \lfloor \alpha'_2{:}i_2 \rfloor_{A^d}$.

Let $\alpha_2 = \sigma_2{:}i'_2{:}\alpha'_2$. Since $q_2 \xrightarrow{\sigma_2:i'_2} q''_2$ and $q''_2 \xrightarrow{\alpha'_2:i_2} q'_2$, $q_2 \xrightarrow{\alpha_2:i_2} q'_2$. Since $\iota_2 = i'_2{:}\iota'_2$ and $\lfloor \alpha'_2 \rfloor_{A^d} = \iota'_2$, $\lfloor \alpha_2 \rfloor_{A^d} = \iota_2$. Since $\lfloor \sigma_1{:}i'_1 \rfloor_{A^d} = \lfloor \sigma_2{:}i'_2 \rfloor_{A^d}$ and $\lfloor \alpha'_1{:}i_1 \rfloor_{A^d} = \lfloor \alpha'_2{:}i_2 \rfloor_{A^d}$, $\lfloor \sigma_1{:}i'_1{:}\alpha'_1{:}i_1 \rfloor_{A^d} = \lfloor \sigma_2{:}i'_2{:}\alpha'_2{:}i_2 \rfloor_{A^d}$. That is, $\lfloor \alpha_1{:}i_1 \rfloor_{A^d} = \lfloor \alpha_2{:}i_2 \rfloor_{A^d}$. □

Theorem 2 is a corollary of the next lemma.

Lemma 10. *For all $d \in D$, $\iota_1, \iota_2 \in I^*$, $\alpha_1 \in A^*$, and $q_1 \in Q$, if $\iota_1 \cong^{\rightsquigarrow,d}_{\mathrm{II}} \iota_2$, $q_0 \xrightarrow{\alpha_1} q_1$, and $\lfloor \alpha_1 \rfloor_I = \iota_1$, then there exists $\alpha_2 \in A^*$ and $q_2 \in Q$ such that $q_0 \xrightarrow{\alpha_2} q_2$, $\lfloor \alpha_2 \rfloor_I = \iota_2$, and $\lfloor \alpha_1 \rfloor_{A^d} = \lfloor \alpha_2 \rfloor_{A^d}$.*

Proof. Consider the case where $\alpha_1 = \alpha'_1{:}i_1{:}\sigma_1{:}o{:}\sigma'_1$ with $\alpha'_1 \in A^*$, $i_1 \in I$, and $\sigma_1, \sigma'_1 \in O^*$, $o \in O$, $\mathsf{dom}(o) = d$, and $\lfloor \sigma'_1 \rfloor_{A^d} = []$. Since $q_0 \xrightarrow{\alpha_1} q_1$, there must exist $q'_1, q''_1 \in Q$ such that $q_0 \xrightarrow{\alpha'_1:i_1} q'_1 \xrightarrow{\sigma_1:o} q''_1 \xrightarrow{\sigma'_1} q_1$.

Since $\lfloor \alpha_1 \rfloor_I = \iota_1$, it must be the case that $\iota_1 = \iota'_1{:}i_1$ where $\iota'_1 = \lfloor \alpha'_1 \rfloor_I$. Furthermore, since $\iota_1 \cong^{\rightsquigarrow,d}_{\mathrm{II}} \iota_2$, $\iota'_1{:}i_1 \cong^{\rightsquigarrow,d}_{\mathrm{II}} \iota_2$. This implies that ι_2 must have the form $\iota'_2{:}i_2$ where $\mathsf{dom}(i_1) = \mathsf{dom}(i_2)$ and $\mathsf{dom}(i_1) \rightsquigarrow d$ implies that $i_1 = i_2$. Thus, by Lemma 9, there must exist $\alpha'_2 \in A^*$ and $q'_2 \in Q$ such that $q'_1 \overset{d}{\sim} q'_2$, $q_0 \xrightarrow{\alpha'_2:i_2} q'_2$, $\lfloor \alpha'_2 \rfloor_I = \iota'_2$, and $\lfloor \alpha'_1{:}i_1 \rfloor_{A^d} = \lfloor \alpha'_2{:}i_2 \rfloor_{A^d}$.

Since $q'_1 \overset{d}{\sim} q'_2$ and $q'_1 \xrightarrow{\sigma_1:o} q''_1$, Lemma 7 implies that there must exist $\sigma_2 \in O^*$ and $q_2 \in Q$ such that $q'_2 \xrightarrow{\sigma_2:o} q_2$, $q''_1 \overset{d}{\sim} q_2$, and $\lfloor \sigma_1 \rfloor_{A^d} = \lfloor \sigma_2 \rfloor_{A^d}$.

Let $\alpha_2 = \alpha'_2{:}i_2{:}\sigma_2{:}o$. Since $q_0 \xrightarrow{\alpha'_2:i_2} q'_2$ and $q'_2 \xrightarrow{\sigma_2:o} q_2$, $q_0 \xrightarrow{\alpha_2} q_2$. $\lfloor \alpha_2 \rfloor_I = \lfloor \alpha'_2{:}i_2{:}\sigma_2{:}o \rfloor_I = \lfloor \alpha'_2 \rfloor_I{:}i_2 = \iota'_2{:}i_2 = \iota_2$. Since $\lfloor \alpha'_1{:}i_1 \rfloor_{A^d} = \lfloor \alpha'_2{:}i_2 \rfloor_{A^d}$, $\lfloor \sigma_1 \rfloor_{A^d} = \lfloor \sigma_2 \rfloor_{A^d}$, and $\lfloor \sigma'_1 \rfloor_{A^d} = []$, $\lfloor \alpha_2 \rfloor_{A^d} = \lfloor \alpha'_2{:}i_2{:}\sigma_2{:}o \rfloor_{A^d} = \lfloor \alpha'_2{:}i_2 \rfloor_{A^d}{:}\lfloor \sigma_2 \rfloor_{A^d}{:}o = \lfloor \alpha'_1{:}i_1 \rfloor_{A^d}{:}\lfloor \sigma_1 \rfloor_{A^d}{:}o = \lfloor \alpha'_1{:}i_1{:}\sigma_1{:}o{:}\sigma'_1 \rfloor_{A^d} = \lfloor \alpha_1 \rfloor_{A^d}$.

In cases where α_1 is not of the form $\alpha'_1{:}i_1{:}\sigma_1{:}o{:}\sigma'_1$, some subset of the above arguments are sufficient to achieve the same result. □

B Proof of Theorem 3

First, we must define some new notation.

Let $q_1 \underset{d':\delta}{\overset{d}{\sim}} q_2$ for $d'{:}\delta \in D^*$ iff $q_1 \underset{d'}{\overset{d}{\sim}} q_3$ and $q_3 \underset{\delta}{\overset{d}{\sim}} q_2$. Let $q \underset{[]}{\overset{d}{\sim}} q$ hold for all d and q. For $D' \subseteq D$, let $q_1 \underset{D'}{\overset{d}{\sim}} q_2$ iff there exists $\delta \in (D')^*$ such that $q_1 \underset{\delta}{\overset{d}{\sim}} q_2$.

For $D' \subseteq D$, let $D' \not\rightsquigarrow^{q} d$ mean that for all $d' \in D'$, $d' \not\rightsquigarrow^{q} d$.

Lemma 11. *For all $D' \subseteq D$, $\alpha \in A^*$, and $q, q' \in Q$, if the the state-based dynamic policy $\rightsquigarrow$ is a non-revoking safe approximation of the dynamic policy $\rightsquigarrow$, $q \xrightarrow{\alpha} q'$, and $D' \not\rightsquigarrow^{q'} d$, then $D' \not\rightsquigarrow^{q} d$.*

Proof. Consider each $d' \in D'$ separately, this follows from the contrapositive of the fact that $\rightsquigarrow$ is non-revoking. □

Lemma 12. *Let $\rightsquigarrow$ be a state-based safe approximation of the dynamic policy $\rightsquigarrow$ for some automaton $m = \langle I, O, D, \mathsf{dom}, Q, q_0, \rightarrow \rangle$. For all $d \in D$, $q_1 \in Q$, $\iota_1, \iota_2 \in I^*$, $i_1, i_2 \in I$, and $\alpha \in A^*$, if $\iota_1{:}i_1 \cong^{\rightsquigarrow^{\iota_1:i_1},d}_{\mathrm{II}} \iota_2{:}i_2$, $\lfloor \alpha \rfloor_I = \iota_1$, and $q_0 \xrightarrow{\alpha:i_1} q_1$, then $\mathsf{dom}(i_1) \rightsquigarrow^{q_1} d$ implies $i_1 = i_2$.*

Proof. Since $\iota_1{:}i_1 \cong^{\rightsquigarrow^{\iota_1:i_1},d}_{\mathrm{II}} \iota_2{:}i_2$, $\mathsf{dom}(i_1) = \mathsf{dom}(i_2)$ and $\mathsf{dom}(i_1) \rightsquigarrow^{\iota_1:i_1} d$ implies $i_1 = i_2$. Since $\rightsquigarrow$ is a safe approximation of $\leadsto$ and $q_0 \xrightarrow{\alpha:i_1} q_1$, $\mathsf{dom}(i_1) \not\leadsto^{\iota_1:i_1} d$ implies $\mathsf{dom}(i_1) \not\rightsquigarrow^{q_1} d$. Thus, by taking the contrapositive, $\mathsf{dom}(i_1) \rightsquigarrow^{q_1} d$ implies $\mathsf{dom}(i_1) \leadsto^{\iota_1:i_1} d$. This means that $\mathsf{dom}(i_1) \rightsquigarrow^{q_1} d$ implies $i_1 = i_2$. □

In each of the following lemmas let $\cdot \overset{\cdot}{\underset{\cdot}{\sim}} \cdot$ be an dynamic unwinding relation for the automaton $m = \langle I, O, D, \mathsf{dom}, Q, q_0, \rightarrow\rangle$ and state-based dynamic policy $\rightsquigarrow$ where $\rightsquigarrow$ is a non-revoking safe approximation of the dynamic policy $\leadsto$.

The next two lemmas just raise up the second two unwinding conditions to work over sets.

Lemma 13 (Set Step Consisteny). *For all $d \in D$, $D' \subseteq D$, $i \in I$, and $q_1, q_1', q_2 \in Q$, if $q_1 \overset{d}{\underset{D'}{\sim}} q_2$, $q_1 \xrightarrow[d]{i} q_1'$, and $D' \not\rightsquigarrow^{q_1'} d$, then there must exist $q_2' \in Q$ such that $q_2 \xrightarrow[d]{i} q_2'$ and $q_1' \overset{d}{\underset{D'}{\sim}} q_2'$.*

Proof. We will actually prove the following slightly stronger statement: For all $d \in D$, $D' \subseteq D$, $\delta \in (D')^*$, $i \in I$, and $q_1, q_1', q_2 \in Q$, if $q_1 \overset{d}{\underset{\delta}{\sim}} q_2$, $q_1 \xrightarrow[d]{i} q_1'$, and $D' \not\rightsquigarrow^{q_1'} d$, then there must exist $q_2' \in Q$ such that $q_2 \xrightarrow[d]{i} q_2'$ and $q_1' \overset{d}{\underset{\delta}{\sim}} q_2'$.

Proof by induction over the structure of δ.

Case: $\delta = []$. In this case $q_1 = q_2$. Thus, let $q_2' = q_1'$. Then $q_1' \overset{d}{\underset{[]}{\sim}} q_2'$ by definition.

Case: $\delta = d':\delta'$. In this case, $q_1 \overset{d}{\underset{d'}{\sim}} q_3$ and $q_3 \overset{d}{\underset{\delta'}{\sim}} q_2$ for some d' and q_3. Since $d' \in D'$, $d' \not\rightsquigarrow^{q_1'} d$. Thus, by Step Consistency, there must exist a q_3' such that $q_3 \xrightarrow[d]{i} q_3'$ and $q_1' \overset{d}{\underset{d'}{\sim}} q_3'$. By the inductive hypothesis, there must exist $q_2' \in Q$ such that $q_2 \xrightarrow[d]{i} q_2'$ and $q_3' \overset{d}{\underset{\delta'}{\sim}} q_2'$. Thus, $q_1' \overset{d}{\underset{d':\delta'}{\sim}} q_2'$. □

Lemma 14 (Set Output Consisteny). *For all $d \in D$, $D' \subseteq D$, $o \in O$, and $q_1, q_1', q_2 \in Q$, if $\mathsf{dom}(o) = d$, $q_1 \overset{d}{\underset{D'}{\sim}} q_2$, $q_1 \xrightarrow[d]{o} q_1'$, and $D' \not\rightsquigarrow^{q_1'} d$, then there must exist $q_2' \in Q$ such that $q_2 \xrightarrow[d]{o} q_2'$ and $q_1' \overset{d}{\underset{D'}{\sim}} q_2'$.*

Proof. We will actually prove the following slightly stronger statement: For all $d \in D$, $D' \subseteq D$, $\delta \in (D')^*$, $o \in O$, and $q_1, q_1', q_2 \in Q$, if $\mathsf{dom}(o) = d$, $q_1 \overset{d}{\underset{\delta}{\sim}} q_2$, $q_1 \xrightarrow[d]{o} q_1'$, and for all $D' \not\rightsquigarrow^{q_1'} d$, then there must exist $q_2' \in Q$ such that $q_2 \xrightarrow[d]{o} q_2'$ and $q_1' \overset{d}{\underset{\delta}{\sim}} q_2'$.

Proof by induction over the structure of δ.

Case: $\delta = []$. In this case $q_1 = q_2$. Thus, let $q_2' = q_1'$. Then $q_1' \overset{d}{\underset{[]}{\sim}} q_2'$ by definition.

Case: $\delta = d':\delta'$. In this case, $q_1 \overset{d}{\underset{d'}{\sim}} q_3$ and $q_3 \overset{d}{\underset{\delta'}{\sim}} q_2$ for some d' and q_3. Since $d' \in D'$, $d' \not\rightsquigarrow^{q_1'} d$. Thus, by Output Consistency, there must exist a q_3' such that $q_3 \xrightarrow[d]{o} q_3'$ and $q_1' \overset{d}{\underset{d'}{\sim}} q_3'$. By the inductive hypothesis, there must exist $q_2' \in Q$ such that $q_2 \xrightarrow[d]{o} q_2'$ and $q_3' \overset{d}{\underset{\delta'}{\sim}} q_2'$. Thus, $q_1' \overset{d}{\underset{d':\delta'}{\sim}} q_2'$. □

Now we raise Step Respect to work over sets.

Lemma 15 (Set Step Respect). *For all $d_{\mathsf{t}}, d_{\mathsf{f}} \in D$, $D' \subseteq D$, $i_1, i_2 \in I$, and $q_1, q_1', q_2 \in Q$, if $\mathsf{dom}(i_1) = \mathsf{dom}(i_2) = d_{\mathsf{f}}$, $q_1 \overset{d_{\mathsf{t}}}{\underset{D'}{\sim}} q_2$, $q_1 \xrightarrow[d_{\mathsf{t}}]{i_1} q_1'$, $d_{\mathsf{f}} \in D'$, and $D' \not\rightsquigarrow^{q_1'} d_{\mathsf{t}}$, then there must exist $q_2' \in Q$ such that $q_2 \xrightarrow[d_{\mathsf{t}}]{i_2} q_2'$ and $q_1' \overset{d_{\mathsf{t}}}{\underset{D'}{\sim}} q_2'$.*

Proof. Since $q_1 \overset{d_{\mathsf{t}}}{\underset{D'}{\sim}} q_2$, Set Step Consistency (Lemma 13) implies that there must exist a q_2'' such that $q_2 \xrightarrow[d_{\mathsf{t}}]{i_1} q_2''$ and $q_1' \overset{d_{\mathsf{t}}}{\underset{D'}{\sim}} q_2''$. By Local Respect, there must exist a q_2' such that $q_2 \xrightarrow[d_{\mathsf{t}}]{i_2} q_2'$ and $q_2'' \overset{d_{\mathsf{t}}}{\underset{d_{\mathsf{f}}}{\sim}} q_2'$.

Since $q_1' \overset{d_{\mathsf{t}}}{\underset{D'}{\sim}} q_2''$, there must exist $\delta \in (D')^*$ such that $q_1' \overset{d_{\mathsf{t}}}{\underset{\delta}{\sim}} q_2''$. Since $q_2'' \overset{d_{\mathsf{t}}}{\underset{d_{\mathsf{f}}}{\sim}} q_2'$, $q_1' \overset{d_{\mathsf{t}}}{\underset{\delta:d_{\mathsf{f}}}{\sim}} q_2'$. Thus, since $d', d_{\mathsf{f}} \in D'$, $q_1' \overset{d_{\mathsf{t}}}{\underset{D'}{\sim}} q_2'$. □

The next five lemmas mirror the corresponding five lemmas (Lemmas 5 to 9) of Section A.3 very closely.

Lemma 16. *For all $d \in D$, $D' \subseteq D$, $o \in O$, $\sigma_1 \in O^*$, and $q_1, q_1', q_2 \in Q$, if $\mathsf{dom}(o) = d$, $\lfloor \sigma_1 \rfloor_{A^d} = []$, $q_1 \overset{d}{\underset{D'}{\sim}} q_2$, $q_1 \xrightarrow{\sigma_1:o} q_1'$, and $D' \not\rightsquigarrow^{q_1'} d$, then there must exist $\sigma_2 \in O^*$ and $q_2' \in Q$ such that $q_2 \xrightarrow{\sigma_2:o} q_2'$, $q_1' \overset{d}{\underset{D'}{\sim}} q_2'$, and $\lfloor \sigma_2 \rfloor_{A^d} = []$.*

Proof. Since $\lfloor \sigma_1 \rfloor_{A^d} = []$, $q_1 \xrightarrow{\sigma_1:o} q_1'$ implies that $q_1 \xrightarrow[d]{o} q_1'$. Thus, by Set Output Consistency (Lemma 14), there must exist q_2' such that $q_2 \xrightarrow[d]{o} q_2'$ and $q_1' \overset{d}{\underset{D'}{\sim}} q_2'$. $q_2 \xrightarrow[d]{o} q_2'$ implies that there exists σ_2 such that $q_2 \xrightarrow{\sigma_2:o} q_2'$ and $\lfloor \sigma_2 \rfloor_{A^d} = []$. □

Lemma 17. *For all $d \in D$, $D' \subseteq D$, $i_1, i_2 \in I$, $\sigma_1 \in O^*$, and $q_1, q_1', q_2 \in Q$, $\alpha \in A^*$, if $\lfloor \sigma_1 \rfloor_{A_d} = []$, $q_1 \overset{d}{\underset{D'}{\sim}} q_2$, $q_1 \xrightarrow{\sigma_1:i_1} q_1'$, $\mathsf{dom}(i_1) = \mathsf{dom}(i_2)$, $\mathsf{dom}(i_1) \notin D'$ implies $i_1 = i_2$, and $D' \not\rightsquigarrow^{q_1'} d$, then there must exist $\sigma_2 \in O^*$ and $q_2' \in Q$ such that $q_2 \xrightarrow{\sigma_2:i_2} q_2'$, $q_1' \overset{d}{\underset{D'}{\sim}} q_2'$, and $\lfloor \sigma_2 \rfloor_{A^d} = []$.*

Proof. $\lfloor\sigma_1\rfloor_{A^d} = []$, $q_1 \xrightarrow{\sigma_1:i_1} q_1'$ implies that $q_1 \xrightarrow[d]{i_1} q_1'$. Consider the following two cases:

- $\mathsf{dom}(i_1) \in D'$. Since $D' \not\rightsquigarrow^{q_1'} d$, $\mathsf{dom}(i_1) \not\rightsquigarrow^{q_1'} d$. Thus, by Set Step Respect (Lemma 15, there must exist q_2' such that $q_2 \xrightarrow[d]{i_2} q_2'$ and $q_1' \overset{d}{\underset{D'}{\sim}} q_2'$.
- $d \notin D'$. Set Step Consistency (Lemma 13) implies the same thing in this case.

In either case, $q_2 \xrightarrow[d]{i} q_2'$ implies that there exists σ_2 such that $q_2 \xrightarrow{\sigma_2:i} q_2'$ and $\lfloor\sigma_2\rfloor_{A^d} = []$. □

Lemma 18. *For all $d \in D$, $D' \subseteq D$, $o \in O$, $\sigma_1, \sigma_d \in O^*$, and $q_1, q_1', q_2 \in Q$, if $\mathsf{dom}(o) = d$, $q_1 \overset{d}{\underset{D'}{\sim}} q_2$, $q_1 \xrightarrow{\sigma_1:o} q_1'$, $\sigma_d = \lfloor\sigma_1\rfloor_{A^d}$, and $D' \not\rightsquigarrow^{q_1'} d$, then there must exist $\sigma_2 \in O^*$ and $q_2' \in Q$ such that $q_2 \xrightarrow{\sigma_2:o} q_2'$, $q_1' \overset{d}{\underset{D'}{\sim}} q_2'$, and $\lfloor\sigma_2\rfloor_{A^d} = \sigma_d$.*

Proof. Proof by induction over the structure of σ_d.

Case: $\sigma_d = []$. The result follows directly from Lemma 16.

Case: $\sigma_d = o':\sigma_d''$. In this case, σ_1 must have the form $\sigma_1':o':\sigma_1''$ where $\lfloor\sigma_1'\rfloor_{A^d} = []$ and $\lfloor\sigma_1''\rfloor_{A^d} = \sigma_d''$. Since $q_1 \xrightarrow{\sigma_1:o} q_1'$, there must exist q_2'' such that $q_1 \xrightarrow{\sigma_1':o'} q_1'' \xrightarrow{\sigma_1'':o} q_1'$. By Lemma 11, $q_1'' \xrightarrow{\sigma_1'':o} q_1'$ implies $D' \not\rightsquigarrow^{q_1''} d$.

By Lemma 16, there must exist $\sigma_2' \in O^*$ and $q_2'' \in Q$ such that $q_2 \xrightarrow{\sigma_2':o'} q_2''$, $q_1'' \overset{d}{\underset{D'}{\sim}} q_2''$, and $\lfloor\sigma_2'\rfloor_{A^d} = []$. By the inductive hypothesis, there must exist $\sigma_2'' \in O^*$ and $q_2' \in Q$ such that $q_2'' \xrightarrow{\sigma_2'':o} q_2'$, $q_1' \overset{d}{\underset{D'}{\sim}} q_2'$, and $\lfloor\sigma_2''\rfloor_{A^d} = \sigma_d''$.

Let $\sigma_2 = \sigma_2':o':\sigma_2''$. $q_2 \xrightarrow{\sigma_2':o'} q_2''$ and $q_2'' \xrightarrow{\sigma_2'':o} q_2'$ implies $q_2 \xrightarrow{\sigma_2:o} q_2'$. $\lfloor\sigma_2'\rfloor_{A^d} = []$ and $\lfloor\sigma_2''\rfloor_{A^d} = \sigma_d''$ implies $\lfloor\sigma_2\rfloor_{A^d} = \lfloor\sigma_2':o':\sigma_2''\rfloor_{A^d} = o':\sigma_d'' = \sigma_d$. □

Lemma 19. *For all $d \in D$, $D' \subseteq D$, $i_1, i_2 \in I$, $\sigma_1 \in O^*$, and $q_1, q_1', q_2 \in Q$, if $q_1 \overset{d}{\underset{D'}{\sim}} q_2$, $q_1 \xrightarrow{\sigma_1:i_1} q_1'$, $\mathsf{dom}(i_1) = \mathsf{dom}(i_2)$, $D' \not\rightsquigarrow^{q_1'} d$, and $\mathsf{dom}(i_1) \notin D'$ implies $i_1 = i_2$, then there must exist $\sigma_2 \in O^*$ and $q_2' \in Q$ such that $q_2 \xrightarrow{\sigma_2:i_2} q_2'$, $q_1' \overset{d}{\underset{D'}{\sim}} q_2'$, and $\lfloor\sigma_1:i_1\rfloor_{A^d} = \lfloor\sigma_2:i_2\rfloor_{A^d}$.*

Proof. In the case where $\lfloor\sigma_1\rfloor_{A^d} = []$, the result follows directly from Lemma 17.

Otherwise, σ_1 has the form $\sigma_1':o:\sigma_1''$ where $\sigma_1', \sigma_2'' \in O^*$, $o \in O$, $\mathsf{dom}(o) = d$, and $\lfloor\sigma_1''\rfloor_{A^{d_t}} = []$. Since $q_1 \xrightarrow{\sigma_1:i_1} q_1'$, there must exist $q_1'' \in Q$ such that $q_1 \xrightarrow{\sigma_1':o_1} q_1'' \xrightarrow{\sigma_1'':i_1} q_1'$. By Lemma 11, $q_1'' \xrightarrow{\sigma_1'':i_1} q_1'$ implies $D' \not\rightsquigarrow^{q_1''} d$.

By Lemma 18, there must exist $\sigma_2' \in O^*$ and $q_2'' \in Q$ such that $q_2 \xrightarrow{\sigma_2':o} q_2''$, $q_1'' \overset{d}{\underset{D'}{\sim}} q_2''$, and $\lfloor\sigma_1':o\rfloor_{A^d} = \lfloor\sigma_2':o\rfloor_{A^d}$. Since $q_1'' \overset{d}{\underset{D'}{\sim}} q_2''$, $q_1'' \xrightarrow{\sigma_1'':i_1} q_1'$, and $\lfloor\sigma_1''\rfloor_{A^d} = []$, Lemma 17 implies that there exists $\sigma_1'' \in O$ and $q_2' \in Q$ such that $q_2'' \xrightarrow{\sigma_2'':i_2} q_2'$, $q_1' \overset{d}{\underset{D'}{\sim}} q_2'$, and $\lfloor\sigma_2''\rfloor_{A^d} = []$.

Let $\sigma_2 = \sigma_2':o:\sigma_2''$. Since $q_2 \xrightarrow{\sigma_2':o} q_2'' \xrightarrow{\sigma_2'':i_2} q_2'$, $q_2 \xrightarrow{\sigma_2:i_2} q_2'$ where $q_1' \overset{d}{\underset{D'}{\sim}} q_2'$. From $\lfloor\sigma_1':o\rfloor_{A^d} = \lfloor\sigma_2':o\rfloor_{A^d}$, $\lfloor\sigma_1''\rfloor_{A^d} = [] = \lfloor\sigma_2''\rfloor_{A^d}$, and the fact that $\mathsf{dom}(i_1) = d$ implies $i_1 = i_2$ ($\rightsquigarrow^{q_1'}$ is reflexive, so $d \notin D'$), it follows that $\lfloor\sigma_1:i_1\rfloor_{A^d} = \lfloor\sigma_2:i_2\rfloor_{A_2}$. □

Lemma 20. *For all $d \in D$, $D' \subseteq D$, $q_1 \in Q$, $\iota_1, \iota_2 \in I^*$, $i_1, i_2 \in I$, and $\alpha_1 \in A^*$, if $D' = \{\, d' \in D \mid d' \not\rightsquigarrow^{q_1} d \,\}$, $\iota_1:i_1 \cong_{\text{II}}^{\rightsquigarrow^{\iota_1:i_1},d} \iota_2:i_2$, $\lfloor\alpha_1\rfloor_I = \iota_1$, and $q_0 \xrightarrow{\alpha_1:i_1} q_1$, then there exists $\alpha_2 \in A^*$ and $q_2 \in Q$ such that $q_1 \overset{d}{\underset{D'}{\sim}} q_2$, $q_0 \xrightarrow{\alpha_2:i_2} q_2$, $\lfloor\alpha_2\rfloor_I = \iota_2$, and $\lfloor\alpha_1:i_1\rfloor_{A^d} = \lfloor\alpha_2:i_2\rfloor_{A^d}$.*

Proof. Proof by induction over the structure of ι_1.

Case: $\iota_1 = []$. Since $\iota_1:i_1 \cong_{\text{II}}^{\rightsquigarrow^{\iota_1:i_1},d} \iota_2:i_2$, Lemma 12 yields that $\mathsf{dom}(i_1) \rightsquigarrow^{q_1} d$ implies $i_1 = i_2$. Also, ι_2 must be $[]$. Since $D' = \{\, d' \in D \mid d' \not\rightsquigarrow^{q_1} d \,\}$, $\mathsf{dom}(i_1) \notin D'$ implies $i_1 = i_2$. Since $\lfloor\alpha_1\rfloor_I = [i_1]$, there must exist $\sigma_1 \in O^*$ such that $\alpha_1 = \sigma_1:i_1$. Thus, Lemma 19 implies that there must exist $\sigma_2 \in O^*$ and $q_2' \in Q$ such that $q_0 \xrightarrow{\sigma_2:i_2} q_2$, $q_1 \overset{d}{\underset{D'}{\sim}} q_2$, and $\lfloor\sigma_1:i_1\rfloor_{A^d} = \lfloor\sigma_2:i_2\rfloor_{A^d}$. Since $\lfloor\sigma_2\rfloor_I = []$, $\lfloor\sigma_2:i_2\rfloor_I = \iota_2:i_2$. Thus, the result holds with $\alpha_2 = \sigma_2$.

Case: $\iota_1 = i_1':\iota_1'$. Since $\iota_1 \cong_{\text{II}}^{\rightsquigarrow^{\iota_1:i_1},d} \iota_2$, there must exist $i_2' \in I$ and $\iota_2' \in I^*$ such that $\iota_2 = i_2':\iota_2'$, $\iota_1' \cong_{\text{II}}^{\rightsquigarrow^{\iota_1:i_1},d} \iota_2'$, $\mathsf{dom}(i_1') = \mathsf{dom}(i_2')$, and $\mathsf{dom}(i_1') \rightsquigarrow^{\iota_1:i_1} d$ implies $i_1' = i_2'$. Following the same logic as above, this allows us to conclude that $\mathsf{dom}(i_1') \notin D'$ implies $i_1' = i_2'$.

Since $\lfloor\alpha_1\rfloor_I = \iota_1 = i_1':\iota_1'$, there must exist $\sigma_1 \in O^*$ and $\alpha_1' \in A^*$ such that $\alpha_1 = \sigma_1:i_1':\alpha_1'$, $\lfloor\alpha_1'\rfloor_I = \iota_1'$, and $q_0 \xrightarrow{\sigma_1:i_1'} q_1' \xrightarrow{\alpha_1':i_1} q_1$. By Lemma 11, $q_1' \xrightarrow{\sigma_1'':i_1} q_1$ implies $D' \not\rightsquigarrow^{q_1'} d$.

Since $\mathsf{dom}(i_1') = \mathsf{dom}(i_2')$, $\mathsf{dom}(i_1') \rightsquigarrow^{q_1'} d$ implies $i_1' = i_2'$, $q_0 \overset{d}{\underset{D'}{\sim}} q_0$, and $q_0 \xrightarrow{\sigma_1:i_1'} q_1'$, Lemma 19 implies that there exists $q_2' \in Q$ and $\sigma_2 \in O^*$ such that $q_1' \overset{d}{\underset{D'}{\sim}} q_2'$, $q_0 \xrightarrow{\sigma_2:i_2'} q_2'$ and $\lfloor\sigma_1:i_1'\rfloor_{A^d} = \lfloor\sigma_2:i_2'\rfloor_{A^d}$.

Since $\iota_1':i_1 \cong_{\text{II}}^{\rightsquigarrow,d} \iota_2':i_2$, $\lfloor\alpha_1'\rfloor_I = \iota_1'$, $q_1' \overset{d}{\underset{D'}{\sim}} q_2'$, and $q_1' \xrightarrow{\alpha_1':i_1} q_1$, the inductive hypothesis implies that there must exist $\alpha_2' \in A^*$ and $q_2 \in Q$ such that $q_1 \overset{d}{\underset{D'}{\sim}} q_2$, $q_2' \xrightarrow{\alpha_2':i_2} q_2$, $\lfloor\alpha_2'\rfloor_I = \iota_2'$, and $\lfloor\alpha_1':i_1\rfloor_{A^d} = \lfloor\alpha_2':i_2\rfloor_{A^d}$.

Let $\alpha_2 = \sigma_2:i_2':\alpha_2'$. Since $q_0 \xrightarrow{\sigma_2:i_2'} q_2'$ and $q_2' \xrightarrow{\alpha_2':i_2} q_2$, $q_0 \xrightarrow{\alpha_2:i_2} q_2$. Since $\iota_2 = i_2':\iota_2'$ and $\lfloor\alpha_2'\rfloor_{A^d} = \iota_2'$, $\lfloor\alpha_2\rfloor_{A^d} = \iota_2$. Since $\lfloor\sigma_1:i_1'\rfloor_{A^d} = \lfloor\sigma_2:i_2'\rfloor_{A^d}$ and $\lfloor\alpha_1':i_1\rfloor_{A^d} =$

$\lfloor\alpha'_2{:}i_2\rfloor_{A^d}$, $\lfloor\sigma_1{:}i'_1{:}\alpha'_1{:}i_1\rfloor_{A^d} = \lfloor\sigma_2{:}i'_2{:}\alpha'_2{:}i_2\rfloor_{A^d}$. That is, $\lfloor\alpha_1{:}i_1\rfloor_{A^d} = \lfloor\alpha_2{:}i_2\rfloor_{A^d}$. □

Theorem 3 is a corollary of the next lemma.

Lemma 21. *For all $d \in D$, $\iota_1, \iota_2 \in I^*$, $\alpha_1 \in A^*$, and $q_1 \in Q$, if $\iota_1 \cong_{\text{II}}^{\rightsquigarrow^{\iota_1},d} \iota_2$, $q_0 \xrightarrow{\alpha_1} q_1$, and $\lfloor\alpha_1\rfloor_I = \iota_1$, then there exists $\alpha_2 \in A^*$ and $q_2 \in Q$ such that $q_0 \xrightarrow{\alpha_2} q_2$, $\lfloor\alpha_2\rfloor_I = \iota_2$, and $\lfloor\alpha_1\rfloor_{A^d} = \lfloor\alpha_2\rfloor_{A^d}$.*

Proof. Consider the case where $\alpha_1 = \alpha'_1{:}i_1{:}\sigma_1{:}o{:}\sigma'_1$ with $\alpha'_1 \in A^*$, $i_1 \in I$, $\sigma_1, \sigma'_1 \in O^*$, $o \in O$, $\mathsf{dom}(o) = d$, and $\lfloor\sigma'_1\rfloor_{A^{d_\mathsf{t}}} = []$. Since $q_0 \xrightarrow{\alpha_1} q_1$, there must exist $q'_1, q''_1 \in Q$ such that $q_0 \xrightarrow{\alpha'_1:i_1} q'_1 \xrightarrow{\sigma_1:o} q''_1 \xrightarrow{\sigma'_1} q_1$.

Since $\lfloor\alpha_1\rfloor_I = \iota_1$ and $\alpha_1 = \alpha'_1{:}i_1{:}\sigma_1{:}o{:}\sigma'_1$, it must be the case that $\iota_1 = \iota'_1{:}i_1$. Furthermore since $\iota_1 \cong_{\text{II}}^{\rightsquigarrow^{\iota_1},d} \iota_2$, ι_2 must have the form $\iota'_2{:}i_2$ for some $\iota'_2 \in I^*$ and $i_2 \in I$. Thus, by Lemma 12, this means that $\mathsf{dom}(i_1) \rightsquigarrow^{q_1} d$ implies $i_1 = i_2$. Let $D' = \{\, d' \in D \mid d' \not\rightsquigarrow^{q'_1} d \,\}$. By Lemma 11, $q''_1 \xrightarrow{\sigma'_1} q_1$ implies $D' \not\rightsquigarrow^{q''_1} d$, and $q'_1 \xrightarrow{\sigma_1:o} q''_1$ implies $D' \not\rightsquigarrow^{q'_1} d$.

By Lemma 20, there must exist $\alpha'_2 \in A^*$ and $q'_2 \in Q$ such that $q'_1 \overset{d}{\underset{D'}{\sim}} q'_2$, $q_0 \xrightarrow{\alpha'_2:i_2} q'_2$, $\lfloor\alpha'_2\rfloor_I = \iota'_2$, and $\lfloor\alpha'_1{:}i_1\rfloor_{A^d} = \lfloor\alpha'_2{:}i_2\rfloor_{A^d}$.

Since $q'_1 \overset{d}{\underset{D'}{\sim}} q'_2$ and $q'_1 \xrightarrow{\sigma_1:o} q''_1$, Lemma 18 implies that there must exist $\sigma_2 \in O^*$ and $q_2 \in Q$ such that $q'_2 \xrightarrow{\sigma_2:o} q_2$, $q''_1 \overset{d}{\underset{D'}{\sim}} q_2$, and $\lfloor\sigma_1\rfloor_{A^d} = \lfloor\sigma_2\rfloor_{A^d}$.

Let $\alpha_2 = \alpha'_2{:}i_2{:}\sigma_2{:}o$. Since $q_0 \xrightarrow{\alpha'_2:i_2} q'_2$ and $q'_2 \xrightarrow{\sigma_2:o} q_2$, $q_0 \xrightarrow{\alpha_2} q_2$. $\lfloor\alpha_2\rfloor_I = \lfloor\alpha'_2{:}i_2{:}\sigma_2{:}o\rfloor_I = \lfloor\alpha'_2\rfloor_I{:}i_2 = \iota'_2{:}i_2 = \iota_2$. Since $\lfloor\alpha'_1{:}i_1\rfloor_{A^d} = \lfloor\alpha'_2{:}i_2\rfloor_{A^d}$, $\lfloor\sigma_1\rfloor_{A^d} = \lfloor\sigma_2\rfloor_{A^d}$, and $\lfloor\sigma'_1\rfloor_{A^d} = []$, $\lfloor\alpha_2\rfloor_{A^d} = \lfloor\alpha'_2{:}i_2{:}\sigma_2{:}o\rfloor_{A^d} = \lfloor\alpha'_2{:}i_2\rfloor_{A^d}{:}\lfloor\sigma_2\rfloor_{A^d}{:}o = \lfloor\alpha'_1{:}i_1\rfloor_{A^d}{:}\lfloor\sigma_1\rfloor_{A^d}{:}o = \lfloor\alpha'_1{:}i_1{:}\sigma_1{:}o{:}\sigma'_1\rfloor_{A^d} = \lfloor\alpha_1\rfloor_{A^d}$.

In cases where α_1 is not of the form $\alpha'_1{:}i_1{:}\sigma_1{:}o{:}\sigma'_1$, some subset of the above arguments are sufficient to achieve the same result. □

C The Correctness of Our Approach

First, we must relate $\mathsf{model}(s)$ and $\mathsf{autom}(s)$. Given the model $\mathsf{model}(s) = \langle I, O, D, \mathsf{dom}, Q, q_0, \rightarrowtail\rangle$, let $\exists(\mathsf{model}(s))$ be the system automaton $\langle I, O, D, \mathsf{dom}, Q, q_0, \to\rangle$ where $q_1 \overset{a}{\to} q_2$ iff $q_1 \overset{a}{\underset{\mathsf{T}}{\rightarrowtail}} q_2$ or $q_1 \overset{a}{\underset{\mathsf{F}}{\rightarrowtail}} q_2$.

Lemma 22. *For all programs s, $\exists(\mathsf{model}(s))$ and $\mathsf{autom}(s)$ have the same set of behaviors.*

This lemma means that if we can prove that $\exists(\mathsf{model}(s))$ obeys some policy, then we know that $\mathsf{autom}(s)$ obeys that policy.

Now we must more formally define our approach to producing a policy from $\mathsf{model}(s)$.

Let $q \overset{a:\alpha}{\rightarrowtail\!\!\rightarrowtail} q'$ iff

- there exists b such that either $q \overset{a}{\underset{b}{\rightarrowtail}} q''$ or $q \overset{\tau}{\underset{b}{\rightarrowtail}} q''$, and
- $q'' \overset{\alpha}{\rightarrowtail} q'$.

where $q \overset{[]}{\rightarrowtail\!\!\rightarrowtail} q'$ only if $q = q'$.

Let $q \overset{a:\alpha}{\underset{b}{\rightarrowtail\!\!\rightarrowtail}} q'$ iff

- $q \overset{a}{\underset{b}{\rightarrowtail}} q''$ or $q \overset{\tau}{\underset{b}{\rightarrowtail}} q''$, and
- $q'' \overset{\alpha}{\underset{b}{\rightarrowtail}} q'$.

where $q \overset{[]}{\underset{b}{\rightarrowtail\!\!\rightarrowtail}} q'$ only if $q = q'$.

Let $\mathsf{statePolicy}(s)$ be the state-based dynamic policy $\rightsquigarrow$ where $d_\mathsf{f} \rightsquigarrow^q d_\mathsf{t}$ iff there exists $\alpha_1, \alpha_2 \in A^*$ and $a \in A$ such that $q_0 \overset{\alpha_1}{\rightarrowtail\!\!\rightarrowtail} q' \overset{[a]}{\underset{\mathsf{F}}{\rightarrowtail\!\!\rightarrowtail}} q'' \overset{\alpha_2}{\rightarrowtail\!\!\rightarrowtail} q$. This means that $d_\mathsf{f} \rightsquigarrow^q d_\mathsf{t}$ for any state q such that it is reachable from a transition that produces the boolean F and that transition is reachable.

$\mathsf{policy}(s)$ is $\mathsf{statePolicy}(s)$ lifted from working on states to input sequences. Let $\mathsf{policy}(s)$ be the input-based dynamic policy $\leadsto$ where $d_\mathsf{f} \leadsto^\iota d_\mathsf{t}$ iff there exists $\alpha_1, \alpha_2 \in A^*$, and $a \in A$, and $q \in Q$ such that $q_0 \overset{\alpha_1}{\rightarrowtail\!\!\rightarrowtail} q'$, $q' \overset{[a]}{\underset{\mathsf{F}}{\rightarrowtail\!\!\rightarrowtail}} q''$, $q'' \overset{\alpha_2}{\rightarrowtail\!\!\rightarrowtail} q$, and $\iota = \lfloor\alpha{:}a\rfloor_I$.

Lemma 23. *For all programs s, $\mathsf{statePolicy}(s)$ is a non-revoking safe approximation of $\mathsf{policy}(s)$ for $\exists(\mathsf{model}(s))$.*

Proof. Let $\mathsf{policy}(s)$ be $\leadsto$ and $\mathsf{statePolicy}(s)$ be $\rightsquigarrow$. $d_\mathsf{f} \leadsto^\iota d_\mathsf{t}$ iff there exists $\alpha \in A^*$, and $a \in A$, and $q \in Q$ such that $q_0 \overset{\alpha}{\underset{\mathsf{F}}{\rightarrowtail\!\!\rightarrowtail}} q'$, $q' \overset{[a]}{\underset{\mathsf{T}}{\rightarrowtail\!\!\rightarrowtail}} q$, and $\iota = \lfloor\alpha{:}a\rfloor_I$. Thus, if $d_\mathsf{f} \not\leadsto^\iota d_\mathsf{t}$, then there does not $\alpha \in A^*$, $a \in A$, and $q \in Q$ such that $q_0 \overset{\alpha}{\underset{\mathsf{F}}{\rightarrowtail\!\!\rightarrowtail}} q'$, $q' \overset{[a]}{\underset{\mathsf{T}}{\rightarrowtail\!\!\rightarrowtail}} q$, and $\iota = \lfloor\alpha{:}a\rfloor_I$. If $q_0 \longrightarrow q$ in $\exists(\mathsf{model}(s))$, then it must not be the case that $q_0 \overset{\alpha}{\underset{\mathsf{F}}{\rightarrowtail\!\!\rightarrowtail}} q'$ and $q' \overset{[a]}{\underset{\mathsf{T}}{\rightarrowtail\!\!\rightarrowtail}} q$. Thus, $d_\mathsf{f} \not\rightsquigarrow^q d_\mathsf{t}$.

It is non-revoking because of how any states reachable from a state q where $d_\mathsf{f} \rightsquigarrow d_\mathsf{t}$ has been added to $\mathsf{statePolicy}(\mathsf{model}(s))$ also has $d_\mathsf{f} \rightsquigarrow d_\mathsf{t}$ added. □

Before proving Theorem 4, we must prove that for all programs s, $\exists(\mathsf{model}(s))$ obeys the DIINI policy $\mathsf{policy}(s)$. Since the above lemma tells us that $\mathsf{statePolicy}(s)$ is a non-revoking safe approximation of $\mathsf{policy}(s)$ for $\exists(\mathsf{model}(s))$, we may use the unwinding conditions to prove this. First, we explain the unwinding relation we will demonstrate, and then we prove that it indeed satisfies each of the unwinding conditions.

Recall that we have limited our construction to extracting the policy for when d_f flows d_t. Thus, $\mathsf{statePolicy}(s)$ has $d_1 \rightsquigarrow^q d_2$ for all q when $d_1 \neq d_\mathsf{f}$ or $d_2 \neq d_\mathsf{t}$. Thus, the unwinding conditions places no requirements on such d_1 and d_2. That is, $q_1 \overset{d_1}{\underset{d_2}{\sim}} q_2$ must only be defined for the case where $d_1 = d_\mathsf{t}$ and $d_2 = d_\mathsf{f}$ for our unwinding condition. Thus, to streamline notation, we usually drop the domains and just write $q_1 \sim q_2$.

Given two stores Γ_1 and Γ_2, let $\Gamma_1 \equiv^\eta \Gamma_2$ iff for all $x \in X$ such that $\eta(x) = \mathsf{T}$, $\Gamma_1(x) = \Gamma_2(x)$. Let the dynamic view partition $\sim$ be such that $\langle \Gamma_1, \ell_1, \eta_1 \rangle \sim \langle \Gamma_2, \ell_2, \eta_2 \rangle$ iff $\ell_1 = \ell_2$, $\eta_1 = \eta_2$, and $\Gamma_1 \equiv^{\eta_1} \Gamma_2$. We will show that $\sim$ is an unwinding relation.

Lemma 24. *$\sim$ has dynamic local respect for $\exists(\mathsf{model}(s))$ and $\mathsf{statePolicy}(\mathsf{model}(s))$.*

Proof. Since $\exists(\mathsf{model}(s))$ is constructed from $\mathsf{model}(s)$, the only transitions in $\exists(\mathsf{model}(s))$ of the form $q \overset{i_1}{\to} q_1$ come from a transition in $\mathsf{model}(s)$ of the form $q \overset{i_1}{\underset{b}{\rightarrowtail}} q_1$ for $b = \mathsf{T}$ or $b = \mathsf{F}$. Since the transitions of $\mathsf{model}(s)$ come from $>s>$, we may examine the definition of $>\cdot>$ to find when transitions of the form $q \overset{i_1}{\underset{b}{\rightarrowtail}} q_1$ are possible. These are only possible when there exists a statement s' that is a sub-statement of s (or equal to s) such that s' has form $\texttt{read(}x\texttt{, }d\texttt{)}$. Furthermore, the state q must have the form $\langle \Gamma, \mathsf{pre}(s'), \eta \rangle$

By requiring $\mathsf{dom}(i_1)$ to be d_f, we further limit of the form of s' to $\texttt{read(}x\texttt{, }d_\mathsf{f}\texttt{)}$ and the form of i_1 to $\langle \mathsf{i}, d_\mathsf{f}, n_1$ for some n_1. Also the boolean b must be T. This implies that q_1 has the form $\langle \Gamma[x \mapsto n_1], \mathsf{post}(s'), \eta[x \mapsto \mathsf{F}] \rangle$. Thus, if $q \xrightarrow[d_\mathsf{t}]{i_1} q'$ in $\exists(\mathsf{model}(s))$, it is because $q \overset{i_1}{\underset{\mathsf{T}}{\rightarrowtail}} q_1$ in $\mathsf{model}(s)$ where i_1 and q_1 are of the above form.

For another input i_2 to be such that $\mathsf{dom}(i_2) = d_\mathsf{f}$, it must have the form $\langle \mathsf{i}, d_\mathsf{f}, n_2 \rangle$ for some n_2. Let $q_2 = \langle \Gamma[x \mapsto n_2], \mathsf{post}(s), \eta[x \mapsto \mathsf{F}] \rangle$. By the construction of $>s>$, $q \overset{i_2}{\underset{\mathsf{T}}{\rightarrowtail}} q_2$. Thus, $q \xrightarrow[d_\mathsf{t}]{i_2} q_2$ in $\exists(\mathsf{model}(s))$.

Since $\eta[x \mapsto \mathsf{F}](x) = \mathsf{F}$, and $\Gamma[x \mapsto n_1]$ and $\Gamma[x \mapsto n_2]$ agree on all other variables, $\Gamma[x \mapsto n_1] \equiv^\eta \Gamma[x \mapsto n_2]$. Thus, $q_1 \sim q_2$. □

To prove step consistency we must strengthen the hypothesis and introduce some additional concepts.

$\rightsquigarrow_1 \trianglelefteq \rightsquigarrow_2$ if whenever $d_\mathsf{f} \rightsquigarrow_2^q d_\mathsf{t}$, $d_\mathsf{f} \rightsquigarrow_1^q d_\mathsf{t}$. Note that $\rightsquigarrow_1$ may be defined for more states than $\rightsquigarrow_2$. $\rightsquigarrow_1 \trianglelefteq \rightsquigarrow_2$ implies that if $\rightsquigarrow_2$ is defined at q and $d_\mathsf{f} \not\rightsquigarrow_1^q d_\mathsf{t}$, then $d_\mathsf{f} \not\rightsquigarrow_2^q d_\mathsf{t}$.

Let $q_1 \xrightarrow[d]{} q_2$ iff

- $q_1 = q_2$,
- $q_1 \overset{\tau}{\to} q'$ and $q' \xrightarrow[d]{} q_2$, or
- $q_1 \overset{o}{\to} q'$ and $q' \xrightarrow[d]{} q_2$ where $\mathsf{dom}(o) \neq d$.

First we prove a key lemma to Step Consistency and Output Consistency. One can views step Consistency as requiring two states that are related by $\sim$ will transition to other related states. Likewise, Output Consistency requires that two related states transition to two other related states after producing an output. Under this view, the next lemma requires that two related states transition to two other related states upon finishing the execution of a statement.

Lemma 25. *For all statements s and s' where $\exists(\mathsf{model}(s)) = \langle I, O, D, \mathsf{dom}, Q, q_0, \to \rangle$ and s' is a sub-statement of s, state-based dynamic policies $\rightsquigarrow$, stores Γ'_1, independence predicates η', $q_1, q'_1, q_2 \in Q$, if $\rightsquigarrow \; \trianglelefteq \mathsf{statePolicy}(\mathsf{model}(s))$, $q_1 \sim q_2$, $q_1 \xrightarrow[d_\mathsf{t}]{} q'_1$, $q'_1 = \langle \Gamma'_1, \mathsf{post}(s'), \eta' \rangle$, and $d_\mathsf{f} \not\rightsquigarrow^{q'_1} d_\mathsf{t}$, then there must exist $q'_2 \in Q$ such that $q_2 \xrightarrow[d_\mathsf{t}]{} q'_2$ and $q'_1 \overset{d_\mathsf{t}}{\underset{d_\mathsf{f}}{\sim}} q'_2$.*

Proof. Since $q_1 \sim q_2$ we know that q_1 has the form $\langle \Gamma_1, \ell, \eta \rangle$ and q_2 the form $\langle \Gamma_2, \ell, \eta \rangle$ where $\Gamma_1 \equiv^\eta \Gamma_2$.

Proof by induction over the derivation of $>s>$.

Case: s has the form $x{:=}e$. Since $q_1 \xrightarrow[d_\mathsf{t}]{} q'_1$, ℓ must be $\mathsf{pre}(s)$. Furthermore, Γ'_1 must be $\Gamma_1[x \mapsto \Gamma_1(e)]$ and η' must be $\eta[x \mapsto \eta(e)]$.

Let $q'_2 = \langle \Gamma_2[x \mapsto \Gamma_2(e)], \mathsf{post}(s), \eta[x \mapsto \eta(e)] \rangle$. $q_2 \overset{\tau}{\to} q'_2$ by the construction of $\mathsf{model}(s)$. If $\eta(e) = \mathsf{T}$, then $\Gamma_1(e) = \Gamma_2(e)$ since $\Gamma_1 \equiv^\eta \Gamma_2$. If $\eta(e) = \mathsf{F}$, then $\eta[x \mapsto \eta(e)](x) = \mathsf{F}$ Either way, $\Gamma_1[x \mapsto \Gamma_1(e)] \equiv^{\eta[x \mapsto \eta(e)]} \Gamma_2[x \mapsto \Gamma_2(e)]$. Thus, $q'_1 \sim q'_2$.

Case: s has the form $\texttt{read(}x\texttt{, }d\texttt{)}$ with $d_\mathsf{f} \neq d \neq d_\mathsf{t}$. In this case, q_1 cannot make a transition without consuming input. Thus, this case is trivially satisfied.

Case: s has the form $\texttt{read(}x\texttt{, }d\texttt{)}$ with $d = d_\mathsf{f}$ or $d = d_\mathsf{t}$. Ditto.

Case: s has the form $\texttt{print(}e\texttt{, }d\texttt{)}$ with $d \neq d_\mathsf{t}$. Since $q_1 \xrightarrow[d_\mathsf{t}]{} q'_1$, ℓ must be $\mathsf{pre}(s)$. Furthermore, $\Gamma'_1 = \Gamma_1$ and $\eta' = \eta$. Let $q'_2 = \langle \Gamma_2, \mathsf{post}(s), \eta \rangle$. $q_2 \overset{\tau}{\to} q'_2$ by the construction of $\mathsf{model}(s)$ and $q'_1 \sim q'_2$.

Case: s has the form $\texttt{print(}e\texttt{, }d_\mathsf{t}\texttt{)}$. In this case q_1 cannot make a transition without producing output to d_t. Thus, the statement is trivially satisfied.

Case: s has the form $s_\mathsf{a}\texttt{; }s_\mathsf{b}$. Since L_s is the disjoint union of L_{s_a}, L_{s_b}, and $\{\mathsf{pre}(s), \mathsf{post}(s)\}$, one of the following cases must hold:

- q_1 and q_1' are both in Q_{a}. $\mathsf{statePolicy}(\mathsf{model}(s)) \trianglelefteq \mathsf{statePolicy}(\mathsf{model}(s_{\mathsf{a}}))$ since $\mathsf{model}(s)$ has at least as many transitions under a true boolean as $\mathsf{model}(s_{\mathsf{a}})$. The needed result follows from the inductive hypothesis.

- q_1 and q_1' are both in Q_{b}. $\mathsf{statePolicy}(\mathsf{model}(s)) \trianglelefteq \mathsf{statePolicy}(\mathsf{model}(s_{\mathsf{b}}))$ since $\mathsf{model}(s)$ has at least as many transitions under a true boolean as $\mathsf{model}(s_{\mathsf{b}})$. The needed result follows from the inductive hypothesis.

- $q_1 \in Q_{\mathsf{a}}$ and $q_1' \in Q_{\mathsf{b}}$. Since $q_1 \xrightarrow[d]{} q_2$ and the construction of $\mathsf{model}(s)$, there must exist a state $q_{1\mathsf{a}}$ of the form $\langle \Gamma_{1\mathsf{a}}, \mathsf{post}(s_{\mathsf{a}}), \eta_{1\mathsf{a}} \rangle$ and a state $q_{1\mathsf{b}}$ of the form $\langle \Gamma_{1\mathsf{a}}, \mathsf{pre}(s_{\mathsf{b}}), \eta_{1\mathsf{a}} \rangle$ such that $q_1 \xrightarrow[d_{\mathsf{t}}]{} q_{1\mathsf{a}} \overset{\tau}{\to} q_{1\mathsf{b}} \xrightarrow[d_{\mathsf{t}}]{} q_1'$.

  By the inductive hypothesis, this means there exists a state $q_{2\mathsf{a}}$ such that $q_2 \xrightarrow[d_{\mathsf{t}}]{} q_{2\mathsf{a}}$ and $q_{1\mathsf{a}} \sim q_{2\mathsf{a}}$. This implies that the form of $q_{2\mathsf{a}}$ is $\langle \Gamma_{2\mathsf{a}}, \mathsf{post}(s_{\mathsf{a}}), \eta_{1\mathsf{a}} \rangle$. Thus, by the construction of $\mathsf{model}(s)$, $q_{2\mathsf{a}} \overset{\tau}{\to} q_{2\mathsf{b}}$ where $q_{2\mathsf{b}} = \langle \Gamma_{2\mathsf{a}}, \mathsf{pre}(s_{\mathsf{b}}), \eta_{1\mathsf{a}} \rangle$.

  Since $q_{1\mathsf{b}} \sim q_{2\mathsf{b}}$, the inductive hypothesis again applies and there must exist q_2' such that $q_{2\mathsf{b}} \xrightarrow[d_{\mathsf{t}}]{} q_2'$ and $q_1' \sim q_2'$.

- $q_1 \in Q_{\mathsf{b}}$ and $q_1' \in Q_{\mathsf{a}}$. Since $q_1 \xrightarrow[d_{\mathsf{t}}]{} q_1'$ cannot hold in this case, we need not consider it.

- q_1 has the form $\langle \Gamma_1, \mathsf{pre}(s), \eta \rangle$ and q_1' is in Q_{a} or Q_{b}. Since $q_1 \sim q_2$, q_2 must have the form $\langle \Gamma_2, \mathsf{pre}(s), \eta \rangle$ where $\Gamma_1 \equiv^{\eta} \Gamma_2$. By the construction of $\exists(\mathsf{model}(s))$, $q_1 \overset{\tau}{\to} q_{1\mathsf{a}}$ where $q_{1\mathsf{a}} = \langle \Gamma_1, \mathsf{pre}(s_{\mathsf{a}}), \eta \rangle$ and $q_2 \overset{\tau}{\to} q_{2\mathsf{a}}$ where $q_{2\mathsf{a}} = \langle \Gamma_2, \mathsf{pre}(s_{\mathsf{a}}), \eta \rangle$. Since $q_{1\mathsf{a}} \sim q_{2\mathsf{a}}$ and they are both in $\exists(\mathsf{model}(s_{\mathsf{a}}))$, the proof continues as above.

- q_1 is in Q_{a} or Q_{b} and q_1' has the form $\langle \Gamma_1', \mathsf{post}(s), \eta' \rangle$. If $q_1 \xrightarrow[d_{\mathsf{t}}]{} q_1'$, then $q_1 \xrightarrow[d_{\mathsf{t}}]{} q_{1\mathsf{b}}$ where $q_{1\mathsf{b}} = \langle \Gamma_1', \mathsf{post}(s_{\mathsf{b}}), \eta' \rangle$. Thus, as argued above, there exists $q_{2\mathsf{b}} = \langle \Gamma_2', \mathsf{post}(s_{\mathsf{b}}), \eta' \rangle$ such that $q_{1\mathsf{b}} \sim q_{2\mathsf{b}}$. By the construction of $\mathsf{model}(s)$, $q_{1\mathsf{b}} \overset{\tau}{\to} q_1'$ and $q_{2\mathsf{b}} \overset{\tau}{\to} q_2'$ where $q_2' = \langle \Gamma_2', \mathsf{post}(s), \eta' \rangle$. $q_1' \sim q_2'$.

- q_1 has the form $\langle \Gamma_1, \mathsf{pre}(s), \eta \rangle$ and q_1' has the form $\langle \Gamma_1', \mathsf{post}(s), \eta' \rangle$. In this case, just use the arguments found in the two cases above.

Case: s has the form `if(`e`)`s_{a}`else`s_{b}. If $\eta(e) = \mathsf{T}$, then $\Gamma_1(e) = \Gamma_2(e)$ since $\Gamma_1 \equiv^{\eta} \Gamma_2$. In this case, the result follows from using the inductive hypothesis on s_{a} if $\Gamma_1(e) \neq 0$ and on s_{b} if $\Gamma_1(e) = 0$ and the methods used above for dealing with the cases where q_1 has the form $\langle \Gamma_1, \mathsf{pre}(s), \eta \rangle$ or q_1' has the form $\langle \Gamma_1', \mathsf{post}(s), \eta' \rangle$.

The same holds even if $\eta(e) = \mathsf{F}$ as long and $\Gamma_1(e) = \Gamma_2(e)$.

If $d_{\mathsf{f}} \rightsquigarrow^{q_1'} d_{\mathsf{t}}$, then we need not prove anything since it violates a premise of the lemma. Note that if $\eta(e) = \mathsf{F}$ and either s_{a} or s_{b} contained a `while` loop, a `read` statement, or a statement of the form `print(`e`,` d_{t}`)`, then $\mathsf{statePolicy}(\mathsf{model}(s))$ would allow information to flow from d_{f} to d_{t} at q_1. Since $\mathsf{statePolicy}(\mathsf{model}(s))$ is non-revoking and $q_1 \xrightarrow[d]{} q_1'$, the same would be true at q_1'. Since $\rightsquigarrow\ \trianglelefteq \mathsf{statePolicy}(\mathsf{model}(s))$, $d_{\mathsf{f}} \rightsquigarrow^{q_1'} d_{\mathsf{t}}$ would be true. Thus, we have dealt with these cases.

This leaves the case where $\eta(e) = \mathsf{F}$, $\Gamma_1(e) \neq \Gamma_2(e)$, and neither s_{a} nor s_{b} contains a `while` loop, a `read` statement, or a statement of the form `print(`e`,` d_{t}`)`. Since there are no `read` statements, all the transitions in s_{a} and s_{b} are transitions that the automaton has control over and there is no chance of a transition being blocked by a user not offering input. Since there are no `while` loops, once s_{a} or s_{b} is entered, they will surely be exited. This means that there must exist q_2' such that $q_2 \xrightarrow[d_{\mathsf{t}}]{} q_2'$ and $q_2' = \langle \Gamma_2', \mathsf{post}(s), \eta' \rangle$ for some Γ_2' and η'. Since $\Gamma_1 \equiv^{d} \Gamma_2$ and η' assigns F to any variable altered in either s_{a} or s_{b}, $\Gamma_1' \equiv^{\eta'} \Gamma_2'$. Thus, $q_1' \sim q_2'$.

Case: s has the form `while(`e`)`s_{a} with $\Gamma_1(e) \neq 0$. If $\eta(e) = \mathsf{T}$, then $\Gamma_1(e) = \Gamma_2(e)$ and the inductive hypothesis may be applied to s_{a}. If $\eta(e) = \mathsf{F}$, then $\mathsf{statePolicy}(\mathsf{model}(s))$ would allow information to flow from d_{f} to d_{t} at q_1. As above, this implies that $d_{\mathsf{f}} \rightsquigarrow^{q_1'} d_{\mathsf{t}}$ and thus the result is trivially true.

Case: s has the form `while(`e`)`s_{a} with $\Gamma_1(e) = 0$. The case is proved as the previous one was.

□

Now we prove a result slightly stronger than Step Consistency.

Lemma 26. *For all statements s where $\exists(\mathsf{model}(s)) = \langle I, O, D, \mathsf{dom}, Q, q_0, \to \rangle$, state-based dynamic policies $\rightsquigarrow$, $q_1, q_1', q_2 \in Q$, and $i \in I$, if $\rightsquigarrow\ \trianglelefteq \mathsf{statePolicy}(\mathsf{model}(s))$, $q_1 \sim q_2$, $q_1 \xrightarrow[d_{\mathsf{t}}]{i} q_1'$, and $d_{\mathsf{f}} \not\rightsquigarrow^{q_1'} d_{\mathsf{t}}$, then there must exist $q_2' \in Q$ such that $q_2 \xrightarrow[d_{\mathsf{t}}]{i} q_2'$ and $q_1' \overset{d_{\mathsf{t}}}{\underset{d_{\mathsf{f}}}{\sim}} q_2'$.*

Proof. Since $q_1 \sim q_2$ we know that q_1 has the form $\langle \Gamma_1, \ell, \eta \rangle$ and $\langle \Gamma_2, \ell, \eta \rangle$ where $\Gamma_1 \equiv^{\eta} \Gamma_2$. Let $q_1' = \langle \Gamma_1', \ell', \eta' \rangle$.

Proof by induction over the derivation of $>s>$.

Case: s has the form x:=e. In this case q_1 does not transition to any other state under an input i in $\mathsf{model}(s)$. Thus, the statement is trivially satisfied.

Case: s has the form `read(`x`,` d`)` with $d_{\mathsf{f}} \neq d \neq d_{\mathsf{t}}$. In this case, for q_1 to transition to q_1', ℓ must be $\mathsf{pre}(s)$ and ℓ' must be $\mathsf{post}(s)$. The input i must be of the form $\langle \mathsf{i}, d, n \rangle$.

Furthermore, $\Gamma'_1 = \Gamma_1[x \mapsto n]$ and $\eta' = \eta[x \mapsto \mathsf{T}]$. This means that q_2 has the form $\langle \Gamma_2, \mathsf{pre}(s), \eta \rangle$.

Let $q'_2 = \langle \Gamma_2[x \mapsto n], \mathsf{post}(s), \eta[x \mapsto \mathsf{T}] \rangle$. Since $\Gamma_1 \equiv^\eta \Gamma_2$, $\Gamma_1[x \mapsto n] \equiv^{\eta[x \mapsto \mathsf{T}]} \Gamma_2[x \mapsto n]$. Thus, $q'_1 \sim q'_2$ and $q_2 \xrightarrow[d_\mathsf{t}]{i} q'_2$ in $\exists(\mathsf{model}(s))$.

Case: s has the form $\texttt{read(}x\texttt{, }d\texttt{)}$ with $d = d_\mathsf{f}$ or $d = d_\mathsf{t}$. In this case, for q_1 to transition to q'_1, ℓ must be $\mathsf{pre}(s)$ and ℓ' must be $\mathsf{post}(s)$. The input i must be of the form $\langle \mathsf{i}, d, n \rangle$. Furthermore, $\Gamma'_1 = \Gamma_1[x \mapsto n]$ and $\eta' = \eta[x \mapsto \mathsf{F}]$. This means that q_2 has the form $\langle \Gamma_2, \mathsf{pre}(s), \eta \rangle$.

Let $q'_2 = \langle \Gamma_2[x \mapsto n], \mathsf{post}(s), \eta[x \mapsto \mathsf{F}] \rangle$. Since $\Gamma_1 \equiv^\eta \Gamma_2$, $\Gamma_1[x \mapsto n] \equiv^\eta \Gamma_2[x \mapsto n]$. Thus, $q'_1 \sim q'_2$ and $q_2 \xrightarrow[d_\mathsf{t}]{i} q'_2$ in $\exists(\mathsf{model}(s))$.

Case: s has the form $\texttt{print(}e\texttt{, }d\texttt{)}$ with $d \neq d_\mathsf{t}$. In this case q_1 does not transition to any other state under an input i in $\mathsf{model}(s)$. Thus, the statement is trivially satisfied.

Case: s has the form $\texttt{print(}e\texttt{, }d_\mathsf{t}\texttt{)}$. Ditto.

Case: s has the form $s_\mathsf{a}\texttt{; }s_\mathsf{b}$.

Since L_s is the disjoint union of L_{s_a}, L_{s_b}, and $\{\mathsf{pre}(s), \mathsf{post}(s)\}$, one of the following cases must hold:

- q_1 and q'_1 are both in Q_a. $\mathsf{statePolicy}(\mathsf{model}(s)) \trianglelefteq \mathsf{statePolicy}(\mathsf{model}(s_\mathsf{a}))$ since $\mathsf{model}(s)$ has at least as many transitions under a true boolean as $\mathsf{model}(s_\mathsf{a})$. The needed result follows from the inductive hypothesis.

- q_1 and q'_1 are both in Q_b. $\mathsf{statePolicy}(\mathsf{model}(s)) \trianglelefteq \mathsf{statePolicy}(\mathsf{model}(s_\mathsf{b}))$ since $\mathsf{model}(s)$ has at least as many transitions under a true boolean as $\mathsf{model}(s_\mathsf{b})$. . The needed result follows from the inductive hypothesis.

- $q_1 \in Q_\mathsf{a}$ and $q'_1 \in Q_\mathsf{b}$. Since $q_1 \xrightarrow[d]{i} q_2$ and the construction of $\mathsf{model}(s)$, there must exist a state $q_{1\mathsf{a}}$ of the form $\langle \Gamma_{1\mathsf{a}}, \mathsf{post}(s_\mathsf{a}), \eta_{1\mathsf{a}} \rangle$ and a state $q_{1\mathsf{b}}$ of the form $\langle \Gamma_{1\mathsf{a}}, \mathsf{pre}(s_\mathsf{b}), \eta_{1\mathsf{a}} \rangle$ such that $q_1 \xrightarrow[d_\mathsf{t}]{} q_{1\mathsf{a}} \xrightarrow{\tau} q_{1\mathsf{b}} \xrightarrow[d_\mathsf{t}]{i} q'_1$.

  By Lemma 25, this means there exists a state $q_{2\mathsf{a}}$ such that $q_2 \xrightarrow[d_\mathsf{t}]{} q_{2\mathsf{a}}$ and $q_{1\mathsf{a}} \sim q_{2\mathsf{a}}$. This implies that the form of $q_{2\mathsf{a}}$ is $\langle \Gamma_{2\mathsf{a}}, \mathsf{post}(s_\mathsf{a}), \eta_{1\mathsf{a}} \rangle$. Thus, by the construction of $\mathsf{model}(s)$, $q_{2\mathsf{a}} \xrightarrow{\tau} q_{2\mathsf{b}}$ where $q_{2\mathsf{b}} = \langle \Gamma_{2\mathsf{a}}, \mathsf{pre}(s_\mathsf{b}), \eta_{1\mathsf{a}} \rangle$.

  Since $q_{1\mathsf{b}} \sim q_{2\mathsf{b}}$, the inductive hypothesis on s_b applies as above and there must exist q'_2 such that $q_{2\mathsf{b}} \xrightarrow[d_\mathsf{t}]{i} q'_2$ and $q'_1 \sim q'_2$.

- $q_1 \in Q_\mathsf{b}$ and $q'_1 \in Q_\mathsf{a}$. Since $q_1 \xrightarrow[d_\mathsf{t}]{i} q'_1$ cannot hold in this case, we need not consider it.

- q_1 has the form $\langle \Gamma_1, \mathsf{pre}(s), \eta \rangle$ and q'_1 is in Q_a or Q_b. Since $q_1 \sim q_2$, q_2 must have the form $\langle \Gamma_2, \mathsf{pre}(s), \eta \rangle$ where $\Gamma_1 \equiv^\eta \Gamma_2$. By the construction of $\exists\mathsf{model}(s)$, $q_1 \xrightarrow{\tau} q_{1\mathsf{a}}$ where $q_{1\mathsf{a}} = \langle \Gamma_1, \mathsf{pre}(s_\mathsf{a}), \eta \rangle$ and $q_2 \xrightarrow{\tau} q_{2\mathsf{a}}$ where $q_{2\mathsf{a}} = \langle \Gamma_2, \mathsf{pre}(s_\mathsf{a}), \eta \rangle$. Since $q_{1\mathsf{a}} \sim q_{2\mathsf{a}}$ and they are both in $\exists\mathsf{model}(s_\mathsf{a})$, the proof continues as above.

- q'_1 has the form $\langle \Gamma'_1, \mathsf{post}(s), \eta' \rangle$. $q_1 \xrightarrow[d_\mathsf{t}]{i} q'_1$ is impossible in this case, so we need not consider it.

Case: s has the form $\texttt{if(}e\texttt{)}s_\mathsf{a}\texttt{else}s_\mathsf{b}$. If $\eta(e) = \mathsf{T}$, then $\Gamma_1(e) = \Gamma_2(e)$ since $\Gamma_1 \equiv^\eta \Gamma_2$. In this case, the result follows from using the inductive hypothesis on s_a if $\Gamma_1(e) \neq 0$ and on s_b if $\Gamma_1(e) = 0$ and the methods used above for dealing with the cases where q_1 has the form $\langle \Gamma_1, \mathsf{pre}(s), \eta \rangle$ or q'_1 has the form $\langle \Gamma'_1, \mathsf{post}(s), \eta' \rangle$.

The same holds even if $\eta(e) = \mathsf{F}$ as long and $\Gamma_1(e) = \Gamma_2(e)$.

As argued in Lemma 25, cases where $\eta(e) = \mathsf{F}$ and either s_a or s_b contains a $\texttt{while}$ loop, a $\texttt{read}$ statement, or a statement of the form $\texttt{print(}e\texttt{, }d_\mathsf{t}\texttt{)}$ are handled by the construction of $\mathsf{statePolicy}(\mathsf{model}(s))$. However, if no $\texttt{read}$ statements are in s_a or s_b, then clearly $q_1 \xrightarrow[d_\mathsf{t}]{i} q'_1$ cannot hold. Thus, all cases have be covered.

Case: s has the form $\texttt{while(}e\texttt{)}s_\mathsf{a}$ with $\Gamma_1(e) \neq 0$. If $\eta(e) = \mathsf{T}$, then $\Gamma_1(e) = \Gamma_2(e)$ and the inductive hypothesis may be applied to s_a. If $\eta(e) = \mathsf{F}$, then $\mathsf{statePolicy}(\mathsf{model}(s))$ would allow information to flow from d_f to d_t at q_1. As above, this implies that $d_\mathsf{f} \rightsquigarrow^{q'_1} d_\mathsf{t}$ and thus the result is trivially true.

Case: s has the form $\texttt{while(}e\texttt{)}s_\mathsf{a}$ with $\Gamma_1(e) = 0$. Ditto. □

Now to prove a statement slightly stronger than Output Consistency.

Lemma 27. *For all statements s where $\exists(\mathsf{model}(s)) = \langle I, O, D, \mathsf{dom}, Q, q_0, \rightarrow \rangle$, state-based dynamic policies $\rightsquigarrow$, $q_1, q'_1, q_2 \in Q$, and $o \in O$, if $\mathsf{dom}(o) = d_\mathsf{t}$, $\rightsquigarrow \trianglelefteq \mathsf{statePolicy}(\mathsf{model}(s))$, $q_1 \sim q_2$, $q_1 \xrightarrow[d_\mathsf{t}]{o} q'_1$, and $d_\mathsf{f} \not\rightsquigarrow^{q'_1} d_\mathsf{t}$, then there must exist $q'_2 \in Q$ such that $q_2 \xrightarrow[d_\mathsf{t}]{o} q'_2$ and $q'_1 \overset{d_\mathsf{t}}{\underset{d_\mathsf{f}}{\sim}} q'_2$.*

Proof. Since $q_1 \sim q_2$ we know that q_1 has the form $\langle \Gamma_1, \ell, \eta \rangle$ and $\langle \Gamma_2, \ell, \eta \rangle$ where $\Gamma_1 \equiv^\eta \Gamma_2$. Let $q'_1 = \langle \Gamma'_1, \ell', \eta' \rangle$.

Proof by induction over the derivation of $>s>$.

Case: s has the form $x{:=}e$. In this case q_1 does not transition to any other state under an input o in $\mathsf{model}(s)$. Thus, the statement is trivially satisfied.

Case: s has the form $\texttt{read(}x\texttt{, }d\texttt{)}$ with $d_\mathsf{f} \neq d \neq d_\mathsf{t}$. Ditto.

Case: s has the form $\texttt{read(}x\texttt{, }d\texttt{)}$ with $d = d_\mathsf{f}$ or $d = d_\mathsf{t}$. Ditto.

Case: s has the form `print(`e`, `d`)` with $d \neq d_{\mathsf{t}}$. In this case, q_1 will only transition to another state under an output o such that $\mathsf{dom}(o) \neq d_{\mathsf{t}}$. Thus, the statement is trivially satisfied.

Case: s has the form `print(`e`, `d_{t}`)`. $q_1 \overset{o}{\to} q_1'$ if $o = \langle \mathsf{o}, d_{\mathsf{t}}, \Gamma_1(e)\rangle$, $\Gamma_1' = \Gamma_1$, $\ell = \mathsf{pre}(s)$, $\ell' = \mathsf{post}(s)$, and $\eta' = \eta$.

If $\eta(e) = \mathsf{T}$, $\Gamma_1(e) = \Gamma_2(e)$ since $\Gamma_1 \equiv^{\eta} \Gamma_2$. Let $q_2' = \langle \Gamma_2, \mathsf{post}(s), \eta\rangle$. $q_2 \overset{o}{\to} q_2'$ and $q_1' \sim q_2'$.

If $\eta(e) = \mathsf{F}$, then $\mathsf{statePolicy}(\mathsf{model}(s))$ would allow d_{t} access to d_{f} at q_1'. Since $\rightsquigarrow \trianglelefteq \mathsf{statePolicy}(\mathsf{model}(s))$, $d_{\mathsf{f}} \rightsquigarrow^{q_1} d_{\mathsf{t}}$. Thus, the result is satisfied trivially.

The remaining cases are as in Lemma 26 just replacing $\xrightarrow[d_{\mathsf{t}}]{i}$ with $\xrightarrow[d_{\mathsf{t}}]{o}$. □

Theorem 4 can now be proved:

Proof. Lemmas 24, 26, and 27 show that $\sim$ is an unwinding relation for $\mathsf{statePolicy}(\mathsf{model}(s))$ and $\exists(\mathsf{model}(s))$. Since by Lemma 23, $\mathsf{statePolicy}(\mathsf{model}(s))$ is a non-revoking safe approximation of $\mathsf{policy}(\mathsf{model}(s))$ this means that $\exists(\mathsf{model}(s))$ obeys $\mathsf{policy}(\mathsf{model}(s))$. By Lemma 22, this means that $\mathsf{autom}(s)$ obeys $\mathsf{policy}(\mathsf{model}(s))$. □