An Epistemic Formulation of Information Flow Analysis

Arbob Ahmad
Computer Science Dept.
Carnegie Mellon University
Pittsburgh, USA
adahmad@cs.cmu.edu

Robert Harper
Computer Science Dept.
Carnegie Mellon University
Pittsburgh, USA
rwh@cs.cmu.edu

***Abstract*—Most accounts of information flow security in programming languages emphasize non-interference to characterize security: in a secure program, changes to high-security inputs do not alter the values of low-security outputs. The definition of non-interference is incompatible with declassification, which allows some low-security outputs to be influenced by high-security inputs. We propose an alternative account of information flow based on an epistemic logic of computational effects. Rather than view a program as a function from inputs to outputs, we instead embrace the principle that information flow security is concerned with the effects a program has on its execution environment. These effects are modelled using a substructural epistemic logic that tracks the flow of knowledge gained by principals and communication channels during execution. Confidentiality is expressed by proving necessary conditions for a principal to know a sensitive fact at the end of an execution. In the simplest case the necessary condition is falsehood, which means that a principal cannot know a secret as a result of a well-typed execution of a program. In the presence of declassification a necessary condition for disclosure is the existence of a proof of authorization in a formal authorization logic, expressing that sensitive data is disclosed only when explicitly authorized. Rather than taken as the primary result, the classical non-interference property arises in the proof of adequacy of the epistemic theory of disclosure, ensuring that it accurately models program behavior. It is suggested that an epistemic account of information flow security is both more natural and more expressive than classical accounts based only on non-interference.**

***Index Terms*—security; non-interference; authorized declassification; epistemic logic**

I. Introduction

Most accounts of information flow security treat a program as a function from inputs to outputs, and express security in terms of *non-interference* [1], which states that a change to "low security" inputs cannot influence "high security" outputs. Following [2], we instead stress that security properties are concerned with the *effects* that a computation has on its execution environment. These include, for example, messages sent on communication channels or modifications to storage visible to other parties in a computation. In this setting the security properties of a program are expressed directly as statements about the flow of *knowledge* among principals engendered by a *trace* of these effects. Confidentiality is expressed, roughly, by proving that no run of a well-typed program can result in a low-security principal knowing a high-security fact. One advantage of this approach is that it naturally accommodates declassification by integrating authorization logic [3] with epistemic logic [4]. Then one may say that in a well-typed execution a principal may learn a sensitive fact only if there is an authorization proof allowing that disclosure. The classical non-interference property arises in the proof of *adequacy* of the theory of knowledge flow in a program, validating the theory against the execution behavior of the program. Altogether we obtain a generalization of the usual accounts of information flow security that directly addresses the properties of interest (knowledge gained by principals during execution) and accommodates declassification (authorized disclosure of sensitive information).

The knowledge transfers that result from each effect are specified in the substructural epistemic logic of [5]. It includes an ephemeral form of knowledge, known as *possession*. Since mutable storage, such as assignable variables or channels, changes over time, it may possess knowledge of a fact at one moment in time, but lose that knowledge at some later point. In contrast *persistent* knowledge is retained permanently. The point of using a substructural epistemic logic is precisely that it models both ephemeral and persistent knowledge.

The very idea of confidentiality is that certain principals do not know certain information. In our setting an absence of knowledge is expressed by stating that a statement of knowledge is *not derivable* in the theory. This is to be contrasted with the statement that its *negation* is derivable in the theory. A negated knowledge statement would mean that the principal cannot know a given fact under any circumstances, including those that violate the most fundamental abstractions. Such statements are never provable, because we can never rule out the possibility of "side channels" (such as power consumption or other analog effects [6]) through which knowledge may be acquired. Instead, by showing that a knowledge statement is not derivable, we are merely stating that the given fact is not learnable within the limitations of the model. This formulation provides a crisp formulation of standard disclaimers about attacks "lying outside the model."

For example, if the assignable `a` is intended to be kept secret from the principal `k` then we would like to prove that any trace resulting from the execution of a well-typed program proves that `k` does not acquire the knowledge of `a`. Using

square brackets to indicate knowledge and $\mathtt{s(a)}$ to represent the secrets of the assignable $\mathtt{a}$, $\mathtt{k}$ not knowing $\mathtt{s(a)}$ is written as $\neg([\mathtt{k}]\mathtt{s(a)})$. But such a negation is never provable, for reasons just discussed. Instead, we show that $[\mathtt{k}]\mathtt{s(a)}$ is not derivable in the epistemic theory used to model program execution. Showing such a non-derivability relies on a deep result about epistemic logic, known as cut elimination, that permits us to analyze all possible derivations succinctly.

We still must connect the derivability of knowledge statements in an epistemic theory to the execution of a program, which is to say we must show that the theory is *computationally adequate*. For example, a theory that does not permit transfers of knowledge as a result of any action would yield trivial proofs that a given transfer is not derivable, but non-derivability in this theory does not correspond to the expected behavior of the program. Adequacy ensures that the execution of two runs of the same command will have a different result only if there is an assignable in the initial store that differs and that assignable is disclosed to the executing principal according to the epistemic logic. It is here that the standard notion of non-interference emerges in our development. The adequacy theorem does not prevent the model from conservatively assuming more flows of knowledge than are actually possible through the dynamic semantics, but a model that is too conservative would be useless for proving that undesirable flows are not derivable. We exhibit a particular theory that allows useful facts to be derived about the flow of knowledge in a well-typed program.

Declassification is permitted through an explicit syntactic construct that requires a proof of authorization for the declassification. This authorization proof is in the same epistemic logic that represents the knowledge transfers resulting from each effect. Therefore, when a knowledge transfer that uses declassification is derivable, the derivation also includes an authorization proof for the declassification. Doing so links up authorization logic, which merely ensures that certain protocols are obeyed, with epistemic logic, which relates those protocols to the confidentiality properties of data, the property of interest. Without this connection, authorization is an empty charade; with it, we have a clear formulation of how authorization ensures confidentiality.

The remainder of the paper is organized as follows. We conclude this section with some concrete examples to elucidate the proposed methodology. In Section II we introduce the programming language that enforces access control and define the dynamic semantics so that it generates an action trace. Initially this language is presented without authorized declassification so that it enforces strict non-interference. In Section III we describe the pertinent features of the epistemic logic. In Section IV we state the technical results of adequacy and non-interference. Section V adds authorized declassification and revises the theorems to account for it. Related work is surveyed in Section VI, and Section VII describes some avenues for future work.

Examples

To explain the methodology we propose, we will give a brief description of each of its components through a simple example. Commands specifying effectful computation are run on behalf of a principal whereas pure expressions are not. Running on behalf of a principal permits the commands to read from and write to the store, which contains assignables with permission sets restricting who may access their contents. A command executing as one principal may perform a `sudo` to execute on behalf of another principal provided the type of the value returned by the command is non-informative and the new principal is more privileged than the principal that switched to it. These "upcalls" are based on the work of [2] where they are used to forget about reads in a command if its result type is non-informative, but there they are not explicitly represented as a change of principal in the syntax.

Below are two blocks of an imperative program in pseudo code. The first block is run by some principal $\mathtt{k}_1$, and the second is run by another principal $\mathtt{k}_2$. A third principal $\mathtt{k}$ is running at the top level and performs a `sudo` to $\mathtt{k}_1$ and then a `sudo` to $\mathtt{k}_2$.

```
{x <- get[a];
 set[b](x);} as k1
{y <- get[b];
 if y then
   set[c](y);
 else
   skip;} as k2
```

The static semantics of a command summarizes the read and write effects that take place when the command is executed. This summary consists of a set of principals who may see the result of the command and a set of principals to whom the command may disclose information. The first set is determined by the reads the command performs, and the second set is determined by the writes it performs. The read set for $\mathtt{k}_1$'s command above is restricted to the principals that may see the contents of the assignable `a`. Its write set is restricted by the principals that may receive information through the write to the assignable `b`. If the principals that may read `b` are not contained in the set of principals that may read `a` then this program would not preserve the confidentiality of `a`. Therefore, the static semantics imposes restrictions on the sequencing of commands. In $\mathtt{k}_2$'s command, the write set of the if-statement must reflect the possibility of a write to the assignable `c` even though the second branch has no writes. This requirement for the write set is necessary, as not modifying the value stored in `c` may also leak information.

In addition to executing these commands for their effects, the dynamic semantics produces a trace of the security relevant actions. For example, executing the first block of code as principal $\mathtt{k}_1$ generates the trace $\mathtt{rd}_{\mathtt{k}_1}[\mathtt{a}], \mathtt{wr}_{\mathtt{k}_1}[\mathtt{b}]$. The second block is more interesting as the actions executed depend on the initial state. If the boolean value in `b` is true then it will be written to `c` yielding the trace $\mathtt{rd}_{\mathtt{k}_2}[\mathtt{b}], \mathtt{wr}_{\mathtt{k}_2}[\mathtt{c}]$. Otherwise, there will not be a write and the trace will just contain $\mathtt{rd}_{\mathtt{k}_2}[\mathtt{b}]$.

This variation in the possible traces of a program exposes the deficiency of traces for identifying non-interference results shown in [8]. To avoid this deficiency we augment the trace with additional information from the type of the commands in the program. The type of the if-statement indicates that a write to c is possible. The trace for the second block will always reflect this possible write even if it does not occur dynamically. Representing possible writes in the trace is critical, as information can be disclosed by not modifying an assignable.

The next step of the methodology is to reason about the epistemic consequences of these trace actions. For example, the trace $\mathtt{rd}_{\mathtt{k}_1}[\mathtt{a}], \mathtt{wr}_{\mathtt{k}_1}[\mathtt{b}]$ should make it possible for the assignable b to know anything previously known by the assignable a. These epistemic consequences are represented in the logic by semantic actions defined for each of the actions in the trace. We present two of these semantic actions here to demonstrate what is expressed by the semantic actions for the read and write actions.

$$\mathtt{do}(\mathtt{rd}_{\mathtt{k}}[\mathtt{a}]) \otimes [\mathtt{k}]\mathtt{s}(X) \otimes [\mathtt{a}]\mathtt{s}(Y) \multimap [\mathtt{k}]\mathtt{s}(X * Y) \otimes [\mathtt{a}]\mathtt{s}(Y)$$

$$\mathtt{do}(\mathtt{wr}_{\mathtt{k}}[\mathtt{a}]) \otimes [\mathtt{k}]\mathtt{s}(X) \otimes [\mathtt{a}]\mathtt{s}(Y) \multimap [\mathtt{k}]\mathtt{s}(X) \otimes [\mathtt{a}]\mathtt{s}(X)$$

The propositions of the form $[p]\mathtt{s}(X)$ represent the knowledge of an entity p, which may be either a principal or an assignable. The special atomic proposition $\mathtt{s}()$ maintains all of the knowledge in a single proposition. The symbols X and Y represent two such sets of secrets. The $*$ operator combines these into a single set of secrets. Each entity's secrets are grouped in a single $\mathtt{s}()$ as otherwise a semantic action that causes an entity to forget may only consume part of that entity's knowledge. The $\mathtt{do}(\alpha)$ proposition indicates the next trace action to be performed. The action $\mathtt{rd}_{\mathtt{k}}[\mathtt{a}]$ shares the assignable's knowledge, $\mathtt{s}(Y)$, with the principal k performing the action. The previous knowledge of the principal, $\mathtt{s}(X)$, is preserved so the resulting knowledge is $\mathtt{s}(X * Y)$. The knowledge of the assignable is also preserved. The action $\mathtt{wr}_{\mathtt{k}}[\mathtt{a}]$ is similar except the knowledge is transferred from the principal k to the assignable a and the previous knowledge of a is forgotten. Forgetting knowledge is an example of a use of the linearity of this knowledge modality.

To go beyond non-interference results, we introduce a declassification command that permits a principal to read an assignable it cannot directly access according to the permission set. The declassification command requires an authorization proof that the assignable may be declassified. These authorization proofs are expressed in the same epistemic logic used to represent the knowledge transfers from the actions. We focus on simple authorization proofs that are acquired through performing an authentication command that prompts the user to enter a password, but more general authorization policies are possible.

II. Language

In this section we describe the language presented in Figures 1 and 2. The terms of the language are divided into two syntactic categories yielding a monadic structure as in [2], but the type system and dynamic semantics differ in several ways including the explicit representation of non-informative upcalls as changing principals, the replacement of read and write levels with sets of principals, and the addition of the action traces.

A. Type system

The types $\mathtt{A}, \mathtt{B}$ are mostly familiar. The one notable exception is the type of suspended commands, $\bigcirc^{\mathtt{k}}_{(\Phi^{\mathtt{r}}, \Phi^{\mathtt{w}})}\mathtt{A}$. It indicates the type A returned by the command as well as its read and write sets and the principal on behalf of whom it is run. The expressions e are pure as their evaluation cannot cause writes or reads from assignables. Therefore, most of the typing rules for this part of the language are familiar as seen in the $:_{\mathtt{case}}$ rule. The only interesting typing rule for expressions is the $:_{\mathtt{cmd}}$ rule. This rule embeds a command in an expression as a suspended computation. Note that a suspended command is always a value so it will not perform any reads or writes when it is evaluated within an expression.

The commands m are effectful computations that may read or write to the store. Most of them are standard for a computational monadic language. For example, the sequencing command, $\mathtt{x} \leftarrow \mathtt{e};\ \mathtt{m}$, executes two commands one after another by substituting the value returned by the first into the second. The first command is suspended in an expression. The sudo command, $\mathtt{sudo}[\mathtt{k} \to \mathtt{k}'](\mathtt{m})$, is run on behalf of k, but it executes the command m on behalf of the more privileged principal, $\mathtt{k}'$. The command $\mathtt{a}_\Phi := \mathtt{e}\ \mathtt{in}\ \mathtt{m}$ declares a new assignable a with permission set Φ initialized to the value of e and then executes the command m.

The typing rules for commands have all of the restrictions necessary for controlling information flows. The $\div_{\mathtt{ret}}$ rule imposes no restrictions on the read or write sets as it simply returns a pure expression without performing any effects. As get is used to read an assignable, the $\div_{\mathtt{get}}$ rule requires that the executing principal is in the permission set Φ associated with the assignable being read and that this set contains the read level of the command. Intuitively, it imposes no restrictions on the write set. Conversely, the $\div_{\mathtt{set}}$ rule requires that the permission set is contained in the write set and imposes no restriction on the read set. Note that no restriction is made on the principal on behalf of whom the set command executes as we focus on confidentiality and ignore the integrity of the assignables. As usual, integrity can be addressed with a dual technique, but for simplicity we initially address confidentiality. In this setting the dual representation flips the lattice of principals upside down, but most of the other components of the system remain the same.

The $\div_{\mathtt{sudo}}$ rule warrants more explanation. The idea of forgetting the reads performed by a more privileged command is from [2]. The key is that the type of the privileged command is required to be only informative at that read level or higher (Here the read set $\Phi^{\mathtt{r}}_1$ is higher than $\Phi^{\mathtt{r}}_2$ if $\Phi^{\mathtt{r}}_1 \subset \Phi^{\mathtt{r}}_2$ as a smaller read set imposes more restrictions on what may follow the command). Therefore, if another command can extract

$$
\begin{array}{llll}
\text{Types} & \mathtt{A},\mathtt{B} & ::= & \mathtt{A}+\mathtt{B} \mid \bigcirc^{\mathtt{k}}_{(\Phi^{\mathtt{r}},\Phi^{\mathtt{w}})}\,\mathtt{A} \mid 1 \mid \mathtt{A}\times\mathtt{B} \mid \mathtt{A}\to\mathtt{B} \mid \cdots \\
\text{Expressions} & \mathtt{e} & ::= & \mathtt{case}(\mathtt{e},\mathtt{x}_1.\mathtt{e}_1,\mathtt{x}_2.\mathtt{e}_2) \mid \mathtt{cmd}^{\mathtt{k}}_{(\Phi^{\mathtt{r}},\Phi^{\mathtt{w}})}(\mathtt{m}) \mid () \mid (\mathtt{e}_1,\mathtt{e}_2) \mid \cdots \\
\text{Commands} & \mathtt{m} & ::= & \mathtt{x}\leftarrow\mathtt{e};\ \mathtt{m} \mid \mathtt{ret}\ \mathtt{e} \mid \mathtt{get}[\mathtt{a}] \mid \mathtt{set}[\mathtt{a}](\mathtt{e}) \mid \mathtt{sudo}[\mathtt{k}\to\mathtt{k}'](\mathtt{m}) \mid \mathtt{a}_\Phi := \mathtt{e}\ \mathtt{in}\ \mathtt{m} \\
\text{Context} & \Gamma & ::= & \cdot \mid \Gamma,\mathtt{x}:\mathtt{A} \\
\text{Store Ctx.} & \Sigma & ::= & \cdot \mid \Sigma,\mathtt{a}:\mathtt{A}|\Phi
\end{array}
$$

$\boxed{\Sigma;\Gamma\vdash\mathtt{e}:\mathtt{A}}$

$$
\frac{\Sigma;\Gamma\vdash\mathtt{m}\div_{\mathtt{k}}\mathtt{A}@(\Phi^{\mathtt{r}},\Phi^{\mathtt{w}})}{\Sigma;\Gamma\vdash\mathtt{cmd}^{\mathtt{k}}_{(\Phi^{\mathtt{r}},\Phi^{\mathtt{w}})}(\mathtt{m}):\bigcirc^{\mathtt{k}}_{(\Phi^{\mathtt{r}},\Phi^{\mathtt{w}})}\mathtt{A}}\ :_{\mathtt{cmd}} \qquad \frac{\Sigma;\Gamma\vdash\mathtt{e}:\mathtt{B}_1+\mathtt{B}_2 \quad \Sigma;\Gamma,\mathtt{x}_i:\mathtt{B}_i\vdash\mathtt{e}_i:\mathtt{A} \quad (i=1,2)}{\Sigma;\Gamma\vdash\mathtt{case}(\mathtt{e},\mathtt{x}_1.\mathtt{e}_1,\mathtt{x}_2.\mathtt{e}_2):\mathtt{A}}\ :_{\mathtt{case}}
$$

$\boxed{\Sigma;\Gamma\vdash\mathtt{m}\div_{\mathtt{k}}\mathtt{A}@(\Phi^{\mathtt{r}},\Phi^{\mathtt{w}})}$

$$
\frac{\Sigma;\Gamma\vdash\mathtt{e}:\mathtt{A}}{\Sigma;\Gamma\vdash\mathtt{ret}\ \mathtt{e}\div_{\mathtt{k}}\mathtt{A}@(\Phi^{\mathtt{r}},\Phi^{\mathtt{w}})}\ \div_{\mathtt{ret}} \qquad \frac{\Sigma;\Gamma\vdash\mathtt{m}\div_{\mathtt{k}_2}\mathtt{A}@(\Phi^{\mathtt{r}}_2,\Phi^{\mathtt{w}}) \quad \mathtt{k}_1\sqsubset\mathtt{k}_2 \quad \mathtt{A}\nearrow\Phi^{\mathtt{r}}_2 \quad \Phi^{\mathtt{r}}_1\supseteq\Phi^{\mathtt{r}}_2}{\Sigma;\Gamma\vdash\mathtt{sudo}[\mathtt{k}_1\to\mathtt{k}_2](\mathtt{m})\div_{\mathtt{k}_1}\mathtt{A}@(\Phi^{\mathtt{r}}_1,\Phi^{\mathtt{w}})}\ \div_{\mathtt{sudo}}
$$

$$
\frac{\mathtt{a}:\mathtt{A}|\Phi\in\Sigma \quad \Phi^{\mathtt{r}}\subseteq\Phi \quad \mathtt{k}\in\Phi}{\Sigma;\Gamma\vdash\mathtt{get}[\mathtt{a}]\div_{\mathtt{k}}\mathtt{A}@(\Phi^{\mathtt{r}},\Phi^{\mathtt{w}})}\ \div_{\mathtt{get}} \qquad \frac{\mathtt{a}:\mathtt{A}|\Phi\in\Sigma \quad \Sigma;\Gamma\vdash\mathtt{e}:\mathtt{A} \quad \Phi^{\mathtt{w}}\supseteq\Phi}{\Sigma;\Gamma\vdash\mathtt{set}[\mathtt{a}](\mathtt{e})\div_{\mathtt{k}}1@(\Phi^{\mathtt{r}},\Phi^{\mathtt{w}})}\ \div_{\mathtt{set}}
$$

$$
\frac{\Sigma;\Gamma\vdash\mathtt{e}:\bigcirc^{\mathtt{k}}_{(\Phi^{\mathtt{r}}_1,\Phi^{\mathtt{w}}_1)}\mathtt{A} \quad \Sigma;\Gamma,\mathtt{x}:\mathtt{A}\vdash\mathtt{m}\div_{\mathtt{k}}\mathtt{B}@(\Phi^{\mathtt{r}}_2,\Phi^{\mathtt{w}}_2) \quad \Phi^{\mathtt{w}}_2\subseteq\Phi^{\mathtt{r}}_1 \quad \Phi^{\mathtt{w}}\supseteq\Phi^{\mathtt{w}}_1\cup\Phi^{\mathtt{w}}_2 \quad \Phi^{\mathtt{r}}\subseteq\Phi^{\mathtt{r}}_1\cap\Phi^{\mathtt{r}}_2}{\Sigma;\Gamma\vdash\mathtt{x}\leftarrow\mathtt{e};\ \mathtt{m}\div_{\mathtt{k}}\mathtt{B}@(\Phi^{\mathtt{r}},\Phi^{\mathtt{w}})}\ \div_{\mathtt{seq}}
$$

$$
\frac{\Sigma;\Gamma\vdash\mathtt{e}:\mathtt{A} \quad \Sigma,\mathtt{a}:\mathtt{A}|\Phi;\Gamma\vdash\mathtt{m}\div_{\mathtt{k}}\mathtt{B}@(\Phi^{\mathtt{r}},\Phi^{\mathtt{w}})}{\Sigma;\Gamma\vdash\mathtt{a}_\Phi:=\mathtt{e}\ \mathtt{in}\ \mathtt{m}\div_{\mathtt{k}}\mathtt{B}@(\Phi^{\mathtt{r}},\Phi^{\mathtt{w}})}\ \div_{\mathtt{new}}
$$

$$
\begin{array}{llll}
\text{Actions} & \alpha & ::= & \mathtt{rd}_{\mathtt{k}}[\mathtt{a}] \mid \mathtt{wr}_{\mathtt{k}}[\mathtt{a}] \mid \mathtt{new}_{\mathtt{k}}[\mathtt{a}_\Phi] \mid \mathtt{leak}_{\mathtt{k}}(\Phi) \\
\text{Traces} & \mathtt{T} & ::= & \varepsilon \mid \alpha,\mathtt{T} \mid \mathtt{T}_1;\mathtt{T}_2 \mid [\mathtt{T}]_{\mathtt{k}\to\mathtt{k}'}
\end{array}
$$

$\boxed{\nu\Sigma.(\mathtt{m}\mid\mid\mu)\Downarrow_{\mathtt{k}}\nu\Sigma'.(\mathtt{m}'\mid\mid\mu');\mathtt{T}}$

$$
\frac{\mathtt{e}\Downarrow\mathtt{v} \quad \nu\Sigma,\mathtt{a}.(\mathtt{m}\mid\mid\mu\otimes\langle\mathtt{a}:\mathtt{v}\rangle)\Downarrow_{\mathtt{k}}\nu\Sigma'.(\mathtt{m}'\mid\mid\mu');\mathtt{T}}{\nu\Sigma.(\mathtt{a}_\Phi:=\mathtt{e}\ \mathtt{in}\ \mathtt{m}\mid\mid\mu)\Downarrow_{\mathtt{k}}\nu\Sigma'.(\mathtt{m}'\mid\mid\mu');\mathtt{new}_{\mathtt{k}}[\mathtt{a}_\Phi],\mathtt{T}}\ \Downarrow_{\mathtt{new}}
$$

$$
\frac{\mathtt{e}\Downarrow\mathtt{cmd}^{\mathtt{k}}_{(\Phi^{\mathtt{r}}_1,\Phi^{\mathtt{w}}_1)}(\mathtt{m}_1) \quad \nu\Sigma.(\mathtt{m}_1\mid\mid\mu)\Downarrow_{\mathtt{k}}\nu\Sigma'.(\mathtt{ret}\ \mathtt{v}_1\mid\mid\mu');\mathtt{T}_1 \quad \nu\Sigma'.([\mathtt{v}_1/\mathtt{x}]\mathtt{m}\mid\mid\mu')\Downarrow_{\mathtt{k}}\nu\Sigma''.(\mathtt{m}''\mid\mid\mu'');\mathtt{T}_2}{\nu\Sigma.(\mathtt{x}\leftarrow\mathtt{e};\ \mathtt{m}\mid\mid\mu)\Downarrow_{\mathtt{k}}\nu\Sigma''.(\mathtt{m}''\mid\mid\mu'');\mathtt{T}_1;\mathtt{leak}_{\mathtt{k}}(\Phi^{\mathtt{w}}_1);\mathtt{T}_2}\ \Downarrow_{\mathtt{seq}}
$$

$$
\frac{}{\nu\Sigma.(\mathtt{get}[\mathtt{a}]\mid\mid\mu\otimes\langle\mathtt{a}:\mathtt{v}\rangle)\Downarrow_{\mathtt{k}}\nu\Sigma.(\mathtt{ret}\ \mathtt{v}\mid\mid\mu\otimes\langle\mathtt{a}:\mathtt{v}\rangle);\mathtt{rd}_{\mathtt{k}}[\mathtt{a}]}\ \Downarrow_{\mathtt{get}}
$$

$$
\frac{\mathtt{e}\Downarrow\mathtt{v}'}{\nu\Sigma.(\mathtt{set}[\mathtt{a}](\mathtt{e})\mid\mid\mu\otimes\langle\mathtt{a}:\mathtt{v}\rangle)\Downarrow_{\mathtt{k}}\nu\Sigma.(\mathtt{ret}\ ()\mid\mid\mu\otimes\langle\mathtt{a}:\mathtt{v}'\rangle);\mathtt{wr}_{\mathtt{k}}[\mathtt{a}]}\ \Downarrow_{\mathtt{set}}
$$

$$
\frac{\mathtt{e}\Downarrow\mathtt{v}}{\nu\Sigma.(\mathtt{ret}\ \mathtt{e}\mid\mid\mu)\Downarrow_{\mathtt{k}}\nu\Sigma.(\mathtt{ret}\ \mathtt{v}\mid\mid\mu);\varepsilon}\ \Downarrow_{\mathtt{ret}} \qquad \frac{\nu\Sigma.(\mathtt{m}\mid\mid\mu)\Downarrow_{\mathtt{k}'}\nu\Sigma'.(\mathtt{ret}\ \mathtt{v}\mid\mid\mu');\mathtt{T}}{\nu\Sigma.(\mathtt{sudo}[\mathtt{k}\to\mathtt{k}'](\mathtt{m})\mid\mid\mu)\Downarrow_{\mathtt{k}}\nu\Sigma'.(\mathtt{ret}\ \mathtt{v}\mid\mid\mu');[\mathtt{T}]_{\mathtt{k}\to\mathtt{k}'}}\ \Downarrow_{\mathtt{sudo}}
$$

Fig. 1. Security language

$\boxed{\mathtt{A}\nearrow\Phi}$ $\quad \dfrac{}{\mathtt{A}\nearrow U}\nearrow_U$ $\quad \dfrac{}{1\nearrow\Phi}\nearrow_1$ $\quad \dfrac{\mathtt{B}\nearrow\Phi}{\mathtt{A}\to\mathtt{B}\nearrow\Phi}\nearrow_\to$ $\quad \dfrac{\mathtt{A}\nearrow\Phi_{\mathtt{A}} \quad \mathtt{B}\nearrow\Phi_{\mathtt{B}}}{\mathtt{A}\times\mathtt{B}\nearrow\Phi_{\mathtt{A}}\cup\Phi_{\mathtt{B}}}\nearrow_\times$ $\quad \dfrac{\mathtt{A}\nearrow\Phi_{\mathtt{A}}}{\bigcirc^{\mathtt{k}}_{(\Phi^{\mathtt{r}},\Phi^{\mathtt{w}})}\mathtt{A}\nearrow\Phi_{\mathtt{A}}\cup\Phi^{\mathtt{w}}}\nearrow_\bigcirc$

Fig. 2. Non-informativeness

information from the result of the computation then it could have performed the reads directly for itself.

The rules for when a type is informative are given in Figure 2. The set U indicates the set of all principals. It is the lowest read set so rule $\nearrow_U$ says that a command returning values of an arbitrary type A is informative at this read set or above. The unit type is never informative so rule $\nearrow_1$ asserts it is only informative at Φ or above for an arbitrary Φ including the empty set. The rule $\nearrow_\to$ depends on the informativeness of the return type as information is extracted from a function by applying it to an argument and observing the result. A pair is analyzed by projecting its components so it is as informative as the infimum of the informativeness of its two component types as indicated by taking their union in rule $\nearrow_\times$. A suspended command is analyzed through running it and analyzing the result, but information can also be extracted by reading the values written during the execution of the command. Therefore, rule $\nearrow_\bigcirc$ states that it is as informative as the infimum of its write set and the informativeness of its return type.

The upcalls are represented syntactically with the sudo command. The explicit sudo command facilitates representing the changes of principal in the traces produced by the dynamic semantics. As the name implies, upcalls can only be made to more privileged principals. The privilege restriction is enforced the ordering $\mathtt{k}_1 \sqsubset \mathtt{k}_2$ in the rule. Because the principal that makes the sudo upcall can transfer information to the more privileged principal, the partial order $\sqsubset$ imposes the restriction that the permission sets must be upward closed under $\sqsubset$. That is, if a principal $\mathtt{k} \in \Phi$ and $\mathtt{k} \sqsubset \mathtt{k}'$ then $\mathtt{k}' \in \Phi$. Therefore, the more privileged principal could have directly read any secret passed to it through a sudo.

The $\div_{\mathtt{seq}}$ rule requires that the second command's write set is contained in the first command's read set because the second command has access to the result of the first so any writes it performs may disclose secrets learned by the first part. The following example explains this condition. Henceforth, we omit the annotations on suspended commands.

$$\mathtt{x} \leftarrow \mathtt{cmd}(\mathtt{get}[\mathtt{a}]);\ \mathtt{set}[\mathtt{b}](\mathtt{x})$$

If the permission sets of the two assignables are $\Phi_\mathtt{a}$ and $\Phi_\mathtt{b}$, respectively, then the requirement imposed by our typing rules is that $\Phi_\mathtt{b} \subseteq \Phi_\mathtt{a}$. This containment is exactly what we need to prevent writing a secret from one assignable to another whose permission is less restrictive. Note that the same restriction is imposed even if the value written to b is a constant that does not depend on the value read from a. The type system is conservative in its enforcement.

As observed in [2], the write set of the $\div_{\mathtt{new}}$ rule is not restricted by the permission set of the new assignable because the initial value stored in the assignable cannot be leaked unless a reference to the assignable itself is leaked first. Therefore, even if a secret value is written to a freshly allocated assignable with no restrictions on who may read it, the fresh assignable cannot leak the secret unless the assignable itself is first leaked, which is prevented by the type system.

B. Dynamic semantics

Figure 1 defines actions and traces and presents the rules for evaluating a command with a store. As usual, a store is just a collection of assignables paired with values representing their contents. We consider this collection unordered for ease of use in the evaluation rules. There are actions in the traces corresponding to reading, writing, or declaring a new assignable. These actions only include the principal performing the action and the assignable. They do not include the values read or written. The leak action accounts for the writes that could have occurred in the first command of a sequence but possibly did not due to branching. These writes may still disclose information. For example, observing that an assignable is unchanged may reveal that a branch of a conditional statement was not taken, which reveals the value of the condition of that statement, which may depend on the value read from another assignable. A trace is either empty, ε, an action consed to the front of a trace, two traces appended together, or a sudo trace, $[\mathtt{T}]_{\mathtt{k}\to\mathtt{k}'}$, indicating that the subtrace T is executed on behalf of $\mathtt{k}'$.

The dynamic semantics are mostly straightforward with the small addition of the traces for recording the effects. The $\Downarrow_{\mathtt{ret}}$ rule produces an empty trace. The $\Downarrow_{\mathtt{new}}$ rule conses an action indicating the assignable declared to the front of the trace of its subcommand. The $\Downarrow_{\mathtt{get}}$ and $\Downarrow_{\mathtt{set}}$ rules just produce a single element trace to represent that the corresponding action has occurred. The $\Downarrow_{\mathtt{seq}}$ rule appends the traces of its subcommands together as expected, but it also adds a leak action in the middle. The $\Downarrow_{\mathtt{seq}}$ rule is also significant in that it demonstrates how a suspended command is executed. The expression is evaluated to a suspended command, but it is not until the command itself is paired with the current store and evaluated that the effects of the suspended command are executed. The resulting value from this execution along with the resulting store are then extracted and used to execute the second part of the command to produce a final result. The $\Downarrow_{\mathtt{sudo}}$ rule is primarily noteworthy for the change in principal from k in the conclusion to $\mathtt{k}'$ in the premise. The trace produced indicates this change of principals.

III. Epistemic Logic

The logic we employ to model the information flows is directly taken from [5]. In addition to the cut elimination property of the substructural epistemic logic, which permits proving the non-derivability of undesired knowledge transfers, the features of this logic that are of particular importance for our representation are the linear knowledge modality and the monad. The linear knowledge modality is also known as possession. It represents the possibility of an entity possessing a secret temporarily. For example, an assignable may temporarily possess one secret until another value is written to the assignable causing it to possess a different secret. Similarly, a principal may temporarily acquire some secrets during a sudo

$$\mathtt{do}(\mathtt{rd}_{\mathtt{k}}[\mathtt{a}]) \otimes [\mathtt{k}]\mathtt{s}(X) \otimes [\mathtt{a}]\mathtt{s}(Y) \multimap \{[\mathtt{k}]\mathtt{s}(X * Y) \otimes [\mathtt{a}]\mathtt{s}(Y) \otimes \mathtt{next}\}$$

$$\mathtt{do}(\mathtt{wr}_{\mathtt{k}}[\mathtt{a}]) \otimes [\mathtt{k}]\mathtt{s}(X) \otimes [\mathtt{a}]\mathtt{s}(Y) \multimap \{[\mathtt{k}]\mathtt{s}(X) \otimes [\mathtt{a}]\mathtt{s}(X) \otimes \mathtt{next}\}$$

$$\mathtt{do}(\lfloor_{\mathtt{k}\to\mathtt{k}'}) \otimes [\mathtt{k}]\mathtt{s}(X) \otimes [\mathtt{k}']\mathtt{s}(Y) \multimap \{[\mathtt{k}]\mathtt{s}(X) \otimes [\mathtt{k}']\mathtt{s}(X * Y) \otimes \mathtt{next}\}$$

$$\mathtt{do}(\rfloor_{\mathtt{k}'\to\mathtt{k}}) \otimes [\mathtt{k}']\mathtt{s}(X) \multimap \{[\mathtt{k}']\mathtt{s}(1) \otimes \mathtt{next}\}$$

$$\mathtt{do}(\mathtt{leak}_{\mathtt{k}}(\Phi)) \multimap \{\mathtt{do}(\mathtt{wr}_{\mathtt{k}}[\mathtt{a}])\}, \quad \text{if } \Phi \supseteq \Phi_{\mathtt{a}} \text{ where } \mathtt{a} : \mathtt{A}|\Phi_{\mathtt{a}} \in \Sigma$$

$$\mathtt{next} \otimes \mathtt{doTrace}(\alpha, T) \multimap \{\mathtt{do}(\alpha) \otimes \mathtt{doTrace}(T)\}$$

Fig. 3. Semantic Actions

and then lose them at the end of the sudo block. The monad indicated by $\{\}$ marks the main branch of a derivation making it possible to restrict the semantic actions to a single branch of the derivation so that the actions of the trace are processed one after another. The remaining features of the logic are familiar from standard linear logic.

The use of the logic is specialized to our application through the definition of semantic actions that define the epistemic consequences of each action in the trace. The semantic actions for the trace actions defined in the dynamic semantics are given in Figure 3. The traces have been slightly modified to encode the actions as a single sequence rather than containing the nesting structure given before. In particular, note that switching principals is represented by two different semantic actions. There is one for entering a sudo and another for leaving it. These play different roles as when principal k switches to run as principal $\mathtt{k}'$, $\mathtt{k}'$ receives secrets from k but when $\mathtt{k}'$ returns to k it does not disclose any secrets as the type is non-informative. Furthermore, the secrets acquired are forgotten by $\mathtt{k}'$ indicated by the knowledge $\mathtt{s}(1)$. The loss of all knowledge gained in the sudo means these secrets must be reacquired in any subsequent sudo. The modification of the traces also removes all actions for the declaration of new assignables and extends the store context for the entire trace with these assignables.

The semantic action for leak replaces it with one of the write actions that could have occurred. Choosing any single write that could have occurred may seem insufficient, but to prove a flow of knowledge is not possible, all derivations for the traces must be considered including any possible write that can be substituted for the leak. In general, a trace may cause a leak if there is any derivation that demonstrates the leak. This flexibility lends the logic its expressiveness. It would be more difficult to define precisely what happens as a result of a trace of actions. Instead the logic expresses what is possible, and if no derivation can be found then the information flow is not possible in the model.

The next proposition in the conclusions is a form of feedback to enable the next action to be processed forcing the actions to be consumed one after another as they appear in the trace. The final semantic action in Figure 3 consumes a next and makes the next action of the trace available to be processed. Using a proposition like next to restrict the order in which semantic actions are applied is a standard technique for this type of representation. If the traces and semantic actions were represented in an ordered logic there would be a more direct representation of this requirement for the order in which actions are processed.

In [5] a rewrite step, $\Gamma;\Delta \longrightarrow \Gamma';\Delta'$, is defined to essentially correspond to applying one of the semantic actions to consume some of the resources in $\Gamma;\Delta$ and produce some of the resources in $\Gamma';\Delta'$.

Definition III.1. ***(Rewrite Step).*** *The* ***rewrite step*** $\Gamma;\Delta \longrightarrow \Gamma';\Delta'$ *holds if and only if there exists a derivation of the form*

$$\begin{array}{c} \Gamma';\Delta' \vdash C \\ \vdots \\ \dfrac{\Gamma;\Delta_2, A_2^+ \vdash C}{\Gamma;\Delta_2, [\{A_2^+\}] \vdash C}\ \{\}_L \\ \vdots \\ \dfrac{\Gamma;\Delta_1, [A^-] \vdash C}{\Gamma;\Delta \vdash C}\ * \end{array}$$

parametric in C, where the rule marked $*$ is a rule transitioning from a neutral sequent to a left focused sequent, and the subderivation above $\{\}_L$ uses only invertible left rules.

In our application we impose some restrictions on the states Δ. These restrictions are specified in the following definition.

Definition III.2. *For a given set of assignables Σ and a set of principals, the state Δ is* ***valid*** *if and only if all of the following conditions are satisfied:*

1) *There is exactly one proposition of the form $[p]\mathtt{s}(X)$ for each entity p.*
2) *There is either a* next *or a* do *but not both*
3) *There is exactly one* doTrace*.*

As usual, we let $\Gamma;\Delta \longrightarrow^* \Gamma';\Delta'$, represent the reflexive, transitive closure of $\longrightarrow$. It is also convenient to define $\Gamma;\Delta \longrightarrow_T \Gamma';\Delta'$ when $\Gamma;\Delta, \mathtt{next}, \mathtt{doTrace}(T) \longrightarrow^* \Gamma';\Delta', \mathtt{next}, \mathtt{doTrace}(\epsilon)$. This terminology provides the basis to give the following definition.

Definition III.3. ***(Disclosure).*** *We say the trace T* ***discloses*** *p to p' if for all valid states Δ such that $[p]\mathtt{s}(X) \in \Delta$ there is a derivation of $\Gamma;\Delta \longrightarrow_T \Gamma';\Delta'$ such that $[p']\mathtt{s}(Y) \in \Delta'$ and $Y \supseteq X$.*

The definition only requires at least one such derivation to exist. Not all derivations will include the disclosure. For a trace T with leak actions, we say T discloses p to p' if any of the valid replacements of the leak actions with writes result in the disclosure.

$$\boxed{\Sigma; \Sigma'; \Gamma \vdash \mathtt{e} \approx_{\mathtt{k}} \mathtt{e}' : \mathtt{A}}$$

$$\frac{\mathtt{A} \nearrow \Phi \quad \mathtt{k} \notin \Phi \quad \Sigma; \Gamma \vdash \mathtt{v} : \mathtt{A} \quad \Sigma'; \Gamma \vdash \mathtt{v}' : \mathtt{A}}{\Sigma; \Sigma'; \Gamma \vdash \mathtt{v} \approx_{\mathtt{k}} \mathtt{v}' : \mathtt{A}} \approx$$

Fig. 4. Equivalence

IV. Adequacy and Non-Interference

In this section we describe the main technical results about the information flow properties of our language. Some technical results that are not essential for understanding the main technical development are deferred to the appendix. To state these results precisely we must define what is considered a flow of information in the language. If an information flow was defined to include too much the adequacy theorem would not hold. For example, if observing a difference in the number of steps of execution was considered a form of information flow then the theorem would be false. More importantly, if a principal $\mathtt{k}$ computes two different values but those values appear equivalent to $\mathtt{k}$ then we do not consider it a leak. For example, the type of the may not be informative to $\mathtt{k}$. We compare values up to equivalence relative to the view of a principal. This equivalence relative to a principal is defined to be the least congruence containing the rule in Figure 4. This $\approx$ rule equates two values if their type is not informative to the principal. A difference in the contents of the assignables in the initial store may affect whether certain new assignables are allocated. Therefore, the equivalence judgment is relative to two different assignable typing contexts. The notation $\Sigma_1 \cap \Sigma_2$ is used to indicate the assignable typing context containing assignables that appear in both, and the notation $\Sigma\restriction_{\mathtt{k}}$ indicates the context containing only those assignables whose permission sets include $\mathtt{k}$.

We will employ several properties of this equivalence. For example, Lemma A.1 states that substitution respects the equivalence judgment. It is essential to the proof of Lemma A.2, which states that equivalence is preserved under evaluation. The statements and proofs of these lemmas are deferred to the appendix.

A. Preliminary Technical Results

We prove some results about the definitions from the previous section that will be used to prove the central results of this section. The first theorem, which corresponds to the analogous result in [5], states a property of our model for reasoning about what is and is not derivable.

Theorem IV.1. ***(Rewrite Step Schemata).*** *Each rewrite step from a valid state has exactly one of the following forms:*

1) $\Gamma; \Delta, \mathtt{do}(\mathtt{rd}_{\mathtt{k}}[\mathtt{a}]), [\mathtt{k}]\mathtt{s}(X), [\mathtt{a}]\mathtt{s}(Y) \longrightarrow \Gamma; \Delta, [\mathtt{k}]\mathtt{s}(X * Y), [\mathtt{a}]\mathtt{s}(Y), \mathtt{next}$
2) $\Gamma; \Delta, \mathtt{do}(\mathtt{wr}_{\mathtt{k}}[\mathtt{a}]), [\mathtt{k}]\mathtt{s}(X), [\mathtt{a}]\mathtt{s}(Y) \longrightarrow \Gamma; \Delta, [\mathtt{k}]\mathtt{s}(X), [\mathtt{a}]\mathtt{s}(X), \mathtt{next}$
3) $\Gamma; \Delta, \mathtt{do}([_{\mathtt{k}\to\mathtt{k}'}), [\mathtt{k}]\mathtt{s}(X), [\mathtt{k}']\mathtt{s}(Y) \longrightarrow \Gamma; \Delta, [\mathtt{k}]\mathtt{s}(X), [\mathtt{k}']\mathtt{s}(X * Y), \mathtt{next}$
4) $\Gamma; \Delta, \mathtt{do}(]_{\mathtt{k}'\to\mathtt{k}}), [\mathtt{k}']\mathtt{s}(X) \longrightarrow \Gamma; \Delta, [\mathtt{k}']\mathtt{s}(1), \mathtt{next}$
5) $\Gamma; \Delta, \mathtt{do}(\mathtt{leak}_{\mathtt{k}}(\Phi)) \longrightarrow \Gamma; \Delta, \mathtt{do}(\mathtt{wr}_{\mathtt{k}}[\mathtt{a}])$ *such that* $\Phi \supseteq \Phi_{\mathtt{a}}$ *where* $\mathtt{a} : \mathtt{A} | \Phi_{\mathtt{a}} \in \Sigma$
6) $\Gamma; \Delta, \mathtt{doTrace}(\alpha, T), \mathtt{next} \longrightarrow \Gamma; \Delta, \mathtt{do}(\alpha), \mathtt{doTrace}(T)$

Moreover, the states in the conclusions of these rewrite steps are valid.

Proof: The proof relies on the observation that only the semantic actions have the monadic heads required for a left focusing step to begin a rewrite derivation. There is then a case for each semantic action. This proof is possible because of the structure of the logic and its cut elimination property. ■

This theorem is essential for showing that only permissible disclosures are derivable as it specifies the possible rewrite steps. Due to the monad restrictions in the logic, there are only a few possible rewrite steps, each corresponding to one of the semantic actions. It simplifies reasoning about disclosure by characterizing each step in a disclosure.

Theorem IV.1 facilitates the proofs of several lemmas about our disclosure judgment found in the appendix as well as the following lemma, which is used to prove properties of the traces of well-typed programs.

Lemma IV.1. ***(Disclosure Interpolation).*** *If a trace* $\mathtt{T}$ *is the concatenation of two traces (i.e.,* $\mathtt{T} = \mathtt{T}_1; \mathtt{T}_2$*) and* $\mathtt{T}$ *discloses* p *to* p' *then there is an entity* q *such that* $\mathtt{T}_1$ *discloses* p *to* q *and* $\mathtt{T}_2$ *discloses* q *to* p'*.*

Proof: The essential observation is that the rewrite steps in Theorem IV.1 only permit actions to be processed one at a time and in the specified order. Therefore, a derivation of $\Gamma; \Delta \longrightarrow_{\mathtt{T}} \Gamma'; \Delta'$ in which $[p]\mathtt{s}(X) \in \Delta$ and $[p']\mathtt{s}(Y) \in \Delta'$ yields two derivations: $\Gamma; \Delta \longrightarrow_{\mathtt{T}_1} \Gamma^*; \Delta^*$ and $\Gamma^*; \Delta^* \longrightarrow_{\mathtt{T}_2} \Gamma'; \Delta'$. As none of the rewrite steps create new knowledge, there must be at least one entity q such that $[q]\mathtt{s}(Z) \in \Delta^*$ and $Z \supseteq X$. The derivations of $\Gamma; \Delta \longrightarrow_{\mathtt{T}_1} \Gamma^*; \Delta^*$ for varying $\Gamma; \Delta$ then yield that $\mathtt{T}_1$ discloses p to q for any entity q that is always in possession of the secret. Such an entity may be found by considering the case where only p has the secret X initially. We similarly find that at least one of the entities also has the property that $\mathtt{T}_2$ discloses q to p' from the derivations of $\Gamma^*; \Delta^* \longrightarrow_{\mathtt{T}_2} \Gamma'; \Delta'$. ■

B. Adequacy

As already stated, it is not immediately obvious that the semantic actions correspond to the actual information flows in the program. The adequacy theorem asserts that any information disclosed according to the dynamic semantics of the language is also disclosed in the logic according to the formal definition given in the previous section.

We define information flow in a program both between an assignable and a principal and between two assignables in terms of the definition of equivalence. A flow between an assignable and a principal occurs when executing the program with different initial values for the assignable affects the

resulting value of a command executed by the principal. A flow between two assignables occurs when executing the program with different initial values for one assignable affects the final value of the other. This intuition for when there is a flow of information in the language yields the following adequacy theorem.

Theorem IV.2. ***(Adequacy).*** *If*
$\Sigma; \Sigma'; \cdot \vdash \mathtt{m} \approx_{\mathtt{k}} \mathtt{m}' \div_{\mathtt{k}} \mathtt{A@}(\Phi^{\mathtt{r}}, \Phi^{\mathtt{w}})$, $\mu : \Sigma$, $\mu' : \Sigma'$,
$\nu\Sigma.(\mathtt{m} \mid\mid \mu) \Downarrow_{\mathtt{k}} \nu\Sigma_{\mathtt{f}}.(\mathtt{ret\ v} \mid\mid \mu_{\mathtt{f}}); \mathtt{T}$, *and*
$\nu\Sigma'.(\mathtt{m}' \mid\mid \mu') \Downarrow_{\mathtt{k}} \nu\Sigma'_{\mathtt{f}}.(\mathtt{ret\ v}' \mid\mid \mu'_{\mathtt{f}}); \mathtt{T}'$ *then*

1) *Either* $\mathtt{v} \approx_{\mathtt{k}} \mathtt{v}'$ *or there exists* $b \in \Sigma \cap \Sigma'\restriction_{\mathtt{k}}$ *such that* $\mu(b) \not\approx_{\mathtt{k}} \mu'(b)$ *and* $\mathtt{T}$ *and* $\mathtt{T}'$ *disclose* b *to* $\mathtt{k}$.
2) *For all* $c \in \Sigma_{\mathtt{f}} \cap \Sigma'_{\mathtt{f}}\restriction_{\mathtt{k}}$ *such that* $\mu_{\mathtt{f}}(c) \not\approx_{\mathtt{k}} \mu'_{\mathtt{f}}(c)$ *there exists* $b \in \Sigma \cap \Sigma'\restriction_{\mathtt{k}}$ *such that* $\mu(b) \not\approx_{\mathtt{k}} \mu'(b)$ *and* $\mathtt{T}$ *and* $\mathtt{T}'$ *disclose* b *to* c.

Proof: Structural induction on the derivation of $\Downarrow$. We will give a representative case of the proof for the `seq` command. The other cases for when the command is a `seq` are similar, and the remaining cases are straightforward.

In the `seq` case we have $\mathtt{m} = \mathtt{x} \leftarrow \mathtt{e}_1;\ \mathtt{m}_2$ and $\mathtt{m}' = \mathtt{x} \leftarrow \mathtt{e}'_1;\ \mathtt{m}'_2$ such that:

- $\mathtt{e}_1 \Downarrow \mathtt{cmd}(\mathtt{m}_1)$ and $\mathtt{e}'_1 \Downarrow \mathtt{cmd}(\mathtt{m}'_1)$
- $\nu\Sigma.(\mathtt{m}_1 \mid\mid \mu) \Downarrow_{\mathtt{k}} \nu\Sigma_1.(\mathtt{ret\ v}_1 \mid\mid \mu_1); \mathtt{T}_1$
- $\nu\Sigma.(\mathtt{m}'_1 \mid\mid \mu') \Downarrow_{\mathtt{k}} \nu\Sigma'_1.(\mathtt{ret\ v}'_1 \mid\mid \mu'_1); \mathtt{T}'_1$
- $\nu\Sigma_1.([\mathtt{v}_1/x]\mathtt{m}_2 \mid\mid \mu_1) \Downarrow_{\mathtt{k}} \nu\Sigma_{\mathtt{f}}.(\mathtt{ret\ v} \mid\mid \mu_{\mathtt{f}}); \mathtt{T}_2$
- $\nu\Sigma'_1.([\mathtt{v}'_1/x]\mathtt{m}'_2 \mid\mid \mu'_1) \Downarrow_{\mathtt{k}} \nu\Sigma'_{\mathtt{f}}.(\mathtt{ret\ v}' \mid\mid \mu'_{\mathtt{f}}); \mathtt{T}'_2$

By Lemma A.2 we have $\mathtt{cmd}(\mathtt{m}_1) \approx_{\mathtt{k}} \mathtt{cmd}(\mathtt{m}'_1) : \bigcirc^{\mathtt{k}}_{(\Phi^{\mathtt{r}}_1, \Phi^{\mathtt{w}}_1)}\mathtt{B}$ as $\mathtt{e}_1 \approx_{\mathtt{k}} \mathtt{e}'_1$. By inversion we have either $\mathtt{m}_1 \approx_{\mathtt{k}} \mathtt{m}'_1$ or $\bigcirc^{\mathtt{k}}_{(\Phi^{\mathtt{r}}_1, \Phi^{\mathtt{w}}_1)}\mathtt{B} \nearrow \Phi$ and $\mathtt{k} \notin \Phi$. In the latter case, we have $\mathtt{B} \nearrow \Phi$ by inversion and therefore $\mathtt{v}_1 \approx_{\mathtt{k}} \mathtt{v}'_1 : \mathtt{B}$.
Otherwise, by the first case of the induction hypothesis on the evaluation derivation of $\mathtt{m}_1$ we have either $\mathtt{v}_1 \approx_{\mathtt{k}} \mathtt{v}'_1$ or there exists $\mathtt{b} \in \Sigma$ such that $\mu(\mathtt{b}) \not\approx_{\mathtt{k}} \mu'(\mathtt{b})$ and $\mathtt{T}_1$ and $\mathtt{T}'_1$ disclose $\mathtt{b}$ to $\mathtt{k}$. In the latter case, Lemma A.3 yields $\mathtt{T} = \mathtt{T}_1, \mathtt{leak}_{\mathtt{k}}(\Phi^{\mathtt{w}}_1), \mathtt{T}_2$ and $\mathtt{T}' = \mathtt{T}'_1, \mathtt{leak}_{\mathtt{k}}(\Phi^{\mathtt{w}}_1), \mathtt{T}'_2$ also disclose $\mathtt{b}$ to $\mathtt{k}$ completing this case.
In the cases when $\mathtt{v}_1 \approx_{\mathtt{k}} \mathtt{v}'_1$, we apply the first case of the induction hypothesis to $[\mathtt{v}_1/x]\mathtt{m}_2$ and $[\mathtt{v}'_1/x]\mathtt{m}'_2$ to conclude that $\mathtt{v} \approx_{\mathtt{k}} \mathtt{v}'$ or there exists $\mathtt{b} \in \Sigma$ such that $\mu_1(\mathtt{b}) \not\approx_{\mathtt{k}} \mu'_1(\mathtt{b})$ and $\mathtt{T}_2$ and $\mathtt{T}'_2$ disclose $\mathtt{b}$ to $\mathtt{k}$. In the former case, we are done. Otherwise, we apply the second case of the induction hypothesis to $\mathtt{m}_1$ and $\mathtt{m}'_1$ to conclude that there exists an assignable $\mathtt{c} \in \Sigma$ such that $\mu(\mathtt{c}) \not\approx_{\mathtt{k}} \mu'(\mathtt{c})$ and $\mathtt{T}_1$ and $\mathtt{T}'_1$ disclose $\mathtt{c}$ to $\mathtt{b}$. Then Lemma A.4 yields $\mathtt{T} = \mathtt{T}_1, \mathtt{leak}_{\mathtt{k}}(\Phi^{\mathtt{w}}_1), \mathtt{T}_2$ and $\mathtt{T}' = \mathtt{T}'_1, \mathtt{leak}_{\mathtt{k}}(\Phi^{\mathtt{w}}_1), \mathtt{T}'_2$ disclose $\mathtt{c}$ to $\mathtt{k}$ completing this case. ∎

The theorem essentially states that if there is a flow of information in the language then there is a corresponding disclosure in the trace according to the logic. This result is essential because of its contrapositive. When there is no disclosure in the trace according to the logic, there is no flow of information in the language. Therefore, to prove a non-interference result it is sufficient to consider whether a disclosure is derivable in the logic.

C. Non-Interference

To demonstrate the utility of this methodology we now present the proof of a non-interference result for this language. The adequacy theorem reduces proving a non-interference result about the language to proving that only permissible disclosures are derivable in the logic when reasoning about the traces of well-typed programs.

We can now prove the following lemma about the confidentiality properties of well-typed programs.

Lemma IV.2. ***(Typing Soundness).*** *If*
$\Sigma; \cdot \vdash \mathtt{m} \div_{\mathtt{k}} \mathtt{A@}(\Phi^{\mathtt{r}}, \Phi^{\mathtt{w}})$, $\mu : \Sigma$ *and*
$\nu\Sigma.(\mathtt{m} \mid\mid \mu) \Downarrow_{\mathtt{k}} \nu\Sigma_{\mathtt{f}}.(\mathtt{ret\ v} \mid\mid \mu_{\mathtt{f}}); \mathtt{T}$ *then*

1) *For all* $\mathtt{a} : \mathtt{A}|\Phi_{\mathtt{a}} \in \Sigma$ *if* $\mathtt{T}$ *discloses* $\mathtt{a}$ *to* $\mathtt{k}$ *then* $\Phi^{\mathtt{r}} \subseteq \Phi_{\mathtt{a}}$ *and* $\mathtt{k} \in \Phi_{\mathtt{a}}$
2) *For all* $\mathtt{a} : \mathtt{A}|\Phi_{\mathtt{a}} \in \Sigma$ *if* $\mathtt{T}$ *discloses* p *to* $\mathtt{a}$ *and* $p \neq \mathtt{a}$ *then* $\Phi_{\mathtt{a}} \subseteq \Phi^{\mathtt{w}}$
3) *For all* $\mathtt{a} : \mathtt{A}|\Phi_{\mathtt{a}} \in \Sigma$ *and* $\mathtt{b} : \mathtt{B}|\Phi_{\mathtt{b}} \in \Sigma$ *if* $\mathtt{T}$ *discloses* $\mathtt{a}$ *to* $\mathtt{b}$ *then* $\Phi_{\mathtt{a}} \supseteq \Phi_{\mathtt{b}}$

Proof: The proof is by structural induction on the evaluation derivation. The base cases for `set`, `get`, and `ret` are straightforward consequences of the typing rules. We give some representative cases for sequencing. They critically depend on Lemma IV.1 to reason about the disclosures of the two subcommands independently.
In these cases we have $m = \mathtt{x} \leftarrow \mathtt{e}_1;\ \mathtt{m}_2$ and:

- $\mathtt{e}_1 \Downarrow \mathtt{cmd}(\mathtt{m}_1)$
- $\nu\Sigma.(\mathtt{m}_1 \mid\mid \mu) \Downarrow_{\mathtt{k}} \nu\Sigma_1.(\mathtt{ret\ v}_1 \mid\mid \mu_1); \mathtt{T}_1$
- $\nu\Sigma_1.([\mathtt{v}_1/x]\mathtt{m}_2 \mid\mid \mu_1) \Downarrow_{\mathtt{k}} \nu\Sigma_{\mathtt{f}}.(\mathtt{ret\ v} \mid\mid \mu_{\mathtt{f}}); \mathtt{T}_2$
- $\mathtt{T} = \mathtt{T}_1, \mathtt{leak}_{\mathtt{k}}(\Phi^{\mathtt{w}}_1), \mathtt{T}_2$

For the first case, we assume $\mathtt{T}$ discloses $\mathtt{a}$ to $\mathtt{k}$. Therefore, we may apply Lemma IV.1 to conclude that there exists an entity p such that $\mathtt{T}_1, \mathtt{leak}_{\mathtt{k}}(\Phi^{\mathtt{w}}_1)$ discloses $\mathtt{a}$ to p and $\mathtt{T}_2$ discloses p to $\mathtt{k}$. We proceed by case analysis depending on whether p is and assignable or a principle.
In the former case where p is some assignable $\mathtt{b}$, we apply the first case of the induction hypothesis to $[\mathtt{v}_1/x]\mathtt{m}_2$ to conclude that $\Phi^{\mathtt{r}} \subseteq \Phi^{\mathtt{r}}_2 \subseteq \Phi_{\mathtt{b}}$ and $\mathtt{k} \in \Phi_{\mathtt{b}}$. If the derivation of disclosure actually used the $\mathtt{leak}_{\mathtt{k}}(\Phi^{\mathtt{w}}_1)$ then we would have $\mathtt{T}_1$ discloses $\mathtt{a}$ to $\mathtt{k}$ and would be finished by the induction hypothesis. Therefore, we may also apply the third case of the induction hypothesis to conclude that $\Phi_{\mathtt{a}} \supseteq \Phi_{\mathtt{b}}$. Combining this containment with the result of the earlier appeal to the induction hypothesis yields $\Phi^{\mathtt{r}} \subseteq \Phi_{\mathtt{a}}$ and $\mathtt{k} \in \Phi_{\mathtt{a}}$ completing this case.
In the latter case where p is a principal, $\mathtt{k}'$, Lemma A.5 yields that we must have $\mathtt{k}' = \mathtt{k}$ since any other principal could not carry its knowledge through the complete execution of $\mathtt{m}_1$ as the end of a `sudo` block would clear its knowledge. Therefore, this case also holds by an appeal to the first case of the induction hypothesis on $\mathtt{m}_1$.

The other cases employ similar reasoning about how the disclosure is divided between the two commands. ∎

Composing the adequacy and typing soundness results yields the expected non-interference result.

Theorem IV.3. ***(Non-interference).*** *If*
$\Sigma; \Sigma'; \cdot \vdash \mathtt{m} \approx_{\mathtt{k}} \mathtt{m}' \div_{\mathtt{k}} \mathtt{A@}(\Phi^{\mathtt{r}}, \Phi^{\mathtt{w}})$, $\mu : \Sigma$, $\mu' : \Sigma'$,
$\nu\Sigma.(\mathtt{m} \mid\mid \mu) \Downarrow_{\mathtt{k}} \nu\Sigma_{\mathtt{f}}.(\mathtt{ret\ v} \mid\mid \mu_{\mathtt{f}}); \mathtt{T}$, *and*
$\nu\Sigma'.(\mathtt{m}' \mid\mid \mu') \Downarrow_{\mathtt{k}} \nu\Sigma'_{\mathtt{f}}.(\mathtt{ret\ v}' \mid\mid \mu'_{\mathtt{f}}); \mathtt{T}'$ *then*

1) *If* $\mathtt{v} \not\approx_{\mathtt{k}} \mathtt{v}'$ *then there exists* $\mathtt{b} : \mathtt{B}|\Phi_{\mathtt{b}} \in \Sigma \cap \Sigma'$ *such that* $\mu(b) \not\approx_{\mathtt{k}} \mu'(b)$, $\Phi^{\mathtt{r}} \subseteq \Phi_{\mathtt{b}}$, *and* $\mathtt{k} \in \Phi_{\mathtt{b}}$
2) *For all* $\mathtt{b} : \mathtt{B}|\Phi_{\mathtt{b}} \in \Sigma_{\mathtt{f}} \cap \Sigma'_{\mathtt{f}}$ *such that* $\mu_{\mathtt{f}}(b) \not\approx_{\mathtt{k}} \mu'_{\mathtt{f}}(b)$, *there exists* $\mathtt{a} : \mathtt{A}|\Phi_{\mathtt{a}} \in \Sigma \cap \Sigma'$ *such that* $\mu(a) \not\approx_{\mathtt{k}} \mu'(a)$ *and* $\Phi_{\mathtt{a}} \supseteq \Phi_{\mathtt{b}}$

Proof: Non-interference is a corollary of Theorem IV.2 and Lemma IV.2 by just applying the latter to the disclosures in the conclusions of the former. ∎

The non-interference result may be better understood by considering the contrapositives of the two parts. If all assignables with k in their permission set are equivalent from the view of k then the resulting values will also be equivalent from the view of k, and if the initial values of every assignable with a permission set at least as permissive as a given assignable are equivalent from the view of k then the final values of the assignable will also be equivalent. Therefore, if only an assignable that does not have k in its permission set is modified then it will not interfere with the final value observed by k, and similarly, if only an assignable with a more restrictive permission set is modified then it will not interfere with the final value of the given assignable.

V. Declassification

The expressiveness of the logic facilitates the addition of authorized declassification. For simplicity, we focus on authorization proofs that take the form of a certificate verifying that the appropriate user's password has been correctly entered. These proofs are atomic in the logic. In this section we describe how the system changes to accommodate authorized declassification and how the theorems change to correspond to the more permissive policy.

A. Static and dynamic semantics

We introduce new commands for authentication and declassification as shown in Figure 5. The auth command prompts the user for the password of the indicated principal and verifies that it is entered correctly. Therefore, the type is an option as the verification may fail. The read level of the command is arbitrary as we assume all prompts for a given principal's password will receive a predetermined response that is the same between different runs of the command. The write level reflects that information in the form of a prompt is written to the user whose permission set is indicated by $\Phi_{\mathtt{IO}}$. The typing rule $\div_{\mathtt{decl}}$ is similar to the rule $\div_{\mathtt{get}}$ except that instead of requiring that the executing principal is allowed to access the assignable according to the permission set, the declassify command includes a proof in the form of a certificate that one of the principals able to access the assignable has authenticated itself. This certificate is obtained from a successful execution

$$\frac{\Phi^{\mathtt{w}} \supseteq \Phi_{\mathtt{IO}}}{\Sigma; \Gamma \vdash \mathtt{auth[k]} \div_{\mathtt{k}'} \mathtt{iam(k)\ option@}(\Phi^{\mathtt{r}}, \Phi^{\mathtt{w}})} \div_{\mathtt{auth}}$$

$$\frac{\mathtt{a} : \mathtt{A}|\Phi \in \Sigma \quad \Sigma; \Gamma \vdash \mathtt{e} : \mathtt{iam(k)} \quad \mathtt{k} \in \Phi}{\Sigma; \Gamma \vdash \mathtt{decl[a](e)} \div_{\mathtt{k}'} \mathtt{A@}(\Phi^{\mathtt{r}}, \Phi^{\mathtt{w}})} \div_{\mathtt{decl}}$$

$$\frac{\mathtt{S(k')} = \mathtt{NONE}}{\nu\Sigma.(\mathtt{auth[k']} \mid\mid \mu) \Downarrow^{\mathtt{S}}_{\mathtt{k}} \nu\Sigma.(\mathtt{ret\ NONE} \mid\mid \mu); \varepsilon} \Downarrow_{\mathtt{auth,NONE}}$$

$$\frac{\mathtt{S(k')} = \mathtt{SOME}\ \varphi}{\nu\Sigma.(\mathtt{auth[k']} \mid\mid \mu) \Downarrow^{\mathtt{S}}_{\mathtt{k}} \nu\Sigma.(\mathtt{ret\ SOME}\ \varphi \mid\mid \mu); \mathtt{auth}_{\mathtt{k}}[\mathtt{k}']} \Downarrow_{\mathtt{auth,SOME}}$$

$$\frac{\mathtt{e} \Downarrow \varphi \quad \mu(\mathtt{a}) = \mathtt{v}}{\nu\Sigma.(\mathtt{decl[a](e)} \mid\mid \mu) \Downarrow^{\mathtt{S}}_{\mathtt{k}} \nu\Sigma.(\mathtt{ret\ v} \mid\mid \mu); \mathtt{decl}_{\mathtt{k}}[\varphi][\mathtt{a}]} \Downarrow_{\mathtt{decl}}$$

$$\mathtt{do}(\mathtt{auth}_{\mathtt{k}}[\mathtt{k}']) \multimap \{!\mathtt{Auth(k')}\}$$
$$\mathtt{do}(\mathtt{decl}_{\mathtt{k}}[\varphi][\mathtt{a}]) \otimes [\mathtt{k}]\mathtt{s}(X) \otimes [\mathtt{a}]\mathtt{s}(Y) \otimes \mathtt{Auth(k')}$$
$$\multimap \{[\mathtt{k}]\mathtt{s}(X * Y) \otimes [\mathtt{a}]\mathtt{s}(Y) \otimes \mathtt{next}\}$$

Fig. 5. Rules for authorized declassification and additional semantic actions

of the auth command. Also unlike a standard get, the read level of the command does not reflect the permission set of the declassified assignable as this access is permitted through the authentication proof rather than the permission set.

The evaluation judgment now contains an additional parameter, S, that determines the outcome of each attempted authentication. Explicitly parameterizing by S rather than defining a non-deterministic evaluation judgment facilitates comparing two runs. This parameter is used to specify the two evaluation rules for authentication such that the resulting value depends on the value associated with the authenticating principal in S. The evaluation rule $\Downarrow_{\mathtt{decl}}$ is also very similar to the rule $\Downarrow_{\mathtt{get}}$. The only differences are that the authorization proof must be first evaluated and the result is included in the new trace action.

The semantic action modeling authorization adds a persistent proposition to the context indicating that the authenticated principal has authorized declassification. The semantic action modeling declassification is similar to the semantic action for a read as both have the same dynamic behavior in terms of the value produced. The only difference is that the declassification action requires a proof of authorization. This proof is found in the trace of any well-typed program that performs the declassification.

B. Authorized declassification

With the addition of authorized declassification it is no longer possible to prove a non-interference theorem as privileged inputs now affect public outputs through a declassification. Nevertheless, it is possible to prove that each disclosure

of this type requires a proof of authorization. Many of the lemmas previously given in the case without declassification still hold though possibly in a slightly modified form.

Theorem IV.1 that states the possible rewrite steps still holds with additional cases for the new actions. Theorem IV.2 that states the adequacy of the semantic actions for modeling the actual observed values produced by the dynamic semantics is modified to accommodate the disclosure of values through declassification. For the purposes of this theorem, the `decl` command behaves like a `get` command as the theorem does not mention permission sets. Therefore, the proof for this base case relies on the similarity between the semantic actions for `decl` and `get`. The only difference is the additional tracking of the authorization proofs, which does not affect the disclosures. However other cases must be more significantly modified to adjust to the more complicated form of disclosures in the presence of declassification.

Theorem V.1. ***(Adequacy).*** *If*
$\Sigma;\Sigma';\cdot\vdash \mathtt{m}\approx_{\mathtt{k}}\mathtt{m}'\div_{\mathtt{k}}\mathtt{A@}(\Phi^{\mathtt{r}},\Phi^{\mathtt{w}})$, $\mu:\Sigma$, $\mu':\Sigma'$,
$\nu\Sigma.(\mathtt{m}\mid\mid\mu)\Downarrow^{\mathtt{S}}_{\mathtt{k}}\nu\Sigma_{\mathtt{f}}.(\mathtt{ret\ v}\mid\mid\mu_{\mathtt{f}});\mathtt{T}$, *and*
$\nu\Sigma'.(\mathtt{m}'\mid\mid\mu')\Downarrow^{\mathtt{S}}_{\mathtt{k}}\nu\Sigma'_{\mathtt{f}}.(\mathtt{ret\ v}'\mid\mid\mu'_{\mathtt{f}});\mathtt{T}'$ *then*

1) *Either* $\mathtt{v}\approx_{\mathtt{k}}\mathtt{v}'$ *or there exists* `b` *such that* $\mathtt{b}\in\Sigma\cap\Sigma'\restriction_{\mathtt{k}}$ *or* $\exists\mathtt{a}\in\Sigma\cap\Sigma'\restriction_{\mathtt{k}'}$, $\mathtt{decl}_{\mathtt{k}}[\varphi][\mathtt{a}]\in\mathtt{T}$ *or* $\mathtt{T}'$, *and the respective* T *or* $\mathtt{T}'$ *discloses* `b` *to* `a` *such that* $\mu(\mathtt{b})\not\approx_{\mathtt{k}}\mu'(\mathtt{b})$ *and* T *and* $\mathtt{T}'$ *disclose* `b` *to* `k`.
2) *For all* $\mathtt{c}\in\Sigma_{\mathtt{f}}\cap\Sigma'_{\mathtt{f}}\restriction_{\mathtt{k}}$ *such that* $\mu_{\mathtt{f}}(\mathtt{c})\not\approx_{\mathtt{k}}\mu'_{\mathtt{f}}(\mathtt{c})$ *there exists* `b` *such that* $\mathtt{b}\in\Sigma\cap\Sigma'\restriction_{\mathtt{k}}$ *or* $\exists\mathtt{a}\in\Sigma\cap\Sigma'\restriction_{\mathtt{k}'}$, $\mathtt{decl}_{\mathtt{k}}[\varphi][\mathtt{a}]\in\mathtt{T}$ *or* $\mathtt{T}'$, *and the respective* T *or* $\mathtt{T}'$ *discloses* `b` *to* `a` *such that* $\mu(\mathtt{b})\not\approx_{\mathtt{k}}\mu'(\mathtt{b})$ *and* T *and* $\mathtt{T}'$ *disclose* `b` *to* `c`.

Proof: The proof is still by structural induction on the derivation of $\Downarrow$. We will give the same representative case of the proof for when the command is a `seq` to demonstrate how it must be changed to accommodate authorized declassification.

In the `seq` case we have $\mathtt{m}=\mathtt{x}\leftarrow\mathtt{e}_1;\ \mathtt{m}_2$ and $\mathtt{m}'=\mathtt{x}\leftarrow\mathtt{e}'_1;\ \mathtt{m}'_2$ such that:

- $\mathtt{e}_1\Downarrow\mathtt{cmd}(\mathtt{m}_1)$ and $\mathtt{e}'_1\Downarrow\mathtt{cmd}(\mathtt{m}'_1)$
- $\nu\Sigma.(\mathtt{m}_1\mid\mid\mu)\Downarrow_{\mathtt{k}}\nu\Sigma_1.(\mathtt{ret\ v}_1\mid\mid\mu_1);\mathtt{T}_1$
- $\nu\Sigma.(\mathtt{m}'_1\mid\mid\mu')\Downarrow_{\mathtt{k}}\nu\Sigma'_1.(\mathtt{ret\ v}'_1\mid\mid\mu'_1);\mathtt{T}'_1$
- $\nu\Sigma_1.([\mathtt{v}_1/x]\mathtt{m}_2\mid\mid\mu_1)\Downarrow_{\mathtt{k}}\nu\Sigma_{\mathtt{f}}.(\mathtt{ret\ v}\mid\mid\mu_{\mathtt{f}});\mathtt{T}_2$
- $\nu\Sigma'_1.([\mathtt{v}'_1/x]\mathtt{m}'_2\mid\mid\mu'_1)\Downarrow_{\mathtt{k}}\nu\Sigma'_{\mathtt{f}}.(\mathtt{ret\ v}'\mid\mid\mu'_{\mathtt{f}});\mathtt{T}'_2$

By Lemma A.2 we have $\mathtt{cmd}(\mathtt{m}_1)\approx_{\mathtt{k}}\mathtt{cmd}(\mathtt{m}'_1):\bigcirc^{\mathtt{k}}_{(\Phi^{\mathtt{r}}_1,\Phi^{\mathtt{w}}_1)}\mathtt{B}$ as $\mathtt{e}_1\approx_{\mathtt{k}}\mathtt{e}'_1$. By inversion we have either $\mathtt{m}_1\approx_{\mathtt{k}}\mathtt{m}'_1$ or $\bigcirc^{\mathtt{k}}_{(\Phi^{\mathtt{r}}_1,\Phi^{\mathtt{w}}_1)}\mathtt{B}\nearrow\Phi$ and $\mathtt{k}\notin\Phi$. In the latter case, we have $\mathtt{B}\nearrow\Phi$ by inversion and therefore $\mathtt{v}_1\approx_{\mathtt{k}}\mathtt{v}'_1:\mathtt{B}$.
Otherwise, by the first case of the induction hypothesis on the evaluation derivation of $\mathtt{m}_1$ we have either $\mathtt{v}_1\approx_{\mathtt{k}}\mathtt{v}'_1$ or there exists `b` such that $\mathtt{b}\in\Sigma\cap\Sigma'\restriction_{\mathtt{k}}$ or $\exists\mathtt{a}\in\Sigma\cap\Sigma'\restriction_{\mathtt{k}'}$, $\mathtt{decl}_{\mathtt{k}}[\varphi][\mathtt{a}]\in\mathtt{T}_1$ or $\mathtt{T}'_1$, and the respective $\mathtt{T}_1$ or $\mathtt{T}'_1$ discloses `b` to `a` and $\mu(\mathtt{b})\not\approx_{\mathtt{k}}\mu'(\mathtt{b})$ and $\mathtt{T}_1$ and $\mathtt{T}'_1$ disclose `b` to `k`.
In the latter case, Lemma A.3 yields $\mathtt{T}=\mathtt{T}_1,\mathtt{leak}_{\mathtt{k}}(\Phi^{\mathtt{w}}_1),\mathtt{T}_2$ and $\mathtt{T}'=\mathtt{T}'_1,\mathtt{leak}_{\mathtt{k}}(\Phi^{\mathtt{w}}_1),\mathtt{T}'_2$ also disclose `b` to `k`, and the traces satisfy the other conditions on `b` to complete this case.
In the cases when $\mathtt{v}_1\approx_{\mathtt{k}}\mathtt{v}'_1$, we apply the first case of the induction hypothesis to $[\mathtt{v}_1/x]\mathtt{m}_2$ and $[\mathtt{v}'_1/x]\mathtt{m}'_2$ to conclude that $\mathtt{v}\approx_{\mathtt{k}}\mathtt{v}'$ or there exists `b` such that $\mathtt{b}\in\Sigma\cap\Sigma'\restriction_{\mathtt{k}}$ or $\exists\mathtt{a}\in\Sigma\cap\Sigma'\restriction_{\mathtt{k}'}$, $\mathtt{decl}_{\mathtt{k}}[\varphi][\mathtt{a}]\in\mathtt{T}_2$ or $\mathtt{T}'_2$, and the respective $\mathtt{T}_2$ or $\mathtt{T}'_2$ discloses `b` to `a` and $\mu_1(\mathtt{b})\not\approx_{\mathtt{k}}\mu'_1(\mathtt{b})$ and $\mathtt{T}_2$ and $\mathtt{T}'_2$ disclose `b` to `k`.
In the former case, we have completed this case.
Otherwise, we apply the second case of the induction hypothesis to $\mathtt{m}_1$ and $\mathtt{m}'_1$ to conclude that there exists a `c` such that $\mathtt{c}\in\Sigma\cap\Sigma'\restriction_{\mathtt{k}}$ or $\exists\mathtt{a}\in\Sigma\cap\Sigma'\restriction_{\mathtt{k}'}$, $\mathtt{decl}_{\mathtt{k}}[\varphi][\mathtt{a}]\in\mathtt{T}_1$ or $\mathtt{T}'_1$, and the respective $\mathtt{T}_1$ or $\mathtt{T}'_1$ discloses `c` to `a` such that $\mu(c)\not\approx_{\mathtt{k}}\mu'(c)$ and T and $\mathtt{T}'$ disclose c to b. Then Lemma A.4 yields $\mathtt{T}=\mathtt{T}_1,\mathtt{leak}_{\mathtt{k}}(\Phi^{\mathtt{w}}_1),\mathtt{T}_2$ and $\mathtt{T}'=\mathtt{T}'_1,\mathtt{leak}_{\mathtt{k}}(\Phi^{\mathtt{w}}_1),\mathtt{T}'_2$ disclose `c` to `k` completing this case. Note that in this case the assignable that differs in the initial store is `c` while it may be the assignable `b` that is declassified to `k`. This possibility is what necessitated the more general statement of the theorem. ■

The typing soundness lemma changes more substantially as the theorem as originally stated no longer holds due to the addition of declassifications. The revised statement only guarantees non-interference for assignables that could not have been disclosed through declassification.

Lemma V.1. ***(Typing Soundness).*** *If*
$\Sigma;\cdot\vdash\mathtt{m}\div_{\mathtt{k}}\mathtt{A@}(\Phi^{\mathtt{r}},\Phi^{\mathtt{w}})$, $\mu:\Sigma$ *and*
$\nu\Sigma.(\mathtt{m}\mid\mid\mu)\Downarrow_{\mathtt{k}}\nu\Sigma_{\mathtt{f}}.(\mathtt{ret\ v}\mid\mid\mu_{\mathtt{f}});\mathtt{T}$ *then if each assignable* `c` *associated with a declassification action in* T *has permission set* $\Phi_{\mathtt{c}}$ *such that* $\Phi_{\mathtt{c}}\not\subseteq\Phi$ *then:*

1) *For all* $\mathtt{a}:\mathtt{A}|\Phi_{\mathtt{a}}\in\Sigma$ *if* $\Phi_{\mathtt{a}}\subseteq\Phi$ *and* T *discloses* `a` *to* `k` *then* $\Phi^{\mathtt{r}}\subseteq\Phi_{\mathtt{a}}$ *and* $\mathtt{k}\in\Phi_{\mathtt{a}}$
2) *For all* $\mathtt{a}:\mathtt{A}|\Phi_{\mathtt{a}}\in\Sigma$ *if* T *discloses* p *to* `a` *and* $p\neq\mathtt{a}$ *then* $\Phi_{\mathtt{a}}\subseteq\Phi^{\mathtt{w}}$
3) *For all* $\mathtt{a}:\mathtt{A}|\Phi_{\mathtt{a}}\in\Sigma$ *and* $\mathtt{b}:\mathtt{B}|\Phi_{\mathtt{b}}\in\Sigma$ *if* $\Phi_{\mathtt{a}}\subseteq\Phi$ *and* T *discloses* `a` *to* `b` *then* $\Phi_{\mathtt{a}}\supseteq\Phi_{\mathtt{b}}$

Proof: The proof follows the same structure as before. Most of the cases are essentially the same. The case for a declassification command uses the fact that the assignable being declassified cannot be the `a` in cases 1 and 3 because its permission set is not contained in Φ by assumption. ■

The non-interference theorem becomes the following theorem that includes the possibility of an authorized declassification that could subvert the permission sets. It is still a direct corollary of the type soundness and adequacy results.

Theorem V.2. ***(Authorized Declassification).***
If $\Sigma;\Sigma';\cdot\vdash\mathtt{m}\approx_{\mathtt{k}}\mathtt{m}'\div_{\mathtt{k}}\mathtt{A@}(\Phi^{\mathtt{r}},\Phi^{\mathtt{w}})$, $\mu:\Sigma$, $\mu':\Sigma'$,
$\nu\Sigma.(\mathtt{m}\mid\mid\mu)\Downarrow_{\mathtt{k}}\nu\Sigma_{\mathtt{f}}.(\mathtt{ret\ v}\mid\mid\mu_{\mathtt{f}});\mathtt{T}$, *and*
$\nu\Sigma'.(\mathtt{m}'\mid\mid\mu')\Downarrow_{\mathtt{k}}\nu\Sigma'_{\mathtt{f}}.(\mathtt{ret\ v}'\mid\mid\mu'_{\mathtt{f}});\mathtt{T}'$ *then*

1) *If* $\mathtt{v}\not\approx_{\mathtt{k}}\mathtt{v}'$ *then there exists* $\mathtt{b}:\mathtt{B}|\Phi_{\mathtt{b}}\in\Sigma\cap\Sigma'$ *such that* $\mu(b)\not\approx_{\mathtt{k}}\mu'(b)$ *and either* $\Phi^{\mathtt{r}}\subseteq\Phi_{\mathtt{b}}$ *and* $\mathtt{k}\in\Phi_{\mathtt{b}}$ *or there is a proof authorizing the declassification of an assignable* `c` *with permission set* $\Phi_{\mathtt{c}}\subseteq\Phi_{\mathtt{b}}$
2) *For all* $\mathtt{b}:\mathtt{B}|\Phi_{\mathtt{b}}\in\Sigma_{\mathtt{f}}\cap\Sigma'_{\mathtt{f}}$ *such that* $\mu_{\mathtt{f}}(b)\not\approx_{\mathtt{k}}\mu'_{\mathtt{f}}(b)$, *there exists* $\mathtt{a}:\mathtt{A}|\Phi_{\mathtt{a}}\in\Sigma\cap\Sigma'$ *such that* $\mu(a)\not\approx_{\mathtt{k}}\mu'(a)$

and either $\Phi_a \supseteq \Phi_b$ *or there is a proof authorizing the declassification of an assignable* c *with permission set* $\Phi_c \subseteq \Phi_a$

Proof: The presence of declassification actions in the trace in Theorem V.1 guarantees the existence of corresponding authorization proofs when composed with Lemma V.1. ■

VI. Related Work

There are numerous techniques for language-based reasoning about the information flow properties of programs. Some of the earlier work is summarized in [9]. In this section we summarize some points of contrast with this work.

The most closely related work is [2]. Our type system is based on theirs with the notable exception of our explicit representation of principals to directly model non-informative upcalls as principal switches. This modification facilitates the formation of traces to connect the programs with our substructural epistemic logic. The incorporation of this logic for reasoning about flows of information is the most significant deviation from this prior work.

In [10], epistemic logic is also applied to reason about information flow. This work employs a temporal epistemic logic for reasoning about security properties. Using a substructural epistemic logic grants us some of the advantages of a temporal logic as the linear context changes after each trace action corresponding to different temporal states. In [11], they extend this work by employing an SMT solver along with concrete and symbolic execution to verify the enforcement of the policies. This work is also similar in its use of traces of actions from the execution of the program to mediate between the program and the logic. Despite these similarities, this work substantially differs from ours in its overall approach. By analyzing programs individually rather than employing a type system to avoid illicit flows to all programs, their work is able to validate the non-interference of many programs that appear to leak information under most type system based approaches including ours. However, our approach provides a proof that the type system ensures non-interference or only authorized declassifications so that any program that is well-typed will be safe to execute without further analysis. Our work is distinguished from [12] as they employ the interpreted systems formalism [13] for reasoning about secrecy requirements in multiagent systems whereas we employ a static language-based approach

The epistemic reformulating of information flow properties and the repositioning of non-interference as a validation of the epistemic theory distinguish our work from other approaches to information flow. Early work on information flow includes that of [14]. In the Dependency Core Calculus, the monadic type seals values with a security level whereas following the approach of [2], the monad we employ encapsulates storage effects and specifies bounds on the security levels read and written. An encoding of this style of dependency analysis into the style of type system presented here is given in [2].

One motivation for our approach is its potential compatibility with authorized declassification through the use of the logic. A survey of work on declassification is given in [15], which also provides dimensions of declassification for analyzing these various approaches. Our approach does not directly address *what* information is released in a quantification sense, but it can be encoded in the system by choosing a sufficiently fine stratification of the security levels. If only parts of a value should be declassified, such as the last four digits of a credit card, then this part of the number could be stored in a separate assignable with a less restrictive permission set. Preventing an attacker from obtaining more information than intended through declassification motivates the definition of robust declassification in [16]. Our approach for encoding what-dimension declassification can partially address this issue at the cost of dealing with more cumbersome fine-grained security levels. Control of *who* releases information is enforced through our protocol that requires authentication to supply the proof that declassification is permitted. We restrict *when* information is released in the relative sense that it must follow authentication. Our policies do not dictate *where* information is declassified in either the level locality or code locality sense, but both could be addressed by increasing the expressiveness of the policies.

The authentication proofs and declassification policies presented here are simple. More complicated policies are necessary for many applications of authorized declassification. One technique for expressing policies for declassification is presented in [17]. Modeling information flow in systems with declassification is addressed in [16].

An approach for enforcing information flow policies with run-time principals is proposed in [18]. It provides a flexible method for enforcing policies that depend on data that is only available dynamically. It is unclear how our approach can be adapted to this setting as it relies on statically determining what traces are possible and reasoning about them in the epistemic logic. It would only validate a conservative subset of the safe programs whose policies depend on information only available at run time.

Other approaches have supported information flow reasoning in existing functional languages. For example, [19], [20], and [21] have demonstrated how to reason about information flow in Haskell, and [22] has addressed information flow in ML. Our work focuses on a simple monadically structured imperative language, and we designed the type system to explicitly restrict information flow. However, the general techniques are applicable to more robust languages. [23] defines non-interference in reactive systems such as web browsers. They develop a technique for proving non-interference properties in this setting.

In [24], information flow properties including non-interference are modeled with dependent types. The type system is expressive enough to state that low outputs must be equal if the low inputs are equal. Other approaches to enforcing security properties using dependent types are given in [25], which modifies the dependently-typed language Agda [26] for this purpose, and in [27], which employs the access control language AURA [28]. These approaches employing

dependent types are inherently more powerful in their ability to express and enforce policies. If this expressiveness is required, applying our epistemic reformulation of information flow properties to a dependently-typed language may be an interesting direction for future work.

VII. Future Work

The security policies considered in this paper were essentially normal access control lists for the assignables. However, the use of the substructural logic should facilitate enforcing more complicated security policies. These policies would be represented in the logic, and assessing the security of a trace would involve verifying that the relevant policy is satisfied for each disclosure. This initial presentation of our approach avoids this additional complexity.

The authorization proofs we consider are atomic tokens generated dynamically by having the user follow a protocol. One motivation for our approach is that it seems conducive to augmenting the authorization policy and representing it in the logic. The transfers of knowledge are represented in the logic so it seems natural to also express whether a leak is authorized by the policy if the policy is expressed in the logic. Properly representing these disclosures will require more precisely modeling the specific pieces of knowledge in the logic. Currently all of the knowledge of a principal is modeled by a single abstract predicate as we are interested in proving a non-interference property. However, if the policy distinguishes the disclosure permissions for different pieces of a principal's knowledge then this distinction must also be represented in the logic, which would require a more fine-grained approach to representing the knowledge exchanges in the trace. The type system would also have to be modified to account for these authorized declassifications.

The focus of this work is confidentiality although the appendix briefly addresses integrity as well. Our approach to preserving the integrity of data is the straightforward dual of confidentiality [29], but there are other techniques that are appropriate for more permissive integrity policies. Redesigning the system with the initial goal of enforcing integrity policies could yield an additional modality in the logic besides knowledge that is more appropriate for reasoning about integrity. Some integrity policies permit implicit flows as these are less likely to enable certain classes of attacks such as SQL injection. The type system and traces would need to be modified to account for these differences.

In [30], the Fabric system is proposed for building secure distributed information systems. Fabric builds on the Jif programming language [31]. One application for this work would be extending it to work in building distributed information systems. Integrating this work with the ML5 system for distributed systems proposed in [32] is one possible approach for advancing in this direction.

References

[1] J. A. Goguen and J. Meseguer, "Security Policies and Security Models," in *IEEE Symposium on Security and Privacy*, 1982, pp. 11–20.

[2] K. Crary, A. Kliger, and F. Pfenning, "A monadic analysis of information flow security with mutable state," *Journal of Functional Programming*, vol. 15, no. 2, pp. 249–291, 2005.

[3] M. Abadi, M. Burrows, B. Lampson, and G. Plotkin, "A calculus for access control in distributed systems," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 15, no. 4, pp. 706–734, 1993.

[4] J. Hintikka, *Knowledge and belief: an introduction to the logic of the two notions*. Cornell University Press Ithaca, 1962.

[5] H. DeYoung and F. Pfenning, "Reasoning about the consequences of authorization policies in a linear epistemic logic," in *Workshop on Foundations of Computer Security (FCS)*, 2009.

[6] S. Govindavajhala and A. W. Appel, "Using memory errors to attack a virtual machine," in *IEEE Symposium on Security and Privacy*, 2003, pp. 154–165.

[7] F. B. Schneider, "Enforceable security policies," *ACM Transactions on Information and System Security*, vol. 3, no. 1, pp. 30–50, 2000.

[8] A. Sabelfeld and A. C. Myers, "Language-based information-flow security," *IEEE Journal on Selected Areas in Communications*, vol. 21, 2003.

[9] M. Balliu, M. Dam, and G. L. Guernic, "Epistemic temporal logic for information flow security," in *Programming Languages and Analysis for Security (PLAS 2011)*, 2011.

[10] ——, "Encover: Symbolic exploration for information flow security," in *Computer Security Foundations Symposium (CSF), 2012 IEEE 25th*. IEEE, 2012, pp. 30–44.

[11] J. Halpern and K. O'Neill, "Secrecy in multiagent systems," *ACM Transactions on Information and System Security (TISSEC)*, vol. 12, no. 1, p. 5, 2008.

[12] R. Fagin, J. Halpern, Y. Moses, and M. Vardi, *Reasoning about knowledge*. MIT press Cambridge, MA, 1995, vol. 4.

[13] M. Abadi, A. Banerjee, N. Heintze, and J. G. Riecke, "A core calculus of dependency," in *26TH ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL)*. ACM Press, 1999, pp. 147–160.

[14] A. Sabelfeld and D. Sands, "Declassification: Dimensions and principles," *Journal of Computer Security*, vol. 17, no. 5, pp. 517–548, 2009.

[15] S. Zdancewic and A. C. Myers, "Robust declassification," in *IEEE Computer Security Foundations Workshop*. IEEE Computer Society Press, 2001, pp. 15–23.

[16] A. Banerjee, D. Naumann, and S. Rosenberg, "Expressive declassification policies and modular static enforcement," in *Security and Privacy, 2008. SP 2008. IEEE Symposium on*, May 2008, pp. 339 –353.

[17] S. Tse and S. Zdancewic, "Run-time principals in information-flow type systems," *ACM Trans. Program. Lang. Syst.*, vol. 30, no. 1, 2007.

[18] P. Li and S. Zdancewic, "Encoding information flow in haskell," in *Proceedings of the 19th IEEE workshop on Computer Security Foundations*, ser. CSFW '06. Washington, DC, USA: IEEE Computer Society, 2006.

[19] ——, "Arrows for secure information flow," *Theor. Comput. Sci.*, vol. 411, no. 19, pp. 1974–1994, 2010.

[20] A. Russo, K. Claessen, and J. Hughes, "A library for light-weight information-flow security in haskell," *SIGPLAN Not.*, vol. 44, no. 2, pp. 13–24, Sep. 2008.

[21] F. Pottier and V. Simonet, "Information flow inference for ml," in *Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, ser. POPL '02. New York, NY, USA: ACM, 2002, pp. 319–330.

[22] A. Bohannon, B. C. Pierce, V. Sjöberg, S. Weirich, and S. Zdancewic, "Reactive noninterference," in *Proceedings of the 16th ACM conference on Computer and communications security*, ser. CCS '09. New York, NY, USA: ACM, 2009, pp. 79–90.

[23] A. Nanevski, A. Banerjee, and D. Garg, "Verification of information flow and access control policies with dependent types," in *Security and Privacy (SP), 2011 IEEE Symposium on*, May 2011, pp. 165 –179.

[24] J. Morgenstern and D. R. Licata, "Security-typed programming within dependently-typed programming," in *International Conference on Functional Programming*, 2010.

[25] U. Norell, "Towards a practical programming language based on dependent type theory," Ph.D. dissertation, Department of Computer Science and Engineering, Chalmers University of Technology, SE-412 96 Göteborg, Sweden, September 2007.

[26] L. Jia and S. Zdancewic, "Encoding information flow in aura," in *Proceedings of the ACM SIGPLAN Fourth Workshop on Programming*

Languages and Analysis for Security, ser. PLAS '09. New York, NY, USA: ACM, 2009, pp. 17–29.

[27] L. Jia, J. A. Vaughan, K. Mazurak, J. Zhao, L. Zarko, J. Schorr, and S. Zdancewic, "Aura: a programming language for authorization and audit," in *Proceedings of the 13th ACM SIGPLAN international conference on functional programming*, ser. ICFP '08. New York, NY, USA: ACM, 2008, pp. 27–38.

[28] K. J. Biba, "Integrity considerations for secure computer systems," *Proceedings of the 4th annual symposium on Computer architecture*, vol. 5, no. 7, pp. 135–140, 1977.

[29] J. Liu, M. D. George, K. Vikram, X. Qi, L. Waye, and A. C. Myers, "Fabric: a platform for secure distributed computation and storage," in *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, ser. SOSP '09. New York, NY, USA: ACM, 2009, pp. 321–334.

[30] A. C. Myers, "Jflow: practical mostly-static information flow control," in *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, ser. POPL '99. New York, NY, USA: ACM, 1999, pp. 228–241.

[31] T. Murphy, VII, "Modal types for mobile code," Ph.D. dissertation, Carnegie Mellon, January 2008, available as technical report CMU-CS-08-126.

[32] J. Andreoli, "Logic programming with focusing proofs in linear logic," *Journal of Logic and Computation*, vol. 2, no. 3, pp. 297–347, 1992.

APPENDIX

A. Technical Lemmas

The following technical lemmas are not essential for understanding the development of this paper but may be interesting to the curious reader. The first lemma essentially states that substitution respects the equivalence judgment we defined.

Lemma A.1. ***(Functionality).***

1) *If* $\Sigma;\Sigma';\Gamma \vdash \mathtt{e} \approx_{\mathtt{k}} \mathtt{e}' : \mathtt{A}$ *and* $\Sigma;\Sigma';\Gamma, \mathtt{x} : \mathtt{A} \vdash \mathtt{m} \approx_{\mathtt{k}} \mathtt{m}' \div_{\mathtt{k}'} \mathtt{B@}(\Phi^{\mathtt{r}}, \Phi^{\mathtt{w}})$ *then* $\Sigma;\Sigma';\Gamma \vdash [\mathtt{e}/\mathtt{x}]\mathtt{m} \approx_{\mathtt{k}} [\mathtt{e}'/\mathtt{x}]\mathtt{m}' \div_{\mathtt{k}'} \mathtt{B@}(\Phi^{\mathtt{r}}, \Phi^{\mathtt{w}})$
2) *If* $\Sigma;\Sigma';\Gamma \vdash \mathtt{e} \approx_{\mathtt{k}} \mathtt{e}' : \mathtt{A}$ *and* $\Sigma;\Sigma';\Gamma, \mathtt{x} : \mathtt{A} \vdash \mathtt{e}_1 \approx_{\mathtt{k}} \mathtt{e}_1' : \mathtt{B}$ *then* $\Sigma;\Sigma';\Gamma \vdash [\mathtt{e}/\mathtt{x}]\mathtt{e}_1 \approx_{\mathtt{k}} [\mathtt{e}'/\mathtt{x}]\mathtt{e}_1' : \mathtt{B}$

Proof: The proof is essentially the same as the proof of the corresponding result in [2]. ∎

Functionality is used to prove the following lemma, which states that equivalence is preserved under evaluation.

Lemma A.2. ***(Equivalence under Evaluation).***

1) *If* $\Sigma;\Sigma';\cdot \vdash \mathtt{m} \approx_{\mathtt{k}} \mathtt{m}' \div_{\mathtt{k}} \mathtt{A@}(\Phi^{\mathtt{r}}, \Phi^{\mathtt{w}})$, $\Sigma;\Sigma' \vdash \mu \approx_{\mathtt{k}} \mu' : \Sigma \cap \Sigma'\restriction_{\mathtt{k}}$, $\nu\Sigma.(\mathtt{m} \mid\mid \mu) \Downarrow_{\mathtt{k}} \nu\Sigma_{\mathtt{f}}.(\mathtt{ret\ v} \mid\mid \mu_{\mathtt{f}});\mathtt{T}$, *and* $\nu\Sigma'.(\mathtt{m}' \mid\mid \mu') \Downarrow_{\mathtt{k}} \nu\Sigma'_{\mathtt{f}}.(\mathtt{ret\ v}' \mid\mid \mu'_{\mathtt{f}});\mathtt{T}'$ *then* $\Sigma_{\mathtt{f}};\Sigma'_{\mathtt{f}};\cdot \vdash \mathtt{v} \approx_{\mathtt{k}} \mathtt{v}' : \mathtt{A}$ *and* $\Sigma_{\mathtt{f}};\Sigma'_{\mathtt{f}} \vdash \mu_{\mathtt{f}} \approx_{\mathtt{k}} \mu'_{\mathtt{f}} : \Sigma_{\mathtt{f}} \cap \Sigma'_{\mathtt{f}}\restriction_{\mathtt{k}}$
2) *If* $\Sigma;\Sigma';\cdot \vdash \mathtt{e} \approx_{\mathtt{k}} \mathtt{e}' : \mathtt{A}$, $\mathtt{e} \Downarrow \mathtt{v}$, *and* $\mathtt{e}' \Downarrow \mathtt{v}'$ *then* $\Sigma;\Sigma';\cdot \vdash \mathtt{v} \approx_{\mathtt{k}} \mathtt{v}' : \mathtt{A}$

Proof: We give the following interesting case:

1) $\mathtt{m} = \mathtt{x} \leftarrow \mathtt{e}_1;\ \mathtt{m}_2$ and $\mathtt{m}' = \mathtt{x} \leftarrow \mathtt{e}_1';\ \mathtt{m}_2'$
2) $\Sigma;\Sigma';\cdot \vdash \mathtt{e}_1 \approx_{\mathtt{k}} \mathtt{e}_1' : \bigcirc^{\mathtt{k}}_{(\Phi_1^{\mathtt{r}},\Phi_1^{\mathtt{w}})}\mathtt{B}$ and $\Sigma;\Sigma';\mathtt{x} : \mathtt{B} \vdash \mathtt{m}_2 \approx_{\mathtt{k}} \mathtt{m}_2' \div_{\mathtt{k}} \mathtt{A@}(\Phi_2^{\mathtt{r}}, \Phi_2^{\mathtt{w}})$
3) $\mathtt{e}_1 \Downarrow \mathtt{cmd}(\mathtt{m}_1)$ and $\mathtt{e}_1' \Downarrow \mathtt{cmd}(\mathtt{m}_1')$

The induction hypothesis yields $\mathtt{cmd}(\mathtt{m}_1) \approx_{\mathtt{k}} \mathtt{cmd}(\mathtt{m}_1')$. This case is interesting when the equivalence is by virtue of the $\approx$ rule rather than the compatibility rule. By inversion, we have $\bigcirc^{\mathtt{k}}_{(\Phi_1^{\mathtt{r}},\Phi_1^{\mathtt{w}})}\mathtt{B} \nearrow \Phi$ and $\mathtt{k} \notin \Phi$. Again by inversion, we have $\mathtt{B} \nearrow \Phi_{\mathtt{B}}$ and $\Phi_{\mathtt{B}} \cup \Phi_1^{\mathtt{w}} \subseteq \Phi$. Therefore, $\mathtt{k} \notin \Phi_1^{\mathtt{w}}$ and $\mathtt{k} \notin \Phi_{\mathtt{B}}$. So we have $\nu\Sigma.(\mathtt{m}_1 \mid\mid \mu) \Downarrow_{\mathtt{k}} \nu\Sigma_1.(\mathtt{ret\ v}_1 \mid\mid \mu_1);\mathtt{T}_1$ and $\nu\Sigma'.(\mathtt{m}_1' \mid\mid \mu') \Downarrow_{\mathtt{k}} \nu\Sigma_1'.(\mathtt{ret\ v}_1' \mid\mid \mu_1');\mathtt{T}_1'$ such that $\Sigma_1;\Sigma_1';\cdot \vdash \mathtt{v}_1 \approx_{\mathtt{k}} \mathtt{v}_1' : \mathtt{B}$ as $\mathtt{k} \notin \Phi_{\mathtt{B}}$ and $\mathtt{B} \nearrow \Phi_{\mathtt{B}}$. Moreover, as $\mathtt{k} \notin \Phi_1^{\mathtt{w}}$, $\Sigma_1;\Sigma_1' \vdash \mu_1 \approx_{\mathtt{k}} \mu_1' : \Sigma_1 \cap \Sigma_1'\restriction_{\mathtt{k}}$ so the induction hypothesis can be applied to $[\mathtt{v}_1/\mathtt{x}]\mathtt{m}_2$ and $[\mathtt{v}_1'/\mathtt{x}]\mathtt{m}_2'$ to complete this case. ∎

The next three lemmas assert properties of the disclosure judgment defined in Definition III.3. They are all proved using Theorem IV.1.

Lemma A.3. ***(Disclosure Extension).*** *If* $\Sigma;\cdot \vdash \mathtt{m} \div_{\mathtt{k}} \mathtt{A@}(\Phi^{\mathtt{r}}, \Phi^{\mathtt{w}})$, $\nu\Sigma.(\mathtt{m} \mid\mid \mu) \Downarrow_{\mathtt{k}} \nu\Sigma_{\mathtt{f}}.(\mathtt{ret\ v} \mid\mid \mu_{\mathtt{f}});\mathtt{T}$, $\mathtt{T} = \mathtt{T}_1, \mathtt{T}_2$, *and* $\mathtt{T}_1$ *discloses* $\mathtt{a}$ *to* $\mathtt{k}$ *then* $\mathtt{T}$ *discloses* $\mathtt{a}$ *to* $\mathtt{k}$.

Lemma A.4. ***(Disclosure Composition).*** *If* $\Sigma;\cdot \vdash \mathtt{m} \div_{\mathtt{k}} \mathtt{A@}(\Phi^{\mathtt{r}}, \Phi^{\mathtt{w}})$, $\nu\Sigma.(\mathtt{m} \mid\mid \mu) \Downarrow_{\mathtt{k}} \nu\Sigma_{\mathtt{f}}.(\mathtt{ret\ v} \mid\mid \mu_{\mathtt{f}});\mathtt{T}$, $\mathtt{T} = \mathtt{T}_1, \mathtt{T}_2$, $\mathtt{T}_1$ *discloses* p_1 *to* p_2, *and* $\mathtt{T}_2$ *discloses* p_2 *to* p_3 *then* $\mathtt{T}$ *discloses* p_1 *to* p_3.

Lemma A.5. ***(Principal Disclosure).*** *If* $\Sigma;\cdot \vdash \mathtt{m} \div_{\mathtt{k}} \mathtt{A@}(\Phi^{\mathtt{r}}, \Phi^{\mathtt{w}})$, $\nu\Sigma.(\mathtt{m} \mid\mid \mu) \Downarrow_{\mathtt{k}} \nu\Sigma_{\mathtt{f}}.(\mathtt{ret\ v} \mid\mid \mu_{\mathtt{f}});\mathtt{T}$, $\mathtt{T}$ *discloses* $\mathtt{a}$ *to* $\mathtt{k}'$ *then* $\mathtt{k} = \mathtt{k}'$.

B. Integrity

We have focused on confidentiality, but this methodology may also be applied for preserving integrity by modifying the interpretation of a few components change. For example, when enforcing confidentiality the set $\Phi^{\mathtt{r}}$ represented the set of principals permitted to know the value of the computation. When enforcing integrity, this set represents the set of principals that trust the value of the computation. Similarly, when enforcing confidentiality the set $\Phi^{\mathtt{w}}$ represented the set of principals to whom the computation may disclose information, but when enforcing integrity this set represents the set of principals whose assignables may be influenced by the computation. Therefore, we still require $\Phi^{\mathtt{r}} \supseteq \Phi^{\mathtt{w}}$ as the principals that trust the integrity of an assignable that may be influenced by a computation must also trust the integrity of the computation.

Despite these slight differences in semantics, the type system remains unchanged. The only difference is that we use the dual lattice for our relation between principals. If before $\mathtt{k} \sqsubset \mathtt{k}'$, we now have $\mathtt{k}' \sqsubset \mathtt{k}$ instead because rather than protecting the secrets of principal $\mathtt{k}'$ from being disclosed to principal $\mathtt{k}$, we are now protecting the integrity of the assignables of principal $\mathtt{k}'$ from principal $\mathtt{k}$. Abstractly, we are still preventing a flow of information from one principal to the other. As a result of this change in the lattice, the upward closed condition on permission sets also changes correspondingly. Now rather than always giving a more privileged principal permission to access an assignable, we always force a lower integrity principal to trust an assignable.

The same non-interference theorem still applies but may now be interpreted through the lens of integrity. The condition

that states the final values of the two computations will only differ if there is an assignable with initial values that differ in the two stores and with the executing principal in its permission set now indicates that these final values only differ if an assignable trusted by the executing principal differs in the two initial states. Therefore, if a computation is run in two different stores that only differ in the assignables not trusted by the principal then the result of the computation will be the same. The other condition now states that if the integrity of an assignable is compromised by a computation then there must be a more trusted assignable that had differing values in the initial state. The contrapositive of these two conditions implies that modifying the low integrity inputs will not affect the final values of the high integrity outputs. There are other integrity results that may be more appropriate for particular applications. We discuss one of these in the future work.

C. Linear Epistemic Logic

The important rules of the linear epistemic logic from [5] are presented in Figure 6. The monad, $\{\}$, is essential for isolating the application of the semantic effects to a single spine of the derivation. For simplicity, we have omitted polarity annotations in this work, but they are a familiar part of the focusing methodology of [33]. The rules are divided into right and left focusing sequents to chain non-invertible rules together as is common in focusing calculi.

Initial Sequents

$$\frac{}{\Gamma; p^+ \vdash [p^+]}\ \mathtt{atom}^+ \qquad \frac{}{\Gamma; [p^-] \vdash p^-}\ \mathtt{atom}^-$$

Positive Connectives

$$\frac{\Gamma; \Delta_1 \vdash [A^+] \quad \Gamma; \Delta_2 \vdash [B^+]}{\Gamma; \Delta_1, \Delta_2 \vdash [A^+ \otimes B^+]}\ \otimes_R$$

$$\frac{\Gamma; \Delta, A^+, B^+ \vdash J}{\Gamma; \Delta, A^+ \otimes B^+ \vdash J}\ \otimes_L$$

$$\frac{}{\Gamma; \cdot \vdash [\mathbf{1}]}\ \mathbf{1}_R \quad \frac{\Gamma; \Delta \vdash J}{\Gamma; \Delta, \mathbf{1} \vdash J}\ \mathbf{1}_L$$

$$\frac{\Gamma; \cdot \vdash A^-}{\Gamma; \cdot \vdash [!A^-]}\ !_R \quad \frac{\Gamma; \Delta, !A^- \vdash J}{\Gamma, A^-; \Delta \vdash J}\ !_L$$

$$\frac{\Gamma, A^-; \Delta, [A^-] \vdash J}{\Gamma, A^-; \Delta \vdash J}\ \mathtt{copy} \quad \frac{\Gamma; \Delta, [A^-] \vdash J}{\Gamma; \Delta, K\ \mathtt{has}\ A^- \vdash J}\ \mathtt{has}_L$$

$$\frac{\Gamma|_K; \Delta|_K \vdash A^-}{\Gamma; \Delta|_K \vdash [[K]A^-]}\ []_R \quad \frac{\Gamma; \Delta, K\ \mathtt{has}\ A^- \vdash J}{\Gamma; \Delta, [K]A^- \vdash J}\ []_L$$

$$\frac{\Gamma; \Delta \vdash A^-}{\Gamma; \Delta \vdash [A^-]}\ \mathtt{blur} \quad \frac{\Gamma; \Delta, [A^-] \vdash J}{\Gamma; \Delta, A^- \vdash J}\ \mathtt{lfoc}$$

Negative Connectives

$$\frac{\Gamma; \Delta, A^+ \vdash B^-}{\Gamma; \Delta \vdash A^+ \multimap B^-}\ \multimap_R$$

$$\frac{\Gamma; \Delta_1 \vdash [A^+] \quad \Gamma; \Delta_2, [B^-] \vdash J}{\Gamma; \Delta_1, \Delta_2, [A^+ \multimap B^-] \vdash J}\ \multimap_L$$

$$\frac{\Gamma; \Delta \vdash [a/x]A^-}{\Gamma; \Delta \vdash \forall x : \tau . A^-}\ \forall^a_R \quad \frac{\Gamma; \Delta, [[t/x]A^-] \vdash J}{\Gamma; \Delta, [\forall x : \tau . A^-] \vdash J}\ \forall_L$$

$$\frac{\Gamma; \Delta \vdash [A^+]}{\Gamma; \Delta \vdash A^+\mathtt{lax}}\ \mathtt{lax}_R$$

$$\frac{\Gamma; \Delta \vdash A^+\mathtt{lax}}{\Gamma; \Delta \vdash \{A^+\}}\ \{\}_R \quad \frac{\Gamma; \Delta, A^+ \vdash C^+\mathtt{lax}}{\Gamma; \Delta, [\{A^+\}] \vdash C^+\mathtt{lax}}\ \{\}_L$$

Fig. 6. Inference rules for the weakly focused sequent calculus