# An Epistemic Formulation of Information Flow Security

(Draft, Under Review)

Arbob Ahmad
Carnegie Mellon University
adahmad@cs.cmu.edu

Robert Harper
Carnegie Mellon University
rwh@cs.cmu.edu

## Abstract

The *non-interference (NI)* property defines a program to be secure if changes to high-security inputs cannot alter the values of low-security outputs. NI indirectly states the epistemic property that no low-security principal acquires knowledge of high-security data. We consider a directly epistemic account of *information flow (IF)* security focusing on the knowledge flows engendered by the program's execution. Storage effects are of primary interest, since principals acquire knowledge from the execution only through these effects. The IF properties of the individual effectful actions are characterized using a *substructural epistemic logic* that accounts for the knowledge transferred through their execution. We prove that a low-security principal never acquires knowledge of a high-security input by executing a well-typed program.

The epistemic approach has several advantages over NI. First, it directly accounts for the knowledge flow engendered by a program. Second, in contrast to the bimodal NI property, the epistemic approach accounts for authorized declassification. We prove that a low-security principal acquires knowledge of a high-security input *only if* it is authorized by a proof in authorization logic. Third, the explicit formulation of IF properties as an epistemic theory provides a crisp treatment of "side channels." Rather than prove that a principal *does not know* a secret, we instead prove that *it is not provable* that the principal knows that secret. The latter statement characterizes the "minimal model," for which a precise statement may be made, whereas the former applies to "any model," including those with "side channels" that violate the model's basic premises. Fourth, the NI property is re-positioned as providing an *adequacy* proof of the epistemic theory of effects, ensuring that the logical theory corresponds to the actual program behavior. In this way we obtain a generalization of the classical approach to IF security that extends to authorized declassification.

***Keywords*** information flow security, non-interference, authorized declassification, epistemic logic

## 1. Introduction

Standard accounts of *information flow security (IFS)* in programming languages express confidentiality in terms of the *non-interference (NI)* definition given by Goguen and Meseguer [16] stating that changes to high-security inputs cannot influence low-security outputs. This criterion may be seen as an indirect expression of the desired property that information provided by a high-security source is not disclosed to a low-security destination. Put in other terms, IFS seeks to enforce the *epistemic* requirement that knowledge possessed by high-security principals should not flow to low-security principals. In this paper we develop another approach to IFS based on the following key ideas:

1. IFS is defined in terms of *effects* that a program has on its execution environment, following Crary et al. [11].
2. A *substructural epistemic logic* proposed by DeYoung and Pfenning [13] tracks the flow of knowledge engendered by program actions.
3. *Authorized declassification* of sensitive data is permitted, using authorization logic [1, 15] to validate its use in a program.
4. The *computational adequacy* of the epistemic theory of program actions is verified using a generalization of NI [16].

The distinctive feature of the present approach is that IFS is phrased directly in terms of the knowledge that a principal may gain by executing a well-typed program. More precisely, IFS is expressed by necessary conditions on the derivability of statements of the form $[k]\,S$ stating that a principal $k$ possesses (knows) fact $S$. When $k$ is a low-security principal, and $S$ is a high-security fact, this amounts to proving that $k$ can come to know $S$ only under specified conditions—in the simplest case, falsehood, which amounts to showing that $k$ cannot know $S$ as the result of a well-typed execution. More generally, we show that $[k]\,S$ may be derived under such circumstances only if there is a proof in authorization logic underwriting the information flow. This generalizes the previous statement by admitting controlled declassification.

It is important to our account that security is phrased in terms of the *non-derivability* of a knowledge statement, rather than *derivability of its negation*. The distinction lies in the ever-present possibility of "side channels", such as measurements of time- and power-consumption, that are not typically modeled in a type system for IFS [8, 17]. Because of these "out of the model" attacks, it is almost never possible to prove the *negation* of the statement that a low-security principal knows a high-security fact. On the other hand it is possible to show that such a statement is *underivable* within the epistemic theory characterizing the effects of the program execution. This amounts to a precise characterization of the attacks against which the type system mounts a successful defense, avoiding the need for vague provisos about the limitations of the model.

The role of the NI property is re-positioned in our work as the basis for the proof of the computational adequacy of the epistemic theory. After all, a theory that predicts no knowledge disclosures would ensure that no knowledge statement is ever derivable. We must show, contrarily, that the epistemic theory is adequate in the

sense that it properly detects interference, even in the presence of declassification. This is achieved using a generalization of NI that, when coupled with authorization logic, provides a logical account of IFS in the presence of declassification, something that is problematic in the pure NI framework.

The epistemic framework we propose offers several advantages over purely NI-based accounts of IFS:

1. It focuses attention directly on the core problem of confidentiality, which is to control the *knowledge* that a principal may acquire as a result of executing a program.
2. It provides a linkup between *authorization* [1, 15] and *security* in a single framework. Authorization logic ensures that security protocols are *obeyed*; epistemic logic gives these protocols *meaning* in terms of confidentiality and integrity.
3. It naturally encompasses *declassification* [37], a well-known weakness of the pure NI framework.
4. It uses substructural reasoning to express *ephemeral*, as well as *persistent*, knowledge [13]. For example, an internal data structure may "know" a security at one moment, but "forget" that secret at the next moment (if, say, some datum is erased).
5. It provides the basis for developing a tool to explore the *epistemic consequences* of a programming language and its associated authorization logic. Such consequences can be hard to envision without mechanical assistance to trace out the possible consequences of a policy.

One notable weakness of our framework is the simplicity of its authorization logic. A more expressive authorization logic such as $\text{PCML}_5$ [4] can encode more interesting authorization policies and would demonstrate the full advantage of the linkup between authorization logic and epistemic logic.

### Overview

This section outlines the methodology of the paper by describing how it applies to some example programs. Although these descriptions omit many details, the important ideas are introduced.

The security language SL we analyze distinguishes computations that may have storage effects from effect free expressions by dividing them into two distinct syntactic classes in a monadic language for effects [24, 25, 32]. We write `e` for expressions and `m` for commands. An expression may contain a suspended command, which is denoted $\texttt{cmd}(\texttt{m})$, but evaluation of the expression never executes the command. Other than these suspended commands, expressions closely resemble the pure expressions of functional languages. The return command `ret` forms a command from an expression by returning the value of the expression without performing any effects. A suspended command is eliminated by the bind command $\texttt{x} \leftarrow \texttt{e}_1;\ \texttt{m}_2$. Bind evaluates the expression $\texttt{e}_1$ to a suspended command, which is a value. The suspended command is then executed, and the value it returns is bound to the variable `x` in the scope of the command $\texttt{m}_2$. For brevity, when the variable is not used we omit the binding, and when $\texttt{e}_1$ is just a suspended command we write $\texttt{x} \leftarrow \texttt{m}_1;\ \texttt{m}_2$ for $\texttt{x} \leftarrow \texttt{cmd}(\texttt{m}_1);\ \texttt{m}_2$. The commands `get` and `set` read from and write to assignable variables, which we will call assignables.

Each command is run on behalf of a principal, which dictates the storage operations it may perform. The `sudo` command switches the principal on behalf of whom a command is run enabling it to read assignables that the less privileged principal cannot access directly. Therefore, principals may also be thought of as roles. The `sudo` switches are reminiscent of upcalls performed by operating systems such as the UNIX sudo. Passing arbitrary values computed by a more privileged principal back to a less privileged principal would subvert the access restriction. To avoid this

$\Sigma = \texttt{a} : \texttt{bool@}\{\texttt{k}_1, \texttt{k}_2\}, \texttt{b} : \texttt{bool@}\{\texttt{k}_2\}, \texttt{c} : \texttt{bool@}\{\texttt{k}_2\}$

```
set[c](false);
x <- get[a];
sudo[k1->k2](y <- get[b];
              if y then
                if x then
                  cmd(set[c](true))
                else
                  cmd(ret <>)
              else
                cmd(ret <>);
              ret <>
             )
```

**Figure 1.** Example program 1

the `sudo` requires that the type of the value returned is informative only to principals that can perform all of the reads within the `sudo`. Crary et al. [11] defined non-informativeness to permit more privileged reads in a limited scope. Our approach is distinguished by the choice to syntactically represent the non-informative commands with a `sudo` command. We also differ in explicitly running commands on behalf of principals to dictate the effects that may be performed rather than implicitly representing this information in the type system.

The example program in Figure 1 reads the contents of the assignables `a` and `b` and writes their conjunction to the assignable `c` under the store typing $\Sigma$. The assignables `a`, `b`, and `c` store values of type `bool`. Each assignable is associated with a *permission set* of principals that may read its contents. The permission set of `a` is $\{\texttt{k}_1, \texttt{k}_2\} = \Phi_\texttt{a}$ indicating that both principals may read it. The permission set of both `b` and `c` is $\{\texttt{k}_2\} = \Phi_\texttt{b} = \Phi_\texttt{c}$ indicating that only $\texttt{k}_2$ may read them. As we focus on confidentiality rather than integrity, there is not an analogous restriction for writing to an assignable. The example program initially runs on behalf of $\texttt{k}_1$ and then performs a `sudo` to run on behalf of $\texttt{k}_2$. In this simplest case, the type returned by the `sudo` command is completely non-informative because it is unit. The `sudo` is necessary because the permission set $\Phi_\texttt{b}$ does not include the principal $\texttt{k}_1$. The conjunction is computed with a nested if-then-else expression.

The type system constrains which assignables a principal may read and in what order reads and writes may be performed in order to restrict the possible information flows. Each command is associated with an *effect level* based on the reads and writes it performs. The effect level consists of a *read set* that is a set of principals permitted to see the result returned by the command and a *write set* that is a set of principals to whom the command may disclose information through its writes. The read set is *a* set that may see the result rather than *the* set that may see the result because the type system builds in weakening as an admissible property. It is always safe from a confidentiality perspective to impose a more severe restriction on the principals that may see the result. For example, it is safe to assert that no principal is permitted to see the result. Similarly, it is safe to assert that a command may write to assignables readable by more principals than is actually true since this imposes a greater restriction on the information that may be passed into the command.

The intersection of the permission sets of the assignables read by a command is an upper bound on its read set as the value returned by the command may contain information read from any or all of these assignables. Only principals permitted to read all of the assignables may see the result. The union of the permission sets of the assignables written by a command is a lower bound on its write set as information passed into the command may affect

the value written to any one of the assignables modified by the command. Therefore, a principal that belongs to the permission set of any of the assignables may indirectly read this value from the assignable written. As a result, the bind command that sequences two commands requires that the read set of the bind is contained in the intersection of the read sets of the respective commands and that the write set of the bind contains the union of the write sets of the respective commands.

The read set of the first two commands of the example program must be contained in the permission set $\Phi_{\mathtt{a}}$, and their write set must contain $\Phi_{\mathtt{c}}$. The write to `c` imposes no constraint on the read set of the first command, and the read from `a` imposes no constraint on the write set of the second command.

The `sudo` command has an unrestricted read set because the resulting value is non-informative thereby preventing the reads performed within the `sudo` from disclosing information through its result. In the second part of the example run by $\mathtt{k}_2$, the read set of the `sudo` is unrestricted even though the read set of the command within the `sudo` must be contained in $\Phi_{\mathtt{b}}$. Therefore, the read set of the full example program need only be contained in the permission set $\Phi_{\mathtt{a}}$. The `sudo` preserves the write set of the command within it as writes performed within a `sudo` can be affected by values passed into the command just like other writes. The write set of the `sudo` in the example must contain the permission set $\Phi_{\mathtt{c}}$. This is also the write set of the full program as only `c` is written to by either principal. Note that the read set for the full program is $\Phi_{\mathtt{a}}$ reflecting that principal $\mathtt{k}_1$ only learns the contents of `a`, not `b`.

To prevent reading from one assignable and then writing the result to another with a less restrictive permission set, the bind command requires that the read set of the first command contains the write set of the second. For example, the bind executed by $\mathtt{k}_2$ first reads from `b` and then writes to `c`. Therefore, this restriction requires $\Phi_{\mathtt{b}} \supseteq \Phi_{\mathtt{c}}$. In the full program, the read from `a` also precedes the write to `c` within the `sudo` so $\Phi_{\mathtt{a}}$ must also contain $\Phi_{\mathtt{c}}$. These two requirements are reasonable as the final value written to `c` is the conjunction of the other two assignables so information may flow from both of them to `c`.

We are interested in analyzing how flows of knowledge are engendered by the computation. Therefore, we explicitly define the possible transfers of knowledge by specifying the epistemic consequences of the effects of a program. Each storage effect of a program has a corresponding semantic action in the epistemic logic. The semantic action expresses how the effect may transfer knowledge between the principals and assignables involved. The dynamic semantics produces a trace of the security-relevant effects when a command is executed.

Schneider [38] identifies the deficiency of traces for identifying NI. In the example program, if the initial contents of the assignables `a` and `b` are true and false, then the trace would contain effects for the first write to `c` and read from `a` by $\mathtt{k}_1$ and for the read from `b` by $\mathtt{k}_2$, but the second write to `c` is not performed when `b` is false as the else branch just returns unit. However, information is disclosed from the assignables `a` and `b` to the assignable `c` even when the second write isn't performed as the final value stored in `c` is always the conjunction of the other two assignables. To always represent these disclosures regardless of the initial contents of the assignables, the trace is augmented with pseudo effects that are derived from the type of the commands in the program. Using static typing information derived from the whole program avoids the issue with traces for NI identified by Schneider [38]. The type of the if-then-else statement indicates that a write to `c` is possible because the type summarizes what may happen in either branch. The pseudo effects in the trace reflect that this write is possible even when it does not occur dynamically. The pseudo effects are critical as information can be disclosed by not modifying an assignable as indicated by the example.

We have an epistemic theory that governs the epistemic consequences of traces. The semantic actions express the epistemic consequences of each effect in the trace as linear implications that consume the next trace action in the context and modify the knowledge of principals and assignables appropriately. Linearity enables principals and assignables to "forget" knowledge. An assignable may forget its previous contents when it is overwritten, and a principal may forget what it learned within a `sudo` command when it leaves the scope of the variables it bound in the `sudo`. 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 proposition $[p]\mathtt{s}(X)$ represents the knowledge of an entity $p$, which may be either a principal or an assignable. The special atomic proposition $\mathtt{s}(X)$ maintains all of the knowledge in a single proposition. We abstractly represent the knowledge of the principals because we are primarily interested in how information flows between principals, not precisely what information is transferred. The symbols $X$ and $Y$ represent two such sets of secrets implicitly universally quantified over each formula. The $*$ operator combines these into a single set of secrets. Each entity's secrets are grouped in a single $\mathtt{s}(X)$ as otherwise a semantic action that causes an entity to forget may only consume part of that entity's knowledge. For example, if there was another proposition, $[\mathtt{a}]\mathtt{s}(Z)$, in the context then the semantic action for writing to `a` would only erase part of the previous information stored in the assignable. 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.

Returning to the example, if the initial contents of `a` and `b` are both true then the trace of the example program would be:

$$\mathtt{wr}_{\mathtt{k}_1}[\mathtt{c}], \mathtt{rd}_{\mathtt{k}_1}[\mathtt{a}], \mathtt{sudo}[\mathtt{k}_1 \rightarrow \mathtt{k}_2], \mathtt{rd}_{\mathtt{k}_2}[\mathtt{b}], \mathtt{wr}_{\mathtt{k}_2}[\mathtt{c}], \mathtt{sudo}[\mathtt{k}_1 \leftarrow \mathtt{k}_2]$$

This trace makes `c` learn anything previously known by `a` and `b`. The flow from `b` to `c` is immediate as $\mathtt{k}_2$ writes to `c` after reading `b`. The flow from `a` to `c` requires an intermediate flow of knowledge from $\mathtt{k}_1$ to $\mathtt{k}_2$, which results from the semantic action for $\mathtt{sudo}[\mathtt{k}_1 \rightarrow \mathtt{k}_2]$ since any variable bound before the `sudo` remains in scope within the `sudo`. The trace also results in a flow of knowledge from `a` to $\mathtt{k}_1$ but not from `b` to $\mathtt{k}_1$ as this was the purpose of reading `b` within the `sudo`. If `a` contained false instead then the trace would be the same except the $\mathtt{wr}_{\mathtt{k}_2}[\mathtt{c}]$ action would be replaced by a pseudo action reflecting that this write was possible even though it did not occur dynamically.

The disclosure of knowledge between principals and assignables defined by the semantic actions has several important properties that will be formally stated and proved later, but we briefly introduce a few of them here. The disclosure interpolation lemma states that if $\mathtt{T}$ discloses `a` to `c` and $\mathtt{T} = \mathtt{T}_1, \mathtt{T}_2$ then there is an entity $p$ such that $\mathtt{T}_1$ discloses `a` to $p$ and $\mathtt{T}_2$ discloses $p$ to `c`. In the example trace the lemma holds of the disclosure from `a` to `c` with $p = \mathtt{k}_2$ by taking $\mathtt{T}_1 = \mathtt{wr}_{\mathtt{k}_1}[\mathtt{c}], \mathtt{rd}_{\mathtt{k}_1}[\mathtt{a}], \mathtt{sudo}[\mathtt{k}_1 \rightarrow \mathtt{k}_2]$ and $\mathtt{T}_2 = \mathtt{rd}_{\mathtt{k}_2}[\mathtt{b}], \mathtt{wr}_{\mathtt{k}_2}[\mathtt{c}], \mathtt{sudo}[\mathtt{k}_1 \leftarrow \mathtt{k}_2]$.

The knowledge transfers derived from the trace in the epistemic theory are related to the values computed through the execution of the program that produced the trace by the adequacy theorem for the epistemic theory. The adequacy theorem implies that if we

$\Sigma' = \mathtt{a} : \mathtt{bool@}\{\mathtt{k}_1, \mathtt{k}_2\}, \mathtt{b} : \mathtt{bool@}\{\mathtt{k}_2\}, \mathtt{c} : \mathtt{bool@}\{\mathtt{k}_1, \mathtt{k}_2\}$

```
set[c](false);
x <- get[a];
pfOption <- auth[k2];
case pfOption of
  NONE => cmd(ret <>)
| SOME pf => cmd(y <- decl[b](pf);
                 if y then
                   if x then
                     cmd(set[c](true))
                   else
                     cmd(ret <>)
                 else
                   cmd(ret <>)
                );
ret <>
```

**Figure 2.** Example program 2

change the initial contents of a from true to false while leaving the rest of the state unchanged and if in the final state, the value stored in c changes from true to false then the trace must disclose a to c. The adequacy theorem demonstrates that the semantic actions correctly derive all information flows of a trace that may occur through the execution of the program that generated that trace. In fact, the semantic actions conservatively assume that more disclosures may occur than are actually possible.

The typing soundness lemma asserts that if the trace of a well-typed program discloses a to $\mathtt{k}_1$ then $\mathtt{k}_1$ must belong to $\Phi_\mathtt{a}$. In the example, this means $\mathtt{k}_1$ and $\mathtt{k}_2$ must belong to $\Phi_\mathtt{a}$ and $\mathtt{k}_2$ must belong to $\Phi_\mathtt{b}$. The conservativeness of the semantic actions makes the requirements of the typing soundness lemma more strict.

The familiar NI theorem implies that if two executions produce final states that differ in the value of c then there must be an assignable whose values differ in the initial states and whose permission set is no more restrictive than that of c. In the example, this could be either a or b as both have permission sets that are no more restrictive than c and the final value stored in c is computed from their initial values. NI is proved as a corollary of adequacy and typing soundness.

The epistemic approach can go beyond NI results. We introduce a declassification command decl that permits a principal to read an assignable it cannot directly access according to the permission set. The decl command requires an authorization proof that the assignable may be declassified. For simplicity, these authorization proofs are abstract tokens of authentication obtained by executing the auth command. The authorization proofs are proofs of propositions in the same epistemic logic used to represent the knowledge.

Figure 2 defines a second example program run on behalf of $\mathtt{k}_1$. It illustrates the use of declassification and authentication. As in the first example, the program computes the conjunction of the values stored in the assignables a and b and stores the result in c, but the store typing $\Sigma'$ differs from $\Sigma$ in the permission set $\Phi_\mathtt{c}$, which is now $\{\mathtt{k}_1, \mathtt{k}_2\}$. The first example program is ill-typed in $\Sigma'$ because $\Phi_\mathtt{b} \not\supseteq \Phi_\mathtt{c}$ so the write to c following the read from b is disallowed. To make the program well-typed, we declassify b rather than performing a sudo to access it. This declassification requires an authorization proof from an auth command that authenticates $\mathtt{k}_2$. If the authentication fails, the value stored in c will remain false; however, if it succeeds, b is declassified using the resulting proof, and the conjunction of a and b is written to c as before.

Authorized declassification complicates reasoning about information flow. In the example, the final contents of c in two runs of the program may differ based on the success or failure of the auth command even if the initial values stored in a and b are true in both runs. Many of the theorem statements are modified to account for the outcomes of authentication. Moreover, additional trace effects for declassification and authentication require semantic actions specifying their epistemic consequences. An authentication effect in the trace creates a persistent proof authorizing appropriate declassifications. This proof is discharged in the semantic action for declassification. Finally, the NI theorem is generalized to assert that a low-security principal acquires knowledge of a high-security input only if it is authorized by a proof in authorization logic.

## 2. Security language SL

Figures 3, 4, and 5 formally define SL. The terms are divided into two syntactic categories yielding a monadic structure as in Crary et al. [11], but the type system and dynamic semantics differ in a few ways including the explicit representation of non-informative upcalls as changing principals and the addition of effect traces.

### 2.1 Type system

The types $\mathtt{A}, \mathtt{B}$ defined in Figure 3 are mostly familiar. One notable exception is the type of suspended commands, $\mathtt{cmd}_\mathtt{k}[\Phi^\mathtt{r}, \Phi^\mathtt{w}](\mathtt{A})$. It indicates the type A returned by the command as well as its effect level 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, the typing judgment for expressions, $\Sigma; \Gamma \vdash \mathtt{e} : \mathtt{A}$, is mostly familiar as seen in rule 3.1. The most notable difference is the separation of the context into the variable context $\Gamma$ and the store context $\Sigma$. The store context is only used to type the suspended commands within expressions in rule 3.2. This rule embeds a command in an expression as a suspended computation. A suspended command is always a value so it does not perform any reads or writes when it is evaluated within an expression.

The commands m are computations that read from and write to the store. Most are standard for a monadic language such as Pfenning and Davies [33]. The judgment $\Sigma; \Gamma \vdash \mathtt{m} \div_\mathtt{k} \mathtt{A@}[\Phi^\mathtt{r}, \Phi^\mathtt{w}]$ is more elaborate than traditional monadic type systems because it restricts the effect level $[\Phi^\mathtt{r}, \Phi^\mathtt{w}]$ to control information flow. The subscript k indicates the principal on behalf of whom the command is run. The read set $\Phi^\mathtt{r}$ is a set of principals permitted to see the value returned by the command. The write set $\Phi^\mathtt{w}$ is a set of principals to whom the command may disclose information through its writes. The type system builds in weakening of the effect level as an admissible property. Therefore, a command with effect level $[\Phi_1^\mathtt{r}, \Phi_1^\mathtt{w}]$ may also be typed with effect level $[\Phi_2^\mathtt{r}, \Phi_2^\mathtt{w}]$ if $\Phi_2^\mathtt{r} \subseteq \Phi_1^\mathtt{r}$ and $\Phi_2^\mathtt{w} \supseteq \Phi_1^\mathtt{w}$. For example, rule 3.3 for typing the command ret e permits any effect level because the ret command evaluates the pure expression e without performing any effects.

Rule 3.4 for typing the bind command, $\mathtt{x} \leftarrow \mathtt{e}; \mathtt{m}$, is critical for restricting information flow. The expression e is a suspended command so its type $\mathtt{cmd}_\mathtt{k}[\Phi_1^\mathtt{r}, \Phi_1^\mathtt{w}](\mathtt{A})$ includes its effect level. The typing judgment for the command m gives the effect level $[\Phi_2^\mathtt{r}, \Phi_2^\mathtt{w}]$ of the second part of the bind. Rule 3.4 restricts how the subcommands of a bind may be chained together. Consider the bind when e is a suspended command that reads the assignable a, $\mathtt{cmd}_\mathtt{k}[\Phi_1^\mathtt{r}, \Phi_1^\mathtt{w}](\mathtt{get}[\mathtt{a}])$, and m writes the value read from a to another assignable b, $\mathtt{set}[\mathtt{b}](\mathtt{x})$. As get is used to read an assignable, rule 3.5 requires that the executing principal is in the permission set $\Phi_\mathtt{a}$ associated with a and that $\Phi_\mathtt{a} \supseteq \Phi_1^\mathtt{r}$. Intuitively, the rule imposes no restrictions on the write set. Conversely, rule 3.6 requires that the permission set $\Phi_\mathtt{b} \subseteq \Phi_2^\mathtt{w}$ and imposes no restriction on the read set. Therefore, the restriction of rule 3.4 that $\Phi_2^\mathtt{w} \subseteq \Phi_1^\mathtt{r}$ implies $\Phi_\mathtt{b} \subseteq \Phi_\mathtt{a}$ as $\Phi_\mathtt{b} \subseteq \Phi_2^\mathtt{w} \subseteq \Phi_1^\mathtt{r} \subseteq \Phi_\mathtt{a}$. This means that the principals that may learn the contents of a indirectly by reading the contents of b can also just directly read the contents of a.

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

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

$$\frac{\left\{\begin{array}{c}\Sigma; \Gamma \vdash \mathtt{e} : \mathtt{B}_1 + \mathtt{B}_2 \\ \Sigma; \Gamma, \mathtt{x_i} : \mathtt{B}_i \vdash \mathtt{e}_i : \mathtt{A} \quad (i = 1, 2)\end{array}\right\}}{\Sigma; \Gamma \vdash \mathtt{case}(\mathtt{e}, \mathtt{x}_1.\mathtt{e}_1, \mathtt{x}_2.\mathtt{e}_2) : \mathtt{A}}\ 3.1$$

$$\frac{\Sigma; \Gamma \vdash \mathtt{m} \div_{\mathtt{k}} \mathtt{A}@[\Phi^{\mathtt{r}}, \Phi^{\mathtt{w}}]}{\Sigma; \Gamma \vdash \mathtt{cmd_k}[\Phi^{\mathtt{r}}, \Phi^{\mathtt{w}}](\mathtt{m}) : \mathtt{cmd_k}[\Phi^{\mathtt{r}}, \Phi^{\mathtt{w}}](\mathtt{A})}\ 3.2$$

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

$$\frac{\Sigma; \Gamma \vdash \mathtt{e} : \mathtt{A}}{\Sigma; \Gamma \vdash \mathtt{ret}\ \mathtt{e} \div_{\mathtt{k}} \mathtt{A}@[\Phi^{\mathtt{r}}, \Phi^{\mathtt{w}}]}\ 3.3$$

$$\frac{\left\{\begin{array}{c}\Sigma; \Gamma \vdash \mathtt{e} : \mathtt{cmd_k}[\Phi_1^{\mathtt{r}}, \Phi_1^{\mathtt{w}}](\mathtt{A}) \\ \Sigma; \Gamma, \mathtt{x} : \mathtt{A} \vdash \mathtt{m} \div_{\mathtt{k}} \mathtt{B}@[\Phi_2^{\mathtt{r}}, \Phi_2^{\mathtt{w}}] \\ \Phi_2^{\mathtt{w}} \subseteq \Phi_1^{\mathtt{r}} \quad \Phi^{\mathtt{w}} \supseteq \Phi_1^{\mathtt{w}} \cup \Phi_2^{\mathtt{w}} \quad \Phi^{\mathtt{r}} \subseteq \Phi_1^{\mathtt{r}} \cap \Phi_2^{\mathtt{r}}\end{array}\right\}}{\Sigma; \Gamma \vdash \mathtt{x} \leftarrow \mathtt{e};\ \mathtt{m} \div_{\mathtt{k}} \mathtt{B}@[\Phi^{\mathtt{r}}, \Phi^{\mathtt{w}}]}\ 3.4$$

$$\frac{\mathtt{a} : \mathtt{A}@\Phi \in \Sigma \quad \Phi^{\mathtt{r}} \subseteq \Phi \quad \mathtt{k} \in \Phi}{\Sigma; \Gamma \vdash \mathtt{get}[\mathtt{a}] \div_{\mathtt{k}} \mathtt{A}@[\Phi^{\mathtt{r}}, \Phi^{\mathtt{w}}]}\ 3.5$$

$$\frac{\mathtt{a} : \mathtt{A}@\Phi \in \Sigma \quad \Sigma; \Gamma \vdash \mathtt{e} : \mathtt{A} \quad \Phi^{\mathtt{w}} \supseteq \Phi}{\Sigma; \Gamma \vdash \mathtt{set}[\mathtt{a}](\mathtt{e}) \div_{\mathtt{k}} 1@[\Phi^{\mathtt{r}}, \Phi^{\mathtt{w}}]}\ 3.6$$

$$\frac{\left\{\begin{array}{c}\Sigma; \Gamma \vdash \mathtt{m} \div_{\mathtt{k}_2} \mathtt{A}@[\Phi_2^{\mathtt{r}}, \Phi^{\mathtt{w}}] \\ \mathtt{k}_1 \sqsubset \mathtt{k}_2 \quad \mathtt{A} \nearrow \Phi_2^{\mathtt{r}} \quad \mathtt{k}_1 \notin \Phi_2^{\mathtt{r}}\end{array}\right\}}{\Sigma; \Gamma \vdash \mathtt{sudo}[\mathtt{k}_1 \to \mathtt{k}_2](\mathtt{m}) \div_{\mathtt{k}_1} \mathtt{A}@[\Phi_1^{\mathtt{r}}, \Phi^{\mathtt{w}}]}\ 3.7$$

$$\frac{\Sigma; \Gamma \vdash \mathtt{e} : \mathtt{A} \quad \Sigma, \mathtt{a} : \mathtt{A}@\Phi; \Gamma \vdash \mathtt{m} \div_{\mathtt{k}} \mathtt{B}@[\Phi^{\mathtt{r}}, \Phi^{\mathtt{w}}]}{\Sigma; \Gamma \vdash \mathtt{new}\ \mathtt{a}@\Phi := \mathtt{e}\ \mathtt{in}\ \mathtt{m} \div_{\mathtt{k}} \mathtt{B}@[\Phi^{\mathtt{r}}, \Phi^{\mathtt{w}}]}\ 3.8$$

**Figure 3.** SL static semantics

If each of the subcommands of a bind command reads from some assignables then the premise of rule 3.4 that requires $\Phi^{\mathtt{r}} \subseteq \Phi_1^{\mathtt{r}} \cap \Phi_2^{\mathtt{r}}$ forces the read level to be restrictive enough to protect all of the assignables read. For example, if the permission set of $\mathtt{a}$ is $\Phi_{\mathtt{a}} = \{\mathtt{k}_1, \mathtt{k}_2\}$ and the permission set of $\mathtt{b}$ is $\Phi_{\mathtt{b}} = \{\mathtt{k}_2, \mathtt{k}_3\}$ then the read level of a bind command that reads both of these assignables would be a subset of $\{\mathtt{k}_2\}$ as only $\mathtt{k}_2$ may read both assignables. This restriction is necessary in the case that the bind's result contains information about both values read from the assignables. For example, the result returned by the command could be a pair containing both values. Similarly, the premise that requires $\Phi^{\mathtt{w}} \supseteq \Phi_1^{\mathtt{w}} \cup \Phi_2^{\mathtt{w}}$ forces the write level to reflect all of the assignables that may be written in either part of the sequencing command. A value passed into the sequencing command may be written in either of the subcommands and therefore, any of the principals that can read any of the assignables written to by the command may be able to read that value.

Rules 3.5 and 3.6 for the `set` and `get` commands are asymmetric because there is no restriction on the principal on behalf of whom the `set` command executes whereas rule 3.5 requires that the executing principal is in the permission set of the assignable read. This asymmetry reflects the focus on the confidentiality instead of the integrity of the assignables. As usual, integrity in the sense of NI can be addressed with a dual technique [9]. This form of integrity prevents low-integrity inputs from influencing high-integrity outputs, but does not make any other guarantees about the values written to high-integrity outputs. The application to integrity is outlined in section B of the supplementary material. Essentially, the dual representation flips the lattice of principals upside down, but most of the other components of SL remain the same.

$\boxed{\mathtt{A} \nearrow \Phi}$

$$\frac{}{\mathtt{A} \nearrow \mathbf{U}}\ 4.1 \quad \frac{}{1 \nearrow \emptyset}\ 4.2$$

$$\frac{\mathtt{B} \nearrow \Phi}{\mathtt{A} \to \mathtt{B} \nearrow \Phi}\ 4.3 \quad \frac{\mathtt{A} \nearrow \Phi_{\mathtt{A}} \quad \mathtt{B} \nearrow \Phi_{\mathtt{B}}}{\mathtt{A} \times \mathtt{B} \nearrow \Phi_{\mathtt{A}} \cup \Phi_{\mathtt{B}}}\ 4.4$$

$$\frac{\mathtt{A} \nearrow \Phi_{\mathtt{A}}}{\mathtt{cmd_k}[\Phi^{\mathtt{r}}, \Phi^{\mathtt{w}}](\mathtt{A}) \nearrow \Phi_{\mathtt{A}} \cup \Phi^{\mathtt{w}}}\ 4.5 \quad \frac{\mathtt{A} \nearrow \Phi_1 \quad \Phi_1 \subseteq \Phi_2}{\mathtt{A} \nearrow \Phi_2}\ 4.6$$

**Figure 4.** Non-informativeness

Rule 3.7 for the `sudo` command relies on the informativeness judgment $\mathtt{A} \nearrow \Phi$ given in Figure 4. If $\mathtt{A} \nearrow \Phi$ then the type $\mathtt{A}$ is informative *only* to the principals in $\Phi$. The set $\mathbf{U}$ is the set of all principals. Asserting that a type is informative only to the principals in the set of all principals is trivially true so rule 4.1 says that any type $\mathtt{A}$ is informative only to the principals in $\mathbf{U}$. Rule 4.1 gives the informativeness rule for any type not otherwise specified. For example, it applies to the sum type $\mathtt{A} + \mathtt{B}$ because even if $\mathtt{A}$ and $\mathtt{B}$ are both uninformative, the sum type is still informative to any principal since it can see the outer tag. The unit type is never informative. Rule 4.2 asserts it is informative only to the principals in the empty set $\emptyset$. Rule 4.3 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 informative to a principal if either component is informative to that principal. Rule 4.4 specifies this by taking union of the two sets of principals to whom the component types are informative. A suspended command is analyzed by running it and viewing the result, but information can also be extracted by reading values written during the execution of the command. Therefore, rule 4.5 states that it is as informative as the union of the informativeness set of its return type and the set of principals that can read assignables written by it. It is safe to assert that a type is informative to more principals than it actually is. Rule 4.6 asserts that a type that is informative only to principals in $\Phi_1$ is also informative only to principals in a larger set.

The `sudo` command is the only one to change the principal on behalf of whom a command is run. The command $\mathtt{sudo}[\mathtt{k}_1 \to \mathtt{k}_2](\mathtt{m})$ switches principals from $\mathtt{k}_1$ to the more privileged $\mathtt{k}_2$ provided the result returned by $\mathtt{m}$ is uninformative to $\mathtt{k}_2$. The privilege restriction is enforced by the ordering $\mathtt{k}_1 \sqsubset \mathtt{k}_2$ in rule 3.4. 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`. While executing the privileged command as $\mathtt{k}_2$, values may be read that $\mathtt{k}_1$ cannot read. The premises $\mathtt{A} \nearrow \Phi_2^{\mathtt{r}}$ and $\mathtt{k}_1 \notin \Phi_2^{\mathtt{r}}$ assure that these values are not returned back to $\mathtt{k}_1$ through the result of the `sudo` because the type of the

value returned is uninformative to $\mathtt{k}_1$. In the first example program in Figure 1, principal $\mathtt{k}_1$ uses a `sudo` to $\mathtt{k}_2$ to read `b` and write its conjunction with `a` to `c`. The permission set of `b` is $\Phi_\mathtt{b} = \{\mathtt{k}_2\}$, so $\mathtt{k}_1$ cannot directly read `b`. The read set of the command in the `sudo` is contained in $\Phi_\mathtt{b}$ so rule 3.7 requires that the type returned by the command is only informative to the principal $\mathtt{k}_2$. The type returned is unit so it is not informative to any principal, but we weaken this and say it is informative only to $\mathtt{k}_2$. Encapsulating the read of `b` in a `sudo` command permits the surrounding computation to continue execution without any restriction based on the reads performed within the sudo. Note that the write set is not modified by the `sudo` command as any writes it performs can still leak information passed into it.

The command $\mathtt{new\ a@}\Phi := \mathtt{e\ in\ m}$ declares a new assignable `a` with permission set $\Phi$ initialized to the value of `e` and then executes the command `m`. As observed in Crary et al. [11], the write set of rule 3.8 is not restricted by the permission set $\Phi$ 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.

### 2.2 Dynamic semantics

Figure 5 defines memory stores, trace effects, and traces and presents the rules of the execution judgment for a command. A store $\mu$ is a collection of assignables paired with values representing their contents. We view stores modulo the order of assignables for ease of use in the evaluation rules. The effects $\alpha$ correspond to each of the effectful operations a command can perform. These effects include the principal that performed the operation and the assignable involved, not the values read or written. A basic trace is a list of effects so it is either empty, $\varepsilon$, or has an effect added to the front of a trace, $\alpha, \mathtt{T}$. We also form a trace by appending two subtraces together, $\mathtt{T}_1; \mathtt{T}_2$. A `sudo` trace $[\mathtt{T}]_{\mathtt{k}\to\mathtt{k}'}$ reflects that the subtrace `T` is nested inside a `sudo` command. When we reason about these traces in epistemic logic we reduce the traces to simple lists by replacing the `sudo` trace with two `sudo` effects for starting and ending the `sudo` before and after the subtrace. The subtraces of $\mathtt{T}_1; \mathtt{T}_2$ are also appended together to form a single list.

The execution judgment $\nu\Sigma.(\mathtt{m} \mid\mid \mu) \Downarrow_\mathtt{k} \nu\Sigma'.(\mathtt{m}' \mid\mid \mu')\ \{\mathtt{T}\}$ is divided into four parts. There are two execution states $\nu\Sigma.(\mathtt{m} \mid\mid \mu)$ and $\nu\Sigma'.(\mathtt{m}' \mid\mid \mu')$, each consisting of a store context of assignables in scope within the command, the command itself, and the memory store. The store context $\Sigma$ is similar to the store context used in the static semantics but without the type information. The permission set of every assignable in the store is given by the store context. The command in an execution state only contains free assignables in its store. The other two parts of the execution judgment are the principal `k` on behalf of whom the execution is performed and the trace `T` produced by the execution.

The rules of the execution judgment are mostly familiar from similar monadic languages with the small addition of the traces for recording the effects. Rule 5.1 produces an empty trace since the `ret` command just evaluates a pure expression and has no storage effects. Rule 5.2 concatenates the traces of its subcommands together as expected, but it also adds a `leak` pseudo effect in the middle to account for writes that may not have occurred dynamically but were possible according to the write set of the suspended command. When the example program in Figure 1 is executed, the second write to `c` may not occur dynamically but will be reflected in the trace through the `leak` effect. This rule defines how a suspended command is executed. The expression is evaluated to a suspended command, which is a value. Then the suspended command

$$
\begin{array}{llll}
\text{Store} & \mu & ::= & \cdot \mid \mu \otimes \langle \mathtt{a} : \mathtt{v} \rangle \\
\text{Effects} & \alpha & ::= & \mathtt{rd_k[a]} \mid \mathtt{wr_k[a]} \mid \mathtt{new_k[a]} \mid \mathtt{leak_k}(\Phi) \\
\text{Traces} & \mathtt{T} & ::= & \varepsilon \mid \alpha, \mathtt{T} \mid \mathtt{T}_1; \mathtt{T}_2 \mid [\mathtt{T}]_{\mathtt{k}\to\mathtt{k}'}
\end{array}
$$

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

$$\frac{\mathtt{e} \Downarrow \mathtt{v}}{\nu\Sigma.(\mathtt{ret\ e} \mid\mid \mu) \Downarrow_\mathtt{k} \nu\Sigma.(\mathtt{ret\ v} \mid\mid \mu)\ \{\varepsilon\}}\ 5.1$$

$$\frac{\left\{\begin{array}{c} \mathtt{e} \Downarrow \mathtt{cmd_k}[\Phi_1^\mathtt{r}, \Phi_1^\mathtt{w}](\mathtt{m}_1) \\ \nu\Sigma.(\mathtt{m}_1 \mid\mid \mu) \Downarrow_\mathtt{k} \nu\Sigma'.(\mathtt{ret\ v}_1 \mid\mid \mu')\ \{\mathtt{T}_1\} \\ \nu\Sigma'.([\mathtt{v}_1/\mathtt{x}]\mathtt{m} \mid\mid \mu') \Downarrow_\mathtt{k} \nu\Sigma''.(\mathtt{m}'' \mid\mid \mu'')\ \{\mathtt{T}_2\} \end{array}\right\}}{\nu\Sigma.(\mathtt{x} \leftarrow \mathtt{e};\ \mathtt{m} \mid\mid \mu) \Downarrow_\mathtt{k} \nu\Sigma''.(\mathtt{m}'' \mid\mid \mu'')\ \{\mathtt{T}_1; \mathtt{leak_k}(\Phi_1^\mathtt{w}); \mathtt{T}_2\}}\ 5.2$$

$$\frac{}{\nu\Sigma.(\mathtt{get[a]} \mid\mid \mu \otimes \langle \mathtt{a} : \mathtt{v} \rangle) \Downarrow_\mathtt{k} \nu\Sigma.(\mathtt{ret\ v} \mid\mid \mu \otimes \langle \mathtt{a} : \mathtt{v} \rangle)\ \{\mathtt{rd_k[a]}\}}\ 5.3$$

$$\frac{\mathtt{e} \Downarrow \mathtt{v}'}{\nu\Sigma.(\mathtt{set[a](e)} \mid\mid \mu \otimes \langle \mathtt{a} : \mathtt{v} \rangle) \Downarrow_\mathtt{k} \nu\Sigma.(\mathtt{ret}\ () \mid\mid \mu \otimes \langle \mathtt{a} : \mathtt{v}' \rangle)\ \{\mathtt{wr_k[a]}\}}\ 5.4$$

$$\frac{\nu\Sigma.(\mathtt{m} \mid\mid \mu) \Downarrow_{\mathtt{k}'} \nu\Sigma'.(\mathtt{ret\ v} \mid\mid \mu')\ \{\mathtt{T}\}}{\nu\Sigma.(\mathtt{sudo}[\mathtt{k} \to \mathtt{k}'](\mathtt{m}) \mid\mid \mu) \Downarrow_\mathtt{k} \nu\Sigma'.(\mathtt{ret\ v} \mid\mid \mu')\ \{[\mathtt{T}]_{\mathtt{k}\to\mathtt{k}'}\}}\ 5.5$$

$$\frac{\mathtt{e} \Downarrow \mathtt{v} \quad \nu\Sigma, \mathtt{a@}\Phi.(\mathtt{m} \mid\mid \mu \otimes \langle \mathtt{a} : \mathtt{v} \rangle) \Downarrow_\mathtt{k} \nu\Sigma'.(\mathtt{m}' \mid\mid \mu')\ \{\mathtt{T}\}}{\nu\Sigma.(\mathtt{new\ a@}\Phi := \mathtt{e\ in\ m} \mid\mid \mu) \Downarrow_\mathtt{k} \nu\Sigma'.(\mathtt{m}' \mid\mid \mu')\ \{\mathtt{new_k[a]}, \mathtt{T}\}}\ 5.6$$

**Figure 5.** SL dynamic semantics

is paired with the current store and executed to produce an intermediate execution state. Since this execution state is the result of an execution, its command is a `ret` with a value. This value is substituted into the second subcommand of the bind. The result of the substitution is combined with the store context and store from the intermediate execution state and then executed to produce the final execution state. Rules 5.3 and 5.4 read from and write to the memory store, respectively. Each produces a single element trace to represent that the corresponding operation has occurred. Rule 5.5 is primarily noteworthy for the change in principal from `k` in the conclusion to $\mathtt{k}'$ in the premise. The trace $[\mathtt{T}]_{\mathtt{k}\to\mathtt{k}'}$ nests the trace `T` of the subcommand of the `sudo` in square brackets indicating the principals involved and the extent of the non-informative block. Rule 5.6 conses an effect indicating the assignable declared to the front of the trace of its subcommand. The subcommand is executed in an extended memory store that includes the new assignable and its initial value.

## 3. Epistemic Logic

The logic we employ to model information flow is directly taken from DeYoung and Pfenning [13]. In addition to the cut elimination property of the substructural epistemic logic, which enables proving that undesired knowledge transfers are not derivable, 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 allows an entity to possess 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` and then lose them at the end of the `sudo` block. The principal must perform additional reads in subsequent `sudo` blocks if it needs to reacquire these secrets. 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 in which the actions of

$$\texttt{do}([_{\texttt{k}\to\texttt{k}'}) \otimes [\texttt{k}]\texttt{s}(X) \otimes [\texttt{k}']\texttt{s}(Y)$$
$$\multimap \{[\texttt{k}]\texttt{s}(X) \otimes [\texttt{k}']\texttt{s}(X * Y) \otimes \texttt{next}\} \quad (6.1)$$
$$\texttt{do}(]_{\texttt{k}'\to\texttt{k}}) \otimes [\texttt{k}']\texttt{s}(X) \multimap \{[\texttt{k}']\texttt{s}(1) \otimes \texttt{next}\} \quad (6.2)$$
$$\texttt{do}(\texttt{rd}_{\texttt{k}}[\texttt{a}]) \otimes [\texttt{k}]\texttt{s}(X) \otimes [\texttt{a}]\texttt{s}(Y)$$
$$\multimap \{[\texttt{k}]\texttt{s}(X * Y) \otimes [\texttt{a}]\texttt{s}(Y) \otimes \texttt{next}\} \quad (6.3)$$
$$\texttt{do}(\texttt{wr}_{\texttt{k}}[\texttt{a}]) \otimes [\texttt{k}]\texttt{s}(X) \otimes [\texttt{a}]\texttt{s}(Y)$$
$$\multimap \{[\texttt{k}]\texttt{s}(X) \otimes [\texttt{a}]\texttt{s}(X) \otimes \texttt{next}\} \quad (6.4)$$
$$\texttt{do}(\texttt{leak}_{\texttt{k}}(\Phi)) \multimap \{\texttt{do}(\texttt{wr}_{\texttt{k}}[\texttt{a}])\},$$
$$\text{if } \Phi \supseteq \Phi_{\texttt{a}} \text{ where } \texttt{a}@\Phi_{\texttt{a}} \in \Sigma \quad (6.5)$$
$$\texttt{next} \otimes \texttt{doTrace}(\alpha, T) \multimap \{\texttt{do}(\alpha) \otimes \texttt{doTrace}(T)\} \quad (6.6)$$

**Figure 6.** Semantic Actions

the trace are processed one after another. The remaining features of the logic are familiar from standard linear logic.

Semantic actions specify the epistemic consequences of each trace effect. The semantic actions for the effects defined in the dynamic semantics are given in Figure 6. The semantic action for each effect $\alpha$ has a $\texttt{do}(\alpha)$ on the left of the linear implication. These trace effects differ from those found in the traces in the dynamic semantics because the nested `sudo` trace is replaced by two actions making the entire trace a single list of actions. We also remove all of the effects declaring new assignables and extend the store context $\Sigma$ with these assignables. Semantic action (6.1) specifies that the effect $[_{\texttt{k}\to\texttt{k}'}$, which begins a `sudo` block by switching principal $\texttt{k}$ to $\texttt{k}'$, augments the knowledge of $\texttt{k}'$, which is initially $\texttt{s}(Y)$, with the knowledge of $\texttt{k}$ (*i.e.,* $\texttt{s}(X)$). The new knowledge of $\texttt{k}'$ is $\texttt{s}(X * Y)$, which combines the secrets of $X$ and $Y$. Semantic action (6.2) specifies that effect $]_{\texttt{k}'\to\texttt{k}}$ for returning from $\texttt{k}'$ to $\texttt{k}$ at the end of a `sudo` block erases the knowledge of $\texttt{k}'$. The knowledge $\texttt{s}(1)$ represents no knowledge. Linearity and the invariant that there is exactly one knowledge proposition $\texttt{s}(X)$ for each principal guarantee that the knowledge acquired by $\texttt{k}'$ within the `sudo` block is completely erased. The knowledge of $\texttt{k}$ is unaffected as the type returned by the `sudo` is uninformative to $\texttt{k}$.

Semantic actions (6.3) and (6.4) define the consequences of reading and writing assignables as discussed in the overview. Semantic action (6.5) replaces `leak` with one of the writes that could have occurred according to the storage context of the trace $\Sigma$. Choosing any single write that could have occurred seems insufficient, but to prove a flow of knowledge is not possible, all derivations for the trace are considered including any write that can replace the `leak`. A trace makes a flow of knowledge possible if there is any derivation in which that knowledge is derivable. This flexibility lends the logic its expressiveness. It would be more difficult to define precisely what happens as a result of a trace. Instead the logic expresses what is possible, and if no derivation can be found then the information flow is not possible in the epistemic model.

The `next` proposition enables the next trace effect to be processed forcing the effects to be consumed one after another as they appear in the trace. Semantic action (6.6) consumes a `next` and makes the next action of the trace available. 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.

The unrestricted and linear contexts of a logical sequent are written $\Gamma; \Delta$. The unrestricted context $\Gamma$ includes all of the semantic actions and will not change until we consider authorized declassification. The linear context $\Delta$ changes as we reason about the trace, but we require it to satisfy Definition 3.1.

**Definition 3.1.** *For a given storage context $\Sigma$ and set of principals $\mathbf{U}$, the state $\Delta$ is **valid** if and only if all of the following hold:*

1. *There is exactly one proposition of the form $[p]\texttt{s}(X)$ for each entity $p \in \mathbf{U} \cup \Sigma$.*
2. *There is either one* `next` *or one* `do` *but not both*
3. *There is exactly one* `doTrace`*.*

In DeYoung and Pfenning [13] 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 3.2.** ***(Rewrite Step).*** *The **rewrite step** $\Gamma; \Delta \longrightarrow \Gamma'; \Delta'$ holds if and only if there exists a derivation of the form*

$$\Gamma'; \Delta' \vdash C$$
$$\vdots$$
$$\frac{\Gamma; \Delta_2, A_2^+ \vdash C}{\Gamma; \Delta_2, [\{A_2^+\}] \vdash C}\ \{\}_L$$
$$\vdots$$
$$\frac{\Gamma; \Delta_1, [A^-] \vdash C}{\Gamma; \Delta \vdash C}\ *$$

*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.*

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

**Definition 3.3.** ***(Disclosure).*** *We say the trace $T$ **discloses** $p$ to $p'$ if for all valid states $\Delta$ such that $[p]\texttt{s}(X) \in \Delta$ there is a derivation of $\Gamma; \Delta \longrightarrow_T \Gamma'; \Delta'$ such that $[p']\texttt{s}(Y) \in \Delta'$ 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 a `leak` action, $T$ discloses $p$ to $p'$ if any valid replacement of the `leak` action with a write results in the disclosure.

## 4. Adequacy and Non-Interference

In this section we describe the main technical results about the information flow properties of SL. Some properties that are not essential for understanding the main technical development are deferred to section A of the supplementary material. To state these results precisely we must define what is considered a flow of information in SL. 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 $\texttt{k}$ computes two different values but those values appear equivalent to $\texttt{k}$ because their type is uninformative then we do not consider it a leak. We compare values up to equivalence relative to the view of a principal, which is defined to be the least congruence containing the rule in Figure 7. 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 storage contexts. The notation $\Sigma_1 \cap \Sigma_2$ indicates the storage

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

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

**Figure 7.** Equivalence

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 formal statements and proofs of these lemmas are deferred to the supplementary material.

### 4.1 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 DeYoung and Pfenning [13], states a property of our model for reasoning about what is and is not derivable.

**Theorem 4.1.** ***(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 4.1 facilitates the proofs of several lemmas about our disclosure judgment found in the supplementary material as well as the following lemma, which is used to prove properties of the traces of well-typed programs.

**Lemma 4.1.** ***(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 4.1 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 $\Gamma^*;\Delta^* \longrightarrow_{\mathtt{T}_2} \Gamma';\Delta'$. □

### 4.2 Adequacy

The adequacy theorem proves that any information disclosed according to the dynamic semantics of SL is also disclosed in the logic according to the formal definition given in the previous section, thereby establishing a correspondence between the semantic actions and the actual information flows in the program.

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 SL yields the following adequacy theorem.

**Theorem 4.2.** ***(Adequacy).*** *If*
$\Sigma;\Sigma';\cdot \vdash \mathtt{m} \approx_{\mathtt{k}} \mathtt{m}' \div_{\mathtt{k}} \mathtt{A}@[\Phi^{\mathtt{r}},\Phi^{\mathtt{w}}]$*,* $\mu : \Sigma$*,* $\mu' : \Sigma'$*,*
$\nu\Sigma.(\mathtt{m} \mid\mid \mu) \Downarrow_{\mathtt{k}} \nu\Sigma_{\mathtt{f}}.(\mathtt{ret}\ \mathtt{v} \mid\mid \mu_{\mathtt{f}})\ \{\mathtt{T}\}$*, and*
$\nu\Sigma'.(\mathtt{m}' \mid\mid \mu') \Downarrow_{\mathtt{k}} \nu\Sigma'_{\mathtt{f}}.(\mathtt{ret}\ \mathtt{v}' \mid\mid \mu'_{\mathtt{f}})\ \{\mathtt{T}'\}$ *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 bind command. The other cases for when the command is a bind are similar, and the remaining cases are straightforward.

In the bind case, $\mathtt{m} = \mathtt{x} \leftarrow \mathtt{e}_1;\ \mathtt{m}_2$ and $\mathtt{m}' = \mathtt{x} \leftarrow \mathtt{e}'_1;\ \mathtt{m}'_2$ such that:

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

By Lemma A.2 we have $\mathtt{cmd}(\mathtt{m}_1) \approx_{\mathtt{k}} \mathtt{cmd}(\mathtt{m}'_1) : \mathtt{cmd}_{\mathtt{k}}[\Phi^{\mathtt{r}}_1,\Phi^{\mathtt{w}}_1](\mathtt{B})$ as $\mathtt{e}_1 \approx_{\mathtt{k}} \mathtt{e}'_1$. By inversion, either $\mathtt{m}_1 \approx_{\mathtt{k}} \mathtt{m}'_1$ or $\mathtt{cmd}_{\mathtt{k}}[\Phi^{\mathtt{r}}_1,\Phi^{\mathtt{w}}_1](\mathtt{B}) \nearrow \Phi$ 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 for extending disclosures from a prefix of a trace to a whole trace 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 for composing disclosures yields $\mathtt{T} = \mathtt{T}_1, \mathtt{leak}_\mathtt{k}(\Phi_1^\mathtt{w}), \mathtt{T}_2$ and $\mathtt{T}' = \mathtt{T}_1', \mathtt{leak}_\mathtt{k}(\Phi_1^\mathtt{w}), \mathtt{T}_2'$ disclose $\mathtt{c}$ to $\mathtt{k}$ completing this case. □

The theorem states that if there is a flow of information in SL then there is a corresponding disclosure in the trace. 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 SL. Therefore, to prove a NI result it is sufficient to prove no disclosure is derivable in the logic.

### 4.3 Non-Interference

To demonstrate the utility of this methodology we now present the proof of a NI result for SL. The adequacy theorem reduces proving a NI result about SL 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 4.2.** ***(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 bind. They critically depend on Lemma 4.1 to reason about the disclosures of the two subcommands independently.

In these cases we have $\mathtt{m} = \mathtt{x} \leftarrow \mathtt{e}_1;\ \mathtt{m}_2$ and:

- $\mathtt{e}_1 \Downarrow \mathtt{cmd}(\mathtt{m}_1)$
- $\nu\Sigma.(\mathtt{m}_1 \mid\mid \mu) \Downarrow_\mathtt{k} \nu\Sigma_1.(\mathtt{ret\ v}_1 \mid\mid \mu_1)\ \ \{\mathtt{T}_1\}$
- $\nu\Sigma_1.([\mathtt{v}_1/x]\mathtt{m}_2 \mid\mid \mu_1) \Downarrow_\mathtt{k} \nu\Sigma_\mathtt{f}.(\mathtt{ret\ v} \mid\mid \mu_\mathtt{f})\ \ \{\mathtt{T}_2\}$
- $\mathtt{T} = \mathtt{T}_1, \mathtt{leak}_\mathtt{k}(\Phi_1^\mathtt{w}), \mathtt{T}_2$

For the first case, we assume $\mathtt{T}$ discloses $\mathtt{a}$ to $\mathtt{k}$. Therefore, we may apply Lemma 4.1 to conclude that there exists an entity $p$ such that $\mathtt{T}_1, \mathtt{leak}_\mathtt{k}(\Phi_1^\mathtt{w})$ 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 $\Phi^\mathtt{r} \subseteq \Phi_2^\mathtt{r} \subseteq \Phi_\mathtt{b}$ and $\mathtt{k} \in \Phi_\mathtt{b}$. If the derivation of disclosure actually used the $\mathtt{leak}_\mathtt{k}(\Phi_1^\mathtt{w})$ 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 states 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 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 NI result.

**Theorem 4.3.** ***(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.* NI is a corollary of Theorem 4.2 and Lemma 4.2 by just applying the latter to the disclosures in the conclusions of the former. □

The NI result may be better understood by considering the contrapositives of the two parts. If all assignables with $\mathtt{k}$ in their permission set are equivalent from the view of $\mathtt{k}$ then the resulting values will also be equivalent from the view of $\mathtt{k}$, 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 $\mathtt{k}$ then the final values of the assignable will also be equivalent. Therefore, if only an assignable that does not have $\mathtt{k}$ in its permission set is modified then it will not interfere with the final value observed by $\mathtt{k}$, 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.

## 5. Security language with declassification SLD

The expressiveness of the logic facilitates the addition of authorized declassification to SL. For simplicity, we focus on authorization proofs that take the form of a certificate verifying that the appropriate principal has authenticated. 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.

### 5.1 Static and dynamic semantics

The new commands of SLD for authentication and declassification are given in Figure 8. The `auth` command verifies that the indicated principal has authenticated. Therefore, the type is an option as the verification may fail. The type $\mathtt{iam}(\mathtt{k})$ is the type of a proof that $\mathtt{k}$ has authenticated. The read level of the command is arbitrary as we assume the database of authentication credentials is readable by all principals. Intuitively, the write level is also unrestricted. Rule 8.2 is similar to rule 3.5 except that instead of requiring that $\mathtt{k}'$ is in the permission set $\Phi$, the `decl` command requires an authorization proof in the form of an authentication certificate of one of the principals in $\Phi$. This certificate is obtained from a successful execution 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, $\mathtt{S}$, which corresponds to the authentication database. Explicitly parameterizing by $\mathtt{S}$ 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 $\mathtt{S}$. Rule 8.5 is also very similar to rule 5.3.

$$\frac{}{\Sigma;\Gamma \vdash \mathtt{auth}[\mathtt{k}] \div_{\mathtt{k}'} \mathtt{iam}(\mathtt{k})\ \mathtt{option@}[\Phi^{\mathtt{r}}, \Phi^{\mathtt{w}}]}\ 8.1$$

$$\frac{\mathtt{a} : \mathtt{A@}\Phi \in \Sigma \quad \Sigma;\Gamma \vdash \mathtt{e} : \mathtt{iam}(\mathtt{k}) \quad \mathtt{k} \in \Phi}{\Sigma;\Gamma \vdash \mathtt{decl}[\mathtt{a}](\mathtt{e}) \div_{\mathtt{k}'} \mathtt{A@}[\Phi^{\mathtt{r}}, \Phi^{\mathtt{w}}]}\ 8.2$$

$$\frac{\mathtt{S}(\mathtt{k}') = \mathtt{NONE}}{\nu\Sigma.(\mathtt{auth}[\mathtt{k}'] \mid\mid \mu) \Downarrow^{\mathtt{S}}_{\mathtt{k}} \nu\Sigma.(\mathtt{ret}\ \mathtt{NONE} \mid\mid \mu)\ \ \{\varepsilon\}}\ 8.3$$

$$\frac{\mathtt{S}(\mathtt{k}') = \mathtt{SOME}\ \varphi}{\nu\Sigma.(\mathtt{auth}[\mathtt{k}'] \mid\mid \mu) \Downarrow^{\mathtt{S}}_{\mathtt{k}} \nu\Sigma.(\mathtt{ret}\ \mathtt{SOME}\ \varphi \mid\mid \mu)\ \ \{\mathtt{auth}_{\mathtt{k}}[\varphi][\mathtt{k}']\}}\ 8.4$$

$$\frac{\mathtt{e} \Downarrow \varphi \quad \mu(\mathtt{a}) = \mathtt{v}}{\nu\Sigma.(\mathtt{decl}[\mathtt{a}](\mathtt{e}) \mid\mid \mu) \Downarrow^{\mathtt{S}}_{\mathtt{k}} \nu\Sigma.(\mathtt{ret}\ \mathtt{v} \mid\mid \mu)\ \ \{\mathtt{decl}_{\mathtt{k}}[\varphi][\mathtt{a}]\}}\ 8.5$$

$$\mathtt{do}(\mathtt{auth}_{\mathtt{k}}[\varphi][\mathtt{k}']) \multimap \{!\mathtt{Auth}(\mathtt{k}')\} \tag{8.6}$$

$$\mathtt{do}(\mathtt{decl}_{\mathtt{k}}[\varphi][\mathtt{a}]) \otimes [\mathtt{k}]\mathtt{s}(X) \otimes [\mathtt{a}]\mathtt{s}(Y) \otimes \mathtt{Auth}(\mathtt{k}') \multimap \{[\mathtt{k}]\mathtt{s}(X * Y) \otimes [\mathtt{a}]\mathtt{s}(Y) \otimes \mathtt{next}\} \tag{8.7}$$

**Figure 8.** SLD rules for authorized declassification and additional semantic actions

The only differences are that the authorization proof must be first evaluated and the result is included in the new trace effect.

Semantic action (8.6) for authorization adds a persistent proposition to the context indicating that the authenticated principal has authorized declassification. Semantic action (8.7) for declassification is similar to semantic action (6.3) 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 authorization proof. The value $\varphi$ witnesses that the authorization proof exists in the trace of any well-typed program because $\varphi$ is such a proof.

### 5.2 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 for SL still hold for SLD though possibly in a slightly modified form.

Theorem 4.1 that states the possible rewrite steps still holds with additional cases for the new actions. Theorem 4.2 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.

The statement of the theorem distinguishes the assignable `a` that is declassified from the assignable `b` that differs in the initial store because any assignable that is disclosed to the declassified assignable may be leaked through the declassification. To prevent the value of an assignable `c` from being leaked by the declassification of `a`, the permission set $\Phi_{\mathtt{a}}$ must not be contained in the permission set $\Phi_{\mathtt{c}}$. If $\Phi_{\mathtt{a}} \not\subseteq \Phi_{\mathtt{c}}$ then `c` cannot be disclosed to `a` without an additional declassification. Alternatively, simultaneously enforcing integrity would allow us to avoid undesired leaks by making all declassified assignables have high integrity and all assignables that should not be declassified have low integrity.

**Theorem 5.1.** ***(Adequacy).*** *If*
$\Sigma;\Sigma';\cdot \vdash \mathtt{m} \approx_{\mathtt{k}} \mathtt{m}' \div_{\mathtt{k}} \mathtt{A@}[\Phi^{\mathtt{r}}, \Phi^{\mathtt{w}}]$, $\mu : \Sigma$, $\mu' : \Sigma'$,
$\nu\Sigma.(\mathtt{m} \mid\mid \mu) \Downarrow^{\mathtt{S}}_{\mathtt{k}} \nu\Sigma_{\mathtt{f}}.(\mathtt{ret}\ \mathtt{v} \mid\mid \mu_{\mathtt{f}})\ \ \{\mathtt{T}\}$, *and*
$\nu\Sigma'.(\mathtt{m}' \mid\mid \mu') \Downarrow^{\mathtt{S}}_{\mathtt{k}} \nu\Sigma'_{\mathtt{f}}.(\mathtt{ret}\ \mathtt{v}' \mid\mid \mu'_{\mathtt{f}})\ \ \{\mathtt{T}'\}$ *then*

1. *Either* $\mathtt{v} \approx_{\mathtt{k}} \mathtt{v}'$ *or there exists* `b` *such that either*
   - $\mathtt{b} \in \Sigma \cap \Sigma'\restriction_{\mathtt{k}}$ *or*
   - $\exists \mathtt{a} \in \Sigma \cap \Sigma'\restriction_{\mathtt{k}'}$, $\mathtt{decl}_{\mathtt{k}}[\varphi][\mathtt{a}] \in \mathtt{T}$ *and* $\mathtt{T}'$, $\mathtt{S}(\mathtt{k}') = \mathtt{SOME}\ \varphi$, *and* `T` *and* $\mathtt{T}'$ *disclose* `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 either*
   - $\mathtt{b} \in \Sigma \cap \Sigma'\restriction_{\mathtt{k}}$ *or*
   - $\exists \mathtt{a} \in \Sigma \cap \Sigma'\restriction_{\mathtt{k}'}$, $\mathtt{decl}_{\mathtt{k}}[\varphi][\mathtt{a}] \in \mathtt{T}$ *and* $\mathtt{T}'$, $\mathtt{S}(\mathtt{k}') = \mathtt{SOME}\ \varphi$, *and* `T` *and* $\mathtt{T}'$ *disclose* `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 bind to demonstrate how it must be changed to accommodate authorized declassification.

In the bind case, $\mathtt{m} = \mathtt{x} \leftarrow \mathtt{e}_1;\ \mathtt{m}_2$ and $\mathtt{m}' = \mathtt{x} \leftarrow \mathtt{e}'_1;\ \mathtt{m}'_2$ such that:

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

By Lemma A.2 we have $\mathtt{cmd}(\mathtt{m}_1) \approx_{\mathtt{k}} \mathtt{cmd}(\mathtt{m}'_1) : \mathtt{cmd}_{\mathtt{k}}[\Phi^{\mathtt{r}}_1, \Phi^{\mathtt{w}}_1](\mathtt{B})$ as $\mathtt{e}_1 \approx_{\mathtt{k}} \mathtt{e}'_1$. By inversion either $\mathtt{m}_1 \approx_{\mathtt{k}} \mathtt{m}'_1$ or $\mathtt{cmd}_{\mathtt{k}}[\Phi^{\mathtt{r}}_1, \Phi^{\mathtt{w}}_1](\mathtt{B}) \nearrow \Phi$ 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 $\exists \mathtt{a} \in \Sigma \cap \Sigma'\restriction_{\mathtt{k}'}$, $\mathtt{decl}_{\mathtt{k}}[\varphi][\mathtt{a}] \in \mathtt{T}_1$ and $\mathtt{T}'_1$, $\mathtt{S}(\mathtt{k}') = \mathtt{SOME}\ \varphi$, and $\mathtt{T}_1$ and $\mathtt{T}'_1$ 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 $\exists \mathtt{a} \in \Sigma \cap \Sigma'\restriction_{\mathtt{k}'}$, $\mathtt{decl}_{\mathtt{k}}[\varphi][\mathtt{a}] \in \mathtt{T}_2$ and $\mathtt{T}'_2$ $\mathtt{S}(\mathtt{k}') = \mathtt{SOME}\ \varphi$, and $\mathtt{T}_2$ and $\mathtt{T}'_2$ disclose `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 $\exists \mathtt{a} \in \Sigma \cap \Sigma'\restriction_{\mathtt{k}'}$, $\mathtt{decl}_{\mathtt{k}}[\varphi][\mathtt{a}] \in \mathtt{T}_1$ and $\mathtt{T}'_1$ $\mathtt{S}(\mathtt{k}') = \mathtt{SOME}\ \varphi$, and $\mathtt{T}_1$ and $\mathtt{T}'_1$ disclose `c` to `a` such that $\mu(c) \not\approx_{\mathtt{k}} \mu'(c)$ and $\mathtt{T}_1$ and $\mathtt{T}'_1$ 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`, which necessitates 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 5.1.** ***(Typing Soundness).*** *If* $\Sigma; \cdot \vdash \mathtt{m} \div_{\mathtt{k}} \mathtt{A@}[\Phi^{\mathtt{r}}, \Phi^{\mathtt{w}}]$, $\mu : \Sigma$ *and* $\nu\Sigma.(\mathtt{m} \mid\mid \mu) \Downarrow^{\mathtt{S}}_{\mathtt{k}} \nu\Sigma_{\mathtt{f}}.(\mathtt{ret\ v} \mid\mid \mu_{\mathtt{f}})\ \ \{\mathtt{T}\}$ *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 $\Phi$ 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 5.2.** ***(Authorized Declassification).*** *If* $\Sigma; \Sigma'; \cdot \vdash \mathtt{m} \approx_{\mathtt{k}} \mathtt{m}' \div_{\mathtt{k}} \mathtt{A@}[\Phi^{\mathtt{r}}, \Phi^{\mathtt{w}}]$, $\mu : \Sigma$, $\mu' : \Sigma'$, $\nu\Sigma.(\mathtt{m} \mid\mid \mu) \Downarrow^{\mathtt{S}}_{\mathtt{k}} \nu\Sigma_{\mathtt{f}}.(\mathtt{ret\ v} \mid\mid \mu_{\mathtt{f}})\ \ \{\mathtt{T}\}$, *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. *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* $\Phi_{\mathtt{a}} \supseteq \Phi_{\mathtt{b}}$ *or there is a proof authorizing the declassification of an assignable* c *with permission set* $\Phi_{\mathtt{c}} \subseteq \Phi_{\mathtt{a}}$

*Proof.* The presence of declassification actions in the trace in Theorem 5.1 guarantees the existence of corresponding authorization proofs when composed with Lemma 5.1. □

## 6. Related Work

Early work on language-based IFS is summarized in Sabelfeld and Myers [36]. The most closely related work is Crary et al. [11]. Our type system is based on theirs with the notable exception of our syntactic representation of non-informative upcalls as principal switches. This modification facilitates forming traces to connect the programs with our epistemic logic model of information flow.

In Balliu et al. [5], 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 Balliu et al. [6], they extend this work by employing an SMT solver along with concrete and symbolic execution to verify the enforcement of the policies. This work also uses 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 NI 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 NI or only authorized declassifications so that any program that is well-typed will be safe to execute without further analysis.

Halpern and O'Neill [18] employs the interpreted systems formalism [14] for reasoning about secrecy in multiagent systems whereas we employ a static language-based approach. Secure multi-execution (SME) [12] dynamically enforces IFS. Rather than just monitoring the behavior to prevent insecurities, SME repairs them. In the Dependency Core Calculus (DCC) [2], the monadic type seals values with a security level. Following Crary et al. [11], the monad we employ encapsulates storage effects and specifies bounds on the security levels read and written. Crary et al. [11] encodes DCC dependency analysis into this style of type system.

Sabelfeld and Sands [37] surveys work on declassification and provides dimensions of declassification for analyzing these approaches. SLD 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 or by enforcing integrity constraints on declassified assignables. If only part of a value should be declassified, such as the last four digits of a credit card, then this part can be stored in a separate assignable with a less restrictive permission set. Zdancewic and Myers [40] define robust declassification to prevent an attacker from obtaining more information than intended through declassifications. Our approach for encoding what-dimension declassification can address this issue at the cost of dealing with more cumbersome fine-grained security levels or integrity. Control of *who* releases information is enforced through our protocol that requires authentication certificates to form authorization proof. 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.

Montagu et al. [26] defines *label algebras* to compare the expressiveness of various IF languages with labels. Our approach roughly corresponds to their reader model where each label is a set of principals and each authority is the set of principals that have authenticated. The authorization 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 Banerjee et al. [7]. Modeling information flow in systems with declassification is addressed in Zdancewic and Myers [40].

Tse and Zdancewic [39] proposes a method for enforcing information flow policies with run-time principals. 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. Bohannon et al. [10] defines NI in reactive systems such as web browsers and develops a technique for proving NI properties in this setting.

Other approaches support information flow reasoning in existing functional languages. For example, Li and Zdancewic [21], Li and Zdancewic [22], and Russo et al. [35] demonstrate how to reason about information flow in Haskell, and Pottier and Simonet [34] addresses 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.

Nanevski et al. [30] models information flow properties including NI 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 Morgenstern and Licata [27], which modifies the dependently-typed language Agda [31] for this purpose, and in Jia and Zdancewic [19], which employs the access control language AURA [20]. 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.

## 7. 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 consulting an authentication database. 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 is natural to also express whether a leak is authorized by the policy if the policy is expressed in the logic. Properly representing these disclosures requires more precise modeling of 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 NI-like 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 section B of the supplementary material briefly addresses integrity as well. Our approach to preserving the integrity of data is the straightforward dual of confidentiality [9], 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.

Liu et al. [23] proposes the Fabric system for building secure distributed information systems. Fabric builds on the Jif programming language [29]. Distributed information systems would be an interesting extension of our work. Integrating this work with the ML5 system for distributed systems proposed in Murphy [28] is one possible approach for advancing in this direction.

## 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 Crary et al. [11]. □

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

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

1. *If* $\Sigma; \Sigma'; \cdot \vdash \mathtt{m} \approx_{\mathtt{k}} \mathtt{m}' \div_{\mathtt{k}} \mathtt{A}@[\Phi^{\mathtt{r}}, \Phi^{\mathtt{w}}]$,
   $\Sigma; \Sigma' \vdash \mu \approx_{\mathtt{k}} \mu' : \Sigma \cap \Sigma'\restriction_{\mathtt{k}}$,
   $\nu\Sigma.(\mathtt{m} \mid\mid \mu) \Downarrow_{\mathtt{k}} \nu\Sigma_{\mathtt{f}}.(\mathtt{ret}\ \mathtt{v} \mid\mid \mu_{\mathtt{f}})\ \ \{\mathtt{T}\}$,
   *and* $\nu\Sigma'.(\mathtt{m}' \mid\mid \mu') \Downarrow_{\mathtt{k}} \nu\Sigma'_{\mathtt{f}}.(\mathtt{ret}\ \mathtt{v}' \mid\mid \mu'_{\mathtt{f}})\ \ \{\mathtt{T}'\}$
   *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. $\Sigma; \Sigma'; \cdot \ \vdash\ \mathtt{e}_1 \approx_{\mathtt{k}} \mathtt{e}'_1 : \mathtt{cmd}_{\mathtt{k}}[\Phi^{\mathtt{r}}_1, \Phi^{\mathtt{w}}_1](\mathtt{B})$ and $\Sigma; \Sigma'; \mathtt{x} : \mathtt{B} \ \vdash\ \mathtt{m}_2 \approx_{\mathtt{k}} \mathtt{m}'_2 \div_{\mathtt{k}} \mathtt{A}@[\Phi^{\mathtt{r}}_2, \Phi^{\mathtt{w}}_2]$
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 $\mathtt{cmd}_{\mathtt{k}}[\Phi^{\mathtt{r}}_1, \Phi^{\mathtt{w}}_1](\mathtt{B}) \nearrow \Phi$ and $\mathtt{k} \notin \Phi$. Again by inversion, we have $\mathtt{B} \nearrow \Phi_{\mathtt{B}}$ and $\Phi_{\mathtt{B}} \cup \Phi^{\mathtt{w}}_1 \subseteq \Phi$. Therefore, $\mathtt{k} \notin \Phi^{\mathtt{w}}_1$ and $\mathtt{k} \notin \Phi_{\mathtt{B}}$. So we have $\nu\Sigma.(\mathtt{m}_1 \mid\mid \mu) \Downarrow_{\mathtt{k}} \nu\Sigma_1.(\mathtt{ret}\ \mathtt{v}_1 \mid\mid \mu_1)\ \ \{\mathtt{T}_1\}$ and $\nu\Sigma'.(\mathtt{m}'_1 \mid\mid \mu') \Downarrow_{\mathtt{k}} \nu\Sigma'_1.(\mathtt{ret}\ \mathtt{v}'_1 \mid\mid \mu'_1)\ \ \{\mathtt{T}'_1\}$ 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 $\mathtt{k} \notin \Phi^{\mathtt{w}}_1$, $\Sigma_1; \Sigma'_1 \vdash \mu_1 \approx_{\mathtt{k}} \mu'_1 : \Sigma_1 \cap \Sigma'_1\restriction_{\mathtt{k}}$ 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 3.3. They are all proved using Theorem 4.1.

**Lemma A.3.** ***(Disclosure Extension).*** *If*
$\Sigma; \cdot \vdash \mathtt{m} \div_{\mathtt{k}} \mathtt{A}@[\Phi^{\mathtt{r}}, \Phi^{\mathtt{w}}]$,
$\nu\Sigma.(\mathtt{m} \mid\mid \mu) \Downarrow_{\mathtt{k}} \nu\Sigma_{\mathtt{f}}.(\mathtt{ret}\ \mathtt{v} \mid\mid \mu_{\mathtt{f}})\ \ \{\mathtt{T}\}$, $\mathtt{T} = \mathtt{T}_1, \mathtt{T}_2$, *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*
$\Sigma; \cdot \vdash \mathtt{m} \div_{\mathtt{k}} \mathtt{A}@[\Phi^{\mathtt{r}}, \Phi^{\mathtt{w}}]$,
$\nu\Sigma.(\mathtt{m} \mid\mid \mu) \Downarrow_{\mathtt{k}} \nu\Sigma_{\mathtt{f}}.(\mathtt{ret}\ \mathtt{v} \mid\mid \mu_{\mathtt{f}})\ \ \{\mathtt{T}\}$, $\mathtt{T} = \mathtt{T}_1, \mathtt{T}_2$,
$\mathtt{T}_1$ *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*
$\Sigma; \cdot \vdash \mathtt{m} \div_{\mathtt{k}} \mathtt{A}@[\Phi^{\mathtt{r}}, \Phi^{\mathtt{w}}]$,
$\nu\Sigma.(\mathtt{m} \mid\mid \mu) \Downarrow_{\mathtt{k}} \nu\Sigma_{\mathtt{f}}.(\mathtt{ret}\ \mathtt{v} \mid\mid \mu_{\mathtt{f}})\ \ \{\mathtt{T}\}$, $\mathtt{T}$ *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. 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 DeYoung and Pfenning [13] are presented in Figure 9. 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 Andreoli [3]. 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

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

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

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

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

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

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

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

Negative Connectives

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

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

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

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

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

**Figure 9.** Inference rules for the weakly focused sequent calculus

## References


[1] 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)*, 15(4):706–734, 1993.

[2] 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)*, pages 147–160. ACM Press, 1999.

[3] J. Andreoli. Logic programming with focusing proofs in linear logic. *Journal of Logic and Computation*, 2(3):297–347, 1992.

[4] K. Avijit, A. Datta, and R. Harper. Distributed programming with distributed authorization. In *Proceedings of the 5th ACM SIGPLAN workshop on Types in language design and implementation*, pages 27–38. ACM, 2010.

[5] 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.

[6] M. Balliu, M. Dam, and G. L. Guernic. Encover: Symbolic exploration for information flow security. In *Computer Security Foundations Symposium (CSF), 2012 IEEE 25th*, pages 30–44. IEEE, 2012.

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

[8] H. Bar-El, H. Choukri, D. Naccache, M. Tunstall, and C. Whelan. The sorcerer's apprentice guide to fault attacks. *Proceedings of the IEEE*, 94(2):370–382, 2006.

[9] K. J. Biba. Integrity considerations for secure computer systems. *Proceedings of the 4th annual symposium on Computer architecture*, 5(7):135–140, 1977.

[10] 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*, CCS '09, pages 79–90, New York, NY, USA, 2009. ACM.

[11] K. Crary, A. Kliger, and F. Pfenning. A monadic analysis of information flow security with mutable state. *Journal of Functional Programming*, 15(2):249–291, 2005.

[12] D. Devriese and F. Piessens. Noninterference through secure multi-execution. In *Security and Privacy (SP), 2010 IEEE Symposium on*, pages 109–124. IEEE, 2010.

[13] 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.

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

[15] D. Garg and F. Pfenning. Stateful authorization logic–proof theory and a case study. *Journal of Computer Security*, 20(4):353–391, 2012.

[16] J. A. Goguen and J. Meseguer. Security Policies and Security Models. In *IEEE Symposium on Security and Privacy*, pages 11–20, 1982.

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

[18] J. Halpern and K. O'Neill. Secrecy in multiagent systems. *ACM Transactions on Information and System Security (TISSEC)*, 12(1):5, 2008.

[19] 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*, PLAS '09, pages 17–29, New York, NY, USA, 2009. ACM.

[20] 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*, ICFP '08, pages 27–38, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-919-7.

[21] P. Li and S. Zdancewic. Encoding information flow in haskell. In *Proceedings of the 19th IEEE workshop on Computer Security Foundations*, CSFW '06, Washington, DC, USA, 2006. IEEE Computer Society. ISBN 0-7695-2615-2.

[22] P. Li and S. Zdancewic. Arrows for secure information flow. *Theor. Comput. Sci.*, 411(19):1974–1994, 2010.

[23] 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*, SOSP '09, pages 321–334, New York, NY, USA, 2009. ACM.

[24] E. Moggi. Computational lambda-calculus and monads. In *Logic in Computer Science, 1989. LICS'89, Proceedings., Fourth Annual Symposium on*, pages 14–23. IEEE, 1989.

[25] E. Moggi. *An abstract view of programming languages*. University of Edinburgh, Department of Computer Science, Laboratory for Foundations of Computer Science, 1990.

[26] B. Montagu, B. C. Pierce, and R. Pollack. A theory of information-flow labels. In *Proceedings of the 2013 IEEE Computer Security Foundations Symposium*, June 2013.

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

[28] T. Murphy, VII. *Modal Types for Mobile Code*. PhD thesis, Carnegie Mellon, January 2008. Available as technical report CMU-CS-08-126.

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

[30] 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*, pages 165 –179, May 2011.

[31] U. Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Department of Computer Science and Engineering, Chalmers University of Technology, SE-412 96 Göteborg, Sweden, September 2007.

[32] S. L. Peyton Jones and P. Wadler. Imperative functional programming. In *Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 71–84. ACM, 1993.

[33] F. Pfenning and R. Davies. A judgmental reconstruction of modal logic. *Mathematical structures in computer science*, 11(04):511–540, 2001.

[34] F. Pottier and V. Simonet. Information flow inference for ml. In *Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '02, pages 319–330, New York, NY, USA, 2002. ACM. ISBN 1-58113-450-9.

[35] A. Russo, K. Claessen, and J. Hughes. A library for light-weight information-flow security in haskell. *SIGPLAN Not.*, 44(2):13–24, Sept. 2008. ISSN 0362-1340.

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

[37] A. Sabelfeld and D. Sands. Declassification: Dimensions and principles. *Journal of Computer Security*, 17(5):517–548, 2009.

[38] F. B. Schneider. Enforceable security policies. *ACM Transactions on Information and System Security*, 3(1):30–50, 2000.

[39] S. Tse and S. Zdancewic. Run-time principals in information-flow type systems. *ACM Trans. Program. Lang. Syst.*, 30(1), 2007.

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