Modular Implementation and Formalization of Dynamic Policies Work In Progress

Antonio Zegarelli
IMDEA Software Institute
Madrid, Spain

Niki Vazou
IMDEA Software Institute
Madrid, Spain

Marco Guarnieri
IMDEA Software Institute
Madrid, Spain

ABSTRACT

In Information Flow Control (IFC) expressiveness, i.e. being able to model multiple scenarios, is a crucial aspect, especially in the context of dynamically evolving security requirements. Those dynamic scenarios introduce complexities due to varying security interpretations. Broberg et al. identified "facets of dynamic policies", patterns of information flow that may be considered secure or insecure depending on the context. Therefore, our research aims to establish a robust framework that facilitates the design and implementation of the different interpretations of facets within a single, modular enforcement mechanism. We propose, in Haskell, an abstract definition of a monadic IFC mechanism, that, based on the instantiation, can account for the different interpretations of facets. Our ongoing work involves the formalization of the respective security conditions, the study of their combination and a LiquidHaskell proof mechanization. This with the aspiration to improve the reasoning about dynamic policies and their associated facets, especially in the case of future extensions.

KEYWORDS

Information Flow Control, Language-based Security, Dynamic Policies, Haskell, Formal Verification

1 INTRODUCTION

Information Flow Control (IFC) is a set of techniques for ensuring the secure exchange of information in a system. For example, in a system that processes the sensitive information of patients in a hospital, we want to ensure that the patients' information flows to the doctors, but not to unauthorized entities.

Stating which entities are authorized to access the information is done by defining a security policy which is said *static* if it doesn't change over time, or *dynamic* otherwise. For example, on the left of Table 1 we have an example, presented in [4], where information, at line 1, is allowed to flow from the patient to the hospital, denoted as `Patient` $\longrightarrow$ `Hospital`. However, after the patient has left the hospital, denoted as `Patient` $\nrightarrow$ `Hospital` at line 3, and a new doctor joins the hospital, denoted as `Hospital` $\longrightarrow$ `Doc` at line 4, the information should not flow to the doctor.

Interestingly, the semantics of IFC security properties, when policies change over time, are not well defined. For example, in the right part of Table 1 we have another example from [4], with the same structure of information flows, but different security interpretations. In this case, line 1 allows information to flow from the input to the sanitizer, denoted as `Input` $\longrightarrow$ `Sanitizer`. Once the input has been sanitized this flow is stopped, denoted as `Input` $\nrightarrow$ `Sanitizer` at line 3, and the information can flow to the database, denoted as `Sanitizer` $\longrightarrow$ `DB` at line 4. Now the input should be allowed to be stored into the database.

Table 1: Examples of time transitive flow

Forbidden	Allowed
1 -- Patient $\longrightarrow$ Hospital	1 -- Input $\longrightarrow$ Sanitizer
2 h <- receive p	2 s <- sanitize input
3 -- Patient $\nrightarrow$ Hospital	3 -- Input $\nrightarrow$ Sanitizer
4 -- Hospital $\longrightarrow$ Doc	4 -- Sanitizer $\longrightarrow$ DB
5 share h Doc	5 store s

From these two examples, we see that the security properties of programs with dynamic policies can have different interpretations. Broberg et al. [4] call such interpretations *facets of dynamic policies*. In the right of Table 1, we have a *time-transitive flow* facet, where information can flow from the input to the database via the sanitizer, even though there is no moment in time where the flow from the input to the database is directly enabled. In the left of Table 1, we have another facet, where time-transitive flow is forbidden. As another facet, we can consider the *replaying* facet where the information released is permanent, and the *non-replaying* facet where the information released is temporary.

In this work, we aim to establish a framework that enables the design and implementation of different facet interpretations within a single enforcement mechanism. In particular, in § 4 we define a family of security conditions that can be used to define different security interpretations. All these conditions share the same structure, which can be instantiated to different facets.

These security conditions are enforced by an IFC system, which generalizes the SLIO system [5]. Our IFC system is a label based, monadic system, where the labels are used to track the information flows and monadic operations are used to enforce the security conditions.

In § 2 we present the core calculus formalism of the system, which is parametric with respect to an interface that in § 5 is instantiated to different facets. In § 7 we present the implementation of the system in Haskell, which uses Haskell's type class mechanism to define the interface and the instances for the facets.

Finally, we use the theorem prover of Liquid Haskell to mechanize the metaproof that the system enforces the security conditions, for each of the facets. Such mechanizations are difficult, due to the hight complexity of the dynamic policies. To the best of our knowledge, there is no mechanized security proof for IFC systems enforcing dynamic policies, while the paper and pencil proof of SLIO [5] is incorrect since the security metafunction (namely the multilevel erasure) was wrongly defined (§ 4.1) and as a result the implementation of the system is not correct with respect to the definition of the security condition, as we discuss in § 6.2. Our mechanization is a work in progress. Concretely, we have started the mechanization for the time-transitive flow facet, but we have

Constants	c	::=	$\texttt{True} \mid \texttt{False} \mid i \mid () \mid \bullet$
Values	v	::=	$c \mid rel \mid ps \mid l \mid x \mid \lambda x.e \mid \texttt{fix}\ e \mid \texttt{M}\ e \mid \texttt{LB}\ l\ e\ i$
Expressions	e	::=	$v \mid e\ e \mid \texttt{if}\ e\ \texttt{then}\ e\ \texttt{else}\ e$
monadic		$\mid$	$\texttt{ret}\ e \mid e \gg\!= e$
labeled expressions		$\mid$	$\texttt{label}\ e\ e \mid \texttt{unlabel}\ e$
toLabeled		$\mid$	$\texttt{toLabeled}\ l\ e \mid \texttt{toLabRet}\ \Sigma\ l\ e$
policy expressions		$\mid$	$\texttt{setRel}\ e \mid \texttt{getRel}$
Labels	l	::=	PARAMETRIC
System refined policy	ps	::=	PARAMETRIC
User defined policy	rel	::=	PARAMETRIC
Relation	R	::=	$\texttt{lrt}(\Sigma, l, l)$
Label Ids	sid	::=	$\emptyset \mid (l, i); sid$
State	Σ	::=	$\langle sid, ls, ps, rel \rangle$
Label Set	ls	::=	$\emptyset \mid (l, i); ls$
Configuration	C	::=	$\langle \Sigma \mid e \rangle$
Instance	I	::=	$\texttt{opLabel}(\Sigma, l, i)$ & $\texttt{opUnlabel}(\Sigma, l, i)$
		&	$\texttt{opToLabRet}(\Sigma, \Sigma, l, i)$ & $\texttt{guard}(\Sigma, l)$
		&	$\texttt{stateGuard}(\Sigma, \Sigma)$ & $\texttt{opState}(\Sigma, e)$
		&	$\texttt{policy}(\Sigma, l, i, l)$

Figure 1: Syntax of λ^{DP}.

set up the system in a modular way such that it can be extended to other facets.

In summary, the main contributions of this work are:

- We formalize a core calculus for a parametric IFC system (§ 2), instantiate it to different facets (§ 5), and implement it in Haskell (§ 7) using Haskell's type class mechanism.
- Define security conditions (§ 4) for different security interpretations (§ 5).
- We mechanize the proof that the enforcement system actually satisfies the security condition (§ 6.2). Our proof is a work in progress but is aimed to be modularly extended to different facets.

2 FORMALIZATION

This section presents the formalization of λ^{DP}, a monadic λ-calculus and its abstract operational semantics. The semantics are abstract in the sense that they are parametric in a set of operations that, when instantiated, can implement various custom security conditions. Hence the semantics is parametric in the security condition. § 2.1 presents the syntax of λ^{DP} and § 2.2 presents its operational semantics.

2.1 Syntax

Figure 1 presents the syntax of λ^{DP}, which contains the syntactic categories e, v, c that represent expressions, values, and constants respectively.

Constants c consist of the boolean values `True` and `False`, the unit value `()`, and identifiers, which are integers denoted by the symbol i. The symbol $\bullet$ is used to represent erased values and is required by our metatheory (cf. § 6.2).

Values consist of constants (c), the policy relation (rel), the system refined policy (ps), labels (l), variables (x), functions ($\lambda x.e$), and recursive functions ($\texttt{fix}\ e$). The value ($\texttt{M}\ e$) is used to represent a monadic value and the construct ($\texttt{LB}\ l\ e\ i$) represents a value that labels with expression e with label label l and the unique identifier i.

$$\textit{Context}\quad C ::= \cdot \mid C\ e \mid \texttt{fix}\ C \mid \texttt{if}\ C\ \texttt{then}\ e\ \texttt{else}\ e \mid \texttt{ret}\ C \mid C \gg\!= e \mid \texttt{unlabel}\ C \mid \texttt{label}\ C\ e \mid \texttt{toLabeled}\ C\ e$$

$$\frac{I \vdash \langle \Sigma_1 \mid e_1 \rangle \rightarrow \langle \Sigma_2 \mid e_2 \rangle}{I \vdash \langle \Sigma_1 \mid C[e_1] \rangle \rightarrow \langle \Sigma_2 \mid C[e_2] \rangle}\ \text{E-Cntx}$$

Figure 2: Context for λ^{DP}.

Pure Semantics $\boxed{I \vdash \langle \Sigma \mid e \rangle \rightarrow \langle \Sigma \mid e \rangle}$

$$\frac{}{I \vdash \langle \Sigma \mid (\lambda x.e)\ e_x \rangle \rightarrow \langle \Sigma \mid e[e_x/x] \rangle}\ \text{E-App}$$

$$\frac{}{I \vdash \langle \Sigma \mid \texttt{if True then } e_1 \texttt{ else } e_2 \rangle \rightarrow \langle \Sigma \mid e_1 \rangle}\ \text{E-TIf}$$

$$\frac{}{I \vdash \langle \Sigma \mid \texttt{if False then } e_1 \texttt{ else } e_2 \rangle \rightarrow \langle \Sigma \mid e_2 \rangle}\ \text{E-FIf}$$

$$\frac{}{I \vdash \langle \Sigma \mid \texttt{fix } (\lambda f.e) \rangle \rightarrow \langle \Sigma \mid e[(\texttt{fix } \lambda f.e)/f] \rangle}\ \text{E-Fix}$$

Figure 3: Pure fragment of the operational semantics.

Finally, *expressions* include the standard pure constructs, that is values (v), conditionals ($\texttt{if}\ e\ \texttt{then}\ e\ \texttt{else}\ e$), and function application ($e\ e$). The monadic constructs include the return ($\texttt{ret}\ e$) and bind ($e \gg\!= e$) operations. The labeled construct $\texttt{label}\ e_l\ e$ is used to label an expression e with label e_l and the $\texttt{unlabel}\ e$ operator is used to remove the label from an expression. Following [8], the $\texttt{toLabeled}\ l\ e$ operator is used to label an expression e with the label l, to avoid the label creep problem, i.e., when the ls is increased so much that no useful computation can be done. To securely execute this expression, the $\texttt{toLabRet}\ \Sigma\ l\ e$ operator is used internally to keep track of the original state during evaluation. Finally, the $\texttt{setRel}\ e$ and $\texttt{getRel}$ operators are used to change and retrieve the policy state, respectively.

2.2 Semantics

Here, we present the small-step, contextual operational semantics of λ^{DP}. This semantics is formalized as the relation $I \vdash \langle \Sigma_1 \mid e_2 \rangle \rightarrow \langle \Sigma_2 \mid e_2 \rangle$ indicating that, given the instance I, the *configuration* $\langle \Sigma_1 \mid e_1 \rangle$ (consisting of a state Σ_1 and a monadic expression e_1) evaluates, in one step, to the configuration $\langle \Sigma_2 \mid e_2 \rangle$. Next, we first precisely formalize the notion of state for our semantics. Then, we overview the operational rules formalizing our semantics. We conclude by introducing the notion of traces, i.e., sequences of configurations associated with a valid execution.

States. A state Σ is $\langle sid, ls, ps, rel \rangle$, where sid is used to generate unique identifiers for each label, ls is the current label set that contains the labels of the values accessible in the current computation, *i.e.,* represents the security level of the computation, rel is the user defined policy relation as in [5], and ps is the system refined policy state, which tracks specific facet information. We define the initial state Σ_0 as having:

- $\Sigma_0.ls = \emptyset$
- $\Sigma_0.sid = \emptyset$
- $\Sigma_0.ps = \varepsilon$
- $\Sigma_0.rel = \varepsilon$

Effectful Semantics $\boxed{I \vdash \langle \Sigma \mid e \rangle \rightarrow \langle \Sigma \mid e \rangle}$

$$\frac{}{I \vdash \langle \Sigma \mid \texttt{ret}\ v \rangle \rightarrow \langle \Sigma \mid \texttt{M}\ v \rangle}\ \text{E-Ret}$$

$$\frac{}{I \vdash \langle \Sigma \mid \texttt{M}\ e_x \gg\!= e \rangle \rightarrow \langle \Sigma \mid e\ e_x \rangle}\ \text{E-Bind}$$

$$\frac{\Sigma_2 = I.\texttt{opLabel}(\Sigma_1, l, i) \quad I.\texttt{guard}(\Sigma_1, l) \quad I.\texttt{stateGuard}(\Sigma_1, \Sigma_2)}{I \vdash \langle \Sigma_1 \mid \texttt{label}\ l\ e \rangle \rightarrow \langle \Sigma_2 \mid \texttt{ret}\ (\texttt{LB}\ l\ e\ (\Sigma_1.sid.l)) \rangle}\ \text{E-Label}$$

$$\frac{I.\texttt{stateGuard}(\Sigma_1, \Sigma_2) \quad \Sigma_2 = I.\texttt{opUnlabel}(\Sigma_1\ \{ls := (l, i) : \Sigma_1.ls\}, l, i)}{I \vdash \langle \Sigma_1 \mid \texttt{unlabel}\ (\texttt{LB}\ l\ e\ i) \rangle \rightarrow \langle \Sigma_2 \mid \texttt{ret}\ e \rangle}\ \text{E-Unlabel}$$

$$\frac{}{I \vdash \langle \Sigma \mid \texttt{toLabeled}\ l\ e \rangle \rightarrow \langle \Sigma \mid e \gg\!= \lambda x.\texttt{toLabRet}\ \Sigma\ l\ x \rangle}\ \text{E-ToLab}$$

$$\frac{I.\texttt{guard}(\Sigma_1, l) \quad I.\texttt{stateGuard}(\Sigma_1, \Sigma_2) \quad \Sigma_2 = I.\texttt{opToLabRet}(\Sigma_1, \Sigma, l, \Sigma_1.sid.l)}{I \vdash \langle \Sigma_1 \mid \texttt{toLabRet}\ \Sigma\ l\ (\texttt{M}\ e) \rangle \rightarrow \langle \Sigma_2 \mid \texttt{ret}\ (\texttt{LB}\ l\ e\ (\Sigma_1.sid.l)) \rangle}\ \text{E-RToLab}$$

$$\frac{\Sigma_2 = I.\texttt{opState}(\Sigma_1, e) \quad I.\texttt{stateGuard}(\Sigma_1, \Sigma_2)}{I \vdash \langle \Sigma_1 \mid \texttt{setRel}\ e \rangle \rightarrow \langle \Sigma_2 \mid \texttt{ret}\ () \rangle}\ \text{E-SetRel}$$

$$\frac{}{I \vdash \langle \Sigma \mid \texttt{getRel} \rangle \rightarrow \langle \Sigma \mid \texttt{ret}\ \Sigma.rel \rangle}\ \text{E-GetRel}$$

Figure 4: Effectful fragment of the operational semantics.

where ε denotes the empty component with no information,

The Relation R provides the operator for the user-defined policy relation, i.e, $\texttt{lrt}(\Sigma, l, l')$ that returns *True* if l is allowed to flow to l' according to Σ.

The instance I is a set of abstract operations that are used to define the semantics of the language and can be instantiated based on the intended security:

- $\texttt{opLabel}(\Sigma, l, i)$ returns a state tracking the label operation effect.
- $\texttt{opUnlabel}(\Sigma, l, i)$ returns a state tracking the unlabel operation effect.
- $\texttt{opToLabRet}(\Sigma, \Sigma_0, l, i)$ returns a state tracking the tolabeled operation effect.
- $\texttt{guard}(\Sigma, l)$ is a predicate that checks if the information in the computation is observable by l.
- $\texttt{stateGuard}(\Sigma, \Sigma')$ is a predicate that checks if Σ' is an allowed state after Σ.
- $\texttt{opState}(\Sigma, e)$ returns a state whose current *rel* is set to e.
- $\texttt{policy}(l, i, l', \Sigma)$ is the operator on the system refined policy.

In § 6.1 we provide a set of requirements that these operations should satisfy to ensure the security of the system.

Operational Rules. First, we define context evaluation for expressions in Figure 2 together with rule E-Cntx used for the evaluation. Then, Figure 3 presents the pure fragment of the operational semantics, i.e., E-App, E-TIf, E-FIf, E-Fix, that are the standard λ-calculus rules with capture-avoiding substitution denoted by $e[e_x/x]$ meaning that x in e is substituted with e_x.

Figure 4 presents the effectful fragment of the operational semantics. The monadic rule E-Ret lifts a value in the monad, and rule E-Bind binds the monadic value to the functional argument. Rule E-Label creates a labeled value and uses the guard operation to ensure that the labeling is allowed with respect to the current state. Dually, rule E-Unlabel extracts data from a labeled values and updates the state accordingly. Rule E-ToLab is used to avoid "*label creep*" [9], to do so it binds the inner monad with the $\texttt{toLabRet}\ \Sigma\ l\ e$ operation that captures the original state. Rule E-RToLab is used to restore the original state, taking into account the effects after executing the inner monad of the corresponding E-ToLab, and return a labeled value. Finally, rule E-SetRel sets the current user policy state and dually, rule E-GetRel returns the current user policy state.

Example: Operational Semantics of SLIO. Based on the definitions of the instance I operations, our system encodes different security policies. For example, the below instance defines the operational semantics for SLIO:

- $\texttt{opLabel}(\Sigma, l, i) = \Sigma$,
- $\texttt{opUnlabel}(\Sigma, l, i) = \Sigma$,
- $\texttt{opToLabRet}(\Sigma, \Sigma_0, l, i) = \Sigma\ \{ls :=, \Sigma_0.ls, ps := \Sigma_0.ps\}$
- $\texttt{guard}(\Sigma, l') = \forall (l, i) \in \Sigma.ls.\texttt{policy}(\Sigma, l, i, l')$,
- $\texttt{stateGuard}(\Sigma, \Sigma') = \forall (l', i') \in ls.\neg\text{incUpperSet}(\Sigma, \Sigma', l', i')$,
- $\texttt{opState}(\Sigma, e) = \Sigma\ \{rel := e\}$,
- $\texttt{policy}(\Sigma, l, i, l') = \texttt{lrt}(\Sigma, l, l')$.

where:
$\text{incUpperSet}(\Sigma, \Sigma', l, i) = \exists l'.\neg\texttt{policy}(\Sigma, l, i, l') \wedge \texttt{policy}(\Sigma', l, i, l')$

Traces. Finally, we define a trace t^n of length n as the sequence of the n configurations generated by applying the rules of the operational semantics for a starting configuration $\langle \Sigma \mid e \rangle$. Since programs are deterministic, a trace t^n is uniquely determined by its initial configuration. For simplicity we omit the length n whenever it is not needed, i.e., we write t instead of t^n.

Definition 1 (Trace)**.** A configuration $\langle \Sigma_0 \mid e_0 \rangle$ produces a trace t^n of configurations written $\langle \Sigma_0 \mid e_0 \rangle \Downarrow t^n$ if there are configurations $\langle \Sigma_0 \mid e_0 \rangle \cdot \langle \Sigma_1 \mid e_1 \rangle \cdot \ldots \cdot \langle \Sigma_n \mid e_n \rangle$ s.t. $\forall i \in [0, n-1].I \vdash \langle \Sigma_i \mid e_i \rangle \rightarrow \langle \Sigma_{i+1} \mid e_{i+1} \rangle$.

3 ATTACKER MODEL

This section presents the attacker model, § 3.1 presents the observational power of the attacker, *i.e.,* the information that the attacker can get from the system, and § 3.2 presents two types of attackers and how their knowledge is extended through observations.

3.1 Observation power

Let the initial expression be the initial secret input and the configurations are the observations of the attacker we use term erasure to remove non-observable information from the configurations as in [5]. Below we define the function $obs_A(\langle \Sigma \mid e \rangle)$ that determines if an attacker can observe a configuration.

Definition 2. A configuration $\langle \Sigma \mid e \rangle$ is observable by an attacker A if the label set of the configuration $\Sigma.ls$ is observable by A, *i.e.,*

$$obs_A(\langle \Sigma \mid e \rangle) \doteq \forall (l, i) \in \Sigma.ls.\ \texttt{policy}(\Sigma, l, i, A)$$

Figure 5 defines the interesting cases of $\varepsilon_A(\cdot)$ on traces, state, and labeled values. The erasure of a trace t, denoted $\varepsilon_A(t)$, removes from a trace t all configurations not observable on level A, and applies the erasure function on the remaining ones such that erasure is applied both on the state and expression components.

Trace Erasure $\boxed{\varepsilon_A(t) = t'}$

$$\varepsilon_A(\langle\Sigma \mid e\rangle \cdot t) = \begin{cases} \langle \varepsilon_A(\Sigma) \mid \varepsilon_A^\Sigma(e)\rangle \cdot \varepsilon_A(t) & \text{if } obs_A(\langle\Sigma \mid e\rangle) \\ \varepsilon_A(t) & \text{otherwise} \end{cases}$$

State Erasure $\boxed{\varepsilon_A(\Sigma) = \Sigma'}$

$$\varepsilon_A(\Sigma) = \langle ls, \varepsilon_A^\Sigma(ps), rel, \varepsilon_A^\Sigma(sid)\rangle$$

Label Ids Erasure $\boxed{\varepsilon_A^\Sigma(sid) = sid'}$

$$\varepsilon_A^\Sigma(sid) = \begin{cases} (l, ix) : \varepsilon_A^\Sigma(sid) & \text{if } \forall i < ix.\ \texttt{policy}(\Sigma, l, i, A) \\ \varepsilon_A^\Sigma(sid) & \text{otherwise} \end{cases}$$

System Policy Erasure $\boxed{\varepsilon_A^\Sigma(ps) = ps'}$

$$\varepsilon_A^\Sigma(ps) \text{ s.t. } \forall (l, i), l'.\ \texttt{policy}(\varepsilon_A(\Sigma), l, i, l') = \begin{cases} \texttt{policy}(\Sigma, l, i, l') & \text{if } \texttt{policy}(\Sigma, l, i, A) \\ \texttt{false} & \text{otherwise} \end{cases}$$

Expression Erasure $\boxed{\varepsilon_A^\Sigma(e) = e'}$

$$\varepsilon_A^\Sigma(\texttt{LB}\ l\ e\ i) = \begin{cases} \texttt{LB}\ l\ \varepsilon_A^\Sigma(e)\ i & \text{if } \texttt{policy}(\Sigma, l, i, A) \\ \texttt{LB}\ l\ \bullet\ \bullet & \text{otherwise} \end{cases}$$

Figure 5: Erasure for non-trivial cases, the rest are homomorphisms.

The erasure on the state is defined such that *ls* and *rel* are left unchanged, and *sid* and *ps* are erased with their respective erasure functions. The *sid* component is erased so that the attacker can only observe the current identifier of a label if all the labeled values with such a label are observable. Then, for system policy component *ps*, since the actual data structure is parametric, we define the behavior of the *policy* function on the erased state, *i.e.,* the attacker can only observe the system policy of observable labeled values.

Finally, we report the erasure of labeled values that hides the unique identifier and value for non-observable labeled values. The remaining cases (omitted) are defined as homomorphisms (*e.g.,* $\varepsilon_A(\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3) = \texttt{if}\ \varepsilon_A(e_1)\ \texttt{then}\ \varepsilon_A(e_2)\ \texttt{else}\ \varepsilon_A(e_3)$).

3.2 Attacker knowledge

The attacker's knowledge captures how much information the attacker knows about the secret, *i.e.,* the initial configuration. Following [5], we model the attacker's knowledge using the attacker exclusion knowledge set, *i.e.,* the set of initial configurations that the attacker can exclude given the observations associated with a trace. Let $\langle\Sigma_0 \mid e_0\rangle$ be the initial configuration,

and $o = \varepsilon_A(t)$ the observations made by the attacker A along trace t, then the exclusion knowledge is the set that contains all the initial $\langle\Sigma \mid e\rangle$ that the attacker can exclude since they could not have led to the observations [10].

Following Chong and Askarov [6], we consider two different models of attacker knowledge, which we overview next. In the **perfect recall model** (§ 3.2.1), we consider an attacker that remembers all the information that they have observed. In contrast, in the **forgetful model** (§ 3.2.2), we consider an attacker that resets their knowledge after every policy change. We remark that these models are associated with different security guarantees, reflected in their different notions of knowledge § 4.

3.2.1 Perfect recall. In the perfect recall model, attackers remember all information they observed since the beginning of the computation. This is reflected in § 3.

Definition 3 (Exclusion knowledge)**.** Given a trace t, let $o = \varepsilon_A(t)$ be the observations made by the attacker A. Then the exclusion knowledge set for A is:

$$ek_A^{prf}(o) = \{e' \mid \neg\exists t'.\langle\Sigma_0 \mid e'\rangle \Downarrow t' \text{ with } \varepsilon_A(t') = o\}$$

In dynamic security monitors, attackers can learn information from whether the computation can perform one more step or has terminated [6]. To account for these leaks, we define the progress exclusion knowledge set [6] as the set of $\langle\Sigma \mid e\rangle$ that the attacker A can exclude because could not have led to the current observation o and can produce another observable configuration α.

Definition 4 (Exclusion progress knowledge)**.** Given a trace t and $o = \varepsilon_A(t)$ the exclusion progress knowledge of A is defined as:

$$ek_A^{+prf}(o) = \{e' \mid \neg\exists t', \alpha'.\langle\Sigma_0 \mid e'\rangle \Downarrow t' \cdot \alpha' \text{ with } \varepsilon_A(t') = o \ \wedge\ obs_A(\alpha')\}$$

Hence, we have that for $\langle\Sigma_0 \mid e_0\rangle \Downarrow t \cdot \alpha$ with $obs_A(\alpha)$ the increase in knowledge of A observing α is the difference between the exclusion knowledge set and the exclusion progress knowledge set.

3.2.2 Forgetful attacker knowledge. In the forgetful model, attackers forget all information they observed from the beginning of the computation until the last change in the policy. We call epoch the piece of trace between two policy changes. Hence, policy events partition the trace into epochs. At each step, only the observable events of the current epoch affect the knowledge of the attacker. Given the trace t produced by the evaluation of the initial configuration, we define the function $splitPolicy(t) = (t_1, t_2)$ that splits the trace on the last policy change where t_1 is the trace before the last policy change and t_2 is the last epoch.

Definition 5 (Exclusion knowledge)**.** Given a trace t and assuming that $splitPolicy(t) = (t_1, t_2)$ with $o = \varepsilon_A(t_2)$ the exclusion progress knowledge of the attacker A is defined as:

$$ek_A^{frg}(o) = \{e' \mid \neg\exists t'.\langle\Sigma \mid e'\rangle \Downarrow t' \ \wedge\ (t_1', t_2') = splitPolicy(t') \ \wedge\ \varepsilon_A(t_2') = o\}$$

Hence, we have that for $\langle\Sigma_0 \mid e_0\rangle \Downarrow t \cdot \alpha$ with $obs_A(\alpha)$ the increase in knowledge of A observing α is the difference between the exclusion knowledge set and the exclusion progress knowledge set.

Similarly to Section 3.2.1, we introduce the exclusion progress knowledge for the forgetful model.

Definition 6 (Exclusion progress knowledge)**.** Given a trace t and $splitPolicy(t) = (t_1, t_2)$ with $o = \varepsilon_A(t_2)$ the increase in knowledge

of A is:

$$ek_A^{+frg}(o) = \{e' \mid \neg\exists t', \alpha'.\langle \Sigma_0 \mid e'\rangle \Downarrow t' \cdot \alpha' \text{ with } (t'_1, t'_2) = splitPolicy(t') \wedge \varepsilon_A(t'_2) = o \wedge obs_A(\alpha')\}$$

4 SECURITY

This section presents the analysis of the security for the identified facets. In § 4.1 we define the structure of a security condition. In § 4.2 we present the security analysis of the time-transitive flows and formalize the respective security conditions for the allowed and forbidden cases. In § 4.3 we present the *replaying flow* facet and provide the same analysis as for the time-transitive flows.

4.1 Security conditions

The goal of our security conditions is to constraint how the knowledge changes during the execution in such a way that agrees with the security policy (describing what the attacker is allowed to learn). For this, we formalize all our conditions by bounding the increase in knowledge of the attacker, *i.e.,* the difference between the attacker's exclusion knowledge and the exclusion progress knowledge. In the following, with an abuse of notation, we avoid writing the specific type of attacker, *i.e.,* perfect recall or forgetful, in the security conditions and we just use $ek_A(\cdot)$ and $ek_A^+(\cdot)$ to denote the attacker's exclusion knowledge and the exclusion progress knowledge, respectively. Hence, given $\langle \Sigma \mid e\rangle \Downarrow t \cdot \alpha$ where α is the new observation, and an attacker A we want to define a bound $\mathcal{B}_A(t, \alpha)$ such that

$$ek_A(\varepsilon_A(t \cdot \alpha)) \backslash ek_A^+(\varepsilon_A(t)) \subseteq \mathcal{B}_A(t, \alpha)$$

In all four security conditions that we study, the attacker A is allowed to exclude the set of expressions e' that are not erasure-equal to the initial expression e under a current state Σ. This set of expressions is defined below as the Input Release $I_A(e, \Sigma)$.

Definition 7 (Input Release [5])**.** Given the initial configuration $\langle \Sigma_0 \mid e_0\rangle$, and a state Σ.

$$I_A(e_0, \Sigma) = \{e' \mid \varepsilon_A^{\Sigma}(e_0) \neq \varepsilon_A^{\Sigma}(e')\}$$

4.2 Time-transitive security

A flow is time-transitive if it moves information from a security level A to level C via a third level B, while there is no moment in time where the flow from A to C is directly enabled. In the following, we give the intuition of this facet and its interpretations, and then we define the security conditions for the allowed and forbidden cases.

Intuition. Table 1 shows the two interpretations of time transitive flows. The example on the left represents the context in which one may want to allow the flow to permit the input to flow to the database but only after being sanitized. Instead, one may forbid this flow when a hospital is trying to share patient data after the patient has left the hospital.

Allowed. To allow the attacker to learn time transitive information, Buiras and van Delft [5] define a specific erasure function *multilevel erasure* that is only used in Definition 8. Although their

```
ex = do
   b <- label Bob "secret"
   -- Bob -> Carla
   c <- toLabeled Carla (do
      unlabel b
   )
   -- Bob -/> Carla
   -- Carla -> Dave
   d <- toLabeled Dave (do
      unlabel c
   )
   -- Carla -/> Dave
   -- Dave -> Atk
```

Figure 6: Example showing that multi-level erasure in [5] is wrong

$$\varepsilon_{A_\downarrow}(t \cdot \langle \Sigma_n \mid e_n\rangle) = \begin{cases} \varepsilon_{A'_\downarrow}(t) \cdot \langle \varepsilon_{A_\downarrow}(\Sigma_n) \mid \varepsilon_{A_\downarrow}^{\Sigma_n}(e)\rangle & \text{if } \exists l' \in A_\downarrow.obs_{A_\downarrow}(\langle \Sigma_n \mid e_n\rangle) \text{ with } A'_\downarrow = A_\downarrow(A_\downarrow, \Sigma_n) \\ \varepsilon_{A_\downarrow^{n-1}}(t) & \text{otherwise} \end{cases}$$

$$\varepsilon_{A_\downarrow}(\Sigma) = \langle ls, \varepsilon_{A_\downarrow}^{\Sigma}(ps), rel, \varepsilon_{A_\downarrow}^{\Sigma}(sid)\rangle$$

$$\varepsilon_{A_\downarrow}^{\Sigma}(sid) = \begin{cases} (l, ix) : \varepsilon_{A_\downarrow}^{\Sigma}(sid) & \text{if } \exists l' \in A_\downarrow.\texttt{lrt}(\Sigma, l, l') \\ \varepsilon_{A_\downarrow}^{\Sigma}(sid) & \text{otherwise} \end{cases}$$

$$\varepsilon_{A_\downarrow}^{\Sigma}(ps) \text{ s.t. } \forall (l, i), l' \Rightarrow \texttt{policy}(\varepsilon_{A_\downarrow}(\Sigma), l, i, l') = \begin{cases} \texttt{policy}(\Sigma, l, i, l') & \text{if } l' \in A_\downarrow \\ \text{false} & \text{otherwise} \end{cases}$$

$$\varepsilon_{A_\downarrow}^{\Sigma}(\texttt{LB}\ l\ e\ i) = \begin{cases} \texttt{LB}\ l\ \varepsilon_{A_\downarrow}^{\Sigma}(e)\ i & \text{if } \exists l' \in A_\downarrow.\texttt{policy}(\Sigma, l, i, l') \\ \texttt{LB}\ l\ \bullet\ \bullet & \text{otherwise} \end{cases}$$

Figure 7: Multi-level erasure for non-trivial cases

intuition is correct, their definition is not complete, as we show in Figure 6.

Buiras and van Delft [5] define the multilevel erasure function to represent the information collectively known by a set L of labels that the attacker can observe based on the policy, *i.e.,* given a state Σ then $L = \{l \mid \texttt{lrt}(\Sigma, l, A)\}$. We observe that if the attacker is allowed to learn from labels in L it will also learn information that was released to those labels. From Figure 6, we observe that when the policy allows $Dave \rightarrow Atk$ the attacker is able to learn b from d. Their definition would not allow the attacker to learn b, because $L = \{Dave, Atk\}$, but their implementation does not consider this program insecure. We think that since we are allowing time transitive flows, the attacker should be able to learn b from d. Hence, we adjust the multilevel erasure function to allow the attacker to learn all time transitive information.

Our multilevel erasure function $\varepsilon_{A_\downarrow}(\cdot)$ is reported in Figure 7. We define $A_\downarrow$ as the down closure of the policy given by configuration n in trace t for the attacker A: $A_\downarrow(\{l\}, \Sigma) = \{l\} \cup \{l' \mid \exists l \in \{l\}.\texttt{lrt}(\Sigma, l, l')\}$. In this way, we can expand this set by going backward in the trace, allowing the attacker to learn information that was released to the labels in the down closure of the policy. Now we can instantiate relabeling release [5] to specify the information that A can exclude based on the multilevel erasure *i.e.,* the information that could have transitively flown to A.

Allowed	Forbidden
1 -- NSA ⟶ Military	1 -- Card ⟶ Log
2 share Military nsaSecret	2 write Log cardNumber
3 -- NSA ↛ Military	3 -- Card ↛ Log
4 share Military nsaSecret	4 write Log cardNumber

Table 2: Examples of replaying flow

Definition 8 (Relabeling release)**.** Given $\langle \Sigma \mid e\rangle \Downarrow t \cdot \langle \Sigma_n \mid e_n\rangle$ with $obs_A(\langle \Sigma_n \mid e_n\rangle)$, and $A_\downarrow(A, \Sigma_n)$. Relabeling release $R_A(t, A_\downarrow)$ is defined as the set of inputs e', released by relabeling on trace t, that an attacker A is able to exclude.

$$R_A(t, A_\downarrow) = \{e' \mid \neg\exists t', \alpha'. \langle \Sigma_0 \mid e'\rangle \Downarrow t' \cdot \alpha' \text{ with } obs_A(\alpha') \\ \wedge\ \varepsilon_{A_\downarrow}(t) = \varepsilon_{A_\downarrow}(t')\}$$

Definition 9 (Termination-insensitive security allowing TT flows)**.** The expression e is secure against an attacker A if for all traces t and configurations α such that $\langle \Sigma_0 \mid e\rangle \Downarrow t \cdot \alpha$ with $obs_A(\alpha), \alpha = \langle \Sigma_n \mid e_n\rangle$, the attacker's increase in knowlegde is bounded by $I_A(e, \Sigma_n)$ and $R_A(t, A_\downarrow)$:

$$ek_A(\epsilon_A(t \cdot \alpha)) \backslash ek_A^+(\epsilon_A(t)) \subseteq I_A(e, \Sigma_n) \cup R_A(t, A_\downarrow)$$

Forbidden. By restricting the attacker's increase in knowledge to the input release only we obtain a security condition that disallows time-transitive flows. This is because the attacker is only allowed to exclude inputs that are not A-equal to the initial input using the current policy state.

Definition 10 (Non Time-transitive termination-insensitive security)**.** The expression e is secure against an attacker A if for all traces t and configurations α such that $\langle \Sigma_0 \mid e\rangle \Downarrow t \cdot \alpha$ with $obs_A(\alpha), \alpha = \langle \Sigma_n \mid e_n\rangle$, the attacker's increase in knowlegde is bounded by $I_A(e, \Sigma_n)$

$$ek_A(\epsilon_A(t \cdot \alpha)) \backslash ek_A^+(\epsilon_A(t)) \subseteq I_A(e, \Sigma_n)$$

4.3 Replaying flows

A flow is said to be replaying when the release of information is considered permanent and, therefore, can be securely repeated. In the following, we give the intuition of this facet with its interpretations, and then we define the security conditions for the allowed and forbidden cases.

Intuition. In Table 2 the example on the left shows a context in which we want to allow this type of flow. If *NSA* shares some information, permanently, with *Military* it makes sense to always allow the same information to be shared again, since nothing new is going to be learned.

Instead, the example on the right, shows that it may be insecure when secret information is not released permanently, like a credit card number is written to a log, then after the policy disallows the flow from *Card* ⟶ *Log* at line 3, we want to forbid the program that tries to write the value again.

Allowed. We observe that under a perfect recall attacker, the structure of the security condition given in § 4 *allows* replaying flows by default. Indeed, given the perfect recall nature of the knoweldge, releasing again the same information does not result in knowledge increases. That is,

$$ek_A^{prf}(\varepsilon_A(t \cdot \alpha)) \backslash ek_A^{+prf}(\varepsilon_A(t)) = \emptyset$$

i.e., the attacker's increase in knowledge is always empty under the perfect recall model.

In the example from table 2, when the policy allows *NSA* to flow to *Military*, the attacker on level *Military* learns the value of n. Hence, even if later the policy forbids the flow from *NSA* to *Military*, the second relabel is still safe since the attacker already knows the value of n. Finally, also observe that the policy bound allowing for *Relabel Release* also allows for replaying flows.

Definition 11 (Replaying termination-insensitive security)**.** Command e is secure against an attacker A with replaying flows if for all traces t and configurations α such that $\langle \Sigma_0 \mid e\rangle \Downarrow t \cdot \alpha$ with $obs_A(\alpha), \alpha = \langle \Sigma_n \mid e_n\rangle$, the attacker's increase in knowlegde is bounded by $I_A(e, \Sigma_n) \cup R_A(t, A_\downarrow)$.

$$ek_A^{prf}(\varepsilon_A(t \cdot \alpha)) \backslash ek_A^{+prf}(\varepsilon_A(t)) \subseteq I_A(e, \Sigma_n) \subseteq I_A(e, \Sigma_n) \cup R_A(t, A_\downarrow)$$

Forbidden. If we want to disallow replaying flows we have to consider the forgetful attacker. This approach is based on the fact that a forgetful attacker perceives relearned information as if encountering it for the first time, thereby reflecting an actual increase in knowledge. However, simply considering the forgetful attacker is insufficient. Additionally, the policy bound must prevent the attacker from learning such past information. Hence the security condition that disallows replaying flows is defined as follows.

Definition 12 (Non replaying termination-insensitive security)**.** Command e is secure against an attacker A and does not allow replaying flows if for all traces t and configurations α such that $\langle \Sigma_0 \mid e\rangle \Downarrow t \cdot \alpha$ with $obs_A(\alpha), \alpha = \langle \Sigma_n \mid e_n\rangle$, the attacker's increase in knowlegde is bounded by $I_A(e, \Sigma_n)$.

$$ek_A^{frg}(\varepsilon_A(t \cdot \alpha)) \backslash ek_A^{+frg}(\varepsilon_A(t)) \subseteq I_A(e, \Sigma_n).$$

5 INSTANCES

This section presents how the instances I of the λ^{DP} system are defined to satisfy the security conditions of the different interpretations of the facets. In § 5.1 we present instances for time-transitive flows and in § 5.2 we present the ones for replaying flows.

5.1 Time-Transitive Flows

Here we present the instances for the time-transitive flows interpretations.

5.1.1 Allowed. Since SLIO allows explicitly time-transitive flows, we use the instance given in § 2.2 for such system.

5.1.2 Forbidden. To instantiate a system without time-transitive flows we use a map that associates the newly created labeled values with the *ls*. This way we can track all the labeled values that were in the computation so that we can avoid any time transitive flow. To do this, we model *ps* as a map such that $ps(l, i)$ retrieves the associated *ls* of the computation when the labeled vaue (l,i) was created. Then $ps[(l, i) \rightarrow ls]$ represents the update of the map with the new association.

When we create a labeled value the $\texttt{opLabel}(\Sigma, l, i)$ updates the state by adding the association $(l, i) \rightarrow ls$ to ps. Then, when unlabeling, $\texttt{opUnlabel}(\Sigma, l, i)$ returns a state in which ls contains $\Sigma.ps(l, i)$.

- $\texttt{opLabel}(\Sigma, l, i) = \Sigma\ \{ps := ps[(l, i) \rightarrow ls]\}$
- $\texttt{opUnlabel}(\Sigma, l, i) = \Sigma\ \{ls := ls \cup ps(l, i) \cup (l, i)\}$
- $\texttt{opToLabRet}(\Sigma, \Sigma_0, l, i) = \Sigma \begin{Bmatrix} ls := \Sigma_0.ls, \\ ps := \Sigma.ps[(l, i) \rightarrow ls] \end{Bmatrix}$
- $\texttt{guard}(\Sigma, l') = \forall (l, i) \in ls.\texttt{policy}(\Sigma, l, i, l')$,
- $\texttt{stateGuard}(\Sigma, \Sigma') = \forall (l', i') \in ls.\neg\text{incUpperSet}(\Sigma, \Sigma', l', i')$
- $\texttt{opState}(\Sigma, e) = \Sigma\ \{rel\ := e\}$
- $\texttt{policy}(\Sigma, l, i, l') = \texttt{lrt}(\Sigma, l, l') \bigwedge_{(l_p, i_p) \in \Sigma.ps(l,i)} \texttt{policy}(\Sigma, l_p, i_p, l')$

5.2 Replaying Flows

Here we present the instances for the replaying flows interpretations.

5.2.1 Allowed. To instantiate a system with replaying flows we need to track releases so that we can allow in the future the same release. After the first release, the policy should always allow the following ones. To do this we model ps as a map from labeled values to labels, representing for each labeled value the labels to which it was released and hence to which can be replayed.

- $\texttt{opLabel}(\Sigma, l, i) = \Sigma\ \{ps := ps[\forall (l', i') \in ls. (l', i') \rightarrow \Sigma.ps(l', i') \cup l]\}$
- $\texttt{opUnlabel}(\Sigma, l, i) = \Sigma$
- $\texttt{opToLabRet}(\Sigma, \Sigma_0, l, i) = \Sigma \begin{Bmatrix} ls := \Sigma_0.ls, \\ ps := ps[\forall (l', i') \in ls. \\ (l', i') \rightarrow \Sigma.ps(l', i') \cup l] \end{Bmatrix}$
- $\texttt{guard}(\Sigma, l') = \forall (l, i) \in ls.\texttt{policy}(\Sigma, l, i, l')$
- $\texttt{stateGuard}(\Sigma, \Sigma') = \forall (l', i') \in ls.\neg\text{incUpperSet}(\Sigma, \Sigma', l', i')$
- $\texttt{opState}(\Sigma, e) = \Sigma\ \{rel\ := e\}$
- $\texttt{policy}(\Sigma, l, i, l') = \texttt{lrt}(\Sigma, l, l') \vee l \in \Sigma.ps(l', i')$

5.2.2 Forbidden. We observe that the implementation that forbidds time-transitive flows also forbidds replaying flows hence we use the instance given in § 5.1.2 for such system.

6 PROOF & MECHANIZATION

This section presents the security proof of the system. In § 6.1, we define the requirements that the interface I needs to satisfy. Then, § 6.2 presents the security theorem, obtained by rewriting the security condition with logical connectives, and a proof sketch. Finally, § 6.3 discusses the mechanization of the proof in LiquidHaskell. The security theorem is instantiated with the security condition with allowed time transitive flows defined in § 4.2.

6.1 Requirements

The interface I is required to satisfy certain invariants to ensure that the system behaves as expected and that such operations do not reveal any information to the attacker.

Requirement 1. During the evaluation of the monadic expression e in $\langle \Sigma \mid \texttt{toLabeled}\ l\ e \rangle$ the component $\Sigma.rel$ is not allowed to change.

Requirement 2. For a state transition $\Sigma_2 = \texttt{opUnlabel}(\Sigma_1, l, i)$, the following hold:

- $\Sigma_2.sid = \Sigma_1.sid$
- $\Sigma_2.rel = \Sigma_1.rel$
- $\Sigma_2.ls \supseteq \Sigma_1.ls$
- $\forall (l, i), l'.\ \texttt{policy}(\Sigma_2, l, i, l') = \texttt{policy}(\Sigma_1, l, i, l')$

Requirement 3. For a state transition $\Sigma_2 = \texttt{opLabel}(\Sigma_1, l, i)$, the following hold:

- $\Sigma_2.sid = \Sigma_1.sid[l \mapsto \Sigma_1.sid.l + 1]$
- $\Sigma_2.rel = \Sigma_1.rel$
- $\Sigma_2.ls = \Sigma_1.ls$
- $\forall (l, i), l'.\ \texttt{policy}(\Sigma_2, l, i, l') = \texttt{policy}(\Sigma_1, l, i, l')$

Requirement 4. For a state transition $\Sigma_2 = \texttt{opToLabRet}(\Sigma_1, \Sigma_0, l, i)$, the following hold:

- $\Sigma_2.sid = \Sigma_1.sid[l \mapsto \Sigma_1.sid.l + 1]$
- $\Sigma_2.rel = \Sigma_0.rel$
- $\Sigma_2.ls = \Sigma_0.ls$
- $\forall (l, i), l'.\ \texttt{policy}(\Sigma_2, l, i, l') = \texttt{policy}(\Sigma_1, l, i, l')$

Definition 13. A step from $C_1 = \langle \Sigma_1 \mid e_1 \rangle$ to $C_2 = \langle \Sigma_2 \mid e_2 \rangle$ is said to be at level (l, i) if the operation is performed on the labeled value (l, i). This is denoted by $C_1 \xrightarrow{(l,i)} C_2$ and occurs if the step involves $\texttt{opLabel}(\Sigma, l, i)$, $\texttt{opUnlabel}(\Sigma, l, i)$, or $\texttt{opToLabRet}(\Sigma, \Sigma_0, l, i)$.

The following lemmas establish foundational properties of state transformations and erasures under observation constraints. They are crucial for simplifying the proof structure, as they provide reusable, generalized cases of behavior under specific conditions. These lemmas ensure that transformations maintain consistent erasures across steps, supporting the integrity of the system's security properties throughout the operational sequence.

Lemma 1. Consider a step from $C_1 = \langle \Sigma_1 \mid e_1 \rangle \xrightarrow{(l,i)} C_2 = \langle \Sigma_2 \mid e_2 \rangle$ observed by an attacker A, and suppose both $obs_A(C_1)$ and $obs_A(C_2)$ are true, with no change in the relevant system policies ($\Sigma_1.rel = \Sigma_2.rel$ and $\neg\texttt{policy}(\Sigma_1, l, i, A)$). Then:

- $\varepsilon_A(\Sigma_1.sid) = \varepsilon_A(\Sigma_2.sid)$
- $\varepsilon_A(\Sigma_1.ps) = \varepsilon_A(\Sigma_2.ps)$

The proof follows from the requirements, demonstrating that unchanged policies and identifiers ensure consistent erasures.

Lemma 2. Consider a step from $C_1 = \langle \Sigma_1 \mid e_1 \rangle \rightarrow C_2 = \langle \Sigma_2 \mid e_2 \rangle$ such that $\Sigma_1.rel = \Sigma_2.rel$ and $\neg obs_A(C_2)$ holds. Then:

- $\varepsilon_A(\Sigma_1.sid) = \varepsilon_A(\Sigma_2.sid)$
- $\varepsilon_A(\Sigma_1.ps) = \varepsilon_A(\Sigma_2.ps)$

The proof leverages the requirements that ensure unchanged identifiers and system policies when transitions are unobservable to the attacker.

Lemma 3. Consider a step, excluding E-RToLab, from $C_1 = \langle \Sigma_1 \mid e_1 \rangle \rightarrow C_2 = \langle \Sigma_2 \mid e_2 \rangle$ such that $obs_A(C_2)$, then $obs_A(C_1)$ holds. The proof goes by induction on the step derivation.

Lemma 4. Given two execution traces t^n and t^m with C_n and C_m the last configurations of t^n and t^m respectively, such that $\varepsilon_A(t^n) = \varepsilon_A(t^m)$ holds. If C_n and C_m are observable by A then it holds that: $\varepsilon_A(C_n) = \varepsilon_A(C_m)$.
The proof goes by the definition of the erasure.

Lemma 5 (Determinism)**.** Given $I \vdash \langle \Sigma_1 \mid e_1 \rangle \rightarrow \langle \Sigma_2 \mid e_2 \rangle$ and $I \vdash \langle \Sigma_1 \mid e_1 \rangle \rightarrow \langle \Sigma_3 \mid e_3 \rangle$ then $\langle \Sigma_2 \mid e_2 \rangle = \langle \Sigma_3 \mid e_3 \rangle$ Proof by structural induction on the rules.

Lemma 6 (Step Erasure)**.** Given $I \vdash \langle \Sigma_1 \mid e_1 \rangle \rightarrow \langle \Sigma_2 \mid e_2 \rangle$ with $\Sigma_1.rel = \Sigma_2.rel$ then $I \vdash \varepsilon_A(\langle \Sigma_1 \mid e_1 \rangle) \rightarrow \varepsilon_A(\langle \Sigma_2 \mid e_2 \rangle)$ Proof by structural induction on the rules.

Lemma 7 (Fixed State Lemma)**.** Given two singe-step evaluations $I \vdash \langle \Sigma_1 \mid e_1 \rangle \rightarrow \langle \Sigma_2 \mid e_2 \rangle$ and $I \vdash \langle \Sigma_1' \mid e_1' \rangle \rightarrow \langle \Sigma_2' \mid e_2' \rangle$ with $\Sigma_1.rel = \Sigma_1'.rel$. For all levels A, if $\varepsilon_A(\langle \Sigma_1 \mid e_1 \rangle) = \varepsilon_A(\langle \Sigma_1' \mid e_1' \rangle)$ then $\varepsilon_A(\langle \Sigma_2 \mid e_2 \rangle) = \varepsilon_A(\langle \Sigma_2' \mid e_2' \rangle)$.
Proof by the definition of $\varepsilon_A(\cdot)$ we know that $\Sigma_1.rel = \Sigma_1'.rel$ and that the operational rule taken is the same. By Lemma 6 (Step Erasure) we know that $I \vdash \varepsilon_A(\langle \Sigma_1 \mid e_1 \rangle) \rightarrow \varepsilon_A(\langle \Sigma_2 \mid e_2 \rangle)$ and $I \vdash \varepsilon_A(\langle \Sigma_1' \mid e_1' \rangle) \rightarrow \varepsilon_A(\langle \Sigma_2' \mid e_2' \rangle)$. Then using Lemma 5 (Determinism) on both steps we obtain $\varepsilon_A(\langle \Sigma_2 \mid e_2 \rangle) = \varepsilon_A(\langle \Sigma_2' \mid e_2' \rangle)$.

6.2 Security Theorem & Proof Sketch

Here, we present the security theorem of Definition 9 rewritten using logical connectives, instead of set notation, and a proof sketch. Since the bound is specific to the security condition, here we report the one that allows time transitive flows.

Theorem 1. Given two execution traces t and t' starting from $\langle \Sigma_0 \mid e \rangle$ and $\langle \Sigma_0 \mid e' \rangle$ respectively, if both traces are erasure equivalent under erasure for A and are equivalent for the bound $\mathcal{B}$ then if we take a step on both traces resulting in observable configurations C_n and C_m respectively, they are also equivalent under erasure for A.

$$
\begin{array}{lll}
\forall & e, e', t, t'. & \\
 & \left.\begin{array}{l} \langle \Sigma_0 \mid e \rangle \Downarrow t \cdot \langle \Sigma_n \mid e_n \rangle \\ \wedge \quad obs_A(\langle \Sigma_n \mid e_n \rangle) \end{array}\right\} & \text{SetUp} \\
\wedge & \left.\begin{array}{l} \langle \Sigma_0 \mid e' \rangle \Downarrow t' \cdot \langle \Sigma_m \mid e_m \rangle \\ \wedge \quad obs_A(\langle \Sigma_m \mid e_m \rangle) \\ \wedge \quad \varepsilon_A(t) = \varepsilon_A(t') \end{array}\right\} & ek_A^+(\epsilon_A(t)) \\
\wedge & \left.\begin{array}{l} \varepsilon_A^{\Sigma}(e) = \varepsilon_A^{\Sigma}(e') \\ \wedge \quad \varepsilon_{A_\downarrow}(t) = \varepsilon_{A_\downarrow}(t') \end{array}\right\} & \mathcal{B}_A(t, \alpha) \\
\Rightarrow & \varepsilon_A(\langle \Sigma_n \mid e_n \rangle) = \varepsilon_A(\langle \Sigma_m \mid e_m \rangle) \} & ek_A(\varepsilon_A(t \cdot \alpha))
\end{array}
$$

Proof. The proof proceeds by induction on the evaluation steps of the execution traces:

- **Base Cases**:
  - $\dfrac{\Sigma' = \mathbf{opState}(\Sigma, e) \quad \mathbf{stateGuard}(\Sigma, \Sigma')}{I \vdash \langle \Sigma \mid \mathbf{setRel}\ e \rangle \rightarrow \langle \Sigma' \mid \mathbf{ret}\ () \rangle}$ **E-SetRel:**
    From Lemma 3 and Lemma 4 we get $\varepsilon_A(\langle \Sigma_{n-1} \mid \texttt{setRel}\ e \rangle)$ $= \varepsilon_A(\langle \Sigma_{m-1} \mid e_{m-1} \rangle)$. By definition of erasure we derive that $e_{m-1} = \texttt{setRel}\ e$. Hence in both configurations, the new state is updated with the same user policy e. Now we have to show that also the states are erasure equal, from now on ε-equal. The components *ls* and *rel* are trivially equal, so we have to show that the *sid* and *ps* are ε-equal.
    We know that changing *rel* allows information in $A_\downarrow$ to be observable, so we have to show that this information is ε-equal on both traces. This is obtained by the $\mathcal{B}_A(t, \alpha)$ assumption. Therefore $\varepsilon_A(\Sigma_n.sid) = \varepsilon_A(\Sigma_m.sid)$ and $\varepsilon_A(\Sigma_n.ps)$ $= \varepsilon_A(\Sigma_m.ps)$.
    Hence $\varepsilon_A(\langle \Sigma_n \mid \texttt{ret}\ () \rangle) = \varepsilon_A(\langle \Sigma_m \mid \texttt{ret}\ () \rangle)$.
  - $\dfrac{I.\mathbf{guard}(\Sigma_{n-1}, l) \quad I.\mathbf{stateGuard}(\Sigma_{n-1}, \Sigma_n) \quad \Sigma_n = I.\mathbf{opToLabRet}(\Sigma_{n-1}, \Sigma_i, l, \Sigma_{n-1}.sid.l)}{I \vdash \langle \Sigma_{n-1} \mid \mathbf{toLabRet}\ \Sigma_i\ l\ (\mathbf{M}\ e) \rangle \rightarrow \langle \Sigma_n \mid \mathbf{ret}\ (\mathbf{LB}\ l\ e\ (\Sigma_{n-1}.sid.l)) \rangle}$ **E-RToLab :**
    We start by case splitting on the observability of the label l by A.
    - $\mathbf{lrt}(\Sigma_n, l, A)$: By $\texttt{guard}(\Sigma_{n-1}, l)$ we know that $\forall l' \in \Sigma_{n-1}.ls.\texttt{lrt}(\Sigma_{n-1}, l', l)$, hence since $\texttt{lrt}(\Sigma_n, l, A)$ and using requirement 1 we have that $\forall l' \in \Sigma_{n-1}.ls.$ $\texttt{lrt}(\Sigma_{n-1}, l', A)$. Now we have to show that the last step on m is indeed a E-RToLab. By $ek_A^+(\epsilon_A(t))$ we know that $\varepsilon_A(\langle \Sigma_{n-1} \mid e_{n-1} \rangle) = \varepsilon_A(\langle \Sigma_j \mid E_j \rangle)$.
      We need to show that j is indeed the last step in the trace ($j = m - 1$), and we know by the following configuration $j + 1$ has to be observable.
      By absurd $j \neq m - 1$ then we know that there is a step $j + 1$ between j and $m - 1$ that is observable by A, hence j is not the last step in the trace **ABSURD**.
      Hence $j = m - 1$ is the last step in the trace and we know that $\varepsilon_A(\langle \Sigma_{n-1} \mid e_{n-1} \rangle) = \varepsilon_A(\langle \Sigma_{m-1} \mid e_{m-1} \rangle)$.
      Now since they are ε-equal they have the same shape, suppose $e_{m-1} = \texttt{toLabRet}\ \Sigma_k\ l'\ e'$. By ε we have that $\Sigma_k = \Sigma_i$ and $l' = l$ and $\varepsilon_A(e') = \varepsilon_A(e)$. Hence we have that also $\varepsilon_A(\langle \Sigma_n \mid \texttt{ret}\ (\texttt{LB}\ l\ e\ \Sigma.sid) \rangle) = \varepsilon_A(\langle \Sigma_m \mid \texttt{ret}\ (\texttt{LB}\ l\ e'\ \Sigma.sid) \rangle)$.
    - $\neg\mathbf{lrt}(\Sigma_n, l, A)$: in this case we cannot directly derive that the previous configuration was observable, it could also be not observable. Hence we case split the analysis on that.
      - $obs_A(\langle \Sigma_{n-1} \mid e_{n-1} \rangle)$: in this case the proof is similar to the previous case.
      - $\neg obs_A(\langle \Sigma_{n-1} \mid e_{n-1} \rangle)$: in this case we need to show that the last step on t' is indeed a E-RToLab. First we prove by absurd that $\neg obs_A(C_{m-1})$: suppose that $obs_A(C_{m-1})$, we know by $ek_A^+(\epsilon_A(t))$ that exists an i in t^{n-1} s.t. $\varepsilon_A(C_i) = \varepsilon_A(C_{m-1})$. Then since we know that $obs_A(C_m)$ also C_{i+1} must be observable. Hence C_i is not the last

step in the trace **ABSURD**. Hence $\neg obs_A(C_{m-1})$. Now by we know that the only way to step from a non-observable configuration to an observable one is with E-RToLab. Hence we know that the last step on m is indeed a E-RToLab.
Since $obs_A(C_n)$ and $obs_A(C_m)$ we know that both traces entered in a E-ToLab step at some point in the trace, say i and j respectively. By $ek^+_A(\epsilon_A(t))$ we know that $\varepsilon_A(C_i) = \varepsilon_A(C_j)$. Now $\varepsilon_A(C_i) =$ `toLabeled` $l \bullet$ and $\varepsilon_A(C_j) =$ `toLabeled` $l' \bullet$ hence $l = l'$. Now we need also to show that the state components along the E-ToLab evaluation are also ε-equal. We know that the semantics doesn't allow the E-SetRel operation in the inner monad, thus we know that in both cases the *rel* component stays the same. Now we can use the requirements to derive that the state components are also ε-equal to the last observable configuration between C_i and C_m, say C_k. Since $obs_A(C_k)$ there exists a C_p between C_j and C_m s.t. $\varepsilon_A(C_p) = \varepsilon_A(C_k)$. Now we have that $\varepsilon_A(\Sigma_k) = \varepsilon_A(\Sigma_n)$ and $\varepsilon_A(\Sigma_p) = \varepsilon_A(\Sigma_m)$ hence from this we derive $\varepsilon_A(\langle \Sigma_n \mid$ `ret (LB` $l\ e\ \Sigma.sid)\rangle) = \varepsilon_A(\langle \Sigma_m \mid$ `ret (LB` $l\ e'\ \Sigma.sid)\rangle)$, since the return value is erased to `LB` $l \bullet \bullet$.

  - **All other cases**: proved using Lemma 3, Lemma 4 and Lemma 7.

- **Inductive Cases i.e, E-Cntx**:
  - $\dfrac{I \vdash \langle \Sigma_{n-1} \mid e_{n-1}\rangle \rightarrow \langle \Sigma_n \mid e_n\rangle}{I \vdash \langle \Sigma_{n-1} \mid e_{n-1} \gg= e_f\rangle \rightarrow \langle \Sigma_n \mid e_n \gg= e_f\rangle}$ **E-Bind**:
  This is the most interesting case since it is here that change in the state has effects on the erasure of the whole expression.
  Hence we need to case analyze the evaluation of e_{n-1} and e'_{n-1}. We know by $\varepsilon_A(\langle \Sigma_{n-1} \mid e_{n-1}\rangle) = \varepsilon_A(\langle \Sigma_{m-1} \mid e_{m-1}\rangle)$ and rule induction that $\varepsilon_A(\langle \Sigma'_n \mid e'_n\rangle) = \varepsilon_A(\langle \Sigma'_m \mid e'_m\rangle)$.
  Hence we need to show that $\varepsilon_A(\langle \Sigma'_n \mid e^f_n\rangle) = \varepsilon_A(\langle \Sigma'_m \mid e^f_m\rangle)$.
    - **Case** $e_{n-1} =$ `setRel` st: The new state now allows information in $A^{st}_{\downarrow}$ to be observable by A, thus we need to show that they are $\epsilon - equal$.
    By $\mathcal{B}_A(t, \alpha)$ we know that anything that could be revealed is $\epsilon - equal$.
    Hence $\varepsilon_A(\langle \Sigma_n \mid e_n\rangle) = \varepsilon_A(\langle \Sigma_m \mid e_m\rangle)$.
    - **Case** $e_{n-1} =$ `toLabRet` $\Sigma_i\ l\ e$: we use the induction hypothesis and proceed as in the base case.
    - **All other cases**: proved by rule induction and Lemma 7.
  - **All other cases**: in the other cases *ps* remains the same. Hence $obs_A(\langle \Sigma_n \mid e_n\rangle)$ implies $obs_A(\langle \Sigma_{n-1} \mid e_{n-1}\rangle)$ and $obs_A(\langle \Sigma_m \mid e_m\rangle)$ implies $obs_A(\langle \Sigma_{m-1} \mid e_{m-1}\rangle)$. Then by $ek^+_A(\epsilon_A(t))$ we derive $\varepsilon_A(\langle \Sigma_{n-1} \mid e_{n-1}\rangle) = \varepsilon_A(\langle \Sigma_{m-1} \mid e_{m-1}\rangle)$. This and Lemma 7 (Fixed State Lemma) give us $\varepsilon_A(\langle \Sigma_n \mid e_n\rangle) = \varepsilon_A(\langle \Sigma_m \mid e_m\rangle)$.

```
data State l ps rel = St
    { lset :: Lset l
    , sid  :: SId l
    , ps   :: ps
    , rel  :: rel
    }

data Expr l ps rel = ELabel (Expr l ps rel) |  ELab l
                   | ERet (Expr l ps rel) | ...

data Program l ps rel = Pg
    { pstate :: State l ps rel
    , pExpr  :: Expr l ps rel
    }
```

Figure 8: Syntax in Liquid Haskell.

```
data Step l ps rel where
    {-@ SLabel :: i:IFC l ps rel -> st:State l ps rel
        -> l:{eGuard i st l} -> Expr l ps rel
        -> Step i (Pg st (ELabel (ELab l) e))
                  (Pg st (ERet (ELB (ELab l) e))) @-}
    SLabel :: IFC l ps rel -> State l ps rel -> l
           -> Expr l ps rel -> Step l ps rel
```

Figure 9: Operational semantics in Liquid Haskell

```
security :: i:IFC l ps rel -> atk:l
    -> p0:Program l ps rel
    -> p0':{Program l ps rel | eqState p0 p0'}
    -> pn1:Program l ps rel -> pm1:Program l ps rel
    -> pn:{Program l ps rel | isObs i atk pn }
    -> pm:{Program l ps rel | isObs i atk pm}
    -> tn1:Eval i p0 pn1 -> tm1:Eval i p0' pm1
    -> sn:Step i pn1 pn -> sm:Step i pm1 pm
    -> LowEquivTrace i atk p0 p0' pn1 pm1 tn1 tm1
    -> InputRelease  i atk (pstate pn) (pExpr p0) (pExpr p0')
    -> SecurityBound i atk p0 p0' pn1 pm1 tn1 tm1
    -> LowEquivProgram i atk pn pm
```

Figure 10: Security Condition in Liquid Haskell

6.3 Mechanization

We use LiquidHaskell [11] to mechanize the security proof of the system. LiquidHaskell is a refinement type checker for Haskell that uses SMT solvers to verify properties of Haskell programs. Refinement types are written as Haskell types with decidable logical predicates, e.g., `{ v:Int | v > 0 }` is the type of positive integers.

The expressions of λ^{DP} are embedded as a data type shown in Figure 8. The operational semantics, an example rule of which is shown in Figure 9, is embedded as a data proposition [3], Liquid Haskell's mechanism for defining inductive predicates. We use the refinement types to express the preconditions and postconditions of the operational semantics. Finally, the security theorem is mechanized as a function in LiquidHaskell, as shown in Figure 10. The proof is still a *work in progress*, with the main challenge being that because of the complexity of the property the proof development is tedious and requires time to complete. Currently, we have ~ 7000 lines of code.

7 IMPLEMENTATION

We implemented *DypLIO* as a State monad in Haskell using typeclasses that directly reflect the formalization and allow for the

```haskell
type Id = Int
data Labeled l a = LB l a Id
type IFC l st ps rel m = (MonadState st m, IFCI l st ps rel)

label :: IFC l st ps rel m => l -> a -> m (Labeled l a)
label l v = do
    st <- get
    unless (guard l st) (throwError "label check failed1")
    put (incId l  opLabel l (getId l st) st)
    return  LB l v (getId l st)
unlabel :: IFC l st ps rel m => Labeled l a -> m a
unlabel (LB l e i) = do
    st <- get
    put $ opUnlabel l i (modifyLSet' (insert (l,i) ) st)
    return e
toLabeled :: IFC l st ps rel m => l -> m a -> m (Labeled l a)
toLabeled l m = do
    st0 <- get
    e <- m
    toLabeledRet st0 l e
setRelation :: IFC l st ps rel m => rel -> m ()
setRelation rel = do
    st <- get
    let st' = opState st rel
    if stateGuard st st' then throwError "state guard fail"
    else put st'
getRelation :: IFC l st ps rel m => m rel
getRelation = getRelation' <$> get
-- Hidden
toLabRet :: IFC l st ps rel m => st -> l -> a -> m (Labeled l a)
toLabRet st0 l e = do
    st <- get
    unless (guard l st) (throwError "label check failed")
    put (incId l  opToLabRet st l (getId l st) st0)
    return  LB l e (getId l st)
```

Figure 11: *DypLIO* **API**

```haskell
class Relation l rel where
    lrt :: rel -> l -> l -> Bool

class Relation l rel => IFCI l st ps rel where
    guard :: l -> st -> Bool
    stateGuard :: st -> st -> Bool
    opState :: st -> rel -> st
    opLabel :: l -> Id -> st -> st
    opUnlabel :: l -> Id -> st -> st
    opToLabeledRet :: st -> l -> Id -> st -> st
    policy :: l -> Id -> l -> st -> l
```

Figure 12: IFC inferface typeclass

different implementations. Figure 11 shows the API of *DypLIO*, with direct translation of the rules from the formalization.

Then, we show the `Relation` and `IFCI` typeclasses in Figure 12 that represent the relation R and abstract interface I respectively. Finally, our implementation provides the instances for the 4 systems presented in Section 5.

8 RELATED WORK

In this section, we discuss related work on Information Flow Control and dynamic policies. *DypLIO* is inspired by SLIO [5] an Haskell IFC enforcement mechanism for dynamic polcies and [4] a survey where different security conditions are discussed with respect to facets of dynamic policies. The base for knowledge based IFC is [2, 6] where they introduce the concept of attacker knowledge, i.e. the set of possible secret inputs as functions of publicly observable outputs, and the knowledge based security condition that only allows the attacker to learn information in accordance to the current policy. In [6] they also observe that different types of attackers model different types of security. This is further explored in [1] where they also analyze the relations of different attackers with the facets identified in [4]. They propose a formalization with a while language with input channels, and SMT-based verification. Another work on dynamic policies is [7] where they propose a general-purpose security condition for an imperative language. In [10] they extend the framework from [6] to account for progress-insensitive security.

9 CONCLUSION & FUTURE WORK

We presented a modular framework for IFC with dynamic policies, that can be instantiated with different interpretations of security. Concretely, we formalized the core language of an enforcement mechanism that is parametric with respect to the information flow instance, such that can enforce the four facets of time-transitive flow, non-replaying, replaying, and non-time-transitive flow. We encoded the security conditions for each of these facets, and initiated the mechanization of the security proof in Liquid Haskell.

In the immediate future, we plan to complete the mechanization of the security proof for the time transitive flow facet and next, modify the mechanization to consider the other three facets, thus testing our claim that the system is modular with respect to the facets and that the main proof can be reused for the different facets. Once this framework is complete, we plan to explore the composition and combination of the different facets as well as extend the framework with more language features, such as exceptions, references, and concurrency.

REFERENCES

[1] A. M. Ahmadian and M. Balliu. 2022. Dynamic Policies Revisited. In *2022 IEEE 7th European Symposium on Security and Privacy (EuroS&P)*. IEEE Computer Society, Los Alamitos, CA, USA, 448–466. https://doi.org/10.1109/EuroSP53844.2022.00035

[2] Aslan Askarov and Andrei Sabelfeld. 2007. Gradual Release: Unifying Declassification, Encryption and Key Release Policies. In *2007 IEEE Symposium on Security and Privacy (SP '07)*. 207–221. https://doi.org/10.1109/SP.2007.22

[3] Michael H. Borkowski, Niki Vazou, and Ranjit Jhala. 2024. Mechanizing Refinement Types. *Proc. ACM Program. Lang.* 8, POPL, Article 70 (jan 2024), 30 pages. https://doi.org/10.1145/3632912

[4] N. Broberg, B. van Delft, and D. Sands. 2015. The Anatomy and Facets of Dynamic Policies. In *2015 IEEE 28th Computer Security Foundations Symposium (CSF)*. IEEE Computer Society, Los Alamitos, CA, USA, 122–136. https://doi.org/10.1109/CSF.2015.16

[5] Pablo Buiras and Bart van Delft. 2015. Dynamic Enforcement of Dynamic Policies. In *Proceedings of the 10th ACM Workshop on Programming Languages and Analysis for Security* (Prague, Czech Republic) *(PLAS'15)*. Association for Computing Machinery, New York, NY, USA, 28–41. https://doi.org/10.1145/2786558.2786563

[6] S. Chong and A. Askarov. 2012. Learning is Change in Knowledge: Knowledge-Based Security for Dynamic Policies. In *2012 IEEE 25th Computer Security Foundations Symposium*. IEEE Computer Society, Los Alamitos, CA, USA, 308–322. https://doi.org/10.1109/CSF.2012.31

[7] P. Li and D. Zhang. 2022. Towards a General-Purpose Dynamic Information Flow Policy. In *2022 2022 IEEE 35th Computer Security Foundations Symposium (CSF) (CSF)*. IEEE Computer Society, Los Alamitos, CA, USA, 260–275. https://doi.org/10.1109/CSF54842.2022.9919639

[8] Deian Stefan, Alejandro Russo, John C. Mitchell, and David Mazières. 2011. Flexible dynamic information flow control in Haskell. *SIGPLAN Not.* 46, 12 (sep 2011), 95–106. https://doi.org/10.1145/2096148.2034688

[9] Deian Stefan, Alejandro Russo, John C. Mitchell, and David Mazières. 2011. Flexible dynamic information flow control in Haskell. In *Proceedings of the 4th ACM Symposium on Haskell* (Tokyo, Japan) *(Haskell '11)*. Association for Computing Machinery, New York, NY, USA, 95–106. https://doi.org/10.1145/2034675.2034688

[10] Bart van Delft, Sebastian Hunt, and David Sands. 2015. Very Static Enforcement of Dynamic Policies. *CoRR* abs/1501.02633 (2015). arXiv:1501.02633 http://arxiv.org/abs/1501.02633

[11] Niki Vazou, Eric L. Seidel, Ranjit Jhala, Dimitrios Vytiniotis, and Simon Peyton-Jones. 2014. Refinement Types for Haskell. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming* (Gothenburg, Sweden) *(ICFP '14)*. Association for Computing Machinery, New York, NY, USA, 269–282. https://doi.org/10.1145/2628136.2628161