# Typed-based Declassification for Free

Minh Ngo[1,2], David Naumann[1], and Tamara Rezk[2]

[1] Stevens Institute of Technology
[2] Inria

**Abstract.** This work provides a study to demonstrate the potential of using off-the-shelf programming languages and their theories to build sound language-based-security tools. Our study focuses on information flow security encompassing declassification policies that allow us to express flexible security policies needed for practical requirements. We translate security policies, with declassification, into an interface for which an unmodified standard typechecker can be applied to a source program—if the program typechecks, it provably satisfies the policy. Our proof reduces security soundness -with declassification- to the mathematical foundation of data abstraction, Reynolds' abstraction theorem.

## 1 Introduction

A longstanding challenge for software systems is the enforcement of security in applications implemented in conventional general-purpose programming languages. For high assurance, precise mathematical definitions are needed for policies, enforcement mechanism, and program semantics. The latter, in particular, is a major challenge for languages in practical use. In order to minimize the cost of assurance, especially over time as systems evolve, it is desirable to leverage work on formal modeling with other goals such as functional verification, equivalence checking, and compilation.

To be auditable by stakeholders, policy should be expressed in an accessible way. This is one of several reasons why types play an important role in many works on information flow (IF) security. For example, Flowcaml [32] and Jif [26] express policy using types that include IF labels. They statically enforce policy using dedicated IF type checking and inference. Techniques from type theory are also used in security proofs such as those for Flowcaml and the calculus DCC [1].

IF is typically formalized as the preservation of indistinguishability relations between executions. Researchers have hypothesized that this should be an instance of a celebrated semantics basis in type theory: relational parametricity [35]. Relational parametricity provides an effective basis for formal reasoning about program transformations [47], representation independence and information hiding for program verification [25,6]. The connection between IF and relational parametricity has been made precise in 2015, for DCC, by translation to the calculus $F_\omega$ and use of the existing parametricity theorem for $F_\omega$ [13]. The connection is also made, perhaps more transparently, in a translation of

DCC to dependent type theory, specifically the calculus of constructions and its parametricity theorem [4].

In this work, we advance the state of the art in the connection between IF and relational parametricity, guided by three main goals. One of the goals motivating our work is to *reduce the burden of defining dedicated type checking, inference, and security proofs* for high assurance in programming languages. A promising approach towards this goal is the idea of leveraging type abstraction to enforce policy, and in particular, *leveraging the parametricity theorem to obtain security guarantees.* A concomitant goal is *to do so for practical IF policies* that encompass selective declassification, which is needed for most policies in practice. For example, a password checker program or a program that calculates aggregate or statistical information must be considered insecure without declassification.

To build on the type system and formal theory of a language without *a priori* IF features, policy needs to be encoded somehow, and the program may need to be transformed. For example, to prove that a typechecked DCC term is secure with respect to the policy expressed by its type, Bowman and Ahmed [13] encode the typechecking judgment by nontrivial translation of both types and terms into types and terms of $F_\omega$. Any translation becomes part of the assurance argument. Most likely, complicated translation will also make it more difficult to use extant type checking/inference (and other development tools) in diagnosing security errors and developing secure code. This leads us to highlight a third goal, needed to achieve the first goal, namely to *minimize the complexity of translation.*

There is a major impediment to leveraging type abstraction: few languages are relationally parametric or have parametricity theorems. The lack of parametricity can be addressed by focusing on well behaved subsets and leveraging additional features like ownership types that may be available for other purposes (e.g., in the Rust language). As for the paucity of parametricity theorems, we take hope in the recent advances in machine-checked metatheory, such as correctness of the CakeML and CompCert compilers, the VST logic for C, the relational logic of Iris. For parametricity specifically, the most relevant work is Crary's formal proof of parametricity for the ML module calculus [15].

*Contributions* Our *first contribution* is to translate policies with declassification—in the style of relaxed noninterference [24]—into abstract types in a functional language, in such a way that typechecking the original program implies its security. For doing so, we neither rely on a specialized security type system [13] nor on modifications of existing type systems [16]. A program that typechecks may use the secret inputs parametrically, e.g., storing in data structures, but cannot look at the data until declassification has been applied. Our *second contribution* is to prove security by direct application of a parametricity theorem. We carry out this development for the polymorphic lambda calculus, using the original theorem of Reynolds. (We provide an appendix that shows this development for the ML module calculus using Crary's theorem [15], enabling the use of ML to check security.)

## 2 Background: Language and Abstraction Theorem

To present our results we choose the simply typed and call-by-value lambda calculus, with integers and type variables, because of two reasons: (1) the chosen language is similar to the language used in the paper of Reynolds [35] where the abstraction theorem was first proven, and (2) we want to illustrate our encoding approach (§4) in a minimal calculus. This section defines the language we use and recalls the abstraction theorem, a.k.a. parametricity. Our language is very close to the one in Reynolds [35, § 2]; we prove the abstraction theorem using contemporary notation[3].

*Language* The syntax of the language is as below, where $\alpha$ denotes a type variable, $x$ a term variable, and $n$ an integer value. A value is *closed* when there is no free term variable in it. A type is *closed* when there is no type variable in it.

$$\begin{array}{rll}
\tau ::= & \mathbf{int} \mid \alpha \mid \tau_1 \times \tau_2 \mid \tau_1 \to \tau_2 & \text{Types} \\
v ::= & n \mid \langle v, v \rangle \mid \lambda x : \tau.e & \text{Values} \\
e ::= & x \mid v \mid \langle e, e \rangle \mid \pi_i e \mid e_1 e_2 & \text{Terms} \\
E ::= & [.] \mid \langle E, e \rangle \mid \langle v, E \rangle \mid \pi_i E \mid E\ e \mid v\ E & \text{Eval. Contexts}
\end{array}$$

We use small-step semantics, with the reduction relation $\rightharpoonup$ defined inductively by these rules.

$$\pi_i\langle v_1, v_2\rangle \rightharpoonup v_i \qquad (\lambda x : \tau.e)v \rightharpoonup e[x \mapsto v] \qquad \frac{e \rightharpoonup e'}{E[e] \rightharpoonup E[e']}$$

We write $e[x \mapsto e']$ for capture-avoiding substitution of $e'$ for free occurrences of $x$ in $e$. We use parentheses to disambiguate term structure and write $\rightharpoonup^*$ for the reflexive, transitive closure of $\rightharpoonup$.

A *typing context* $\Delta$ is a set of type variables. A *term context* $\Gamma$ is a mapping from term variables to types. We write $\Delta \vdash \tau$ to mean that $\tau$ is *well-formed w.r.t.* $\Delta$, that is, all type variables in $\tau$ are in $\Delta$. We say that $e$ is *typable w.r.t.* $\Delta$ *and* $\Gamma$ (denoted by $\Delta, \Gamma \vdash e$) when there exists a well-formed type $\tau$ such that $\Delta, \Gamma \vdash e : \tau$. The derivable typing judgments are defined inductively in Fig. 1. The rules are to be instantiated only with $\Gamma$ that is well-formed under $\Delta$, in the sense that $\Delta \vdash \Gamma(x)$ for all $x \in \mathit{dom}(\Gamma)$. When the term context and the type context are empty, we write $\vdash e : \tau$.

*Logical relation* A logical relation is a type-indexed family of relations on values, based on given relations for type variables. From it, we derive a relation on terms. The abstraction theorem says the latter is reflexive.

[3] See appendix. Some readers may find it helpful to consult references such as these for background on logical relations and parametricity: [22, Chapt. 49], [25, Chapt. 8], [14], [30].

$$\text{FT-INT}\ \frac{}{\Delta, \Gamma \vdash n : \mathbf{int}} \qquad \text{FT-VAR}\ \frac{x : \tau \in \Gamma}{\Delta, \Gamma \vdash x : \tau}$$

$$\text{FT-PAIR}\ \frac{\Delta, \Gamma \vdash e_1 : \tau_1 \qquad \Delta, \Gamma \vdash e_2 : \tau_2}{\Delta, \Gamma \vdash \langle e_1, e_2 \rangle : \tau_1 \times \tau_2} \qquad \text{FT-PRJ}\ \frac{\Delta, \Gamma \vdash e : \tau_1 \times \tau_2}{\Delta, \Gamma \vdash \pi_i e : \tau_i}$$

$$\text{FT-FUN}\ \frac{\Delta, \Gamma, x : \tau_1 \vdash e : \tau_2}{\Delta, \Gamma \vdash \lambda x : \tau_1 . e : \tau_1 \to \tau_2}$$

$$\text{FT-APP}\ \frac{\Delta, \Gamma \vdash e_1 : \tau_1 \to \tau_2 \qquad \Delta, \Gamma \vdash e_2 : \tau_1}{\Delta, \Gamma \vdash e_1\ e_2 : \tau_2}$$

**Fig. 1.** Typing rules

Let $\gamma$ be a *term substitution*, i.e., a finite map from term variables to closed values, and $\delta$ be a *type substitution*, i.e., a finite map from type variables to closed types. In symbols:

$$\begin{array}{ll} \gamma ::= . \mid \gamma, x \mapsto v & \text{Term Substitutions} \\ \delta ::= . \mid \delta, \alpha \mapsto \tau, \ \text{where} \vdash \tau & \text{Type Substitutions} \end{array}$$

We say $\gamma$ *respects* $\Gamma$ (denoted by $\gamma \models \Gamma$) when $dom(\gamma) = dom(\Gamma)$ and $\vdash \gamma(x) : \Gamma(x)$ for any $x$. We say $\delta$ *respects* $\Delta$ (denoted by $\delta \models \Delta$) when $dom(\delta) = \Delta$. Let $Rel(\tau_1, \tau_2)$ be the set of all binary relations over closed values of closed types $\tau_1$ and $\tau_2$. Let $\rho$ be an *environment*, a mapping from type variables to relations $R \in Rel(\tau_1, \tau_2)$. We write $\rho \in Rel(\delta_1, \delta_2)$ to say that $\rho$ is compatible with $\delta_1, \delta_2$ as follows: $\rho \in Rel(\delta_1, \delta_2) \triangleq dom(\rho) = dom(\delta_1) = dom(\delta_2) \wedge \forall \alpha \in dom(\rho). \rho(\alpha) \in Rel(\delta_1(\alpha), \delta_2(\alpha))$. The logical relation is inductively defined in Fig. 2, where $\rho \in Rel(\delta_1, \delta_2)$ for some $\delta_1$ and $\delta_2$. For any $\tau$, $[\![\tau]\!]_\rho$ is a relation on closed values. In addition, $[\![\tau]\!]^{\mathsf{ev}}_\rho$ is a relation on terms.

**Lemma 1.** *Suppose that $\rho \in Rel(\delta_1, \delta_2)$ for some $\delta_1$ and $\delta_2$. For $i \in \{1, 2\}$, it follows that:*

- *if $\langle v_1, v_2 \rangle \in [\![\tau]\!]_\rho$, then $\vdash v_i : \delta_i(\tau)$, and*
- *if $\langle e_1, e_2 \rangle \in [\![\tau]\!]^{\mathsf{ev}}_\rho$, then $\vdash e_i : \delta_i(\tau)$.*

We write $\delta(\Gamma)$ to mean a term substitution obtained from $\Gamma$ by applying $\delta$ on the range of $\Gamma$, i.e.:

$$dom(\delta(\Gamma)) = dom(\Gamma) \text{ and } \forall x \in dom(\Gamma) : \delta(\Gamma)(x) = \delta(\Gamma(x)).$$

Suppose that $\Delta, \Gamma \vdash e : \tau$, $\delta \models \Delta$, and $\gamma \models \delta(\Gamma)$. Then we write $\delta\gamma(e)$ to mean the application of $\delta$ and $\gamma$ to $e$. For example, suppose that $\delta(\alpha) = \mathbf{int}$, $\gamma(x) = n$ for some $n$, and $\alpha, x : \alpha \vdash \lambda y : \alpha.x : \alpha \to \alpha$, then $\delta\gamma(\lambda y : \alpha.x) = \lambda y : \mathbf{int}.n$. We write $\langle \gamma_1, \gamma_2 \rangle \in [\![\Gamma]\!]_\rho$ for some $\rho \in Rel(\delta_1, \delta_2)$ when $\gamma_1 \models \delta_1(\Gamma)$, $\gamma_2 \models \delta_2(\Gamma)$, and $\langle \gamma_1(x), \gamma_2(x) \rangle \in [\![\Gamma(x)]\!]_\rho$ for all $x \in dom(\Gamma)$.

$$\text{FR-INT}\ \frac{}{\langle n, n\rangle \in [\![\mathbf{int}]\!]_\rho} \qquad \text{FR-PAIR}\ \frac{\langle v_1, v_1'\rangle \in [\![\tau_1]\!]_\rho \quad \langle v_2, v_2'\rangle \in [\![\tau_2]\!]_\rho}{\langle\langle v_1, v_2\rangle, \langle v_1', v_2'\rangle\rangle \in [\![\tau_1 \times \tau_2]\!]_\rho}$$

$$\text{FR-FUN}\ \frac{\forall\langle v_1', v_2'\rangle \in [\![\tau_1]\!]_\rho.\langle v_1\ v_1', v_2\ v_2'\rangle \in [\![\tau_2]\!]_\rho^{\mathsf{ev}}}{\langle v_1, v_2\rangle \in [\![\tau_1 \to \tau_2]\!]_\rho}$$

$$\text{FR-VAR}\ \frac{\langle v_1, v_2\rangle \in R \in Rel(\tau_1, \tau_2)}{\langle v_1, v_2\rangle \in [\![\alpha]\!]_{\rho[\alpha\mapsto R]}}$$

$$\text{FR-TERM}\ \frac{\vdash e_1 : \delta_1(\tau) \quad \vdash e_2 : \delta_2(\tau) \quad e_1 \twoheadrightarrow^* v_1 \quad e_2 \twoheadrightarrow^* v_2 \quad \langle v_1, v_2\rangle \in [\![\tau]\!]_\rho}{\langle e_1, e_2\rangle \in [\![\tau]\!]_\rho^{\mathsf{ev}}}$$

**Fig. 2.** Logical relation

**Definition 1 (Logical equivalence).** *Terms $e$ and $e'$ are* logically equivalent *at $\tau$ in $\Delta$ and $\Gamma$ (written $\Delta, \Gamma \vdash e \sim e' : \tau$) if $\Delta, \Gamma \vdash e : \tau$, $\Delta, \Gamma \vdash e' : \tau$, and for all $\delta_1, \delta_2 \models \Delta$, all $\rho \in Rel(\delta_1, \delta_2)$, and all $\langle\gamma_1, \gamma_2\rangle \in [\![\Gamma]\!]_\rho$, we have $\langle\delta_1\gamma_1(e), \delta_2\gamma_2(e')\rangle \in [\![\tau]\!]_\rho^{\mathsf{ev}}$.*

**Theorem 1 (Abstraction [35]).** *If $\Delta, \Gamma \vdash e : \tau$, then $\Delta, \Gamma \vdash e \sim e : \tau$.*

## 3 Declassification Policies

Confidentiality policies can be expressed by information flows of confidential sources to public sinks in programs. Confidential sources correspond to the secrets that the program receives and public sinks correspond to any results given to a public observer, a.k.a. the attacker. These flows can either be direct -e.g. if a function, whose result is public, receives a confidential value as input and directly returns the secret- or indirect -e.g. if a function, whose result is public, receives a confidential boolean value and returns 0 if the confidential value is false and 1 otherwise. Classification of program sources as confidential or public, a.k.a. *security policy*, must be given by the programmer or security engineer: for a given security policy the program is said to be secure for *noninterference* if public resources do not depend on confidential ones. Thus, noninterference for a program means total independence between public and confidential information. As simple and elegant as this information flow policy is, noninterference does not permit to consider as secure programs that purposely need to release information in a controlled way: for example a password-checker function that receives as confidential input a boolean value representing if the system password is equal to the user's input and returns 0 or 1 accordingly. In order to consider such intended dependences of public sinks from confidential sources, we need to consider more relaxed security policies than noninterference, a.k.a. *declassification policies*. Declassification security policies allow us to specify controlled ways to release confidential inputs [37].

Declassification policies that we consider in this work map confidential inputs to functions, namely *declassification functions.* These functions allow the programmer to specify what and how information can be released. The formal syntax for declassification functions in this work is given below,[4] where $n$ is an integer value, and $\oplus$ represents primitive arithmetic operators.

$$
\begin{array}{rll}
\tau ::= & \mathbf{int} \mid \tau \to \tau & \text{Types} \\
e ::= & \lambda x : \tau . e \mid e\ e \mid x \mid n \mid e \oplus e & \text{Terms} \\
f, g ::= & \lambda x : \mathbf{int} . e & \text{Declass. Functions}
\end{array}
$$

The static and dynamic semantics are standard. To simplify the presentation we suppose that the applications of primitive operators on well-typed arguments terminates. Therefore, the evaluations of declassification functions on values terminate.

For policies we refrain from using concrete syntax and instead give a simple formalization that facilitates later definitions.

**Definition 2 (Policy).** *A policy $\mathcal{P}$ is a tuple $\langle \mathbf{V}_\mathcal{P}, \mathbf{F}_\mathcal{P} \rangle$, where $\mathbf{V}_\mathcal{P}$ is a finite set of variables for confidential inputs, and $\mathbf{F}_\mathcal{P}$ is a partial mapping from variables in $\mathbf{V}_\mathcal{P}$ to declassification functions.*

For simplicity we require that if $f$ appears in the policy then it is a closed term of type $\mathbf{int} \to \tau_f$ for some $\tau_f$.

In the definition of policies, if a confidential input is not associated with a declassification function, then it cannot be declassified.

*Example 1 (Policy $\mathcal{P}_{OE}$ using $f$).* Consider policy $\mathcal{P}_{OE}$ given by $\langle \mathbf{V}_{\mathcal{P}_{OE}}, \mathbf{F}_{\mathcal{P}_{OE}} \rangle$ where $\mathbf{V}_{\mathcal{P}_{OE}} = \{x\}$ and $\mathbf{F}_{\mathcal{P}_{OE}}(x) = f = \lambda x : \mathbf{int}.\, x \mathit{mod}\, 2$. Policy $\mathcal{P}_{OE}$ states that only the parity of the confidential input $x$ can be released to a public observer.

## 4 Type-based Declassification

In this section, we show how to encode declassification policies as standard types in the language of § 2, we define and we prove our free theorem. We consider the information flow security property, with declassification, called typed-based relaxed noninterference (TRNI) and taken from Cruz et al [16].

It is important to notice that our developement, in this section, studies the reuse for security of standard programming languages type systems together with soundness proofs for security for free by using the abstraction theorem. In contrast, Cruz et al [16] use a modified type system for security and prove soundness from scratch, without apealing to parametricity.

Through this section, we consider a fixed policy $\mathcal{P}$ (see Def. 2) given by $\langle \mathbf{V}_\mathcal{P}, \mathbf{F}_\mathcal{P} \rangle$. We treat free variables in a program as inputs and, without loss of generality, we assume that there are two kinds of inputs: integer values, which are

[4] In this paper, the type of confidential inputs is $\mathbf{int}$.

considered as confidential, and declassification functions, which are fixed according to policy. A public input can be encoded as a confidential input that can be declassified via the identity function. We consider terms without type variables as source programs. That is we consider terms $e$ s.t. for all type substitutions $\delta$, $\delta(e)$ is syntactically the same as $e$. [5]

### 4.1 Views and indistinguishability

In order to define TRNI we define two term contexts, called the confidential view and public view. The first view represents an observer that can access to all confidential inputs, while the second one represents an observer that can only observe declassified inputs. The views are defined using fresh term and type variables.

*Confidential view* Let $\mathbf{V}_\top = \{x \mid x \in \mathbf{V}_\mathcal{P} \setminus \mathit{dom}(\mathbf{F}_\mathcal{P})\}$ be the set of inputs that cannot be declassified. First we define the encoding for these inputs as a term context:

$$\Gamma^\mathcal{P}_{C,\top} \triangleq \{x : \mathbf{int} \mid x \in \mathbf{V}_\top\}.$$

Next, we specify the encoding of confidential inputs that can be declassified. To this end, define $\langle\!\langle \_, \_ \rangle\!\rangle_C$ as follows, where $f : \mathbf{int} \to \tau_f$ is in $\mathcal{P}$.

$$\langle\!\langle x, f \rangle\!\rangle_C \triangleq \{x : \mathbf{int}, x_f : \mathbf{int} \to \tau_f\}$$

Finally, we write $\Gamma^\mathcal{P}_C$ for the term context encoding the confidential view for $\mathcal{P}$.

$$\Gamma^\mathcal{P}_C \triangleq \Gamma^\mathcal{P}_{C,\top} \cup \bigcup_{x \in \mathit{dom}(\mathbf{F}_\mathcal{P})} \langle\!\langle x, \mathbf{F}_\mathcal{P}(x) \rangle\!\rangle_C.$$

We assume that, for any $x$, the variable $x_f$ in the result of $\langle\!\langle x, \mathbf{F}_\mathcal{P}(x) \rangle\!\rangle_C$ is distinct from the variables in $\mathbf{V}_\mathcal{P}$, distinct from each other, and distinct from $x_{f'}$ for distinct $f'$. We make analogous assumptions in later definitions.

From the construction, $\Gamma^\mathcal{P}_C$ is a mapping, and for any $x \in \mathit{dom}(\Gamma^\mathcal{P}_C)$, it follows that $\Gamma^\mathcal{P}_C(x)$ is a closed type. Therefore, $\Gamma^\mathcal{P}_C$ is well-formed for the empty set of type variables, so it can be used in typing judgments of the form $\Gamma^\mathcal{P}_C \vdash e : \tau$.

*Example 2 (Confidential view).* For $\mathcal{P}_{OE}$ in Example 1, the confidential view is: $\Gamma^{\mathcal{P}_{OE}}_C = x : \mathbf{int}, x_f : \mathbf{int} \to \mathbf{int}$.

*Public view* The basic idea is to encode policies by using type variables. First we define the encoding for confidential inputs that cannot be declassified. We define a set of type variables, $\Delta^\mathcal{P}_{P,\top}$ and a mapping $\Gamma^\mathcal{P}_{P,\top}$ for confidential inputs that cannot be declassified.

$$\Delta^\mathcal{P}_{P,\top} \triangleq \{\alpha_x \mid x \in \mathbf{V}_\top\} \qquad \Gamma^\mathcal{P}_{P,\top} \triangleq \{x : \alpha_x \mid x \in \mathbf{V}_\top\}$$

[5] An example of a term with type variables is $\lambda x : \alpha.x$. We can easily check that there exists a type substitutions $\delta$ s.t. $\delta(e)$ is syntactically different from $e$ (e.g. for $\delta$ s.t. $\delta(\alpha) = \mathbf{int}$, $\delta(e) = \lambda x : \mathbf{int}.x$).

This gives the program access to $x$ at an opaque type.

In order to define the encoding for confidential inputs that can be declassified, we define $\langle\!\langle \_, \_ \rangle\!\rangle_P$:

$$\langle\!\langle x, f \rangle\!\rangle_P \triangleq \langle \{\alpha_f\}, \{x : \alpha_f, x_f : \alpha_f \to \tau_f\}\rangle$$

The first form will serve to give the program access to $x$ only via function variable $x_f$ that we will ensure is interpreted as the policy function $f$. We define a type context $\Delta_P^{\mathcal{P}}$ and term context $\Gamma_P^{\mathcal{P}}$ that comprise the public view, as follows.

$$\langle \Delta_P^{\mathcal{P}}, \Gamma_P^{\mathcal{P}} \rangle \triangleq \langle \Delta_{P,\top}^{\mathcal{P}}, \Gamma_{P,\top}^{\mathcal{P}} \rangle \cup \bigcup_{x \in dom(\mathbf{F}_{\mathcal{P}})} \langle\!\langle x, \mathbf{F}_{\mathcal{P}}(x) \rangle\!\rangle_P,$$

where $\langle S_1, S_1' \rangle \cup \langle S_2, S_2' \rangle = \langle S_1 \cup S_2, S_1' \cup S_2' \rangle$.

*Example 3 (Public view).* For $\mathcal{P}_{OE}$, the typing context in the public view has one type variable: $\Delta_P^{\mathcal{P}_{OE}} = \alpha_f$. The term context in the public view is $\Gamma_P^{\mathcal{P}_{OE}} = x : \alpha_f,\ x_f : \alpha_f \to \mathbf{int}$.

From the construction, $\Gamma_P^{\mathcal{P}}$ is a mapping, and for any $x \in dom(\Gamma_P^{\mathcal{P}})$, it follows that $\Gamma_P^{\mathcal{P}}(x)$ is well-formed in $\Delta_P^{\mathcal{P}}$ (i.e. $\Delta_P^{\mathcal{P}} \vdash \Gamma_P^{\mathcal{P}}(x)$). Thus, $\Gamma_P^{\mathcal{P}}$ is well-formed in the typing context $\Delta_P^{\mathcal{P}}$. Therefore, $\Delta_P^{\mathcal{P}}$ and $\Gamma_P^{\mathcal{P}}$ can be used in typing judgments of the form $\Delta_P^{\mathcal{P}}, \Gamma_P^{\mathcal{P}} \vdash e : \tau$.

Notice that in the public view of a policy, types of variables for confidential inputs are not **int**. Thus, the public view does not allow programs where concrete declassifiers are applied to confidential input variables even when the applications are semantically correct according to the policy (e.g. for $\mathcal{P}_{OE}$, the program $f\ x$ does not typecheck in the public view). Instead, programs should apply named declassifers (e.g. for $\mathcal{P}_{OE}$, the program $x_f\ x$ is well-typed in the public view).

*Indistinguishability* The security property TRNI is defined in a usual way, using partial equivalence relations called indistinguishability. To define indistinguishability, we define a type substitution $\delta_{\mathcal{P}}$ such that $\delta_{\mathcal{P}} \models \Delta_P^{\mathcal{P}}$, as follows:

$$\text{for all } \alpha_x, \alpha_f \text{ in } \Delta_P^{\mathcal{P}},\ \text{let } \delta_{\mathcal{P}}(\alpha_x) = \delta_{\mathcal{P}}(\alpha_f) = \mathbf{int}. \qquad (1)$$

The inductive definition of indistinguishability for a policy $\mathcal{P}$ is presented in Figure 3, where $\alpha_x$ and $\alpha_f$ are from $\Delta_P^{\mathcal{P}}$. Indistinguishability is defined for $\tau$ s.t. $\Delta_P^{\mathcal{P}}, \Gamma_P^{\mathcal{P}} \vdash \tau$. The definitions of indistinguishability for **int** and $\tau_1 \times \tau_2$ are straightforward. We say that two functions are indistinguishable at $\tau_1 \to \tau_2$ if on any indistinguishable inputs they generate indistinguishable outputs. Since we use $\alpha_x$ to encode confidential integer values that cannot be declassified, any integer values $v_1$ and $v_2$ are indistinguishable, according to rule Eq-Var1. Notice that $\delta_{\mathcal{P}}(\alpha_x) = \mathbf{int}$. Since we use $\alpha_f$ to encode confidential integer values that can be declassified via $f$ where $\vdash f : \mathbf{int} \to \tau_f$, we say that $\langle v_1, v_2 \rangle \in \mathcal{I}_V[\![\alpha_f]\!]$ when $\langle f\ v_1, f\ v_2 \rangle \in \mathcal{I}_E[\![\tau_f]\!]$.

$$\text{Eq-Int}\ \frac{}{\langle n, n\rangle \in \mathcal{I}_V[\![\mathbf{int}]\!]} \qquad \text{Eq-Pair}\ \frac{\langle v_1, v_1'\rangle \in \mathcal{I}_V[\![\tau_1]\!] \qquad \langle v_2, v_2'\rangle \in \mathcal{I}_V[\![\tau_2]\!]}{\langle\langle v_1, v_2\rangle, \langle v_1', v_2'\rangle\rangle \in \mathcal{I}_V[\![\tau_1 \times \tau_2]\!]}$$

$$\text{Eq-Fun}\ \frac{\forall\langle v_1', v_2'\rangle : \langle v_1', v_2'\rangle \in \mathcal{I}_V[\![\tau_1]\!].\langle v_1\ v_1', v_2\ v_2'\rangle \in \mathcal{I}_E[\![\tau_2]\!]}{\langle v_1, v_2\rangle \in \mathcal{I}_V[\![\tau_1 \to \tau_2]\!]}$$

$$\text{Eq-Var1}\ \frac{\vdash v_1, v_2 : \delta_{\mathcal{P}}(\alpha_x)}{\langle v_1, v_2\rangle \in \mathcal{I}_V[\![\alpha_x]\!]} \qquad \text{Eq-Var2}\ \frac{\vdash v_1, v_2 : \delta_{\mathcal{P}}(\alpha_f) \qquad \langle f\ v_1, f\ v_2\rangle \in \mathcal{I}_E[\![\tau_f]\!]}{\langle v_1, v_2\rangle \in \mathcal{I}_V[\![\alpha_f]\!]}$$

$$\text{Eq-Term}\ \frac{\vdash e_1, e_2 : \delta_{\mathcal{P}}(\tau) \qquad e_1 \rightarrowtriangle^* v_1 \qquad e_2 \rightarrowtriangle^* v_2 \qquad \langle v_1, v_2\rangle \in \mathcal{I}_V[\![\tau]\!]}{\langle e_1, e_2\rangle \in \mathcal{I}_E[\![\tau]\!]}$$

**Fig. 3.** Indistinguishability

*Example 4 (Indistinguishability).* For $\mathcal{P}_{OE}$ (of Example 1), two values $v_1$ and $v_2$ are indistinguishable at $\alpha_f$ when both of them are even numbers or odd numbers.

$$\mathcal{I}_V[\![\alpha_f]\!] = \{\langle v_1, v_2\rangle \mid\ \vdash v_1 : \mathbf{int},\ \vdash v_2 : \mathbf{int},\ (v_1\ \mathit{mod}\ 2) =_{\mathbf{int}} (v_2\ \mathit{mod}\ 2)\}.$$

We write $e_1 =_{\mathbf{int}} e_2$ to mean that $e_1 \rightarrowtriangle^* v$ and $e_2 \rightarrowtriangle^* v$ for some integer value $v$.

Term substitutions $\gamma_1$ and $\gamma_2$ are called *indistinguishable w.r.t.* $\mathcal{P}$ (denoted by $\langle\gamma_1, \gamma_2\rangle \in \mathcal{I}_V[\![\mathcal{P}]\!]$) if the following hold.

- $\gamma_1 \models \delta_{\mathcal{P}}(\Gamma_P^{\mathcal{P}})$ and $\gamma_2 \models \delta_{\mathcal{P}}(\Gamma_P^{\mathcal{P}})$,
- for all $x_f \in \mathit{dom}(\Gamma_P^{\mathcal{P}})$, $\gamma_1(x_f) = \gamma_2(x_f) = f$,
- for all other $x \in \mathit{dom}(\Gamma_P^{\mathcal{P}})$, $\langle\gamma_1(x), \gamma_2(x)\rangle \in \mathcal{I}_V[\![\Gamma_P^{\mathcal{P}}(x)]\!]$.

Note that each $\gamma_i$ maps $x_f$ to the specific functions $f$ in the policy. Input variables are mapped to indistinguishable values.

We now define type-based relaxed noninterference w.r.t. $\mathcal{P}$ for a type $\tau$ well-formed in $\Delta_P^{\mathcal{P}}$. It says that indistinguishable inputs lead to indistinguishable results.

**Definition 3.** *A term $e$ is TRNI($\mathcal{P}, \tau$) provided that $\Gamma_C^{\mathcal{P}} \vdash e$, and $\Delta_P^{\mathcal{P}} \vdash \tau$, and for all $\langle\gamma_1, \gamma_2\rangle \in \mathcal{I}_V[\![\mathcal{P}]\!]$ we have $\langle\gamma_1(e), \gamma_2(e)\rangle \in \mathcal{I}_E[\![\tau]\!]$.*

Notice that if a term is well-typed in the public view then by replacing all type variables in it with **int**, we get a term which is also well-typed in the confidential view (that is, if $\Delta_P^{\mathcal{P}}, \Gamma_P^{\mathcal{P}} \vdash e : \tau$, then $\Gamma_C^{\mathcal{P}} \vdash \delta(e) : \delta(\tau)$ where $\delta$ maps all type variables in $\Delta_P^{\mathcal{P}}$ to **int**). However, Definition 3 also requires that the term $e$ is itself well-typed in the confidential view. This ensures that the definition is applied, as intended, to programs that do not contain type variables.

The definition of TRNI is indexed by a type for the result of the term. The type can be interpreted as constraining the observations to be made by the public observer. We are mainly interested in concrete output types, which express that the observer can do whatever they like and has full knowledge of the result. Put differently, TRNI for an abstract type expresses security under the assumption that the observer is somehow forced to respect the abstraction. For example, we consider the policy $\mathcal{P}_{OE}$ (of Example 1) where $x$ can be declassified via $f = \lambda x : \textbf{int}.x \ mod \ 2$. As described in Example 3, $\Delta_P^{\mathcal{P}_{OE}} = \alpha_f$ and $\Gamma_P^{\mathcal{P}_{OE}} = x : \alpha_f,\ x_f : \alpha_f \to \textbf{int}$. We have that the program $x$ is TRNI($\mathcal{P}_{OE}, \alpha_f$) since the observer cannot do anything to $x$ except for applying $f$ to $x$ which is allowed by the policy. This program, however, is not TRNI($\mathcal{P}_{OE}, \textbf{int}$) since the observer can apply any function of the type $\textbf{int} \to \tau'$ (for some closed $\tau'$), including the identity function, to $x$ and hence can get the value of $x$.

*Example 5.* The program $x_f\ x$ is TRNI($\mathcal{P}_{OE}, \textbf{int}$). Indeed, for any arbitrary $\langle \gamma_1, \gamma_2 \rangle \in \mathcal{I}_V[\![\mathcal{P}]\!]$, we have that $\gamma_1(x_f) = \gamma_2(x_f) = f = \lambda x : \textbf{int}.x \ mod \ 2$, and $\langle v_1, v_2 \rangle \in \mathcal{I}_V[\![\alpha_f]\!]$, where $\gamma_1(x) = v_1$ and $\gamma_2(x) = v_2$ for some $v_1$ and $v_2$. When we apply $\gamma_1$ and $\gamma_2$ to the program, we get respectively $v_1 \ mod \ 2$ and $v_2 \ mod \ 2$. Since $\langle v_1, v_2 \rangle \in \mathcal{I}_V[\![\alpha_f]\!]$, as described in Example 4, $(v_1 \ mod \ 2) =_{\textbf{int}} (v_2 \ mod \ 2)$. Thus, $\langle \gamma_1(x_f\ x), \gamma_2(x_f\ x) \rangle \in \mathcal{I}_E[\![\textbf{int}]\!]$. Therefore, the program $x_f\ x$ satisfies the definition of TRNI.

## 4.2 Free theorem: typing in the public view implies security

In order to prove the free theorem, we define $\rho_{\mathcal{P}}$ as follows:

- for all $\alpha_x \in \Delta_P^{\mathcal{P}}$, $\rho_{\mathcal{P}}(\alpha_x) = \mathcal{I}_V[\![\alpha_x]\!]$,
- for all $\alpha_f \in \Delta_P^{\mathcal{P}}$, $\rho_{\mathcal{P}}(\alpha_f) = \mathcal{I}_V[\![\alpha_f]\!]$.

It is a relation on the type substitution $\delta_{\mathcal{P}}$ defined in Eqn. (1).

**Lemma 2.** *$\rho_{\mathcal{P}} \in Rel(\delta_{\mathcal{P}}, \delta_{\mathcal{P}})$.*

From Lemma 2, we can write $[\![\tau]\!]_{\rho_{\mathcal{P}}}$ or $[\![\tau]\!]^{\mathsf{ev}}_{\rho_{\mathcal{P}}}$ for any $\tau$ such that $\Delta_P^{\mathcal{P}} \vdash \tau$. We next establish the relation between $[\![\tau]\!]^{\mathsf{ev}}_{\rho}$ and $\mathcal{I}_E[\![\tau]\!]$: under the interpretation corresponding to the desired policy $\mathcal{P}$, they are equivalent. In other words, indistinguishability is an instantiation of the logical relation.

**Lemma 3.** *For any $\tau$ such that $\Delta_P^{\mathcal{P}} \vdash \tau$, we have $\langle v_1, v_2 \rangle \in [\![\tau]\!]_{\rho_{\mathcal{P}}}$ iff $\langle v_1, v_2 \rangle \in \mathcal{I}_V[\![\tau]\!]$, and also $\langle e_1, e_2 \rangle \in [\![\tau]\!]^{\mathsf{ev}}_{\rho_{\mathcal{P}}}$ iff $\langle e_1, e_2 \rangle \in \mathcal{I}_E[\![\tau]\!]$.*

By analyzing the type of $\Gamma_P^{\mathcal{P}}(x)$, we can establish the relation of $\gamma_1$ and $\gamma_2$ when $\langle \gamma_1, \gamma_2 \rangle \in \mathcal{I}_V[\![\mathcal{P}]\!]$.

**Lemma 4.** *If $\langle \gamma_1, \gamma_2 \rangle \in \mathcal{I}_V[\![\mathcal{P}]\!]$, then $\langle \gamma_1, \gamma_2 \rangle \in [\![\Gamma_P^{\mathcal{P}}]\!]_{\rho_{\mathcal{P}}}$.*

The main result of this section is that a term is TRNI at $\tau$ if it has type $\tau$ in the public view that encodes the policy.

**Theorem 2.** *If $e$ has no type variables and $\Delta_P^{\mathcal{P}}, \Gamma_P^{\mathcal{P}} \vdash e : \tau$, then $e$ is TRNI($\mathcal{P}, \tau$).*

*Proof.* From the abstraction theorem (Theorem 1), for all $\delta_1, \delta_2 \models \Delta_P^{\mathcal{P}}$, for all $\langle \gamma_1, \gamma_2 \rangle \in [\![\Gamma_P^{\mathcal{P}}]\!]_\rho$, and for all $\rho \in Rel(\delta_1, \delta_2)$, it follows that

$$\langle \delta_1\gamma_1(e), \delta_2\gamma_2(e)\rangle \in [\![\tau]\!]_\rho^{\mathsf{ev}}.$$

Consider $\langle \gamma_1, \gamma_2 \rangle \in \mathcal{I}_V[\![\mathcal{P}]\!]$. Since $\langle \gamma_1, \gamma_2 \rangle \in \mathcal{I}_V[\![\mathcal{P}]\!]$, from Lemma 4, we have that $\langle \gamma_1, \gamma_2 \rangle \in [\![\Gamma_P^{\mathcal{P}}]\!]_{\rho_{\mathcal{P}}}$. Thus, we have that $\langle \delta_{\mathcal{P}}\gamma_1(e), \delta_{\mathcal{P}}\gamma_2(e)\rangle \in [\![\tau]\!]_{\rho_{\mathcal{P}}}^{\mathsf{ev}}$. Since $e$ has no type variable, we have that $\delta_{\mathcal{P}}\gamma_i(e) = \gamma_i(e)$. Therefore, $\langle \gamma_1(e), \gamma_2(e)\rangle \in [\![\tau]\!]_{\rho_{\mathcal{P}}}^{\mathsf{ev}}$. Since $\langle \gamma_1(e), \gamma_2(e)\rangle \in [\![\tau]\!]_{\rho_{\mathcal{P}}}^{\mathsf{ev}}$, from Lemma 3, it follows that $\langle \gamma_1(e), \gamma_2(e)\rangle \in \mathcal{I}_E[\![\tau]\!]$. In addition, since $e$ has no type variable and $\Delta_P^{\mathcal{P}}, \Gamma_P^{\mathcal{P}} \vdash e : \tau$, we have that $\delta_{\mathcal{P}}(\Gamma_P^{\mathcal{P}}) \vdash e : \delta_{\mathcal{P}}(\tau)$ and hence, $\Gamma_C^{\mathcal{P}} \vdash e$. Therefore, $e$ is TRNI$(\mathcal{P}, \tau)$.

*Example 6 (Typing implies TRNI).* Consider the policy $\mathcal{P}_{OE}$. As described in Examples 2 and 3, the confidential view $\Gamma_C^{\mathcal{P}_{OE}}$ is $x : \mathbf{int}, x_f : \mathbf{int} \to \mathbf{int}$ and the public view $\Delta_P^{\mathcal{P}_{OE}}, \Gamma_P^{\mathcal{P}_{OE}}$ is $\alpha_f, x : \alpha_f, x_f : \alpha_f \to \mathbf{int}$. We look at the program $x_f\ x$. We can easily verify that $\Gamma_C^{\mathcal{P}_{OE}} \vdash x_f\ x : \mathbf{int}$ and $\Delta_P^{\mathcal{P}_{OE}}, \Gamma_P^{\mathcal{P}_{OE}} \vdash x_f\ x : \mathbf{int}$. Therefore, by Theorem 2, the program is TRNI$(\mathcal{P}_{OE}, \mathbf{int})$.

*Example 7.* In this example, we illustrate that if a program is well-typed in the confidential view and is not TRNI$(\mathcal{P}, \tau)$ for some $\tau$ well-formed in the public view of $\mathcal{P}$, then the type of the program in the public view is not $\tau$ or the program is not well-typed in the public view.

We consider the policy $\mathcal{P}_{OE}$. As described in Example 6, its public view is $\alpha_f, x : \alpha_f, x_f : \alpha_f \to \mathbf{int}$. We first look at the program $x$. This program is not TRNI$(\mathcal{P}_{OE}, \mathbf{int})$ since $x$ itself is confidential and cannot be directly declassified. In the public view of the policy, the type of this program is $\alpha_f$ which is not $\mathbf{int}$.

We now look at the program $x\ mod\ 3$. This program is not TRNI$(\mathcal{P}_{OE}, \alpha_f)$ since it takes indistinguishable inputs at $\alpha_f$ (e.g. 2 and 4) and produces results that are not indistinguishable at $\alpha_f$ (e.g. $2 = 2\ mod\ 3$, $1 = 4\ mod\ 3$, and $\langle 2, 1\rangle \notin \mathcal{I}_V[\![\alpha_f]\!]$). We can easily verify that this program is not well-typed in the public view since the type of $x$ in the public view is $\alpha_f$, while *mod* expects arguments of the $\mathbf{int}$ type.

*Remark 1 (Extension).* Our encoding can be extended to support richer policies (details in appendix). To support policies where an input $x$ can be declassified via two declassifiers $f : \mathbf{int} \to \tau_f$ and $g : \mathbf{int} \to \tau_g$ for some $\tau_f$ and $\tau_g$, we use type variable $\alpha_{f,g}$ as the type for $x$ and use $\alpha_{f,g} \to \tau_f$ and $\alpha_{f,g} \to \tau_g$ as types for $x_f$ and $x_g$. To support policies where multiple inputs can be declassified via a declassifier, e.g. inputs $x$ and $y$ can be declassified via $f = \lambda z : \mathbf{int} \times \mathbf{int}.(\pi_1 z + \pi_2 z)/2$, we introduce a new term variable $z$ which is corresponding to a tuple of two inputs $x$ and $y$ and we require that only $z$ can be declassified. The type of $z$ is $\alpha_f$ and two tuples $\langle v_1, v_2\rangle$ and $\langle v_1', v_2'\rangle$ are indistinguishable at $\alpha_f$ when $f\ \langle v_1, v_2\rangle = f\ \langle v_1', v_2'\rangle$.

## 5 Related Work

*Typing secure information flow* Pottier and Simonet [31] implement FlowCaml [32], the first type system for information flow analysis dealing with a real-sized programming language (a large fragment of OCaml), and they prove soundness. In comparison with our results, we do not consider any imperative features; they do not consider any form of declassification, their type system significantly departs from ML typing, and their security proof is not based on an abstraction theorem. An interesting question is whether their type system can be translated to system F or some other calculus with an abstraction theorem. FlowCaml provides type inference for security types. Our work relies on the Standard ML type system to enforce security. Standard ML provides type inference, which endows our approach with an inference mechanism.

Barthe et al. [10] propose a modular method to reuse type systems and proofs for noninterference for declassification. They also provide a method to conclude declassification soundness by using an existing noninterference theorem. In contrast to our work, their type system significantly departs from standard typing rules, and does not make use of parametricity.

Tse and Zdancewic [44] propose a security-typed language for robust declassification: declassification cannot be triggered unless there is a digital certificate to assert the proper authority. Their language inherits many features from System $F_{<:}$ and uses monadic labels as in DCC [1]. In contrast to our work, security labels are based on the Decentralized Label Model (DLM) [27], and are not semantically unified with the standard safety types of the language.

The Dependency Core Calculus (DCC) [1] expresses security policies using monadic types indexed on levels in a security lattice with the usual interpretation that flows are only allowed between levels in accordance with the ordering. DCC does not include declassification and the noninterference theorem of [1] is proved from scratch (not leveraging parametricity). While DCC is a theoretical calculus, its monadic types fit nicely with the monads and monad transformers used by the Haskell language for computational effects like state and I/O. Algehed and Russo [5] encode the typing judgment of DCC in Haskell using closed type families, one of the type system extensions supported by GHC that brings it close to dependent types. However, they do not prove security.

Compared with type systems, relational logics can specify IF policy and prove more programs secure through semantic reasoning [28,9,21,11], but at the cost of more user guidance and less familiar notations. Aguirre et al [2] use relational higher order logic to prove soundness of DCC essentially by formalizing the semantics of DCC [1].

*Connections between secure IF and type abstraction* Tse and Zdancewic [43] translate the recursion-free fragment of DCC to System F. The main theorem for this translation aims to show that parametricity of System F implies noninterference. Shikuma and Igarashi identify a mistake in the proof [38]; they also give a noninterference-preserving translation for a version of DCC to the simply-typed lambda calculus. Although they make direct use of a specific logical relation, their results are not obtained by instantiating a parametricity theorem. Bowman and Ahmed [13] finally provide a translation from the recursion-free fragment of DCC to System $F_{\omega}$, proving that parametricity implies noninterference, via a correctness theorem for the translation (which is akin to a full abstraction property). Bowman and Ahmed's translation makes essential use of the power of System $F_\omega$ to encode judgments of DCC. Algehed and Bernardy [4] translate a label-polymorphic variant DCC (without recursion) into the calculus of constructions (CC) and prove noninterference directly from a parametricity result for CC [12]. The authors note that it is not obvious this can be extended to languages with nontermination or other effects. Their results have been checked in Agda and the presentation achieves elegance owing to the fact that parametricity and noninterference can be explicitly defined in dependent type theory; indeed, CC terms can represent proof of parametricity [12].

Our goals do not necessitate a system like DCC for policy, raising the question of whether a simpler target type system can suffice for security policies expressed differently from DCC. We answer the question in the affirmative, and believe our results for polymorphic lambda (and for ML) provide transparent explication of noninterference by reduction to parametricity.

The preceding works on DCC are "translating noninterference to parametricity" in the sense of translating both programs and types. The implication is that one might leverage an existing type checker by translating both a program and its security policy into another program such that it's typability implies the original conforms to policy. Our work aims to cater more directly for practical application, by minimizing the need to translate the program and hence avoiding the need to prove the correctness of a translation.

Cruz et al. [16] show that type abstraction implies relaxed noninterference. Similar to ours, their definition of relaxed noninterference is a standard extensional semantics, using partial equivalence relations. This is in contrast with Li and Zdancewic [24] where the semantics is entangled with typability.

Protzenko et al. [33] propose to use abstract types as the types for secrets and use standard type systems for security. This is very close in spirit to our work. Their soundness theorem is about a property called "secret independence", very close to noninterference. In contrast to our work, there is no declassification and no use of the abstraction theorem.

Rajani and Garg [34] connect fine- and coarse-grained type systems for information flow in a lambda calculus with general references, defining noninterference (without declassification) as a step-indexed Kripke logical relation that expresses indistinguishability. Further afield, a connection between security and parametricity is made by Devriese et al [17], featuring a negative result: System F cannot be compiled to the the Sumii-Pierce calculus of dynamic sealing [41] (an idealized model of a cryptographic mechanism). Finally, information flow analyses have also been put at the service of parametricity [48].

*Abstraction theorems for other languages* Parametricity remains an active area of study [40]. Vytiniotis and Weirich [46] prove the abstraction theorem for $R_\omega$, which extends $F_\omega$ with constructs that are useful for programming with type equivalence propositions. Rossberg et al [36] show another path to parametricity for ML modules, by translating them to $F_\omega$. Crary's result [15] covers a large fragment of ML but without references and mutable state. Abstraction theorems have been given for mutable state, based on ownership types [6] and on more semantically based reasoning [7,3,19,8,42].

## 6 Discussion and Conclusion

In this work, we show how to express declassification policies by using standard types of the simply typed lambda calculus. By means of parametricity, we prove that type checking implies relaxed noninterference, showing a direct connection between declassification and parametricity.

Our approach should be applicable to other languages that have an abstraction theorem (e.g [7,8,3,19,42]) with the potential benefit of strong security assurance from off-the-shelf type checkers. In particular, we demonstrate in appendix the results can be extended to a large fragment of ML including general recursion.

Although in this paper we demonstrate our results using confidentiality and declassification, our approach applies as well to integrity and endorsement, as they have been shown to be information flow properties analog to confidentiality [23,20].

The simple encodings in the preceding sections do not support computation and output at multiple levels. For example, consider a policy where $x$ is a confidential input that can be declassified via $f$ and we also want to do the computation $x + 1$ of which the result is at confidential level. Clearly, $x + 1$ is ill-typed in the public interface. We provide in an appendix more involved encodings supporting computation at multiple levels. To have an encoding that support multiple levels, we add universally quantified types $\forall\alpha.\tau$ to the language presented in §2. However, this goes against our goal of minimizing complexity of translation. Observe that many applications are composed of programs which, individually, do not output at multiple levels; for example, the password checker, and data mining computations using sensitive inputs to calculate aggregate or statistical information. For these the simpler encoding suffices.

Vanhoef et al. [45] and others have proposed more expressive declassification policies than the ones in Li and Zdancewic [24]: policies that keep state and can be written as programs. We speculate that TRNI for stateful declassification policies can be obtained for free in a language with state—indeed, our work provides motivation for development of abstraction theorems for such languages.

## References

1. Abadi, M., Banerjee, A., Heintze, N., Riecke, J.G.: A core calculus of dependency. In: ACM POPL. pp. 147–160 (1999)
2. Aguirre, A., Barthe, G., Gaboardi, M., Garg, D., Strub, P.: A relational logic for higher-order programs. PACMPL **1**(ICFP), 21:1–21:29 (2017)
3. Ahmed, A., Dreyer, D., Rossberg, A.: State-dependent representation independence. In: ACM POPL. pp. 340–353 (2009)
4. Algehed, M., Bernardy, J.: Simple noninterference from parametricity. PACMPL **3**(ICFP), 89:1–89:22 (2019)

5. Algehed, M., Russo, A.: Encoding DCC in Haskell. In: Workshop on Programming Languages and Analysis for Security. pp. 77–89 (2017)
6. Banerjee, A., Naumann, D.A.: Ownership confinement ensures representation independence for object-oriented programs. Journal of the ACM **52**(6), 894–960 (2005)
7. Banerjee, A., Naumann, D.A.: State based ownership, reentrance, and encapsulation. In: ECOOP. vol. 3586, pp. 387–411 (2005)
8. Banerjee, A., Naumann, D.A.: State based encapsulation for modular reasoning about behavior-preserving refactorings. In: Aliasing in Object-oriented Programming. Springer State-of-the-art Surveys (2012)
9. Banerjee, A., Naumann, D.A., Nikouei, M.: Relational logic with framing and hypotheses. In: FSTTCS. LIPIcs, vol. 65, pp. 11:1–11:16 (2016)
10. Barthe, G., Cavadini, S., Rezk, T.: Tractable enforcement of declassification policies. In: IEEE Computer Security Foundations Symposium. pp. 83–97 (2008)
11. Beckert, B., Ulbrich, M.: Trends in relational program verification. In: Müller, P., Schaefer, I. (eds.) Principled Software Development - Essays Dedicated to Arnd Poetzsch-Heffter on the Occasion of his 60th Birthday. pp. 41–58. Springer (2018)
12. Bernardy, J.P., Jansso, P., Paterson, R.: Proofs for free: Parametricity for dependent types. Journal of Functional Programmming **22**(2), 107–152 (2012)
13. Bowman, W.J., Ahmed, A.: Noninterference for free. In: International Conference on Functional Programming. pp. 101–113 (2015)
14. Crary, K.: Logical relations and a case study in equivalence checking. In: Pierce, B.C. (ed.) Advanced Topics in Types and Programming Languages, chap. 6, pp. 245–289. The MIT Press (2005)
15. Crary, K.: Modules, abstraction, and parametric polymorphism. In: ACM POPL. pp. 100–113 (2017)
16. Cruz, R., Rezk, T., Serpette, B.P., Tanter, É.: Type abstraction for relaxed noninterference. In: ECOOP. pp. 7:1–7:27 (2017)
17. Devriese, D., Patrignani, M., Piessens, F.: Parametricity versus the universal type. PACMPL **2**(POPL), 38:1–38:23 (2018)
18. Dreyer, D.: Understanding and Evolving the ML Module System. Ph.D. thesis, Carnegie Mellon University (2005)
19. Dreyer, D., Neis, G., Rossberg, A., Birkedal, L.: A relational modal logic for higher-order stateful ADTs. In: ACM POPL. pp. 185–198 (2010)
20. Fournet, C., Rezk, T.: Cryptographically sound implementations for typed information-flow security. In: ACM POPL (2008)
21. Grimm, N., Maillard, K., Fournet, C., Hritcu, C., Maffei, M., Protzenko, J., Ramananandro, T., Rastogi, A., Swamy, N., Béguelin, S.Z.: A monadic framework for relational verification: applied to information security, program equivalence, and optimizations. In: Certified Programs and Proofs. pp. 130–145 (2018)
22. Harper, R.: Practical foundations for programming languages. Cambridge University Press (2016)
23. Li, P., Mao, Y., Zdancewic, S.: Information integrity policies. In: In Proceedings of the Workshop on Formal Aspects in Security and Trust (FAST) (2003)
24. Li, P., Zdancewic, S.: Downgrading policies and relaxed noninterference. In: ACM POPL. pp. 158–170 (2005)
25. Mitchell, J.C.: Foundations for Programming Languages. MIT Press (1996)
26. Myers, A.C.: Jif homepage. http://www.cs.cornell.edu/jif/ (accessed July 2018)
27. Myers, A.C., Liskov, B.: Protecting privacy using the decentralized label model. ACM Trans. on Software Engineering and Methodology **9**, 410–442 (Oct 2000)

28. Nanevski, A., Banerjee, A., Garg, D.: Dependent type theory for verification of information flow and access control policies. ACM Trans. Program. Lang. Syst. **35**(2), 6 (2013). https://doi.org/10.1145/2491522.2491523
29. Ngo, M., Naumann, D.A., Rezk, T.: Typed-based relaxed noninterference for free. CoRR **abs/1905.00922** (2019), `http://arxiv.org/abs/1905.00922`
30. Pitts, A.M.: Typed operational reasoning. In: Pierce, B.C. (ed.) Advanced Topics in Types and Programming Languages, chap. 7, pp. 245–289. The MIT Press (2005)
31. Pottier, F., Simonet, V.: Information flow inference for ML. In: ACM POPL. pp. 319–330 (2002)
32. Pottier, F., Simonet, V.: Flowcaml homepage. https://www.normalesup.org/ simonet/soft/flowcaml/index.html (accessed July 2018)
33. Protzenko, J., Zinzindohoué, J.K., Rastogi, A., Ramananandro, T., Wang, P., Béguelin, S.Z., Delignat-Lavaud, A., Hritcu, C., Bhargavan, K., Fournet, C., Swamy, N.: Verified low-level programming embedded in F. PACMPL **1**(ICFP), 17:1–17:29 (2017)
34. Rajani, V., Garg, D.: Types for information flow control: Labeling granularity and semantic models. In: IEEE Computer Security Foundations Symposium (2018)
35. Reynolds, J.C.: Types, abstraction and parametric polymorphism. In: IFIP Congress. pp. 513–523 (1983)
36. Rossberg, A., Russo, C.V., Dreyer, D.: F-ing modules. J. Funct. Program. **24**(5), 529–607 (2014)
37. Sabelfeld, A., Sands, D.: Declassification: Dimensions and principles. Journal of Computer Security **17**(5), 517–548 (2009)
38. Shikuma, N., Igarashi, A.: Proving noninterference by a fully complete translation to the simply typed lambda-calculus. Logical Methods in Computer Science **4**(3) (2008)
39. Standard ML of New Jersey homepage. https://www.smlnj.org/
40. Sojakova, K., Johann, P.: A general framework for relational parametricity. In: IEEE Symp. on Logic in Computer Science. pp. 869–878 (2018)
41. Sumii, E., Pierce, B.C.: A bisimulation for dynamic sealing. In: ACM POPL. pp. 161–172 (2004)
42. Timany, A., Stefanesco, L., Krogh-Jespersen, M., Birkedal, L.: A logical relation for monadic encapsulation of state: Proving contextual equivalences in the presence of runST. Proc. ACM Program. Lang. **2**(POPL), 64:1–64:28 (Dec 2017)
43. Tse, S., Zdancewic, S.: Translating dependency into parametricity. In: International Conference on Functional Programming. pp. 115–125 (2004)
44. Tse, S., Zdancewic, S.: A design for a security-typed language with certificate-based declassification. In: ESOP. pp. 279–294 (2005)
45. Vanhoef, M., Groef, W.D., Devriese, D., Piessens, F., Rezk, T.: Stateful declassification policies for event-driven programs. In: IEEE Computer Security Foundations Symposium. pp. 293–307 (2014)
46. Vytiniotis, D., Weirich, S.: Parametricity, type equality, and higher-order polymorphism. J. Funct. Program. **20**(2), 175–210 (2010)
47. Wadler, P.: Theorems for free! In: International Conference on Functional Programming. pp. 347–359 (1989)
48. Washburn, G., Weirich, S.: Generalizing parametricity using information-flow. In: IEEE Symp. on Logic in Computer Science. pp. 62–71 (2005)

## A Extensions

The extensions in this section are corresponding to the encoding in §4.

## A.1 Declassification policies

Variations of our encoding can support richer declassification policies and accept more secure programs. We consider two ways to extend our encoding.

*More declassification functions* The notation in [24] labels an input with a set of declassification functions, so in general an input can be declassified in more than one way. To show how this can be accomodated, we present an extension for a policy $\mathcal{P}$ where $\mathbf{V}_{\mathcal{P}} = \{x\}$, and $x$ can be declassified via $f$ or $g$ for some $f$ and $g$, where $\vdash f : \mathbf{int} \to \tau_f$ and $\vdash g : \mathbf{int} \to \tau_g$. The confidential view and the public view for this policy are as below:

$$\begin{aligned}
&\Gamma_C^{\mathcal{P}} = x{:}\mathbf{int}, x_f{:}\mathbf{int} \to \tau_f, x_g{:}\mathbf{int} \to \tau_g\\
&\Delta_P^{\mathcal{P}} = \alpha_{f,g}\\
&\Gamma_P^{\mathcal{P}} = x{:}\alpha_{f,g}, x_f{:}\alpha_{f,g} \to \tau_f,\ x_g{:}\alpha_{f,g} \to \tau_g
\end{aligned}$$

We now have a new definition of indistinguishability. The definition is similar to the one presented in §4, except that we add a new rule for $\alpha_{f,g}$.

$$\textsc{Eq-Var4}\ \frac{\vdash v_1, v_2 : \mathbf{int} \qquad \langle f\ v_1, f\ v_2\rangle \in \mathcal{I}_E[\![\tau_f]\!] \qquad \langle g\ v_1, g\ v_2\rangle \in \mathcal{I}_E[\![\tau_g]\!]}{\langle v_1, v_2\rangle \in \mathcal{I}_V[\![\alpha_{f,g}]\!]}$$

With the new encoding and the new definition of indistinguishability, we can define TRNI($\mathcal{P}, \tau$) as in Definition 3. From the abstraction theorem, we again obtain that for any program $e$, if $\Gamma_C^{\mathcal{P}} \vdash e$, and $\Delta_P^{\mathcal{P}}, \Gamma_P^{\mathcal{P}} \vdash e : \tau$, then $e$ is TRNI($\mathcal{P}, \tau$).

For example, we consider programs $e_1 = x_f\ x$ and $e_2 = x_g\ x$. These two programs are well-typed in both views of $\mathcal{P}$, and in the public view, their types are respectively $\tau_f$ and $\tau_g$. Thus, $e_1$ is TRNI($\mathcal{P}, \tau_f$), and $e_2$ is TRNI($\mathcal{P}, \tau_g$).

*Using an equivalent function to declassify* In most type systems for declassification, the declassifier function or expression must be identical to the one in the policy. Indeed, policy is typically expressed by writing a "declassify" annotation on the expression [37]. However, the type system presented in [24, § 5] is more permissive: it accepts a declassification if it is semantically equivalent to the policy function, according to a given syntactically defined approximation of equivalence. Verification tools can go even further in reasoning with semantic equivalence [28,21], but any automated checker is limited due to undecidability of semantic equivalence.

We consider a policy $\mathcal{P}$ where there are two confidential inputs $x$ and $y$, $x$ can be declassified via $f$, and $y$ can be declassified via $g$, $f : \mathbf{int} \to \tau$, and $g : \mathbf{int} \to \tau$ for some $\tau$. Suppose that there exists a function $a$ s.t. $f \circ a = g$ semantically. With the encoding in §4, we accept $g\ y$, or rather $x_g\ y$, but we cannot accept $f(a\ y)$ even though it is semantically the same.

To accept programs like $f(a\ y)$, based on the idea of the first extension, we encode the policy as below, where $y$ is viewed as a confidential input that can be declassified via $g$ or $f \circ a$. Note that in the following encoding, we have two type variables: $\alpha_{g,f\circ a}$ for the confidential input, and $\alpha_f$ for the result of $x_a\ x$ which can be declassified via $f$.

$$\begin{aligned}
&\Gamma_C^{\mathcal{P}} = x : \mathbf{int},\ x_f : \mathbf{int} \to \tau, y : \mathbf{int}, y_g : \mathbf{int} \to \tau, y_a : \mathbf{int} \to \mathbf{int} \\
&\Delta_P^{\mathcal{P}} = \alpha_f,\ \alpha_{g,f\circ a} \\
&\Gamma_P^{\mathcal{P}} = x : \alpha_f,\ x_f : \alpha_f \to \tau,\ y : \alpha_{g,f\circ a},\ y_g : \alpha_{g,f\circ a} \to \tau,\ x_a : \alpha_{g,f\circ a} \to \alpha_f
\end{aligned}$$

Indistinguishability for this policy is defined similarly to the one in Section 4, except that we have the following rule for $\alpha_{g,f\circ a}$.

$$\textsc{Eq-Var6}\ \frac{\vdash v_1 : \mathbf{int} \qquad \vdash v_2 : \mathbf{int} \qquad \langle g\ v_1, g\ v_2\rangle \in \mathcal{I}_E[\![\tau]\!]}{\langle v_1, v_2\rangle \in \mathcal{I}_V[\![\alpha_{g,f\circ a}]\!]}$$

As in the first extension, we can define TRNI for a type $\tau$ well-formed in $\Delta_P^{\mathcal{P}}$ and we have the free theorem stating that if $\Gamma_C^{\mathcal{P}} \vdash e$, and $\Delta_P^{\mathcal{P}}, \Gamma_P^{\mathcal{P}} \vdash e : \tau$, then $e$ is TRNI$(\mathcal{P}, \tau)$.

W.r.t. the new encoding, both $y_g\ y$ and $x_f(x_a\ y)$ are well-typed in the public view. In other words, we accepts both $y_g\ y$ and $x_f(x_a\ y)$.

Notice that as discussed in [24], the problem of establishing relations between declassification functions in general is undecidable. Thus, the relations should be provided or can be found in a predefined amount of time. Otherwise, the relations are not used in the encoding and programs like $x_f(y_a\ y)$ will not typecheck.

### A.2 Global policies

The policies considered in §4 and §A.1 are corresponding to local policies in [24]. We now consider policies where a declassifier can involve more than one confidential input. To be consistent with [24], we call such policies global policies. For simplicity, in this subsection, we consider a policy $\mathcal{P}$ where there are two confidential inputs, $x_1$ and $x_2$, which can be declassified via $f$ of the type $\mathbf{int}_1 \times \mathbf{int}_2 \to \tau_f$ [6]. Notice that here we use subscripts for the input type of $f$ to mean that the confidential input $x_i$ is corresponding to $i$-th element of an input of $f$.

*Example 8 (Average can be declassified).* We consider the policy $\mathcal{P}_{\text{Ave}}$ where there are two confidential inputs $x_1$ and $x_2$ and their average can be declassified. That is $x_1$ and $x_2$ can be declassified via $f = \lambda x : \mathbf{int} \times \mathbf{int}.(\pi_1 x + \pi_2 x)/2$.

In our encoding, we need to maintain the correspondence between inputs and arguments of the declassifier since we want to prevent laundering attacks [37]. A laundering attack occurs, for example, when the declassifier $f$ is applied to $\langle x_1, x_1\rangle$, since then the value of $x_1$ is leaked.

[6] We can extend the encoding presented in this section to have policies where different subsets of $\mathbf{V}_{\mathcal{P}}$ can be declassified and to have more than one declassifier associated with a set of confidential inputs.

In the general case, to encode the requirement that a specific $n$-tuple of confidential inputs can be declassified via $f$, we introduce a new variable $y$. The basic idea is that $y$ is corresponding to that $n$-tuple of confidential inputs, $x_i$ cannot be declassified, and only $y$ can be declassified via $f$. Therefore, the confidential and public views are as below, where for readability we show the case $n = 2$.

$$\begin{aligned}
\Gamma_C^{\mathcal{P}} &\triangleq \{x_1 : \mathbf{int}, x_2 : \mathbf{int}, y : \mathbf{int} \times \mathbf{int}, y_f : \mathbf{int} \times \mathbf{int} \to \tau_f\} \\
\Delta_P^{\mathcal{P}} &\triangleq \{\alpha_{x_1}, \alpha_{x_2}, \alpha_f\} \\
\Gamma_P^{\mathcal{P}} &\triangleq \{x_1 : \alpha_{x_1}, x_2 : \alpha_{x_2}, y : \alpha_f, y_f : \alpha_f \to \tau_f\}
\end{aligned}$$

For each $i \in \{1, \ldots, n\}$, since $x_i$ cannot be declassified, the indisinguishability for $\alpha_{x_i}$ is the same as the one for $\alpha_x$ described in Fig. 3. Since $y$ corresponds to the tuple of confidential inputs and only it can be declassified via $f$, indistinguishability for the type of $y$ in the public view $\alpha_f$ is as below (again, case $n = 2$).

$$\textsc{Eq-Var5}\ \frac{\vdash v, v' : \mathbf{int} \times \mathbf{int} \qquad \langle f\ v, f\ v' \rangle \in \mathcal{I}_E[\![\tau_f]\!]}{\langle v, v' \rangle \in \mathcal{I}_V[\![\alpha_f]\!]}$$

We next encode the correspondence between inputs and argument of the declassifier. We say that a term substitution $\gamma$ is *consistent* w.r.t. $\Gamma_P^{\mathcal{P}}$ if $\gamma \models \delta_{\mathcal{P}}(\Gamma_P^{\mathcal{P}})$ and in addition, for all $i \in \{1, 2\}$, $\pi_i(\gamma(y)) = \gamma(x_i)$. As we can see, the additional condition takes care of the correspondence of inputs and the arguments of the intended declassifier.

We next define the type substitution and indistinguishable term substitutions for $\mathcal{P}$. We say that $\delta_{\mathcal{P}} \models \Delta_P^{\mathcal{P}}$ when $\delta_{\mathcal{P}}(\alpha_f) = \mathbf{int} \times \mathbf{int}$ and for all $\alpha_{x_i}$, $\delta_{\mathcal{P}}(\alpha_{x_i}) = \mathbf{int}$. We say that two term substitutions $\gamma_1$ and $\gamma_2$ are indistinguishable w.r.t. $\mathcal{P}$ (denoted by $\langle \gamma_1, \gamma_2 \rangle \in \mathcal{I}_V[\![\mathcal{P}]\!]$) if $\gamma_1$ and $\gamma_2$ are consistent w.r.t. $\Gamma_P^{\mathcal{P}}$, $\gamma_1(y_f) = \gamma_2(y_f) = f$, for all other $x \in dom(\Gamma_P^{\mathcal{P}})$, $\langle \gamma_1(x), \gamma_2(x) \rangle \in \mathcal{I}_V[\![\Gamma_P^{\mathcal{P}}(x)]\!]$.

Then we can define TRNI$(\mathcal{P}, \tau)$ as in Def. 3 (except that we use the new definition of indistinguishable term substitutions). We also have the free theorem stating that if $e$ has no type variable and $\Delta_P^{\mathcal{P}}, \Gamma_P^{\mathcal{P}} \vdash e : \tau$, then $e$ is TRNI$(\mathcal{P}, \tau)$. The proof goes through without changes.

*Example 9 (Average can be declassified - cont.).* Here we present the encoding for the policy $\mathcal{P}_{\text{Ave}}$ described in Example 8. The confidential and public views for this policy is as below:

$$\begin{aligned}
\Gamma_C^{\mathcal{P}_{\text{Ave}}} &\triangleq \{x_1 : \mathbf{int}, x_2 : \mathbf{int}, y : \mathbf{int} \times \mathbf{int}, y_f : \mathbf{int} \times \mathbf{int} \to \mathbf{int}\} \\
\Delta_P^{\mathcal{P}_{\text{Ave}}} &\triangleq \{\alpha_{x_1}, \alpha_{x_2}, \alpha_f\} \\
\Gamma_P^{\mathcal{P}_{\text{Ave}}} &\triangleq \{x_1 : \alpha_{x_1}, x_2 : \alpha_{x_2}, y : \alpha_f, y_f : \alpha_f \to \mathbf{int}\}
\end{aligned}$$

We can easily check that the program $y_f\ y$ is TRNI$(\mathcal{P}_{\text{Ave}}, \mathbf{int})$; it is well-typed in both views, and in the public view its type is $\mathbf{int}$.

$$
\begin{array}{rll}
k ::= & \mathsf{1} \mid \mathsf{T} \mid \mathsf{S}(c) \mid \Pi\alpha : k.k \mid \Sigma\alpha : k.k & \text{Kind} \\
c, \tau ::= & \alpha \mid \star \mid \lambda\alpha : k.c \mid c\ c \mid \langle c, c\rangle & \text{Type constr.} \\
 & \mid \pi_1 c \mid \pi_2 c \mid \mathbf{unit} \mid \mathbf{int} \mid \tau_1 \to \tau_2 & \\
 & \mid \tau_1 \times \tau_2 \mid \forall\alpha : k.\tau \mid \exists\alpha : k.\tau & \\
\sigma ::= & \mathsf{1} \mid (\!| k |\!) \mid \langle\!| \tau |\!\rangle \mid \Pi^{\mathrm{gn}}\alpha : \sigma.\sigma & \text{Signature} \\
 & \mid \Pi^{\mathrm{ap}}\alpha : \sigma.\sigma \mid \Sigma\alpha : \sigma.\sigma & \\
e ::= & x \mid \star \mid n \mid \lambda x : \tau.e \mid e\ e \mid \langle e, e\rangle & \text{Term} \\
 & \mid \pi_1 e \mid \pi_2 e \mid \Lambda\alpha : k.e \mid e[c] & \\
 & \mid \mathsf{pack}[c, e]\ \mathsf{as}\ \exists\alpha : k.\tau & \\
 & \mid \mathsf{unpack}[\alpha, x] = e\ \mathsf{in}\ e \mid \mathsf{fix}_\tau e & \\
 & \mid \mathsf{let}\ x = e\ \mathsf{in}\ e & \\
 & \mid \mathsf{let}\ \alpha/m = M\ \mathsf{in}\ e \mid \mathsf{Ext}\ M & \\
M ::= & m \mid \star \mid (\!| c |\!) \mid \langle\!| e |\!\rangle \mid \lambda^{\mathrm{gn}}\alpha/m : \sigma.M & \text{Module} \\
 & \mid M\ M \mid \lambda^{\mathrm{ap}}\alpha/m : \sigma.M & \\
 & \mid M \cdot M \mid \langle M, M\rangle \mid \pi_1 M \mid \pi_2 M & \\
 & \mid \mathsf{unpack}[\alpha, x] = e\ \mathsf{in}\ (M : \sigma) & \\
 & \mid \mathsf{let}\ x = e\ \mathsf{in}\ M & \\
 & \mid \mathsf{let}\ \alpha/m = M\ \mathsf{in}\ (M : \sigma) \mid M :> \sigma & \\
\Gamma ::= & . \mid \Gamma, \alpha : k \mid \Gamma, x : \tau \mid \Gamma, \alpha/m : \sigma & \text{Context}
\end{array}
$$

**Fig. 4.** Module calculus

## B TRNI for Module Calculus

This section recapitulates the development of §4 but using an encoding suited to the module calculus of Crary and Dreyer [18,15].[7] It is a core calculus that models Standard ML including higher order generative and applicative functors, sharing constraints (via singleton kinds), and sealing. Sealing ascribes a signature to a module expression and thereby enforces data abstraction.

The syntax is in Fig. 4. The calculus has static expressions: kinds ($k$), constructors ($c$) and signatures ($\sigma$), and dynamic expressions: terms ($e$) and modules ($M$). The full formal system is given in our technical report [29]; here we sketch highlights.

The unit kind $\mathsf{1}$ has only the unit constructor $\star$. The base kind, $\mathsf{T}$, is for types that can be used to classify terms. By convention, we use the metavariable $\tau$ for constructors that are types (i.e. of the kind $\mathsf{T}$). The singleton kind $\mathsf{S}(c)$ classifies constructors that are definitionally equivalent to $c$. In addition, we have higher kinds: dependent functions $\Pi\alpha : k_1.k_2$ and dependent pairs $\Sigma\alpha : k_1.k_2$.

The syntax for terms is standard and includes general recursion ($\mathsf{fix}_\tau e$). Module expressions include unit module ($\star$), pairing/projection, atomic modules with

[7] Our only change is to add **int** and arithmetic primitives, for examples.

a single static or dynamic component ($(\!|c|\!)$, $\langle\!|e|\!\rangle$), generative and applicative functors ($\lambda^{\mathrm{gn}}\alpha/m : \sigma.M$, $\lambda^{\mathrm{ap}}\alpha/m : \sigma.M$, the applications of which are written resp. $M_1\ M_2$, $M_1 \cdot M_2$), and unpacking ($\mathsf{unpack}[\alpha, x] = e\ \mathsf{in}\ (M : \sigma)$). While term binding is as usual ($\mathsf{let}\ x = e\ \mathsf{in}\ M$), the module binding construct is unusual: $\mathsf{let}\ \alpha/m = M_1\ \mathsf{in}\ (M_2 : \sigma)$ binds a pair of names, where constructor variable $\alpha$ is used to refer to the static part of $M_1$ (and $m$ to the full module). This is used to handle the phase distinction between compile-time and run-time expressions.

A signature describes an interface for a module. Signatures include unit signature, atomic kind and atomic type signature, generative and applicative functors, and dependent pairs ($\Sigma\alpha : \sigma_1.\sigma_2$). A signature $\sigma$ is *transparent* when it exposes the implementation of the static part of modules of $\sigma$. A signature $\sigma$ is *opaque* when it hides some information about the static part of modules of $\sigma$. The sealing construct, $M :> \sigma$, ascribes a signature to the module in the sense of enforcing $\sigma$ as an abstraction boundary.

*Abstraction theorem* The static semantics includes judgments $\vdash \Gamma\ \mathsf{ok}$, $\Gamma \vdash e : \tau$, $\Gamma \vdash_{\mathsf{P}} M : \sigma$, and $\Gamma \vdash_{\mathsf{I}} M : \sigma$ for resp. well-formed context, well-typed term, pure well-formed module, and impure well-formed module. The pure and impure judgment forms roughly correspond to unsealed and sealed modules; the formal system treats sealing as an effect, introduced by application of a generative functor as well as by the sealing construct.

The dynamic semantics is call-by value, with these values:

$$\begin{aligned} v &:= x \mid \star \mid n \mid \lambda x : \tau.e \mid \langle v, v\rangle \mid \Lambda\alpha : k.e && \text{Term values} \\ &\quad \mid \mathsf{pack}[c, v]\ \mathsf{as}\ \exists\alpha : k.\tau \\ V &:= m \mid \star \mid (\!|c|\!) \mid \langle\!|v|\!\rangle \mid \langle V, V\rangle && \text{Module values} \\ &\quad \mid \lambda^{\mathrm{gn}}\alpha/m : \sigma.M \mid \lambda^{\mathrm{ap}}\alpha/m : \sigma.M \end{aligned}$$

The logical relation for the calculus is more complicated than the one in §2. Even so, the statement of the abstraction theorem for terms is similar to the one in §2.

**Theorem 3 (Abstraction theorem [15]).** *Suppose that* $\vdash \Gamma \mathsf{ok}$*. If* $\Gamma \vdash e : \tau$*, then* $\Gamma \vdash e \sim e : \tau$*.*

Modules and terms are interdependent, and Crary's theorem includes corresponding results for pure and for impure modules. We express security in terms of sealed modules, but our security proof only relies on the abstraction theorem for expressions.

*Free theorem: TRNI for the module calculus* We present the idea of the encoding for the module calculus. (Formalization of the encoding can be found in appendix.) To make the presentation easier to follow, in this section, we write examples in Standard ML (SML). These examples are checked with SML of New Jersey, version 110.96 [39].

For a policy $\mathcal{P}$, we construct the public view and the confidential view by using signatures containing type information of confidential inputs and their associated declassifiers. In particular, the signature for the confidential view is a transparent signature which exposes the concrete type of confidential input, while the signature for the public view is an opaque one which hides the type information of confidential inputs. For example, for the policy $\mathcal{P}_{OE}$ (see Example 1), we have the following signatures, where `transOE` and `opaqOE` are respectively the transparent signature for the confidential view and the opaque signature for the public view.

```
signature transOE = signature opaqOE =
 sig                 sig
  type t = int        type t
  val x:t             val x:t
  val f:t->int        val f:t->int
 end                 end
```

Different from §4, a program has only a module input which is of the transparent signature and contains all confidential inputs and their declassifiers. A program can use the input via the module variable `m`. For example, for $\mathcal{P}_{OE}$, we have the program `m.f m.x`, which is corresponding to the program $x_f\ x$ in Example 5.

Using the result in §4, we define indistinguishability as an instantation of the logical relation, and we say that a term $e$ is TRNI$(\mathcal{P}, \tau)$ if on indistinguishable substitutions w.r.t. $\mathcal{P}$, it generates indistinguishable outputs at $\tau$. By using the abstraction theorem 3 for terms, we obtain our main result.

**Theorem 4.** *If the type of $e$ in the public view is $\tau$, then $e$ is TRNI$(\mathcal{P}, \tau)$.*

For the module calculus, when $e$ is well-typed in the public view, $e$ is also well-typed in the confidential view. Therefore, different from Theorem 2 which requires that $e$ has no type variable, Theorem 4 simply requires that $e$ is well-typed in the public view. Our example program `m.f m.x` typechecks at **int**, so by Theorem 4 it is TRNI$(\mathcal{P}_{OE}, \textbf{int})$.

*Usage of our approach* We can use our approach with ordinary ML implementations. In the case that the source programs are already parameterized by one module for their confidential inputs and their declassifiers, then there is no need to modify source programs at all.

For example, we consider `program` described below. Here `M` is a module of the transparent signature `transOE`. By sealing this module with the opaque signature `opaqOE`, we get the module `opaqM`. Intuitively, `program` is TRNI$(\mathcal{P}_{OE}, \textbf{int})$ since the declassifier `f` is applied to the confidential input `x`. We also come to the same conclusion from the fact that the type of this program is **int**.

```
structure M = struct
   type t = int
   val x : t = 1
   val f : t -> int = fn x => x mod 2
end
structure opaqM :> opaqOE = M
val program : int = opaqM.f opaqM.x
```

So far our discussion is about open terms but the ML type checker only applies to closed terms. In the case that the client program is open (i.e. that it can receive any module of the transparent signature as an input, as in the program `m.f m.x` presented above), in order to be able to type check it for a policy, we need to close it by putting in a closing context, which we call wrapper. For any program $e$ and policy $\mathcal{P}$, the wrapper is written using a functor as shown below, where `opaqP` is the opaque signature for the public view of $\mathcal{P}$. Type $\tau$ is the type at which we want to check security of $e$. (The identifiers `program` and `wrapper` are arbitrary.)

```
functor wrapper (structure m: opaqP) =
  struct
      val program : τ = e
  end
```

Note that $e$ is unchanged.

We have proved that if the wrapper $wrap_{\mathcal{P}}(e)$ is of the signature from `opaqP` to $\tau$, then the type of $e$ in the public view is $\tau$. Therefore, from Theorem 4, $e$ is TRNI at $\tau$. For instance, for the policy $\mathcal{P}_{OE}$, we have that $wrap_{\mathcal{P}}($`m.f m.x`$)$ is of the signature from `transOE` to **int** and hence, we infer that the type of `m.f m.x` in the public view is **int** and hence, `m.f m.x` is TRNI$(\mathcal{P}_{OE}, \textbf{int})$.

*Extension* As in the case of the simple calculus, our encoding for ML can also be extended for policies where multiple inputs are declassified via a declassifier. Here, for illustration purpose, we present the encoding for a policy which is inspired by two-factor authentication.

*Example 10.* The policy $\mathcal{P}_{\text{Aut}}$ involves two confidential passwords and two declassifiers `checking1` and `checking2` as below, where `input1` and `input2` are respectively the first input and the second input from a user. Notice that `checking2` takes a tuple of two passwords as its input.

```
fun checking1(password1:int) =
  if (password1 = input1) then 1 else 0
fun checking2(passwords:int*int) =
  if ((#1 passwords) = input1) then
    if ((#2 passwords) = input2) then 1 else 0
  else 2
```

We next construct the confidential view and the public view for the policy. To encode the requirement that two passwords can be declassified via `checking2`, we introduce a new variable `passwords` which is corresponding to the tuple of the two passwords, and only `passwords` can be declassified via `checking2`. The transparent signature for the confidential view of $\mathcal{P}_{\text{Aut}}$ is below.

```
signature transAut = sig
  type t1 = int
  val password1:t1
  val checking1:t1->int
  type t2 = int
  val password2:t2
  type t3 = int * int
  val passwords:t3
  val checking2:t3 ->int
end
```

The signature `opaqAut` for the public view is the same except the types `t1`, `t2`, and `t3` are opaque.

We have that the programs `m.checking2 m.passwords` and `m.checking1 m.password1`, where `m` is a module variable of the transparent signature `transAut`, have the type **int** in the public view. Hence both programs are TRNI($\mathcal{P}_{\mathrm{Aut}}$, **int**).

## C Computation at multiple security levels

The encodings in the preceding sections do not support computation and output at multiple levels. For example, consider a policy where $x$ is a confidential input that can be declassified via $f$ and we also want to do the computation $x + 1$ of which the result is at confidential level. Clearly, $x + 1$ is ill-typed in the public interface. To support computation at multiple levels we develop a monadic encoding inspired by DCC, and a public interface that represents policy for multiple levels.

To have an encoding that support multiple levels, we add universally quantified types $\forall\alpha.\tau$ to the language presented in §2 (already present in ML). In addition, to simplify the encoding, we add the unit type **unit**. W.r.t. these new types, we have new values: the unit value $\langle\rangle$ of **unit**, and values $\Lambda\alpha.e$ of $\forall\alpha.\tau$.

To facilitate the presentation of the idea of the encoding, we consider a lattice $\mathcal{L}$ with three different levels $L$, $M$, $H$ such that $L \sqsubset M \sqsubset H$. We also use a simple policy $\mathcal{P}$ with three inputs *hi*, *mi* and *li* at resp. $H$, $M$ and $L$, and *hi* can be declassified via $f : \mathbf{int} \to \mathbf{int}$ to $M$. (The encoding with an arbitrary finite lattice and policy is in the appendix.) For simplicity, we suppose that values on input and output channels are of **int** type.

To model multiple outputs we consider programs that return a tuple of values, one component for each output channel. To model channel access being associated with different security levels, the output values are wrapped, in the form $\lambda x : \mathbf{unit}.n$. To read such a value, an observer needs to provide an appropriate key. By giving $x$ an abstract type corresponding to a security level, we can control access.

Similar to the previous sections, we assume that free variables in programs are their inputs, but now the values will be wrapped integers. Intuitively, a wrapped value $v$ can be unwrapped by *unwrap* $k$ $v$, where $k$ is an appropriate key, and *unwrap* $k$ $v$ can be implemented as the application of $v$ on $k$ (i.e. $v$ $k$). Concretely, $k$ will be the unit value $\langle\rangle$.

We further assume that programs are executed in a context where there are several output channels, each corresponding to a security level. A program will compute a tuple of wrapped values, where each element of the output tuple can be unwrapped by using an appropriate key and the unwrapped value is sent to the channel. In short, we assume the program of interest is executed in a context that wraps its inputs, and also unwraps each components of the output tuple and sends the value on the corresponding channel. This assumption is illustrated in the following pseudo program, where $e$ is the program of interest, $o$ is the computed tuple, $\texttt{Output.Channel}_l$ is an output channel at $l$, $k_l$ is a key to unwrap value at $l$, and $\pi_l$ projects the output value for the channel $l$.

```
let o = e in
    Output.Channel$_L$ := $unwrap$ $k_L$ ($\pi_L$ $o$)
    Output.Channel$_M$ := $unwrap$ $k_M$ ($\pi_M$ $o$)
    Output.Channel$_H$ := $unwrap$ $k_H$ ($\pi_H$ $o$)
```

The keys are not made directly available to $e$, which must manipulate its inputs via an interface described below.

*Encoding* Different from the previous sections, we use type variables $\alpha_H, \alpha_M$ and $\alpha_L$ as the types of keys for unwrapping wrapped values at $H$, $M$ and $L$. This idea is similar to the idea in [24]. Different from [24], we do not translate DCC and we support declassification.

For an input at $l$ that cannot be declassified ($mi$ or $li$), its type in the public view is $\alpha_l \to \textbf{int}$. For the input $hi$ which can be declassified via $f$, we use another type variable (i.e. $\alpha_H^f$) as the type of key to unwrapped values.[8] Similar to the previous sections, we use $\alpha^f$ to encode number values at $H$. Therefore, the type of $hi$ in the public view is $\alpha_H^f \to \alpha^f$. As we use **unit** as the type for key, in the confidential view the type of $hi$, $mi$, and $li$ is $\textbf{unit} \to \textbf{int}$.

As assumed above, a program computes an output which is a tuple of three wrapped values. Since we use type variables as keys to unwrap wrapped values, the type of outputs of programs we consider is $(\alpha_H \to \textbf{int}) \times (\alpha_M \to \textbf{int}) \times (\alpha_L \to \textbf{int})$.

To support computing outputs at a level $l$, by using the idea of monad, we have interfaces $\mathsf{cp}_l$ and $\mathsf{wr}_l$ which are the bind and unit expressions for a monad. In addition, to support converting a wrapped value at $l$ to $l'$ (where $l \sqsubset l'$), we have interfaces $\mathsf{cvu}_l^{l'}$. To use $hi$ in a computation at $H$, we have $\mathsf{cv}_f$. Similar to the previous sections, we have $hi_f$ for the declassfier $f$. The types of there interfaces are described in Fig. 5.

*Example 11.* We illustrate the idea of the encoding by writing a program that computes the triple $hi + li + 1$, $f$ $hi$ and $li + 1$ at resp. $H$, $M$, and $L$.

[8] If we use $\alpha_H$ instead, since this input can be declassified to $M$, the indistinguishability will be incorrect: all wrapped values at $H$ are wrongly indistinguishable to observer $M$.

$$
\begin{aligned}
\Delta_{\mathcal{P}} &= \{\alpha_L, \alpha_M, \alpha_H, \alpha^f_H, \alpha^f\} \\
\Gamma_{\mathcal{P}} &= \{hi : \alpha^f_H \to \alpha^f, mi : \alpha_M \to \mathbf{int}, li : \alpha_L \to \mathbf{int}\} \cup \\
&\quad \{\mathsf{cp}_l : \forall \beta_1, \beta_2.(\alpha_l \to \beta_1) \to \big(\beta_1 \to (\alpha_l \to \beta_2)\big) \\
&\qquad\qquad \to \alpha_l \to \beta_2 \mid l \in \mathcal{L}\} \cup \\
&\quad \{\mathsf{cvu}^{l'}_l : \forall \beta.(\alpha_l \to \beta) \to (\alpha_{l'} \to \beta) \mid l, l' \in \mathcal{L} \wedge l \sqsubset l'\} \cup \\
&\quad \{\mathsf{wr}_l : \forall \beta.\beta \to \alpha_l \to \beta \mid l \in \mathcal{L}\} \cup \\
&\quad \{hi_f : (\alpha^f_H \to \alpha^f) \to (\alpha_M \to \mathbf{int})\} \cup \\
&\quad \{\mathsf{cv}_f : (\alpha^f_H \to \alpha^f) \to (\alpha_H \to \mathbf{int})\}
\end{aligned}
$$

**Fig. 5.** Contexts for $\mathcal{P}$

First, we will have $e_1$ that does the computation at $L$: $li + 1$. Let $plus\_one = \lambda x : \mathbf{int}.x + 1$. In order to use $\mathsf{cp}_L$, we first wrap $plus\_one$ by using $\mathsf{wr}_L$.

$$wrap\_plus\_one = \lambda x : \mathbf{int}.\mathsf{wr}_L(plus\_one\ x)$$

Then $e_1$ is as below:

$$e_1 = \mathsf{cp}_L[\mathbf{int}][\mathbf{int}]\ li\ wrap\_plus\_one$$

Next, we have $e_2$ that does the computation $hi + li$ at $H$. Let $add$ be a function of the type $\mathbf{int} \to \mathbf{int} \to \mathbf{int}$. From $add$, we construct $wrap\_addc$ of the type $\mathbf{int} \to \alpha_H \to \mathbf{int}$, where $c$ is a variable of the type $\mathbf{int}$.

$$wrap\_addc = \lambda y : \mathbf{int}.\mathsf{wr}_H(add\ \ c\ \ y)$$

Then we have $e_2$ as below. Note that in order to use $li$ in $\mathsf{cp}_H$, we need to convert $li$ from level $L$ to level $H$ by using $\mathsf{cvu}^H_L$.

$$
\begin{aligned}
&e_2 = \\
&\mathsf{cp}_H[\mathbf{int}][\mathbf{int}]\ hi\ (\lambda c : \mathbf{int}.(\mathsf{cp}_H[\mathbf{int}][\mathbf{int}]\ (\mathsf{cvu}^H_L\ li)\ wrap\_addc))
\end{aligned}
$$

At this point, we can write the program $e$ that computes $hi + li + 1$, $f\ \ hi$, and $li + 1$ at resp. $H$, $M$, and $L$.

$$e = \big(\lambda li : \alpha_L \to \mathbf{int}.\langle e_2,\ \langle hi_f\ \ hi,\ li\rangle\rangle\big)\ e_1$$

The implementations of the defined interfaces are straightforward. For example, on a protected input of type $\mathbf{unit} \to \beta_1$ and a continuation of type $\beta_1 \to (\mathbf{unit} \to \beta_2)$, the implementation $\mathsf{comp}$ of $\mathsf{cp}_l$ first unfolds the protected input by applying it to the key $\langle\rangle$ and then applies the continuation on the result.

$$\mathsf{comp} = \Lambda \beta_1, \beta_2.\lambda x : \mathbf{unit} \to \beta_1.\lambda f : \beta_1 \to (\mathbf{unit} \to \beta_2).f(x\ \langle\rangle)$$

The implementation of $\mathsf{wr}_l$ is $\Lambda\beta.\lambda x : \beta.\lambda\_ : \mathbf{unit}.\,x$. The conversions $\mathsf{cvu}^{l'}_l$ are implemented by the identity function.

*Indistinguishability* Different from §4, indistinguishability is defined for observer $\zeta$ ($\zeta \in \mathcal{L}$). [9] The indistinguishability relations for $\zeta$ at type $\tau$ on values (denoted as $\mathcal{I}^{\zeta}_{\mathcal{P}}[\![\tau]\!]$) is defined as an instance of the logical relation with a careful choice of interpretations for $\alpha_l$ and $\alpha^f_H$. The idea is that if the observer $\zeta$ cannot observe data at $l$ (i.e. $l \not\sqsubseteq \zeta$), $\zeta$ does not have any key to unwrap these values and hence, all wrapped values at $l$ are indistinguishable to $\zeta$. Thus, $\alpha_l$ is interpreted as the empty relation for $\zeta$. Otherwise, since $\zeta$ has key and can unwrapped values, values wrapped at $l$ are indistinguishable to $\zeta$ if they are equal and hence, $\alpha_l$ is interpreted as $\{\langle\langle\rangle,\langle\rangle\rangle\}$ (note that the concrete type for key is **unit**). Since wrapped numbers from *hi* are indistinguishable to observer $\zeta$ when they cannot be distinguish by the declassifier $f$, the interpretation of $\alpha^f_H$ for the observer $M$ is $\{\langle\langle\rangle,\langle\rangle\rangle\}$.[10] By using the idea in the previous sections, based on $\mathcal{I}^{\zeta}_{\mathcal{P}}[\![\tau]\!]$, we define indistinguishability relations for $\zeta$ at type $\tau$ on terms (denoted as $\mathcal{I}^{\zeta}_{\mathcal{P}}[\![\tau]\!]^{ev}$)).

*Free theorem* We write $\rho$ as an environment that maps type variables to its interpretations of form $\langle\tau_1,\tau_2,R\rangle$ and maps term variables to tuples of values. (This is similar to the formalization for §B.) We write $\rho_L$ and $\rho_R$ for the mappings that map every variable in the domain of $\rho$ to respectively the first element and the second element of the tuple that $\rho$ maps that variable to. The application of $\rho_L$ (resp. $\rho_R$) to $e$ is denoted by $\rho_L(e)$ (resp. $\rho_R(e)$) (this notations is similar to $\delta\gamma(e)$ in §2). We write $\rho \models^{\text{full}}_{\zeta} \mathcal{P}$ to mean that $\rho$ maps inputs to tuples of indistinguishable values.

By leveraging the abstraction theorem, we get the free theorem saying that a well-typed program $e$ maps indistinguishable inputs to indistinguishable outputs.

**Theorem 5.** *If $\Delta_{\mathcal{P}}, \Gamma_{\mathcal{P}} \vdash e : \tau$, then for any $\zeta \in \mathcal{L}$ and $\rho \models^{full}_{\zeta} \mathcal{P}$,*

$$\langle \rho_L(e), \rho_R(e)\rangle \in \mathcal{I}^{\zeta}_{\mathcal{P}}[\![\tau]\!]^{ev}.$$

We state it this way to avoid spelling out the definition of TRNI for this encoding. The encoding presented here can also be extended to support richer policies described in Remark 1.

*Remark 2.* Our encoding supports declassification while DCC does not. However, if we consider programs without declassification then DCC is more expressive since in our encoding, to use a wrapped value in a computation at $l$, this value must be wrapped at $l'$ such that $l' \sqsubseteq l$. However, in DCC, this is not the case due to the definition of the "protected at" judgment in DCC: if type $\tau$ is already protected at $l$ then so is $T_{l'}\tau$ for any $l'$. Therefore, data protected at $l$ can be used in a computation protected at $l'$ even when $l' \not\sqsubseteq l$. For example, we consider the encoding for a policy defined in a lattice with four levels $\top$, $M_1$, $M_2$, $\bot$ where $\top \sqsubset M_i \sqsubset \top$ but $M_1$ and $M_2$ are incomparable. We can have the

[9] Following [13], we use $\zeta$ for observers.

[10] Therefore, if we used $\alpha_H \to \alpha_f$ for *hi*, all wrapped values from *hi* would be indistinguishable to the observer $M$ since this observer do not have any key to open data at $H$ that cannot be declassified (to observer $M$, the interpretation of $\alpha_H$ is empty.

following well-typed program in DCC (the program is written in the notations in [13]).

$$bind\ y = (\eta_{M_1} 1)\ in\ \eta_{M_2}(\eta_{M_1}(y+1))$$

In our encoding, this program can be rewritten as below, where $f : \mathbf{int} \to \alpha_{M_2} \to \alpha_{M_1} \to \mathbf{int}$ is from function $\lambda y : \mathbf{int}.y + 1$ (see a similar function in Example 11).

$$\mathsf{cp}_{M_2}[\mathbf{int}][\mathbf{int}]\ (\mathsf{wr}_{M_1} 1)\ f$$

This program is not well-typed in our encoding. This feature of DCC, allowing multiple layers of wrapping, is needed to encode state-passing programs (in particular, to encode the Volpano-Smith system for while programs) where low data is maintained unchanged through high computations. The feature seems unnecessary for functional programs.