Type-Based Declassification for Free

Minh Ngo[1,2], David A. Naumann[1], and Tamara Rezk[2(✉)]

[1] Stevens Institute of Technology, Hoboken, USA
[2] Inria, Sophia Antipolis, France
tamara.rezk@inria.fr

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 [33] 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 [36]. Relational parametricity provides an effective basis for formal reasoning about program transformations ("theorems for free" [49]), representation independence and information hiding for program verification [6,25]. The connection between IF and relational parametricity has been made precise in 2015, for DCC, by translation to the calculus F_ω and use of the existing parametricity theorem

S.-W. Lin et al. (Eds.): ICFEM 2020, LNCS 12531, pp. 181–197, 2020.
https://doi.org/10.1007/978-3-030-63406-3_11

for F_ω [12]. 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 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 [12] encode the typechecking judgment by nontrivial translation of both types and terms into F_ω. 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 [14].

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 [12] nor on modifications of existing type systems [15]. 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 also provide an extended version [29] that shows this development for the ML module calculus using Crary's theorem [14], 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, for two reasons: (1) the chosen language is similar to the language used in the paper of Reynolds [36] where the abstraction theorem was first proven, and (2) we want to illustrate our encoding approach (Sect. 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 [36, Sect. 2]; we prove the abstraction theorem using contemporary notation.[1]

Language. The syntax of the language is as below, where α 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.

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

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

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

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 $\twoheadrightarrow^*$ for the reflexive, transitive closure of $\twoheadrightarrow$.

A *typing context* Δ is a set of type variables. A *term context* Γ is a mapping from term variables to types, written like $x : \mathbf{int}, y : \mathbf{int} \to \mathbf{int}$. We write $\Delta \vdash \tau$ to mean that τ is *well-formed w.r.t.* Δ, that is, all type variables in τ are in Δ. We say that e is *typable w.r.t.* Δ *and* Γ (denoted by $\Delta, \Gamma \vdash e$) when there exists a well-formed type τ 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 Γ that is well-formed under Δ, in the sense that $\Delta \vdash \Gamma(x)$ for all $x \in dom(\Gamma)$. When the term context and the type context are empty, we write $\vdash e : \tau$.

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

[1] Some readers may find it helpful to consult the following references for background on logical relations and parametricity: [22, Chapt. 49], [25, Chapt. 8], [13,31].

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

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

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

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

Fig. 1. Typing rules

Let γ be a *term substitution*, i.e., a finite map from term variables to closed values, and δ 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 γ *respects* Γ (denoted by $\gamma \models \Gamma$) when $dom(\gamma) = dom(\Gamma)$ and $\vdash \gamma(x) : \Gamma(x)$ for any x. We say δ *respects* Δ (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 τ_1 and τ_2. Let ρ 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 ρ is compatible with δ_1, δ_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 δ_1 and δ_2. For any τ, $[\![\tau]\!]_\rho$ 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 δ_1 and δ_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 Γ by applying δ on the range of Γ, 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 γ and then δ 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 $x \in dom(\Gamma)$.

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

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

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

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

Fig. 2. The logical relation

Definition 1 (Logical equivalence). *Terms e and e' are* logically equivalent *at τ in Δ and Γ (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 $\rho \in \mathit{Rel}(\delta_1, \delta_2)$, 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 [36]). *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 [39].

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,[2] where n is an integer value, and $\oplus$ represents primitive arithmetic operators.

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

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. A policy simply defines which are the confidential variables and their authorized declassifications. 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 τ_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 $\mathbf{F}_{\mathcal{P}_{OE}}(x) = f = \lambda x : \mathbf{int}.\, x \ mod\ 2$. 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 Sect. 2, we define and we prove our free theorem. We consider the termination-insensitive [30] information flow security property,[3] with declassification, called type-based relaxed noninterference (TRNI) and taken from Cruz et al. [15]. It is important to notice that our development, 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 [15] 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 Definition 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

[2] In this paper, the type of confidential inputs is **int**.

[3] Our security property is termination sensitive but programs in the language always terminate. In the extended version [29], in the development for ML, programs may not terminate and the security property is also termination sensitive.

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 δ, $\delta(e)$ is syntactically the same as e.[4]

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 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 $\mathbf{V}_\top = \{x \mid x \in \mathbf{V}_\mathcal{P} \setminus dom(\mathbf{F}_\mathcal{P})\}$ 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 $\mathcal{P}$.

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

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 $x \in dom(\Gamma^\mathcal{P}_C)$, 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.

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

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

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

The inductive definition of indistinguishability for a policy $\mathcal{P}$ is presented in Fig. 3, where α_x and α_f are from $\Delta_P^{\mathcal{P}}$. Indistinguishability is defined for τ 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 α_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 α_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 $\langle f\ v_1, f\ v_2 \rangle \in \mathcal{I}_E[\![\tau_f]\!]$.

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

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

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

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

Fig. 3. Indistinguishability

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

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

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

Term substitutions γ_1 and γ_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 $x_f \in dom(\Gamma_P^{\mathcal{P}})$, $\gamma_1(x_f) = \gamma_2(x_f) = f$,
- for all other $x \in dom(\Gamma_P^{\mathcal{P}})$, $\langle\gamma_1(x), \gamma_2(x)\rangle \in \mathcal{I}_V[\![\Gamma_P^{\mathcal{P}}(x)]\!]$.

Note that each γ_i maps x_f to the specific function 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 τ 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 δ 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 merely 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. Consider the policy $\mathcal{P}_{OE}$ (of Example 1) where x can be declassified via $f = \lambda x : \mathbf{int}.x \; mod \; 2$. As described in Example 3, $\Delta_P^{\mathcal{P}_{OE}} = \alpha_f$ and $\Gamma_P^{\mathcal{P}_{OE}} = x : \alpha_f,\ x_f : \alpha_f \to \mathbf{int}$. 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$(\mathcal{P}_{OE}, \mathbf{int})$ since the observer can apply any function of the type $\mathbf{int} \to \tau'$ (for some closed τ'), including the identity function, to x and hence can get the value of x.

Example 5. The program $x_f\ x$ is TRNI$(\mathcal{P}_{OE}, \mathbf{int})$. Indeed, for any arbitrary $\langle \gamma_1, \gamma_2 \rangle \in \mathcal{I}_V[\![\mathcal{P}]\!]$, we have that $\gamma_1(x_f) = \gamma_2(x_f) = f = \lambda x : \mathbf{int}.x \; mod \; 2$, 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 γ_1 and γ_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, $(v_1 \; mod \; 2) =_{\mathbf{int}} (v_2 \; mod \; 2)$. Thus, $\langle \gamma_1(x_f\ x), \gamma_2(x_f\ x) \rangle \in \mathcal{I}_E[\![\mathbf{int}]\!]$. 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 security "for free", i.e., as consequence of Theorem 1, 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 Eq. (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 τ 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 τ 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 γ_1 and γ_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 τ if it has type τ 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. If a program is well-typed in the confidential view but not TRNI$(\mathcal{P}, \tau)$ for some τ well-formed in the public view of $\mathcal{P}$, then the type of the program in the public view is not τ or the program is not well-typed in the public view. In policy $\mathcal{P}_{OE}$, from Example 6, the public view is $\alpha_f, x : \alpha_f, x_f : \alpha_f \to \mathbf{int}$. We first look at the program x that 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 α_f which is not **int**. We now look at the program $x\ mod\ 3$ that is not TRNI$(\mathcal{P}_{OE}, \alpha_f)$ since it takes indistinguishable inputs at α_f (e.g. 2 and 4) and produces results that are not indistinguishable at α_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 α_f, while *mod* expects arguments of the **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 τ_f and τ_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 α_f and two tuples $\langle v_1, v_2 \rangle$ and $\langle v_1', v_2' \rangle$ are indistinguishable at α_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 [32] implement FlowCaml [33], 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. [9] propose a modular method to reuse type systems and proofs for noninterference [40] for declassification. They also provide a method to conclude declassification soundness by using an existing noninterference theorem [37]. In contrast to our work, their type system significantly departs from standard typing rules, and does not make use of parametricity. Tse and Zdancewic [46] 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 [8,10,21,28], 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 [45] 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 [41]; 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 [12] finally provide a translation from the recursion-free fragment of DCC to System F_ω, 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_ω 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 [11]. 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 [11]. 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. [15] 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. [34] 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 [35] 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 [16], featuring a negative result: System F cannot be compiled to the the Sumii-Pierce calculus of dynamic sealing [43] (an idealized model of a cryptographic mechanism). Finally, information flow analyses have also been put at the service of parametricity [50].

Abstraction Theorems for Other Languages. Parametricity remains an active area of study [42]. Vytiniotis and Weirich [48] prove the abstraction theorem for R_ω, which extends F_ω with constructs that are useful for programming with type equivalence propositions. Rossberg et al [38] show another path to parametricity for ML modules, by translating them to F_ω. Crary's result [14] 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 [3,7,17,44].

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 [3,7,17,44]) with the potential benefit of strong security assurance from off-the-shelf type checkers. In particular, we demonstrate (in an extended version [29]) that 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 [18–20,23].

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 the extended version) 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 Sect. 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. [47] 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.

Acknowledgements. We thank anonymous reviewers for their suggestions. This work was partially supported by CISC ANR-17-CE25-0014-01, IPL SPAI, the European Union's Horizon 2020 research and innovation programme under grant agreement No 830892, US NSF CNS 1718713, and ONR N00014-17-1-2787.

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. J. ACM **52**(6), 894–960 (2005)
7. Banerjee, A., Naumann, D.A.: State based encapsulation for modular reasoning about behavior-preserving refactorings. In: Clarke, D., Noble, J., Wrigstad, T. (eds.) Aliasing in Object-Oriented Programming. Types, Analysis and Verification. LNCS, vol. 7850, pp. 319–365. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-36946-9_12
8. Banerjee, A., Naumann, D.A., Nikouei, M.: Relational logic with framing and hypotheses. In: FSTTCS. LIPIcs, vol. 65, pp. 11:1–11:16 (2016)
9. Barthe, G., Cavadini, S., Rezk, T.: Tractable enforcement of declassification policies. In: IEEE Computer Security Foundations Symposium, pp. 83–97 (2008)
10. 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, Heidelberg (2018). https://doi.org/10.1007/978-3-319-98047-8_3
11. Bernardy, J.P., Jansso, P., Paterson, R.: Proofs for free: parametricity for dependent types. J. Func. Program. **22**(2), 107–152 (2012)
12. Bowman, W.J., Ahmed, A.: Noninterference for free. In: ICFP, pp. 101–113 (2015)
13. 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)
14. Crary, K.: Modules, abstraction, and parametric polymorphism. In: ACM POPL, pp. 100–113 (2017)
15. Cruz, R., Rezk, T., Serpette, B.P., Tanter, É.: Type abstraction for relaxed noninterference. In: ECOOP, pp. 7:1–7:27 (2017)
16. Devriese, D., Patrignani, M., Piessens, F.: Parametricity versus the universal type. PACMPL **2**(POPL), 38:1–38:23 (2018)
17. Dreyer, D., Neis, G., Rossberg, A., Birkedal, L.: A relational modal logic for higher-order stateful ADTs. In: ACM POPL, pp. 185–198 (2010)
18. Fournet, C., Guernic, G.L., Rezk, T.: A security-preserving compiler for distributed programs: from information-flow policies to cryptographic mechanisms. In: ACM Conference on Computer and Communications Security, CCS (2009)
19. Fournet, C., Planul, J., Rezk, T.: Information-flow types for homomorphic encryptions. In: ACM CCS, pp. 351–360 (2011)
20. Fournet, C., Rezk, T.: Cryptographically sound implementations for typed information-flow security. In: ACM POPL (2008)
21. Grimm, N., et al.: 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: 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. Softw. Eng. Methodol. **9**, 410–442 (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)
29. Ngo, M., Naumann, D.A., Rezk, T.: Type-based declassification for free. CoRR abs/1905.00922 (2020). http://arxiv.org/abs/1905.00922
30. Ngo, M., Piessens, F., Rezk, T.: Impossibility of precise and sound termination-sensitive security enforcements. In: 2018 IEEE Symposium on Security and Privacy, SP. IEEE Computer Society (2018)
31. 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)
32. Pottier, F., Simonet, V.: Information flow inference for ML. In: ACM POPL, pp. 319–330 (2002)
33. Pottier, F., Simonet, V.: Flowcaml homepage. https://www.normalesup.org/simonet/soft/flowcaml/index.html. Accessed July 2018
34. Protzenko, J., et al.: Verified low-level programming embedded in F. PACMPL **1**(ICFP), 17:1–17:29 (2017)
35. Rajani, V., Garg, D.: Types for information flow control: labeling granularity and semantic models. In: IEEE Computer Security Foundations Symposium (2018)
36. Reynolds, J.C.: Types, abstraction and parametric polymorphism. In: IFIP Congress, pp. 513–523 (1983)
37. Rezk, T.: Verification of confidentiality policies for mobile code. Ph.D. thesis, University of Nice-Sophia Antipolis (2006)
38. Rossberg, A., Russo, C.V., Dreyer, D.: F-ing modules. J. Funct. Program. **24**(5), 529–607 (2014)
39. Sabelfeld, A., Sands, D.: Declassification: dimensions and principles. J. Comput. Secur. **17**(5), 517–548 (2009)
40. Fragoso Santos, J., Jensen, T., Rezk, T., Schmitt, A.: Hybrid typing of secure information flow in a JavaScript-like language. In: Ganty, P., Loreti, M. (eds.) TGC 2015. LNCS, vol. 9533, pp. 63–78. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-28766-9_5
41. Shikuma, N., Igarashi, A.: Proving noninterference by a fully complete translation to the simply typed lambda-calculus. Logical Methods Comp. Sci. **4**(3) (2008)
42. Sojakova, K., Johann, P.: A general framework for relational parametricity. In: IEEE Symposium on Logic in Computer Science, pp. 869–878 (2018)
43. Sumii, E., Pierce, B.C.: A bisimulation for dynamic sealing. In: ACM POPL, pp. 161–172 (2004)
44. 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 (2017)
45. Tse, S., Zdancewic, S.: Translating dependency into parametricity. In: International Conference on Functional Programming, pp. 115–125 (2004)
46. Tse, S., Zdancewic, S.: A design for a security-typed language with certificate-based declassification. In: Sagiv, M. (ed.) ESOP 2005. LNCS, vol. 3444, pp. 279–294. Springer, Heidelberg (2005). https://doi.org/10.1007/978-3-540-31987-0_20
47. 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)

48. Vytiniotis, D., Weirich, S.: Parametricity, type equality, and higher-order polymorphism. J. Funct. Program. **20**(2), 175–210 (2010)
49. Wadler, P.: Theorems for free! In: International Conference on Functional Programming, pp. 347–359 (1989)
50. Washburn, G., Weirich, S.: Generalizing parametricity using information-flow. In: IEEE Symposium on Logic in Computer Science, pp. 62–71 (2005)