Type Abstraction for Relaxed Noninterference*

Raimil Cruz[1], Tamara Rezk[2], Bernard Serpette[2], and Éric Tanter[1]

1 **PLEIAD Lab, Computer Science Department (DCC), University of Chile**
{racruz,etanter}@dcc.uchile.cl
2 **INRIA - Indes Project-Team, Sophia Antipolis, France**
first.last@inria.fr

Abstract

Information-flow security typing statically prevents confidential information to leak to public channels. The fundamental information flow property, known as *noninterference*, states that a public observer cannot learn anything from private data. As attractive as it is from a theoretical viewpoint, noninterference is impractical: real systems need to intentionally declassify some information, selectively. Among the different information flow approaches to declassification, a particularly expressive approach was proposed by Li and Zdancewic, enforcing a notion of *relaxed noninterference* by allowing programmers to specify *declassification policies* that capture the intended manner in which public information can be computed from private data. This paper shows how we can exploit the familiar notion of type abstraction to support expressive declassification policies in a simpler, yet more expressive manner. In particular, the type-based approach to declassification—which we develop in an object-oriented setting—addresses several issues and challenges with respect to prior work, including a simple notion of label ordering based on subtyping, support for recursive declassification policies, and a local, modular reasoning principle for relaxed noninterference. This work paves the way for integrating declassification policies in practical security-typed languages.

1998 ACM Subject Classification D.4.6 Security and Protection: Information flow controls, D.3.2 Language Classifications: Object-oriented languages

Keywords and phrases type abstraction, relaxed noninterference, information flow control

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2017.53

1 Introduction

Information-flow security typing enables statically classifying program entities with respect to their confidentiality levels, expressed via a lattice of security labels [17]. For instance, a two-level lattice $\mathsf{L} \preccurlyeq \mathsf{H}$ allows distinguishing public or low data (*e.g.* Int_L) from confidential or high data (*e.g.* Int_H). An information-flow security type system statically ensures *noninterference*, *i.e.* that high-confidentiality data may not flow directly or indirectly to lower-confidentiality channels [35]. To do so, the security type system tracks the confidentiality level of computation based on the confidentiality of the data involved.

As attractive as it is, noninterference is too strict to be useful in practice, as it prevents confidential data to have *any* influence whatsoever on observable, public output. Indeed, even a simple password checker function violates noninterference. Consider the following:

* This work was partially funded by Project Conicyt REDES 140219 "CEV: Challenges in Practical Electronic Voting". Raimil Cruz is funded by CONICYT-PCHA/Doctorado Nacional/2014-63140148.

```
String login(String guess, String password)
  if(password == guess)
    return "Login Successful"
  else
     return "Login failed"
}
```

By definition, a *public observer* that tries to log in *should* be able to "learn something" about the confidential input (here, password), thereby violating the confidentiality restriction imposed by noninterference.

This problem with noninterference has long been recognized. Supporting such intentional downward information flows is called *declassification*, which can be supported in many different ways [28]. For example, Jif [23] supports an explicit `declassify` operator to allow downward flows to be accepted by the security type system. In the above example, one can use `declassify(password == guess)` to state that the returned value is public knowledge. However, arbitrary uses of a `declassify` operator may lead to serious information flow leaks; for instance `declassify(password)` simply makes the password publicly available. One solution adopted by Jif is to control declassification using principals with privileges, as in the Decentralized Label Model (DLM) [24]. Trusted declassification [20] restricts Jif's mechanism to specify authorization in a global policy file and formulate *noninterference modulo trusted methods*. Robust declassification [39] relies on integrity to ensure that low integrity flows do not influence high confidentiality data that will later be declassified.

To capture the essence of expressive declassification without appealing to additional mechanisms like integrity or authority, Li and Zdancewic proposed an expressive mechanism for declassification policies that supports the extensional specification of secrets and their intended declassification [21]. A declassification policy is a function that captures *what* information on a confidential value can be *declassified* to eventually produce a public value. For the password checker example, if the declassification policy for password is $\lambda x.\lambda y.x\texttt{==}y$, then an equality comparison with password can be declassified (and thus be made public). However, this declassification policy for password disallows arbitrary declassifications such as revealing the password. Furthermore, declassification can be *progressive*, requiring several operations to be performed in order to obtain public data: *e.g.* $\lambda x.\lambda y.\texttt{hash}(x)\texttt{==}y$ specifies that only the result of comparing the hash of the password for equality can be made public.

The formal security property, called *relaxed noninterference*, states that a secure program can be rewritten into an equivalent program without any variable containing confidential data but whose inputs are confidential and declassified. For the password checker example with $p \triangleq \lambda x.\lambda y.x\texttt{==}y$ as the declassification policy for password, the program login(guess,password) can be rewritten to the equivalent program login'(guess,p(password)) where login' is:

```
String login'(String guess, String→Bool eq){
  if(eq(guess)) ...
}
```

Note that p(password) is a closure that strongly encapsulates the secret value, and only allows equality comparisons.

While the proposal of Li and Zdancewic elegantly and formally captures the essence of flexible declassification while retaining a way to state a clear and extensional security property of interest, it suffers from a number of limitations that jeopardize its practical adoption. First, security labels are sets of functions that form a security lattice whose ordering, based on a semantic interpretation of these sets of functions, is far from trivial [21]:

it relies on a general notion of program equivalences that would be both hard to implement and to comprehend. Second, Li and Zdancewic explicitly rule out *recursive* declassification policies, which are however natural when expressing declassification of recursive data structures. Finally, the rewriting-based definition of relaxed noninterference is unsatisfying for practical software development, as it rigidly requires all secrets to be both *global* and *external*, thereby losing modular reasoning; as recognized by the authors, local language constructs for introducing secrets and their policies are lacking [21].

In this work, we exploit the familiar notion of *type abstraction* to capture declassification policies in a simpler, yet more expressive manner. Type abstraction in programming languages manifests in different ways [25]; here, we specifically adopt the setting of object-oriented programming, where object types are *interfaces*, *i.e.* the set of methods available to the client of an object, and type abstraction is driven by subtyping. For instance, the empty interface type—the root of the subtyping hierarchy—denotes an object that hides all its attributes, which intuitively coincides with secret data, while the interface that coincides with the implementation type of an object exposes all of them, which coincides with public data. Our initial observation is that any interface in between these two extremes denotes *declassification* opportunities. Additionally, choosing objects, as opposed to records, allows us to explore recursive declassification policies from the start, given that the essence of data abstraction in OOP are recursive types [16].

The type-based approach to confidentiality is very intuitive as it only relies on concepts that are readily available in object-oriented languages: a declassification policy is simply a *method signature*, a security label is an *object interface*, and label ordering boils down to *subtyping*. Progressive declassification occurs through chaining of *method invocations*. In fact, the only extension to the standard programming model is that a security type has two facets, each representing the view available to a private and public observer, respectively. In addition to being intuitive, the type-based approach addresses the issues and challenges of the downgrading policies of Li and Zdancewic: *a)* there is no need to rely on general program equivalences to define and decide label ordering, which is just standard, syntactic subtyping; *b)* declassification naturally scales to recursive policies over recursive data structures; and *c)* type-based relaxed noninterference is formulated as a *modular* reasoning principle, and local secrets can be introduced with standard type annotations.

This work makes the following contributions:

- We develop a novel type-based approach to declassification policies, which supports interesting scenarios while appealing to standard programming concepts such as interface types and subtyping (Section 2).
- We capture the essence of type-based declassification in a core object-oriented language, $\mathsf{Ob}_{\mathsf{SEC}}$, in which a security type is a pair of (recursive) object types (Section 3). We describe the static and dynamic semantics of $\mathsf{Ob}_{\mathsf{SEC}}$ and prove type *safety*.
- We specify the formal semantic notion of *type-based relaxed noninterference*, which accounts for type-based declassification policies, independently of any enforcement mechanism (Section 4). We then prove type *soundness* of $\mathsf{Ob}_{\mathsf{SEC}}$: a well-typed program satisfies type-based relaxed noninterference.
- We informally explore how the expressiveness of declassification policies scales with the expressiveness of types (Section 5), identifying interesting venues for extensions.

Section 6 discusses related work and Section 7 concludes. Auxiliary definitions are provided in Appendix.

2 Type-Based Declassification Policies

We now progressively and informally introduce the type-based approach to declassification policies, appealing first to a simple intuitive connection with type abstraction. We then explain why this first intuition is insufficient, and refine it in order to support the key features of a security-typed language with expressive declassification. We end by discussing the security guarantee supported by the approach.

Type abstraction and confidentiality. It is well-known that type abstraction can capture the need to expose only a subset of the operations of an object. For instance, if the password secret is made available using the interface type StringEq $\triangleq$ [eq : String $\to$ Bool], the `login` function from Section 1 can be rewritten as follows:

```
String login(String guess, StringEq password){
  if(password.eq(guess)) ...
}
```

Because password has type StringEq, the login function cannot accidentally leak information about the password. In particular, note that the function cannot even return the password because StringEq is a *supertype* of String, not a subtype. Therefore, the standard substitutability expressed by subtyping seems to align well with the valid information flows permitted in a confidentiality type system: a (public) string value at type String can be used freely, and passed as argument expecting a (mostly) private StringEq, which only exposes equality comparison. Similarly, any value can flow to a private variable, characterized by the empty interface type, $\top \triangleq [\,]$.[1]

Progressive declassification policies can be expressive with *nested* interface types. For instance, assume that String objects have a hash method, of type Unit $\to$ Int. To specify that only the hash of the password can be compared for equality, it suffices to expose the password at type StringHashEq $\triangleq$ [hash : Unit $\to$ IntEq], where IntEq $\triangleq$ [eq : Int $\to$ Bool]:

```
String login(Int guess, StringHashEq password){
  if(password.hash().eq(guess)) ...
}
```

In the code above, the only available operation on password is hash(), which in turn returns an integer that only exposes an equality comparison. Note that here again, StringHashEq $>:$ String and IntEq $>:$ Int.

Recursive declassification. The informal presentation of type-based declassification so far has exemplified two of the main advantages of our approach: security label ordering is syntactic subtyping, and secrets and their declassification policies can be declared locally, by standard type annotations. We now illustrate recursive declassification policies.

Recursive declassification policies are desirable to express interesting declassification of either inductive data structures or object interfaces (whose essence are recursive types [16]). Consider for instance a secret list of strings, for which we want to allow traversal of the

[1] The reader might wonder at this point about the effect of using arbitrary downcasts, as supported in Java. Indeed, downcasts are a way to violate type abstraction, and therefore to violate the type-based security guarantees. For instance, the login function could return (String)password, thereby returning the password for public consumption. Fortunately, there is a simple solution to this issue, which we discuss in Section 5.

structure and comparison of its elements with a given string. This can be captured by the recursive type StrEqList defined as:

$$\mathsf{StrEqList} \triangleq [\mathsf{isEmpty} : \mathsf{Unit} \to \mathsf{Bool},\ \mathsf{head} : \mathsf{Unit} \to \mathsf{StringEq},\ \mathsf{tail} : \mathsf{Unit} \to \mathsf{StrEqList}]$$

To allow traversal, the declassification policy exposes the methods isEmpty, head and tail, with the specific constraints that *a)* accessing an element through head yields a StringEq, not a full String, and *b)* the tail method returns the tail of the list *with the same* declassification policy. Type-based declassification policies can therefore naturally be recursive, as long as the underlying type language allows (some form of) recursive types.

Facets of computation. With the standard programming approach described so far, a program that attempts to violate the declassification protocol of an object is rejected by the (standard) type system because it is ill-typed. For instance:

```
String login(Int guess, StringEq password){
  if(password.length().eq(guess)) ...
}
```

is rejected because length is not part of the exposed interface of password.

However, security-typed languages typically are more flexible than this: they allow computation to proceed with private information, but ensure the result of such computation is itself private [37]. For instance, adding a public integer and a private integer yields a private result. Li and Zdancewic follow the same approach with declassification policies: using a secret in a way that does not follow its declassification policy yields a private result [21]. The justification of these approaches is that computation with private data *is* relevant, but only visible to a high security, private observer; noninterference only dictates that a low security, public observer should not be able to deduce information about private data by observing public outputs.

This means that security-typed languages inherently adopt a multi-faceted view of computation, where each observation level corresponds to a different facet. Sticking to a two-facet, private/public model, the definition of login above is well-typed if one "knows" that password is in fact a String object. In this case using length is valid: it just yields a private result. Flow-sensitivity then ensures that the result of login, which follows from a conditional branching computed based on a private value, is also private.

Faceted types. To accommodate the possibility of computing with private data, we extend standard types to *faceted types*. A security type S, noted $T \triangleleft U$, consists of two standard types: type T for the private interface, and type U for the public interface.[2]. In this paper, we often use the notation T_{L} as a shortcut for the lowest-confidentiality security type $T \triangleleft T$, in which the public facet exposes the same interface as the private facet, and T_{H} for the fully-confidential security type $T \triangleleft \top$ in which the public facet is empty.

To express that password is a private string that can only be declassified through equality comparison, we can use the following signature for login:

```
String_L login(Int_L guess, String◁StringEq password)
```

[2] Similarly to multi-faceted execution [**?**], one can extend the model to support n levels of observations, by introducing security types with n facets.

With this signature the previous definition of login, which invokes length, is still ill-typed. Indeed, the body of the function now has type $\mathsf{String}_\mathsf{L}$, capturing the fact that the resulting string is private, but the signature pretends that the result of login is public, which violates noninterference. For login to be well-typed, either the declared return type should be changed to $\mathsf{String}_\mathsf{H}$, or the conditional should adhere to the public facet StringEq.

Note that subtyping naturally extends *covariantly* to faceted types, *i.e.* $T_1 \triangleleft U_1 <: T_2 \triangleleft U_2$ iff both $T_1 <: T_2$ and $U_1 <: U_2$. Therefore, it is invalid to pass a private string of type $\mathsf{String} \triangleleft \top$ to a function expecting a declassifiable string of type $\mathsf{String} \triangleleft \mathsf{StringEq}$, because $\top$ is not a subtype of StringEq. Subtyping on the public facet corresponds to security label ordering; compared to the semantic, equivalence-based interpretation of labels of Li and Zdancewic, here label ordering is just standard syntactic subtyping.

Object types directly support the possibility to offer different declassification paths for the same secret. For instance, the security type $\mathsf{String} \triangleleft [\mathsf{hash} : \mathsf{Unit}_\mathsf{L} \to \mathsf{Int}_\mathsf{L}, \mathsf{length} : \mathsf{Unit}_\mathsf{L} \to \mathsf{Int}_\mathsf{L}]$ allows a client to obtain a public integer from a string by using either its hash or its length. Naturally, by breadth subtyping, such a secret with two possible declassification paths can also be used as a more restricted secret, *e.g.* one that only exposes its hash publicly.

Type-based relaxed noninterference. The security property we establish in this work is a particular form of termination insensitive noninterference, called *typed-based relaxed noninterference* (TRNI for short). Like the relaxed noninterference result of Li and Zdancewic [21], TRNI accounts for declassification policies.

To understand the intuition behind TRNI, we must first establish a notion of type-based observational equivalence between objects. The starting point of the notion of equivalence is that an object is defined by the observations that can be made on it, that is, by invoking its methods [16]. More precisely, two objects o_1 and o_2 are said to be observationally equivalent at type S, with $S \triangleq T \triangleleft U$, if for each method $m : S_1 \to S_2$ of the *public* facet U, invoking m on o_1 and o_2 with equivalent arguments at type S_1, yields equivalent results at type S_2. Crucially, the definition of equivalence uses the *public* facet of the type, thereby accounting for observational equivalence only up to declassified information.

For example, the strings `"john"` and `"mary"` are not equivalent at type $\mathsf{String} \triangleleft \mathsf{String}$, because a public observer can observe the first character of each string and realize they are different. However, these strings are equivalent when observed at $\mathsf{String} \triangleleft \mathsf{StringLen}$, where $\mathsf{StringLen} \triangleq [\mathsf{length} : \mathsf{Unit}_\mathsf{L} \to \mathsf{Int}_\mathsf{L}]$, because the only declassified information about the strings is their length, which is here equal. This also means that `"john"` and `"james"` are equivalent when are observed at type $\mathsf{String}_\mathsf{H}$ (*i.e.* $\mathsf{String} \triangleleft \top$) since there are no observations available to distinguish them. In fact, any two objects of type T are equivalent at type T_H.

Given this notion of equivalence, a program satisfies TRNI at type S_{out}, if given two inputs that are equivalent at type S_{in}, it produces two results that are equivalent at type S_{out}. Intuitively, the types S_{in} and S_{out} capture the *knowledge* of public observers. Another way to understand TRNI is that, if the initial knowledge *implies* the final knowledge, then the program is secure for the public observer.

For instance, consider a program with an input x of type $\mathsf{String} \triangleleft \mathsf{StringLen}$. The program x.length satisfies TRNI at type $\mathsf{Int} \triangleleft \mathsf{Int}$: two executions of the program with related inputs at $\mathsf{String} \triangleleft \mathsf{StringLen}$, such as `"john"` and `"mary"`, yields two identical results at type $\mathsf{Int} \triangleleft \mathsf{Int}$ (*i.e.* 4 in both cases). However, the program if(x.eq("mary")) return 1 else 2 does not satisfy TRNI at type $\mathsf{Int} \triangleleft \mathsf{Int}$ because there are equivalent inputs at type $\mathsf{String} \triangleleft \mathsf{StringLen}$ (`"john"` and `"mary"`) that yield different outputs at type $\mathsf{Int} \triangleleft \mathsf{Int}$ (1 and 2). For this program, the only secure observation level is $\mathsf{Int} \triangleleft \top$.

$$
\begin{array}{rcll}
e & ::= & v \mid e.m(e) \mid x & \text{(terms)} \\
v & ::= & \left[z : S \Rightarrow \overline{m\,(x)\,e}\right] & \text{(values)} \\
T, U & ::= & O \mid \alpha & \text{(types)} \\
O & ::= & \mathbf{Obj}(\alpha).\,\left[\overline{m : S \to S}\right] & \text{(object type)} \\
S & ::= & T \triangleleft U & \text{(security type)}
\end{array}
\qquad
\begin{array}{rl}
x, y, z & \text{(variables)} \\
\alpha, \beta & \text{(type variables)} \\
m & \text{(method labels)}
\end{array}
$$

$$T_{\mathsf{L}} \triangleq T \triangleleft T \qquad T_{\mathsf{H}} \triangleq T \triangleleft \top$$

Figure 1 $\mathsf{Ob}_{\mathsf{SEC}}$: Syntax

We formally define these notions, and prove that the type system we propose enforces TRNI, in Section 4.

3 An Object Language for Type-Based Declassification

We develop type-based declassification and relaxed noninterference using a core object-oriented language, $\mathsf{Ob}_{\mathsf{SEC}}$, whose syntax is presented in Figure 1. The syntax of $\mathsf{Ob}_{\mathsf{SEC}}$ is similar to that of the object calculi of Abadi and Cardelli [2]. It includes three kinds of expressions: variables, objects and method invocations. Note that we do not include method updates or classes, both unnecessary to formulate our proposal. An object $\left[z : S \Rightarrow \overline{m\,(x)\,e}\right]$ is a collection of method definitions, where method names are unique. The object definition explicitly binds the self variable z in method bodies, with ascribed security type S. The distinguishing feature of $\mathsf{Ob}_{\mathsf{SEC}}$ are security types: as introduced in Section 2, a security type S is a two-faceted type $T \triangleleft U$, where T (resp. U) is the private (resp. public) facet. The public facet corresponds to the declassification policy of an object. A fully opaque secret has type $T \triangleleft \top$ (also noted T_{H}), exposing no method at all, while a low-confidentiality object has type $T \triangleleft T$ (also noted T_{L}), publicly exposing its full interface. A type T or U is either a (recursive) object type $\mathbf{Obj}(\alpha).\,\left[\overline{m : S \to S}\right]$, where method types can use the self type variable α, or a type variable. Note that we do not model parametric polymorphism in this core calculus, so type variables are only used for self types. Following the tradition of Abadi and Cardelli [2], $\mathsf{Ob}_{\mathsf{SEC}}$ does not include base (non-object) types, however they can be easily added or encoded.

Subtyping. The $\mathsf{Ob}_{\mathsf{SEC}}$ subtyping judgment $\Phi \vdash T <: U$ is presented in Figure 2. The subtyping environment Φ is a set of subtyping assumptions between type variables, *i.e.* $\Phi ::= \cdot \mid \Phi, \alpha <: \beta$.[3] For all judgments in this work, we often omit the empty environment, *e.g.* we write $\vdash T <: U$ for $\cdot \vdash T <: U$.

Rule (SObj) accounts for subtyping between object types. Object type T_1 is a subtype of object type T_2 if T_1 has at least the same methods as T_2, possibly more specialized. For this, the rule checks subtyping between method types under a subtyping assumption between the self type variable of T_1 and that of T_2. For instance, consider the following object types:

$$
\begin{array}{rcl}
\mathsf{Counter} & \triangleq & \mathbf{Obj}(\alpha).\,[\mathsf{get} : \mathsf{Unit}_{\mathsf{L}} \to \mathsf{Int}_{\mathsf{L}},\ \mathsf{inc} : \mathsf{Unit}_{\mathsf{L}} \to \alpha_{\mathsf{L}},\ \mathsf{dec} : \mathsf{Unit}_{\mathsf{L}} \to \alpha_{\mathsf{L}}] \\
\mathsf{IncCounter} & \triangleq & \mathbf{Obj}(\beta).\,[\mathsf{get} : \mathsf{Unit}_{\mathsf{L}} \to \mathsf{Int}_{\mathsf{L}},\ \mathsf{inc} : \mathsf{Unit}_{\mathsf{L}} \to \beta_{\mathsf{L}}]\,.
\end{array}
$$

To establish that Counter is a subtype of IncCounter, the covariance between the return types of the `inc` method requires a subtyping assumption between type variables, here $\alpha <: \beta$. Rule (SVar) specifies subtyping between type variables, which only holds if the relation is

[3] Type variables must appear at most once in the subtyping environment.

$\boxed{\Phi \vdash T <: T}$

$$(\text{SObj})\ \frac{O_1 \triangleq \mathbf{Obj}(\alpha).\ [\overline{m : S_1 \to S_2}] \quad O_2 \triangleq \mathbf{Obj}(\beta).\ [\overline{m' : S'_1 \to S'_2}] \quad \overline{m'} \subseteq \overline{m} \quad m_i = m'_j \implies (\Phi, \alpha <: \beta \vdash S'_{1j} <: S_{1i} \quad \Phi, \alpha <: \beta \vdash S_{2i} <: S'_{2j})}{\Phi \vdash O_1 <: O_2}$$

$$(\text{SVar})\ \frac{\alpha <: \beta \in \Phi}{\Phi \vdash \alpha <: \beta} \qquad (\text{SSubEq})\ \frac{O_1 \equiv O_2}{\Phi \vdash O_1 <: O_2} \qquad (\text{STrans})\ \frac{\Phi \vdash T_1 <: T_2 \quad \Phi \vdash T_2 <: T_3}{\Phi \vdash T_1 <: T_3}$$

$\boxed{\Phi \vdash S <: S}$

$$(\text{TSubST})\ \frac{\Phi \vdash T_1 <: T_2 \quad \Phi \vdash U_1 <: U_2}{\Phi \vdash T_1 \triangleleft U_1 <: T_2 \triangleleft U_2}$$

Figure 2 $\mathsf{Ob}_{\mathsf{SEC}}$: Subtyping rules

$\boxed{\mathsf{methsig}(O, m) = S \to S}$

$$\frac{O \triangleq \mathbf{Obj}(\alpha).\ [\overline{m : S_1 \to S_2}] \quad S \triangleq S_{1i}\,[O/\alpha] \quad S' \triangleq S_{2i}\,[O/\alpha]}{\mathsf{methsig}(O, m_i) = S \to S'}$$

$\boxed{m \in O}$

$$\frac{O \triangleq \mathbf{Obj}(\alpha).\ [\overline{m : S_1 \to S_2}]}{m_i \in O}$$

$\boxed{\mathsf{methimpl}(o, m) = x.e}$

$$\frac{o \triangleq [z : S \Rightarrow \overline{m\,(x)\,e}]}{\mathsf{methimpl}(o, m_i) = x.e_i}$$

Figure 3 $\mathsf{Ob}_{\mathsf{SEC}}$: Some auxiliary definitions

in the subtyping environment. Rule (SSubEq) justifies subtyping between *equivalent types*. We consider type equivalence up to renaming and folding/unfolding of self type variables; for instance:

$\mathbf{Obj}(\alpha).\ [\mathsf{m} : \alpha_{\mathsf{L}} \to \alpha_{\mathsf{L}}] \equiv \mathbf{Obj}(\beta).\ [\mathsf{m} : \beta_{\mathsf{L}} \to \beta_{\mathsf{L}}]$ (alpha equivalence)
$\mathbf{Obj}(\alpha).\ [\mathsf{m} : S \to \alpha_{\mathsf{L}}] \equiv \mathbf{Obj}(\alpha).\ [\mathsf{m} : S \to \mathbf{Obj}(\beta).\ [\mathsf{m} : S \to \beta_{\mathsf{L}}]_{\mathsf{L}}]$ (fold/unfold equivalence)

(Appendix A.4 provides the complete definition of type equivalence.)

Rule (STrans) is standard. Rule (TSubST) justifies subtyping between security types, which is covariant in both facets.

Figure 3 presents auxiliary functions used to test method membership in a type ($m \in T$), to get the type of a method in an object type (methsig) and to get the implementation of a method (methimpl). These operations are standard; the only interesting thing to note is that in methsig we close the types in the method signature, by replacing type variables with their object types.

Static semantics. Figure 4 shows the typing rules of $\mathsf{Ob}_{\mathsf{SEC}}$. The type judgment $\Gamma \vdash e : S$ gives a security type to an expression under a type environment Γ that binds variables to types ($\Gamma ::= \cdot \mid \Gamma, x : S$). In what follows, we assume well-formedness of types and environments: informally, an environment is well-formed if all security types are closed and well-formed; a well-formed security type satisfies the requirement that the private type is a subtype of the public type. We further discuss well-formedness at the end of this section.

Rules (TVar) and (TSub) are standard. The (TObj) rule accounts for objects. It requires

$\boxed{\Gamma \vdash e : S}$

$$(\text{TVar})\frac{x \in dom(\Gamma)}{\Gamma \vdash x : \Gamma(x)} \qquad (\text{TSub})\frac{\Gamma \vdash e : S' \quad \vdash S' <: S}{\Gamma \vdash e : S}$$

$$(\text{TObj})\frac{S \triangleq T \triangleleft U \quad \mathsf{methsig}(T, m_i) = S_i' \to S_i'' \quad \Gamma, z : S, x : S_i' \vdash e_i : S_i''}{\Gamma \vdash \left[z : S \Rightarrow \overline{m\,(x)\,e}\right] : S}$$

$$(\text{TmD})\frac{\Gamma \vdash e_1 : T \triangleleft U \qquad m \in U \qquad \mathsf{methsig}(U, m) = S_1 \to S_2 \qquad \Gamma \vdash e_2 : S_1}{\Gamma \vdash e_1.m(e_2) : S_2}$$

$$(\text{TmH})\frac{\Gamma \vdash e_1 : T \triangleleft U \qquad m \notin U \qquad \mathsf{methsig}(T, m) = S_1 \to T_2 \triangleleft U_2 \qquad \Gamma \vdash e_2 : S_1}{\Gamma \vdash e_1.m(e_2) : T_2 \triangleleft \top}$$

Figure 4 $\mathsf{Ob}_{\mathsf{SEC}}$: Static semantics

each method body to be well-typed with respect to the private facet of the object. In particular, the method body must match the return type of the method signature in the private facet of the self type S.

From a security point of view, the interesting rules are the ones for method invocation. Rule (TmD) applies when the invoked method is part of the *public* facet of the receiver. In this case, because the method invocation respects the declassification policy, the overall type of the invocation is the return type of the method in the public facet. This expresses that the invocation advances a step in the progressive declassification of the object. For instance, if the expression e_1 has the public type $\mathsf{StringHashEq} \triangleq [\mathsf{hash} : \mathsf{Unit}_{\mathsf{L}} \to \mathsf{Int} \triangleleft \mathsf{IntEq}]$, the invocation $e_1.\mathsf{hash}()$ has type $\mathsf{Int} \triangleleft \mathsf{IntEq}$, expressing that the returned value is a secret that can further be declassified by calling the method eq from IntEq.

Rule (TmH) applies when the method is not in the public type U, but only in the private type T (if the method is not in T, the expression is ill typed). In this case, the method call is accessing the "secret" part of the object: the result of the method invocation must therefore be protected by changing its public facet to $\top$. This rule captures the design decision that using a secret beyond its declassification policy is allowed, but the result must be secret. In other words, only a private observer can use objects beyond their declassification policies; to a public observer, the results of these interactions are unobservable.[4]

Dynamic semantics. We define a standard call-by-value small-step semantics for $\mathsf{Ob}_{\mathsf{SEC}}$, based on evaluation contexts $E ::= [\,] \mid E.m(e) \mid v.m(E)$.

The language includes a single reduction rule, for method invocation, which is standard:

$$(\text{EMInv})\frac{o \triangleq [z : \_ \Rightarrow \_] \qquad \mathsf{methimpl}(o, m) = x.e}{E[o.m(v)] \longmapsto E[e\,[o/z]\,[v/x]]}$$

Type safety. We now establish that well-typed $\mathsf{Ob}_{\mathsf{SEC}}$ programs are safe. Note that type safety does not provide any *security guarantees* for $\mathsf{Ob}_{\mathsf{SEC}}$. (Security guarantees will be

[4] Access modifiers in object-oriented languages, such as private and public in Java are a really different mechanism. Such modifiers are about *encapsulation*, not about *information flow*. The essential difference can be observed in rule (TmH), which propagates privacy on return values.

$$\begin{array}{rcl}
\mathcal{V}_k[\![S]\!] & = & \{v = [z : S_1 \Rightarrow _] \mid S \triangleq T \triangleleft U \quad \vdash S_1 <: S \wedge \\
& & \quad (\forall j < k.\ v \in \mathcal{V}_j[\![S_1]\!] \wedge \\
& & \quad\quad (\forall m \in T, v'.\ \mathsf{methsig}(T, m) = S' \to S'' \quad \mathsf{methimpl}(v, m) = x.e \\
& & \quad\quad\ v' \in \mathcal{V}_j[\![S']\!] \implies e\,[v/z]\,[v'/x] \in \mathcal{C}_j[\![S'']\!]))\} \\
\\
\mathcal{C}_k[\![S]\!] & = & \{e \mid \forall j < k.\ \forall e'.(e \longmapsto^j e' \ \wedge\ \mathsf{irred}(e')) \implies e' \in \mathcal{V}_{k-j}[\![S]\!]\}
\end{array}$$

Figure 5 $\mathsf{Ob}_{\mathsf{SEC}}$: Unary logical relation for safety

addressed in Section 4.) A program e is *safe*, noted $\mathsf{safe}(e)$, if it does not get stuck, *i.e.* if it either reduces to a value or diverges.

▶ **Definition 1** (Safety). $\mathsf{safe}(e) \iff \forall e'.\ e \longmapsto^* e' \implies e' = v \text{ or } \exists e''.\ e' \longmapsto e''$

We prove type safety for $\mathsf{Ob}_{\mathsf{SEC}}$ using a semantic interpretation of types as a unary logical relation [3]. We cannot however define the logical relation based on a direct induction over the structure of types, because of recursive types, which would make such a definition ill-founded. Therefore, we use a step-indexed logical relation [4, 6]. We establish an intermediary result for a fixed number k of steps, meaning that a term is safe for k evaluation steps, and then quantify $\forall k \geq 0$ to obtain the general result. Step indexing ensures the well-foundedness of the logical relation.

Figure 5 defines the unary logical relation that captures the safety interpretation of types as values and computations, in a mutually recursive manner. The set $\mathcal{V}_k[\![S]\!]$ denotes the safe value interpretation of type S for k steps; it contains all the *values* (*i.e.* objects) for which it is safe (for any $j < k$ number of steps) to invoke methods of the private type T of the security type $S \triangleq T \triangleleft U$. Note that the definition needs to assume that the self object is in the value interpretation of S, for $j < k$ steps; without step-indexing, this relation would be ill-founded due to the recursive nature of objects through their self variables. The set $\mathcal{C}_k[\![S]\!]$ contains all the *expressions* that can be safely executed for k steps at the security type S. In the definition, the $\mathsf{irred}(e)$ predicate denotes irreducible expressions, *i.e.* expressions e such that $\nexists e'.e \longmapsto e'$.

We define *semantic typing*, written $\models e : S$, to denote that a closed expression e executes safely for any fixed number of steps:

▶ **Definition 2** (Semantic typing). $\models e : S \iff \forall k \geq 0.\ e \in \mathcal{C}_k[\![S]\!]$.

We then first prove that semantic typing does imply safety as per Definition 1.

▶ **Lemma 3** (Semantic type safety). $\models e : S \implies \mathsf{safe}(e)$

Proof. To show $\mathsf{safe}(e)$ we need to consider an arbitrary e' such that $e \longmapsto^* e'$ and then show that either $e' = v$ or $\exists e''.\ e' \longmapsto e''$
Let us consider an arbitrary j_1 to count the step that takes $e \longmapsto^* e'$. Let us denote $l = j_1 + 1$
By expanding the definition of $\models e : S$ we have $\forall k \geq 0.\ e \in \mathcal{C}_k[\![S]\!]$. We instantiate this with $k = l$ to obtain $e \in \mathcal{C}_l[\![S]\!]$. By expanding this we have:
$\forall j < l.\ \forall e_1.(e \longmapsto^j e_1 \ \wedge\ \mathsf{irred}(e_1)) \implies e_1 \in \mathcal{V}_{k-j}[\![S]\!]$. We instantiate $e \in \mathcal{C}_l[\![S]\!]$ with j_1 and e' and we obtain: $(e \longmapsto^{j_1} e' \ \wedge\ \mathsf{irred}(e')) \implies e' \in \mathcal{V}_{k-j_1}[\![S]\!]$.
There are two cases to consider: $\neg\mathsf{irred}(e')$ and $\mathsf{irred}(e')$. If $\neg\mathsf{irred}(e')$, then by definition $\exists e''.\ e' \longmapsto e''$. If $\mathsf{irred}(e')$, we have that $e' \in \mathcal{V}_{k-j}[\![S]\!]$, so e' is a value. ◀

Second, we prove that syntactic typing (Figure 4) implies semantic typing.

▶ **Lemma 4** (Syntactic typing implies semantic typing). $\vdash e : S \implies \models e : S$

Proof. The result follows from a similar lemma on open terms: $\Gamma \vdash e : S \implies \Gamma \models e : S$. We define a standard notion of safe value substitutions [3], *i.e.* partial maps from variables to safe values, $\gamma \in \mathcal{G}_k[\![\Gamma]\!]$ and $\Gamma \models e : S$ as follows:
$\gamma \in \mathcal{G}_k[\![\Gamma]\!] \iff dom(\gamma) = dom(\Gamma)$ and $\forall x \in dom(\Gamma).\gamma(x) \in \mathcal{V}_k[\![\Gamma(x)]\!]$
$\Gamma \models e : S \iff \forall k \geq 0,\ \forall \gamma.\ \gamma \in \mathcal{G}_k[\![\Gamma]\!] \implies \gamma(e) \in \mathcal{C}_k[\![S]\!]$.
Then we prove that $\Gamma \vdash e : S \implies \Gamma \models e : S$ by induction on the typing derivation of e. The case (TVar) is direct from the definition of $\gamma \in \mathcal{G}_k[\![\Gamma]\!]$. The case (TSub) follows directly from a subsumption lemma ($e \in \mathcal{C}_k[\![S]\!] \wedge \vdash S <: S' \implies e \in \mathcal{C}_k[\![S']\!]$). Cases (TObj), (TmD) and (TmH) are proven by unfolding the definitions of $\mathcal{C}_k[\![S]\!]$ and $\mathcal{V}_k[\![S]\!]$, and applying the induction hypotheses for smaller indexes. For these cases, we use mainly a monotonicity lemma for the value interpretation of a type regarding the index, *i.e.* $e \in \mathcal{V}_k[\![S]\!] \wedge j \leq k \implies v \in \mathcal{V}_j[\![S]\!]$. ◀

Together, Lemmas 3 and 4 imply that well-typed programs are safe.

▶ **Theorem 5** (Syntactic type safety). $\vdash e : S \implies \mathsf{safe}(e)$

Now that we have established that $\mathsf{Ob}_{\mathsf{SEC}}$ is a well-defined, type-safe language, Section 4 will develop its security guarantees.

A note on well-formedness. Before we proceed, however, we need to mention a technical yet important issue that we overlooked so far. For the main results of Section 4 to hold, we need to ensure that we work with *well-formed* security types, *i.e.* that the private facet type is a subtype of the public facet type. In a language with simple, non-recursive types, defining such subtyping constraints is straightforward. However, in the presence of recursive (object) types, defining the rules for the subtyping constraint of security types is rather subtle and involved. The subtlety with type variables is that, at some point, we might have to check well-formedness of a security type with a type variable in one of its facets, *e.g.* $\alpha \triangleleft T$, without knowing any relation between α and T. To address this, we need *to remember* the surrounding recursive object type O that binds α, and to transform the check $\vdash \alpha <: T$ to $\vdash O <: T$. For conciseness, we leave out the well-formedness rules from the main body of the paper; they are fully described in Appendix A.2. In what follows, we systematically assume that security types (and by extension, type environments) are well-formed.

4 Type-Based Relaxed Noninterference

Faceted security types support information-flow security with declassification. The security property that type-based declassification supports is a form of relaxed noninterference [21], which we informally explained in Section 2. This section formally defines the notion of type-based relaxed noninterference (TRNI) *independently of any enforcement mechanism*. Then, we prove that the type system of $\mathsf{Ob}_{\mathsf{SEC}}$ is sound with respect to this property.

Type-based equivalence. As introduced in Section 2, TRNI is defined in terms of a notion of type-based equivalence between objects: a program satisfies TRNI at type S_{out}, if given two inputs at type S_{in}, it produces two equivalent results at type S_{out}. Equivalence at a type accounts for the possible observations (*i.e.* method invocations) that one is allowed to make on an object. We define this equivalence as a step-indexed logical relation [4], in Figure 6.

$$
\begin{array}{lcl}
v_1 \approx_k v_2 : \mathcal{V}[\![S]\!] & \iff & S \triangleq T \triangleleft U \quad v_i \triangleq [z : \_ \Rightarrow \_] \\
& & \vdash_1 v_i : T \wedge (\forall m \in U.\quad \mathsf{methsig}(U, m) = S' \to S'' \quad \mathsf{methimpl}(v_i, m) = x.e_i \\
& & \forall j < k, v_1', v_2'.\ \ v_1 \approx_j v_2 : \mathcal{V}[\![S]\!] \wedge \\
& & \quad (v_1' \approx_j v_2' : \mathcal{V}[\![S']\!] \implies e_1\,[v_1/z]\,[v_1'/x] \approx_j e_2\,[v_2/z]\,[v_2'/x] : \mathcal{C}[\![S'']\!])) \\
\\
e_1 \approx_k e_2 : \mathcal{C}[\![S]\!] & \iff & S \triangleq T \triangleleft U \\
& & \vdash_1 e_i : T \wedge (\forall j < k.(e_1 \longmapsto^{\leq j} v_1 \wedge e_2 \longmapsto^{\leq j} v_2) \implies v_1 \approx_{k-j} v_2 : \mathcal{V}[\![S]\!])
\end{array}
$$

Figure 6 Step-indexed logical relation for type-based equivalence

We define how to relate values (*i.e.* objects) as well as computations (*i.e.* expressions). Step indexing is required due to the recursive nature of object types, as explained below.

Note that the definitions use a simple typing judgment that does not account for security typing at all; its sole purpose is to ensure safety. This is crucial: the public facets of security types only play the role of *specifications* of declassification policies, and the logical relation specifies the *meaning* of these specifications, without any consideration for an enforcement mechanism. In particular, observe that the definitions in Figure 6 do *not* appeal to security type judgments ($\vdash$), but only to simple type judgments ($\vdash_1$).

▶ **Definition 6** (Simple typing judgment)**.** Based on the security typing judgment $\Gamma \vdash e : S$, we define the simple typing judgment $\Gamma \vdash_1 e : T$ by focusing only on the private facet of security types. Formally: $\Gamma \vdash_1 e : T \iff \Gamma \vdash e : T \triangleleft U$ for some U.
(The inductive definition of simple typing is in Appendix A.5.)

Intuitively, two objects v_1 and v_2 are equivalent at type $S \triangleq T \triangleleft U$ for k steps, noted $v_1 \approx_k v_2 : \mathcal{V}[\![S]\!]$, when one cannot distinguish them by invoking any method m of U. More precisely, to ensure safety, we first demand that both values are well-typed at T with the simple type system. Then, for each method $m \in U$ and every $j < k$, the invocations of m on v_1 and v_2 with related arguments at the argument type S' of m must be equivalent computations at the return type S'' for j steps, as defined below. Finally, note that the definition also requires that v_1 and v_2 are related self objects, for $j < k$ steps; this is necessary for the relation to be well-founded. (Observe that two simply well-typed objects are vacuously equivalent for zero steps.)

Two expressions e_1 and e_2 are equivalent at security type $S \triangleq T \triangleleft U$ for k steps, noted $e_1 \approx_k e_2 : \mathcal{C}[\![S]\!]$, if they are both (simply) well-typed at T and, provided that they both reduce to values in *at most* $j < k$ steps (noted $e \longmapsto^{\leq j} v$), then both values are equivalent at type S for the remaining $k - j$ steps. Note that this definition is termination insensitive: if one expression does not terminate in less than k steps, then both expressions are deemed equivalent.

Defining TRNI. The type-based approach to declassification policies allows us to formulate the corresponding relaxed noninterference property as a *modular* reasoning principle, similarly to the common formulation of noninterference in languages without declassification [37], thereby avoiding the global and external formulation of the transformation approach [21].

Standard noninterference is usually stated as a modular reasoning principle on open terms [37]: given a well-typed open term, which depends on some private variables, closing the term with private inputs yields equivalent programs when observed by a low-confidentiality observer. This statement can be generalized using the notion of *value substitutions*, *i.e.* partial maps from variables to values: given an open term that typechecks in

a given environment Γ, applying two *related* substitutions yields equivalent computations. Applying a substitution, noted $\gamma(e)$, substitutes the free variables of e with their values in γ.

▶ **Definition 7** (Satisfactory substitution)**.** A substitution γ satisfies type environment Γ, noted $\gamma \models \Gamma$, iff $dom(\gamma) = dom(\Gamma) \wedge \forall x \in dom(\Gamma).\ \vdash_1 \gamma(x) : T$ where $\Gamma(x) \triangleq T \triangleleft U$

▶ **Definition 8** (Related substitutions)**.** Two substitutions γ_1 and γ_2 are equivalent for k steps with respect to a type environment Γ, noted $\gamma_1 \approx_k \gamma_2 : \mathcal{G}[\![\Gamma]\!]$, if $\gamma_i \models \Gamma$ and

$$\forall x \in dom(\Gamma).\gamma_1(x) \approx_k \gamma_2(x) : \mathcal{V}[\![\Gamma(x)]\!]$$

The statement of type-based relaxed noninterference is a direct generalization of standard noninterference: an open term e, simply well-typed in environment Γ, satisfies type-based relaxed noninterference at security type S, noted $\mathsf{TRNI}(\Gamma, e, S)$, if two executions of e with related substitutions with respect to Γ produce equivalent computational expressions at type S, for any number of steps.

▶ **Definition 9** (Type-based relaxed noninterference)**.**

$$\begin{aligned}\mathsf{TRNI}(\Gamma, e, S) \iff\ & S \triangleq T \triangleleft U \quad \Gamma \vdash_1 e : T \wedge \\ & \forall k \geq 0.\ \forall \gamma_1, \gamma_2.\ \gamma_1 \approx_k \gamma_2 : \mathcal{G}[\![\Gamma]\!] \implies \gamma_1(e) \approx_k \gamma_2(e) : \mathcal{C}[\![S]\!]\end{aligned}$$

This definition captures the semantic characterization of TRNI-secure expressions, independently of any enforcement mechanism (recall that, in Figure 6, the public facets of security types only play the role of *specifications* of declassification policies). The $\mathsf{Ob}_{\mathsf{SEC}}$ type system is a sound, conservative enforcement mechanism for TRNI.

Security type soundness. To establish that well-typed $\mathsf{Ob}_{\mathsf{SEC}}$ programs satisfy TRNI, we first introduce a general notion of type-based equivalence between open expressions. Two open expressions, well-typed under a type environment Γ, are equivalent at a security type $S \triangleq T \triangleleft U$, if both expressions have simple type T, and given two related value substitutions for Γ, closing each expression with a satisfactory substitution yields equivalent expressions at type S.

▶ **Definition 10** (Equivalence of open terms)**.**

$$\begin{aligned}\Gamma \vdash e_1 \approx e_2 : S \iff\ & S \triangleq T \triangleleft U \quad \Gamma \vdash_1 e_i : T \wedge \\ & \forall k \geq 0.\ \forall \gamma_1, \gamma_2.\ \gamma_1 \approx_k \gamma_2 : \mathcal{G}[\![\Gamma]\!] \implies \gamma_1(e_1) \approx_k \gamma_2(e_2) : \mathcal{C}[\![S]\!]\end{aligned}$$

As is clear from the definitions, if a term is equivalent to itself at type S, then it satisfies TRNI at S.

▶ **Lemma 11** (Self-equivalence)**.** $\Gamma \vdash e \approx e : S \implies \mathsf{TRNI}(\Gamma, e, S)$

Type soundness of $\mathsf{Ob}_{\mathsf{SEC}}$ follows from the fact that the $\mathsf{Ob}_{\mathsf{SEC}}$ type system enforces such a self-equivalence.

▶ **Lemma 12** (Fundamental property)**.** $\Gamma \vdash e : S \implies \Gamma \vdash e \approx e : S$

Proof. The proof is by induction on the typing derivation of e. The (TVar) case follows directly from Definition 8 and the (TSub) case follows from a subtyping lemma: if $e_1 \approx_k e_2 : \mathcal{C}[\![S]\!]$ and $\vdash S <: S'$ then $e_1 \approx_k e_2 : \mathcal{C}[\![S']\!]$. The (TObj) case applies the induction hypothesis (IH) on method bodies. To use the IH results, we need to show that the value

substitutions that result from extending the current substitutions with both self and actual arguments are also related. This step requires auxiliary lemmas of monotonicity of the logical relations regarding smaller indexes. The (TmD) case follows from applying the IH over both subexpressions, selecting adequate indexes. The (TmH) case is simpler because there is no method to invoke in the public type $\top$. ◀

Finally, type soundness for $\mathsf{Ob}_{\mathsf{SEC}}$ follows directly from Lemmas 11 and 12.

▶ **Theorem 13** (Security type soundness). $\Gamma \vdash e : S \implies \mathsf{TRNI}(\Gamma, e, S)$

Illustration. We now illustrate the relation between the security typing and the definition of TRNI. In the examples we use some standard constructs like conditionals, not included in $\mathsf{Ob}_{\mathsf{SEC}}$, but easily encodable.

As introduced in Section 2, the property $\mathsf{TRNI}(\Gamma, e, T \triangleleft U)$ can be intuitively understood as: the initial knowledge of a public observer in Γ (*i.e.* the declassification policies) implies the final knowledge (*i.e.* the resulting public type U) that the observer has at hand to distinguish the results of two arbitrary executions of the *secure* program e of simple type T.

Let us recall the type $\mathsf{StringLen} \triangleq [\mathsf{length} : \mathsf{Unit_L} \to \mathsf{Int_L}]$ from the end of Section 2. Consider the open term $e \triangleq \mathsf{x.length}$ under the type environment $\Gamma \triangleq x : \mathsf{String} \triangleleft \mathsf{StringLen}$. The judgment $\Gamma \vdash e : \mathsf{Int_L}$ ensures that $\mathsf{TRNI}(\Gamma, e, \mathsf{Int_L})$ holds. It says that executing e, with two different strings v_1 and v_2 *of the same length* is secure because the observer *does not learn anything new* by exploiting the knowledge of distinguishing the resulting integers with any method of Int. In fact, if we use the definition of TRNI, for any equivalent substitutions γ_1 and γ_2 such that $\gamma_1 \approx_k \gamma_2 : \mathcal{G}[\![\Gamma]\!]$, such as $\gamma_i \triangleq x \mapsto v_i$, we need to show $\gamma_1(x).\mathsf{length}() \approx_k \gamma_2(x).\mathsf{length}() : \mathcal{C}[\![\mathsf{Int_L}]\!]$. It is easy to see that this result follows from the assumption that v_1 and v_2 have the same length (*i.e.* are equivalent at $\mathsf{String} \triangleleft \mathsf{StringLen}$).

We have a different situation if we consider $e' \triangleq \mathsf{if(x.eq("mary"))\ return\ 1\ else\ 2}$, with the same type environment Γ. We cannot prove that $\mathsf{TRNI}(\Gamma, e', \mathsf{Int_L})$ holds, meaning this program is *not* secure at type $\mathsf{Int_L}$. Indeed, take $\gamma_1 \triangleq x \mapsto \texttt{"mary"}$ and $\gamma_2 \triangleq x \mapsto \texttt{"john"}$. Because both strings have the same length, we have $\texttt{"mary"} \approx_k \texttt{"john"} : \mathcal{V}[\![\mathsf{String} \triangleleft \mathsf{StringLen}]\!]$, so the two substitutions are equivalent. However, we cannot show that $\gamma_1(e') \approx_k \gamma_2(e') : \mathcal{C}[\![\mathsf{Int_L}]\!]$, because this requires to show that $1 \approx_k 2 : \mathcal{V}[\![\mathsf{Int_L}]\!]$, which is obviously false.

The type system of $\mathsf{Ob}_{\mathsf{SEC}}$ indeed rejects the judgment $\Gamma \vdash e : \mathsf{Int_L}$. It does accept the judgment $\Gamma \vdash e : \mathsf{Int_H}$, meaning that e' *is* secure at type $\mathsf{Int_H}$. This is correct because then the public observer has no ability to compare the resulting values of e'. Note in fact that any simply well-typed expression of type T is secure at type T_{H}. Such expressions are opaque to a public observer, but are observable by a private observer.

Principles of declassification. Our approach to type-based declassification satisfies the declassification principles stated by Sabelfeld and Sands [28].[5] We now briefly introduce each principle and informally argue why it is respected.

- *Conservativity—i.e.* "*Security for programs with no declassification is equivalent to noninterference*". It is easy to see that if a program satisfies $\mathsf{TRNI}(\Gamma, e, T_{\mathsf{L}})$, for some T, and all security types in both Γ and e are either highly confidential (T_{H}) or not confidential

[5] Sabelfeld and Sands mention a fourth principle, *non-occlusion*, which addresses the interaction between declassification and covert channels, such as heap assignments, exceptions or termination behavior. $\mathsf{Ob}_{\mathsf{SEC}}$ has neither mutation nor control operators, and termination is not considered a covert channel because we only deal with termination-insensitive noninterference.

at all (T_{L}), then the definition of TRNI coincides exactly with the definition of pure noninterference [37]. Therefore type-based relaxed noninterference is a generalization of pure noninterference.

- *Monotonicity of Release—i.e.* "*Adding further declassifications to a secure program cannot render it insecure*". This lemma follows from subtyping naturally. Recall that in our approach, in the judgment $\mathsf{TRNI}(\Gamma, e, S)$, declassification policies come from types ascribed in both Γ and e. "Adding further declassification" in the inputs means in our context replacing security types in Γ with subtypes, more precisely, where the public facets are subtypes of the original types. The security typing judgment also holds in this scenario of additional declassification in the inputs. Similarly, adding declassification in the expression e means specializing the public facets of types in object type declarations. Again, this does not affect the semantic TRNI judgment. Note, however, that if argument types are specialized, the program might not be typable anymore with the security type system, as such a change breaks the contravariance of subtyping for argument method types.

- *Semantic Consistency—i.e.* "*The (in)security of a program is invariant under semantics-preserving transformations of declassification-free subprograms.*". The principle says that it is possible to replace an expression that does not use declassification with another semantically-equivalent expression, without affecting security. As observed by Sabelfeld and Sands, the approach to declassification policies of Li and Zdancewic [21] violates this principle, because they rely on a restricted, mostly-syntactic form of program equivalence to decide label ordering. Therefore, many semantically-equivalent programs are not deemed equivalent, hence affecting their (in)security. In contrast, our notion of type-based equivalence (Figure 6) is semantic, not syntactic.

Limitations of security typing. The $\mathsf{Ob}_{\mathsf{SEC}}$ type system is a *static* enforcement mechanism for type-based relaxed noninterference. As such, it is inherently conservative. This has two implications regarding Theorem 12.

First, the type system can reject some programs that are in fact secure. For example, consider the following definitions:

$T \triangleq \mathbf{Obj}(\alpha).\,[n : \mathsf{String}_{\mathsf{L}} \to \mathsf{String}_{\mathsf{L}}]$
$T' \triangleq \mathbf{Obj}(\alpha).\,[m : \mathsf{String}_{\mathsf{H}} \to \mathsf{String}_{\mathsf{H}}]$
$v \triangleq [z : T_{\mathsf{L}} \Rightarrow n\,(x)\,\texttt{"hello"}]$
$v' \triangleq [z : T'_{\mathsf{L}} \Rightarrow m\,(x)\,v.n(x)]$

Here, v' is not well-typed using the security type system, because of the call $v.n(x)$ ($\vdash \mathsf{String}_{\mathsf{H}} \not<: \mathsf{String}_{\mathsf{L}}$). However, we can show that v' does satisfy $\mathsf{TRNI}(\cdot, v', T'_{\mathsf{L}})$, because a public observer always obtains the same result (*i.e.* "hello") for any two secrets passed to method m; the program is not leaking any information.

Second, the type system can assign the security type $T \triangleleft \top$ to an expression, despite the fact that $\top$ is not the tighter secure type for TRNI to hold. For instance, let us assume that Int has built-in methods mod2 and mod4 with the standard mathematical meaning, and we define the type $\mathsf{IntMod4} \triangleq [\mathsf{mod4} : \mathsf{Unit}_{\mathsf{L}} \to \mathsf{Int}_{\mathsf{L}}]$. Consider $\Gamma \triangleq v : \mathsf{Int} \triangleleft \mathsf{IntMod4}$ and $e \triangleq \mathsf{v.mod2()}$. The type system admits $\Gamma \vdash e : \mathsf{Int}_{\mathsf{H}}$, which implies $\mathrm{TRNI}(\Gamma, e, \mathsf{Int}_{\mathsf{H}})$, but it does not admit $\Gamma \vdash e : \mathsf{Int}_{\mathsf{L}}$; despite the fact that $\mathrm{TRNI}(\Gamma, e, \mathsf{Int}_{\mathsf{L}})$ also holds—because if a and b are equivalent modulo 4, then they are also equivalent modulo 2.

5 Expressiveness of Declassification Policies

Our approach to type-based declassification policies builds upon an underlying type system. While we have chosen a simple model of recursive object types to develop the approach in the previous sections, it is interesting to explore how the expressiveness of the underlying type discipline affects the range of declassification policies that can be defined.

Recursive types. It is possible to exploit the idea of type-based declassification policies without recursive object types. We only need a type abstraction mechanism, such as that enabled by subtyping. In fact, with only record types and subtyping, we can already capture a set of interesting policies, such as those mentioned at the begin of Section 2 (*e.g.* StringEq, StringHashEq). TRNI depends on the notion of equivalence between values and computations, which can be easily simplified for the non-recursive setting; in particular, we can get rid of step-indexing in the logical relations.

Of course, without recursive object types in the core formalism, we lose the ability to express recursive declassification policies (which are useful to declassify recursive data structures, as illustrated in Section 2). With records but without objects, we can add general recursive types of the form $\mu X.T$ to support recursive declassification policies. Note however that combining general recursive types and subtyping is challenging, and there are different definitions that may not be complete (*i.e.* unable to establish a subtyping relation that indeed holds); in particular, our subtyping rules are not complete regarding subtyping between infinite trees [5]. This challenge solely affects the kinds of security types that can be defined and deemed well-formed.

Finally, one characteristic of recursive declassification policies is that they potentially allow to chain arbitrarily many invocations of a declassification method. For instance, consider an infinite stream of strings, and a declassification that allows equality comparisons on its elements:

$$\mathsf{StrEqStream} \triangleq [\mathsf{head} : \mathsf{Unit}_\mathsf{L} \to \mathsf{StringEq}_\mathsf{L},\ \mathsf{tail} : \mathsf{Unit}_\mathsf{L} \to \mathsf{StrEqStream}_\mathsf{L}]$$

In case tolerating an unbounded number of observations would represent an unacceptable accumulated leak, the programmer can define a more restrictive declassification policy that restricts the number of tolerated calls by explicitly nesting interface types instead of defining a fully recursive one. Obviously, to be practical, one would need to define a convenient surface syntax such as:

$$\mathsf{StrEqStream} \triangleq [\mathsf{head} : \mathsf{Unit}_\mathsf{L} \to \mathsf{StringEq}_\mathsf{L},\ \mathsf{tail} : \mathsf{Unit}_\mathsf{L} \to \mathsf{StrEqStream}_\mathsf{L}@k]$$

to specify that the declassification policy only supports at most k unfoldings of StrEqStream through tail, and to desugar it to a finite nesting of interface types.

Universal types. Universal types allow programmers to define programs that are parameterized by types. This can be used to define generic data structures, such as lists:

$$\mathsf{List}\,[X] \triangleq \{\mathsf{isEmpty} : \mathsf{Unit}_\mathsf{L} \to \mathsf{Bool}_\mathsf{L},\ \mathsf{head} : \mathsf{Unit}_\mathsf{L} \to X_\mathsf{L},\ \mathsf{tail} : \mathsf{Unit}_\mathsf{L} \to \mathsf{List}[X]_\mathsf{L}\}$$

If we add parametric polymorphism to $\mathsf{Ob}_{\mathsf{SEC}}$, then in addition to get polymorphism over implementation types, we naturally get a general form of *security label polymorphism*, which is very useful (and supported in Jif [23]). For example, we can define generic data structures that are polymorphic with respect to the security labels of their inner data; the list structure defined above is a specific example.

Similarly, a declassification policy can exploit parametric polymorphism. Recall the recursive declassification example of Section 2, in which we allow traversing a list and only comparing its elements with a given public element. We can express a generic version of this declassification policy with the following type:

$$\mathsf{ListEq}\,[X] \triangleq [\mathsf{isEmpty} : \mathsf{Unit_L} \to \mathsf{Bool_L},\ \mathsf{head} : \mathsf{Unit_L} \to X \triangleleft \mathsf{Eq}[X],\ \mathsf{tail} : \mathsf{Unit_L} \to \mathsf{ListEq}[X]_\mathsf{L}]$$
$$\mathsf{Eq}\,[X] \triangleq [\mathsf{eq} : X_\mathsf{L} \to \mathsf{Bool_L}]$$

Note that the above definition is however invalid, because ListEq is not well-formed: in order to satisfy the subtyping constraint between the facets of a security type such as $X \triangleleft \mathsf{Eq}[X]$, we need to bound the type variable X, which leads us to *bounded* parametric polymorphism. Then, the type ListEq can be correctly defined as follows:

$$\mathsf{ListEq}\,[X <: \mathsf{Eq}\,[X]] \triangleq$$
$$[\mathsf{isEmpty} : \mathsf{Unit_L} \to \mathsf{Bool_L},\ \mathsf{head} : \mathsf{Unit_L} \to X \triangleleft \mathsf{Eq}[X],\ \mathsf{tail} : \mathsf{Unit_L} \to \mathsf{ListEq}[X]_\mathsf{L}]$$

Refinement types. Refinement types, as found in *e.g.* LiquidHaskell [34], enrich standard types with predicates over a decidable logic. For instance, the type $\{x : \mathsf{Int} \mid x \geq 0\}$ denotes natural numbers. Additionally, refinement types usually support a form of dependent types, allowing refinements to refer to variables in scope as well as function arguments. Combining such expressive types with our approach allows interesting declassification policies to be defined, such as restricting successive arguments of a progressive declassification.

As an example, consider the following policy:

$$\mathsf{IntModProd} \triangleq [\mathsf{mod} : \{x : \mathsf{Int_L}\} \to [\mathsf{mult} : \{y : \mathsf{Int_L} \mid x = y\} \to \mathsf{Int_L}]_\mathsf{L}]$$

This progressive declassification allows revealing the result of the chain of invocations mod then mult, *only if the argument to both invocations is the same*. Note that IntModProd is a proper supertype of Int, since $\{y : \mathsf{Int} \mid x = y\}$ is a subtype of Int.

More advanced scenarios. There are other interesting declassification policies that seem more challenging to support with our type-based approach. An interesting example is specifying that a string secret can be leaked only after it has been encrypted; it is highly unlikely that the standard String class exposes an encryption method. However, our approach does appeal to the actual interface of an object in order to define its declassification. Hicks *et al.* [20] introduce special *declassifier* functions to express arbitrary declassification that can involve operations that are not defined on the declassified object itself. Therefore a possible solution to address this example in our setting would be to rely on an external method specification mechanism, such as open classes or mixin-based composition of traits in Scala.

Nevertheless, the above approach would still fall short of expressing *global* declassification policies, as described by Li and Zdancewic [21], which can relate the declassification of different secrets at once. While the value dependencies can be expressed using, *e.g.* refinement types, the challenge is to ensure that the obtained security types are still well-formed (*i.e.* the public facet must be a supertype of the private facet). These are interesting challenges for future development of the approach.

A note about casts. In Section 2 we alluded to the challenge of integrating explicit downcasts in a language that adopts type-based declassification policies. Casts can be soundly incorporated in such a language provided that we only allow casting values from a security

type to another one that has the *same public type*, *i.e.* casts cannot affect the declassification policy. Therefore the interesting typing rule for a cast expression $\langle T \rangle\, e$ is:

$$(\text{TCast})\ \frac{\Gamma \vdash e : T' \triangleleft U \qquad \vdash T <: T'}{\Gamma \vdash \langle T \rangle\, e : T \triangleleft U}$$

As usual in security languages with casts, cast errors are seen as a non-termination channel, hence not affecting the security definitions.

6 Related work

Information flow security in general, and declassification in particular, are very active areas of research. We now discuss the most salient proposals related to this work.

Secure information flow and type abstraction. Our work shows a connection between type abstraction and declassification policies for secure information flow. Previous works also attempt to connect type abstraction and secure information flow.

Tse and Zdancewic [31] encode the Dependency Core Calculus (DCC) [**?**] in System F. The correctness theorem of their translation aims at showing that the parametricity theorem of System F implies the noninterference property. Unfortunately, Shikuma and Igarashi identify a mistake in the proof of their main result [29]; they also gave a noninterference-preserving translation for a version of DCC to the simply-typed lambda calculus. However, this translation left open the connection between parametricity and noninterference, initially aimed by Tse and Zdancewic.

Recently, Bowman and Ahmed [**?**] provide a translation from DCC to System F_ω, successfully demonstrating that noninterference can be encoded via parametricity. Our work generalizes this by showing that type abstraction implies *relaxed* noninterference. Information flow analyses have been proposed to generalize parametricity in the presence of runtime type analysis [36]. Using security labels, a programmer can specify data structures that should remain confidential in order to hide implementation details and rely on type abstraction for abstract datatypes.

An interesting research direction is to investigate whether our proposal of solving information flow problems via type abstraction, here through subtyping, can be used to generalize parametricity as proposed by Washburn and Weirich [36].

Declassification. As extensively discussed, our policies and security property are based on the work of Li and Zdancewic [21], which proposes two kinds of downgrading policies (which we call here declassification policies, since they only relate to confidentiality): local and global policies. The declassification policies in this paper directly correspond to local policies, as discussed in the introduction. Global policies refer to declassifications that involve more than one secret simultaneously. As discussed in Section 5, it is unclear if and how global policies can be supported using our type-driven approach; further exploration is necessary to settle this issue. Additionally, in contrast to the definition of relaxed noninterference of Li and Zdancewic [21], our definition is independent from the security enforcement mechanism. This allows us to distinguish programs that are not secure from programs that are not typable due to a necessarily conservative static security mechanism (see Section 4). Also, our definition of relaxed noninterference is formulated as a generalization of the semantic characterization of pure noninterference [37], providing a modular reasoning principle, as opposed to the global translation approach of Li and Zdancewic.

In the following, we focus on the closest related work on declassification policies starting from 2005 and refer the reader to [28] for a survey prior to 2005.

Typing declassification in object-oriented languages. Since 2005, several works have studied static enforcement of declassification in object-oriented languages [9, 20, 10, 15].

Banerjee and Naumann [9] study the interaction between security typing for noninterference and access control in a Java-like language. Security levels are not fixed but rather depend on access permissions. In contrast to our work, security levels are independent of method signatures or types and thus their typing does not relate to type abstraction.

Hicks *et al.* [20] propose trusted declassification for an object calculus. Principals in a program have access to specified trusted *declassifier* functions or methods. Typeable programs are secure for noninterference modulo trusted methods, in the same spirit as typing of noninterference of programs with cryptographic functions [19]. In contrast to relaxed noninterference, trusted declassification does not consider declassifiers as part of security levels. Instead, declassifiers need to be associated by a policy to different principals (security labels in our setting) in the lattice.

Barthe *et al.* [10] propose a modular method to extend type systems and proofs for noninterference to declassification and discuss how the method extends to object-oriented languages. The declassification property called delimited non-disclosure [22] does not support fine-grained specification of how to declassify a given secret, as supported by relaxed noninterference.

Tse and Zdancewic [32] 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 [**?**]. The monadic style allows them to integrate computational effects, which we do not support. In contrast to our work, security labels are based on the Decentralized Label Model (DLM) [24], and are not semantically unified with the standard safety types of the language.

Chong and Myers [15] propose hybrid typing to enforce declassification and erasure policies and implement it in Jif [23]. Their language features a special declassification function that takes as input the expression and levels to declassify and also the conditions under which declassification can occur. Security policies are specified by means of security levels and conditions to downgrade them. This resembles our declassification policies, which specify the methods that can be applied in order to (partially) declassify; at a more abstract level, the interface types of the public facet can be seen as "conditions" for declassifying. The type system developed by Chong and Myers statically checks that conditions in declassification commands comply with the specified security policies. A dynamic mechanism enforces this, or returns a dummy value (instead of the declassified value) at runtime. In contrast to our work, their type system significantly departs from standard typing rules, and dynamic checks are required for guaranteeing security.

Extensional specification of declassification policies. The language Air [30] expresses declassification policies as security automata. The policies, seen as automata, transition when a release obligation is satisfied. When an accepting state is reached, declassification is performed. These policies resemble relaxed noninterference and our own declassification policies but they require very specific typing rules.

Banerjee et al. [**?**] study declassification properties using ideas from epistemic logic can capture global policies (as in the original work of relaxed noninterference) with an extensional

property. Their policies are not expressed using standard types as in our work.

The language Paralocks [14] supports declassification policies represented as Horn clauses, whose antecedents are conditions that should be satisfied for a flow to occur. There is a natural order between declassification policies that correspond to the logical entailment when viewing policies as Horn clauses. The policies together with the logical entailment order define a lattice that supports an extensional specification of secrets and their intended declassification, as in our work. However, declassification policies in Paralocks are not specified by using the standard types of the language, and thus their enforcement requires specific typing rules.

Multiple facets for dynamic enforcement of declassification Austin and Flanagan introduce Multiple Facets [**?**] as a dynamic mechanism to enforce secure information flow. The main idea behind multiple facets is to execute a program using multiple values, one value or facet for each security level of observation. A value considered confidential will only flow to a public facet by facet declassification, based on robust declassification [39]. Robust declassification requires the decision to declassify to be trusted according to integrity labels used to model trust. In our work, we do not consider integrity labels or robust declassification. However, the idea of multiple facets (having a facet for each observer at a given security level) is similar to our faceted types. Just as Austin and Flanagan can run a program for different facets simultaneously, we type check programs providing different views to observers with different security clearances.

Multiple facets are also inspired by Secure Multi Execution (SME) [18, 11], a dynamic mechanism that roughly executes a program multiple times in order to enforce noninterference. Hence, observers with different security clearances will potentially observe different values during the execution of a program. Several works have studied declassification in the context of SME [26, 33, 12]. Rafnsson and Sabelfeld [26] propose declassification in SME based on the gradual release property [7]. This property differs from the property we consider in our work in that it is not possible to extensionally specify what is being released or declassified. The latest works on SME declassification [33, 12] generalize security levels as declassifier functions, resembling declassification policies of both Li and Zdancewic and ours. Since SME is a dynamic enforcement mechanism, these declassification policies are not used for relating declassification and type abstraction.

7 Conclusion

One of the open challenges in the area of information flow security is integrating information flow mechanisms with existing infrastructures [38]. Our work partially addresses this challenge by showing a connection between type abstraction, more precisely that induced by the the subtyping relation in an object-oriented language, and the order relation in security lattices. In particular, we exploit an intuitive connection between object interfaces and declassification policies: an object interface already gives a way to control the exposed behavior of an object. These connections imply that standard type systems can be used as a direct means to enforce secure information flow, when types express security policies. It is left to explore how this connection scales in practice, but we expect the economy of concepts to be an important asset for adoption.

We plan to study the impact of more advanced typing disciplines on the expressiveness of type-based declassification, especially dependent object types [27] and refinement types [34]. It remains to be seen whether global policies can be expressed, and how. Another venue

for future work is to develop our approach in a setting that relies on other forms of type abstraction, such as existential types. Finally, we intend to explore how to *infer* the minimal knowledge that has to be exposed to a public observer in order to guarantee a relaxed noninterference guarantee at a given type. Inferring the minimal input declassifications of a secure program can for instance be useful to assess the impact some refactoring or extensions of that program have on security.

References

1 Martín Abadi, Anindya Banerjee, Nevin Heintze, and Jon G. Riecke. A core calculus of dependency. In *Proceedings of the 26th ACM Symposium on Principles of Programming Languages (POPL 99)*, pages 147–160, San Antonio, TX, USA, January 1999. ACM Press.

2 Martin Abadi and Luca Cardelli. *A Theory of Objects*. Springer-Verlag, 1996.

3 Amal Ahmed. *Semantics of Types for Mutable State*. PhD thesis, Princeton University, 2004.

4 Amal Ahmed. Step-indexed syntactic logical relations for recursive and quantified types. In Peter Sestoft, editor, *Proceedings of the 15th European Symposium on Programming (ESOP 2006)*, volume 3924 of *Lecture Notes in Computer Science*, pages 69–83. Springer-Verlag, 2006.

5 Roberto M. Amadio and Luca Cardelli. Subtyping recursive types. In David S. Wise, editor, *Conference Record of the Eighteenth Annual ACM Symposium on Principles of Programming Languages (POPL 91)*, pages 104–118. ACM Press, 1991.

6 Andrew W. Appel and David McAllester. An indexed model of recursive types for foundational proof-carrying code. *ACM Transactions on Programming Languages and Systems*, 23(5):657–683, September 2001.

7 Aslan Askarov and Andrei Sabelfeld. Gradual release: Unifying declassification, encryption and key release policies. In *Proceedings of the 27th IEEE Symposium on Security and Privacy (S&P 2007)*, pages 207–221. IEEE Computer Society Press, May 2007.

8 Thomas H. Austin and Cormac Flanagan. Multiple facets for dynamic information flow. In *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2012)*, pages 165–178. ACM Press, January 2012.

9 Anindya Banerjee and David A. Naumann. Stack-based access control and secure information flow. *Journal of Functional Programmming*, 15(2):131–177, September 2005.

10 Anindya Banerjee, David A. Naumann, and Stan Rosenberg. Expressive declassification policies and modular static enforcement. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P 2008)*, pages 339–353. IEEE Computer Society Press, May 2008.

11 Gilles Barthe, Salvador Cavadini, and Tamara Rezk. Tractable enforcement of declassification policies. In *Proceedings of the 21st IEEE Computer Security Foundations Symposium (CSF 2008)*, pages 83–97. IEEE Computer Society Press, June 2008.

12 Natalia Bielova and Tamara Rezk. Spot the difference: Secure multi-execution and multiple facets. In *Proceedings of the 21st European Symposium on Research in Computer Security (ESORICS 2016)*, pages 501– 519, 2016.

13 Iulia Bolosteanu and Deepak Garg. Asymmetric secure multi-execution with declassification. In *Proceedings of the 5th International Conference on Principles of Security and Trust (POST 2016)*, pages 24–45. Springer-Verlag, April 2016.

14 William J. Bowman and Amal Ahmed. Noninterference for free. In *Proceedings of the 20th ACM SIGPLAN Conference on Functional Programming (ICFP 2015)*, pages 101–113. ACM Press, August 2015.

15 Niklas Broberg and David Sands. Paralocks: role-based information flow control and beyond. In *Proceedings of the 37th annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2010)*, pages 431–444. ACM Press, January 2010.

16 Stephen Chong and Andrew C. Myers. End-to-end enforcement of erasure and declassification. In *Proceedings of the 21st IEEE Computer Security Foundations Symposium (CSF 2008)*, pages 98–111. IEEE Computer Society Press, June 2008.

17 William R. Cook. On understanding data abstraction, revisited. *ACM SIGPLAN Notices*, 44(10):557–572, 2009.

18 Dorothy E. Denning. A lattice model of secure information flow. *Communications of the ACM*, 19(5):236–243, May 1976.

19 Dominique Devriese and Frank Piessens. Noninterference through Secure Multi-execution. In *Proceedings of the 31st IEEE Symposium on Security and Privacy (S&P 2010)*, pages 109–124. IEEE Computer Society Press, May 2010.

20 Cédric Fournet, Jérémy Planul, and Tamara Rezk. Information-flow types for homomorphic encryptions. In *Proceedings of the Conference on Computer and Communications Security (CCS 2011)*, pages 351–360. ACM Press, October 2011.

21 Boniface Hicks, Dave King, Patrick McDaniel, and Michael Hicks. Trusted declassification: high-level policy for a security-typed language. In *Proceedings of the workshop on Programming Languages and Analysis for Security (PLAS 2006)*, pages 65–74. ACM Press, June 2006.

22 Peng Li and Steve Zdancewic. Downgrading policies and relaxed noninterference. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2005)*, pages 158–170. ACM Press, January 2005.

23 Ana Almeida Matos and Gérard Boudol. On declassification and the non-disclosure policy. In *Proceedings of the IEEE Computer Security Foundations Workshop (CSFW 2005)*, pages 549–597. IEEE Computer Society Press, October 2005.

24 Andrew C. Myers. Jif homepage. http://www.cs.cornell.edu/jif/, accessed May 2017.

25 Andrew C. Myers and Barbara Liskov. Protecting privacy using the decentralized label model. *ACM Transactions on Software Engineering and Methodology*, 9:410–442, October 2000.

26 Benjamin C. Pierce. *Types and programming languages*. MIT Press, Cambridge, MA, USA, 2002.

27 Willard Rafnsson and Andrei Sabelfeld. Secure multi-execution: Fine-grained, declassification-aware, and transparent. In *Proceedings of the 26th IEEE Computer Security Foundations Symposium (CSF 2013)*, pages 33–48. IEEE Computer Society Press, June 2013.

28 Tiark Rompf and Nada Amin. Type soundness for dependent object types (DOT). In Eelco Visser and Yannis Smaragdakis, editors, *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2016)*, pages 624–641. ACM Press, November 2016.

29 Andrei Sabelfeld and David Sands. Declassification: Dimensions and principles. *Journal of Computer Security*, 17(5):517–548, 2009.

30 Naokata Shikuma and Atsushi Igarashi. Proving noninterference by a fully complete translation to the simply typed lambda-calculus. In Mitsu Okada and Ichiro Satoh, editors, *Proceedings of the 11th Asian Computing Science Conference (ASIAN 2006)*, volume 4435 of *Lecture Notes in Computer Science*, pages 301–315. Springer-Verlag, 2006.

31 Nikhil Swamy and Michael Hicks. Verified enforcement of stateful information release policies. In Úlfar Erlingsson and Marco Pistoia, editors, *Proceedings of the Workshop on Programming Languages and Analysis for Security (PLAS 2008)*, pages 21–32. ACM Press, December 2008.

32 Stephen Tse and Steve Zdancewic. Translating dependency into parametricity. In *Proceedings of the 7th ACM SIGPLAN Conference on Functional Programming (ICFP 2004)*, pages 115–125, Snowbird, Utah, USA, September 2004. ACM Press.

33 Stephen Tse and Steve Zdancewic. A design for a security-typed language with certificate-based declassification. In *Proceedings of the 14th European Symposium on Programming Languages and Systems (ESOP 2005)*, volume 2986 of *Lecture Notes in Computer Science*, pages 279–294. Springer-Verlag, 2005.

34 Mathy Vanhoef, Willem De Groef, Dominique Devriese, Frank Piessens, and Tamara Rezk. Stateful declassification policies for event-driven programs. In *Proceedings of the 27th IEEE Computer Security Foundations Symposium (CSF 2014)*. IEEE Computer Society Press, 2014.

35 Niki Vazou, Eric L. Seidel, Ranjit Jhala, Dimitrios Vytiniotis, and Simon Peyton-Jones. Refinement types for haskell. In *Proceedings of the 19th ACM SIGPLAN Conference on Functional Programming (ICFP 2014)*, pages 269–282. ACM Press, August 2014.

36 Dennis Volpano, Cynthia Irvine, and Geoffrey Smith. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(2-3):167–187, January 1996.

37 Geoffrey Washburn and Stephanie Weirich. Generalizing parametricity using information-flow. In *Proceedings of the 20th IEEE Symposium on Logic in Computer Science (LICS 2005)*, pages 62–71. IEEE Computer Society Press, June 2005.

38 Steve Zdancewic. *Programming Languages for Information Security*. PhD thesis, Cornell University, August 2002.

39 Steve Zdancewic. Challenges for information-flow security. In *Proceedings of Programming Language Interference and Dependence*, 2004.

40 Steve Zdancewic and Andrew C. Myers. Robust declassification. In *Proceedings of the 14th IEEE Computer Security Foundations Workshop (CSFW-14)*, pages 15–23. IEEE Computer Society Press, June 2001.

A Auxiliary Definitions

A.1 Environments

$$
\begin{array}{lcll}
\Gamma & ::= & \cdot \mid \Gamma, x : S & \text{(type environment)} \\
\Phi & ::= & \cdot \mid \Phi, \alpha <: \beta & \text{(subtyping environment)} \\
\Delta & ::= & \cdot \mid \Delta, \alpha & \text{(type variable environment)} \\
\Sigma & ::= & \cdot \mid \Sigma, \alpha \triangleq O & \text{(type definition environment)}
\end{array}
$$

- Γ is a finite map from variables to closed and well-formed security types. Σ is a finite map from type variables to object types. Φ is a set of subtyping relations between type variables. Δ is a set of type variables.
- $dom(Env)$ (where Env could be Γ, Σ or Φ) is the set of variables for which the finite map Env is defined. In the case of $dom(\Phi)$, it is the set of the type variables in the left part of the subtyping relation.
- We also use the notations $\Gamma, x : S$ or $\Sigma, \alpha \triangleq O$ or $\Phi, \alpha <: \beta$ to extend the environments Γ, Σ , Φ with a new binding or relation, respectively. If $x \in dom(\Gamma)$, $\alpha \in dom(\Sigma)$ or either α or $\beta \in dom(\Phi) \cup cod(\Phi)$ the extension operation is not defined for the respective environment.
- The notation Δ, α extends the set Δ with a new type variable. If $\alpha \in \Delta$ the operation is not defined.

We use the following functions to access to the elements of the environments:

- $\Gamma(x)$ returns the security type associated to x in Γ. If $x \notin dom(\Gamma)$, then $\Gamma(x)$ is undefined.
- $\Sigma(\alpha)$ returns the type associated to α in Σ. If $\alpha \notin dom(\Sigma)$, then $\Sigma(\alpha)$ is undefined.
- $\alpha <: \beta \in \Phi$ is true if $\Phi(\alpha) = \beta$, false otherwise. $\Phi(\alpha)$ returns the type variable in the right part of the subtyping relation with α in Φ. If $\alpha \notin dom(\Phi)$, then $\Phi(\alpha)$ is undefined.

A.2 Well-formedness of types and environments

For the main results of the Section 4 to hold we need to ensure we work with well-formed security types.

Well formed types. We use the predicate $\mathsf{valid}(S)$ to denote that a security type S is closed and that the object types that S contains have unique method members. The definition of $\mathsf{valid}(S)$ is based on a standard notion well-formedness of object types [2] (Figure 7).

To check for *well-formed security types*, *i.e.* that the private type is a subtype of the public type we define the judgment $\Sigma \vdash_s S$ (Figure 8). The (WFS-ST) rule is the most important. For this rule to hold, the subtyping relation between both facets must hold and also the same principle must hold for the all the security types in each facet.

The presence of type variables in the facets of a security type and the corresponding subtyping constraint introduces subtle cases to manage before using the subtyping judgment. Consider the following object type: $O \triangleq \mathbf{Obj}(\alpha).\,[\mathsf{m} : S \to \alpha \triangleleft \mathbf{Obj}(\beta).\,[\mathsf{m} : S \to \alpha \triangleleft \beta]]$. For $\vdash_s O$ to hold, $\alpha \triangleq O \vdash_s \alpha \triangleleft \mathbf{Obj}(\beta).\,[\mathsf{m} : S \to \alpha \triangleleft \beta]$ must hold. It implies to check $\vdash \alpha <: \mathbf{Obj}(\beta).\,[\mathsf{m} : S \to \alpha \triangleleft \beta]$. Note that, we can not justify that subtyping judgment, because we do not have a subtyping premise involving the type variable α. To address this, we need to remember (in Σ) the surrounding recursive object type O that binds α, and to transform the check $\alpha \triangleq O \vdash_s \alpha \triangleleft \mathbf{Obj}(\beta).\,[\mathsf{m} : S \to \alpha \triangleleft \beta]$ to $\vdash O <: \mathbf{Obj}(\beta).\,[\mathsf{m} : S \to O \triangleleft \beta]$ by closing α with the mappings in Σ (*i.e.* O). We use the notation $\Sigma\,[T]$ to substitute the free variables in type T according to the bindings in Σ.

$\boxed{\Delta \vdash_t T}$

$$\text{(WF-V)}\frac{\alpha \in \Delta}{\Delta \vdash_t \alpha} \qquad \text{(WF-O)}\frac{T \equiv \mathbf{Obj}(\alpha).\ \boxed{m : S_1 \to S_2} \quad (i \neq j \implies m_i \neq m_j) \quad \Delta, \alpha \vdash_t S_{1i} \quad \Delta, \alpha \vdash_t S_{2i}}{\Delta \vdash_t T}$$

$\boxed{\Delta \vdash_t S}$

$$\text{(WF-ST)}\frac{\Delta \vdash_t T \quad \Delta \vdash_t U}{\Delta \vdash_t T \triangleleft U} \qquad \frac{\cdot \vdash_t S}{\mathsf{valid}(S)}$$

Figure 7 Standard well-formedness of object types and type variables, and its lifting to security types.

$\boxed{\Sigma \vdash_s T}$

$$\frac{T \equiv \mathbf{Obj}(\alpha).\ \boxed{m : S_1 \to S_2} \quad \Sigma, \alpha : T \vdash_s S_{1i} \quad \Sigma, \alpha : T \vdash_s S_{2i}}{\Sigma \vdash_s T} \qquad \text{(WFS-V)}\frac{}{\Sigma \vdash_s \alpha}$$

$\boxed{\Sigma \vdash_s S}$ $\qquad$ $\boxed{\vdash S}$

$$\text{(WFS-ST)}\frac{\Sigma \vdash_s T \quad \Sigma \vdash_s U \quad \cdot \vdash \Sigma\,[T] <: \Sigma\,[U]}{\Sigma \vdash_s T \triangleleft U} \qquad \text{(WF)}\frac{\mathsf{valid}(S) \quad \cdot \vdash_s S}{\vdash S}$$

Figure 8 Well-formedness of security types

Finally, we say that a security type S is well-formed (notation $\vdash S$) if the type is valid and the subtyping constraints for S hold ($\cdot \vdash_s S$)

Well-formedness of a type environment. A type environment is well formed, noted $\Gamma \vdash \diamond$, if all types in the environment are well-formed:

$$\text{(EEnvOk)}\frac{}{\cdot \vdash \diamond} \qquad \text{(EnvOk)}\frac{\Gamma \vdash \diamond \quad \vdash S \quad x \notin dom(\Gamma)}{\Gamma, x : S \vdash \diamond}$$

A.3 Subtyping

The gray parts in the subtyping rules of the Figure 9 were not included in the Figure 2 of the main document. They prevent justifying inconsistent subtyping judgments by controlling the uses of type variables.

For example, consider the following types:
$T_1 \triangleq \mathbf{Obj}(\alpha).\,[\mathsf{n} : S \to \mathbf{Obj}(\beta).\,[\mathsf{m}_1 : \beta_{\mathsf{L}} \to S' \quad \mathsf{m}_2 : S_1 \to S_2]_{\mathsf{L}}]$
$T_2 \triangleq \mathbf{Obj}(\beta).\,[\mathsf{n} : S \to \mathbf{Obj}(\alpha).\,[\mathsf{m}_1 : \alpha_{\mathsf{L}} \to S']_{\mathsf{L}}]$
For $\vdash T_1 <: T_2$ to hold, after using the rule (SObj) twice, the contravariance of m_1 parameters $\cdot, \alpha <: \beta, \beta <: \alpha \vdash \alpha <: \beta$ must hold. We can justify this by applying the rule (SVar) because we have the assumption $\alpha <: \beta$ in the subtyping environment. So, we justify $\vdash T_1 <: T_2$ and it is not the case that T_1 is subtype of T_2. The problem is the occurrence of the variables α and β in both types, that creates subtyping assumptions in both directions and it allows to justify subtyping between type variables that represent unrelated types (by subtyping). The well-formedness condition of the subtyping environment Φ prevents this kind of cases,

$\boxed{\Phi \vdash T <: T}$

$$(\text{SObj})\ \frac{\begin{array}{c} O_1 \triangleq \mathbf{Obj}(\alpha).\ \overline{[m : S_1 \to S_2]} \quad O_2 \triangleq \mathbf{Obj}(\beta).\ \overline{[m' : S'_1 \to S'_2]} \quad \overline{m'} \subseteq \overline{m} \\ m_i = m'_j \implies (\Phi, \alpha <: \beta \vdash S'_{1j} <: S_{1i} \quad \Phi, \alpha <: \beta \vdash S_{2i} <: S'_{2j}) \\ \Phi \vdash \diamond \quad dom(\Phi) \cup cod(\Phi) \vdash_t O_i \end{array}}{\Phi \vdash O_1 <: O_2}$$

$$(\text{SVar})\ \frac{\Phi \vdash \diamond \quad \alpha <: \beta \in \Phi}{\Phi \vdash \alpha <: \beta} \qquad (\text{SSubEq})\ \frac{T_1 \equiv T_2}{\Phi \vdash T_1 <: T_2} \qquad (\text{STrans})\ \frac{\Phi \vdash T_1 <: T_2 \quad \Phi \vdash T_2 <: T_3}{\Phi \vdash T_1 <: T_3}$$

$\boxed{\Phi \vdash S <: S}$

$$(\text{TSubST})\ \frac{\Phi \vdash T_1 <: T_2 \quad \Phi \vdash U_1 <: U_2}{\Phi \vdash T_1 \triangleleft U_1 <: T_2 \triangleleft U_2}$$

Figure 9 Subtyping

$\boxed{\Phi \vdash \diamond}$

$$(\text{EEnvSubOk})\ \frac{}{\cdot \vdash \diamond} \qquad (\text{EnvSubOk})\ \frac{\Phi \vdash \diamond \quad \alpha_i \notin dom(\Phi) \cup cod(\Phi)}{\Phi, \alpha_1 <: \alpha_2 \vdash \diamond}$$

Figure 10 Well-formedness of the subtyping environment

because we cannot extend the environment with a subtyping premise, where one of the involved variables is already in the environment (Figure 10).

A.4 Type equivalence

Two types are equivalent (Figure 11) if the equivalence can be derived through the congruence induced by rules (Alpha-Eq) and (Fold-Unfold). For example:
$\mathbf{Obj}(\alpha).\ [m : \alpha \to \alpha] \equiv \mathbf{Obj}(\beta).\ [m : \beta \to \beta]$
$\mathbf{Obj}(\alpha).\ [m : \top \to \alpha] \equiv \mathbf{Obj}(\alpha).\ [m : \top \to \mathbf{Obj}(\beta).\ [m : \top \to \beta]]$

A.5 Simple type system

The simple typing judgment $\Gamma \vdash_1 e : T$ is defined in terms of "single-facet typing" (Figure 12). Single-facet typing $\Gamma \vdash_{\mathsf{sf}} e : S$ is a simplification of security typing: the rules (TmD) and (TmH) are replaced by a single rule (T1mI) that simply ignores the public type. Furthermore, the subtyping judgment $\Phi \vdash S_1 <: S_2$ is replaced by the simple subtyping judgment $\Phi \vdash_{\mathsf{sf}} S_1 <: S_2$ that only takes care of subtyping between the private facets of the security types. Its definition is direct and omitted here.

▶ **Lemma 14.** $\Gamma \vdash \diamond \ \wedge\ \Gamma \vdash e : T \triangleleft U$ *then* $\Gamma \vdash_1 e : T$

Proof. Trivial induction on typing derivations of e. ◀

▶ **Lemma 15.**

$$\Gamma \vdash \diamond \ \wedge\ \Gamma \vdash_1 e : T \implies \exists U.\ \Gamma \vdash e : T \triangleleft U$$

$\boxed{T \equiv T}$

$$\text{(Sym)}\frac{}{T \equiv T} \qquad \text{(Refl)}\frac{T_1 \equiv T_2}{T_2 \equiv T_1} \qquad \text{(Trans)}\frac{T_1 \equiv T_2 \quad T_2 \equiv T_3}{T_1 \equiv T_3}$$

$$\text{(O-Congr)}\frac{S_{1i} \equiv S'_{1i} \quad S_{2i} \equiv S'_{2i}}{\mathbf{Obj}(\alpha).\,[\overline{m : S_1 \to S_2}] \equiv \mathbf{Obj}(\alpha).\,[\overline{m : S'_1 \to S'_2}]}$$

$$\text{(Alpha-Eq)}\frac{O \triangleq \mathbf{Obj}(\alpha).\,[\overline{m : S_1 \to S_2}] \quad \beta \text{ fresh}}{O \equiv O\,[\beta/\alpha]} \qquad \text{(Fold-Unfold)}\frac{}{O \equiv O\,[O/\alpha]}$$

$\boxed{S \equiv S}$

$$\frac{T_1 \equiv T_2 \quad U_1 \equiv U_2}{T_1 \triangleleft U_1 \equiv T_2 \triangleleft U_2}$$

Figure 11 Type equivalence

$\boxed{\Gamma \vdash_{\mathsf{sf}} e : S}$

$$\text{(T1Var)}\frac{x \in dom(\Gamma)}{\Gamma \vdash_{\mathsf{sf}} x : \Gamma(x)} \qquad \text{(T1Sub)}\frac{\Gamma \vdash_{\mathsf{sf}} e : S' \quad \vdash_{\mathsf{sf}} S' <: S \quad \vdash S}{\Gamma \vdash_{\mathsf{sf}} e : S}$$

$$\text{(T1Obj)}\frac{\vdash S \quad S \triangleq T \triangleleft U \quad \mathsf{methsig}(T, m_i) = S'_i \to S''_i \quad \Gamma, z : S, x_i : S'_i \vdash_{\mathsf{sf}} e_i : S''_i}{\Gamma \vdash_{\mathsf{sf}} \left[z : S \Rightarrow \overline{m\,(x)\,e}\right] : S}$$

$$\text{(T1mI)}\frac{\Gamma \vdash_{\mathsf{sf}} e_1 : T \triangleleft U \quad \mathsf{methsig}(T, m) = S_1 \to S_2 \quad \Gamma \vdash_{\mathsf{sf}} e_2 : S_1}{\Gamma \vdash_{\mathsf{sf}} e_1.m(e_2) : S_2}$$

$\boxed{\Gamma \vdash_1 e : T}$

$$\frac{\Gamma \vdash_{\mathsf{sf}} e : T \triangleleft U}{\Gamma \vdash_1 e : T}$$

Figure 12 Simple typing, defined in terms of single-facet typing

Proof. By induction of the typing derivation of $\Gamma \vdash_1 e : T$. In all the cases, we simply choose U to be the private type T. ◀