Polymorphic Relaxed Noninterference

Raimil Cruz
PLEIAD Lab, Computer Science Department (DCC)
University of Chile
Santiago, Chile
racruz@dcc.uchile.cl

Éric Tanter
PLEIAD Lab, Computer Science Department (DCC)
University of Chile
Santiago, Chile
etanter@dcc.uchile.cl

***Abstract*—Information-flow security typing statically preserves confidentiality by enforcing noninterference. To address the practical need of selective and flexible declassification of confidential information, several approaches have developed a notion of *relaxed* noninterference, where security labels are either functions or types. The labels-as-types approach to relaxed noninterference supports expressive declassification policies, including recursive ones, via a simple subtyping-based ordering, and provides a local, modular reasoning principle. In this work, we extend this expressive declassification approach in order to support *polymorphic* declassification. First, we identify the need for bounded polymorphism through concrete examples. We then formalize polymorphic relaxed noninterference in a typed object-oriented calculus, using a step-indexed logical relation to prove that all well-typed terms are secure. Finally, we address the case of primitive types, which requires a form of ad-hoc polymorphism. Therefore, this work addresses practical hurdles to providing controlled and expressive declassification for the construction of information-flow secure systems.**

1 Introduction

An information-flow security type system statically ensures that public outputs (*e.g.* $\mathsf{String}_\mathsf{L}$) cannot depend on secret inputs (*e.g.* $\mathsf{String}_\mathsf{H}$), a property known as noninterference (NI) [1]. NI provides a modular reasoning principle about security, indexed by the observational power of an adversary. For instance, a function $f : \mathsf{String}_\mathsf{H} \to \mathsf{String}_\mathsf{L}$ does not reveal any information about its argument; in fact, in a pure language, it is necessarily a constant function.

But noninterference is too strict in practice: for a system to be useful, confidential information sometimes needs to be *declassified*. Beyond introducing a declassification operator in the language, which compromises formal reasoning, various approaches have explored structured ways to support declassification policies [2, 3, 4, 5]. In particular, Li and Zdancewic [3] introduce *relaxed* noninterference, supporting expressive declassification policies via *security labels as functions*. In this approach, instead of having security labels such as H for private and L for public information that are drawn from a fixed lattice of symbols, security labels *are* the very functions that describe how a given secret can be manipulated in order to produce a public value: for instance, one can realize the declassification policy *"only the result of comparing the hash of the secret string s with a public guess can be made public"* by attaching to s a function that implements this declassification ($\lambda x.\lambda y.\ \mathsf{hash}(x) = y$). Any use of the secret that does not follow the declassification policy yields private results. One can express the standard label H (resp. L) as a constant function (resp. the identity function). A challenging aspect of this approach is that label ordering relies on a semantic interpretation of declassification functions.

A more practical approach than this *labels-as-functions* approach was recently developed by Cruz et al. [5] in an object-oriented setting, with a *labels-as-types* perspective: security types are *faceted types* of the form $T \triangleleft U$ where the first facet T—called the *safety type*—represents the implementation type, exposed to the private observer, and the second facet U—called the *declassification type*–represents the declassification policy as an object interface exposed to the public observer.[1] For instance, the type $\mathsf{String} \triangleleft \top$, where $\top$ is the empty object interface, denotes private values (no method is declassified) and the type $\mathsf{String} \triangleleft \mathsf{String}$ represents public values (all methods are declassified). These security types are abbreviated as $\mathsf{String}_\mathsf{H}$ and $\mathsf{String}_\mathsf{L}$, respectively. Interesting declassification policies stand in between these two extremes: for instance, given the interface $\mathsf{StringLen} \triangleq [\mathsf{length} : \mathsf{Unit}_\mathsf{L} \to \mathsf{Int}_\mathsf{L}]$, the faceted type $\mathsf{String} \triangleleft \mathsf{StringLen}$ exposes the method length to declassify the length of a string as a public integer, but not its content. This type-based approach to declassification is expressive as well as simple—in particular, because labels are types, label ordering is simply subtyping. Also, it extends the modular reasoning principle of NI to account for declassification [5], a property named *type-based relaxed noninterference* (TRNI). For instance, with TRNI one can prove that a function of type $\mathsf{String} \triangleleft \mathsf{StringLen} \to \mathsf{Bool}_\mathsf{L}$ *must* produce equal results for strings of equal lengths.

The labels-as-types approach of Cruz *et al.* however lacks *security label polymorphism*. Security label polymorphism is a very useful feature of practical security-typed languages such as JIF [6] and FlowCaml [7], which has only been explored in the context of standard security labels (symbols from a lattice). To the best of our knowledge, polymorphism has not been studied for expressive declassification mechanisms, such as labels-as-functions [3] or labels-as-types [5]. We extend the

This work is partially funded by CONICYT FONDECYT Regular Projects 1150017 and 1190058. Raimil Cruz is partially funded by CONICYT-PCHA/Doctorado Nacional/2014-63140148

[1] To account for $k > 2$ observation levels, faceted types can be extended to have k facets [5]. Here, we restrict the presentation to two observation levels.

978-1-5386-7289-1/19/$31.00 ©2019 IEEE
DOI 10.1109/SecDev.2019.00021

labels-as-types approach with declassification *polymorphism*, specifically *bounded* polymorphism that specifies both a lower and an upper bound for a polymorphic declassification type.

The main contribution of this paper is to develop the theory of bounded polymorphic declassification as an extension of TRNI, called *polymorphic relaxed noninterference* (PRNI for short). PRNI brings new benefits in the expressiveness and design of declassification interfaces. Additionally, we address the necessary support for primitive types, through a form of ad-hoc polymorphism.

The labels-as-types approach has the practical benefits of relying on concepts that are well-known to developers—object interfaces and subtyping—in order to build systems with information flow security that cleanly account for controlled and expressive declassification. This work addresses the two major shortcomings of prior work in order to bring this approach closer to real-world secure programming.

Section 2 provides background on labels-as-types and TRNI. Section 3 then explains the main aspects of polymorphic relaxed noninterference (PRNI). Section 4 formalizes polymorphic declassification in a core object-oriented language $\mathsf{Ob}^{\langle\rangle}_{\mathsf{SEC}}$. Then, Section 5 develops a logical relation for PRNI and shows that all well-typed $\mathsf{Ob}^{\langle\rangle}_{\mathsf{SEC}}$ terms satisfy PRNI. Section 6 extends $\mathsf{Ob}^{\langle\rangle}_{\mathsf{SEC}}$ with primitive types. Section 7 discusses related work and Section 8 concludes. We have implemented an interactive prototype of $\mathsf{Ob}^{\langle\rangle}_{\mathsf{SEC}}$ that is available at https://pleiad.cl/gobsec/.

2 Background:Type-Based Relaxed Noninterference

We start with a quick review of type-based relaxed noninterference [5]. Faceted security types allow programmers to express declassification policies as type interfaces. For instance, one can express that a login function can reveal the result of comparing a secret password for equality with a public guess.

```
String_L login(String_L guess, String ◁ StringEq password){
 if(password.eq(guess))
    return "Login Successful"
  else
     return "Login failed"
}
```

Note that leaking the secret password by directly returning it would not typecheck, since $\mathsf{StringEq} \triangleq [\mathsf{eq} : \mathsf{String}_\mathsf{L} \to \mathsf{Bool}_\mathsf{L}]$ is not a subtype of String (recall that $\mathsf{String}_\mathsf{L}$ is short for $\mathsf{String} \triangleleft \mathsf{String}$). Taking advantage of the fact that object types are recursive, one can also express *recursive declassification*, for instance that a list of secret strings can only be declassified by comparing its elements for equality. Likewise, one can express *progressive declassification* by nesting type interfaces. For instance, assuming that String has a method $\mathsf{hash} : \mathsf{Unit}_\mathsf{L} \to \mathsf{Int}_\mathsf{L}$, we can specify that only the hash of the password can be compared for equality with the interface type $\mathsf{StringHashEq} \triangleq [\mathsf{hash} : \mathsf{Unit}_\mathsf{L} \to \mathsf{Int} \triangleleft \mathsf{IntEq}]$, where $\mathsf{IntEq} \triangleq [\mathsf{eq} : \mathsf{Int}_\mathsf{L} \to \mathsf{Bool}_\mathsf{L}]$:

```
String_L login(Int_L guess, String ◁ StringHashEq password){
  if(password.hash().eq(guess)) ...
}
```

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

$\boxed{\Gamma \vdash e : S}$ $\qquad \Gamma ::= \bullet \mid \Gamma, x : S$ (type environment)

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

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

Fig. 1. $\mathsf{Ob}_{\mathsf{SEC}}$: Syntax and Static semantics (excerpts from [5])

Cruz *et al.* formalize faceted security types in $\mathsf{Ob}_{\mathsf{SEC}}$, a core object-oriented language with three kinds of expressions: variables, objects and method invocations (Figure 1). An object $\left[z : S \Rightarrow \overline{m\,(x)\,e}\right]$ is a collection of methods that can refer to the defining object with the self variable z. An object type $\mathbf{Obj}(\alpha).\ \left[\overline{m : S \to S}\right]$ is a collection of method signatures that have access to the defining type through the self type variable α. Security types $S = T \triangleleft U$ are composed of two object types T and U. Note that to be well-formed, a security type $T \triangleleft U$ requires U to be a *supertype* of T. The type abstraction mechanism of subtyping (by which a supertype "hides" members of its subtypes) is the key element to express declassification.

The $\mathsf{Ob}_{\mathsf{SEC}}$ type system defines two rules to give a type to a method invocation depending on whether the invoked method is in the declassification type or not. Rule (TmD) specifies that if the invoked method m is in the declassification type U with type $S_1 \to S_2$, then the result type of the method invocation expression is S_2. Conversely, if the method m is only present in the safety type T with type $S_1 \to T_2 \triangleleft U_2$, then the result type of the method invocation is $T_2 \triangleleft \top$ (TmH): if we bypass the declassification type, the result must be protected as a secret.

The security property obtained by this approach is called Type-based Relaxed Noninterference (TRNI). At the core of TRNI is a notion of observational equivalence between objects *up to the discrimination power of the public observer*, which is specified by the declassification type. More precisely, two objects o_1 and o_2 are equivalent at type $T \triangleleft U$ if, for any method m with type $S_1 \to S_2$ in the declassification type U, invoking m with equivalent values v_1 and v_2 at type S_1, produces equivalent results at type S_2.

TRNI is formulated as a *modular* reasoning principle, over open terms: $\mathsf{TRNI}(\Gamma, e, S)$. The closing typing environment

Γ specifies the secrecy of the inputs that e can use, and the security type S specifies the observation power of the adversary on the output.

For instance, suppose StringLen is an interface that exposes a length : $\mathsf{Unit_L} \to \mathsf{Int_L}$ method. Then, with $\Gamma = x : \mathsf{String} \triangleleft \mathsf{StringLen}$, the judgment $\mathsf{TRNI}(\Gamma, \mathsf{x.length}(), \mathsf{Int_L})$ implies: given the knowledge that two input strings v_1 and v_2 have the same length, the lower observer does not learn anything new about the inputs by executing x.length(). Conversely, $\mathsf{TRNI}(\Gamma, \mathsf{x.eq("a")}, \mathsf{Bool_L})$ does *not* hold: executing x.eq("a") and exposing the result as a public value would reveal more information than permitted by the input declassification type (eq $\notin$ StringLen). However, $\mathsf{TRNI}(\Gamma, \mathsf{x.eq("a")}, \mathsf{Bool_H})$ *does* hold, because the result is private and therefore unaccessible to the public observer.

3 Polymorphic Relaxed Noninterference

We first motivate polymorphic declassification with faceted types, and then we illustrate the role of bounded polymorphism for declassification. Finally, we give an overview of the modular reasoning principle of polymorphic relaxed noninterference.

3.1 Polymorphic Declassification

When informally discussing the possible extensions to their approach to declassification, Cruz et al. [5] illustrate the potential benefits of polymorphic declassification by giving the example of a list of strings that is polymorphic with respect to the declassification type of its elements:

$$\begin{aligned}\mathsf{ListStr}\langle X\rangle \triangleq [\ & \mathsf{isEmpty} : \mathsf{Unit_L} \to \mathsf{Bool_L},\\ & \mathsf{head} : \mathsf{Unit_L} \to \mathsf{String} \triangleleft X,\\ & \mathsf{tail} : \mathsf{Unit_L} \to \mathsf{ListStr}\,\langle X\rangle_{\mathsf{L}}]\end{aligned}$$

This recursive polymorphic declassification policy allows a public observer to traverse the list, and to observe *up to* X on each of the elements. This restriction is visible in the signature of the head method, which returns a value of type $\mathsf{String} \triangleleft X$.

Then, with polymorphic declassification we can implement data structures that are agnostic to the declassification policies of their elements, as well as polymorphic methods over these data structures. For example, we can construct declassification-polymorphic lists of strings with the following cons method:

```
ListStr ⟨X⟩_L cons<X>(String ◁ X s, ListStr ⟨X⟩_L l){
  return new {
           self: ListStr ⟨X⟩_L
           isEmpty() => false
           head() => s
           tail() => l
         }
}
```

The cons method does not even access any method of list l, it simply returns a new declassification-polymorphic list of strings as a new object with the expected methods. We can then use this method to define a declassification-polymorphic list concatenation method concat: $\mathsf{ListStr}\,\langle X\rangle_{\mathsf{L}} \times \mathsf{ListStr}\,\langle X\rangle_{\mathsf{L}} \to \mathsf{ListStr}\,\langle X\rangle_{\mathsf{L}}$ defined below:

```
ListStr ⟨X⟩_L concat<X>(ListStr ⟨X⟩_L l1, ListStr ⟨X⟩_L l2){
  if(l1.isEmpty()) return l2
  return cons<X>(l1.head(),
                 concat<X>(l1.tail(),l2))
}
```

The concat and cons methods are standard object-oriented implementations of list concatenation and construction, respectively. The concat method respects the declassification type ListStr $\langle X\rangle$ of both lists because it uses l1.isEmpty() and l1.tail() to iterate over l1, and it uses l1.head() to create a new declassification-polymorphic list of type $\mathsf{ListStr}\,\langle X\rangle_{\mathsf{L}}$. In particular, it uses no string-specific methods.

3.2 Bounded Polymorphic Declassification

The declassification interface ListStr $\langle X\rangle$ above is fully polymorphic, in that a public observer cannot exploit *a priori* any information about the elements of the list. In particular, it is not possible to implement a polymorphic contains method that would yield publicly observable results. Indeed, contains needs to invoke eq over the elements of the list (obtained with head). Because the result of head has declassification type X, for *any* X, the results of equality comparisons are necessarily private.

In order to support polymorphic declassification more flexibly, we turn to *bounded* parametric polymorphism. Bounded parametric polymorphism supports the specification of both upper and lower bounds on type variables. The type ListStr $\langle X\rangle$ is therefore equivalent to ListStr $\langle X : \mathsf{String}..\top\rangle$, where the notation $X : A..B$ is used to denote that X is a type variable that ranges between A and B). Note that for ListStr to be well-formed, the declassification type variable X must at least be a supertype of the safety type String.

Going back to declassification-polymorphic lists, if we want to allow the definition of methods like contains, we can further constrain the type variable X to be a subtype of StringEq:

$$\begin{aligned}&\mathsf{ListEqStr}\langle X : \mathsf{String}..\mathsf{StringEq}\rangle \triangleq\\ &\quad [\ \cdots, \mathsf{tail} : \mathsf{Unit_L} \to \mathsf{ListEqStr}\,\langle X\rangle_{\mathsf{L}}]\end{aligned}$$

The type ListEqStr denotes a recursive polymorphic declassification policy that allows a public observer to traverse the list and compare its elements for equality with a given public element. With this policy we can implement a generic contains function with publicly observable result:

```
Bool_L contains<X : String..StringEq>
            (ListEqStr ⟨X⟩_L l, String_L s){
  if(l.isEmpty()) return false
  if(l.head().eq(s)) return true
  return contains(l.tail(),s)
}
```

The key here is that l.head().eq(s) is guaranteed to be publicly observable, because the actual declassification policy with which X will be instantiated necessarily includes (at least) the eq method. Thus, upper bounds on declassification variables are useful for supporting polymorphic *clients*.

As mentioned above, the lower bound of a type variable used for declassification must at least be the safety type for well-formedness. More interestingly, the lower bound plays

a critical (dual) role for *implementors* of declassification-polymorphic functions. Consider a method with signature

$$\langle X : \mathsf{String}..\top \rangle\ \mathsf{String} \triangleleft \mathsf{StringLen} \to \mathsf{String} \triangleleft X$$

Can this method return non-public values? For instance, can it be the identity function? No, because returning a string of type $\mathsf{String} \triangleleft \mathsf{StringLen}$ would be unsound. Indeed, a client could instantiate X with String, yielding

$$\mathsf{String} \triangleleft \mathsf{StringLen} \to \mathsf{String} \triangleleft \mathsf{String}$$

Therefore, to be sound for all possible instantiations of X, the implementor of the method has no choice but to return a public string.

To recover flexibility and allow a polymorphic implementation to return non-public values, we can constrain the lower bound of X. For instance

$$\langle X : \mathsf{StringLen}..\top \rangle\ \mathsf{String} \triangleleft \mathsf{StringLen} \to \mathsf{String} \triangleleft X$$

admits the identity function as an implementation, in addition to other implementations that produce public results. Returned values cannot be *more private* than specified by the lower bound of X; their type must be a subtype of the lower bound.

Having illustrated the interest of upper and lower bounds of declassification type variables in isolation, we now present an example that combines both. Consider two lists of strings, each with one of the following declassification policies:

$$\begin{aligned}
\mathsf{ListStrLen}\ \langle X : \mathsf{String}..\mathsf{StringLen} \rangle \triangleq\ & [\ \mathsf{isEmpty} : \mathsf{Unit_L} \to \mathsf{Bool_L},\\
& \mathsf{head} : \mathsf{Unit_L} \to \mathsf{String} \triangleleft X,\\
& \mathsf{tail} : \mathsf{Unit_L} \to \mathsf{ListStrLen_L}]\\
\mathsf{ListStrFstLen} \triangleq\ & [\ \mathsf{isEmpty} : \mathsf{Unit_L} \to \mathsf{Bool_L},\\
& \mathsf{head} : \mathsf{Unit_L} \to \mathsf{String} \triangleleft \mathsf{StrFstLen},\\
& \mathsf{tail} : \mathsf{Unit_L} \to \mathsf{ListStrFstLen_L}]
\end{aligned}$$

ListStrLen is declassification polymorphic, ensuring that at least the length of its elements is declassified (X has upper bound StringLen). The second policy, ListStrFstLen, is monomorphic: it declassifies both the first character and the length of its elements. If we want a function able to concatenate these two string lists, its most general polymorphic signature ought to be:

$$\begin{aligned}
&\langle X : \mathsf{StrFstLen}..\mathsf{StringLen} \rangle\\
&\quad \mathsf{ListStrLen}\ \langle X \rangle_{\mathsf{L}} \times \mathsf{ListStrFstLen_L} \to \mathsf{ListStrLen}\ \langle X \rangle_{\mathsf{L}}
\end{aligned}$$

The upper bound StringLen is required to have a valid instantiation of $\mathsf{ListStrLen}\ \langle X \rangle$; the lower bound StrFstLen is required to be able to add elements of the second list to the returned list.

3.3 Reasoning principles for PRNI

Introducing polymorphism in declassification types yields an extended notion of type-based relaxed noninterference called *polymorphic relaxed noninterference* (PRNI). PRNI exactly characterizes that a program with polymorphic types must be secure for any instantiation of its type variables. To account for type variables, the judgment $\mathsf{PRNI}(\Delta, \Gamma, e, S)$ is parametrized by Δ, a set of bounded type variables (*i.e.* $\Delta ::= \cdot \mid \Delta, X : A..B$). As in $\mathsf{TRNI}(\Gamma, e, S)$, the closing typing environment Γ specifies the secrecy of the inputs that e can use, and S specifies the observation level for the output. Δ gives meaning to the type variables that can occur in both S and Γ: e is secure for *any* instantiation of type variables that respects the bounds.

For instance, given $\Delta \triangleq X : \mathsf{StrFstLen}..\mathsf{StringLen}$ and $\Gamma \triangleq x : \mathsf{String} \triangleleft X$, the judgment $\mathsf{PRNI}(\Delta, \Gamma, \mathsf{x.length}(), \mathsf{Int_L})$ holds because for any type T such that $\mathsf{StrFstLen} <: T <: \mathsf{StringLen}$, and the knowledge that two input strings are related at $\mathsf{String} \triangleleft T$, and hence at $\mathsf{String} \triangleleft \mathsf{StringLen}$ (*i.e.* both strings have the same length), the public observer does not learn anything new by executing $\mathsf{x.length}()$. However, $\mathsf{PRNI}(\Delta, \Gamma, \mathsf{x.first}(), \mathsf{String_L})$ does not hold. If we substitute X by StringLen, given two strings with the same length "abc" and "123", the public observer is able to distinguish them by executing "abc".first() and "123".first() and observing the results "a" and "1" as public values.

Also, $\mathsf{PRNI}(\Delta, \Gamma, \mathsf{x}, \mathsf{String} \triangleleft \mathsf{StringFst})$ does not hold. Again, we can substitute X by StringLen, and take input strings "abc" and "123", which can be discriminated by the public observer at type $\mathsf{String} \triangleleft \mathsf{StringFst}$. However, $\mathsf{PRNI}(\Delta, \Gamma, \mathsf{x}, \mathsf{String} \triangleleft \mathsf{StringLen})$ does hold: any two equivalent values at $\mathsf{String} \triangleleft T$ where $\mathsf{StrFstLen} <: T <: \mathsf{StringLen}$ have at least the same length.

The rest of this paper dives into the formalization of polymorphic relaxed noninterference in a pure object-oriented setting (Sections 4 and 5), before discussing the necessary extensions to accommodate primitive types (Section 6).

4 Formal Semantics

We model polymorphic type-based declassification in $\mathsf{Ob}^{\langle\rangle}_{\mathsf{SEC}}$, an extension of the language $\mathsf{Ob_{SEC}}$ [5] with polymorphic declassification. $\mathsf{Ob_{SEC}}$ is based on the object calculi of Abadi and Cardelli [8], and our treatment of type variables and bounded polymorphism is inspired by Featherweight Java [9] and DOT [10].

4.1 Syntax

Figure 2 presents the syntax of $\mathsf{Ob}^{\langle\rangle}_{\mathsf{SEC}}$. We highlight the extension for polymorphic declassification, compared to the syntax of $\mathsf{Ob_{SEC}}$.

The language has three kind of expressions: objects, method invocations and variables. Objects $\left[z : S \Rightarrow \overline{m\,(x)\,e}\right]$ are collections of method definitions. Recall that the self variable z binds the current object.

A security type S is a faceted type $T \triangleleft U$, where T is called the *safety type* of S, and U is called the *declassification type* of S. Types T include object types O and self type variables α. Declassification types U additionally feature type variables X, to express polymorphic declassification. We use metavariables A and B for declassification type bounds.

An object type $\mathbf{Obj}(\alpha).\ \overline{[m : M]}$ is a collection of method signatures with unique names (we sometimes use R to refer

$$
\begin{array}{rll}
e & ::= v \mid e.m \left\langle \; U \; \right\rangle (e) \mid x & \text{(terms)} \\
v & ::= o & \text{(values)} \\
o & ::= \left[z : S \Rightarrow \overline{m\,(x)\,e} \right] & \text{(objects)} \\
S & ::= T \triangleleft U & \text{(security types)} \\
T & ::= O \mid \alpha & \text{(types)} \\
U, A, B & ::= T \mid \; X \; & \text{(declassification types)} \\
O & ::= \mathbf{Obj}(\alpha).R & \text{(object types)} \\
R & ::= \left[\overline{m : M} \right] & \text{(record types)} \\
M & ::= \left\langle \; X : A..B \; \right\rangle S \to S & \text{(method signatures)} \\
\Gamma & ::= \bullet \mid \Gamma, x : S & \text{(type environments)} \\
\Phi & ::= \bullet \mid \Phi, \alpha <: \beta & \text{(subtyping environments)} \\
\Delta & ::= \bullet \mid \Delta, X : A..B & \text{(type variable environments)} \\
\alpha, \beta & & \text{(self type variables)}
\end{array}
$$

Fig. 2. $\mathsf{Ob}^{\langle\rangle}_{\mathsf{SEC}}$: Syntax

to just a collection of methods). The self type variable α binds to the defined object type (*i.e.* object types are recursive types). A method signature $\langle X : A..B \rangle\, S_1 \to S_2$ introduces the type variable X with lower bound A and upper bound B. To simplify the presentation of the calculus, we model single-argument methods with a single type variable.[2]

4.2 Subtyping

Figure 3 presents the $\mathsf{Ob}^{\langle\rangle}_{\mathsf{SEC}}$ subtyping judgment $\Delta; \Phi \vdash U_1 <: U_2$. The type variable environment Δ is a set of type variables with their bounds, *i.e.* $\Delta ::= \bullet \mid \Delta, X : A..B$. The subtyping environment Φ is a set of subtyping assumptions between self type variables, *i.e.* $\Phi ::= \bullet \mid \Phi, \alpha <: \beta$

The rules for the monomorphic part of the language are similar to $\mathsf{Ob}_{\mathsf{SEC}}$. Rule (SObj) justifies subtyping between two object types; it holds if the methods of the left object type O_1 are subtypes of the corresponding methods on O_2. Both width and depth subtyping are supported. Note that to verify subtyping of method collections, *i.e.* $\Delta; \Phi, \alpha <: \beta \vdash R_1 <: R_2$, we put in subtyping relation in Φ the self variables α and β. Rule (SVar) accounts for subtyping between self type variables and it holds if such subtyping relation exists in the subtyping environment.

The rules (STrans) and (SSubEq) justify subtyping by transitivity and type equivalence respectively. We consider type equivalence up to renaming and folding/unfolding of self type variables [5].

The novel part of $\mathsf{Ob}^{\langle\rangle}_{\mathsf{SEC}}$ are type variables, handled by rules (SGVar1) and (SGVar2). We follow the approach of Rompf and Amin [10]. Rule (SGVar1) justifies subtyping between a type variable X and a type B, if B is the upper bound of the type variable in Δ. Rule (SGVar2) is dual to (SGVar1), justifying that $A <: X$ if A is the lower bound of X.

[2]The implementation supports both multiple arguments and multiple type variables.

$\boxed{\Delta; \Phi \vdash U_1 <: U_2}$

$$\text{(SObj)}\ \frac{O_1 \triangleq \mathbf{Obj}(\alpha).R_1 \quad O_2 \triangleq \mathbf{Obj}(\beta).R_2 \quad \Delta; \Phi, \alpha <: \beta \vdash R_1 <: R_2}{\Delta; \Phi \vdash O_1 <: O_2}$$

$$\text{(SVar)}\ \frac{\alpha <: \beta \in \Phi}{\Delta; \Phi \vdash \alpha <: \beta} \qquad \text{(SSubEq)}\ \frac{O_1 \equiv O_2}{\Delta_X; \Phi \vdash O_1 <: O_2}$$

$$\text{(SGVar1)}\ \frac{X : A..B \in \Delta}{\Delta; \Phi \vdash X <: B} \qquad \text{(SGVar2)}\ \frac{X : A..B \in \Delta}{\Delta; \Phi \vdash A <: X}$$

$$\text{(STrans)}\ \frac{\Delta; \Phi \vdash U_1 <: U_2 \quad \Delta; \Phi \vdash U_2 <: U_3}{\Delta; \Phi \vdash U_1 <: U_3}$$

$\boxed{\Delta; \Phi \vdash R_1 <: R_2}$

$$\text{(SR)}\ \frac{\overline{m'} \subseteq \overline{m} \quad m_i = m'_j \implies \Delta; \Phi \vdash M <: M'}{\Delta; \Phi \vdash \left[\overline{m : M} \right] <: \left[\overline{m' : M'} \right]}$$

$\boxed{\Delta; \Phi \vdash M_1 <: M_2}$

$$\text{(SM)}\ \frac{\begin{array}{c}\Delta; \Phi \vdash B' <: B \quad \Delta; \Phi \vdash A <: A' \\ \Delta, \; X : A'..B' \; ; \Phi \vdash S'_1 <: S_1 \\ \Delta, \; X : A'..B' \; ; \Phi \vdash S_2 <: S'_2\end{array}}{\Delta; \Phi \vdash \langle X : A..B \rangle\, S_1 \to S_2 <: \langle X : A'..B' \rangle\, S'_1 \to S'_2}$$

$\boxed{\Delta; \Phi \vdash S_1 <: S_2}$

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

Fig. 3. $\mathsf{Ob}^{\langle\rangle}_{\mathsf{SEC}}$: Subtyping rules

The judgment $\Delta; \Phi \vdash R_1 <: R_2$ accounts for subtyping between collections of methods, and is used in rule (SObj). The judgment $\Delta; \Phi \vdash M_1 <: M_2$ denotes subtyping between method signatures. For this judgment to hold, the type variable bounds $A'..B'$ of the supertype (on the right) must be included within the bounds $A..B$ of the subtype (on the left); this ensures that any instantiation on the right is valid on the left. Then, in a type variable environment extended with $X : A'..B'$, standard function subtyping must hold (contravariant on the argument type, covariant on the return type).

Finally, rule (SST) accounts for subtyping between security types, which requires facets to be pointwise subtypes.

4.3 Type System

The typing rules of $\mathsf{Ob}^{\langle\rangle}_{\mathsf{SEC}}$ appeal to some auxiliary definitions, given in Figure 4. Function $\mathsf{ub}(\Delta, U)$ returns the upper bound of a type U in the type variable environment Δ. Since $\mathsf{Ob}^{\langle\rangle}_{\mathsf{SEC}}$ has a top type ($\mathbf{Obj}(\alpha).[\,]$) this recursive definition of ub is well-founded; as in Featherweight Java [9], we assume that Δ does not contain cycles. The auxiliary judgment $\Delta \vdash m \in U$ holds if method m belongs to type U. For a type variable, this means that the method is in the upper bound $\mathsf{ub}(\Delta, X)$. Function $\mathsf{msig}(\Delta, U, m)$ returns the polymorphic method signature of method m in type U. The rule for type variables

$\boxed{\mathsf{ub}(\Delta, U) = T}$

$$\frac{T \neq X}{\mathsf{ub}(\Delta, T) = T} \qquad \frac{X : A..B \in \Delta}{\mathsf{ub}(\Delta, X) = \mathsf{ub}(\Delta, B)}$$

$\boxed{\Delta \vdash m \in U}$

$$\frac{O \triangleq \mathbf{Obj}(\alpha).\ \overline{[m : M]}}{\Delta \vdash m_i \in O} \qquad \frac{\Delta \vdash m \in \mathsf{ub}(\Delta, X)}{\Delta \vdash m \in X}$$

$\boxed{\mathsf{msig}(\Delta, U, m) = M}$

$$\frac{}{\mathsf{msig}(\Delta, X, m) = \mathsf{msig}(\_, \mathsf{ub}(\Delta, X), m)}$$

$$\frac{O \triangleq \mathbf{Obj}(\alpha).\ \overline{[m : M]}}{\mathsf{msig}(\_, O, m_i) = M\,[O/\alpha]}$$

$\boxed{\Delta \vdash U \in A..B}$

$$\frac{\Delta; \bullet \vdash A <: U \quad \Delta; \bullet \vdash U <: B}{\Delta \vdash U \in A..B}$$

Fig. 4. $\mathsf{Ob}^{\langle\rangle}_{\mathsf{SEC}}$: Some auxiliary definitions

looks up the signature in the upper bound. The rule for object types is standard; note that it returns closed type signatures with respect to the self type variable. Finally, the judgment $\Delta \vdash U \in A..B$ holds if the type U is a super type of A and a subtype of B in the type variable environment Δ.

Figure 5 presents the typing judgment $\Delta; \Gamma \vdash e : S$ for $\mathsf{Ob}^{\langle\rangle}_{\mathsf{SEC}}$, which denotes that "expression e has type S under type variable environment Δ and type environment Γ". We omit the well-formedness rules for types and environments; they are standard, except that they also verify the subtyping relation between the facets of a security type.

The first three typing rules are standard: rule (TVar) types a variable according to the environment, rule (TSub) is the subsumption rule and rule (TObj) types an object. The method definitions of the object must be well-typed with respect to the method signatures taken from the safety type T of the security type S ascribed to the self variable z. For this, the method body e_i must be well-typed in an extended type variable environment with the type variable $\Delta, X : A_i..B_i$, and an extended type environment with the self variable and the method argument.

Rules (TmD) and (TmH) cover method invocation, and account for declassification. The actual argument type U' must satisfy the variable bounds $\Delta \vdash U' \in A..B$. On the one hand, rule (TmD) applies when the method m is in U with signature $\langle X : A..B \rangle\, S_1 \to S_2$; this corresponds to a use of the object at its declassification interface. Then, the method invocation has type S_2 substituting U' for X. On the other hand, rule (TmH) applies when m is not in U, but it is in T; this corresponds to a use beyond declassification and should raise the security to high. This is why the result type is $T_2\,[X/U'] \triangleleft \top$. This is all similar to the non-polymorphic rules (Figure 1), save for

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

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

$$\text{(TObj)}\ \frac{\begin{array}{c} S \triangleq T \triangleleft U \quad \mathsf{msig}(\_, T, m_i) = \langle X : A_i..B_i \rangle\, S_i' \to S_i'' \\ \Delta, X : A_i..B_i; \Gamma, z : S, x : S_i' \vdash e_i : S_i'' \end{array}}{\Delta; \Gamma \vdash \left[z : S \Rightarrow \overline{m\,(x)\,e}\right] : S}$$

$$\text{(TmD)}\ \frac{\begin{array}{c} \Delta; \Gamma \vdash e_1 : T \triangleleft U \qquad \Delta \vdash m \in U \\ \mathsf{msig}(\Delta; U, m) = \langle X : A..B \rangle\, S_1 \to S_2 \\ \Delta \vdash U' \in A..B \qquad \Delta; \Gamma \vdash e_2 : S_1\,[U'/X] \end{array}}{\Delta; \Gamma \vdash e_1.m\,\langle U' \rangle\,(e_2) : S_2\,[U'/X]}$$

$$\text{(TmH)}\ \frac{\begin{array}{c} \Delta; \Gamma \vdash e_1 : T \triangleleft U \qquad \Delta \vdash m \notin U \\ \mathsf{msig}(\Delta; T, m) = \langle X : A..B \rangle\, S_1 \to T_2 \triangleleft U_2 \\ \Delta \vdash U' \in A..B \qquad \Delta; \Gamma \vdash e_2 : S_1\,[U'/X] \end{array}}{\Delta; \Gamma \vdash e_1.m\,\langle U' \rangle\,(e_2) : T_2\,[U'/X] \triangleleft \top}$$

Fig. 5. $\mathsf{Ob}^{\langle\rangle}_{\mathsf{SEC}}$: Static semantics

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

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

$E ::= [\,] \mid E.m(e) \mid v.m(E)$ (evaluation contexts)

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

Fig. 6. $\mathsf{Ob}^{\langle\rangle}_{\mathsf{SEC}}$: Dynamic semantics

the type bounds check, and the type-level substitution.

4.4 Dynamic Semantics

The small-step dynamic semantics of $\mathsf{Ob}^{\langle\rangle}_{\mathsf{SEC}}$ are standard, given in Figure 6. They rely on evaluation contexts and use the auxiliary function $\mathsf{methimpl}(o.m)$ to lookup a method implementation. Note that types in general, and type variables in particular, do not play any role at runtime.

4.5 Safety

We first define what it means for a closed expression e to be *safe*: an expression is safe if it evaluates to a value, or diverges without getting stuck.

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

Well-typed $\mathsf{Ob}^{\langle\rangle}_{\mathsf{SEC}}$ closed terms are safe.

Theorem 1 (Syntactic type safety). $\vdash e : S \implies \mathsf{safe}(e)$

But of course, type safety is far from sufficient; we want to make sure that well-typed $\mathsf{Ob}^{\langle\rangle}_{\mathsf{SEC}}$ terms are *secure*. To this end, the next section formalizes the precise notion of security we consider in $\mathsf{Ob}^{\langle\rangle}_{\mathsf{SEC}}$, and proves that it is implied by typing.

5 Polymorphic Relaxed Noninterference, Formally

We now formally define the security property of *polymorphic type-based relaxed noninterference* (PRNI), and prove that the $\mathsf{Ob}^{\langle\rangle}_{\mathsf{SEC}}$ type system soundly enforces PRNI.

5.1 Logical Relation

We define how values, terms and environments are related through a step-indexed logical relation [11] (Figure 7). Step-indexing is needed to ensure that the logical relation is well-founded in presence of recursive object types.

The main novelty of this logical relation with respect that of $\mathsf{Ob}_{\mathsf{SEC}}$ is that it needs to give an interpretation to polymorphic security types of the form $T \triangleleft X$. We do this by quantifying over all possible actual types U for X and interpreting $T \triangleleft U$. The interpretation of a security type is expressed as sets of *atoms* of the form (k, e_1, e_2), where k is a *step index* meaning that e_1 and e_2 are related for k steps.

The definition also appeals to a *simple* typing judgment $\Gamma \vdash_1 e : T$, which disregards the declassification types, and is therefore standard. We have that $\Delta; \Gamma \vdash e : T \triangleleft U \Rightarrow \Gamma \vdash_1 e : T$. The use of this simple type system in the logical relation clearly separates the definitions of security from its static enforcement by the type system of §4.3 [5].

The logical relation uses several auxiliary definitions. $\mathrm{Atom}_n[T]$ requires e_1 and e_2 to be *simply well-typed* expressions of type T and the index k to be strictly less than n. $\mathrm{Atom}^{val}_n[T]$ restricts $\mathrm{Atom}_n[T]$ to values. $\mathrm{Atom}[T]$ are atoms of simply well-typed expressions of type T (*i.e.* for *any* step-index k).

The definition of $\mathcal{V}[\![T \triangleleft O]\!]$ relates two objects o_1, o_2 for k steps if for any method $m \in O$ and with signature $\langle X : A..B\rangle\, S' \to S''$ and $j < k$, given related arguments for j steps at S', invocations of m produce related results for j steps at S''. More specifically, given *any* actual type T' that satisfies the bounds of the type parameter X (i.e., $T' \in A..B$) and given related arguments in $\mathcal{V}[\![S'[T'/X]]\!]$ we must obtain related computations in $\mathcal{V}[\![S''[T'/X]]\!]$.

The relational interpretation of expressions $\mathcal{C}[\![T \triangleleft U]\!]$ relates atoms of the form (k, e_1, e_2) that satisfy that for all $j < k$, if both expressions e_1 and e_2 reduce to values v_1 and v_2 in at most k steps then v_1 and v_2 must be equivalent for the remaining $k - j$ steps. This definition is termination-insensitive: if one expression does not terminate in less that k steps, then the two expressions are trivially equivalent.

Type environments have standard interpretations. $\mathcal{G}[\![\Gamma]\!]$ relates value substitutions γ, *i.e.* mappings from variables to closed values, as triples of the form (k, γ_1, γ_2), where γ_1 and γ_2 are related if they have the same variables as Γ, and for any variable x, the associated values are related for k steps at type $\Gamma(x)$. Finally, a type substitution σ, *i.e.* a mapping from type variables to closed types, *satisfies* a type variable environment Δ, noted $\mathcal{D}[\![\Delta]\!]$, if it has the same type variables that Δ and the mapped type T is within the type variable bounds.

5.2 Defining Polymorphic Relaxed Noninterference

Having defined the logical relation, we can now formally define PRNI. As standard, noninterference properties allow modular reasoning about open terms with respect to (term-level) variables. For PRNI, we extend this modular reasoning principle to open terms with respect to *type* variables. Then, a simply well-typed expression e under Δ and a well-formed Γ satisfies PRNI at well-formed type S, written $\mathsf{PRNI}(\Delta, \Gamma, e, S)$, if for any type substitution σ that satisfies Δ and two value substitutions γ_1 and γ_2 in the relational interpretation of $\sigma(\Gamma)$, applying the two value substitutions to the expression e produces equivalent expressions at type $\sigma(S)$. As usual, the definition quantifies universally on the step index k. We need only consider a single type substitution σ; indeed, type variables happen only in declassification types, which express the observation power of the public observer. Therefore, for each security type of the form $T \triangleleft X$ we only need to consider *one* actual type U within the bounds of X to pick the observation power of the public observer. The substitution σ captures all these choices.

Definition 2 (Polymorphic relaxed noninterference)**.**

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

This definition captures the intuitive security notion that an expression is secure if it produces indistinguishable outputs up to the declassification power of the public observer (specified by S), when linked with indistinguishable inputs up to their declassification (specified by Γ).

5.3 Security Type Soundness

To establish that all well-typed $\mathsf{Ob}^{\langle\rangle}_{\mathsf{SEC}}$ terms satisfy PRNI, we first introduce a notion of logically-related open terms, and prove that if an expression is related to itself, then it satisfies PRNI. We then prove the fundamental property of the logical relation, which states that well-typed terms are logically-related to themselves.

Two open expressions e_1 and e_2 are logically related at type S in environments Δ and Γ if, given a type substitution σ satisfying Δ and value substitutions γ_1 and γ_2 in the relational interpretation of $\sigma(\Gamma)$, closing these expressions with the given substitutions produces related expressions related at type $\sigma(S)$.

Definition 3 (Logical relatedness of open terms)**.**

$$\begin{aligned}
&\Delta; \Gamma \vdash e_1 \approx e_2 : S \iff \\
&\quad \Delta; \Gamma \vdash e_i : S \ \wedge\ \Delta \vdash \Gamma \ \wedge\ \Delta \vdash S \ \wedge \\
&\quad \forall k \geq 0.\ \forall \sigma, \gamma_1, \gamma_2.\ \sigma \in \mathcal{D}[\![\Delta]\!]. \\
&\qquad (k, \gamma_1, \gamma_2) \in \mathcal{G}[\![\sigma(\Gamma)]\!] \\
&\qquad \implies (k, \sigma(\gamma_1(e_1)), \sigma(\gamma_2(e_2)) \in \mathcal{C}[\![\sigma(S)]\!]
\end{aligned}$$

Trivially, if an expression is logically related to itself, then it satisfies PRNI.

$$
\begin{array}{ll}
\mathrm{Atom}_n\,[T] & = \{(k, e_1, e_2) \mid k < n \ \wedge \ \vdash_1 e_1 : T \ \wedge \ \vdash_1 e_2 : T\} \\
\mathrm{Atom}_n^{val}\,[T] & = \{(k, v_1, v_2) \in \mathrm{Atom}_n\,[T]\} \\
\mathrm{Atom}\,[T] & = \{(k, e_1, e_2) \in \bigcup_{n \geq 0} \mathrm{Atom}_n\,[T]\} \\
\mathcal{V}[\![T \triangleleft O]\!] & = \{(k, v_1, v_2) \in \mathrm{Atom}\,[T] \mid \\
& \quad (\forall m \in O. \quad \mathsf{msig}(O, m) = \langle X : A..B\rangle\, S' \to S'' \\
& \quad\quad \forall j < k,\, T', v_1', v_2'. \quad \vdash T' \ \wedge \ T' \in A..B \ \wedge \\
& \quad\quad (j, v_1, v_2) \in \mathcal{V}[\![T \triangleleft O]\!] \ \wedge \ (j, v_1', v_2') \in \mathcal{V}[\![S'\,[T'/X]]\!] \implies \\
& \quad\quad\quad (j, v_1.m\,\langle\_\rangle\,(v_1'), v_2.m\,\langle\_\rangle\,(v_2')) \in \mathcal{C}[\![S''\,[T'/X]]\!])\} \\
\mathcal{C}[\![T \triangleleft U]\!] & = \{(k, e_1, e_2) \in \mathrm{Atom}\,[T] \mid \ (\forall j < k. (e_1 \longmapsto^{\leq j} v_1 \ \wedge \ e_2 \longmapsto^{\leq j} v_2) \implies (k - j, v_1, v_2) \in \mathcal{V}[\![T \triangleleft U]\!])\} \\
\mathcal{G}[\![\cdot]\!] & = \{(k, \emptyset, \emptyset)\} \\
\mathcal{G}[\![\Gamma; x : S]\!] & = \{(k, \gamma_1\,[x \mapsto v_1], \gamma_2\,[x \mapsto v_2]) \mid \ (k, \gamma_1, \gamma_2) \in \mathcal{G}[\![\Gamma]\!] \ \wedge \ (k, v_1, v_2) \in \mathcal{V}[\![S]\!]\} \\
\mathcal{D}[\![\cdot]\!] & = \{\emptyset\} \\
\mathcal{D}[\![\Delta; X : A..B]\!] & = \{\sigma\,[X \mapsto T] \mid \ \sigma \in \mathcal{D}[\![\Delta]\!] \wedge \Delta \vdash T \in A..B\}
\end{array}
$$

Fig. 7. $\mathsf{Ob}^{\langle\rangle}_{\mathsf{SEC}}$ Step-indexed logical relation for type-based equivalence

Lemma 2 (Self logical relation implies PRNI)**.**
$\Delta; \Gamma \vdash e \approx e : S \implies \mathsf{PRNI}(\Delta, \Gamma, e, S)$

We then turn to proving that all well-typed terms are logically-related to themselves, *i.e.* the fundamental property of the logical relation.

Theorem 3 (Fundamental property)**.**
$\Delta; \Gamma \vdash e : S \implies \Delta; \Gamma \vdash e \approx e : S$

Proof. The proof is by induction on the typing derivation of e. We use the common approach of defining compatibility lemmas for each typing rule [11]. Each case follows from the corresponding compatibility lemma. □

Security type soundness follows directly from Theorem 3 and Lemma 2.

Theorem 4 (Security type soundness)**.**
$\Delta; \Gamma \vdash e : S \implies \mathsf{PRNI}(\Delta, \Gamma, e, S)$

Having proven that well-typed $\mathsf{Ob}^{\langle\rangle}_{\mathsf{SEC}}$ programs are secure, we are almost ready to revisit the examples of Section 3 to illustrate PRNI. We must first address the case of primitive types, discussed next.

6 Ad-hoc Polymorphism for Primitive Types

Both $\mathsf{Ob}_{\mathsf{SEC}}$ [5] and $\mathsf{Ob}^{\langle\rangle}_{\mathsf{SEC}}$ (so far) ignore the case of primitive types, such as integers and strings. However, in an object-oriented language, primitive types are both necessary and challenging from a security viewpoint. In particular, integrating type-based declassification with faceted types requires appealing to a form of *ad hoc* polymorphism.

6.1 The Need and Challenge of Primitive Types

In a pure object-oriented calculus (as in a pure functional calculus) without primitive types, the only real observation that can be made on programs is termination. A termination-insensitive notion of noninterference is therefore useless in a pure setting: one needs some primitive types with a purely syntactic notion of equality. Indeed, all the examples we presented in earlier sections assume a syntactic notion of observation for strings and integers.

Introducing primitive types calls for some form of label polymorphism. Indeed, we do not want to fix the security level of primitive operations, as this would be either impractical for the public observer (if all security labels were high) or for the secret observer (if all security labels were low). This is why practical security-typed languages like FlowCaml [7] and Jif [6] use label-polymorphic primitive operators, specifying that the return label is the *least upper bound* of the argument labels. For instance, a binary integer operator would have type $\forall \ell_1, \ell_2. \mathsf{Int}_{\ell_1} \times \mathsf{Int}_{\ell_2} \to \mathsf{Int}_{\ell_1 \sqcup \ell_2}$. In a monomorphic security language, the same principle is hardcoded in the typing rules for primitive operators [12].

Unfortunately this approach does not work with labels-as-types, even in a label-monomorphic setting. Indeed, because labels are types, returning the join of the argument security labels means computing the *subtyping join* (denoted $\sqcup_{<:}$ hereafter) of the declassification types. This is both impractical, incorrect, and potentially unsound from a security viewpoint:

- *Impractical.* Consider a function of type $\mathsf{Bool} \triangleleft X_1 \times \mathsf{Int} \triangleleft X_2 \to \mathsf{Bool} \triangleleft (X_1 \sqcup_{<:} X_2)$. Given two public arguments (*i.e.* $X_1 = \mathsf{Bool}$, $X_2 = \mathsf{Int}$), then assuming $\mathsf{Bool} \sqcup_{<:} \mathsf{Int} = \top$, the result is necessarily private. While sound, this is far too conservative; it is impractical for primitive operations to always return private values even when given public inputs.
- *Incorrect.* Consider an integer comparison of type $\mathsf{Int} \triangleleft X_1 \times \mathsf{Int} \triangleleft X_2 \to \mathsf{Bool} \triangleleft (X_1 \sqcup_{<:} X_2)$. If we instantiate this signature with Int and Int we obtain an *ill-formed* return type, $\mathsf{Bool} \triangleleft \mathsf{Int}$.
- *Insecure.* Consider a unary integer operator $\mathsf{Int} \triangleleft X \to \mathsf{Int} \triangleleft X$; this signature is not sound security-wise for all unary integer operators. Take an operator that trims the most-significant bit of its argument. If one declassifies only the parity of the argument, two equivalent inputs

will not always yield two equivalent outputs (as the parity of the trimmed values might differ).

6.2 Sound Signatures for Primitive Types

The observations above reveal one of the flip sides of expressive declassification policies: because declassification is captured semantically, declassification polymorphism is a very strong notion compared to standard label polymorphism. In the general case, without appealing to intricate semantic conditions, there are therefore only two simple syntactic principles to define sound signatures for primitive operators:[3]

(P1) if every argument is public (*e.g.* $\mathsf{String_L}$) then the return type can be public.

(P2) if any argument is not public (*e.g.* $\mathsf{String} \triangleleft \mathsf{StringFst}$) then the return type must be secret (*e.g.* $\mathsf{String_H}$).

As is typical, we provide an object-oriented interface for primitive types (*e.g.* $a + b$ is $a.+(b)$ as in Scala for instance). Therefore the principles above must be extended to account for the status of the *receiver* object: if the primitive method invoked on the primitive value is part of its declassification type, then it is considered a public "argument"; otherwise, it is private.

Note that, without any form of polymorphism, *i.e.* picking a single syntactic principle above, primitive types would be impractical. Duplicating all definitions to offer both options is also not a viable approach.

6.3 Polymorphic Primitive Signatures

To support the two syntactic principles exposed above, we propose to use *ad-hoc* polymorphism (akin to overloading) for primitive types P in $\mathsf{Ob}^{\langle\rangle}_{\mathsf{SEC}}$. We introduce *polymorphic primitive signatures*, written $P \triangleleft * \rightarrow P \triangleleft *$. A primitive security type $P \triangleleft *$ is resolved polymorphically *at use site*, following principles (P1) and (P2) above. Object-oriented interfaces for primitive types are exclusively composed of polymorphic primitive signatures. For instance, in $\mathsf{Ob}^{\langle\rangle}_{\mathsf{SEC}}$ strings are primitives, declared by the following String type:

$$\begin{aligned}\mathsf{String} \triangleq [\ & \mathsf{concat} : \mathsf{String} \triangleleft * \rightarrow \mathsf{String} \triangleleft *,\\ & \mathsf{first} : \mathsf{Unit} \triangleleft * \rightarrow \mathsf{String} \triangleleft *,\\ & \mathsf{length} : \mathsf{Unit} \triangleleft * \rightarrow \mathsf{Int} \triangleleft *,\\ & \mathsf{eq} : \mathsf{String} \triangleleft * \rightarrow \mathsf{Bool} \triangleleft *,\\ & \cdots]\end{aligned}$$

To illustrate, assume $\mathsf{a} : \mathsf{String_L}$, $\mathsf{b} : \mathsf{String_L}$ and $\mathsf{c} : \mathsf{String_H}$. Then $\mathsf{a.eq(b)}$ has type $\mathsf{Bool_L}$, while $\mathsf{a.eq(c)}$ has type $\mathsf{Bool_H}$.

Primitive types can also be subject to declassification policies. For instance, consider:

$$\mathsf{StringEqPoly} \triangleq [\mathsf{eq} : \mathsf{String} \triangleleft * \rightarrow \mathsf{Bool} \triangleleft *]$$

and $\mathsf{d} : \mathsf{String} \triangleleft \mathsf{StringEqPoly}$. Then $\mathsf{d.eq(b)}$ has type $\mathsf{Bool_L}$, while $\mathsf{d.concat(a)} : \mathsf{String_H}$.

[3]The syntactic principles (P1) and (P2) are formally justified by the proof of Lemma 5, discussed in Section 6.5.

$$\begin{array}{rll} e & ::= \cdots \mid e.m(e) & \text{(terms)}\\ v & ::= \cdots \mid \mathsf{p} & \text{(values)}\\ T & ::= \cdots \mid P & \text{(types)}\\ M & ::= \cdots \mid I & \text{(method signatures)}\\ S & ::= \cdots \mid P \triangleleft * & \text{(security types)}\\ I & ::= P \triangleleft * \rightarrow P \triangleleft * & \text{(primitive signatures)}\\ P & ::= \text{(eg. Int, String)} & \text{(primitive types)}\\ \Phi & ::= \cdots \mid \Phi, P <: \beta & \text{(subtyping environments)}\end{array}$$

Fig. 8. $\mathsf{Ob}^{\langle\rangle}_{\mathsf{SEC}}$: Extended syntax for primitive types

$\boxed{\Delta; \Phi \vdash U_1 <: U_2}$

$$\cdots \qquad \text{(SPObj)}\ \frac{\mathsf{meths}(P) = R_1 \quad O \triangleq \mathbf{Obj}(\beta).R_2 \quad \Delta; \Phi, P <: \beta \vdash R_1 <: R_2}{\Delta; \Phi \vdash P <: O}$$

$$\text{(SPVar)}\ \frac{P <: \beta \in \Phi}{\Delta; \Phi \vdash P <: \beta}$$

$\boxed{\Delta; \Phi \vdash M_1 <: M_2}$

$$\cdots \quad \text{(SImpl)}\ \frac{}{\Delta; \Phi \vdash I <: I}$$

Fig. 9. $\mathsf{Ob}^{\langle\rangle}_{\mathsf{SEC}}$: Subtyping rules for primitive types

Furthermore, one can use any type signature in a declassification policy for a primitive type, as long as it is sound. For instance, $\mathsf{StringEqL} \triangleq [\mathsf{eq} : \mathsf{String_L} \rightarrow \mathsf{Bool_L}]$ respects principle 1). Conversely, $\mathsf{StringEqBad} \triangleq [\mathsf{eq} : \mathsf{String_H} \rightarrow \mathsf{Bool_L}]$ cannot be used as it would violate the soundness principles above (in $\mathsf{Ob}^{\langle\rangle}_{\mathsf{SEC}}$, $\mathsf{String} \triangleleft \mathsf{StringEqBad}$ is ill-formed).

6.4 Formal Semantics

We now formalize the treatment of primitive types in $\mathsf{Ob}^{\langle\rangle}_{\mathsf{SEC}}$. Figure 8 presents the extended syntax to support primitive values p and primitive types P. Expression $e.m(e)$ is for method invocation on primitives; as explained previously, primitive types expose an object-oriented interface, so $a + b$ is $a.+(b)$. We extend the category S with security types of the form $P \triangleleft *$ and introduce a new category I, for primitive signatures $P \triangleleft * \rightarrow P \triangleleft *$. The security type $P \triangleleft *$ can be used for standard signatures $\langle X : A..B\rangle\, S_1 \rightarrow S_2$ as well.

The changes to subtyping are in Figure 9. Now, subtyping assumptions can be also made between a primitive type and a self type variable, *i.e.* $\Phi ::= \bullet \mid \Phi, \alpha <: \beta \mid \Phi, P <: \beta$, and function meths returns the methods of a primitive type. Rule (SPObj) justifies subtyping between a primitive type and an object type, and it is very similar to rule (SObj) of Figure 3 for object types. Rule (SPVar) accounts for subtyping between primitive types and type variables and it holds if such subtyping relation exists in the subtyping environment Φ. Note that there is no rule for subtyping between an object type and a primitive type, because this would not be sound. Rule (SImpl) accounts for subtyping between the same primitive

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

$$\cdots \quad \text{(TPrim)}\ \frac{P = \Theta(\mathsf{p})}{\Delta;\Gamma \vdash \mathsf{p} : P \triangleleft P}$$

$$\text{(TPmD)}\ \frac{\begin{array}{c}\Delta;\Gamma \vdash e_1 : T \triangleleft U \qquad \Delta \vdash m \in U \\ \mathsf{msig}(\Delta, U, m) = P_1 \triangleleft * \to P_2 \triangleleft * \\ \Delta;\Gamma \vdash e_2 : P_1 \triangleleft U_1 \\ \mathsf{rdecl}(P_1 \triangleleft U_1, P_2) = P_2'\end{array}}{\Delta;\Gamma \vdash e_1.m(e_2) : P_2 \triangleleft P_2'}$$

$$\text{(TPmH)}\ \frac{\begin{array}{c}\Delta;\Gamma \vdash e_1 : T \triangleleft U \qquad \Delta \vdash m \notin U \\ \mathsf{msig}(\Delta, T, m) = P_1 \triangleleft * \to P_2 \triangleleft * \\ \Delta;\Gamma \vdash e_2 : P_1 \triangleleft U_1\end{array}}{\Delta;\Gamma \vdash e_1.m(e_2) : P_2 \triangleleft \top}$$

$\boxed{\mathsf{rdecl}(P_1 \triangleleft U_1, P_1) = U}$

$$\mathsf{rdecl}(P_1 \triangleleft U_1, P_2) = \begin{cases} P_2 & P_1 = U_1 \\ \top & \textit{otherwise} \end{cases}$$

Fig. 10. $\mathsf{Ob}^{\langle\rangle}_{\mathsf{SEC}}$: Extended static semantics for primitive types

signature. There is no rule to justify subtyping between a primitive signature and a standard signature.

As we discussed at the end of Section 6.2, we need an extra condition for the well-formedness of security types to ensure sound declassification. Given the type $T \triangleleft U$, if T has a method $m : I$, the method signature of m in U must be either the same primitive signature I, or a normal signature that is *sound*. We use the predicate soundsig to express that signature $\langle X : A..B\rangle\, S_1 \to S_2$ is sound, which must satisfy that either the argument type is public, or the return type is private:

$$\mathsf{soundsig}(\langle\_\rangle\, P \triangleleft U_1 \to T_2 \triangleleft U_2) \Longleftrightarrow U_1 = P \vee U_2 = \top$$

Figure 10 presents the extension to the typing rules of $\mathsf{Ob}^{\langle\rangle}_{\mathsf{SEC}}$. Rule (TPrim) justifies typing for primitive values, using a function Θ that specifies each primitive type. The new typing rules (TPmD) and (TPmH) realize ad hoc polymorphism for primitive types. Rule (TPmD) is key: it applies when m is in the declassification type U, and uses the function rdecl to calculate the declassification type of the return type, based on the type of the argument: if the argument is public, so is the returned value. Rule (TPmH) applies when m is not in the declassification type U, and similarly to (TmH), ensures that the returned value is private.

Figure 11 shows the extension to the dynamic semantics to support primitive values. Rule (EMInvP) executes a method invocation on a primitive value using the function θ, which abstracts over the internal implementation of primitive values.

To prove type safety for $\mathsf{Ob}^{\langle\rangle}_{\mathsf{SEC}}$ with primitives, we only need to assume that Θ and θ—which are parameters of the language—agree on the specification and implementation of all primitive types and their operations.

$$\text{(EMInvP)}\ \frac{}{E[\mathsf{p}_1.m(\mathsf{p}_2)] \longmapsto E[\theta(m, \mathsf{p}_1, \mathsf{p}_2)]}$$

Fig. 11. $\mathsf{Ob}^{\langle\rangle}_{\mathsf{SEC}}$: Dynamic semantics of primitive values

$$\begin{array}{l}\mathcal{V}[\![P \triangleleft P]\!] = \{(k, \mathsf{p}, \mathsf{p}) \in \mathrm{Atom}\,[P]\} \\ \mathcal{V}[\![T \triangleleft O]\!] = \ \{(k, v_1, v_2) \in \mathrm{Atom}\,[T] \mid \ldots \\ \qquad \forall m \in O.\quad \mathsf{msig}(\bullet, O, m) = P_1 \triangleleft * \to P_2 \triangleleft * \\ \qquad \forall j < k, v_1', v_2'.U_1 >: P_1 \\ \qquad ((j, v_1', v_2') \in \mathcal{V}[\![P_1 \triangleleft U_1]\!] \implies \\ \qquad (j, v_1.m(v_1'), v_2.m(v_2')) \in \mathcal{C}[\![P_2 \triangleleft \mathsf{rdecl}(P_1 \triangleleft U_1, P_2)]\!])\}\end{array}$$

Fig. 12. $\mathsf{Ob}^{\langle\rangle}_{\mathsf{SEC}}$: Step-indexed logical relation with new definitions for primitive types

6.5 Logical Relation for Primitive Types

Figure 12 presents the extension to the logical relation of Figure 7 to account for primitive types. First, $\mathcal{V}[\![P \triangleleft P]\!]$ relates *syntactically equal* primitive values of type P. Second, the definition of $\mathcal{V}[\![T \triangleleft O]\!]$ now accounts for primitive values that are observed with a declassification type O. $\mathcal{V}[\![T \triangleleft O]\!]$ still relates values v_1 and v_2 if, for all methods of O, given related arguments, the invocations of m on v_1 and v_2 produce related computations. However, the definition now discriminates between each type of signatures. For a method m with primitive signature $P_1 \triangleleft * \to P_2 \triangleleft *$, we require one of the following conditions to hold. If we get related arguments v_1' and v_2' at $P_1 \triangleleft P_1$ (*i.e.* public values), method invocations $v_1.m(v_1')$ and $v_2.m(v_2')$ need to be related at the public type $P_2 \triangleleft P_2$. Otherwise, if the arguments v_1' and v_2' are related at a non-public type ($P_1 \triangleleft U,\ P_1 \neq U$), then $v_1.m(v_1')$ and $v_2.m(v_2')$ need to be related at the top type $P \triangleleft \top$. These conditions are expressed in the definition by requiring related method invocations in $\mathcal{C}[\![P_2 \triangleleft \mathsf{rdecl}(P_1 \triangleleft U_1, P_2)]\!]$.

Extending the fundamental property (Lemma 3) for primitive types requires the following lemma, which states that syntactically-equal primitive values of type P are in the object-oriented interpretation of any type $P \triangleleft O$—essentially, equal values cannot be discriminated.

Lemma 5 (Equal values are logically related)**.**
$\forall k \geq 0, \mathsf{p}, P, O.$

$$\vdash_1 \mathsf{p} : O \ \wedge\ P <: O \implies (k, \mathsf{p}, \mathsf{p}) \in \mathcal{V}[\![P \triangleleft O]\!]$$

Proof. Because $P \triangleleft O$ is a well-formed security type, O consists of primitive signatures and standard, sound signatures. For the case of primitive signatures, at some point we have $P_1 \triangleleft * \to P_2 \triangleleft *$ and equivalent values at $(j, v_1, v_2) \in P_1 \triangleleft U_1$ and we have to prove that $(j, \delta(m, b, v_1), \delta(m, b, v_2) \in \mathcal{C}[\![P_2 \triangleleft \mathsf{rdecl}(P_1 \triangleleft U_1, P_2)]\!]$. We do case analysis on U. If $U_1 = P_1$, we know that $v_1 = v_2$ and hence if $\delta(m, b, v_1)$ and $\delta(m, b, v_2)$ are defined, their results are syntactically equal, so $(j, \delta(m, b, v_1), \delta(m, b, v_2)) \in \mathcal{C}[\![P_2 \triangleleft P_2)]\!]$. If $U \neq P_1$, then the proof obligation is $(j, \delta(m, b, v_1), \delta(m, b, v_2) \in \mathcal{C}[\![P_2 \triangleleft \top]\!]$; this is trivial because any two values are re-

lated at $\top$. For the case of standard signatures, at some point we have $P_1 \triangleleft U_1 \to P_2 \triangleleft U_2$ and we have to prove that $(j, \delta(m, b, v_1), \delta(m, b, v_2)) \in \mathcal{C}[\![P_2 \triangleleft \mathsf{rdecl}(P_1 \triangleleft U_1, P_2)]\!]$. Since the signature is sound, we know that either $U_1 = P_1$ or $U_2 = \top$; then the result follows similarly to the primitive signature case. $\square$

Note that the two syntactic principles for sound signatures of primitive types introduced in Section 6.2 are justified by the need to establish Lemma 5. Principle (P1) is necessary because we cannot assume anything about two invocations of an *arbitrary* partial function δ, except that given *syntactically* equal arguments, if it produces results, then those results are syntactically equal. Principle (P2) is justified because any two invocations of δ are observationally equivalent at $\top$ (like any computation in general), so the actual relation between the arguments does not matter. For any primitive operator signature that does not abide by either (P1) or (P2), it is possible to devise a δ that violates Lemma 5

Consequently, $\mathsf{Ob}^{\langle\rangle}_{\mathsf{SEC}}$ with primitive types is a sound security-typed language, *i.e.* all well-typed programs satisfy PRNI (Theorem 4).

6.6 Illustration

In Section 3 we gave informal examples of secure and insecure programs with respect to PRNI. Now, armed with Theorem 4, and the definitions for primitive types, we can formally check if a given program is secure by typechecking it. The prototype implementation of $\mathsf{Ob}^{\langle\rangle}_{\mathsf{SEC}}$ features a number of examples and allows one to try out the language and typechecker. In this section, we unfold the reasoning underlying the proof of Theorem 4 for a specific example, in order to illustrate the technical details of PRNI and the relational interpretation of object types, including primitive signatures.

To lighten notation, we omit unused type parameters in method declarations and type instantiations in method invocations. Note that we introduce the Unit type with its unique unit primitive value.

We illustrate polymorphic declassification by considering type and variable environments $\Delta \stackrel{\triangle}{=} X : \mathsf{String}..\mathsf{StringLen}$ and $\Gamma \stackrel{\triangle}{=} x : \mathsf{String} \triangleleft X$. We discuss two possible formal definitions for StringLen, either using standard method signatures, or using primitive signatures:

1) $\mathbf{Obj}(\alpha).\,[\mathsf{length} : \mathsf{Unit}_{\mathsf{L}} \to \mathsf{Int}_{\mathsf{L}}]$
2) $\mathbf{Obj}(\alpha).\,[\mathsf{length} : \mathsf{Unit} \triangleleft * \to \mathsf{Int} \triangleleft *]$

With definition 1) above, the program $x.\mathsf{length}(\mathsf{unit})$ has type $\mathsf{Int}_{\mathsf{L}}$; *i.e.* $\Delta; \Gamma \vdash x.\mathsf{length}(\mathsf{unit}) : \mathsf{Int}_{\mathsf{L}}$. Then, by Theorem 4, we know that $\mathsf{PRNI}(\Delta, \Gamma, x.\mathsf{length}(\mathsf{unit}), \mathsf{Int}_{\mathsf{L}})$ holds; the program is secure for any public observer.

Let us unfold $\mathsf{PRNI}(\Delta, \Gamma, x.\mathsf{length}(\mathsf{unit}), \mathsf{Int}_{\mathsf{L}})$ to verify why it holds. For any type substitution $X \mapsto T \in \mathcal{D}[\![\Delta]\!]$ and equivalent value substitutions $(k, x \mapsto v_1, x \mapsto v_2) \in \mathcal{G}[\![\bullet, x : \mathsf{String} \triangleleft T]\!]$, we have that $(k, v_1.\mathsf{length}(\mathsf{unit}), v_2.\mathsf{length}(\mathsf{unit})) \in \mathcal{C}[\![\mathsf{Int} \triangleleft \mathsf{Int}]\!]$.

To verify this:

1) By $(k, x \mapsto v_1, x \mapsto v_2) \in \mathcal{G}[\![\bullet, x : \mathsf{String} \triangleleft T]\!]$ we know that $(k, v_1, v_2) \in \mathcal{V}[\![\mathsf{String} \triangleleft T]\!]$. Because $T <: \mathsf{StringLen}$, we have $\mathcal{V}[\![\mathsf{String} \triangleleft T]\!] \subseteq \mathcal{V}[\![\mathsf{String} \triangleleft \mathsf{StringLen}]\!]$ by a subtyping lemma, and hence $(k, v_1, v_2) \in \mathcal{V}[\![\mathsf{String} \triangleleft \mathsf{StringLen}]\!]$.
2) Then, instantiate the definition of $\mathcal{V}[\![\mathsf{String} \triangleleft \mathsf{StringLen}]\!]$ with $\mathsf{length}, k, T, \mathsf{unit}, \mathsf{unit}$. Note that:
   - $\mathsf{length} \in \mathsf{StringLen}$
   - $\mathsf{msig}(\bullet, \mathsf{StringLen}, \mathsf{length}) = \mathsf{Unit}_{\mathsf{L}} \to \mathsf{Int}_{\mathsf{L}}$
   - $T \in \mathsf{String}..\mathsf{StringLen}$, which follows from $X \mapsto T \in \mathcal{D}[\![\Delta]\!]$
   - and $(k, \mathsf{unit}, \mathsf{unit}) \in \mathcal{V}[\![\mathsf{Unit} \triangleleft \mathsf{Unit}]\!]$ (by definition of $\mathcal{V}[\![P \triangleleft P]\!]$),

Then $(k, v_1.\mathsf{length}(\mathsf{unit}), v_2.\mathsf{length}(\mathsf{unit})) \in \mathcal{C}[\![\mathsf{Int} \triangleleft \mathsf{Int}]\!]$.

With definition 2) above of StringLen, we apply the same steps until the instantiation of $\mathcal{V}[\![\mathsf{String} \triangleleft \mathsf{StringLen}]\!]$. At this point, since length has a primitive signature, we have to consider the extended case for primitive type signatures from Figure 12. Instantiate it with $\mathsf{length}, k, \mathsf{unit}, \mathsf{unit}$, and observe that $\mathsf{msig}(\bullet, \mathsf{StringLen}, \mathsf{length}) = \mathsf{Unit} \triangleleft * \to \mathsf{Int} \triangleleft *$. Then, given that $(k, \mathsf{unit}, \mathsf{unit}) \in \mathcal{V}[\![\mathsf{Unit} \triangleleft \mathsf{Unit}]\!]$, we have that $(k, v_1.\mathsf{length}(\mathsf{unit}), v_2.\mathsf{length}(\mathsf{unit})) \in \mathcal{C}[\![\mathsf{Int} \triangleleft \mathsf{rdecl}(\mathsf{Unit} \triangleleft \mathsf{Unit}, \mathsf{Int})]\!] = \mathcal{C}[\![\mathsf{Int} \triangleleft \mathsf{Int}]\!]$.

7 Related work

Declassification. Expressive declassification policies were introduced by Li and Zdancewic [3] with the labels-as-functions approach. They define two kinds of declassification policies: local and global. Local policies are concerned with one secret, while global policies express coordinated declassification of several secrets. Label operations in this approach rely on a semantic interpretation of declassification functions based on a general notion of program equivalence. In addition to the induced complexity, this precludes recursive policies.

The labels-as-types approach [5] uses type interfaces instead of functions to express declassification policies. This simplifies the concepts involved (label ordering is simply subtyping), making an implementation more easily realizable. The approach naturally support local policies. More advanced typing disciplines such as refinement types [13] could in principle be used to express global policies.

Conceptually, Cruz et al. [5] relate secure information flow with type abstraction, a connection also explored under different angles by Bowman and Ahmed [14] and Washburn and Weirich [15]. Bowman and Ahmed [14] translate the noninterference result of the Dependency Core Calculus (DCC) [16] to parametricity, while Washburn and Weirich [15] use information control mechanisms to ensure a generalized from of parametricity in presence of runtime type inspection.

The Decentralized Label Model (DLM) [17] of Jif enforces *robust declassification* [18]: restricting *who* can declassify values, using the *integrity policy* to ensure that the declassification is not triggered by conditions affected by an active attacker. Here, we do not model any notion of authority, focusing on the *what* dimension of declassification [2].

As noted by Sabelfeld and Sands [2], many declassification approaches of the *what* dimension can be expressed using partial equivalence relations to model the public observer knowledge. Here, we use the logical interpretation of security types (Figure 7) to specify the partial equivalence relation that a public observer can use to distinguish values and computations.

Label polymorphism. Support for label polymorphism in security-typed programming languages can be classified in two categories: static and dynamic label polymorphism. Static label polymorphism can either be provided via explicit syntactic constructions to introduce generic labels [6], or implicitly with constraint-based label inference [6, 7, 19]. The dynamic form of label polymorphism relies on labels as first-class entities that can be passed around like standard values [6, 20, 21].

The Jif language [6] supports all three forms of label polymorphism. It provides a direct syntax to introduce labels at the method and class levels, which can be constrained. Also, Jif features label inference: local variables are inferred to have a fresh generic label that is resolved using constraints from the context. Inferred fields and method arguments have default labels. In addition, Jif supports first-class labels. Our work focuses on the foundations of explicit declassification polymorphism, and currently does not address label inference and first-class labels. Because labels are types, label inference would boil down to fairly standard type inference; first-class labels however would require a notion of first-class types, which should be considered with care.

Sun et al. [19] design a constraint-based label inference mechanism for an object-oriented language with classes and inheritance. Classes and methods are label polymorphic. The programmer can rely on the inference mechanism to achieve label polymorphism or to specify generic labels at the class level; method-level explicit polymorphism is not considered. Stefan et al. [20, 21] provide label-polymorphism via first-class labels much like Jif.

Declassification and Polymorphism. When present, the declassification mechanisms of the label-polymorphic proposals discussed above [6, 7, 19, 21] are completely orthogonal to label polymorphism. The polymorphic labels-as-types approach developed in this work allows us to reason about declassification and label polymorphism with the single and unified concept of standard types.

Our approach is closely related to that of Hicks et al. [4], which propose *trusted declassification* in an object-oriented language based on the DLM [17], where each label is composed of principals. Declassification is globally defined, associating principals to the *trusted methods* that can be used to declassify an expression to another principal. Because classes are polymorphic with respect to principals, this induces a form of implicit label polymorphism. More precisely, a class definition is checked at instantiation-time with the actual principals provided for the instantiation. This use-site polymorphism for principals is similar to our treatment of polymorphic primitive signatures (Section 6).

Tse and Zdancewic [22] propose certificate-based declassification and conditioned noninterference. They extend System $\mathsf{F}_{<:}$ with monadic labels similarly to DCC [16], using DLM [17]. Declassification is modeled as a *read privilege* that a principal is allowed to give to another principal in a certain context. Their work merges standard types with labels, principals and privileges in the same syntactic category of types. Since System $\mathsf{F}_{<:}$ supports type polymorphism, the language supports label polymorphism. However, it is not clear how to use label polymorphism to express polymorphic declassification in that setting.

Finally, the syntactic principles we introduce for primitive signatures are related to the work of Li and Zdancewic [3] on labels-as-functions. For local policies, the typing rules for integer primitive operators follow the same principles, but are more expressive. In particular, they provide typing rules for binary integer operators where one operand has an arbitrary declassification policy and the other operand is public; the resulting label is a functional composition of the operand label with a function wrapping the operator. As explained before, this semantic implication cannot be expressed with the labels-as-types approach, unless one is willing to consider much more advanced typing disciplines.

8 Conclusion

We extend relaxed noninterference in a labels-as-types approach to selective and expressive declassification in order to account for polymorphism. The proposed declassification polymorphism is novel and useful to precisely control declassification of polymorphic structures and to define procedures that are polymorphic over the declassification policies of their arguments. Bounded polymorphism further controls the guarantees and expectations of clients and providers with respect to declassification. Bringing type-based declassification to real-world programming also requires addressing the issue of primitive types, which were ignored in prior work. For this we introduce a novel form of ad-hoc polymorphism. We formalize the approach, prove its soundness, and provide a prototype implementation.

This work provides a necessary and solid basis to integrate type-based declassification in existing languages. A particularly appealing alternative is to study the realization of our approach in Scala: its type system is expressive enough to encode bounded polymorphic declassification, and adjusting the typechecker to account for security levels (*i.e.* the additional rule for method invocation) should be achievable via a compiler plugin.

Acknowledgment

We thank Cătălin Hriţcu and the anonymous reviewers for their detailed comments and suggestions.

References

[1] D. Volpano, C. Irvine, and G. Smith, "A sound type system for secure flow analysis," *Journal of Computer Security*, vol. 4, no. 2-3, pp. 167–187, Jan. 1996.

[2] A. Sabelfeld and D. Sands, "Declassification: Dimensions and principles," *Journal of Computer Security*, vol. 17, no. 5, pp. 517–548, 2009.

[3] P. Li and S. Zdancewic, "Downgrading policies and relaxed noninterference," in *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2005)*. Long Beach, CA, USA: ACM Press, Jan. 2005, pp. 158–170.

[4] B. Hicks, D. King, P. McDaniel, and M. 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*, 2006, pp. 65–74.

[5] R. Cruz, T. Rezk, B. Serpette, and É. Tanter, "Type abstraction for relaxed noninterference," in *Proceedings of the 31st European Conference on Object-Oriented Programming (ECOOP 2017)*, ser. Leibniz International Proceedings in Informatics (LIPIcs), P. Müller, Ed., vol. 74. Barcelona, Spain: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Jun. 2017, pp. 7:1–7:27.

[6] A. C. Myers, "Jif homepage," http://www.cs.cornell.edu/jif/, accessed March 2019.

[7] F. Pottier and V. Simonet, "Information flow inference for ML," *ACM Transactions on Programming Languages and Systems*, vol. 25, no. 1, pp. 117–158, 2003.

[8] M. Abadi and L. Cardelli, *A Theory of Objects*. Springer-Verlag, 1996.

[9] A. Igarashi, B. C. Pierce, and P. Wadler, "Featherweight Java: a minimal core calculus for Java and GJ," *ACM Transactions on Programming Languages and Systems*, vol. 23, no. 3, pp. 396–450, 2001.

[10] T. Rompf and N. Amin, "Type soundness for dependent object types (DOT)," in *Proceedings of the 31st ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, (OOPSLA 2016), part of SPLASH 2016*, E. Visser and Y. Smaragdakis, Eds. Amsterdam, The Netherlands: ACM Press, Oct. 2016, pp. 624–641.

[11] A. Ahmed, "Step-indexed syntactic logical relations for recursive and quantified types," in *Proceedings of the 15th European Symposium on Programming Languages and Systems (ESOP 2006)*, ser. Lecture Notes in Computer Science, P. Sestoft, Ed., vol. 3924. Vienna, Austria: Springer-Verlag, Mar. 2006, pp. 69–83.

[12] S. Zdancewic, "Programming languages for information security," Ph.D. dissertation, Cornell University, Aug. 2002.

[13] P. M. Rondon, M. Kawaguchi, and R. Jhala, "Liquid types," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2008)*, R. Gupta and S. P. Amarasinghe, Eds. ACM Press, Jun. 2008, pp. 159–169.

[14] W. J. Bowman and A. Ahmed, "Noninterference for free," in *Proceedings of the 20th ACM SIGPLAN Conference on Functional Programming (ICFP 2015)*. Vancouver, Canada: ACM Press, Aug. 2015, pp. 101–113.

[15] G. Washburn and S. Weirich, "Generalizing parametricity using information-flow," in *Proceedings of the 20th IEEE Symposium on Logic in Computer Science (LICS 2005)*. Chicago, IL, USA: IEEE Computer Society Press, Jun. 2005, pp. 62–71.

[16] M. Abadi, A. Banerjee, N. Heintze, and J. G. Riecke, "A core calculus of dependency," in *Proceedings of the 26th ACM Symposium on Principles of Programming Languages (POPL 99)*. San Antonio, TX, USA: ACM Press, Jan. 1999, pp. 147–160.

[17] A. C. Myers and B. Liskov, "Protecting privacy using the decentralized label model," *ACM Transactions on Software Engineering and Methodology*, vol. 9, pp. 410–442, Oct. 2000.

[18] S. Zdancewic and A. C. Myers, "Robust declassification," in *Proceedings of the 14th IEEE Computer Security Foundations Workshop (CSFW 2001)*. Cape Breton, Nova Scotia, Canada: IEEE Computer Society Press, Jun. 2001, pp. 15–23.

[19] Q. Sun, A. Banerjee, and D. A. Naumann, "Modular and constraint-based information flow inference for an object-oriented language," in *Proceedings of the 11th Static Analysis Symposium (SAS 2004)*, ser. Lecture Notes in Computer Science, R. Giacobazzi, Ed., vol. 3148. Verona, Italy: Springer-Verlag, Aug. 2004, pp. 84–99.

[20] D. Stefan, A. Russo, J. C. Mitchell, and D. Mazières, "Flexible dynamic information flow control in Haskell," in *Proceedings of the 4th ACM Symposium on Haskell*. ACM Press, 2011, pp. 95–106.

[21] D. Stefan, D. Mazières, J. C. Mitchell, and A. Russo, "Flexible dynamic information flow control in the presence of exceptions," *Journal of Functional Programming*, vol. 27, 2017.

[22] S. Tse and S. 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)*, ser. Lecture Notes in Computer Science, S. Sagiv, Ed., vol. 2986. Berlin, Heidelberg: Springer-Verlag, 2005, pp. 279–294.