Confidentiality of Interactions in Concurrent Object-Oriented Systems[⋆]

Olaf Owe[1] and Toktam Ramezanifarkhani[1]

Department of Informatics, University of Oslo, Norway
{olaf,toktamr}@ifi.uio.no

Abstract. We consider a general concurrency model for distributed systems, based on concurrent objects communicating by asynchronous methods. This model is suitable for modeling of service-oriented systems, and gives rise to efficient interaction, avoiding active waiting and low-level synchronization primitives such as explicit signaling lock operations. This concurrency model has a modular semantics and allows us to focus on information flow at a high level of abstraction. Our approach is formalized by a high-level language with a secrecy-type system ensuring noninterference in object interactions. We prove soundness based on an operational semantics incorporating runtime secrecy levels.

Keywords. Active Objects; Asynchronous Methods; Secrecy; Privacy; Noninterference; Information Flow; Distributed Systems; Static Analysis;

1 Introduction

Programming languages can provide fine-grained control for security issues because they allow accurate and flexible security information analysis of program components [8]. In particular, to specify and enforce information-flow policies, the effectiveness of language-based techniques has been established. Secure information flows are often expressed by semantic models of program execution in the form of a *noninterference* policy. Noninterference stipulates that manipulation and modification of confidential data should be allowed in programs, as long as their visible outputs do not improperly reveal information about the confidential data. Attackers are assumed to be able to view "low" information. The usual method for showing that noninterference holds is to demonstrate that the attacker cannot observe any difference between two executions that differ only in their confidential input [7]. However, attackers may also see intermediate outputs [1] and observe the progress of the program, e.g., absence or presence of the next observable value, which leads to the concept of progress-sensitive noninterference [1].

[⋆] Work supported by the *IoTSec* and *DiversIoT* projects, the Norw. Research Council.

In this paper, we are interested in service-oriented and object-oriented systems at a high level of abstraction, and consider the setting of distributed concurrent objects communicating by asynchronous methods calls. We focus on efficient interaction, including non-blocking calls and high-level mechanisms for process control, suitable for modern service-oriented systems. Our notion of noninterference reflects the non-deterministic nature of interacting concurrent objects. Fields are encapsulated by objects and remote access is forbidden. Thus, fields are non-observable, and the (typically) illegal flows in the sense of assignment of confidential values to non-confidential variables inside objects are not critical.

To formalize our approach we introduce a high-level imperative language based on the chosen concurrency model. This language is derived from the object-oriented language *Creol* [12]. We define an extension of Creol called *SeCreol*, adding awareness of secrecy levels as well as secrecy type information. We define an operational semantics, and prove that our secrecy-type system is sound with respect to the operational semantics, ensuring that every well-typed program of our language satisfies the proposed non-interference property.

2 Object-Oriented Distributed Systems in SeCreol

We consider concurrent, distributed objects where each object has its own execution thread. An object does not have access to the internal state variables of other objects. Communication is only by method calls, allowing asynchronous and synchronous communication, implemented by means of asynchronous message passing. In order to avoid undesirable waiting in the distributed setting, we allow mechanisms for non-blocking method calls. By means of a suspension mechanism, unfinished method invocations in an object may be placed on the object's process queue, for instance while waiting for a response from another object. The process will be enabled when then object receives the response. This allows flexible interleaving of incoming calls and (enabled) suspended processes. Internally in an object, there is at most one process executing at any time. Objects reflect concurrent system components, while data structure inside an object is defined by data types using functional programming.

A *SeCreol* program consists of a number of interfaces and classes (with the last class being the "main" class). An interface may have a number of super-interfaces and method declarations. A class C takes a list of class parameters cp, defines fields w, and has an optional initialization part followed by method definitions. Class parameters cp are like fields apart from being initialized through the **new** statement. Class parameters, the implicit class parameter this, and the implicit method parameter caller are read-only. A class may implement a number of interfaces, and for each method of an interface it is required that the class implements the method such that the type and secrecy level information is respected. Additional methods may be defined in a class as well, but these may not be called from outside the class. All variables and parameters are typed by data types or interfaces. Classes are not allowed as types, which means that an object can only be seen through an interface, and therefore, remote access to fields nor methods that are not exported through an interface is not allowed. Thus shared variable concurrency is avoided. With respect to security analysis, fields are then not observable, and observable behavior is limited to interactions by means of method-oriented communication.

Expressions e and functions f are side-effect free, and $\overline{e}$ is a (possibly empty) expression list, comma-separated. Statements include standard constructs for assignment, skip, if, while, object generation, and sequential composition. The *simple call* statement $e!m(\overline{e})$ is like message passing; a message is sent to the object expressed by e (the callee) indicating that it should execute method m (when the callee is free to do this) with a list of actual parameters $\overline{e}$. Thus the current object is not blocked, and will not wait for the return value. If the return value is desired by the calling object, it may use the *blocking call* statement $v := e.m(\overline{e})$ or the *non-blocking call* statement **await** $v := e.m(\overline{e})$. The latter call statement forces the caller object to suspend the current process, allowing it to continue with any enabled suspended process in its process queue or handle incoming calls. Similarly, the conditional await statement **await** e suspends, placing the current process on the process queue. This process is enabled when the Boolean condition e is satisfied. The considered core language allows high-level and yet efficient method-based interaction, supporting both passive and active waiting. The operational semantics of SeCreol is given in Sec. 4.

The language is strongly typed, and a typing system can be given in the style of [13]. A variable is typed either by an interface or by a data type, called *object variable* or *data variable*, respectively. The runtime value of an object variable is an object identity (or null), and that of a data variable is a data value. Data variables are passed by value and object variables are passed by reference (i.e., the object identity is passed by value). Note that all object expressions are typed by an interface, except this, which is typed by the enclosing class. In a well-typed program, we may assume that each call is annotated by the interface/class of the callee, as in $o.m_I(\ldots)$ where I contains a declaration of m.

Secrecy Levels. We enrich the typing system with *secrecy levels.* Secrecy levels range over $\mathcal{L}$ of basic secrecy descriptions with ordering $\sqsubseteq$, such that $(\mathcal{L}, \sqsubseteq)$ is a lattice, i.e., a partially ordered set with *meet* ($\sqcap$), *join* ($\sqcup$), a top element $\top$ and bottom element $\bot$. Higher in the lattice means more secure. A latice may be indexed by object identities for controlling access rights. This would be essential at runtime for controlling object secrecy; however, in our static analysis we will not use levels indexed by identities, since there is limited static knowledge about object identities.

In a program, all declarations of fields, formal parameters, and return values are given a secrecy level, with level Low as default (if none is specified). Local variables do not have a declared secrecy level; their level starts as Low but may change after each statement. At runtime, objects are assigned a secrecy level that protects against unauthorized changes. Such a protected part is typical in policy enforcement research [6]. The statically assigned level of a formal data parameter represents the maximal level of any actual parameter. The declared secrecy level of an object variable expresses the secrecy of the object identity, which is typically *low*, reflecting that object *identities* (as such) are considered non-secret, whereas the runtime secrecy level of an object gives more detailed information, for instance about the access rights of the object.

The static analysis is class-based, and therefore the analysis is based on the (statically) declared levels, and not the runtime object levels. However, the language allows specification of restrictions on the secrecy level of a new object (as in x:=new C():Low) which determines the initial runtime secrecy level of the generated object. At runtime an object generated by the statement x:=new C():l will get the level $l \sqcap l_{\mathsf{this}}$ where l_{this} is the level of the parent object. Note that $l \sqcap l_{\mathsf{this}} \sqsubseteq l_{\mathsf{this}}$, ensuring that the secrecy level of the generated object will not exceed that of the parent object. As an object encapsulates local data and fields, these are not accessible from outside of the object, and we do not need static control of write access to fields of an object. In a program, the runtime secrecy level of an object can be tested using the $\sqsubseteq$ operation.

In the static analysis, we consider all possibilities for levels that can be assigned at runtime. This allows us to detect a maximal secrecy level for each program variables at a given point in a program (see Sec. 3).

3 Secrecy-Type System

Our analysis is done class-wise, which is possible since remote access to fields is forbidden and since all object interaction is done by methods declared in an interface. This means that limitations on information flow between high and low variables (such as $v_{High} := v_{Low}$ and $v_{High} := v_{Low}$) are not needed. However, we rely on level information about fields before and after suspension, maintained in a way similar to a class invariant. The secrecy analysis of a class only depends on that class declaration, related interfaces, and the class parameter declaration of instantiated classes.

We assume a well-typed program and assume each method call $e.m(\ldots)$ is augmented by annotating the method name m by the interface of the callee e (as in $e.m_I(\ldots)$), or the enclosing class when e is this. The secrecy-type system for classes and methods are shown in Fig. 1. The confidentiality of a class definition Cl is formalized by judgments of the form

$$\vdash \ Cl \ \ \mathbf{ok}$$

expressing that the class definition obeys the confidentiality rules. And the confidentiality of a method definition M is formalized by judgments of the form

$$C \vdash M \ \mathbf{ok}$$

where C is the enclosing class. The confidentiality of a *statement* s is formulated by considering judgments of the form

$$C \vdash [\Gamma, pc] \ s \ [\Gamma', pc']$$

where Γ is a mapping binding variable names to confidentiality levels for a given program point, and pc is the confidentiality level of the current program point. As Γ and pc depends on the program point, we let the "pre-binding" $[\Gamma, pc]$ denote the bindings in the pre-state of s and the "post-binding" $[\Gamma', pc']$ those in the post-state of s. Moreover, for a class C we let the mapping Γ_C represent the *declared* secrecy levels of fields and class parameters, as given in the class definition, i.e., if the secrecy level of a field w is declared as l, the binding $w \mapsto l$ is included in Γ_C. The notation $\Lambda[I, m, i]$ denotes the level of the ith parameter of the method as *declared* in interface I, and similarly for classes. For a class C, we let C also denote the class constructor (initialization code). In contrast, Γ expresses confidentiality information depending on a particular program point. Since Γ-levels of class fields can increase and decrease, the type rules insist that at the end of each method (and at each suspension point) their resulting levels should not exceed the declared secrecy levels. This allows us to assume the declared levels at method start and after suspension.

Map notation. A finite mapping M is given by a set of bindings $z_i \mapsto value_i$ for a finite set of disjoint identifiers z_i, the *domain.* The empty map is denoted $\emptyset$. Map look-up is written $M[z]$. A map update, written $M[z \mapsto d]$, is the map M updated by binding z to d, regardless of

$$\textsc{(S-class)}\quad \frac{C \vdash\ M_i\ \mathtt{\mathbf{ok}}, \quad \text{for each } M_i \in \overline{M}}{\vdash \mathtt{\mathbf{class}}\ \ C(\overline{cp} : \overline{U})\{\overline{w} : \overline{U}';\ \ \overline{M}\}\ \mathtt{\mathbf{ok}}}$$

$$\textsc{(S-method)}\quad \frac{\begin{array}{c} C \vdash [\mathit{\Gamma}_C[\overline{y} \mapsto \mathcal{L}(\overline{U}), \overline{x} \mapsto \overline{\mathsf{Low}}], \mathsf{Low}]\ s\ [\mathit{\Gamma}, pc] \\ C \vdash [\mathit{\Gamma}, pc]\ e :: l' \qquad l' \sqsubseteq l \qquad \mathit{\Gamma}[\overline{w}] \sqsubseteq \mathit{\Gamma}_C[\overline{w}] \end{array}}{C \vdash T : l\ m(\overline{y} : \overline{U})\{\mathtt{\mathbf{var}}\ \ \overline{x} : \overline{T}; s; \mathtt{\mathbf{return}}\ \ e\}\ \mathtt{\mathbf{ok}}}$$

Fig. 1. SeCreol confidentiality type system for classes and methods where $\mathit{\Gamma}_C$ denotes the declared secrecy levels for class parameters and fields, in class C, and $\mathit{\Gamma}$ expresses confidentiality information at a particular program point.

any previous bindings of z. Similarly $M[S]$ denotes M updated with a set S of (disjoint) bindings. And the map composition $M + M'$ is the map M overwritten by M' (on the common domain).

According to Rule S-CLASS in Fig. 1, confidentiality of each class is satisfied, or simply is ok, if the confidentiality of each method is satisfied. The confidentiality of a method (see Rule S-METHOD) is satisfied if its body satisfies confidentiality, starting with the declared level bindings (for fields and class parameters, method parameters, and local variables) and with Low as starting pc level, and resulting in some binding $[\mathit{\Gamma}, pc]$ such that $\mathit{\Gamma}$ respects the declared field and class parameter bindings levels (i.e., $\mathit{\Gamma}[z] \sqsubseteq \mathit{\Gamma}_C[z]$ for each field/class-parameter z) and such that the returned value respects the declared output level of the method. As stated before, we check $\mathit{\Gamma}[z] \sqsubseteq \mathit{\Gamma}_C[z]$ because the secrecy level of program variables is allowed to be changed in different program points.

The SeCreol secrecy-type system for expressions and statements is shown in Fig. 2 and 3, respectively. These figures present typing rules describing which secrecy type is assigned to each occurrence of an expression and program variable. The confidentiality of expressions and right-hand-sides rhs, given in Fig. 2, are formulated by judgments of the form

$$C \vdash [\mathit{\Gamma}, pc]\ rhs :: l$$

where l is the resulting confidentiality level of rhs. The rules check that each occurrence of an actual parameter (or return value) respects the declared level of the corresponding formal parameter (or method return level), and that fields and class parameters respect the corresponding declared levels at suspension points and at method returns. In our formalization this is checked by premises in the rules; thus if these premises cannot be derived, the program will not satisfy the secrecy rules. Note that each statement may adjust $\mathit{\Gamma}$, but only **`if`** and **`while`** statements may affect pc. Thus the level of variables and pc may differ at different program points, which for example means that a call that is acceptable at one program point, might be unacceptable at another point.

Rule S-EXP states that the confidentiality of an expression e is achieved by $\mathit{\Gamma}[e] \sqcup pc$, where pc represents the context level of the current program

$$\text{(S-Exp)} \qquad C \vdash [\mathit{\Gamma}, pc]\; e :: \mathit{\Gamma}[e] \sqcup pc$$

$$\text{(S-New)} \qquad \frac{C \vdash [\mathit{\Gamma}, pc]\; e_i :: l_i \qquad l_i \sqsubseteq \mathit{\Gamma}_{C'}[cp_i]}{C \vdash [\mathit{\Gamma}, pc]\; (\textsf{\textbf{new}}\ C'(\overline{e}) : l) :: pc}$$

$$\text{(S-Call)} \qquad \frac{C \vdash [\mathit{\Gamma}, pc]\; e_i :: l_i \qquad l_i \sqsubseteq \mathit{\Lambda}[I, m, i]}{C \vdash [\mathit{\Gamma}, pc]\; e.m_I(\overline{e}) :: \mathit{\Lambda}[I, m] \sqcup pc}$$

$$\text{(S-SelfCall)} \qquad \frac{C \vdash [\mathit{\Gamma}, pc]\; e_i :: l_i \qquad l_i \sqsubseteq \mathit{\Lambda}[C, m, i]}{C \vdash [\mathit{\Gamma}, pc]\; \textsf{this}.m(\overline{e}) :: \mathit{\Lambda}[C, m] \sqcup pc}$$

Fig. 2. SeCreol secure-type system for expressions and right-hand-sides.

branch. Thus a low level expression occurring in a program branch with level *pc*, gets *pc* as its level, since it may reveal context information. We define $\mathit{\Gamma}[e]$ as follows: For a constant c (including null, this, *void*, and caller) $\mathit{\Gamma}[c]$ is Low (i.e., $\bot$), $\mathit{\Gamma}[e \sqsubseteq e']$ is High (i.e., $\top$), and for other kinds of expressions (including function applications) $\mathit{\Gamma}[e]$ is defined as $\sqcup_{v \in e} \mathit{\Gamma}[v]$, where v ranges over the variables textually occurring in e, and $\mathit{\Gamma}[v]$ is its level recorded in $\mathit{\Gamma}$. (For simplicity, we here ignore so-called sanitizer functions, i.e., special functions resulting in a lower level than an input.)

Moreover, object identities are not confidential, thus object variables are typically declared with a Low level. However, the level of such variables in $\mathit{\Gamma}$ is affected by the branch level *pc* as other program variables. Thus the resulting level of object creation is *pc* as object identities as such are considered Low. For the right-hand-side of a call or new construct, corresponding to the other rules in Fig. 2, each actual parameter is required to have a level not exceeding the declared level of the corresponding formal parameter. The resulting level of the call's right-hand-side is the declared return level of the method, joined with the current context level *pc*. We observe that $C \vdash [\mathit{\Gamma}, pc]\; rhs :: l \Rightarrow pc \sqsubseteq l$, which means the *rhs* level is always at least as high as *pc*. This can be easily proved by looking at each case of a right-hand-side *rhs* in the rules.

According to the secure-type system for statements in Fig. 3, a simple call does not change $\mathit{\Gamma}$ nor *pc*, but the actual parameter levels must respect the declared levels of the corresponding formal parameters (as above). And we have $C \vdash [\mathit{\Gamma}, pc]\; skip\; [\mathit{\Gamma}, pc]$. For an assignment, object creation statement, or call, $v := rhs$, with level l for rhs, the level of v in $\mathit{\Gamma}$ is changed to l, which could imply a downgrade or an upgrade (or no change) of level. The *pc* is not modified since such a statement is considered efficiently terminating without any branching.

For an **`await`** statement we must ensure that the declared levels of all fields and class parameters are respected, since the suspension may cause other processes to continue, for which we assume these declared levels. Levels of local variables will remain after an **`await`** statement

$$\text{(S-SIMPLE-CALL)}\quad \frac{C \vdash [\Gamma, \mathit{pc}]\; e.m_I(\overline{e}) :: l}{C \vdash [\Gamma, \mathit{pc}]\; e!m_I(\overline{e})\; [\Gamma, \mathit{pc}]} \qquad \text{(S-RHS)}\quad \frac{C \vdash [\Gamma, \mathit{pc}]\; \mathit{rhs} :: l}{C \vdash [\Gamma, \mathit{pc}]\; v := \mathit{rhs}\; [\Gamma[v \mapsto l], \mathit{pc}]}$$

$$\text{(S-COMPOSITION)}\quad \frac{\begin{array}{c} C \vdash [\Gamma, \mathit{pc}]\; s_1\; [\Gamma_1, \mathit{pc}_1] \\ C \vdash [\Gamma_1, \mathit{pc}_1]\; s_2\; [\Gamma_2, \mathit{pc}_2] \end{array}}{C \vdash [\Gamma, \mathit{pc}]\; s_1 ; s_2\; [\Gamma_2, \mathit{pc}_2]}$$

$$\text{(S-AWAIT)}\quad \frac{\begin{array}{c} C \vdash [\Gamma, \mathit{pc}]\; e :: \mathsf{Low} \\ \Gamma[\overline{w}] \sqsubseteq \Gamma_C[\overline{w}] \end{array}}{C \vdash [\Gamma, \mathit{pc}]\; \mathbf{await}\; e\; [\Gamma + \Gamma_C, \mathit{pc}]} \qquad \text{(S-AWAIT-CALL)}\quad \frac{\begin{array}{c} C \vdash [\Gamma, \mathit{pc}]\; \mathit{rhs} :: l \\ \Gamma[\overline{w}] \sqsubseteq \Gamma_C[\overline{w}] \end{array}}{C \vdash [\Gamma, \mathit{pc}]\; \mathbf{await}\; v := \mathit{rhs}\; [(\Gamma + \Gamma_C)[v \mapsto l], \mathit{pc}]}$$

$$\text{(S-IF)}\quad \frac{\begin{array}{c} C \vdash [\Gamma, \mathit{pc}]\; e :: l \\ C \vdash [\Gamma, l]\; s_1\; [\Gamma_1, \mathit{pc}_1] \\ C \vdash [\Gamma, l]\; s_2\; [\Gamma_2, \mathit{pc}_2] \end{array}}{C \vdash [\Gamma, \mathit{pc}]\; \mathbf{if}\; e\; \mathbf{then}\; s_1\; \mathbf{else}\; s_2\; \mathbf{fi}\; [\Gamma_1 \sqcup \Gamma_2, \mathit{pc}]} \qquad \text{(S-WHILE)}\quad \frac{\begin{array}{c} C \vdash [\Gamma_i, \mathit{pc}_i]\; e :: l_i \\ C \vdash [\Gamma_i, l_i]\; s\; [\Gamma_i', \mathit{pc}_i'] \\ \Gamma_{i+1} = \Gamma_i \sqcup \Gamma_i', \quad \mathit{pc}_{i+1} = \mathit{pc}_i \sqcup \mathit{pc}_i' \end{array}}{C \vdash [\Gamma_1, \mathit{pc}_1]\; \mathbf{while}\; e\; \mathbf{do}\; s\; \mathbf{od}\; [\mathit{FIX}_i(\Gamma_i), \mathit{pc}_1]} \quad i = 1, 2, \ldots$$

Fig. 3. SeCreol secure-type system for statements.

since local variables are not affected by other processes. We therefore use map composition ($+$) in the post-state of an await to overwrite the levels of fields and class parameters by the declared levels (Γ_C). For simplicity we consider only Low await conditions. In the case of a suspending call, the effect of the assignment part is added after the map composition since this assignment happens after suspension.

Rule S-IF lifts the *pc* level of each branch by the level of the test. This will make all expressions occurring in both branches at least as high as the if-test. Thereby implicit leakage is avoided. Since the static analysis does not know which branch is taken at runtime, the resulting value of Γ for each variable is calculated as the highest level of each branch. An **if** statement without an else-branch is like an **if** statement with skip in the else-branch. The treatment of **while** is similar to an **if** statement without an else-branch, except that the static analysis cannot predict how many times the branch is iterated. Each iteration may lift the levels in Γ or *pc*. However, a loop will have a finite number of program variables and since there is a finite number of static levels, there is a minimal fixpoint reachable in a finite number of approximations (typically i equal to one or two). Rule S-WHILE reflects this fixpoint calculation.

The secrecy typing ensures that there is no flow from high values to low values, and that values evaluated in an if-branch with a high test are high (since they may depend on the test), and similarly for values evaluated inside a while-loop with a high test. Thus the values of low variables in any program state do not depend on high inputs. Furthermore, this ensures that for each call (and return) generated by o the values of parameters

```
interface Passw{
 Nat:Low passw(Nat:High x)// store password, return a ref number
 Nat:High check(Nat:Low x)// check validity of password given ref
}
class PASSW implements Passw{ List[Nat]:High p:=empty; Nat n:=0;
 Nat passw(Nat:High x){p:=append(p,x); n:=n+1; return n}// return index
 Nat:High check(Nat x){Nat:High c:=0;
  if p ⊑ caller and 1≤x≤n then c:=p[x]fi;
  return c}//for High callers the value in p is returned (if any)
}
class TEST(Passw o){ Nat:High xh; Nat:Low xl;
 Nat:High test(Nat x){  xh ↦ High . Note: all others are Low
      xl := x;              xl ↦ Low
      x := o.check(x); x ↦ High
      xh := xl;             xh ↦ Low . Note: suspension is ok even with x high
      await true;        xh ↦ High, x ↦ High . Note: all others are Low
      xh := o.passw(x); xh ↦ Low . Note: the call is ok with x high
      await x:=o.check(xh); x ↦ High . Note: the call is ok since xh now is Low
      return x          Note: return is ok with x High, since xh ⊑ High ∧ xl ⊑ Low.
}}
```

Fig. 4. An example showing a password protection class and a test program. In the latter, level changes in fields and local variables are indicated to the right in each line.

declared as low do not depend on high inputs. We provide a proof of this in Sec. 5, based on a semantics that includes runtime secrecy levels.

Example. A small example is given in Fig. 4 to illustrate possible changes in the levels of fields (xh and xl) and local variables (x). The implementation of *Passw* uses an if-test to check $p \sqsubseteq$ caller before returning a high value in *check*. A test class with non-trivial secrecy typing is added. Here, level changes are written to the right of each line, not repeating unchanged information. The program satisfies the rules for confidentiality, i.e., the program does not leak information in its explicit output and respects field levels at return/await statements. Note that the lowering of xh was needed to make the *check* call allowed, that the higher level of the local variable x was maintained over the await (since x is local), that the higher level of x was acceptable in the *passw* call, and that the high level of x is allowed at the return point (after which x is deallocated).

4 Operational Semantics

The operational semantics is given in Fig. 5. We explain the main elements, while a more detailed explanation is given in the extended version

ASSIGN :
$$\begin{array}{rl} & o : \mathbf{ob}(\delta, v := e; \overline{s}) \\ \longrightarrow & o : \mathbf{ob}(\delta[v := e], \overline{s}) \end{array}$$

IF-TRUE :
$$\begin{array}{rl} & o : \mathbf{ob}(\delta, \mathtt{\textbf{if}}\ b\ \mathtt{\textbf{then}}\ \overline{s1}\ \mathtt{\textbf{else}}\ \overline{s2}\ \mathtt{\textbf{fi}}; \overline{s}) \\ \longrightarrow & o : \mathbf{ob}(\delta, pcs := push(pcs, l); \overline{s1}; pcs := pop(pcs); \overline{s}) \\ & \mathtt{\textbf{if}}\ \delta[b] = true_l \end{array}$$

IF-FALSE :
$$\begin{array}{rl} & o : \mathbf{ob}(\delta, \mathtt{\textbf{if}}\ b\ \mathtt{\textbf{then}}\ \overline{s1}\ \mathtt{\textbf{else}}\ \overline{s2}\ \mathtt{\textbf{fi}}; \overline{s}) \\ \longrightarrow & o : \mathbf{ob}(\delta, pcs := push(pcs, l); \overline{s2}; pcs := pop(pcs); \overline{s}) \\ & \mathtt{\textbf{if}}\ \delta[b] = false_l \end{array}$$

WHILE :
$$\begin{array}{rl} & o : \mathbf{ob}(\delta, \mathtt{\textbf{while}}\ b\ \mathtt{\textbf{do}}\ \overline{s1}\ \mathtt{\textbf{od}}; \overline{s}) \\ \longrightarrow & o : \mathbf{ob}(\delta, \mathtt{\textbf{if}}\ b\ \mathtt{\textbf{then}}\ \overline{s1};\ \mathtt{\textbf{while}}\ b\ \mathtt{\textbf{do}}\ \overline{s1}\ \mathtt{\textbf{od}}\ \mathtt{\textbf{fi}}; \overline{s}) \end{array}$$

NEW :
$$\begin{array}{rl} & o : \mathbf{ob}(\delta, v := \mathtt{\textbf{new}}\ C(\overline{e}) : l; \overline{s}) \\ \longrightarrow & o : \mathbf{ob}(\delta[v := o'], \overline{s}) \\ & o' : \mathbf{ob}(\delta_C[\mathsf{this} \mapsto o', \overline{cp} \mapsto \delta[\overline{e}]], init_C) \\ & \mathtt{\textbf{where}}\ \ o' = (fresh, o.level \sqcap l),\ \text{ for a globally fresh reference } fresh \end{array}$$

SIMPLE CALL :
$$\begin{array}{rl} & o : \mathbf{ob}(\delta, a!m(\overline{e}); \overline{s}) \\ \longrightarrow & o : \mathbf{ob}(\delta[\mathsf{nextId} := next(\mathsf{nextId})], \overline{s}) \\ & \mathtt{\textbf{msg}}\ o \to \delta[a].m(\delta[\mathsf{nextId}, \overline{e}]) \end{array}$$

CALL :
$$\begin{array}{rl} & o : \mathbf{ob}(\delta, [\mathtt{\textbf{await}}]\ v := a.m(\overline{e}); \overline{s}) \\ \longrightarrow & o : \mathbf{ob}(\delta, a!m(\overline{e}); [\mathtt{\textbf{await}}]\ v := \mathtt{\textbf{get}}\ \delta[\mathsf{nextId}]; \overline{s}) \end{array}$$

START :
$$\begin{array}{rl} & \mathtt{\textbf{msg}}\ o' \to o.m(u, \overline{c}) \\ & o : \mathbf{ob}((\alpha|\beta'), \mathbf{idle}) \\ \longrightarrow & o : \mathbf{ob}((\alpha|(\beta[\mathsf{caller} \mapsto o', \mathsf{callId} \mapsto u, \overline{y} \mapsto \overline{c}])), \overline{s}) \\ & \mathtt{\textbf{where}}\ m \text{ is statically bound to } (m, \overline{y}, \beta, \overline{s}) \text{ in the class of } \mathsf{this} \end{array}$$

RETURN :
$$\begin{array}{rl} & o : \mathbf{ob}(\delta, \mathtt{\textbf{return}}\ e) \\ \longrightarrow & o : \mathbf{ob}(\delta, \mathbf{idle}) \\ & \mathtt{\textbf{msg}}\ \delta[\mathsf{caller}] \leftarrow \delta[\mathsf{this}].(\delta[\mathsf{callId}], \delta[e]) \end{array}$$

QUERY :
$$\begin{array}{rl} & \mathtt{\textbf{msg}}\ o \leftarrow o'.(u, c) \\ & o : \mathbf{ob}(\ldots [\mathtt{\textbf{await}}]\ v := \mathtt{\textbf{get}}\ u\ \ldots) \\ \longrightarrow & o : \mathbf{ob}(\ldots v := c \ldots) \end{array}$$

AWAIT :
$$\begin{array}{rl} & o : \mathbf{ob}(\delta, \mathtt{\textbf{await}}\ b; \overline{s}) \\ \longrightarrow & o : \mathbf{ob}(\delta, \overline{s}) \\ & \mathtt{\textbf{if}}\ \delta[b] = true_l \end{array}$$

CONTINUE :
$$\begin{array}{rl} & o : \mathbf{ob}((\alpha|\beta'), \mathbf{idle}) \\ \longrightarrow & o : \mathbf{ob}((\alpha[PQ \mapsto rest])|\beta), \overline{s}) \\ & \mathtt{\textbf{if}}\ deq(\alpha[PQ], \alpha) = ((\beta, \overline{s}); rest) \end{array}$$

SUSPEND :
$$\begin{array}{rl} & o : \mathbf{ob}((\alpha|\beta), \overline{s}) \\ \longrightarrow & o : \mathbf{ob}((\alpha[PQ \mapsto enq(\alpha[PQ], (\beta, \overline{s}))], \varepsilon), \mathbf{idle}) \\ & \mathtt{\textbf{if}}\ \overline{s} \text{ starts with } \mathtt{\textbf{await}} \end{array}$$

Fig. 5. Operational rules reflecting small-step semantics of SeCreol with secrecy levels.

[16]. A runtime configuration of a system is a multiset of objects and messages (using blank-space as the binary multiset constructor). Each rule in the operational semantics deals with only one object o, and possibly messages, reflecting that we deal with concurrent distributed systems communicating asynchronously. When a subconfiguration $\mathcal{C}$ can be rewritten to a $\mathcal{C}'$, this means that the whole configuration $\ldots\mathcal{C}\ldots$ can be rewritten to $\ldots\mathcal{C}'\ldots$, reflecting interleaving semantics. Each object o is responsible for executing all method calls to o as well as self-calls. An object has at most one active process, reflecting a method execution, and a sequence of suspended processes organized in a process queue PQ. Remote calls and replies are handled by messages. Objects have the form

$$o : \mathbf{ob}(\delta, \overline{s})$$

where o is the object identity, δ is the current object state, and $\overline{s}$ is a sequence of statements ending with a **return**, representing the remaining part of the active process, or **idle** when no active process. A message has the form

$$\texttt{\textbf{msg}}\ o \to o'.m(\overline{e})$$

representing a call with o as caller, o' callee, and $\overline{e}$ actual parameters, or

$$\texttt{\textbf{msg}}\ o \leftarrow o'.(u, d)$$

representing a completion event where d is the returned value and u the identity of the call. The operational rules reflect small-step semantics. For instance, the rule for skip is given by $o : \mathbf{ob}(\delta, \texttt{\textbf{skip}}; \overline{s}) \longrightarrow o : \mathbf{ob}(\delta, \overline{s})$, saying that the execution of *skip* has no effect on the state δ of the object. A while loop is handled by expanding while b do s od to if b then s; while b do s od fi upon execution of the while-statement. The semantics of an if-statement without an else-part is equivalent to if b then s else skip fi.

The operational semantics uses some additional variables, like PQ for holding the process queue and nextId for generating unique identities for calls. These appear as fields in the operational semantics. Furthermore, this is handled as an implicit class parameter, while callId and caller appear as implicit method parameters, holding the identity of a call and its caller, respectively. The operational semantics uses an additional *query* statement, [**await**] **get** u, for dealing with the termination of call/await call statements. The query **get** u is blocking while waiting for the method response with identity u, and **await get** u is a suspending query.

The state of an object is given by a twin mapping, written $(\alpha|\beta)$, where α is the state of the field variables (including PQ, nextId) and class parameters $\overline{cp}$ (including this), and β is the state of the local variables and formal parameters (including callId and caller) of the current process. Look-up in a twin mapping, $(\alpha|\beta)[z]$, is simply given by $(\alpha + \beta)[z]$. The notation $\alpha[z := e]$ abbreviates $\alpha[z \mapsto alpha[e]]$, and the notation $(\alpha|\beta)[v := e]$ abbreviates **if** v *in* β **then** $(\alpha \mid \beta[v \mapsto (\alpha|\beta)[e]])$ **else** $(\alpha[v \mapsto (\alpha|\beta)[e]] \mid \beta)$, where *in* is used for testing domain membership.

The *process queue* PQ is the queue of suspended processes, of form $(\beta, \overline{s}))$. The operations $enq(PQ, p)$ and $deq(PQ, \alpha)$ are used to add a process p to the queue, and to select an *enabled* process (if any) from the queue, respectively. The latter results in the sequence $(p; PQ')$ of the selected enabled process p and the remainder of the queue PQ' (depending on the specific scheduling policy), or the empty sequence *empty* if no process is enabled. A process $(\beta, \overline{s})$ is *enabled* if it starts with an enabled statement. A conditional **await** is enabled if the condition evaluates to true (in state $\alpha|\beta$), and an **await** call statement is not enabled unless reduced by the QUERY rule. All other statements are enabled.

The given language fragment may be extended with constructs for local (stack-based) method calls, e.g., by using the approach of [12].

Runtime Secrecy Levels. We here explain the secrecy aspects of the operational semantics. We assume a program that has passed the secrecy typing, and therefore the operational semantics does not include explicitly checks for confidentiality errors during reduction. However, we prove that any secrecy level obtained at runtime is less or equal to the one calculated by the static secrecy typing. This property, called *secrecy soundness*, is formalized in the next section. This guarantees that the static secrecy level checks will be satisfied at run-time, even when based on the runtime secrecy levels. And non-interference is then proved.

At runtime the evaluation of an expression e gives a secrecy tag l, in addition to a (normal) value d. We let the tagged value d_l denote this result, and let c denote tagged values. We let $d_l.tag$ be l. If this value is assigned to a program variable v, the binding $v \mapsto d_l$ is added to the state. The state of an object is given by a twin-mapping as above, but the values of variables are now bound to tagged values. Thus the values appearing in the extended semantics are all tagged. Each object identifier has the form of a pair (oid, l) where oid is a normal object identifier and l is the secrecy level of the object. We refer to the secrecy level of an object o by the meta-notation $o.level$, letting $(oid, l).level$ be l. For data values c, we define $c.level$ by $c.tag$. The secrecy semantics uses an additional variable pcs in each method, reflecting the context secrecy level of enclosing if- and while-branches. (pcs is local since it must be retrieved after suspension.) And pcs is a stack of levels reflecting the levels of the enclosing if- and while-branches, such that the top of the stack is the innermost branch.

The evaluation of an expression e in a state δ is denoted $\delta[e]$, where the value is evaluated ignoring tags, and the tag is defined by $\mathit{level}(\mathit{pcs}) \sqcup_i v_i.\mathit{tag}$, where $\sqcup_i v_i.\mathit{tag}$ is the join of the tags of all variables occurring in e, and where $\mathit{level}(\mathit{pcs})$ is the join of all levels in the stack pcs. This assumes strictness of all functions in the language, i.e., the level of $f(\overline{c})$ is simply $\sqcup_i c_i$. The special expression $e \sqsubseteq e'$ is evaluated by $\delta[e].\mathit{level} \sqsubseteq \delta[e'].\mathit{level}$ and with tag as defined above. (Other kinds of non-strict functions are for simplicity ignored here.) The runtime secrecy level of a variable v in an execution state will be less or equal to that of the static level in a corresponding program point. There are several reasons for this. For instance, there can be many calls to the same method, some with actual parameters of less secrecy level than for other calls. And at the start of a method, the static analysis will assume the declared secrecy level for fields, whereas at runtime the levels might be less. Similarly, any expression may have a lower level at runtime since the variables involved might have a lower level than in the static analysis.

5 Theoretical Results

In order to relate runtime states to those of the static secrecy typing, we use statement labels. Following [15], each basic statement and each if- and while-condition in a given program is tagged by a unique statement label (i.e., statement number) n appearing as a superscript (when needed).

The result of the secrecy analysis can be captured by a mapping SL (Static Level) such that $SL(C,n)$ gives the binding environment of the pre-state of statement n in class C. Thus $SL(C,n)[v]$ is the level statically assigned to variable v in this state by the secrecy typing analysis, and $SL(C,n)[\mathit{pc}]$ is the level statically assigned to pc in this state. If an execution reaches a configuration where a C object is about to execute a basic statement s^n, and similarly for another execution, we say that the two pre-states of n are low equal if the values of all variables v such that $SL(C,n)[v]$ is Low are equal in the two pre-states.

In the operational semantics, the level information at time t (i.e., the number of execution steps) in an execution is captured by a function $RT(t)$ returning the executing object (of form $o : \mathbf{ob}(\delta, \overline{s})$) such that $RT(t).\mathit{class}$ is its class, and $RT(t).\mathit{label}$ is the label of the statement to be executed, and $RT(t)[v]$ is the secrecy level of variable v, i.e., the level of $\delta[v]$. Similarly, $RT(t)[\mathit{pc}]$ is the level of pc in this state, and $RT(o)[\mathit{pcs}]$ is the level of the stack pcs given by $\sqcup_i \mathit{pcs}[i]$ where i ranges over all indexes in the stack. The following theorem ensures that the evaluation of variables and expressions at runtime gives levels that are less or equal to those of the static analysis.

Theorem 1 (Soundness). *At any time t in an execution where the active object $RT(t)$ is of the form $o : \boldsymbol{ob}(\delta, s^n; \overline{s})$ of class C, then*

(i) the levels of δ are less or equal to the corresponding ones in $SL(C, n)$, i.e., $\delta[v] \sqsubseteq SL(C, n)[v]$ for all program variables v and $level(\delta[pcs]) \sqsubseteq SL(C, n)[pc]$.

(ii) if $C \vdash [\Gamma, pc]\ e :: l$ and $\delta[e] = d_{l'}$ for an expression e, then $l' \sqsubseteq l$.

Proof. We use induction on the time t, and may assume that the conclusion holds up to a given time t and must ensure that it holds in the next state. We first note that (i) implies (ii) because the static level of an expression e is given by the join of the static levels of all variables in e and of *pc*, whereas the runtime level of e is given by the join of the runtime levels of all variables in e and of $level(\delta[pcs])$. By (i) the latter cannot exceed the former since the runtime level of each variable v cannot exceed the static level of v, and since the runtime level of *pcs* cannot exceed the static level of *pc*.

It remains to show that (i) holds in the next state. Consider all basic statements that modify the state (of the active object). For an assignment $v := e$ the new runtime level of v is the runtime level of e evaluated in the current state. This level is less than the static level of e by (ii), thus the conclusion holds in this case. Similar arguments apply to all assignment-like statements, such as new and call statements, in which cases the assignment to the implicit and unobservable object variable *nextId* is unproblematic. The operational rules for skip and return give no state change. The operational rules for continue and suspend give a twin state where fields are not changed. In the case of suspend, the local state is empty (ignoring the PQ which is not a program variable), and in the case of continue, the local state is reset to an old state, for which we may use the induction hypothesis. The rules for if and while give a next state (after evaluating the condition) that is the same as before except that the *pcs* level may be raised. We need to show $level(\delta[pcs]) \sqsubseteq SL(C, n)[pc]$. This follows by (ii) since the condition is evaluated in the object state of time t. The discussion of the rule for await is similar. □

In our context of message-based systems, we define non-interference by:

Definition 1 (Non-interference). Non-interference *means that if two executions reach the pre-state of a basic statement s^n with configurations*

$\mathcal{C}_1$ and $\mathcal{C}_2$, respectively, such that $\mathcal{C}_1 =_{Low} \mathcal{C}_2$, then the observable output resulting from execution of s^n on the two configurations, will be the same.

The output *of a basic statement s is the message (**msg**) generated by the operational rule for s, if any, and otherwise empty. The* observable *part of a message is the values of parameters/method results declared as* Low *in the method declaration (as detected by the secrecy-type analysis).*

Theorem 2 (Non-interference). *A program that is secrecy-type correct will satisfy non-interference.*

Proof. We consider all basic statements. The ones generating output are the call statements and the return statement. The output of a call statement is given by the rule for SIMPLE CALL, and the observable output is the values of the parameters of m for which the declared level is Low. Since this parameter information is static, the sublist of Low parameters have the same length for two executions. Consider a call statement with label n of a given class C. Each parameter expression e_i of a low parameter has a static level l, which by Theorem 1 must be less than the runtime level l' of the evaluation of $RT(t)[e_i]$ for any execution at time t, where $RT(t)$ has an active object of the given class and with label n. Since the states of the two executions are low equal, the values of any expression with a low runtime label must be the same since only low variables are used on the evaluation (otherwise the runtime label could not be low). Therefore the value of each such e_i must be the same in the two executions. Similarly, the values of any return expression e evaluated in different pre-states of the same statement n are equal if the resulting runtime level is low, provided the two pre-states are low equal. Since static low level implies runtime low level, the two pre-states give the same observable output. The above discussion applies also to object identities since the only observable relation over object identities with low output is equality.

The argument above can be extended to **new** statements and any basic statement. It follows that the new state of all variables is low equal for two executions after a basic statement since each basic statement is deterministic (apart from generated object identities). Thus we have also shown that low equality of states is preserved by all basic statements. □

Note that the code if b then o!m1() else o!m2() fi leaks the outcome of the if-test to object o. To deal with such implicit leakage, one may define a stronger notion of non-interference involving communication events. This is studied in [17] defining *interaction non-interference* and showing that this can be enforced by static analysis involving communication traces.

6 Related Work

A number of complications arise from the different concurrency and communication models [19,3]. For imperative concurrent programs, the multithread, shared variable, and channel-based paradigms have been studied [18]. These paradigms give non-trivial privacy challenges. For instance the channel paradigm gives intricate timing leaks, based on observations of channel size [18,4]. In our paradigm, an object's process queue and queue of incoming calls are encapsulated and are non-observable (as well as their size). There are several works on static checking of noninterference for active objects communicating by asynchronous methods, including [10,11] and work based on [9], but with different goals, assumptions, and results ([9] with other forms of noninterference). Kammüller [11] considers a functional language with futures, with a different treatment of methods. To preserve confidentiality, we have considered Multilevel Security(MLS) which is a well-established concept for confidentiality while the goal of multilateral security in [10] is useful to satisfy complex and very different sets of policies in distributed computer systems. The multilateral security of [10] is relevant for our operational semantics. In our setting, instead of the traditional concept of public and private methods in [11], we use interfaces to control visibility of methods. Moreover, our approach is not dependent on the concept of futures. In addition, in [11] remote method calls are considered side-effect free which guarantees that no information from the caller side is leaked. Therefore, although secure down-calls are supported in [11], interaction noninterference is not preserved.

Our paradigm is based on a simple, compositional semantic model, which gives flexible analysis of program variables, including fields and communicated values, and of synchronization mechanisms, thereby reducing the amount of false positives. Scheduling-related primitives are included in our high-level language; this enables further static analysis than in [3]. Compared to [3], we consider more high-level concurrency constructs such as asynchronous calls and suspension mechanisms. A complementary work on SeCreol [17] focuses on indirect leakage caused by observations of network traffic, where enforcement of network-level non-interference is handled by means of static trace analysis. It assumes a similar secrecy typing system, but without including an operational semantics with secrecy levels nor a soundness proof of the secrecy typing.

While most of the related work aim at preventing traditional progress-insensitive non-interference, we are considering progress-sensitive non-interference, where an attacker can indirectly observe the progress of an object, caused by e.g. process termination or suspension (assuming termination proofs of while loops). Another aim of that paper is minimizing the Trusted Computing Base (TCB) by not trusting the compiler and using Proof-Carrying Code (PCC). Moreover, [3] and [11] prevent all flows from secret to public variables, while in our setting this is not necessary. In addition, for explicit flows, we also consider interaction between objects such as if secret then call fi for different method calls.

Dynamic checking of runtime access control, which has been done in the Java virtual machine and the .NET runtime systems, provides useful guarantee especially in the application of dynamic code involvements like mobile code. For example, in [2] static permissions are assigned to classes based on code origin, and when untrusted code calls trusted code, then the permission is checked using the run-time stack, while our approach is static. However, we aim at an extension to runtime checks in future work.

7 Conclusion

We have considered a model for concurrent object-oriented systems suitable for distributed service-oriented systems. The concurrent objects may communicate confidential and non-confidential information, restricting confidential information to method parameters/returns declared as safe for confidential information. The language is high-level and includes process control and suspension, without explicit signaling and locking operations. Objects are imperative and non-deterministic. We introduce a type and effect system and prove a noninterference property, as well as soundness of the secrecy typing system. Due to hiding and encapsulation, we do not impose unnecessary restrictions on information flow inside objects. The language has a compositional semantics and supports compositional program reasoning [5]; and the process control mechanisms include primitives typically part of an operating system. This allows class-wise secrecy analysis that goes beyond what is normally possible by static checking. The absence of futures simplifies the analysis. As shown in a complimentary work [16], one can deal with implicit leakage caused by network level observations of observable aspects of communicated messages.

The Creol concurrency model is adopted by the ABS language [14], and the work here can be extended to ABS by considering *object groups*, which impose concurrency restrictions, and *futures*, which may give rise to implicit information leakage. We have presented a more high-level language without (explicit) futures and object groups, which simplifies the formalization. We are initiating an implementation based on a Creol interpreter in Maude. The ABS tool support will be used for an ABS implementation.

References

1. A. Askarov, S. Hunt, A. Sabelfeld, and D. Sands. Termination-insensitive noninterference leaks more than just a bit. In *Proc. 13th European Symposium on Research in Computer Security: Computer Security*, ESORICS, pages 333–348. Springer, 2008.
2. A. Banerjee and D. A. Naumann. Stack-based access control and secure information flow. *Journal of functional programming*, 15(02):131–177, 2005.
3. G. Barthe, T. Rezk, A. Russo, and A. Sabelfeld. Security of multithreaded programs by compilation. In *European Symposium on Research in Computer Security*, pages 2–18. Springer, 2007.
4. D. Devriese and F. Piessens. Noninterference through secure multi-execution. In *Security and Privacy (SP), 2010 IEEE Symposium on*, pages 109–124. IEEE, 2010.
5. C. C. Din, J. Dovland, E. B. Johnsen, and O. Owe. Observable behavior of distributed systems: Component reasoning for concurrent objects. Journal of Logic and Algebraic Programming, 81(3):227–256, 2012.
6. U. Erlingsson. *The Inlined Reference Monitor Approach to Security Policy Enforcement.* PhD thesis, Cornell University, Ithaca, NY, USA, 2004. AAI3114521.
7. J. A. Goguen and J. Meseguer. Unwinding and inference control. In *IEEE Symposium on Security and Privacy*, pages 75–75, 1984.
8. N. Heintze and J. G. Riecke. The SLam calculus: Programming with secrecy and integrity. In *Proc. of POPL '98*, pages 365–377. ACM, 1998.
9. S. J. Hodges and C. B. Jones. Non-interference properties of a concurrent object-based language: Proofs based on operational semantics. In *Object orientation with parallelism and persistence.* Kluwer Academic Publishers, p. 1-22. 1996.
10. F. Kammüller. A Semi-Lattice Model for Multi-Lateral Security. Data Privacy Management DPM'12, ESORICS. p. 118-132, LNCS vol. 7731, Springer, 2012.
11. F. Kammüller. Confinement for Active Objects. International Journal of Advanced Computer Science and Applications (IJACSA), 6(2), 2015.
12. E. B. Johnsen and O. Owe. An asynchronous communication model for distributed concurrent objects. *Software and Systems Modeling*, 6(1):35–58, Mar. 2007.
13. E. B. Johnsen, O. Owe, and I. C. Yu. Creol: A type-safe object-oriented model for distributed concurrent systems. *Theoretical Computer Science*, 365(1–2):23–66, 2006.
14. E. B. Johnsen, R. Hähnle, J. Schäfer, R. Schlatte, M. Steffen. ABS: A Core Language for Abstract Behavioral Specification. Formal Methods for Components and Objects (FMCO 2010), *LNCS vol. 6957*, pages 142-164, Springer, 2010.
15. F. Nielson, H.-R. Nielson, and C. L. Hankin. *Principles of Program Analysis.* Springer, 1999.
16. O. Owe and T. Ramezanifarkhani. Static enforcement of confidentiality of interactions in concurrent object-oriented systems. Technical report, Dept. of Informatics, Univ. of Oslo, Norway, 2017. (An extended version of this paper.) `http://heim.ifi.uio.no/olaf/Papers/SeCreolReport.pdf`
17. T. Ramezanifarkhani, O. Owe, and S. Tokas. A secrecy-preserving language for distributed and object-oriented systems. *submitted*, March 2017.
18. A. Sabelfeld and H. Mantel. Static confidentiality enforcement for distributed programs. In *Static Analysis - International Symposium, SAS 2002*, volume 2477 of *LNCS*, pages 376–394. Springer, 2002.
19. A. Sabelfeld and A. C. Myers. Language-based information flow security. *IEEE Journal on Selected Areas in Communications*, 21:5–19, Sept. 2003.