Using Wrappers to Ensure Information-Flow Security for Active Objects with Futures

Farzane Karami
farzanka@ifi.uio.no
Department of Informatics
University of Oslo

Olaf Owe
olaf@ifi.uio.no
Department of Informatics
University of Oslo

Gerardo Schneider
gerardo@cse.gu.se
Dept. of Computer Science and Eng.
University of Gothenburg

ABSTRACT

This paper introduces a run-time mechanism for preventing leakage of secure information in distributed systems. We consider a general concurrency model suitable for designing service-oriented and distributed systems. In this model, concurrent objects communicate by asynchronous method calls and futures. The aim is to prevent leakage of confidential information to low-level viewers. The approach is based on the notion of a wrapper enclosing an object or a component, controlling its interactions with the environment. A wrapper is a mechanism added by the run-time system to provide protection of an insecure component according to some security policies. The security policies of a wrapper are formalized based on a notion of security levels and dynamic information-flow enforcement. At run-time, future components will be wrapped upon need, while only objects of *unsafe classes* will be wrapped, using static checking to limit the number of unsafe classes and thereby reducing run-time overhead. We define an operational semantics and prove that non-interference is satisfied.

Wrappers provide flexibility because they may be added or removed upon need, and they give separation of concerns by allowing the business code to be separated from the security control code. A service provider may use wrappers to protect its services in an insecure environment, and vice-versa: a system platform may use wrappers to protect itself from insecure service providers.

KEYWORDS

active objects, futures, information-flow, security, wrapper, non-interference, dynamic analysis, static analysis, distributed systems

ACM Reference Format:
Farzane Karami, Olaf Owe, and Gerardo Schneider. 2019. Using Wrappers to Ensure Information-Flow Security for Active Objects with Futures. In *Proceedings of PPDP Conference (PPDP'19).* , 15 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Given the large number of users and systems involved in a distributed system, security is a critical concern. It is thus essential to analyze the confidentiality of a system and control how information propagates between nodes. Though *access control* techniques can be used to control access to information, it does not control how the program handles the information. A program might leak secure information ("high") to public ("low") data disclosing it, or send the private information to malicious nodes. *Language-based* techniques, in particular *program analysis*, provide means to specify and enforce security policies for preventing leakage of secure information. *Information-flow controls* track how information propagates through the program during execution. The semantic notion of information-flow security is based on *non-interference* [10]. This means that in any two executions, where a program is run with different secret inputs but the same public values, the public outputs will be the same (at least for deterministic programs). This way, an attacker cannot see any difference between these two executions since public outputs are independent of the secret inputs. Attackers are assumed to be able to observe public information.

In this paper, we are interested in service-oriented and object-oriented systems, where distributed and concurrent objects communicate through asynchronous method calls. *Active object languages* are concurrent programming languages suitable for designing such systems. The goal is to design an efficient, permissive, and precise security mechanism that can be applied to these programming languages, supporting concurrency and communication paradigms like *futures*. A future is a read-only placeholder, which is created as the result of a remote method call and eventually contains the corresponding return value [3]. Therefore, the caller needs not block while waiting to get the return value: it can continue with other tasks and later get the value from the corresponding future. A future as a handler can be passed to other objects, that is they are *first-class futures*. In this case, any object that has a reference to the future can access its content, which may be a security threat as the future might contain highly sensitive data. Futures offer a flexible way of communication and sharing results, but handling them appropriately, in order to avoid security and privacy leakages, is challenging [15].

In this paper, our security mechanism is based on the notion of *wrappers* [18]. A wrapper wraps a component or subsystem at run-time and controls their interactions with the environment. We consider a concurrent core language, supporting high-level active objects, including non-blocking calls and futures, and use wrappers to control the objects and futures. In our core language, fields are encapsulated by objects and remote access is forbidden. Illegal flows may happen inside an object, such as assigning high values to low variables, and they are considered harmless in our setting since they are not observable by attackers as long as there is no illegal output from the object, such as sending high information to a remote object

l := **true**; **if** h **then** l := **false else skip fi**

Figure 1: Implicit flow.

with low protection. Therefore, only object interactions (through calls and return values) are controlled. Wrappers can be used to ensure non-interference in object interactions. They can also protect futures if they contain high return values, preventing illegal access by lower level objects.

The security policies and semantics of a wrapper are defined based on a combination of run-time security levels and dynamic information-flow enforcement. Our dynamic approach guarantees some levels of permissiveness and is precise since it deals with the exact run-time security levels. The use of wrappers and its security semantics are added to the run-time system for our core language. We define an operational semantics in the style of Structured Operational Semantics (SOS), modeled in the Maude system [9], giving an executable interpreter for programs written in our language. The approach gives a guarantee of non-interference, at the cost of some overhead due to the presence of the wrappers. In order to have less run-time overhead, we suggest a static analysis to identify where security checking and wrappers are needed since often only a few methods deal with secure information. Assuming a sound static analysis, we prove that our proposed security mechanism ensures the non-interference property in object communications.

In summary, our contributions are: i) a high-level core language as a model for service-oriented and distributed systems communicating through asynchronous method calls eventually using futures (Section 3.2), ii) a built-in notion of wrappers for enforcing non-interference and security control in object interactions in this model (Section 4), for which we use static analysis to reduce the run-time overhead (Section 4.1), and provide an executable operational semantics for the core language and wrappers (Sections 4.2-4.3), iii) a proof that our model satisfies non-interference (Section 5).

2 BACKGROUND

2.1 Information-flow security

Information-flow control approaches enforce security policies and prevent leakage of sensitive data during program execution. In principle, there are two kinds of leakage of information, namely *explicit* and *implicit flows* [21]. For simplicity, we assume two security levels *Low* and *High*. An explicit flow is direct copying of a high variable to a low one ($l := h$), where l and h are variables that store public and secret values. However, in an implicit flow, there is an indirect flow of information due to control flow structures. That is, an implicit flow happens when there is a low computation in a conditional or loop with a high guard (when the guard involves a high variable). For example, in the **if** statement of Figure 1, the value of h affects the value of l indirectly.

Information-flow control techniques are divided into two categories, static and dynamic approaches [20]. *Static information-flow* analysis, as in the Denning and Denning style [8], is performed by type checking at compile time (static type system). In this approach, the types of all program variables and expressions are extended with annotations for the different security levels. A compiler reads and tracks these security levels and makes sure that there is no leakage of information based on specified security policies. Security policies specify how the data variables can be used in a way such that there is no leakage of information. For example, each assignment is checked to ensure that the level of the assigned variable is high if there is a high variable in the right-hand-side (tracking explicit flows) or if the assignment occurs in a conditional or loop with a high guard (tracking implicit flows). In order to avoid implicit flows, a program-counter label (*pc*) is introduced, which captures the program context security level [21]. For instance, inside the **if** statement in Figure 1, since the guard is *High*, *pc* becomes *High* (a high context). Accordingly, in the branches, an assignment is secure if the security level of the assigned variable is greater than or equal to *pc*. By using this technique, the program in Figure 1 would be rejected. Static analysis is conservative: in order to be sound, it makes a worst case over-approximation of the security levels of variables (for example, it over-approximates a variable to *High*, while at run-time it can be *Low*). This causes unnecessary rejections of programs, especially when the complete program is not statically known, as is usually the case in distributed systems. On the other hand, static analysis has the benefit of less run-time overhead, since all security checks are performed before program execution [20].

Dynamic information-flow techniques may perform security checks similar to static techniques, at run-time. Dynamic techniques introduce a run-time overhead, but they are more permissive and precise since they deal with the exact security levels of variables instead of an over-approximation. Another example to illustrate permissiveness of dynamic techniques is the program

if $l < 10$ **then** $l = 1$ **else** $l = h$ **fi**

A traditional static type system [25], rejects this program as insecure due to the presence of the explicit flow $l = h$ (alternatively, a static analysis may over-approximate l as *High* after the **if** statement.) However, a dynamic technique accepts executions of the program when $l < 10$ holds.

In what follows we briefly explain some of the terminologies of information-flow security that we use in this paper:

Security levels. Variables are tagged with security levels organized in a lattice, a partially ordered set of security levels by the relation $\sqsubseteq$ with join ($\sqcup$), meet ($\sqcap$), a top $\top$ and bottom $\bot$ elements. The join operator returns the least upper bound of two given levels. The policy rules describe the flow of information considering these security levels. Inside a class, declarations of fields, class parameters, and formal parameters may have statically assigned security levels (with level *Low* as default if none is specified). In flow-sensitive static analysis, these variables start with their assigned levels, but the levels may change after each statement according to static information-flow control. Static checking is used to see if any actual parameter (or return value) of a call or object creation statement may possibly have a high security level according to the chosen static policy, if so the class is said to be *unsafe* as explained in more details in Sections 4-4.1. At run-time, objects are assigned security levels as well, and precise levels are evaluated during execution for objects of unsafe classes.

Flow-sensitivity. By *flow-sensitivity*, we mean that the analysis keeps track of the security levels of variables and updates them for each assignment. Let Γ be a function mapping variables to security levels, $Var \to \{Low, High\}$, then Γ might change during program execution. For each assignment, a flow-sensitive technique assigns a variable to high, if the assignment is in a high context ($pc = High$) or if there is a high variable in the right-hand-side. It assigns the level of the variable to low, if the assignment does not occur in a high context, and there are no high variables in the right-hand-side of the assignment. Otherwise, the security level of a variable does not change [20]. On the other hand, in a *flow-insensitive* analysis variables are assigned security levels at the beginning of the execution and this assignment does not change during the execution. For example, in the method body:

$$h := 10;\ \mathtt{if}\ h > 0\ \mathtt{then}\ l := 1\ \mathtt{fi}\ ;\ \mathtt{return}\ l$$

a flow-insensitive analysis may reject the program if the method is supposed to return a low value (since there is an assignment to a low variable in a high context). However, it is accepted by a flow-sensitive analysis since the security level of h is relabeled to low after the first assignment. In an assignment $x := e$, the security level of x changes to the join of the pc and the security level of the expression e. The security level of e is the join of the security levels of the variables that appear in the expression. By this security semantics, variables are instrumented with security levels, which are propagated along with ordinary values.

In [20], it is shown that although a flow-sensitive dynamic technique is permissive, it is unsound because of implicit flows. To overcome the implicit flows when branching on conditionals or loops with high guards, not only the security level of variables in the taken branch, but also the ones in the untaken branch, get updated to high [20]. In our dynamic information-flow enforcement, the same approach is applied to avoid implicit flows. For instance, if we consider an initial environment $\Gamma = \{h \mapsto High, l_1 \mapsto Low, l_2 \mapsto Low\}$ and the program:

$$\mathtt{if}\ h\ \mathtt{then}\ l_1 := 1\ \mathtt{else}\ l_2 := 0\ \mathtt{fi}$$

when the condition is true, the final typing environment should end up as $\Gamma = \{h \mapsto High, l_1 \mapsto High, l_2 \mapsto High\}$ for a sound flow-sensitive analysis. Whereas a naive flow-sensitive dynamic analysis, which only updates the taken branch variables, would end up with $\Gamma = \{h \mapsto High, l_1 \mapsto High, l_2 \mapsto Low\}$.

3 ACTIVE OBJECT LANGUAGES & FUTURES

Active object languages have received a lot of attention in recent years. These are programming languages for concurrent and distributed systems that are inspired by the *actor model* [1] and support asynchronous method calls, as well as standard object-oriented programming features. There are a number of active object languages including ABCL [28], Rebeca [23, 24], Scala/Akka [11, 26], Creol [14], ABS [13], Encore [5], and ASP/ProActive [6, 7]. Active objects communicate through asynchronous method calls. In the call/return paradigm without futures, both the method call and the corresponding return value are transmitted by message passing between the caller and callee objects. An object has an external queue for receiving method call requests and return values from other objects. In a naive model, a caller waits while the callee performs the call, which is a blocking call and undesirable. Therefore non-blocking call mechanisms are needed.

```
 1 interface DataBase { // Medical records
 2   MedData init(); // MedData is defined as a data type.
 3   Void modify();
 4 }
 5 interface Service {
 6   Unit process(Fut[MedData] fd);
 7 }
 8 class Server implements Service{
 9   Unit process(Fut[MedData] fd) {
10     MedData rd = fd.get;
11     rd.modify(); }
12 }
13 { // main block
14 Service s = new Server();
15 DataBase d = new DataBaseClass();
16 Fut[MedData] fd = d!init();
17 s!process(fd);
18 }
```

Figure 2: A simple ABS example [4].

The notion of futures is a common mechanism in active object languages for avoiding blocking calls [4, 15]. Futures are objects with a promise that a return value will become available at some point in the future [12, 27]. When a remote method call is made, a future object with a unique identity is created. A caller may continue with other processes while the callee is computing the return value. A callee sends back the return value to the corresponding future object. The future is then said to be *resolved.* This language construct allows a client program with access to the future object to continue its computation and wait only when one needs to fetch the value of the future object. In the case of first-class futures, a future identity can be passed to objects desiring the return value of the corresponding method call, even before the value is computed. Thus, a future can be distributed to many active objects in a system. In general, the future mechanism makes a language flexible and expressive; however, they may give rise to deadlock of one or more objects when these are waiting for a future that never will be resolved.

Futures can be either explicit with a specific type and access operations like ABS, or it can be implicit with automatic creation and synchronization. An explicit future in ABS is created as $Fut[T]\ f := o!m(\overline{e});\ \ldots;\ v = f.\mathbf{get}$, where f is a future variable and T is the type of the future value. The symbol "!" indicates an asynchronous method call (m) of object o, and the future value is retrieved with a **get** construct. Implicit futures are created by asynchronous and remote calls, and future values are accessed automatically without explicit constructs. In fact, when a future value is needed the only way is to block until the future is resolved; this access mechanism is called *wait-by-necessity* like in ASP and ProActive.

We explain the concurrency model of ABS-like active objects by means of the code in Figure 2, which is a simplified version of an example from [4]. In the main block of the code (lines 14-17), two tasks are started independently, by means of asynchronous calls to *init* and *process*, respectively. In line 14, a new object is created from the class *Server* of interface type *Service*. Objects are typed by interfaces because ABS supports behavioral interface specification.

Therefore, a pointer to an object is typed by an interface limiting the methods available on that pointer to the methods declared in the interfaces. Moreover, fields are not accessible via interfaces. In line 16, a future is created for the result of a call to *init* and immediately passed as a parameter to object *s* in line 17. In the *process* body, the future is retrieved, and there is a synchronous call to *modify*, which blocks the caller until it returns.

3.1 Information-flow security with futures

Information-flow analysis for a future-free language like Creol can be performed by static declaration of security levels for each input parameter and result value of a method, like the approach proposed in [17]. In that paper, the considered language does not support futures, and the security checks are done at static time, rather than at run-time. According to [15], the static analysis would be difficult when allowing futures as parameters. Future variables give a level of indirectness in that the retrieval of the result of a call is no longer syntactically connected to the call, compared to future-free languages. For instance, when a future is received as a parameter, it may not statically correspond to a unique call statement. One may overestimate the set of call statements that correspond to this given future parameter, but it requires access to the whole program. Moreover, these calls are not uniform with respect to security levels. In this case, one must consider the worst case possibility (i.e., the highest security level) for the set of possible corresponding call statements. It is conservative and will easily lead to security levels inflation, something which is not desirable since it would severely limit statically acceptable information passing and call-based interaction, or require dynamic checking.

During static analysis there is limited static knowledge about object identities. Therefore a language may compensate by including a syntactic construct for testing security levels, as in [17]. By testing relevant security levels one may achieve fine-grained security control (at the cost of added branching structure). For example, in the code below *p* is a high variable, and its security level is checked before returning it to the caller.

$Nat : High\ c := 0;$ **if** $p \sqsubseteq caller$ **then** $c := p$ **fi**; **return** c

In a program with futures, the passing of future references to other objects is legal since they are just references. However, the exact security level of a future value is revealed when it becomes resolved, thus an if-test checking for the future passing cannot help. A dynamic approach is required to control access to a future value at run-time when it is resolved. The concept of futures makes static checking less precise, and the need for complementary run-time checking is greater, as provided in the present paper. An example in Figure 4 shows why static analysis alone is not sufficient in the context of a real problem. This example is a health care system written in our core language and involves futures for communication and sharing of secure information.

3.2 The proposed core language syntax

In order to implement our security approach, the security mechanisms are embedded in a core language, using a syntax similar to that of Creol/ABS. It supports non-blocking calls with explicit futures. For simplicity, we assume all remote object interactions

Basic constructs	
$X := E$	assignment (variable X, and expression E)
$X :=$ **new** $C(\overline{E})$	object creation ($\overline{E}$ actual class parameters)
$X :=$ **new**$_{Lev}$ $C(\overline{E})$	object creation with the security level *Lev*
return E	creating a method result/future value
if B **th** S [**el** S'] **fi**	if statement (B a Boolean condition)
while B **do** S **od**	while statement (S a statement list)
Call constructs	
$!O.M(\overline{E})$	simple asynchronous remote call, it is a broadcast when O is a list of objects
$Q!O.M(\overline{E})$	remote asynchronous call (to object O) creating an explicit future, assigned to Q
$Q!this.M(\overline{E})$	asynchronous self call, with future variable Q
$X := O.M(\overline{E})$	synchronous call, i.e., an abbreviation for $Q!O.M(\overline{E});\ Q?(X)$
$X := .M(\overline{E})$	local call, with standard stack-based semantics
Access constructs	
$Q?(X)$	blocking access operation on future Q

Figure 3: Core Syntax.

are made by means of futures. Therefore, the result of a remote call always is returned back to the corresponding future. We allow first-class futures. The syntax of statements is given in Figure 3 letting O denote an object expression, M a method name, Q a future, E an expression and $\overline{E}$ an expression list. A call statement $!M(\overline{E})$ is an *asynchronous local invocation* without waiting for the result, where M is the method name and $\overline{E}$ is a list of actual parameters.

The statement $Q!O.M(\overline{E})$ is an *asynchronous and remote method call* toward object O. The call creates an explicit future with a unique future identity, assigned to the future variable Q. This statement sends a call request message to the callee O, and the caller object proceeds without waiting. A return value can be accessed with a get statement $Q?(X)$, which blocks while waiting for the corresponding future to become resolved and then assigns the future value to the variable X. The statement $Q?(X)$ is similar to $X = Q$.**get** in the ABS language. A synchronous call $X := O.M(\overline{E})$ may be expressed in terms of an asynchronous call: $Q!O.M(\overline{E});\ Q?(X)$.

The simple call statement $!O.M(\overline{E})$ is an *asynchronous invocation* without associating a future to the call, and thus no query is possible. Here O may be a list of objects, and that gives a broadcast to those objects. A local call with the syntax $X := .M(\overline{E})$ uses a standard stack-based execution to perform the call. In contrast, the self call $Q!this.M(\overline{E})$ uses a future to handle the result. (And in this case, a query on Q in the object could lead to deadlock.) Figure 3 provides the core language syntax.

Example. Figure 4 illustrates an example of a health care service in our core language, where high variables are emphasized based on static analysis/user specifications, in this case reflecting medical data. The problem the system is trying to solve is to prevent personnel and patients with lower-level access from accessing the medical records (communicated through futures). In this example, the server, defined by class *Service*, searches for the test result of a patient and publishes it to the patient and associated personnel through the *proxy* object. The server uses futures to communicate test results to the *proxy*, thus it does not wait for the results and is

```
 1 data type Result = ... // definition of medical data
 2
 3 interface ServiceI {
 4   Void produce() ... }
 5
 6 interface ProxyI {
 7   Void publish(Fut[Result] Q, PatientI a, List[PersonnelI] d) ... }
 8
 9 interface LabI {
10   Result_{High} detectResult(PatientI a) ... }
11
12 interface PatientI {
13   Void signal(Result_{High} r) ... }
14
15 interface PersonnelI {
16   Void signal(Result_{High} r) ... }
17
18 interface DataBase {
19   List[PersonnelI] findPersonnel(PatientI a)
20   PatientI getPatient(Int counter) ... }
21
22 class Service(LabI lab, DataBase db) implements ServiceI { List[PersonnelI] d = Nil;  Int counter = 0;
23  ProxyI proxy = new Proxy(this);  //proxy does the main job
24  !this.produce();  // initial action, starting a produce cycle
25
26  Void produce() { Fut[Result] Q;  PatientI a;
27    a = db.getPatient(counter);  // finding a patient in the database
28    d = db.findPersonnel(a);  // finding a group of personnel associated to the patient (a)
29    Q!lab.detectResult(a);
30    !proxy.publish(Q, a, d);  // sending the future (Q), no waiting
31    counter++; }
32 }
33
34 class Proxy(ServiceI s) implements ProxyI{ Result_{High} r;
35  Void publish(Fut[Result] Q, PatientI a, List[PersonnelI] d) {
36   Q?(r);  // waiting for the future and assigning the value to r
37   !a.signal(r);  // r is now High
38   !d.signal(r);  // multicasting, r is High
39   !s.produce(); }
40 }
```

Figure 4: Example showing interaction between a server, laboratory, patients, and health personnel sharing *High* test results.

free to respond to any client request. Instead, the *proxy* waits for the results from the laboratory and publishes them. (A simple asynchronous remote calls $!O.M(\overline{E})$ is a broadcast when O is a list of objects.) In line 24, a produce cycle is initiated between the *server* and *proxy* objects. In line 27, the method call *db.getPatient* gets a patient's identity (a) from the database, and then the *db.findPersonnel* finds the personnel (d) associated with this patient. In line 29, the server searches for the test results of this patient by sending a remote asynchronous call to the laboratory *Q!lab.detectResult*, where Q is the future variable and *lab* is the callee. This method call creates an explicit future, and its identity is assigned to the future variable Q. In line 30, the future is passed to object *proxy* for publishing, and this object waits and assigns the result to the variable r as in line 36. Then object *proxy* sends this value to the patient and personnel.

This example can be extended to a hierarchy of security levels according to different access rights. Here we assume that the possible security levels are *High* and *Low*, and our static analysis over-approximates the security levels of test results as *High*, as emphasized in Figure 4. This over-approximation leads to an inflation of high security levels and unnecessary rejections of information passing or call-based interactions. Note that the two signal calls in class *Proxy* would not be allowed if we only use static checking since we cannot tell which patients and personnel have a high enough level. In the present framework, class *Proxy* is therefore marked as unsafe, and precise level information for objects of this class is calculated at run-time and non-interference is ensured by means of wrappers. A static analysis which considers references as *Low* might leak high information through passing futures. For instance, in line 30, static analysis allows passing the future *Q* as a *Low* reference to object *proxy*, but later when it is resolved it can be *High*, and the object *proxy* sends this value to other objects. We focus in this paper on run-time checking, using the exact security level information at run-time rather than over-approximation.

4 A FRAMEWORK FOR NON-INTERFERENCE

Like Creol and ABS, our core language is equipped with behavioral interfaces, which means that created objects are typed by interfaces, not classes [14]. Therefore, remote access to fields or methods that are not exported through an interface is impossible. It makes the observable behavior of an object limited to interactions by means of remote method calls. Illegal object interactions are the ones leading to an information-flow from more sensitive to less sensitive information holders.

According to [18], a wrapper wraps an object (or component) and filters (modifies, deletes, adds) messages going out from an object and coming inside. A wrapper acts like a "local firewall" enforcing the safety of an object or a component. In this paper, we exploit the notion of wrappers to perform dynamic checking for enforcing non-interference in object interactions. The operational semantics of wrappers is explained in Section 4.3. An object can reveal confidential information through outgoing method calls by sending actual parameters with high security levels to lower level objects. In this case, a wrapper based on the security policies blocks illegal communications. Moreover, if a return value is *High*, then all the objects having a reference to the corresponding future can access the value. In a similar manner, a future wrapper controls access to the value and blocks illegal ones. Inside an object, in order to compute the exact security levels of created messages or return values, flow-sensitivity must be active, using dynamic information-flow enforcement. The operational semantics of our dynamic flow-sensitive enforcement is given in Section 4.2. Security policies of a wrapper are based on the run-time security levels. Thus, it makes our approach to be more permissive and precise. We can be conservative to wrap all objects and correspondingly activate dynamic flow-sensitivity, but it costs run-time overhead. The semantics of a wrapper is defined in a way making it independent of the object, but the flow-sensitivity slows down the object. In order to be more efficient at run-time, it is important to perform dynamic checking only for components where it is necessary.

Table 1 shows the results of a class-wise static analysis to identify where to put wrappers and correspondingly where to activate the flow-sensitivity. The column "static analysis" classifies classes according to their observable behavior through method calls and return values. The column "flow" shows if objects created from these classes need active flow-sensitivity. The column "wrapper" shows if wrappers are needed for objects or futures. The column "class" defines a class as *safe* or *unsafe* according to the static analysis results. For example, a class is *safe* if there are no outgoing calls with high parameters and no high return values; therefore, flow-sensitivity is off and neither the object nor future needs to be wrapped. Otherwise, a class is *unsafe*, and objects created from these classes are active flow-sensitive. In addition, an object is wrapped if there is an outgoing method call, where the security level of at least one of its actual parameters is high, and a future is wrapped if the corresponding return value is high. Based on this Table, objects can be: 1) normal without active flow-sensitivity or a wrapper, 2) active flow-sensitive and not wrapped, 3) active flow-sensitive and wrapped. Similarly, futures can be : 1) unwrapped or 2) wrapped. We define each type of the above-mentioned objects and futures in the operational semantics of the language (see Sections 4.2-4.3).

Table 1: Static analysis results

Static analysis	*flow*	*wrapper*	*class*
No calls with *High* parameters No *High* return values	off	none	safe
No calls with *High* parameters Some *High* return values	on	Fut	unsafe
Some calls with *High* parameters No *High* return values	on	Obj	unsafe
Some calls with *High* parameters Some *High* return values	on	Obj & Fut	unsafe

4.1 Static analysis

According to Table 1, we benefit from static analysis to categorize a class definition as safe or unsafe based on analysis of the parameters of outgoing calls and return values. This will make the execution of objects of safe classes faster, as we avoid a potentially large number of run-time checks and wrappers. Our approach can be combined with any (sound) static over-approximation for detecting security errors and safe classes, for instance the one proposed in [17], which is more permissive (to classify a class as safe) than the static analysis indicated here, in that high communication is considered secure as long as the declared levels of parameters are respected. Thus, the present work is complementary to [17]. For instance, if a class definition obeys the confidentiality typing rules proposed in [17], it is not a threat of leakage and it can be assumed to be safe. If not, then the class may be unsafe. Moreover, in [17] it has been proven that this static type system is sound, ensuring that evaluation of variables and expressions at run-time results in levels less than or equal to those of the static analysis.

Based on our core language, a program consists of interfaces and classes. In a class, fields and formal parameters are declared with static security levels, representing maximal security levels. Local variables do not have a declared security level, they start with *Low* (as default) but may change after each statement due to flow-sensitivity. Inside interfaces, formal parameters of a method and its return value are declared with maximum static security levels. In order to declare a parameter security level, all possible levels that can be assigned at run-time are considered, then the maximum one is taken. This leads to maximum security levels for all variables at a given program point. Since an object encapsulates local data and fields, there is no need to control the flow of information inside an object. However, since all object interactions are done by method

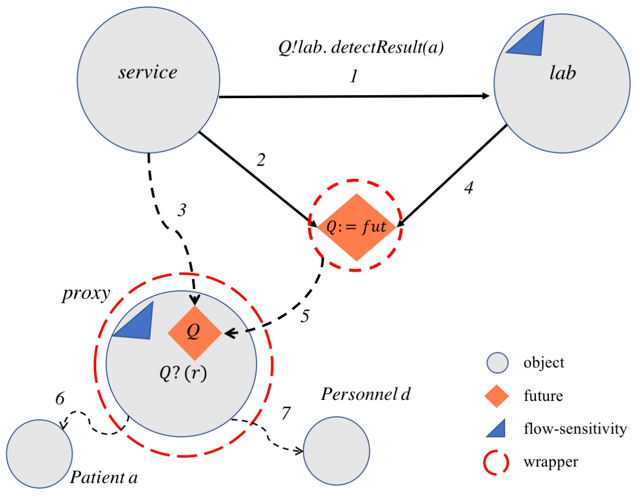

Figure 5: Information flow security regarding wrappers.

calls, typing rules are defined to control security levels of return values and actual parameters. A class is safe if the confidentiality of each method is satisfied. The confidentiality of a method is satisfied if the typing rules for its return value and actual parameters are satisfied. The typing rules check that each occurrence of an actual parameter (or return value) is not high, then the class is safe; otherwise, it is unsafe and needs dynamic checking at run-time.

According to the results of the static analysis and Table 1, we can categorize safe and unsafe classes for the example in Figure 4. As emphasized in this example, the interface laboratory *LabI* has a method with a high return value (*detectResult*). Thus the object *lab* is unsafe and flow-sensitivity must be active to compute the security level of the return value at run-time. The class *Proxy* is unsafe since it has at least one method call with a high actual parameter, thus object *proxy* is active flow-sensitive and wrapped. Figure 5 represents the run-time model of this program, and object communications are illustrated with arrow numbers. Objects are represented as circles, and the active flow-sensitivity is represented by a triangle inside an object. A dashed circle denotes a wrapper, and rhombus is a future. Whenever a remote method call for detecting a test result is made (arrow #1), the object *service* creates a future with identity *fut* (arrow #2). This object sends the future to the object *proxy* (arrow #3). The laboratory object (*lab*) returns back the test result (arrow #4), the object *proxy* gets the result (arrow #5) and sends it to the patient *a* (arrow #6) and the personnel *d* (arrow #7) . According to our security model with wrappers, if the return value of method *detectResult* is high, then the future wrapper protects this value. In fact, the future wrapper lets the object *proxy* get the value only if the security level of the object is greater than or equal to the security level of the future value. Otherwise, it blocks or raises an error. Based on the security levels of objects *a* and *d*, the wrapper of object *proxy* decides whether to send out the *signal* messages (6, 7) or block them.

```
sorts Class Object Queue Msg Future Wrapper Configuration .
subsorts Class Object Queue Msg Future Wrapper < Configuration .

op none : ⟶ Configuration .
op _ _ : Configuration Configuration ⟶ Configuration
          [assoc comm id: none] .
```

Figure 6: Our definition of system configuration in Maude.

4.2 Operational semantics

We here discuss the operational semantics of our secure programming language (including the notions of flow-sensitivity and wrappers). Our secure programming language has a small-step operational semantics defined by a set of rewrite logic (RL) rules in the Maude format. Maude uses sorts to define data types and data structures together with operations (functions) to manipulate values and *constructor functions* to define the value set of a sort. Operator functions can be defined recursively by equations, and rules are used to define (non-reversable) execution steps. Both equations and rules are applied by letting an occurrence of a left-hand-side pattern be replaced by the corresponding instance of the right-hand-side. Rules (and equations) are computed by rewriting from left to right changing (a part of) the system state. Non-overlapping rewrites may be done in parallel. Concurrent and distributed systems are modeled as multisets consisting of the relevant components and messages. The order of objects and messages in a multiset is immaterial, and the outcome of a system can be different based on non-deterministic applications of the rewrite rules. In other words, given a set of rules and equations, the rules are non-deterministically applied. In addition, rules and equations can be conditional, using the format $lhs \longrightarrow rhs$ *if condition* for rules and $rhs = lhs$ *if condition* for equations.

Maude has a built-in understanding of associativity, commutativity, and identity, which are used in constructor functions to define data types, in particular the multiset constructor (denoted by white space). A system state can then be modeled as a *configuration*, which is a multiset of objects (with or without active flow-sensitivity), queues, messages, futures, and wrappers. The *Configuration* sort is a super sort of the sorts of the mentioned components. (Classes are included in a configuration to provide static information about fields and methods.) Assuming a system with objects and messages, the *Configuration* sort is defined by two constructor functions as in Figure 6. The first constructor (which does not take any arguments) defines *none* as a constant of sort *Configuration*, representing an empty state. The binary multiset constructor (denoted by white space) is defined on *Configuration* as associative, commutative, and with *none* as the identity element, reflecting multiset semantics. The underscores (_) determine where the arguments should be placed.

An object in a given state is represented by a term $< O : C \mid A_1 : V_1, ..., A_n : V_n >$, where O is the object's identifier or name, C is its class name, the A_i's are the names of the object's attribute identifiers and the V_i's are the corresponding values. Figure 7 represents the components of a configuration. An **object** component is represented as:

$$< O : C \mid Att : S,\ Pr : (L, SL),\ Lcnt : N,\ Lvl : Lev >$$

object :	$< O : C \mid Att : S,\ Pr : (L, SL),\ Lcnt : N,\ Lvl : Lev >$
class :	$< Cl : C \mid Att : S', Mtds : MM, Ocnt : F >$
unsafe-class :	$< Cl : C \mid Att : S', Mtds : MM, Ocnt : F, Flow : \checkmark >$
flow-sensitive object :	$< O : C \mid Att : S,\ Pr : (L, SL),\ Lcnt : N,\ Lvl : Lev,\ Flow : \checkmark,\ PCstk : pcs >$
queue :	$< Qu : O \mid Ev : MMsg >$
invoc-msg :	$invoc(Fid,\ M,\ DL)\ from\ O\ to\ O'$
comp-msg :	$comp(D)\ from\ O\ to\ Fid$
future :	$fut(Fid,\ D)$
unresolved-future :	$fut(Fid,\ undef)$
wrapper :	$\{Wr : Wid,\ Lvl : Lev \mid Config\}$

Figure 7: The components of a configuration.

where O is the object name, C is the class name that the object is created from, S is the state of object fields (attributes), the pair (L, SL) represents the current active process, where L is the state of local variables defined in a method (a mapping from local variables to values), SL is the statements in the active process, and N is a natural number to assign unique identities to each method invocation created by the object. The states of object fields and local variables are given by mappings from variable names to values. For instance, $S[A \mapsto D]$ denotes updating S so that variable A binds to data value D. Variable Lev is the security level of the object ($Lev \in \{Low,\ High\}$), which is assigned by a programmer at the time of creating an object, using the syntax $A := \mathbf{new}_{Lev}\ C'(..)$.

A **class** is represented as:

$$< Cl : C \mid Att : S', Mtds : MM, Ocnt : F >$$

where C is the class name, S' is the class fields state (attributes), MM is a multiset of method declarations (each with code and local variables), and F is a natural number for generating unique object identities. For simplicity, we ignore inheritance in this paper; otherwise, there would be a field $Inh : S$, where S is the inheritance list.

In Figure 7, the **unsafe-class** definition includes the same fields as a safe class with an additional field $Flow : \checkmark$, denoting that all objects created from this class will be wrapped and with active flow-sensitivity control. The **flow-sensitive object** component shows an object with active flow-sensitivity. The field $Flow : \checkmark$ represents active flow-sensitivity inside an object, and pcs is the stack of pc (a list of security levels), which is used to avoid implicit flows in conditionals with high guards.

The **queue** component represents the external queue of an object for storing method invocations toward the object.

$$< Qu : O \mid Ev : MMsg >$$

The queue is associated with object O, and $MMsg$ is a multiset of stored messages toward object O. The **invoc-msg** component represents an invocation message from object O to O' as:

$$invoc(Fid,\ M,\ DL)\ from\ O\ to\ O'$$

where Fid is the future identity, M is the called method name, and DL is a data list of actual parameters. The **comp-msg** component represents a completion message, where D is the return value from object O (the callee) to a future with identity Fid.

$$comp(D)\ from\ O\ to\ Fid$$

A resolved **future** component with identity Fid and value D has the form

$$fut(Fid,\ D)$$

and an unresolved future component has $undef$ as value. A **wrapper** is represented as:

$$\{Wr : Wid,\ Lvl : Lev \mid Config\}$$

where Wid is the wrapper identity, Lev is its level, and $Config$ represents what the wrapper contains, its internal configuration. A wrapper wraps a future or an object and correspondingly, gets the identity and level of the component that it is wrapping. For instance, an object wrapper has the same identity and level as the object. Figure 8 is a legend of Figure 7. According to the Maude convention, Maude variables (i.e., meta variables) are written in capital letters.

Figure 9 represents the flow-sensitivity semantics of objects. The operational semantics is given by a number of rules, written in the style of SOS rules. A rule can be applied to a configuration if the left-hand-side matches a subset of the configuration (possibly reordered). And since configurations can be nested inside wrappers, the rules can also be applied to inner configurations. If the left-hand-sides of two rules match disjoint parts, they can be applied at the same time, indicating concurrency and non-determinism. In our semantics, each rule involves at most one object, reflecting that objects are executing independently from each other. Thus the objects execute in parallel. For simplicity, in the right-hand-side of the rules, fields that are as in the left-hand-side are ignored and indicated by ..., but changed fields are shown. Comments are shown by *** symbol.

The **new-obj** rule shows creation of a new object from a safe class (ignoring class parameters for simplicity). In the active process of object O, command $A := \mathbf{new}_{Lev}\ C'(EL)$ creates a new object from class C' with a unique identity $newId$ and its external queue. In the rules, for simplicity, $newId$ is an abbreviation for a function $new(C', F)$, which creates a unique object identity out of the class name C' and value of object counter F. This rule assigns $newId$ to

Structural names used in the operational semantics

Cl	indicates a class component
Att	attribute field (followed by the state of the attributes)
Mtds	method declaration field
Ocnt	object counter field
Flow	flow-sensitivity field
Pr	current active process field
Lcnt	method call counter field
Lvl	security level field
PCstk	pc stack field
Qu	indicates a queue component
Ev	stored invocation messages field in a queue
Wr	indicates a wrapper component
fut	indicates a future component
invoc	indicates an invocation message
comp	indicates a completion message

Meta variable names used in the operational semantics

C	class name
S, S'	state of class attributes
MM	a multiset of method declarations
F	a natural number (used for creating unique object identities)
O, O'	object name
L	state of local variables (mapping from names to values)
SL	remaining statement list in the active process
N	a natural number (used for creating unique call identities)
Lev	level (Lev ∈ {*Low*, *High*})
pcs	pc stack
MMsg	a multiset of invocation messages
Fid	future identity
M	method name
D	data value
DL	data value list (actual parameters)
A	program variable
Q	future variable
E	expression
EL	expression list
Wid	wrapper identity (the ID of the component it wraps)
Config	a variable of sort Configuration

Figure 8: A legend for the operational semantics.

the variable A and increases the object counter by one in the class C'. Moreover, the fields of the new object are assigned correspondingly. The active process is initialized with (*empty*, *idle*), where *empty* denotes an empty state of local variables, and *idle* denotes an empty statement list (no active process). The new object level is *Lev* since it is specified in the command $A := \texttt{new}_{Lev}\ C'(EL)$. If it is not specified, then the created object level would be assumed *Low*. Moreover, the new external queue is initialized to the new object's identity (*newId*) and no stored messages (*noMsg*). The semantics of the actual class parameters is here treated like parameters of an asynchronous call $!A.init(EL)$, where *init* is the name of the initialization method of a class.

In the **new-obj'** rule, C' is an unsafe class; therefore, a new object with active flow-sensitivity is created inside a wrapper. The wrapper gets the same identity and level as the new object, $Wr : newId, Lvl : Lev$, and it wraps the new object and its queue. The new object has additional fields $Flow : ✓$ and an empty stack of *pc* (*emp*). In an active flow-sensitive object, variables and fields can be defined with security tags. For example, the evaluation result of an expression E is denoted by $[E]$ which returns back both the value and security tag as D_{Lev}, where D is the value and *Lev* is the tag. If this value is assigned to a program variable A, the binding $A \mapsto D_{Lev}$ is added to the corresponding state in the object. Moreover, a variable without a security tag is assumed *Low*. Static analysis assigns variables to their maximum static security levels due to over-approximation. Then at run-time due to the flow-sensitivity, security levels of variables can change and propagate to other variables. Therefore, a variable without a tag means that it was assumed as *Low* at the static time, and it has not been tainted by *High* information at the time of analysis. Thus at run-time, it is considered as *Low* (which might change afterward). The tag of a variable is extracted by a function $level(D_{Lev})$, which returns back *Lev*. It is worth mentioning that $[E]$ is an abbreviation for $eval(E, S\#L)$. The eval function evaluates an expression E by considering the map composition of S and L, i.e., $S\#L$, reflecting that the binding of a variable name in the inner scope L shadows any binding of that name in the outer scope S. Considering a variable X, $eval(X, S\#L)$ equals $eval(X, L)$ if L has a binding for X, otherwise $eval(X, S)$.

Rules **assign** and **assign'** show an assignment statement of $(A := E)$ in a normal and active flow-sensitive object, respectively. The function $dom(A, L)$ is true if variable A is in the local state L, then in the next step, A with its value is inserted to L by function $insert(A, [E], L)$; otherwise, it is inserted to the attribute state S. In the **assign'** rule, the *insertTag* function inserts A with a value and security tag to the corresponding state. The tag is a join $\sqcup$ of the security level of $[E]$ and *pc*, where $pc = \sqcup_i pcs[i]$, and i ranges over all indexes in the stack ($level([E]_{\sqcup pc}) = level([E]) \sqcup pc$). If the stack is empty, then $[E]_{\sqcup emp}$ equals $[E]$. In the case where a variable is without tag, the function *insertTag* adds it to the domain with the *pc* level. According to the **assign'** rule, inside an active flow-sensitive object, the security level of a variable containing the identity of a new object or future is affected by the context level *pc*. This rule inserts these variables with the *pc* security level to the corresponding state, although an object or a future identity is not confidential.

The rules **if-low** and **if-high** represent how the stack of *pc* (*pcs*) changes according to the guard security level in a conditional structure **`if th el fi`**. In the **if-low** rule, the guard's security level is *Low*, thus the stack of *pc* does not change, and only the corresponding branch is taken. Rule **if-high** applies when branching happens in a high context ($level([E]) \neq Low$). Similar to the approach [20], in order to avoid implicit flows, the security levels of variables appearing in both branches are raised by this rule. In this case, the stack of *pc* is updated by adding the guard security level ($pcs\ level([E])$). In addition, an auxiliary function $endif(SL')$ is defined to mark the join point of the **`if`** structure, and SL' is the statements of the untaken branch. At the join point, security

$$
\begin{array}{lll}
\text{new-obj :} & & < O : C \mid Att : S,\ Pr : (L,\ A := \mathtt{new}_{Lev}\ C'(EL);\ SL),\ Lcnt : N,\ Lvl : Level > \\
& & < Cl : C' \mid Att : S',\ Mtds : MM,\ Ocnt : F > \\
& \longrightarrow & < O : C \mid ...,\ Pr : (L,\ A := newId;\ !A.init(EL);\ SL),\ ... > \\
& & < Cl : C' \mid ...,\ Ocnt : (F+1),\ ... > \\
& & < newId : C' \mid Att : S', this \mapsto newId,\ Pr : (empty, idle),\ Lcnt : 1,\ Lvl : Lev > \\
& & < Qu : newId \mid Ev : noMsg > \\[1ex]
\text{new-obj' :} & & < O : C \mid Att : S,\ Pr : (L,\ A := \mathtt{new}_{Lev}\ C'(EL);\ SL),\ Lcnt : N,\ Lvl : Level > \\
& & < Cl : C' \mid Att : S',\ Mtds : MM,\ Ocnt : F, Flow : \checkmark > \\
& \longrightarrow & < O : C \mid ...,\ Pr : (L,\ A := newId;\ !A.init(EL);\ SL),\ ... > \\
& & < Cl : C' \mid ...,\ Ocnt : (F+1),\ ... > \\
& & \{ Wr : newId,\ Lvl : Lev \mid < newId : C' | Att : S', this \mapsto newId, Pr : (empty, idle), Lcnt : 1, Lvl : Lev, Flow : \checkmark, PCstk : emp > \\
& & < Qu : newId \mid Ev : noMsg > \ \} \\[1ex]
\text{assign :} & & < O : C \mid Att : S,\ Pr : (L,\ A := E;\ SL),\ Lcnt : N > \\
& \longrightarrow & if\ \ dom(A, L)\ then\ \ < O : C \mid ...,\ Pr : (insert(A, [E], L),\ SL), ... > \\
& & \qquad\qquad\quad else\ \ < O : C \mid Att : insert(A, [E], S),\ Pr : (L, SL), ... > \\[1ex]
\text{assign' :} & & < O : C \mid Att : S,\ Pr : (L,\ A := E;\ SL),\ Lcnt : N,\ Lvl : Lev,\ Flow : \checkmark,\ PCstk : pcs > \\
& \longrightarrow & if\ \ dom(A, L)\ then\ \ < O : C \mid ...,\ Pr : (insertTag(A, [E]_{\sqcup pc}, L),\ SL),\ ... > \\
& & \qquad\qquad\quad else\ \ < O : C \mid Att : insertTag(A, [E]_{\sqcup pc},\ S),\ Pr : (L, SL),\ ... > \\[1ex]
\text{if-low :} & & < O : C \mid Att : S, Pr : (L,\ \mathtt{if}\ E\ \mathtt{th}\ SL'\ \mathtt{el}\ SL''\ \mathtt{fi};\ SL),\ Lcnt : N,\ Lvl : Lev,\ Flow : \checkmark,\ PCstk : pcs > \\
& \longrightarrow & if\ \ [E] = true\ then\ \ < O : C \mid ...,\ Pr : (L,\ SL';\ SL),\ ... > \\
& & \qquad\qquad\quad\ \ else\ \ < O : C \mid ...,\ Pr : (L,\ SL'';\ SL),\ ... > \\
& & if\ \ level([E]) = Low \\[1ex]
\text{if-high :} & & < O : C \mid Att : S, Pr : (L, \mathtt{if}\ E\ \mathtt{th}\ SL'\ \mathtt{el}\ SL''\ \mathtt{fi};\ SL),\ Lcnt : N,\ Lvl : Lev,\ Flow : \checkmark,\ PCstk : pcs > \\
& \longrightarrow & if\ \ [E] = true\ then\ \ < O : C \mid ...,\ Pr : (L,\ SL';\ endif(SL'');\ SL),\ ...,\ PCstk : pcs\ level([E]) > \\
& & \qquad\qquad\quad\ \ else\ \ < O : C \mid ...,\ Pr : (L,\ SL'';\ endif(SL');\ SL),\ ...,\ PCstk : pcs\ level([E]) >) \\
& & if\ \ level[E] \neq Low \\[1ex]
\text{endif :} & & < O : C \mid Att : S,\ Pr : (L,\ endif(SL');\ SL),\ Lcnt : N,\ Lvl : Lev,\ Flow : \checkmark,\ PCstk : pcs\ Lev' > \\
& \longrightarrow & < O : C \mid ...,\ Pr : (L,\ update(SL', Lev');\ SL),\ ...,\ PCstk : pcs >
\end{array}
$$

Figure 9: Flow-sensitivity operational semantics, where $newId = new(C', F)$**,** $[E] = eval(E, (S\#L))$**, and** $pc = \sqcup_i pcs[i]$**.**

levels of variables in the untaken branch are updated with the guard level. According to the **endif** rule, the function $update(SL', Lev')$ updates the variables of the untaken branch (SL') with the current context level (Lev'). Moreover, Lev' is removed from the stack of pc, reflecting the previous context.

Figure 10 shows the operational semantics regarding method calls and the future creation. In the rules, we do not cover local calls, which are standard and not involving object interactions (therefore, less interesting here). The **start** rule says that when there is no active process (the object is *idle*) and there is an invocation message in its queue, the object starts to execute the corresponding method. The rule captures the method's body *SL* as the active process, and the object's local state is updated, binding formal parameters to the actual ones, storing the future identity (*Id*) in *label*. The *label* variable is used later to execute the **`return`** statements to return back the value to the corresponding future. Moreover, in the left hand side of this rule, only the fields common for both normal and active flow-sensitive objects are specified to indicate that this rule can be applied to both types of objects.

The **async-call** rule deals with an asynchronous and remote method call $Q!E.M(EL)$, where Q is a future variable, E is the callee, M is the method name and EL is a list of actual parameters. In this rule, the call is reduced to $!E.M(EL)$, and a future with identity *newfId* and value *undef* (not resolved) is created. The *newfId* is an abbreviation for function $new(O, N)$, which creates a unique identity from the object name O and N. The new identity *newfId* is assigned to the future variable Q. A future variable can be used for accessing the return value (future value) or synchronization as explained in Section 3.2. The rule **async-call'** shows an asynchronous call $!E.M(EL)$, which creates an invocation message toward the callee. The invocation message contains *newfId*, method name M, and actual parameters EL.

The **return** rule interprets a **`return`** statement, which creates a completion message toward the corresponding future ($eval(label, L)$), and the object becomes idle. We assume each method body ends with a **`return`** statement. The **return'** rule represents a **`return`** statement in an active flow-sensitive object, where the security level of the completion message is $level(D) \sqcup pc$, and D is the return value.

start :

$$< O : C \mid Att : S,\ Pr : (empty, idle),\ Lcnt : N,\ Lvl : Lev,\ \ldots >$$
$$< Qu : O \mid Ev : MMsg\ invoc(Id,\ M,\ DL) >$$
$$\longrightarrow\ < O : C \mid Att : S,\ Pr : ([label \mapsto Id, \overline{X} \mapsto DL, \overline{L} \mapsto \overline{L_0}], SL),\ \ldots >$$
$$< Qu : O \mid Ev : MMsg >$$

where method M binds to $M(\overline{X})\{\overline{L_0};\ SL;\}$ with initial local state $\overline{L_0}$

async-call :

$$< O : C \mid Att : S,\ Pr : (L,\ Q!E.M(EL);\ SL),\ Lcnt : N,\ Lvl : Lev >$$
$$\longrightarrow\ fut(newfId, undef)$$
$$< O : C \mid Att : S,\ Pr : (L,\ Q := newfId;\ !E.M(EL);\ SL),\ \ldots >$$

async-call' :

$$< O : C \mid Att : S,\ Pr : (L,\ !E.M(EL);\ SL),\ Lcnt : N,\ Lvl : Lev >$$
$$\longrightarrow\ < O\ :\ C \mid \ldots, Pr : (L,\ SL),\ \ldots,\ Lcnt : N+1,\ \ldots >$$
$$invoc(newfId,\ M,\ [EL])\ from\ O\ to\ [E]$$

return :

$$< O : C \mid Att : S, Pr : (L,\ \mathtt{return}(D);),\ Lcnt : N,\ Lvl : Lev >$$
$$\longrightarrow\ < O : C \mid \ldots,\ Pr : (empty, idle),\ \ldots >$$
$$comp(D)\ from\ O\ to\ eval(label, L)\quad {*}{*}{*}\ \text{future Identity}$$

return' :

$$< O : C \mid Att : S, Pr : (L,\ \mathtt{return}(D);),\ Lcnt : N,\ Lvl : Lev,\ Flow : ✓,\ PCstk : pcs >$$
$$\longrightarrow\ < O : C \mid \ldots,\ Pr : (empty, idle),\ \ldots >$$
$$(comp(D)\ from\ O\ to\ eval(label, L))_{level(D) \sqcup pc}$$

Figure 10: Object interaction operational semantics, where $[E] = eval(E, (S\#L))$ **and** $newfId = new(O, N)$**.**

4.3 Wrappers operational semantics

A wrapper is assigned to an object or a future by the run-time system. According to the **new-obj'** rule, the run-time system wraps a new object created from an unsafe class. Not all object wrappers are needed; for example, according to Table 1 if a class is unsafe with high return values but no high outgoing calls, the new object does not need a wrapper (and this can be adjusted as future work). A future is wrapped when there is a high return value. Assuming the static analysis (Table 1), there are some objects which are not wrapped or active flow-sensitive. Considering this assumption, the operational semantics of a wrapper is given in Figure 11.

A wrapper wraps a flow-sensitive object and all invocation messages that the object might create and also the queue. The two rules of **wr-call** and **wr-call'** represent an asynchronous call inside an active flow-sensitive object. They are similar to the two rules **async-call** and **async-call'** in Figure 10. In the **wr-call'** rule, an invocation message is created, where EL is a list of actual parameters. The security level of the message is $level([EL]) \sqcup pc$, which is the join of security levels of the actual parameters in EL and pc. An object may send a future variable, as an actual parameter, to other objects. Although a future identity is not confidential, it is affected by the current context pc level. According to this rule, all outgoing messages at least get the pc level, and the wrapper checks the security levels based on the next rule.

The **wr-invoc** rule represents a wrapper with an invocation message and *Config* inside, denoting the wrapper's configuration. At run-time, a wrapper acquires and records the security levels of destination objects before sending the invocation messages toward them. If the security level of the message is less than or equal to the destination object level ($level(O')$), then the wrapper allows the message to go out. Otherwise, as the else-branch of this rule, the wrapper eats the message and it disappears from inside. In this case, an object performing a get will deadlock because the invocation message was deleted. It is worth mentioning that the setting of asynchronous method calls and futures has an inherent possibility of objects being blocked, and thus deadlocked. For instance, a method result may never appear in the future if the callee object is blocked. The present work makes the situation even worse by deleting certain invocation messages. We have extended the approach with a notion of errors, so that the deletion of an invocation message results in an error value in the corresponding future component. This can be combined with an exception handling mechanism such that an exception is raised when a get operation tries to access an error value. However, as this is beyond the scope of this paper, we ignore the exception handling part here. We simply indicate exceptions by assignments with `error` in the right-hand-side. One could use exception handling as in ABS (where also a time-out mechanism is considered). The **fut-get'** rule represents the case when a future value is `error`, and an object performs a get command $Fid?(A)$ asks for the return value of this future. In this case, the object gets an error instead of being blocked because the corresponding invocation message was deleted by the object wrapper.

The **invoc-wr'** rule represents a wrapper and an incoming invocation message toward the object (O). The notation $\Lambda[M, i]$ denotes the level of the ith formal parameter of the method M as declared in the class. If the security level of each actual parameter (DL_i) is less than or equal to the security level of the corresponding formal parameter, then the wrapper allows the message to go through and adds it to its configuration inside. The **invoc-qu** equation stores an invocation message toward an object in the corresponding queue for later processing. In case there is a wrapper, the message has passed it. This equation for storing a message inside the corresponding queue has priority over the rule sending an invocation message

$$\begin{array}{lll}
\text{wr-call :} & & < O:C \mid Att:S,\ Pr:(L,\ Q!E.M(EL);\ SL),\ Lcnt:N,\ Lvl:Lev,\ Flow:\checkmark,\ PCstk:pcs > \\
& \longrightarrow & fut(newfId, undef) \\
& & < O:C \mid \ldots,\ Pr:(L,\ Q := newfId;\ !E.M(EL);\ SL),\ \ldots >
\end{array}$$

$$\begin{array}{lll}
\text{wr-call' :} & & < O:C \mid Att:S,\ Pr:(L,\ !E.M(EL);\ SL),\ Lcnt:N,\ Lvl:Lev,\ Flow:\checkmark,\ PCstk:pcs > \\
& \longrightarrow & < O:C \mid \ldots, Pr:(L,\ SL),\ \ldots,\ Lcnt:N+1, \ldots > \\
& & (invoc(newfId,\ M,\ [EL])\ from\ O\ to\ [E])_{level([EL])\sqcup pc}
\end{array}$$

$$\begin{array}{lll}
\text{wr-invoc :} & & \{\ Wr:O,\ Lvl:Lev \mid (invoc(Id,\ M,\ DL)\ from\ O\ to\ O')_{Lev'}\ Config\ \} \\
& & fut(Id, undef) \\
& \longrightarrow & if\ \ Lev' \sqsubseteq level(O')\ then\ \{\ Wr:O,\ Lvl:Lev \mid Config\ \}\ \ (invoc(Id,\ M,\ DL)\ from\ O\ to\ O')_{Lev'}\ \ fut(Id, undef) \\
& & \qquad\qquad\qquad\qquad\quad else\ \ \{\ Wr:O,\ Lvl:Lev \mid Config\ \}\ \ fut(Id, \mathtt{error})
\end{array}$$

$$\begin{array}{lll}
\text{fut-get' :} & & fut(Fid, \mathtt{error}) \\
& & < O:C \mid Att:S,\ Pr:(L,\ Fid?(A);\ SL),\ Lcnt:F,\ Lvl:Lev,\ \ldots > \\
& \longrightarrow & fut(Fid, \mathtt{error}) \\
& & < O:C \mid \ldots,\ Pr:(L,\ A := \mathtt{error};\ SL),\ \ldots >
\end{array}$$

$$\begin{array}{lll}
\text{invoc-wr' :} & & \{\ Wr:O,\ Lvl:Lev \mid Config\ \} \\
& & invoc(Id,\ M,\ DL)\ from\ O'\ to\ O \\
& \longrightarrow & if\ \ \forall\, i\ :\ level(DL_i) \sqsubseteq\ \Lambda[M,i]\ then\ \{\ Wr:O,\ Lvl:Lev \mid invoc(Id,\ M,\ DL)\ from\ O'\ to\ O\ Config\ \ \} \\
& & \qquad\qquad\qquad\qquad\qquad\qquad\quad else\ \ \{\ Wr:O,\ Lvl:Lev \mid Config\ \}
\end{array}$$

$$\begin{array}{lll}
\text{invoc-qu :} & & < Qu:O \mid Ev:MMsg >\ \ invoc(Id,\ M,\ DL)\ from\ O'\ to\ O \\
& = & < Qu:O \mid Ev:MMsg\ invoc(Id,\ M,\ DL) >
\end{array}$$

$$\begin{array}{lll}
\text{wr-future :} & & fut(Fid, undef) \\
& & (comp(D)\ from\ O'\ to\ Fid)_{Lev} \\
& \longrightarrow & if\ \ Lev \neq\ Low\ then\ \{\ Wr:Fid,\ Lvl:Lev \mid fut(Fid, D)\ \} \\
& & \qquad\qquad\qquad\quad else\ \ fut(Fid, D)
\end{array}$$

$$\begin{array}{lll}
\text{wr-fut-get :} & & \{\ Wr:Fid,\ Lvl:Lev' \mid fut(Fid, D)\ \} \\
& & < O:C \mid Att:S,\ Pr:(L,\ Fid?(A);\ SL),\ Lcnt:F,\ Lvl:Lev,\ \ldots > \\
& \longrightarrow & if\ \ (Lev \sqsubseteq Lev')\ then\ \{\ Wr:Fid,\ Lvl:Lev' \mid \ldots\ \}\ \ < O:C \mid \ldots,\ Pr:(L,\ A := \mathtt{error};\ SL),\ \ldots > \\
& & \qquad\qquad\qquad\qquad else\ \ \{\ Wr:Fid,\ Lvl:Lev' \mid \ldots\ \}\ \ < O:C \mid \ldots,\ Pr:(L,\ A := D;\ SL),\ \ldots >
\end{array}$$

$$\begin{array}{lll}
\text{fut-get :} & & fut(Fid, D) \\
& & < O:C \mid Att:S,\ Pr:(L,\ Fid?(A);\ SL),\ Lcnt:F,\ Lvl:Lev,\ \ldots > \\
& \longrightarrow & fut(Fid, D) \\
& & < O:C \mid \ldots,\ Pr:(L,\ A := D;\ SL),\ \ldots >
\end{array}$$

Figure 11: Operational semantics involving wrappers, where $[E] = eval(E, (S\#L))$ and $newfId = new(O, N)$.

out of the wrapper (if both rules apply), since in Maude an equation has priority over a rule.

An unresolved future becomes resolved when it sees a completion message. Inside an active flow-sensitive object, the security level of a completion message is computed according to the **return'** rule in Figure 10. If the security level of a completion message is not *Low*, then the run-time system wraps the corresponding future. The **wr-future** rule represents an unresolved future with identity *Fid* and value *undef*, and a corresponding completion message. On the right-hand-side, if the security level of the completion message (*Lev*) is not *Low*, the future becomes both wrapped and resolved. If the future value is *Low*, then there is no need to protect it by a wrapper. The wrapper has the same identity as the future identity *Fid* and the security level of the future value *Lev*.

Now how does a wrapper protects a future value? The **wr-fut-get** rule represents a wrapped future and an object which wants to get the future value. Moreover, in the left hand side of this rule, only the common fields between a normal and active flow-sensitive object is specified to indicate that this rule can be applied to both types of objects. In this rule, the command $Fid?(A)$ asks for the return value of a method call with future given by *Fid* and assigns the value to the variable *A*. If the security level of the object (*Lev*) asking for the value is smaller than the wrapper security level (*Lev'*), then the wrapper sends an error value; otherwise, the object gets the future value (*D*). The **fut-get** rule shows an unwrapped future and an object which wants to get the future value. There is no security checking, and the object gets the value immediately.

We have here focused on rules formalizing inter-object interaction and futures. For simplicity, we have ignored local stack-based calls and while-loops, as well as exception handling in case of errors in communication and futures.

5 THEORETICAL RESULTS

This Section establishes results about local and global non-interference.

Theorem 1 (wrapped futures). *A high future value will not be passed to a low object.*

Proof. Since each call has a unique future identity, there is at most one write operation on a future component, and by Rule **wr-future**, the future component is placed in a wrapper if the future value is high. And this wrapper is not removed by any rule. Any get operation of such a future is handled by rule **wr-fut-get**, and this rule will not pass such a (*High*) future value to a low object. □

We next consider non-interference. In general, the concept of non-interference of non-deterministic systems is more complicated than that of deterministic systems. In our context of message-based systems, each object behaves in a deterministic manner, while the overall system may be non-deterministic due to independent object speeds and overtaking of message passing in the network. Asynchronous local calls could, however, create non-determinism with our semantics, if they are non-deterministically delayed; but we may assume immediate scheduling of such calls, and in a FIFO manner in case several local asynchronous calls are made from the same method. We may therefore use an adapted version of the standard definition of non-interference assuming each object is deterministic.

We let R denote a run-time configuration, and let the projection R/f denote the future component f in R (possibly with value *undef*). Furthermore, we let the projection R/o denote the local run-time configuration of an object o in R as given by the attributes of that object, including the object level and also the local communication history, i.e., the sequence of messages/values communicated from o to other objects/components (the "outputs" of o), or from other objects/components to o (the "inputs" of o), as in [19].

When comparing two run-time configurations $R1$ and $R2$, the object identities and future identities will in general differ. We, therefore, use a correspondence relation ($\sim_{R1,R2}$) between the objects in the two configurations and between the future components in the two configurations. We define equality over such configurations by equality modulo the correspondence of the object and future identities.

Definition 2 (Low equality of configurations, objects, and futures). *We say that two configurations are low equal ($=_{Low}$) if (i) there is a correspondence between the objects in the two configurations and between the future components in the two configurations, and if (ii) all the corresponding objects and futures are low equal. We say that two object-local configurations are low equal ($=_{Low}$) if the low values of the attributes (including the level and the local history) of the objects are equal modulo correspondence of object and future identities. We say that two future components are low equal if their values are low equal.*

We define non-interference by:

Definition 3 (Local Non-Interference for Objects). *Object-local non-interference, means that if two executions reach the pre-state of a basic output statement (call, new, or return) to be performed by corresponding objects o_1 and o_2 and with configurations R_1 and R_2, respectively, such that the local communication histories of o_1 and o_2 are low equal, then the* observable output *resulting from execution of the statement on the two configurations will be the same.*

Definition 4 (Observable output). *The observable output of a call statement s performed by an unwrapped object o is the message generated by s. For a wrapped object we consider only the messages passing through the wrapper. The observable output of a new statement is given by the actual class parameters. The observable output of a return statement is the generated value.*

We define global non-interference by:

Definition 5 (Global Non-Interference). *Global (system-wide) non-interference means that for any two executions with corresponding objects and futures as explained, two configurations that are low equal with respect to their global histories, should satisfy that the next outputs are low equal for each pair of corresponding objects.*

Theorem 6 (Global Non-Interference). *Global (system-wide) non-interference follows if local non-interference holds for all corresponding objects in two executions.*

Proof. Object-local non-interference for all corresponding objects implies that the next outputs are low equal for the corresponding objects. Since the communication histories are low equal, we have that the inputs (through input parameters of messages and queries on futures) are also low equal. For a return (or other outputs) from corresponding objects, the local communication histories are low equal, and by local non-interference, the returned values must be low equal. Therefore we have non-interference for the corresponding future components, and the values communicated from future components (requested by get statements) must be low equal, and the get statements must be low equal as well. □

Theorem 7 (Non-interference). *Our security model using the wrapper mechanism guarantees global non-interference.*

Proof. It suffices to prove object-local non-interference. The proof is by considering all cases of statements s producing an output. We must consider the following kinds of statements: call, new, and return. According to the **new-obj'** rule in Figure 9, a new object becomes wrapped when the class which it is created from is unsafe. Correspondingly, the wrapper includes all the object's invocation messages and the object queue. According to **wr-call'** rule in Figure 11, inside a flow-sensitive object an outgoing method call creates an invocation message. According to the **wr-invoc** rule, a wrapper checks the security levels, and if a message contains confidential data, the wrapper does not send it to a low-level object. The observable output of a call statement is the values of the parameters of M for which the run-time level and also the pc level is low at the time of creating the invocation message. These messages are low since they have not been affected by high variables inside the object. Therefore, a wrapper allows them to be sent to low objects.

When a new object or future are created, their identities are assigned to variables inside an object and due to the **assign'** rule in Figure 9, these variables get the level of pc. An object can send these variables as parameters of a method call to other objects (as an invocation message). However, if the object is wrapped, these variables (identities) are not sent to objects with a lower security level than *pc*. Only if *pc* is low, then low-level objects can have these variables. A `return` statement, based on the **return'** rule in Figure 10, creates a completion message with the join security level of the return value and *pc*. According to the **wr-future** rule in Figure 11, if the security level of a completion message is high, then the future becomes wrapped. Otherwise, low-level objects can access the value. According to the **wr-fut-get**, low-level objects cannot get the value from a wrapped future. □

6 RELATED WORK

A static and class-wise information-flow analysis has been suggested for Creol without futures by Owe and Ramezanifarkhani in [17]. It is a type system for the Creol language, named *SeCreol*. The approach is based on static declaration of security levels for each input parameter and return value of a method, object fields, and local variables. The authors proved soundness and a non-interference property in object interactions based on an operational semantics. In contrast to the present paper, futures are not considered in [17]. The present paper can be applied for languages supporting futures. Our approach is a dynamic technique which is more permissive, precise, and to overcome the run-time overhead we combine it with static analysis similar to the approach in [17]. The static analysis determines where dynamic checking is required and correspondingly wrappers are assigned to objects and futures by the run-time system to protect confidentiality of data.

In a paper by Attali et al. [2], secure information-flow for the ASP language is provided by dynamically checking for unauthorized information flows. ASP is based on active objects and supports asynchronous communications and futures. In their approach, security levels are assigned to activities and transmitted data between these activities (an activity includes one active object and several passive objects manipulated by one thread). The security levels do not change when they are assigned. Their security model guarantees data confidentiality for multi-level security (MLS) systems, which means that an entity will be given access only to the information that it is allowed to handle. Dynamic checks are implemented at activity creation, requests, and replies. Future references can be freely transmitted between activities because they do not hold any valuable information. However, for updating a future and getting its value, the secrecy level of this transmission will be checked dynamically by the security rules of a secure reply transmission. Our approach adds flow-sensitivity which allows security levels of variables to change inside an object. It makes our approach more permissive and a wrapper deals with run-time security levels. In addition to enforcing the non-interference property in object interactions, our approach guarantees that an object will be given access only to the information that it is allowed to handle (based on the MLS definition).

Sabelfeld and Russo [22] prove that a sound and pure dynamic information-flow enforcement is more permissive than static analysis in the case of flow-insensitivity (where variables are assigned security levels at the beginning of program execution, and then their security levels do not change during the execution). In addition, Russo and Sabelfeld [20] show that dynamic flow-sensitive enforcement (where security levels of variables change during execution) is the most permissive but unsound because of implicit flows in the conditional constructs. The authors propose an approach as explained in Section 2.1 to make it sound. We apply the same approach to avoid implicit flows.

Phung et al. [16] describe a method for wrapping built-in methods of JavaScript programs in order to enforce security policies. The security policies avoid web browser vulnerabilities and protect web pages from malicious JavaScript code. A policy specifies under what conditions a page may perform a certain action and a wrapper grants, rejects, or modifies these actions. As mentioned before, the notion of wrappers has been developed for the safety of objects [18]. For instance, when an object is wrapped it controls which actions are to be taken for any input/output communication event. We here exploit wrappers for dealing with information security, by extending the run-time system with secrecy levels and apply dynamic checking for securing object interactions.

7 CONCLUSION

We have proposed a framework for enforcing secure information flow and non-interference in active object languages based on the notion of wrappers. We have considered a high-level core language supporting asynchronous calls and futures. In our model, due to encapsulation, there is no need for information-flow restrictions inside an object. However, information-flow security of object interactions (with method-oriented communications) is enforced by wrappers, performing the dynamic checking. Furthermore, wrappers are used to control the extraction of confidential values from futures.

Security rules of wrappers are defined based on security levels assigned to objects, actual parameters, fields and local variables. Inside an object, the security levels of variables might change at run-time due to flow-sensitivity. In a setting where concurrent objects communicate confidential or non-confidential information, wrappers on unsafe objects and future components protect exchange of confidential values. Wrappers on objects protect outgoing method calls and prevent leakage of information through outgoing parameters. The wrappers are created by the run-time system without the involved parties being aware of it.

The notion of secure wrappers enable dynamic enforcement of avoidance of information flow leakage, and we define and prove non-interference. By combining results from static analysis of security levels, we can improve run-time efficiency by avoiding wrappers where they are superfluous according to the over-approximation of levels given by the static analysis.

ACKNOWLEDGMENTS

We thank Christian Johansen for discussions and comments that have greatly improved the manuscript.
This work was partially supported by the project IoTSec, *Security in IoT for Smart Grids*, with number 248113/O70 the Norwegian Research Council, and by the project SCOTT (www.scott-project.eu) funded by the Electronic Component Systems for European Leadership Joint Undertaking under grant agreement No. 737422.

REFERENCES

[1] Gul A Agha. 1985. *Actors: A model of concurrent computation in distributed systems.* Technical Report. Massachusetts Inst. of Tech, Cambridge Artificial Intelligence Lab.

[2] Isabelle Attali, Denis Caromel, Ludovic Henrio, and Felipe Luna Del Aguila. 2007. Secured information flow for asynchronous sequential processes. *Electronic Notes in Theoretical Computer Science* 180, 1 (2007), 17–34.

[3] Henry C Baker Jr and Carl Hewitt. 1977. The incremental garbage collection of processes. *ACM Sigplan Notices* 12, 8 (1977), 55–59.

[4] Frank De Boer, Vlad Serbanescu, Reiner Hähnle, Ludovic Henrio, Justine Rochas, Crystal Chang Din, Einar Broch Johnsen, Marjan Sirjani, Ehsan Khamespanah, Kiko Fernandez-Reyes, and Albert Mingkun Yang. 2017. A survey of active object languages. *Comput. Surveys* 50, 5 (2017), 76.

[5] Stephan Brandauer, Elias Castegren, Dave Clarke, Kiko Fernandez-Reyes, Einar Broch Johnsen, Ka I Pun, S Lizeth Tapia Tarifa, Tobias Wrigstad, and Albert Mingkun Yang. 2015. Parallel objects for multicores: A glimpse at the parallel language Encore. In *International School on Formal Methods for the Design of Computer, Communication and Software Systems (*Lecture Notes in Computer Science*)*, Vol. 9104. Springer, 1–56.

[6] Denis Caromel, Christian Delbé, Alexandre Di Costanzo, and Mario Leyton. 2006. ProActive: an integrated platform for programming and running applications on Grids and P2P systems. *Computational Methods in Science and Technology* 12, issue 1 (2006), 16.

[7] Denis Caromel and Ludovic Henrio. 2005. *A Theory of Distributed Objects: Asynchrony-Mobility-Groups-Components.* Springer.

[8] Dorothy E Denning and Peter J Denning. 1977. Certification of programs for secure information flow. *Commun. ACM* 20, 7 (1977), 504–513.

[9] F Durán, S Eker, P Lincoln, N Martí-Oliet, J Meseguer, and C Talcott. 2007. All about Maude: A high-performance logical framework. *Lecture Notes in Computer Science* 4350 (2007).

[10] Joseph A Goguen and José Meseguer. 1982. Security policies and security models. In *Security and Privacy, 1982 IEEE Symposium on.* IEEE, 11–11.

[11] Philipp Haller and Martin Odersky. 2009. Scala actors: Unifying thread-based and event-based programming. *Theoretical Computer Science* 410, LAMP-ARTICLE-2008-003 (2009), 202–220.

[12] Robert H Halstead Jr. 1985. Multilisp: A language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 7, 4 (1985), 501–538.

[13] Einar Broch Johnsen, Reiner Hähnle, Jan Schäfer, Rudolf Schlatte, and Martin Steffen. 2011. ABS: A core language for abstract behavioral specification. In *Formal Methods for Components and Objects (*Lecture Notes in Computer Science*)*, Vol. 6957. Springer, 142–164.

[14] Einar Broch Johnsen and Olaf Owe. 2007. An asynchronous communication model for distributed concurrent objects. *Software & Systems Modeling* 6, 1 (2007), 39–58.

[15] Farzane Karami, Olaf Owe, and Toktam Ramezanifarkhani. 2019. An evaluation of interaction paradigms for active objects. *Journal of Logical and Algebraic Methods in Programming* 103 (2019), 154 – 183. https://doi.org/10.1016/j.jlamp.2018.11.008

[16] Jonas Magazinius, Phu H Phung, and David Sands. 2010. Safe wrappers and sane policies for self protecting Javascript. In *Nordic Conference on Secure IT Systems.* Springer, 239–255.

[17] Olaf Owe and Toktam Ramezanifarkhani. 2017. Confidentiality of Interactions in Concurrent Object-Oriented Systems. In *Data Privacy Management, Cryptocurrencies and Blockchain Technology (*Lecture Notes in Computer Science*)*, Vol. 10436. Springer, 19–34.

[18] Olaf Owe and Gerardo Schneider. 2009. Wrap your objects safely. *Electronic Notes in Theoretical Computer Science* 253, 1 (2009), 127–143.

[19] Toktam Ramezanifarkhani, Olaf Owe, and Shukun Tokas. 2018. A secrecy-preserving language for distributed and object-oriented systems. *J. Log. Algebr. Meth. Program.* 99 (2018), 1–25. https://doi.org/10.1016/j.jlamp.2018.04.001

[20] Alejandro Russo and Andrei Sabelfeld. 2010. Dynamic vs. static flow-sensitive security analysis. In *Computer Security Foundations Symposium (CSF), 2010 23rd IEEE.* IEEE, 186–199.

[21] Andrei Sabelfeld and Andrew C Myers. 2003. Language-based information-flow security. *IEEE Journal on selected areas in communications* 21, 1 (2003), 5–19.

[22] Andrei Sabelfeld and Alejandro Russo. 2009. From dynamic to static and back: Riding the roller coaster of information-flow control research. In *International Andrei Ershov Memorial Conference on Perspectives of System Informatics.* Springer, 352–365.

[23] Marjan Sirjani, Ali Movaghar, and Mohammad Reza Mousavi. 2001. Compositional Verification of an Object-Based Model for Reactive Systems. In *Proceedings of the Workshop on Automated Verification of Critical Systems (AVoCS'01), Oxford, UK.* Citeseer, 114–118.

[24] Marjan Sirjani, Ali Movaghar, Amin Shali, and Frank S De Boer. 2004. Modeling and verification of reactive systems using Rebeca. *Fundamenta Informaticae* 63, 4 (2004), 385–410.

[25] Dennis Volpano, Cynthia Irvine, and Geoffrey Smith. 1996. A sound type system for secure flow analysis. *Journal of computer security* 4, 2-3 (1996), 167–187.

[26] Derek Wyatt. 2013. *Akka concurrency.* Artima Incorporation.

[27] Yasuhiko Yokote and Mario Tokoro. 1987. Concurrent Programming in Concurrent SmallTalk. In *Object-oriented concurrent programming.* MIT Press, 129–158.

[28] Akinori Yonezawa (Ed.). 1990. *ABCL: An Object-oriented Concurrent System.* MIT Press, Cambridge, MA, USA.