Information Flow Control of Component-based Distributed Systems

Lilia SFAXI1,2,*, Takoua ABDELLATIF3, Riadh ROBBANA1, Yassine LAKHNECH2

1 LIP2 Laboratory, Faculty of Science of Tunis, Campus University, 2092 Manar 2, TUNISIA
2 Verimag Laboratory, Centre Equation 2, avenue de Vignate, 38610 Gieres, FRANCE
3 University of Sousse, ESSTHS, Lamine Abassi Street, 4011 Hammam Sousse TUNISIE

SUMMARY
Non-interference is a strong security policy that enforces the confidentiality and integrity. Many solutions are proposed in the state of the art for verifying this policy in programs, but few tools are proposed to implement it. In this paper, we define a high level model called CIF (Component Information Flow) and we develop the tools that simplify distributed secure system development and deployment. The developer focuses on developing the functional part of his system and defines his system security properties using a high level configuration. When this configuration is validated, that is no security leak is detected, the system security code is generated. To validate and demonstrate the effectiveness of our approach, we apply the CIF tools to a classical Web Service use case.

1. INTRODUCTION
Building distributed systems that satisfy end-to-end security requirements is a difficult task. Most existing techniques concentrate on access control policies and security protocols that are essential for ensuring data confidentiality and integrity but do not provide end-to-end security guarantees. Access control policies do not track information flow through the entire system and do not cope with implicit information flow. Similarly, cryptographic components are used for secure communication and authentication but do not guarantee any global security properties. Information flow policies [1, 2, 3, 4] are natural for specifying end-to-end confidentiality and integrity requirements because they put global constraints on the flow of information. For example, an access control policy can require that only users with the appropriate read rights can read file f; while an information-flow policy would require that only users with the required security level can get any information about the content of file f even indirectly. Security-typed languages [5, 6, 7, 8] provide a promising framework for describing and implementing such policies. In this framework, types express restrictions on the flow of information. Typing annotations can be used at compile time to check that the program respects the information-flow constraints or at run-time to enforce such constraints. To date, most security typed languages have addressed systems implemented on a single trusted host. In distributed systems, multiple hosts cooperate in order to implement a function. This entails that data and computation are distributed among several, often distant hosts. Moreover,
the dominant work on building distributed secure systems is devoted to access control policies. A notable exception is the approach proposed by the JIF group in [8]. In this work, a program partitioning algorithm is presented. It uses security types in order to split data and computations on heterogeneously trusted hosts while respecting the original security requirements. The distributed system generated by JIF/Split is secure under the assumption that the communication infrastructure is secure, that is, sent messages are confidential and arrive at destination unaltered and in-order. In this paper, we take a complementary view to JIF/Split and develop the Component Information Flow (CIF for short) framework. Our starting point is a specification of the architecture of the system that describes the components of the system and their interaction. Each component includes a set of required and provided ports, respectively. An output port of a component can be linked to an input port of an other component; this defines a link between these components. Each component is implemented with a code describing its behaviour and has an interface describing how it can be connected to other components. CIF components are also equipped with a security interface, i.e., inter-component links are tagged with security labels that express information-flow policies. More specifically, each port $p$ is tagged with a security label $(S(p), I(p))$, where $S(p)$ is the confidentiality label (S for Secrecy) and $I(p)$ the integrity label. In case $p$ is an output port of a component $C$, the security label $(S(p), I(p))$ expresses the requirement that data sent via port $p$ must be protected such that only components with confidentiality level higher than $S(p)$ are allowed to read this data. Moreover, the data sent can only be used by components who require less integrity than $I(p)$. A security-typed architecture is well-typed if the components interconnection is consistent with the security-types: consider a component $C_1$ with output port $p$ connected to an input port $q$ of component $C_2$. Then, the conditions $S(p) \subseteq S(q)$ and $I(p) \supseteq I(q)$ must hold. The condition $S(p) \subseteq S(q)$ means that $C_2$ treats data received on port $q$ as at least as confidential as $S(p)$, thus in particular, $C_2$ does not send this data on a port with less confidentiality. The condition $I(p) \supseteq I(q)$ means that $C_2$ may assume that the received data has at least the integrity level $I(q)$. The significance of the confidentiality and integrity levels as well as their adopted classification is explained in the subsection 3.2. The consistency of the inter-components security types does not guarantee that there is no undesirable information-flow, since it entails restrictions on the behaviour of the components that must be verified. This is achieved by the intra-component type system that verifies that each component satisfies its security interface, i.e., the security labels attached to its ports. The problem that remains to be solved now is to generate code that implements the functional behaviour of the components but also implements the security interfaces. To this end, we develop a system architecture transformer (SAT) that takes as input an architecture with security tags and generates a target system architecture where the security tags are implemented using cryptographic primitives. In our current implementation, we use strongly secure (IND-CCA) asymmetric encryption and (EF-CMA) digital signature to implement confidentiality and integrity requirements, respectively. We demonstrate the feasibility of our approach on two case studies: we reimplemented the battleship game, a prototype developed by JIF team, in order to compare CIF with JIF; and a real application to show CIF usage in existent distributed applications.

The rest of the paper is structured as follows. Section 2 presents the related works. In Section 3, we present the security model. In Section 4, we present our model for component-based information-flow secure distributed systems. In Section 5, we present our code generation strategy. We present the Web service application case study in Section 6. Section 7 discusses the performance evaluation of CIF tools. And finally, in Section 8, we present some concluding remarks.

2. RELATED WORK

There are two primary areas of research related to this work: component-based system design and language-based information-flow. We focus in the following discussion on work that falls in both areas.

Security-typed languages: Information flow analysis at compile-time has a long history. Denning [9] originally proposed a language to enable static verification. Other researchers used type systems and automatic or semi-automatic tools to control information flows[2, 5, 6]. Recently, there has been
more interest in provably-secure programming languages that include cryptographic functions [10, 11, 12, 13, 14]. Except [11], which considers a secure implementation of abstract communicating processes, all the others consider an imperative sequential language. Our implementation strategy is inspired by this work. By comparison with information flow control languages like JIF[15], the security properties in CIF are described at the architectural level, then the appropriate security labels are propagated in the component’s implementation.

[8] shows an application of JIF to distributed systems, called JIF/Split. In JIF/Split, the compiler parses a security-annotated code and partitions it to threads so that each thread represents a non-interferent part of the whole program. Nevertheless, it considers that the network is secure. Likewise, [16] recently presented the same approach, with the advantage of adding cryptographic mechanisms to enforce the communication security. The drawback of this solution is that the architecture of the system is guided by the security aspect. Our approach is different, as our starting point is a component-based distributed application, while the security configuration is represented as a set of parameters expressed by annotations at the architectural level.

Information flow control in operating systems: Some operating systems like Flume[17], HiStar[18] and Asbestos[19] perform information flow control between processes by assigning labels to processes and messages. DStar[20] extends HiStar for distributed systems and adds an exporter for each machine, the only process that controls the information exchange between distant processes. These solutions are interesting, as they are transparent to the developer. However, they have some performance issues and are complicated to apply for other operating systems. CIF can be applied to all distributed applications, on any operating system. Moreover, the information flow control in CIF is enforced by automatically generated cryptographic mechanisms between distant components.

Secure component-based system models: Some component-based and model-oriented approaches are used to guarantee systems security[21, 22, 23]. These approaches, like ours, take advantage of the architecture description language in component-based systems to describe non-functional properties like security, and generate the appropriate code. Particularly, the Service Component Architecture (SCA) specification[24] secures the bindings between components by configuring the services and references, respectively representing server and client ports, thanks to intents. However, only access control is mostly considered. Our work provides automated non-interference verification.

3. SECURITY MODEL AND ADOPTED TOOLS

In this section, we present the security properties we address in our work and the security tools that CIF is based on.

3.1. Security Properties

Securing a distributed system means ensuring two properties: (1) secret information has to remain secret (confidentiality property), and (2) only authorized principals can modify data content(integrity property). Nevertheless, confidentiality and integrity for standalone data are most of the times insufficient to guarantee that the overall system is secure. Constraints on information flow can enforce both the confidentiality and the integrity by observing and tracking the whole system’s behaviour and controlling the information’s propagation. The information flow property we are interested in is called non-interference.

Non-interference is a strict multilevel security policy model that controls information flow by stipulating that no information can be transmitted from secret to public data. This property was defined initially by Goguen and Meseguer in 1982 [2] saying that “Low-security behaviour of the program is not affected by any high-security data”. Foccardi and al. [25] define it by saying that “The low level users should not be able to deduce anything about high level user’s activity”.

Although non-interference can be intuitive, it can also be very strict, and it is hard to find realistic applications with no interfering data. For example a classical password verification communication
can leak the confidentiality of the secret password. In many cases, a malicious program can have an important probability to calculate a password from the answers of the server saying whether a password is correct or not. In other words, there is an information flow from the public data (server’s response) to the secret one (the password).

Several research teams have developed tools and languages for securing programs development from non-interference. In our work, we use JIF (Java Information Flow) language.

### 3.2. JIF

JIF\[15\] is a security-typed programming language that extends Java by adding security labels for information flow control and access control. The JIF compiler checks that the program does not infringe the confidentiality and integrity policies specified. Afterwards, the compiler translates the code to a Java program that will be compiled by ordinary Java compiler to produce an executable program secure by construction. The security policies are expressed using the Decentralized Label Model (DLM). The labels used in this model are composed of a pair of confidentiality and integrity policies. The main entity used to express these policies is the principal. A principal is an entity with power to observe and change certain aspects of the system. Principals are ordered using the can act for relation, which is a delegation mechanism that enables a principal to pass some of his rights to another.

Confidentiality policies are expressed using principals, by defining the owner of the policy and the set of possible readers of the information: it is written \( o \rightarrow r \) where the principal \( o \) is the owner of the information and the principal \( r \) is the specified reader.

A label, being a pair of confidentiality and integrity policies, is written \( \{c;d\} \), where \( c \) is the confidentiality policy and \( d \) the integrity policy. For example, if we consider the following instruction:

\[
\text{int}\{\text{Alice} \rightarrow \text{Bob}; \text{Alice} \leftarrow \ast\}a;
\]

Associating the label \( \{\text{Alice} \rightarrow \text{Bob}; \text{Alice} \leftarrow \ast\} \) to the variable \( a \) means that the information in \( a \) is controlled by the principal Alice, and that Alice permits this information to be seen by the principal Bob (and of course all the principals acting for Bob and Alice). It means also that Alice permits to no-one else but her (and the principals acting for her) to modify the information in \( a \) (as the \( \ast \) here stands for the top principal, which means the principal able to act for all principals).

We order the labels using the no more restrictive than relation, represented by the \( \subseteq \) symbol. As in \[26\], the no more restrictive than relation orders these labels to form a security lattice, which represents the set of security levels. Let’s suppose that \( x \) and \( y \) are both elements in the confidentiality lattice (respectively in the integrity lattice). If \( x \subseteq y \), then data at level \( x \) is no more confidential (no less trustworthy) than data at level \( y \). Figure 1 shows an example of a security lattice, which represents some security policies (confidentiality and integrity) written with the label model described above. Three principals are used here: Alice, Bob and Chuck. The lattice shows the relation of restrictiveness between some of the labels. This lattice orders the labels (from the bottom to the top) from the least restrictive to the most restrictive. The top principal is represented by the symbol: \( \ast \), and the bottom principal (the principal that every other principal acts for) is represented by the symbol: \( \perp \).

Defining the security lattice enables to specify a condition on the information flow: as a data circulates in a system, its labels can only become more restrictive. Satisfying this condition guarantees that no secret information is leaked to low level users, and that no trustworthy information is modified by an untrustworthy user.

In our work, we apply this label model to distributed systems, represented by components. The component-based model we use is presented in the following section.
4. CIF COMPONENT MODEL

A component is a unit of composition that can be deployed independently and composed with other components. Thanks to the modularity they offer, components simplify the development and the management of distributed systems. Many works show the component role in automating the management of distributed systems [27]. The configuration of the distributed system can be expressed in the Architecture Description Language (ADL) and the system is automatically deployed on distributed hosts. Each component can be separately configured through its configuration interfaces that allow setting values to the component attributes. Moreover, the components are loosely coupled which means that the connections between different components, called bindings, can be established in different ways and independently of the components code. These properties attract many industrial and research communities to simplify building and managing their systems. For example, Web services designers currently adopt components to construct services and ensure their interoperability [28].

In the framework CIF, we use component-based models to express the system security properties in addition to its architecture. Indeed, in the CIF Architecture Description Language, each control interface can allow for setting the component security level of data used in the component code. The communication ports configuration positions the constraints on the communication protocols and the security of the exchanged data between components. Our tools can be applied to any kind of component model that considers a distributed system as an assembly of components explicitly bound. In our prototype, we consider two component-based models: Fractal [29] and Service Component Architecture (SCA) [24].

Figure 2 presents an example of a component-based system. C1, C2, C3 and C4 are four components. Each Ci has two kinds of interfaces: Control interfaces li allowing to configure the security level of the component parameters and the communication ports Pi allowing data to be sent from a component to another. The communication ports can be connected with explicit bindings. In the example, C1 is bound to C3 which means that C1 can send data to C3 using ports P1 and P3. There are two kinds of ports: client ports sending requests and server ports receiving requests. For example, in the component C3, P3 is a server port and P’3 is a client port. A port can be bound to several ports, for example P2 is bound to P’4 and P”3. This means, that a same request is sent to C3 and C4 at each time. C2 stores in a list the answer of C3 and C4.

CIF (Component Information Flow) is a high level model designed to represent distributed systems using component-oriented description, and to assign security parameters to the system’s elements at a high level of abstraction. It provides a set of tools which enable to validate this high-level security configuration, and to generate necessary low-level system code. To build a distributed
system with CIF, we use components. The components are connected using asynchronous, unidirectional bindings. They exchange messages via communication ports, labelled using the decentralized label model. The label applied to a port is applied to all the messages sent (or received) via this port. Each component may have a set of attributes, configured by configuration interfaces, and having each a security label.

4.1. System Security

In our work, annotating a component consists in assigning security labels to the interfaces of the component, which means assigning security levels to the attributes of the component and to the outgoing and incoming messages through the communication ports. This annotation is done in the ADL file, which is the XML representation of the system’s architecture (see Section 4.2). The annotation mechanism obliges the system to respect the security restrictions at two levels:

1. The system must satisfy the security properties at a local level: the component itself is required to guarantee that the information flow in its implementation does not break any of the confidentiality, integrity or non-interference properties required by the user. This is the *intra-component security*.
2. The system must guarantee the security of exchanged messages between remote components on an untrusted network; no attacker can succeed in accessing secret data or modifying trustworthy messages, even indirectly. If this happens, it is detected by the target component. This is the *inter-component security*.

4.1.1. Intra-Component Security

The component code has to respect the security properties specified by the user; which means that its implementation code preserves the confidentiality and integrity of the data configured in the ADL files. The information flow needs to be controlled not to infringe the non-interference property (see Section 3.1). The intra-component security is verified for each component by applying the intra-component code generator on the implementation code of the component. The behaviour of the generator is described in the Section 5.1.

4.1.2. Inter-Component Security

In order to define the security constraints on messages exchanged between components, we annotate the communication ports with labels. The semantics of the label differ whether the port is a server port or a client port. Furthermore, it is possible to group a set of components into domains. Within a domain, the components can share a common security policies as sharing a common secret key. In this paper, we handle standalone components without domains;
the same code generation mechanisms can be applied to the specific case of components within domains.

For example, in Figure 2, P1 of component C1 is a client port (represented by convention at the right of the component) and P3 of component C3 is a server port (represented at the left). More generally, annotating a client port P of a component C1 with the label \{S;I\} and a server port P’ of a component C2 with \{S’;I’\} means the following:

- From the confidentiality point of view, having the confidentiality level S for the port P means that C1 wants the message it sends through P to keep the confidentiality level S. As for P’, having the confidentiality level S’ means that C2 considers that the received message has the confidentiality level S’.
- Dually, from the integrity point of view, having the integrity level I for the port P means that C1 guarantees that the message it sends through P has the integrity level I. As for P’, having the integrity level I’ means that it wants the message received through P’ to have at least the integrity I’.

Considering this definition, we can attest that a binding is possible between P and P’ if and only if the label of P is no more restrictive than the label of P’; which coincides with the condition saying that the data only flows to more restrictive targets (see Section 3.2).

4.2. CIF ADL

An ADL description represents the desired architecture of the system, that is, the components and their bindings. In addition to the architectural description, CIF ADL allows specifying the labels of the component data (the attributes), the labels of the communication ports and the desired communication protocol. The protocol description guides the administrator to select the communication security protocol between components. Furthermore, the bindings in CIF ADL are considered unidirectional since we have to distinguish between a request message and its answer that can have different security constraints. Once the user specifies the functional and security properties of his system, using CIF, and writes the implementation code of each component using his target component-based modelling language, he can start up the generation tools. Figure 3 represents an example of a CIF ADL file.

The following part presents the running example we use in our paper: the battleship game. This game was initially implemented with JIF[15]. We decomposed it into components and rewrote it with CIF, by assigning security levels to the ports and attributes.

Figure 3. Example of CIF ADL

```xml
<component name="C">
  <component name="C1">
    <port name="P" role="client"
      signature="security.PItf"
      label="\{S;I\}" />
    <attribute name="M" label="\{Sm;Im\}"
      value="Mon Message"/>
    <content class="security:C1Impl"/>
  </component>
  <component name="C2">
    <port name="P'" role="server"
      signature="security.PItf"
      label="\{S';I'\}" />
    <content class="security:C2Impl"/>
  </component>
  <binding client="C1.P" server="C2.P'">
    <protocol name="RMI"/>
  </binding>
</component>
```
4.3. Example of the Battleship Game System

4.3.1. System Description  The Battleship game is a distributed composite composed of at least three components: two players and a coordinator. Each player owns a secret board containing a fixed number $n$ of ships, having each a coordinate, chosen randomly at the beginning of the game. Each player tries to guess the coordinates of the adversary ships, and the winner is the one who guesses the first the coordinates of the $n$ ships of his adversary. A coordinator keeps a copy of the players’ boards, to check that no player has changed his ships while the game is running. To control the game, the coordinator intercepts all the messages of the players, working as an intermediate for all requests and responses. Figure 4 represents the system corresponding to the scenario described above. The communication protocol is done as follows:

- The top component Battleship starts the game. The coordinator is then initialized, and initializes both the Player1 and Player2.
- The coordinator asks for the boards of the players by sending a request using the P13 and P23 client ports.
- The player1 sends its board to the coordinator using its port P2.
- The player2 sends its board to the coordinator using its port P2.
- The coordinator stores the boards, and asks for the query of the player1, using the P14 client port getRequest.
- The player1 sends his request using the port P1.
- The coordinator sends the request of the player1 to the player2 using the port P21.
- The player2 sends its response to the coordinator using the port P3.
- The coordinator verifies that the response of the player2 is correct. If so, it sends this response to the player1 using the port P12. If not, the coordinator declares that the player2 is a fraud, and stops the game.
- The game continues in the same way, until one player finds all the ships of his adversary. In that case, the coordinator declares that he is the winner.

4.3.2. Security requirements  Let’s consider the example of the port Player1.P2 (sendBoard(1)). The board being sent to the coordinator must have the highest confidentiality level, as it must not be seen by the opponent, and the lowest integrity level, as it must not be modified by anyone. In our example, we use the decentralized label model (DLM) [30] to represent the labels. We consider that each component is a principal. For the port Player1.P2, the board that will be sent is the property of the Player1, and the Player1 authorizes only the Coordinator to see the content of the board. On the other hand, the board is tainted with the Player1’s seal only, since he is the only one that has modified it. That’s why we associate to the client port the label \{Player1 → Coord; Player1 ← \}. 

[Diagram of the Battleship game system is shown here.]

The coordinator component, when receiving the board, will consider that he is the only one who can see or modify it, besides the Player1. That’s why we associate to the server port the label \{Player1 \rightarrow Coord; Player1 \leftarrow Coord\}. The adequate CIF ADL file for this example is represented by Figure 5.

The first thing the CIF tool verifies is that the ports at the ends of the binding are compatible. That is, it verifies that the label of the server port Coordinator.P’12 is at least as restrictive as the client port Player1.P2; which is the case in our example: the confidentiality level is the same for both ports, and the integrity level of Player1.P2 (Player1 \leftarrow *) is less restrictive than that of Coordinator.P’12( Player1 \leftarrow Coord). In fact, by the integrity tag on his port, Player1 testifies that nobody has modified his data but himself. On the other hand, the Coordinator, with his integrity level fixed to Player1 \leftarrow Coord, requires that the received data be tainted only by the Player1 or himself. In this case, a data that is modified only by the Player1 is acceptable.

Once this verification performed for all ports, the cryptographic components are generated between the functional components, to ensure encryption and signing of the data that circulates in an untrusted environment.

5. CODE GENERATION

In this section, we explain how we generate executable code for CIF architectures. We need to generate code for each component as well as code for secure communications between components. The code generated at the level of each component is a Java code annotated with security labels, specifically a mild extension of JIF. The code for inter-component communication is not generated directly. Instead, we transform the initial CIF architecture into a degenerated one, where all ports have the lowest security label. In addition, the labels initially specified by the designer of the application are implemented using new local components that host the cryptographic primitives. These components ensure that the security of the original CIF architecture is preserved and actually applied. Figure 6 shows the main steps done by the generators, starting from a CIF ADL description.

5.1. Intra-Component Verification and Generation

The intra-component security is applied by scanning the implementation code of each component and verifying that the high level annotations are preserved: it verifies that confidentiality and
integrity are kept, but also that the information flow inside the component is not interfering. The verification of intra-component security depends firstly on the implementation code.

- If the component code is written using a security-typed language (ex. JIF [15]), the tool performs a simple type-checking.
- If the component behaviour is implemented using an imperative language (like Java or C), it will perform a label propagation.

The goal of the label propagation is to verify that the component code does not infringe any of the security constraints defined by the user. The result of this operation is an annotated code that associates a label to every variable, method and parameter of the component implementation. If the label propagation succeeds, that is, if the annotated code is generated, it means that the information flow inside the component respects the user-defined security constraints and that the code is non-interferent. In our prototype, the goal is to generate JIF classes, by propagating labels using the Polyglot compiler.

5.1.1. The Polyglot Compiler

Polyglot[31] is an extensible compiler framework built to create language extensions without duplicating code, and to create compilers for languages similar to Java. The Polyglot compiler is structured as a set of passes over source files that ends with the output of Java source code. The passes parse the original source language and create an AST (Abstract Syntax Tree), rewrite the AST to eliminate any ambiguities, type check it, possibly rewrite it to another AST, then output the result as Java source code. Polyglot is composed of a base compiler, called JLC (Java Language Compiler), which parses and checks a Java source code and outputs the same source code: it implements the identity translation. The Polyglot aims initially to create extensions to Java, by defining a new grammar, based on the Java grammar, but adding the needed features. For example, the JIF language extends Java by adding labels, so the compiler scans the JIF code, label checks it, and creates the equivalent Java code. In our case, what we will perform is the exact opposite: we start from a simple Java code, and we want to create a JIF code. In that case, we will not use polyglot in the standard way. We will use JLC to parse the initial Java code and to create the AST. But what is actually extended is the code generator of polyglot: instead of generating Java, we...
generate JIF. This generation is done by creating a set of visitor\(^1\) classes. Visitor classes iterate over the abstract syntax tree. These iterations help us travel through the tree to find the declarations, use and modification of data. We explain in the following paragraph the detailed transformation steps to generate the secured JIF code.

5.1.2. Transformation Steps The tool starts by parsing the code and generating an abstract syntax tree. We define a set of visitor classes to travel all over the AST. The tool proceeds as follows:

- A first visitor parses the CIF ADL file and extracts all the labels of attributes and ports. It then goes through the AST to assign to every declaration of the attributes (represented by Java Fields) and ports (represented by Java Methods) the adequate label.
- A second visitor scans the AST, to parse methods’ implementations. For each method:
  - It looks for the variables called in the return statement of the method, if any. If it is a local variable, it inherits the label of the method. If it is an attribute, it compares their labels: the label of the method must be at least as restrictive as the label of the attribute.
  - It looks for every attribute assignment and label-checks the expression at the right of the assignment, that is, checks that the resulting label of the expression is at most as restrictive as its own.
  - It looks for every local variable assignment and label-checks the expression at the right of the assignment.
  - It looks for every method call and label-checks the effective parameters.
- The next operation deals with implicit information flow. Implicit information corresponds to situations where the control flow of the program reaches an instruction with security level \(\ell_1\) while the value of the program counter depends on a variable of higher level of security. For example, consider the following piece of code and assume that the confidentiality level of variable \(l\) is low while \(h\)’s confidentiality level is higher. Obviously, the value of \(l\) at the end of the code leaks information about the value of \(h\): if \(h\) \{ \(l = \text{true; }\) \} A more interesting example is the following: if \(h\) \{ \(p.\text{send;}\) \} Here \(l\) is as above and \(p\) is a port whose security level can be low or high, it does not matter. This program is security unsafe, since by observing the communications on port \(p\), even if the communicated data is hidden, information about \(h\) is revealed. Our solution to implicit information flow is standard except that we have to treat send actions as having the lowest security label. That is, we ensure that the label of each instruction is at least as restrictive as the label of the program counter when the control reaches the instruction.
- A last visitor prints the resulting code by writing every label in front of its variable declaration, in a JIF-like manner.

The code generated by the generator is meant to be a JIF code, but for now, some features are not yet implemented, as for example how to handle exceptions. As JIF does not cover nested classes, initializer blocks, nor threads, our tool does not either.

The non-interference property is known to be very restrictive: most of the times, it is impossible to realize all the functional operations of the system without any interference between secret and public data. With our intra-component generator, if the propagation completes, the initial program is non-interferent, but the opposite is not always true. Sometimes, an interference must be allowed. That is why we need to loosen up our security constraints to obtain a realistic transformation. To do this, we use a local controller that decides, each time an interference is detected, whether this interference is authorized or not. If it is, the controller can declassify the information. For now, the decision of the controller is based on the choice of the user. In future work, the controller will check whether the leak of information detected may harm the security of the system in a significant way. For instance, we can check the robust declassification property [32] or the safety property[33].

\(^1\)Reference to the GoF Design Pattern Visitor
Once each component’s implementation translated into a JIF code, we obtain a set of secured and non-interferent components. In future work, a formal proof will show that an intra-component non-interference implies the system’s no-interference.

5.2. Inter-Component Verification and Generation

The inter-component security aims essentially to preserve non-interference at the transport level. For this purpose, the CIF tools perform the following steps:

5.2.1. Verification of information flows This step is about verifying that the information flows in the direction that preserves both its confidentiality and integrity. Formally, it means that for each binding linking a source port $P_S$ to a target port $P_T$, the relation: $\text{Label}(P_S) \subseteq \text{Label}(P_T)$ must hold. If not, the binding is considered interferent, and must be broken.

5.2.2. Insertion of Cryptographic Components Once all the bindings proved to be non-interferent, cryptographic methods are used to keep confidentiality and integrity of the sent data: for confidentiality, we use encryption, and for integrity, we use digital signature. This generation modifies the architecture of the system by inserting cryptographic components between the main components where it is required. A set of security protocol components are made available by CIF, in order to be inserted between the functional components. The choice of the security component to insert depends on the communication port labels and the user protocol (which is specified using the $<\text{protocol}>$ element in the CIF ADL). Of course, in order to enable this kind of operations, we suppose that the deployer is aware of the application domain. If, for example, the user chooses the RMI communication, the security protocol can be the insertion of encryption components for confidentiality preservation, and signature components for integrity preservation, as we will see in the battleship example (see Section 5.3). If the user specifies that the communication is done using the web services, the protocol consists on using the WS-policy standard for authentication and header and message encryption. The

5.2.3. Generation of Functional ADL The last step of the CIF tools is to create the equivalent functional ADL file that describes the architecture of the system. The transformation is performed by retrieving the security labels from the CIF ADL file and translating it to the functional model chosen by the system’s designer. This functional model can be any representation model that uses components, and that separates the architectural specifications of the system from its behaviour. Each ADL representation differs from the other by its syntax, like for example interfaces in Fractal ADL become references in SCA ADL, if we are talking about client interfaces, and services, if we deal with server interfaces. This is why each target ADL must have its appropriate generator, so that the CIF terminology is transformed to the equivalent one in the target language. In our prototype, we implemented two functional generators: CIF-to-Fractal, and CIF-to-SCA.

Fractal Fractal [29] is a software components model for the construction of adaptable and configurable distributed systems. It is used to implement, deploy and reconfigure systems and applications. It represents systems as a hierarchy of interconnected components. The main goals of Fractal are to reduce development, deployment and support costs of software systems.

Service Component Architecture SCA [24] is a common effort from vendors like BEA, IBM, Oracle, Sun and others, and is now an OASIS standard. It offers a set of specifications aiming to simplify the task of building service-oriented systems, independently from their implementations, and using components, each of them implementing a business logic. SCA’s aim is to simplify the creation, implementation and deployment of services in a Service Oriented Architecture [28] (SOA), by abstracting the details of the communication beneath.
5.3. Application to the Battleship Game

We apply our CIF tools to the Battleship game system presented previously. In this example, we use Fractal as a target model for components representation.

5.3.1. Intra-Component Transformation  Let us consider a portion of the Player’s code (Figure 7). In the CIF ADL, represented in the Figure 5, we notice the presence of the board attribute, labelled \{Player1 → Coordinator; Player1 ← \} (from now on, we will call this label \(L\)), that stores the player’s board. This board is sent via the port \(P2.sendBoard\). The interface \(P2\) is also labelled \(L\).

In this example, we run the Code Generator on the Player implementation code, written in the Java implementation of Fractal, called Julia [34]. We show how does the label propagation work for the initialization of the Player component.

The first thing the code generator makes is to assign the label to the \(board\) field and to the interface \(P2\) of the Player class. We remind the reader that assigning a label to an interface means assigning it to each message sent by the ports of the interface. For each method, the generator will search for the instructions using the field \(board\): it can be found in the line 5. In this line of code, the \(board\) parameter of the method \(setBoard\) is assigned to the \(board\) field: it must then have at most the label \(L\). Using the same logic, the generator searches for the uses of the interface \(P2\): we can find it at the line 30 where it is used to send the local variable \(myBoard\) to the Coordinator. The generator assigns then the label \(L\) to the local variable \(myBoard\). Next, the tool searches for the calls to the function \(setBoard\). It can be found in the line 29. In this function call, the effective parameter is the local variable \(myBoard\). This variable must then have a label at most as restrictive as the label of the formal parameter board, which is \(L\). As we have already labelled this variable with \(L\), and as we know that \(L \subseteq L\), the generator continues its parsing without throwing an exception. And so on. The propagation continues, until the code in Figure 8 is generated.

5.3.2. Inter-Component Transformation  The last step is the ADL file generation. The board, sent by the player to the coordinator, must be encrypted and signed, before being sent. In that case, no one but the coordinator can consult or modify it. For our use case, we define four different components:

- encrypt: encrypts the received message using the asymmetric encryption RSA.
• sign: signs the received message using the MD5 with RSA algorithm. A tag is added to the signature to preserve the message ordering and avoid replay attack.
• decrypt: decrypts the received encrypted message
• verify: verifies the received signed message

Starting from the CIF ADL file, the tool generates a system ADL file (for example Fractal ADL [29]) by doing the following:
• Creates two composite components, TPlayer1 and TCoord, with the same ports than the initial Player1 and Coord
• Creates inside TPlayer1 instances of both the Player1, encrypt and sign components, and binds them so that the client port of the Player1 is connected to the server port receive of the encrypt component, and the send port of the encrypt component is connected to the receive port of the sign component. The client send port of the sign component is connected to the client port of the TPlayer1 composite.
• The same thing is done respectively for the TCoord composite, the Coord, decrypt and verify components.
• The two composite components TPlayer1 and TCoord are connected with a low-level binding. The generator inserts the necessary components in the ADL file, so that the system becomes as illustrated in Figure 9.

In our prototype, the tool generates a Fractal ADL representation of the system. A portion of the resulting ADL file is represented in Figure 10.

6. SECURING A WEB SERVICE APPLICATION

In this section, we apply the flow control of CIF to control the secrecy of dynamically created data in a classical Web service application. Each data is equipped with a label specifying its classification with respect to different security categories. In our example, such a label would encompass the degree of privacy of the destination. As an example, let’s consider a client using web services to organize his journey to France. Living in Paris, he has to go to Grenoble for a secret mission and wants to keep his travel confidential. Using a graphical interface, he informs his 'Reservation service' about his intentions. This web service decomposes the problem into 'Travel service' and 'Payment service'. The 'Reservation service' contacts the 'Travel service' for the desired flights and obtains in return specific flight information and the corresponding prices. Afterwards, the 'Reservation service' selects a payment service which performs the payment of the selected flights. Figure 11 illustrates a representation of the distributed system. Following the Service Component
Architecture, each service is encapsulated in a component denoted R, T and P for respectively the Reservation service, the Travel service and the Payment service components. Two components can be bound by either a binding for exchanging public data or a binding for exchanging private data. The component R has two ports P1 with a label \( \{R \rightarrow T, R \leftarrow T\} \) and the port P2 with a label \( \{R \rightarrow \bot, R \leftarrow T\} \) (\( \bot \) being principal with the lowest integrity and confidentiality levels) that can be respectively bound to P'1 of the component T for confidential data and P'2 of the component T for public data. The choice between sending a message through P1 or P2 depends on the reservation message Label showing a confidential destination or not. If the message is desired confidential, it will be sent on P2, otherwise it will be sent on P1. The message is here composed of the customer name, the destination and the desired travel date.

In our Web service scenario, all the components’ ports are configured to ensure the integrity of data. We mean by integrity two properties: the message sender is the expected one and a message can not be resent by an attacker. To address these threats, the Web service security standards, the WS-Security [35] specification applies a combination of signatures and security tokens to demonstrate and verify the integrity of a message. Signatures verify the origin of a message and the security Tokens authorize the processing of the message based on the credentials associated with the message. Messages with invalid signatures and incorrect or missing tokens are rejected. The specification describes how to express such information in an XML format and how to include them in SOAP envelopes.

In order to implement these security properties, a set of inserted components allow to sign or encrypt messages. A 'sign' component implements the authentication and authorization mechanisms by including the corresponding tokens into the security header of the message. The choice between simple tokens (Username/Clear Password, Username/Password Digest), Binary Tokens (X.509 certificates, Kerberos) or XML Tokens (SAML assertions, XrML (eXtensible Rights Markup Language), XCBF (XML Common Biometric Format)) can be configured through the control interfaces and depends on the application context and required quality of service. The confidentiality property consists in introducing the 'encrypt' and 'decrypt' components between the Web Service components. The encryption algorithm (3DES, AES) is defined through the ADL file when configuring the 'encrypt' and 'decrypt' components. Furthermore, the choice of the key and its type (public or secret) is configured through the ADL description. The key management is out of
the scope of this current work; we suppose that keys are installed on the Web service machines for instance using TPM [36] at bootstrap time.

Since the destination to Grenoble is confidential, the reservation data is sent through the port P1 of the component R. This involves the encryption and the signature of the message by passing through the ‘encrypt’ and ‘sign’ components. The list of flights including their dates and prices returned by the ‘Travel service’ is sent on the P1 port of the corresponding component because it is considered confidential. Since a flight price is intuitively a public information, the system developer may send the price message on the public port P2 of the Payment service as illustrated in Figure 11. Nevertheless, when using CIF automatic tools, it is easy to detect that the price message label is confidential since this data is calculated from a confidential information: the flight message. Therefore, the safe configuration consists in sending it on a confidential binding on the P1 port of the Payment service.

This example shows that by using the CIF tools, the confidentiality of an information enforces automatically the confidentiality of data dynamically calculated by various web services when planning the trip. As a future work, we plan to handle other security properties in Web services such as security domains and trust delegation on real and scalable applications.

7. EVALUATION

In this section, we discuss the impact of adding information flow control to existent component-based systems using CIF tools.

7.1. Compilation Cost

The main contribution of CIF is its practical usage. Compared to JIF, the user has to annotate 11 parameters and ports against 143 with JIF.

As of the compilation time overhead, we evaluate the two phases of compilation: CIFInter and CIFIntra.

7.1.1. CIFIntra Compilation cost The complexity of our intra-generation tool can be evaluated by the number of passes over the implementation code. To propagate labels inside a component code,
the code is scanned as much times as the generated labels change. Indeed, the compiler stops when it reaches a stabilisation state, where labels no longer change.

Each time the label of a variable changes, the tool reverifies that this change did not affect the rest of the code. The number of passes over the code increases considerably with the code size, of course, but especially with the number and use of local variables. This is because one of the CIFIntra tool’s roles is to assign a label to these variables. The method implementation must then be scanned as much times as it takes to adequately label the variable. Let’s take for example the following code portion. \( a \) is a local variable, which means that it is initially not labelled. \( b \) and \( c \) are component attributes, having respectively the labels \( \ell_b \) and \( \ell_c \) (assigned by the designer of the application), where \( \ell_c \subseteq \ell_b \):

\[
\text{while(true)} \{ \ b = a; \ a = c; \}
\]

The first pass of the compiler finds that \( a \) does not have a label, gives it the label of \( c \) (because if \( a = c \), there is an information flow from \( c \) to \( a \), then \( \ell_a \subseteq \ell_c \) ) and continues without throwing any exception, even though it is clear for the reader that this is a case of interference, because at the second iteration of the loop, the value of \( c \) is assigned to \( b \). This is why the compiler needs to do at least two passes over this portion: in the second pass, it detects that the assignment \( b = a \) is illegal.

The number of passes depends also on the number of nested branches. In fact, the branches must be scanned recursively to verify that the program counter only becomes more and more restrictive, avoiding thus implicit flows.

Note that this algorithm terminates, because (1) the number of labels to modify is finite, (2) the label values change in an incremental way (go from less to more restrictive) and (3) labels are upper bound by the more restrictive label value defined by the user.

7.1.2. CIFInter Execution time CIFInter performs two main steps: (1) Reading the ADL file and verifying if the bindings respect the information flow restriction, (2) Generating the functional ADL description from the initial CIF ADL file by inserting cryptographic components that intercept any ongoing or incoming message.

Concerning the first part, browsing the initial CIF ADL file to extract architecture and security information is done once, and these information are stored in a set of objects created by CIF tools. Once all the information stored in these instances, their handling becomes very simple and fast. As for the label comparison, it amounts to a method call that takes almost no time.

The time needed for the generation of the functional description depends highly on the initial size of the CIF ADL file, and on the number of components and bindings. A template for the cryptographic components’ codes is given by the tool (the designer doesn’t need to write any cryptographic code) and the names of ports, attributes and interfaces are automatically adapted, according to the functional components, to avoid names replication. The result is a new functional ADL file and a set of new cryptographic components implementation files. The execution time of this part can grow linearly depending on the number of bindings, but remains reasonable, especially since this operation is made only once at compile-time.

Initially, CIFInter tool generates at least four security components by binding: one for encryption and one for signature on the client part, one for decryption and one verification on the server part. In order to improve the execution time of CIFInter and to limit the number of new generated components, we configured CIFInter to generate only one security component for each functional component: every outgoing and incoming message is intercepted by this component, which performs the encryption and signature (or decryption and verification) of the message before resending it. In this case, the time needed for generating a security component’s implementation increases, but as the number of generated components decreases a lot, the generation time drops considerably.

7.2. Overhead at Execution Time

The choice of the cryptographic method used for signing and encrypting messages has obviously an impact on the application’s performance. In the current prototype, all the messages are considered
as Java objects, and encrypted using the SealedObject class of the Java security API with the asymmetric cryptographic algorithm RSA, and a key size of 2048 bits. As for signing, the algorithm used is MD5/RSA, with the SignedObject class. The performance can be further improved if the designer chooses to use, for example, symmetric encryption instead of the asymmetric one currently implemented; or even more if hardware components[36] are used for key generation and sharing. This optimization is orthogonal to our work on the framework CIF.

In CIF, the separation of the security concerns from the functional code allows optimizing and modifying the cryptographic protocols without touching the functional code, but can induce an overhead at execution time. We measured this overhead, and found it negligible if compared to the cost of cryptography. Indeed, as the cryptographic components inserted are local components, the overhead induced is equivalent to that of a method call.

8. CONCLUSION

In this paper, we presented CIF, a component-based framework helping to secure distributed systems. Each entity must ensure that the local data handling do not infringe the needed properties, by verifying that the information flow maintains the confidentiality and integrity of information as specified by the user. Secondly, the messages sent through remote channels, which are most of the times untrusted, must be properly subject to cryptographic manipulations ensuring that no external attacker can have access to, or modify, secret information. The CIF framework provides a set of tools for building distributed systems secure by construction. The specification of the security properties are performed at a high level of abstraction and the security code is automatically generated. The paper presents two examples of middleware automatically secured with CIF. The first application allows to compare our tool with a language-based solution like JIF, and the second shows CIF usage in a real application. As a future work, we are interested in securing the runtime adaptation mechanisms of distributed systems. Indeed, the security properties need to be maintained when the system is adapted to external events like failures, performance degradation or security attacks. For example, in some distributed applications, when a faulty component is detected by the system, it is dynamically replaced by a new component. This replacement can cause an interference problem, because CIF tools are deployed only at compile time.

REFERENCES


