FlowR: Aspect Oriented Programming for Information Flow Control in Ruby

Thomas F. J.-M. Pasquier   Jean Bacon
University of Cambridge
{thomas.pasquier, jean.bacon}@cl.cam.ac.uk

Brian Shand
Public Health England
brian.shand@phe.gov.uk

Abstract

This paper reports on our experience with providing Information Flow Control (IFC) as a library. Our aim was to support the use of an *unmodified* Platform as a Service (PaaS) cloud infrastructure by IFC-aware web applications. We discuss how Aspect Oriented Programming (AOP) overcomes the limitations of RubyTrack, our first approach. Although use of AOP has been mentioned as a possibility in past IFC literature we believe this paper to be the first illustration of how such an implementation can be attempted.

We discuss how we built FlowR (Information **Flow** Control for **R**uby), a library extending Ruby to provide IFC primitives using AOP via the Aquarium open source library. Previous attempts at providing IFC as a language extension required either modification of an interpreter or significant code rewriting. FlowR provides a strong separation between functional implementation and security constraints which supports easier development and maintenance; we illustrate with practical examples. In addition, we provide new primitives to describe IFC constraints on objects, classes and methods that, to our knowledge, are not present in related work and take full advantage of an object oriented language (OO language).

The experience reported here makes us confident that the techniques we use for Ruby can be applied to provide IFC for any Object Oriented Program (OOP) whose implementation language has an AOP library.

Categories and Subject Descriptors D.2.2 [*Software Engineering*]: Design Tools and Techniques; D.2.4 [*Software Engineering*]: Software/Program Verification

Keywords Information Flow Control, Aspect Oriented Programming, Security

1. Introduction

In 2012 we developed a web portal, in collaboration with Public Health England, to grant access by brain cancer patients to their records [34]. As well as standard authentication and access control we used Information Flow Control (IFC) to track the flow of data end-to-end through the system. For this purpose, we used RubyTrack, a taint-tracking system for Ruby, developed by the SafeWeb project [17].

However, we came to realise certain limitations of the mechanisms we had deployed. For example, to enforce the required IFC policy, we manually inserted IFC checks at selected application component boundaries. In practice, objects and classes are the natural representation of application components within an object oriented language and it seems natural to relate security concerns with those objects. We should therefore associate the primitives and mechanisms to enforce IFC with selected objects. Furthermore, we wish to be able to assign boundary checks on any class or object without further development overhead. We also wish to be able to exploit the inheritance property to define rules that apply to categories of objects (for example defining a boundary check for all possible children of *I/O*). We therefore decided to investigate the use of Aspect Oriented Programming (AOP), and selected the Aquarium library [53], instead of RubyTrack, to use with our Ruby implementation to provide IFC-aware web applications.

We believe the techniques we have used to provide IFC mechanisms for Ruby can be extended to any Object Oriented Language (OO Language) with an AOP library, such as Java [23], C++ [43] or JavaScript [54]. AOP has advantages over our earlier approach: IFC label tracking and enforcement can be applied to any object and/or method invocation; programmers need have minimal concern about the underlying implementation; maintenance overheads are low, for example, when there are changes in the library code. These factors contribute to the overall reliability of software developed using AOP [49].

It has already been pointed out [11] that AOP can be used to implement security functions such as authentication and access control. Our main objective is to separate IFC concerns from the development of the application; we believe that functional issues and security issues should be kept well separated whenever possible. The AOP paradigm allows us to separate the core functionality developed by a programmer from the policy specified by a security expert [49]. Furthermore, the literature on providing IFC through a library [29, 31, 55] has already hinted that AOP techniques could be used to implement IFC.

However, we make some assumptions on the environment and the problems we are addressing. First, we assume that the developer is not adversarial; the aim is to protect against inadvertent disclosure of information through bugs within the application. Second, we focus on the design of web applications using a framework such as Sinatra or Rails to be, for example, deployed on a PaaS (Platform as a Service) cloud, using readily available languages/interpreters. Third, in this context, we assume the application's host ensures that no data can be disclosed outside of the application. Finally, we assume that the organisation running the application is willing to accept a performance overhead in exchange for increased security assurance. Other solutions can be envisioned for other circumstances, such as using a particular IFC-aware interpreter or running on an IFC-aware operating system. However, these would require control over the infrastructure that is not available in a standard hosting solution or PaaS, this would require the use of self-managed infrastructure or the use of IaaS (Infrastructure as a Service).

The Ruby standard implementation provides no real multi-threading support (more recent versions are starting to address this). Therefore, Ruby web servers tend to be multi-process rather than multi-threaded, which allows us to handle IFC rule violation effectively; we fail completely any process violating an IFC constraint. This is preferable to (per thread) exception handling which may generate implicit information flows [39]; this is discussed further in section 2.2.

Section 2 gives background on Aspect Oriented Programming and Information Flow Control. Section 3 then gives an overview of our work; we specify the basic principles governing flows and the primitives we added to the language to manipulate IFC labels. Section 4 describes our implementation. Section 5 presents a simple use case to demonstrate the simplicity of the approach then describes the web portal for brain cancer patients mentioned above. In section 6 we show that performance is similar to an equivalent solution and we argue that our approach provides better usability. Section 7 presents related work and section 8 summarises and concludes.

2. Background

Our paper targets readers interested in both AOP usages and IFC implementations. Some may not be familiar with both topics so we give a brief introduction to each, indicating the relevant literature. The last subsection discusses problems that are not addressed in this paper and which are generally not addressed by library-level implementations of IFC. We believe readers should be aware of these issues and options, and suggest further reading in section 7.2.

2.1 Aspect Oriented Programming and Security

Aspect Oriented Programming was introduced in 1997 by Kiczales et al. [22]. It is a programming paradigm extending Object Oriented Programming (OOP) by allowing cross-cutting aspects to be expressed. An aspect is a piece of code named an *advice* together with a *pointcut* determining when it should execute. The pointcut is used to determine the join-points (object methods) where the *advice* code will be executed. Fig. 1 provides an illustration of this concept. In the original specification an *advice* could be executed either *before* or *after* the join-point code is executed. The paradigm was later extended with an *around advice* [23] which has control over whether or not the join-point code should be executed.

An *advice* is composed of a primitive to express when the *advice* should be executed (i.e. *before*, *after*, *around*), a pointcut describing where the *advice* should be executed and a block of instructions to specify the behaviour of the *advice*. This is illustrated in Fig. 2 where we define an *advice* to be executed around a call to the method *write* of instances of *File*. The example is in Ruby and using the Aquarium library [53]. The parameters passed to the *advice* are the *join-point* to be executed, the *object* the method belongs to and the *arguments* passed to the method.

A pointcut can be made more expressive by using a regular expression (some implementations may not provide this, however this is provided by the library we are using) to define the methods and classes to which the *advice* should be applied. We can also specify a list of methods to be ignored, or implement different behaviour either after a normal execution or if the method throws an exception.

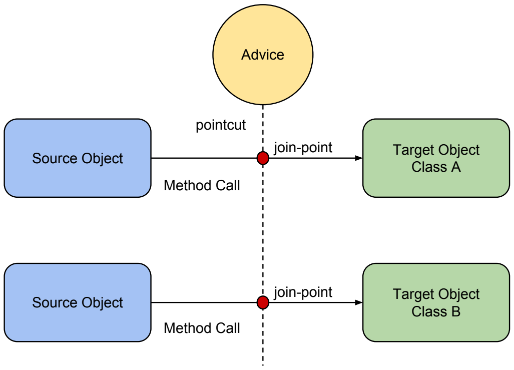

Figure 1. Visual representation of an aspect

2.2 Information Flow Control

It has long been argued that standard security techniques, such as firewalls and access control mechanisms, are not enough to prevent information leakage [9]. Indeed, it is beyond the scope of such mechanisms to determine whether, after the controls they impose, the information is used correctly. For example it is difficult to determine if the confidentiality of decrypted data is respected [39]. We therefore need to protect information flow, that is, how information is transmitted within and between applications.

In 1975, Denning [8] proposed a model to track and enforce rules on information flow within a procedural language. In this model, variables are associated with security classes. The flow of information from a variable a to a variable b is allowed only if the security class of b noted $\underline{b}$ is higher than $\underline{a}$. The security class of an n-function on a classified variable is noted $\underline{a_1} \oplus ... \oplus \underline{a_n}$. This allows the *no-read up, no-write down* principle [4] to be implemented to enforce secrecy. By this means a traditional military classification (public, secret, top secret) can be implemented. A second security class can be associated with each variable to track integrity (quality of data) [6] during *reading down* and *writing up*. Using this model we are able to control and monitor information flow to ensure data secrecy and integrity.

In 1999 Myers [33] proposed *security labels* to replace the security classes of Denning's model [9]. Clearance levels are considered too coarse-grained, permitting unnecessary access and were replaced by the "need-to-know" principle, also known as "Principle of Least Privilege (PoLP)" [41]. Labels are composed of tags representing categories of information or the nature of the information. The *secrecy* label is propagated with data between objects and the *integrity* label is used to define which data are allowed to flow into and within an object. Here, *integrity* relates to the trustworthiness of the source of any data rather than accidental corruption, for example, by hardware. A central authority is not needed in such a model since data flow policy is user-specified (discretionary) rather

```
around method: :write, type: File
                do |join_point, object, *args|
        puts ‘hello’
        returned_value = join_point.proceed
        puts ‘goodbye’
        return returned_value
end
```

Figure 2. Example: An advice in Ruby using Aquarium

```
y := x mod 2
```

Figure 3. Explicit information flow

```
x := x mod 2
y := 0
if x = 1 then y:= 1
```

Figure 4. Implicit information flow[39]

than centrally mandated. However, system support is needed at runtime for the continuous monitoring of data flows.

IFC implementations must ensure that labels can be allocated to principals but not be forged by them; can be allocated to data and "stick" to them; and that label checking enforces security policy regarding all aspects of information flow.

Practical IFC systems cannot work with policies that only allow data to become more restrictively labelled, for example secret data passed to a principal with top secret clearance becomes top secret when incorporated at that level. There are situations where constraints should be relaxed, for example, to enable the public release of previously classified data. The privilege to override secrecy IFC restrictions is known as the *declassification privilege*. In order to declassify an information item, the owner or owners must agree to remove their policy restrictions. This method of declassification again appears to remove the need for a central authority, as every owner is responsible for its own policy. But since the processes running on behalf of a principal o_i, or the precise hierarchy of principals, is only known at runtime, declassification also requires runtime support.

In this style of language a variable declaration can be augmented with an annotation to describe the policy associated with the data item. Examples can be seen in the solutions proposed by Denning [9] or Myers [33]. It is in these cases the programmers' responsibility to not only understand the algorithm being implemented but also the desired security policy [57]. But the security constraints may not all be clear during the functional design phase and inconsistencies can arise at runtime. It is generally better to separate security concerns from functional ones, limiting the impact they have on each other in the engineered system. We decided in this work to explore the use of AOP to enforce IFC constraints specifically in order to provide this separation.

2.3 Implicit Flow and Covert Channels

In this paper, as in most similar projects on IFC enforced at the library level, we do not address the problem of covert channels and *implicit* flow [2, 10, 15]. *Explicit* flows from x to y, noted $x \Rightarrow y$ are caused by passing data between variables, as illustrated in Fig. 3, or performing operations or method calls on such variables.

An *implicit* flow of information arises from the control structure of the program. Fig, 4 illustrates an *implicit* flow $x \Rightarrow y$ equivalent to the *explicit* flow illustrated in Fig. 3. It is possible to track such an assignment by introducing a process sensitivity level, as defined in the US DoD "orange book"[1], in which case the assignment of y can be detected at runtime. We could consider that any variable modified within the if statement (or any function called from it) must be assumed to create an information flow. However, in the case $x = 0$, no value is assigned to y and therefore no flow is detected even if it exists.

It is possible to prevent such flows remaining unnoticed by applying the label from the if to any assignment happening after the if statement. However, this means that the number of labels assigned to variables will increase [15], often unnecessarily. This leads to data with higher sensitivity than intended, known as *label creep* [37]. This phenomenon is illustrated in Fig. 5. From the Denning model, briefly described in section 2.2, we expect that $\underline{w} = \underline{x}$; that is, x and w are of the same security level. However, if we enforce process sensitivity levels, we have $\underline{w} = \underline{x} \oplus \underline{z}$ even if we know there is no $z \Rightarrow w$.

```
x := x mod 2
z := z mod 2
y := 0
w := 0
if x = 1 then y:= 1
if z = 1 then y:= 1
w:= x mod 2
```

Figure 5. Example of label creep

To address the concerns brought by the benevolent developer assumption, it has been suggested that an implicit flow can be prevented by the preemptive halting of program execution [2, 40]. However, this could prevent legitimate applications from terminating [2]. Therefore, to deal with potentially malicious code, variable-level runtime taint tracking can be combined with static analysis techniques [51].

At present in our project we do not consider implicit flow nor other covert channels [20] such as timing channels, storage channels [26, 27] or termination channels [52]. We briefly discuss in section 7 how some of these problems could be solved in an AOP context.

3. The FlowR IFC Model

IFC models are used to represent and constrain the flow of information within an application. In this paper, we focus on the aspects of the model relating to a single application rather than a distributed, multi-application environment.

In the DEFCon project [31], AOP was used with Java to enforce IFC by inserting IFC policy around selected methods. In FlowR, we extend those ideas by providing IFC at the level of objects, classes and methods, and provide basic primitives to enforce IFC. Our approach is not specific to Ruby but can be used with any OO Language that supports AOP. Furthermore, our techniques can work with an arbitrary library, without programmers having to know about its inner workings, so requiring little effort from them.

We provide tracking and flow control on what we define as *basic* variables (strings, integers, floats, etc.) and on arbitrary objects, classes or methods (as required).

In this section we first define the labels associated with objects. We then explain how the labels indicate flows that are and are not allowed and how labels are propagated for allowed flows. Finally, we outline how declassification is achieved.

3.1 Security labels

In order to monitor Information Flow we use labels. Our label model is inspired by that proposed by Efstathopoulos et al. [13].

Every tracked object is associated with two labels: a *Receive* label and a *Send* label. The *Receive* label is used to represent the type of information that is allowed to flow into an object, while *Send* labels are used to represent the nature of the information and its sensitivity. *Send* (S) labels are *sticky*, that is, they will propagate and taint any object they interact with, which ensures that no information can flow untracked. *Receive* (R) labels however do not propagate and concern only a single object or class.

[1] http://www.dtic.mil/whs/directives/corres/pdf/850001p.pdf

A label is composed of a set of tags, each representing an individual concern about the information, for example, the origin of the information, its privacy level or its owner. Tags are composed of two elements: a unique identifier t and a marker $+$ or $-$ representing the privileges an object has over the information labelled with this tag. To guarantee that each tag identifier is unique we represent tags using the Ruby concept of *symbol* which associates with a string an integer guaranteed to be unique in the current execution context. t^+ and t^- indicate the tag with identifier t and privileges $+$ or $-$ respectively.

In a *Receive* label. An object with a tag t^- in its *Receive* label is not allowed to receive information labelled with the tag t. An object with a tag t^+ in its *Receive* label is allowed to receive such information. *Receive* labels are not changed by the flow of information.

In a *Send* label. An object with a tag t^+ in its *Send* label is allowed to flow to an appropriately labelled destination object and the tag will propagate. An object with a tag t^- in its *Send* label is allowed to flow to an appropriately labelled destination object, but the tag does not propagate. For details on tag propagation see sections 3.2 and 3.4.

We need to add an additional constraint, that only one tag can be associated with some identifier t. This means that in any label L, either t^- or t^+ can exist, but not both. In order to simplify the notation for the rest of the paper, when we write $t \in L$, we mean $(t^+ \veebar t^-) \in L$.

We also have a special tag, named $default$. Strictly speaking, when unlabeled data are manipulated the empty labels are interpreted as *Send* label $S = \{default^+\}$ and a *Receive* label $R = \{default^+\}$, that is, such data can be freely transmitted. A label therefore implicitly has $default^+$ added to its tags, i.e. it is assumed to contain the default privilege $default^+$. In order to simplify the notation we can omit the default tag in a label. However, it is also possible to explicitly specify the default label. It is forbidden to set the tag $default^-$ in the *Send* label but it may be appropriate to set the tag $default^-$ in the *Receive* label, as we see in an example below.

3.2 Allowed flows and label propagation

We denote the flow of information between two entities A and B as $A \rightarrow B$. We need to define two rules, the first to describe an allowed flow and the second how tags in labels propagate between entities. We define $h(t, L)$ as the function returning the privilege associated with the identifier t in the label L (either R or S). The flow $A \rightarrow B$ is allowed to occur if $\forall t \in S_A, h(t, R_B) = +$ holds true. We define $S'_A = \{t | t \in S_A \wedge h(t, S_A) = +\}$, as the set of tags that should propagate. After the flow, S_B is modified to become $S'_B = S_B \cup S'_A$.

We define the function $ALLOW(A, B)$ which, given two entities A and B returns true if the flow is allowed and false otherwise. We also define the function $PROPAGATE(A, B)$ which propagates the send label from A to B according to the definition we have just given.

Jajodia et al. [19] specify that information flow occurs only if an object changes its state, i.e. changes the value of one or more of its attributes. However, this assumes that methods cannot be altered at run time [18], which is not the case in Ruby. Therefore we need to consider more possible flows.

Flow of information occurs on method call. A method call is the interaction of several entities: the caller C, the callee O, the method parameters $p_1, ..., p_n$ and the returned value r. We distinguish two phases: the calling of the method and the returning phase.

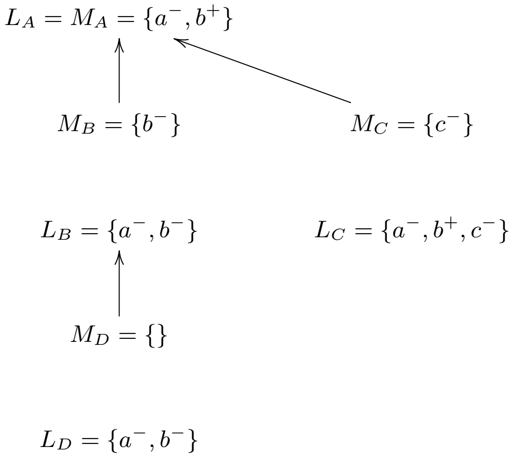

Figure 6. Illustration of label inheritance

During the first phase the flow of information is as follows: $C \rightarrow O$, $p_1 \rightarrow O$, ... $p_n \rightarrow O$. In the second phase the flows first $O \rightarrow r$ and then $r \rightarrow C$. It is important to note that at the end of the second phase we may have $S_r \neq S_O$. This is due to the fact that class/object attributes may have different labels than the class/object they belong to and that there may be label operations within the execution of the method (a method performing an operation on the returned value of another method call, for example). Having different attribute labels may be useful when doing event processing such as in DEFCon [31].

3.3 Methods, instances, class labels

As mentioned earlier, our model has the notion of method label, object label and class label. Object labels are associated with a particular instance of a class, while class labels are associated with all instances of the class or inherited class. Finally, method labels are associated with a particular method of an object or class.

In OO languages classes inherit from their parents. To maintain this logic, the labels defined in a parent class are inherited by its children. Similarly, an object inherits the label of its class and a method inherits the label of its object or class (in the case that this is a static method). It is important to note here that we only support multilevel hierarchical inheritance, but the model could quite easily be extended to support multiple inheritance if implemented in a language supporting this feature.

We now define how labels are inherited. We consider the inheritance from class A to class B. Note that the process would have to be repeated as often as necessary and also that the process is similar when inheriting from class to object or from class or object to method. The inheritance process is identical for *Send* and *Receive* labels.

We note L_A the apparent label for class A (taking into account inheritance) and M_A the label defined at the level of class A. For a class B inheriting from A, $L_B = \{t | t \in L_A \wedge t \notin M_B\} \cup M_B$. Fig. 6 illustrates this principle. For simplicity in the rest of this paper, label will always refer to the apparent label of an object, class or method.

Table 1 illustrates how such a feature can be used to express security concerns throughout an application (we take well known Ruby classes as an example). We first declare that we do not want sensitive information to be written to a file. We also define a tag named $internal$ to protect data that we do not want to leave our

class	receive	send
IO	$\{internal^-\}$	$\{\}$
File	$\{sensitive^-\}$	$\{\}$
NurseReport	$\{\}$	$\{medical^+\}$
Patient	$\{medical^+, default^-\}$	$\{\}$
PublicData	$\{medical^-\}$	$\{\}$

Table 1. Expressing application level security concerns

application (for example a private key used for encryption). We therefore forbid such information to go through any I/O.

We define a class NurseReport which inherits from File to allow the nurse to perform some operation on the report she writes about a patient. We want all data associated with NurseReports to be considered medical. We therefore associate the $\{medical\}$ tag with the *Send* label of NurseReports. An instance of a NurseReport would have the following labels
$R : \{sensitive^-, internal^-\}, S : \{medical^+\}$;
that is, it does not accept sensitive or internal information and contains medical information which it can send to allowed recipients.

We define two other classes inheriting from File that we call Patient and PublicData. Patient labels are as follows:
$R : \{medical^+, internal^-, sensitive^-, default^-\}, S : \{\}$.
As we want our patient well informed, he is only able to read information issued by medical sources (in our case coming from a nurse). He cannot read unlabelled data. PublicData labels are as follows:
$R : \{medical^-, internal^-, sensitive^-\}, S : \{\}$.
This class includes data made public for research. Obviously we do not want confidential medical data to be available to the general public so it is not allowed to flow into PublicData.

However, we want to provide the option for patients to release anonymised data for research purposes. Therefore, we define in the class Patient a method generate_anonymised_record and associate with this method the label $R : \{\}, S : \{medical^-\}$. The medical tag of the data input to the method does not propagate so the data returned by this method would not include the medical tag in its label. It could therefore be used with the PublicData class. Algorithm 1 illustrates how such a method would be used. Section 3.4 contains a general discussion of declassification of data.

Algorithm 1 Example of method label usage

```
p = new Patient
d = new PublicData
d.add(p.generate_anonymised_record)        ▷ succeeds
d.add(p.get_record)                        ▷ fails
```

In order to express real security concerns, we should define a label per patient in order to isolate their respective data. We give an example of this, for records of customers' orders, in section 4.

3.4 Ensuring secrecy and integrity

Information flow control generally enforces two properties throughout the execution of a program. In this section we first describe how we can guarantee the integrity of an entity, then how we can guarantee secrecy of information.

3.4.1 Integrity

Guaranteeing integrity of an entity means accepting data only from trusted sources. The first step to achieve integrity is to set the *Receive* label to $R = \{default^-\}$, that is, no unlabelled data can be read. So far, with this *Receive* label, our entity is unable to receive any information.

Now, we need to set a list of trusted sources. This is done by associating a tag with identifier $source$ with the trusted information and setting the *Receive* label as follows $R = \{source^+, default^-\}$. Here we state that this entity will only accept information associated with the tag with identifier $source$.

Setting $R = \{source\_1^+, source\_2^+, default^-\}$ means that we accept information labelled with one of $source\_1^+$ or $source\_2^+$ or both. Here we are effectively building a white list.

We may also want to prevent onward, indirect propagation of information from a trusted source, i.e. trustworthiness need not be transitive. To achieve this we set the *Send* label of the source to $S = \{source^-\}$. As defined in section 3.2, the tag $source^-$ does not propagate to the *Send* label of the receiver of the information. So an entity that built a white list including the tag $source^+$ would be able to read information directly from the source entity, but would not be able to read it through an intermediate entity. This is important in order to avoid privilege creep.

3.4.2 Secrecy

Secrecy means preventing secret data from being transmitted to an untrustworthy entity. In our context this would generally mean leaving an application or well-known channel. For example, medical data should only be stored in an appropriate database and never be logged or transmitted to a third party server through the network.

In this context the first thing to do is to associate the secret data or the source of the secret data (such as a database) with a tag that will propagate through all the application. That is, we set its *Send* label to $S = \{secret^+\}$. At this point our IFC library will track the data through our application.

The final step to ensure secrecy is to set the receive label of any entity representing a connection outside our application to refuse information with a tag containing this label. This is done simply by setting the entity's *Receive* label to $R = \{secret^-\}$. Here we are effectively building a black list of information which cannot be transmitted to this entity.

3.4.3 Declassification

We have defined how to ensure the secrecy and integrity of information through the manipulation of its associated labels and tags. As mentioned in section 2.2 it is also necessary to be able to declassify information. Declassifying is equivalent to removing a tag from the information in order to allow it to flow to an entity where this would otherwise not be allowed.

Suppose the classified information is stored in data with associated label $S : \{secret^+\}$, $R : \{\}$. To declassify the information we pass the data through a method with the following label $S : \{secret^-\}$, $R : \{\}$. This would mean that the returned value would not carry the $secret^+$ tag and could be used freely. An example of declassification was given above in section 3.3, where a method was defined to input a medical record and output a corresponding declassified, anonymised medical record. Another example is given in section 4.

4. FlowR implementation

We saw in section 3.2, that flows are enforced in two phases: on method call and on method return. This corresponds exactly to the AOP standard *around advice* [23] (discussed in section 2.1). We describe the process in algorithm 2. O is the callee, C is the caller, M the method called, A_s is the set of attributes and $join\_point$ is the *join point* to be executed. We now describe the step described in algorithm 2; *1)* we verify that information is allowed to flow from the caller to the method and we also verify that the information contained in the parameters is allowed to flow in the method; *2)* we propagate the labels from the caller and the parameters to the

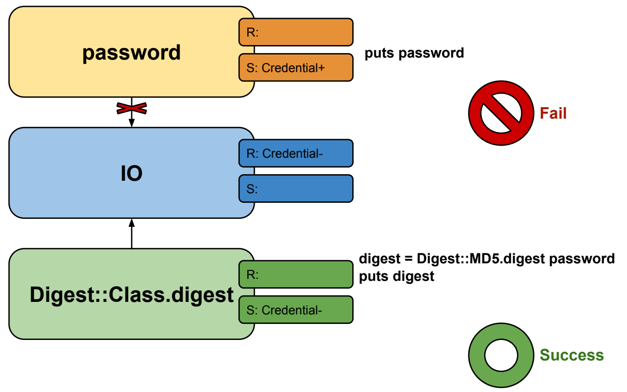

Figure 8. Example 1: Protecting passwords

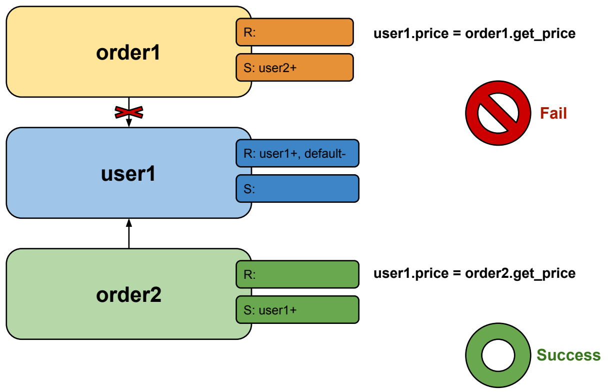

Figure 10. Example 2: Isolating a user's data

callee; *3)* we execute the *join point*; *4)* we propagate the method label to the returned value; *5)* we verify that the information contained in the returned value is allowed to flow to the caller; *6)* we propagate the returned value's labels to the caller. If the flows are found not to be allowed the program is aborted.

We used the AOP library Aquarium [53] to implement this in Ruby. We place an *advice* around any public method of a tracked object (an object is considered to be tracked when there are tags associated with this object). Regardless of the actual object implementation we are therefore able to protect information flow.

Algorithm 2 IFC Around Advice

```
function AROUND($O, C, As, join\_point$)
    if $ALLOW(C, M)$ then                              ▷ step 1 start
        for all $A$ in $As$ do
            if $\neg ALLOW(A, M)$ then
                FAIL
            end if
        end for                                        ▷ step 1 end
        $PROPAGATE(C, O)$                              ▷ step 2 start
        for all $A$ in $As$ do
            $PROPAGATE(A, O)$
        end for                                        ▷ step 2 end
        $RV = join\_point.execute$                     ▷ step 3
        $PROPAGATE(M, RV)$                             ▷ step 4
        if $\neg ALLOW(RV, C)$ then                    ▷ step 5
            Fail
        end if
        $PROPAGATE(RV, C)$                             ▷ step 6
        return $RV$
    else
        FAIL
    end if
end function
```

In order to implement the concepts described in our model (section 3) we provided the API described in table 2. We have instructions to start and stop the tracking of basic variables. Indeed, in some cases, it may be required to activate tracking only on some portion of the code. For example, the loading of a large configuration file could be done before the tracking is activated in order to improve performance. Similarly, $execute\_procedure\_untracked$ allows a single procedure to be executed with tracking deactivated (an illustration of why this may be useful can be found in section 6). Although untracked procedures are executed in Ruby safe mode, the programmer is relied upon to understand the IFC implication of executing a portion of code untracked. The manipulation of this API is illustrated in Fig. 9.

In addition to this API we also support object methods manipulating their labels directly as this may be useful in some circumstances. In Fig. 7 we illustrate how such direct manipulation can be used to prevent information leak on standard output. In this example, we declare that credentials are not allowed to be displayed on the standard output and try to print a password that we previously associated with the credential tag, which causes the program to fail.

A developer should be able to develop an application without initially being concerned about IFC, and with the ability to use a legacy library that was built without IFC in mind. Once the application is developed, the original developer, or another expert, can add IFC rules to ensure that the application behaves correctly with respect to information flow.

In this simple example we consider how to protect the user password from being disclosed unintentionally within our application by printing it out "in clear" in the log, displaying it on a page or saving it "in clear" in a database.

Fig. 9 illustrates a simple way to achieve this without any modification to the code of the original application. We first add a method which is executed before the routing of any request received from a client. In this method we associate with the parameter *password* sent by the client, the send tag *credential*, thus ensuring that any data derived from this information will be tracked in our application.

The second step is to activate tracking for our application which is done by calling *FlowR.start_variable_tracking*. Then we need to specify that we do not want *credential* information to leave our application. This is done simply by invoking the method *FlowR.protect_class IO, nil, credential: false*. Here, we say that any information associated with the send tag *credential* is forbidden to flow towards any *I/O* (files, logs, etc.).

We were able to express policy to protect the user password in six lines of code, with minimal knowledge of the application implementation and without modifying the functional implementation. In addition, we also successfully separated security concerns from the implementation itself.

We have confined our password information to our application. However, in order to provide a useful application we should

FlowR API call	Description
start_variable_tracking	start basic variable tracking.
stop_variable_tracking	stop basic variable tracking.
protect_class / protect_classes	protect all public method of a class(es).
protect_object / protect_objects	protect all public method of an instanc(es).
protect_methods_in_class	protect a defined set of methods in a class.
protect_methods_in_object	protect a defined set of methods in a single instance.
execute_procedure_untracked	allow a procedure to execute without variable tracking for performance reasons detailed in section 6.

Object methods	Description
add_receive_tag / add_receive_tags	add a single or a set of tags to the receive label associated with an object instance or class depending on the context of the call.
add_send_tag / add_send_tags	add a single or a set of tags to the send label associated with an object instance or class depending on the context of the call.
declassify	remove specified tag from the send label.
get_send_label / get_receive_label	get the receive or send label associated with the object/class

Table 2. FlowR API

be able to save the password into the database during the registration and verify the password is correct during authentication. The proper thing to do to store a password is to hash it with the salt. Therefore, we determine that once hashed, the data associated with the send tag *credential* loses its secrecy and becomes safe. We can express this with the following method invocation *FlowR.protect_methods_in_class ([:digest], Digest::Class, credential: false, nil)*. This states that the invocation of the method *Digest::Class.digest* declassifies with respect to the *credential* tag. We illustrate these points in Fig. 8.

We now look at another example. In this case a user class is trying to access an order made on a website and stored in the database. In addition to the usual information associated with the order, we maintain in our database the label associated with each entry. When writing to or reading from the database, we ensure that the label associated with instances of orders are propagated to the database by modifying the *ActiveRecord::Base* implementation. An idea of how this is implemented is illustrated in Fig. 11. Again, here we do not need to modify any implementation code, it would work for any children of *ActiveRecord::Base* and this can easily be added after application development.

In the simple example illustrated in Fig. 10, the instances of the order are associated with a *Receive* label containing the tag representing the user to whom the order belongs. Furthermore, the user1 instance of the user class can only read information associated with its own tag $user1$. Therefore, if the user tries to read information belonging to another user the program will simply fail. During the development and testing phases this allows the programmer to detect bugs in the application, and during the release phase to prevent the user accessing data they do not own.

As attributes are also objects it is also possible to assign labels to each attribute. This would represent the different security and confidentiality requirements of the different fields of this structured document. For example, medical records might be shared between medical professionals and social services. Some sensitive information such as HIV status may be restricted to medical professionals

```
FlowR.start_variable_tracking
FlowR.protect_object $stdout, nil, {credential: false}
 puts 'nothing happens here' # no problem here
 s = 'I can say that!'
 s.add_send_tag :label_s
 puts s # no problem here
 password = '123456789'
 password.add_send_tag :credential
 puts password # here the program fails
```

Figure 7. Example: Applying flow constraints on standard output

```
before do
  params[:password].add_send_tag :credential unless params[:password].nil?
  params[:verify_password].add_send_tag :credential unless params[:verify_password].nil?
end

FlowR.start_variable_tracking
FlowR.protect_class IO, nil, {credential: false}
FlowR.protect_methods_in_class ([:digest], Digest::Class, {credential: false}, nil)
```

Figure 9. Example: Preventing password leak with FlowR

```
module ActiveIFC
        def before_save
                receive_label = self.get_receive_label
                send_label = self.get_send_label
                # save labels to database
        end

        def after_initialize
                # read label from database
                FlowR.protect_object self, send_label, receive_label
        end

        def after_create
                receive_label = self.get_receive_label
                send_label = self.get_send_label
                # save labels to database
        end
end

class ActiveRecord::Base
        include ActiveIFC
end
```

Figure 11. Example: Integrating IFC in ActiveRecord

only, while more general information may be accessible to social services, for example to detect signs of child abuse. Another use of attribute labelling could be to build an event processing system as described in the DEFcon work [31].

5. Use case: Building a medical web portal

In 2012 we developed a web portal, in collaboration with the Eastern Cancer Registry and Information Centre (ECRIC) – now part of Public Health England – to grant access by brain cancer patients to their records [34]. We used IFC to track the flow of data end-to-end through the system. For this purpose, we used RubyTrack, developed by the SafeWeb project [17]. In the work presented here, we have replaced RubyTrack in the web portal by FlowR.

All cancer patients within an administrative region have their data stored in a data centre managed by the cancer registries. Patients within the Eastern Region who have a brain tumour can opt to have their data made accessible to them on an external website managed by the BrainTrust charity.

The data of those patients are encrypted with a unique key per patient. The keys are stored in a dedicated key server, while the individual patients' anonymised medical data, in transit to them, are stored in a separate server. Any patient-provided data is also held separately, thus maintaining a clear separation between patient data associated with the web portal application and the local image of the data held by the cancer registries. Furthermore, patients are invited periodically to respond to a quality of life survey, in order to track the evolution of their condition over time. Those data are regularly retrieved and added to the Cancer Registry's database to improve statistical data about the patient.

We ensure through the use of IFC, that even in the case of unexpected program behaviour, the integrity of patient data is assured, and patients can access only their own data. That is, we ensure isolation of data per purpose and per user. A single request of the data store can manipulate only data for one patient; moreover, the medical data is anonymous and the associated personal data is held separately in isolation.

The requirement for isolating data per patient is an obvious necessity as we want to ensure that patients can only access or amend their own data. The separation and isolation of medical and personal data for a single patient is there to decrease the risk of re-identification. Indeed, very little information is required to uniquely identify an individual [5]. Through encrypting data with a unique key per patient and per usage, and through ensuring isolation of information per patient and per usage, we can reduce the risk that an attacker who gains access to our data store is able to gain medical information on an identified patient.

Fig. 12 illustrates the architecture of our data store. At reception of data we ensure that the user is authenticated. From this authentication, we are able to build what we call the user context. We identify the purpose of the request and associate the appropriate tags (medical, private, etc.) to the parameters transmitted in the HTTP request. In addition, our database models and our controllers have their own set of IFC constraints.

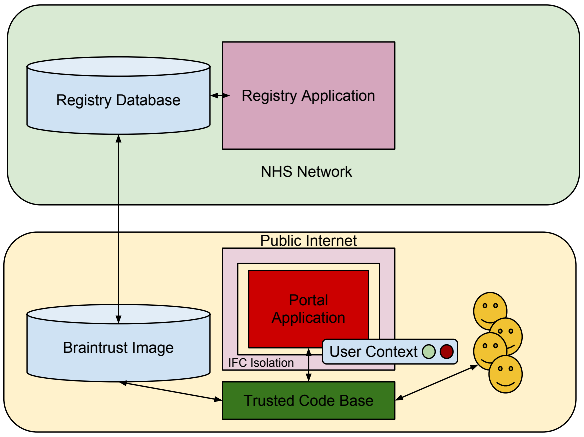

Figure 12. Data store architecture

	RubyTrack	FlowR
Label	a single label	integrity and secrecy
Tag	simple string	symbol + capability
Enforcement	manual by developer at strategic points	at public method call on tracked objects
Engineering	requires overwriting of classes that need to be tracked	minimum

Table 3. Feature comparison of FlowR and RubyTrack

We create an isolation bubble by limiting application access to *IO* classes according to the user context labels and controller labels (in a similar fashion as shown in section 4). In order to propagate labels into and out of the database we store the labels along with the record, i.e. in a row in the database. (We do not support an individual label per column (record field), only per database entry.) We intercept database read and write method calls using the *ActiveRecord* library feature and add the necessary IFC labelling.

Supporting IFC was again done separately from building the actual application, allowing a clear distinction and separation between functional and security concerns.

6. Evaluation

Our tests measure the performance of our solution, FlowR, compared with an equivalent solution, that extends native Ruby with RubyTrack, developed for the SafeWeb project [17]. It is important to note the feature differences that explain the performance difference of FlowR when compared with RubyTrack, as illustrated in table 3.

Our first series of tests concern computing intensive tasks. We demonstrate that FlowR does not perform significantly worse than its equivalent using RubyTrack. In addition, no performance optimisation has been attempted for FlowR, which is beyond the scope of this paper.

Our second series of tests is made on a web application, built to provide patient medical records and similar to the one we built in a previous project [34]. We demonstrate that the performance loss compared with native Ruby is of the same order as the earlier implementation, and acceptable from an end user point of view.

All tests have been performed on an i7 2.2GHz 6GB RAM Fedora 17 GNU/Linux Machine.

6.1 Compute-intensive tasks

We designed two simple tests. The first consists of counting the number of words in text stored in a file on disk ("Les Contemplations" by Victor Hugo). The second test consists of calculating the first n prime numbers. The execution time of the native Ruby code is our time unit. We compare RubyTrack, FlowR and FlowR using untracked procedure calls (section 4).

The results, shown in table 4, show the same order of magnitude for RubyTrack and FlowR. We did not attempt to optimise performance, and the Aquarium library is known to suffer from performance issues [56]. This is because, at present, Aquarium applies advices at runtime whereas AspectJ and AspectC++ apply them at compile time. Furthermore, it is commonly accepted that performing IFC is generally not a good idea while performing computing intensive tasks. Using untracked procedure calls provides much better performance. This figure includes the switching of tracking on, off and on again which induces some overhead. However, this overhead becomes negligible as the execution time becomes large. Therefore, untracked procedure calls can provide performance identical to native Ruby in the case of long computing intensive tasks.

We also measured the execution time for some key primitives which were: starting tracking 325 ms; stopping tracking 108 ms; protecting an additional class 180 ms; adding protection to a single method 5 ms. As mentioned above, adding AOP *advices* at runtime, as in Ruby/Aquarium, incurs performance overhead, and care should be taken in deciding when this is necessary. IFC advices should be added during initialization as much as possible.

On the other hand, the cost of adding a label to an existing object is insignificant (it is simply adding an entry to a hash table). Therefore, adding or removing labels during the lifetime of an application does not amount to a significant performance loss.

6.2 Brain portal: a web portal application

In order to evaluate our library under realistic conditions we used the data store described in section 5. In order to evaluate the performance of our implementation we queried our data store 1000 times, asking for 50 different, randomly chosen data items. We compare the averaged values obtained with native Ruby, RubyTrack and FlowR, as shown in table 5.

We used the "thin" Ruby web server as it provides quite good performance. We first display an unlabelled static page to measure the influence of tracking without flow enforcement. RubyTrack and FlowR add an overhead of 7% and 12% respectively compared to native Ruby. The performance penalties for retrieving a medical record from our database are of the same order (10% and 15% respectively).

We add the IFC advice at initialization; the web server executes the initialization script only once. This removes the very significant overhead generated when creating the advices. Furthermore, as discussed previously, our tracking algorithm is slightly more complicated than RubyTrack and flows are controlled for every protected object (including basic variables), while RubyTrack only enforces flow at strategic points. This explains our performance decrease compared to RubyTrack.

Obtaining more precise results on exactly where compute-time was being spent in our library proved to be impractical, or at least to require too much engineering in the time available. As discussed above in section 6.1, switching on and off advices takes time, i.e. activating IFC slows down the library usually used to trace Ruby application performance. Secondly, activating undiscriminated *I/O*

test	native	RubyTrack	FlowR	untracked
word count	1	6.3	9.8	7.7
prime	1	27	70	1.8

Table 4. Performance comparison of compute-intensive tasks

test	native	RubyTrack	FlowR
hello world	4.1 ms	4.4 ms (+7%)	4.6 ms(+12%)
medical record	62 ms	68 ms (+10%)	71 ms (+15%)

Table 5. Performance comparison for a web portal

protection made the required tracing impossible. This is because the tracer manipulates protected data, which IFC currently prevents it from obtaining via output from the application. We are already working on integrating an IFC-aware, message-passing middleware capable of controlling which connections out of an application are allowed. This will be used to implement debugging and performance monitoring, as well as inter-component communication. At this stage, we can only observe and measure performance "from the outside".

7. Related Work

7.1 Aspect Oriented Programming

Critics of aspect oriented programming [44] argue that AOP reduces the understandability of a program. In our case we are quite confident that expressing constraints in a well-defined and compact fashion in a single file is a better approach than merging them with functional code. Indeed, in standard IFC library, one would need to explore the code in order to understand where IFC constraints are applied.

Others argue [14, 21, 47] that AOP leads to software that is harder than usual to evolve. In our opinion and from experience, the solution we propose is easier to maintain. Indeed, changes to an underlying library implementation do not affect the policy rules defined. The only changes that the security expert needs to keep track of are changes of interface in the case of constraints expressed on methods.

Ramachandran et al. [36] proposed to implement access control using AspectJ [23] around object method calls. In their work each thread is associated with a certain level of clearance, and each object as well. If the current thread level of clearance matches the object on which the program is trying to perform a method call, then the program executes, otherwise it fails. Although our algorithms bear some similarity, [36] does not address information flow control. AspectJ has also been used to implement some of the IFC features of DEFCon [31].

Masuhara et al. [30] discuss the difficulty of implementing crosscutting security features through AOP based on data origin. They propose a new point-cut that they named *dflow*. This point cut allows some procedures to execute on data which flow from object a to b. Although we considered implementing the execution of some programmer-determined procedure on some data with a specific label when it reached a certain object b (effectively providing the same end result), we decided that it was going beyond IFC and therefore beyond the scope of our library.

AOP has been used to implement security features: access control [36, 46], error detection and handling [28], automatic login [50], hardening the security of existing libraries [32] or preventing buffer overflow [42] are some examples where AOP has been used successfully. AOP proves itself a useful and powerful tool as it allows the expression of security concerns that should apply to the whole application while completely decoupling their specification from the application functionality.

7.2 Information Flow Control

Dynamic Library IFC generally presents poor support for implicit flow, as discussed in section 2.3. There are obvious exceptions such as implementation for a purely functional language [38]. In other types of language, solutions include going through static techniques [33] or code rewriting to transform implicit flows into explicit ones [48]. However, AOP techniques present potential alternatives to address implicit flow. Indeed the new "join-point on loop" proposed by Harbulot et al. [16] or "join-point on region" by Akai et al. [1] may be interesting directions to investigate for dynamic implicit flow tracking. We have not yet addressed implicit flow in our work.

Information Flow Control was first enforced statically [9], then Myers introduced some dynamic elements [33] to provide more flexibility at runtime and a decentralised model. More recent work such as SafeWeb [17] or GIFT [25] provides dynamic taint tracking.

We have investigated the provision of IFC at the language level, with its associated tradeoffs. IFC can be provided at other system levels, with different tradeoffs. For example Suh et al. [45] propose a specific hardware architecture to support information flow. Asbestos [13] is an operating system design to provide IFC mechanisms. Finally, there are solutions running on top of a Linux OS [24] or providing a platform for distributed systems [7, 58]. These different levels of implementation, which we discuss in detail in [3], do not necessarily address the same issues. Indeed, while OS level implementation of IFC allows better and easier protection of I/O than the language level, it would require more engineering to create isolation within the application. IFC concerns could have an influence on the application architecture that might be considered too strong.

8. Conclusions and future work

Regardless of the precise implementation properties, we believe that the primitives we propose here are a natural way to express flow constraints within an application in an OO Language. We also believe that the AOP approach discussed in this paper is a good solution to providing IFC when control over the system running an application is not available. That is, our IFC runs above unmodified platforms as well as potentially extending unmodified applications.

We assume a benevolent developer, which is standard for all library-level IFC implementations; and we do not support implicit flow tracking, again, the case for most library-level IFC. We will continue to evaluate the tradeoffs involved in taking the AOP approach compared with using more disruptive and less maintainable mechanisms that might provide higher security and performance. The Brain-Portal implementations using RubyTrack and FlowR have provided a first case study.

Our current implementation does not support multi-threading but this is not inherent to our proposed model. Rather, it is constrained by the AOP library implementation we used and the general poor support of real multithreading in the standard Ruby implementation.

Another issue that may arise when using AOP to enforce IFC is when several AOP advice are implemented over the same object; for example enforcing IFC, logging and authentication. In such a scenario it may be necessary to determine whether composition issues arise, as discussed in [12, 35].

It is important to note that our library does not require the rewriting of any code and therefore *does not modify program behaviour*, except when IFC constraint violation forces the process to abort. So when performance is a critical issue, the library can be used during development, to track unexpected data flows, and ignored in deployment. Again, a tradeoff is involved between performance and security.

In this paper we presented an IFC library implementation using AOP, with primitives to provide IFC concepts, and mechanisms to enforce IFC. We separated application functionality from security concerns. Programmers need not be aware of IFC during application development, and a security specialist can add IFC as a separate phase. This is good engineering practice and achieves better maintainability. However, we described our model informally and a more formal model would be required before substantial future work was carried out.

We believe that using AOP to provide IFC has many advantages which we intend to evaluate further in future work, especially in the context of cloud deployment.

Acknowledgments

This work was supported by the UK Engineering and Physical Sciences Research Council under grant EP/K011510, CloudSafetyNet. We thank Dr Olivier Hermant from MINES ParisTech for his invaluable feedback.

References

[1] S. Akai, S. Chiba, and M. Nishizawa. Region pointcut for AspectJ. In *Proceedings, 8th workshop on Aspects, components, and patterns for infrastructure software*, ACP4IS '09, pages 43–48. ACM, 2009.

[2] T. Austin and C. Flanagan. Efficient purely-dynamic information flow analysis. In *Programming Languages and Analysis for Security (PLAS)*, Dublin, Ireland, 2009. ACM.

[3] J. Bacon, D. Eyers, T. F. Pasquier, J. Singh, I. Papagiannis, and P. Pietzuch. Information Flow Control for secure cloud computing. *submitted to: IEEE Transactions on Network and Service Management,Special Issue on Management of Cloud Services*, 2014.

[4] D. Bell. The Bell-LaPadula model. *Journal of computer security*, 4 (2):3, 1996.

[5] K. Benitez and B. Malin. Evaluating re-identification risks with respect to the HIPAA privacy rule. *Journal of the American Medical Informatics Association*, 17(2):169–177, 2010.

[6] K. J. Biba. Integrity considerations for secure computer systems. Technical report, DTIC Document, 1977.

[7] W. Cheng, D. R. Ports, D. Schultz, V. Popic, A. Blankstein, J. Cowling, D. Curtis, L. Shrira, and B. Liskov. Abstractions for usable information flow control in Aeolus. In *Proceedings, USENIX Annual Technical Conference (Usenix ATC '12)*, 2012.

[8] D. Denning. Secure Information Flow in Computer Systems, 1975. Dissertations Publishing 1975.

[9] D. E. Denning. A lattice model of secure information flow. *Commun. ACM*, 19(5):236–243, May 1976.

[10] D. E. Denning and P. J. Denning. Certification of programs for secure information flow. *Commun. ACM*, 20(7):504–513, July 1977.

[11] B. DeWin, B. DeVanhaute, and B. DeDecker. Security Through Aspect-Oriented Programming. In *Advances in Network and Distributed Systems Security*, volume 78 of *IFIP*, pages 125–138. Springer US, 2002.

[12] C. Disenfeld and S. Katz. A closer look at aspect interference and cooperation. In *Proceedings of the 11th annual international conference on Aspect-oriented Software Development*, AOSD '12, pages 107–118, New York, NY, USA, 2012. ACM.

[13] P. Efstathopoulos, M. Krohn, S. VanDeBogart, C. Frey, D. Ziegler, E. Kohler, D. Mazières, F. Kaashoek, and R. Morris. Labels and event processes in the Asbestos operating system. In *Proceedings, 20th ACM Symposium on Operating Systems Principles*, SOSP '05, pages 17–30. ACM, 2005.

[14] E. Figueiredo, N. Cacho, C. Sant'Anna, M. Monteiro, U. Kulesza, A. Garcia, S. Soares, F. Ferrari, S. Khan, F. Filho, and F. Dantas. Evolving software product lines with Aspects. In *Software Engineering, 2008. ICSE '08. ACM/IEEE 30th International Conference on*, pages 261–270, 2008.

[15] M. Gyung, S. McCamant, P. Poosankam, and D. Song. DTA++: Dynamic Taint Analysis with Targeted Control-Flow Propagation. In *Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, 2011. Internet Society.

[16] B. Harbulot and J. R. Gurd. A join point for loops in AspectJ. In *Proceedings, 5th international conference on Aspect-oriented software development*, AOSD '06, pages 63–74. ACM, 2006.

[17] P. Hosek, M. Migliavacca, I. Papagiannis, D. Eyers, D. Evans, B. Shand, J. Bacon, and P. Pietzuch. SafeWeb: A middleware for securing Ruby-based web applications. In *Middleware*, pages 491–512, 2011.

[18] S. Jajodia and B. Kogan. Integrating an object-oriented data model with multilevel security. In *Proceedings, IEEE Symposium on Security and Privacy*, pages 76–85, 1990.

[19] S. Jajodia, B. Kogan, and R. Sandhu. A multilevel-secure object-oriented data model. *Abrams et al.[AJP95]*, 1995.

[20] V. Kashyap, B. Wiedermann, and B. Hardekopf. Timing- and termination-sensitive secure information flow: exploring a new aproach. In *Symposium on Security and Privacy*, Berkeley, CA, 2011. IEEE.

[21] C. Kastner, S. Apel, and D. Batory. A case study implementing features using AspectJ. In *Software Product Line Conference, 2007. SPLC 2007. 11th International*, pages 223–232, 2007.

[22] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. *Aspect-Oriented Programming*. Springer, 1997.

[23] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In J. L. Knudsen, editor, *ECOOP 2001 Object-Oriented Programming*, volume 2072 of *Lecture Notes in Computer Science*, pages 327–354. Springer, 2001.

[24] M. Krohn, A. Yip, M. Brodsky, N. Cliffer, M. F. Kaashoek, E. Kohler, and R. Morris. Information Flow Control for standard OS abstractions. *SIGOPS Oper. Syst. Rev.*, 41(6):321–334, Oct 2007.

[25] L. C. Lam and T. cker Chiueh. A general dynamic information flow tracking framework for security applications. In *Computer Security Applications Conference, 2006. ACSAC '06. 22nd Annual*, pages 463–472, 2006.

[26] B. W. Lampson. A note on the confinement problem. *Commun. ACM*, 16(10):613–615, Oct. 1973.

[27] S. B. Lipner. A comment on the confinement problem. *SIGOPS Oper. Syst. Rev.*, 9(5):192–196, Nov. 1975.

[28] M. Lippert and C. Lopes. A study on exception detection and handling using Aspect-Oriented Programming. In *Software Engineering, 2000. Proceedings of the 2000 International Conference on*, pages 418–427, 2000.

[29] F. Marchand de Kerchove, J. Noyé, and M. Südholt. Aspectizing javascript security. In *Proceedings of the 3rd Workshop on Modularity in Systems Software*, MISS '13, pages 7–12, New York, NY, USA, 2013. ACM.

[30] H. Masuhara and K. Kawauchi. Dataflow pointcut in Aspect-Oriented Programming. In *Proceedings, First Asian Symposium on Programming Languages and Systems*, APLAS, pages 105–121. Springer, 2003.

[31] M. Migliavacca, I. Papagiannis, D. M. Eyers, B. Shand, J. Bacon, and P. Pietzuch. DEFCon: High-performance event processing with information security. *USENIX Annual Technical Conference*, June 2010.

[32] A. Mourad, M.-A. Laverdière, and M. Debbabi. An aspect-oriented approach for the systematic security hardening of code. *Computers & Security*, 27(3):101–114, 2008.

[33] A. C. Myers. JFlow: practical mostly-static information flow control. In *Proceedings, 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '99, pages 228–241. ACM, 1999.

[34] T. Pasquier, B. Shand, and J. Bacon. Information Flow Control for a Medical Web Portal. In *e-Society 2013*. IADIS, March 2013.

[35] R. Pawlak, L. Duchien, and L. Seinturier. Compar: Ensuring safe around advice composition. In *Formal Methods for Open Object-Based Distributed Systems*, pages 163–178. Springer, 2005.

[36] R. Ramachandran, D. J. Pearce, and I. Welch. AspectJ for multilevel security. *ACP4IS '06*, 20:13–17, March 2006.

[37] D. E. Robling Denning. *Cryptography and data security*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1982.

[38] A. Russo, K. Claessen, and J. Hughes. A library for lightweight information-flow security in Haskell. *SIGPLAN Notices*, 44(2):13–24, Sept. 2008.

[39] A. Sabelfeld and A. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications (JSAC)*, 21 (1):5–19, 2003.

[40] A. Sabelfeld and A. Russo. From dynamic to static and back: Riding the roller coaster of information-flow control research. In *Andrei Ershov International Conference on Perspectives of System Informatics*, Akademgorodok, Novosibirsk, Russia, 2009.

[41] J. Saltzer and M. D. Schroeder. The protection of information in computer systems. *Proc. IEEE*, 63(9):1278–1308, 1975.

[42] V. Shah and F. Hill. An aspect-oriented security framework. In *DARPA Information Survivability Conference and Exposition, 2003. Proceedings*, volume 2, pages 143–145 vol.2, 2003.

[43] O. Spinczyk, A. Gal, and W. Schröder-Preikschat. AspectC++: an aspect-oriented extension to the C++ programming language. In *Proceedings, 40th International Conference on Technology of Object-Oriented Languages and Systems (TOOLS Pacific 2002)*, volume 10 of *CRPIT '02*, pages 53–60. Australian Computer Society, Inc., 2002.

[44] F. Steimann. The paradoxical success of aspect-oriented programming. *SIGPLAN Not.*, 41(10):481–497, Oct. 2006.

[45] G. E. Suh, J. W. Lee, D. Zhang, and S. Devadas. Secure program execution via dynamic information flow tracking. *SIGOPS Oper. Syst. Rev.*, 38(5):85–96, Oct. 2004.

[46] R. Toledo and E. Tanter. Secure and modular access control with Aspects. In *Proceedings of the 12th annual international conference on Aspect-oriented software development*, AOSD '13, pages 157–170, New York, NY, USA, 2013. ACM.

[47] T. Tourwé, J. Brichau, and K. Gybels. On the existence of the AOSD-evolution paradox. *SPLAT: Software engineering Properties of Languages for Aspect Technologies*, 2003.

[48] N. Vachharajani, M. Bridges, J. Chang, R. Rangan, G. Ottoni, J. Blome, G. Reis, M. Vachharajani, and D. August. Rifle: An architectural framework for user-centric information-flow security. In *Microarchitecture, 2004. MICRO-37 2004. 37th International Symposium on*, pages 243–254, 2004.

[49] J. Viega and J. Vuas. Can Aspect-Oriented Programming lead to more reliable software? *Software, IEEE*, 17(6):19–21, 2000.

[50] J. Viega, J. Bloch, and P. Chandra. Applying Aspect-Oriented Programming to security. *Cutter IT Journal*, 14(2):31–39, 2001.

[51] P. Vogt, F. Nentwich, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna. Cross site scripting prevention with dynamic data tainting and static analysis. In *Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, 2007. Internet Society.

[52] D. Volpano and G. Smith. Eliminating covert flows with minimum typings. In *Computer Security Foundations Workshop, 1997. Proceedings., 10th*, pages 156 –168, jun 1997.

[53] D. Wampler. Aquarium: AOP in Ruby. In *Proceedings, Aspect Oriented Software Development (AOSD)*, volume 4, 2008.

[54] H. Washizaki, A. Kubo, T. Mizumachi, K. Eguchi, Y. Fukazawa, N. Yoshioka, H. Kanuka, T. Kodaka, N. Sugimoto, Y. Nagai, and R. Yamamoto. AOJS: An Aspect-Oriented Javascript programming framework for web development. In *Proceedings, 8th workshop on Aspects, Components, and Patterns for Infrastructure Software*, ACP4IS, pages 31–36. ACM, 2009.

[55] A. Yip, X. Wang, N. Zeldovich, and M. F. Kaashoek. Improving application security with data flow assertions. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, SOSP '09, pages 291–304, New York, NY, USA, 2009. ACM.

[56] A. Zambrano, A. Alvarez, J. Fabry, and S. Gordillo. Aspect Coordination for Web Applications in Java/AspectJ and Ruby/Aquarium. *Proceedings, 28th International Conference of Chilean Computer Society*, Nov. 2009.

[57] S. Zdancewic. Challenges for information-flow security. In *Proceedings of the 1st International Workshop on Programming Language Interference and Dependence (PLID04)*, 2004.

[58] N. Zeldovich, S. Boyd-Wickizer, and D. Mazieres. Securing distributed systems with information flow control. In *Networked Systems Design and Implementation (NSDI)*, pages 293–308. Usenix, April 2008.