SNITCH: Dynamic Dependent Information Flow Analysis for Independent Java Bytecode

Eduardo Geraldo João Costa Seco
NOVA LINCS - Faculdade de Ciências e Tecnologia da Universidade Nova de Lisboa
Portugal

Software testing is the most commonly used technique in the industry to certify the correctness of software systems. This includes security properties like access control and data confidentiality. However, information flow control and the detection of information leaks using tests is a demanding task without the use of specialized monitoring and assessment tools.

In this paper, we tackle the challenge of dynamically tracking information flow in third-party Java-based applications using dependent information flow control. Dependent security labels increase the expressiveness of traditional information flow control techniques by allowing to parametrize labels with context-related information and allowing for the specification of more detailed and fine-grained policies. Instead of the fixed security lattice used in traditional approaches that defines a fixed set of security compartments, dependent security labels allow for a dynamic lattice that can be extended at runtime, allowing for new security compartments to be defined using context values.

We present a specification and instrumentation approach for rewriting JVM compiled code with in-lined reference monitors. To illustrate the proposed approach we use an example and a working prototype, SNITCH. SNITCH operates over the static single assignment language Shimple, an intermediate representation for Java bytecode used in the SOOT framework.

1 Introduction

Data confidentiality is central in current software engineering practices. In the past years, there have been recurrent news about information leaks surfacing as a result of subtle programming errors. For instance, GitHub[1] and Twitter[2], both large-scale systems with impact on a large number of users, discovered and reported that their users' passwords were stored in cleartext to internal system logs, from where an ill-intended employee could have access to them and enter the users' accounts. Certifying the functional correctness of software systems by testing is commonly accepted as a satisfactory approximation for compliance with functional specifications and requirement fulfilment in the software industry. However, testing aspects such as data confidentiality is a difficult task when using traditional approaches. Properties like access control and information flow control require setting up complex testing scenarios where the symptoms of an error are hardly detectable. Typically, information leaks are only perceived at a global scale by detailed observation of side-effects.

Information flow analysis [5, 10, 20, 25] is a language-based approach for information leak detection on software systems. Information flow analysis is present in the literature, in the form of static and dynamic analysis, each with their advantages and disadvantages. Static analysis usually requires a considerable effort in code annotation or the complete refactoring of the target system. Besides, the

[1] http://bit.ly/2XNfEEU
[2] http://bit.ly/2XuMH16

D. Ancona and G. Pace (Eds.): Verification
of Objects at RunTime EXecution 2018 (VORTEX 2018)
EPTCS 302, 2019, pp. 16–31, doi:10.4204/EPTCS.302.2

over-approximation of results and the existence of false positives also poses an obstacle to the usability of languages and tools [15, 19, 22] that employ this kind of analysis. The alternative presented by dynamic analysis techniques produces less false positives but has some problems of its own. It needs exhaustive testing to achieve maximum coverage, and it is subject to label creeping [1, 20], i.e., the monotonic increase of the security label of values rendering them unusable.

Information flow control mechanisms depend on a security lattice [10] whose security labels are, usually, fixed. Consequently, existing are usually too restrictive when defining information flow policies, only allowing to define coarse-grained security policies which are inadequate in many cases. For instance, one often groups all users of a system under the same security label "User", which does not prevent user *A* from accessing information of a user *B*. We follow a more expressive approach that introduces dependent types for information flow [14, 16] and access-control [7] allowing for the definition of data-dependent policies. Value-dependent security labels improve the expressiveness of traditional information flow techniques. By allowing the parametrization of security labels with context-related information, it is possible not only to define more detailed and fine-grained policies, but also to create new security compartments at runtime. Java Information Flow (JIF) [18, 19] supports dynamic labels which differ from dependent security labels. Dynamic labels follow a decentralized security model [18] based on the notion of data ownership and authorizations. According to this model, each data item has an owner, and the owner allows, or not, its data to be read or written by some entity. Dependent security labels follow a traditional security model with a security lattice that hierarchically organizes security labels and where a datum has a security label and can only be accessed by entities with sufficient privileges. For instance, using dependent security labels, we can define policies restricting access to an employee's personal telephone number to the employee itself and its department manager.

In this paper, we present a strategy to specify and rewrite the intermediate Java code of applications to embed reference monitors capable of enforcing information flow policies using dependent security labels. Our low-level code rewriting approach for the Java virtual machine language is inspired by tools like SASI [**?**] and TaintDroid [12] that automatically monitor the confidentiality of information of compiled Java programs. To check data confidentiality in an application, TaintDroid [12] instruments the underlying runtime system (Android Java virtual machine) while our approach instruments the application itself. We require the specification of some selected classes that make up the entry points of a system, for instance, service controllers [9] or DAO classes[3] [2]. Then, we use this information to introduce inlined reference monitors and instrument the application code to taint computed values with dependent security labels. Our approach follows the style and semantics of the seminal work by Austin and Flanagan on dynamic information flow analysis [5] and is inspired by works such as the one by Lourenço and Caires [16, 17] on dependent information flow analysis and the work of Chandra and Franz [8] about hybrid information flow analysis for Java bytecode.

The rewriting process, presented in section 3 operates over an intermediate representation in the static single assignment form [4, 26] (SSA). SSA is a way to arrange operations such that each variable is defined only once and allows to simplify and improve some optimizations such as constant propagation, value numbering, common sub-expression elimination and partial redundancy elimination among others. We present an example that illustrates the rewriting process over an intermediate representation in the SSA form. Our approach is backed by a prototype tool, SNITCH, to instrument intermediate Java code. SNITCH was evaluated on small-scale web applications using Java servlets and it uses the SOOT framework[4] [23] for code rewriting, a framework for optimizing and manipulating Java bytecode and

[3] Database access objects. Dtatypes that match database table schemas.

[4] https://github.com/Sable/soot

offers multiple intermediate representations. One of such representations is Shimple, an intermediate representation in the SSA form over which our prototype operates.

Our contributions can be summarized as follows:

- a rewriting system for instrumenting static single assignment instructions with in-line reference monitors for information flow control with dependent information flow labels;
- a specification schema to define dependent information flow policies;
- a way to define dependent security labels in Java; and
- a tool capable of instrumenting third-party compiled code with an in-line reference monitor.

We leave for future work the introduction of abstract interpretation to optimize the computation of security labels and mechanisms like the one presented by Austin and Flanagan [6] to reduce label creeping and increase the number of accepted programs. Abstract interpretation would allow us not only to reduce the number of security label related computations executed at runtime, but also to achieve a gradual approach.

We start this paper by briefly presenting some concepts on dependent information flow labels in Section 2. Section 3 presents our approach and describes an example of a web application. As we present our approach we also describe the steps required to instrument the example application to test it for information leaks. In Section 4 we illustrate the code rewriting process. In Sections 5 and 6 we provide validate our approach and discuss the related work. Finally, in Section 7, we conclude with some remarks on how to pursue this line of work.

2 Dependent Security Information Flow

Language-based security [21], and in particular information flow control [10], specify and provide a platform to enforce security policies from the perspective of data creation, manipulation and data flow operations. Information flow control allows the definition of hierarchic security compartments and the tracking of all uses of data, ensuring that higher security data does not flow (leak) to unrelated or lower security compartments. Traditionally, Security labels are organized in lattices [11] and are associated with value types at compile-time [18, 24] or used to taint values at runtime [5, 8].

Information flow control allows for the detection of both explicit and implicit illegal information flows. Explicit flows result from data transfer operations such as assignments, while implicit information flows arise from the control flow of a program. High security label computations can have side effects on values of lower security labels allowing those with access to lower labelled variables to infer the values of those computations. The side effects of high security computations on lower labelled values go against the non-interference property, a property at the core of information flow control and that denotes the absence of information leaks. According to non-interference, changes to high security label values must not reflect themselves on lower security labelled values, i.e., changes in the secret input of a program must not interfere with the program's public output [25].

Traditional security lattices enforce a significant degree of label squashing due to the lack of precision of the security labels used. For instance, it is usually the case that a single security label is used to represent all the users of a system, not allowing to define fine-grained, per user, information flow restrictions. The introduction of dependent security policies increases the preciseness of security specifications and introduces a higher degree of flexibility and usability. Dependent security policies are present in approaches like value-dependent information flow types [14, 16, 17], and dynamic labels [19]. The former

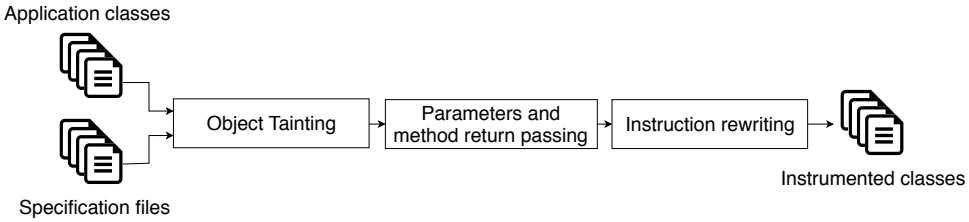

Figure 1: Code Instrumentation Phases

was first introduced in the context of access control policies in [7] and then extended to the domain of static checking of information flow in [14, 16, 17]. With dependent security labels, we can express, for instance, that a given function yields values of a parametric security label `user(u)` where `u` is a runtime value, allowing for row-level compartmentalization of security data visibility (c.f., [16]). The predefined lattice is automatically extended to capture dependent security labels like `user(u)`, `user(`$\top$`)`, and `user(`$\bot$`)`. The generic security label `user(`$\top$`)` is the label taht allows access to all users' data. The security label `user(`$\bot$`)` is the label all users can read. We have the relations `user(`$\bot$`)` $\sqsubset$ `user(u)` $\sqsubset$ `user(`$\top$`)`, for any user `u`, and `user(u)` # `user(v)` for all users `u` and `v` such that `u` $\neq$ `v`. The first relation means that, for any user `u`, data can flow from the user's security compartment `user(u)`, to label `user(`$\top$`)`, and from `user(`$\bot$`)` to label `user(u)`.

3 Technical Approach

Our technical approach follows the phases depicted in Figure 1. Given an application, a dependent security lattice, and a security specification for key classes of the application, our approach instruments the application with an in-lined reference monitor capable of enforcing information flow policies. We next illustrate our approach with the help of an example.

Let us consider a small web application to implement a directory for a given company integrated in their website. The information stored by such a system for each employee includes its identifier, name, address, salary, and its password. We define two kinds of employees in this example: supervisor and associate. The latter category includes extra information, namely its supervisor and information about its last evaluation. Other users include unregistered users accessing the company's website.

In this example, we consider the following access constraints to the stored information:

- only registered users can see the address of other users (employees);
- an associate employee can only access its salary;
- a supervisor user can access all associate users' salary information;
- the information regarding who supervises who can only be accessed by supervisor users;
- the information about the evaluation of an associate user can only be accessed by supervisor users;
- passwords are always secret, and no one but its owner should access it.

Our example implements both retrieval and insertion operations. The operations are the following: it is possible to list the employees in the system; to retrieve the information about a specific employee; to compute the average salary; to add new employees to the system. Only a registered user, an employee, can execute the operation that retrieves the information about any other employee. This operation exhibits different behaviours depending on who is executing it and what information is retrieved.

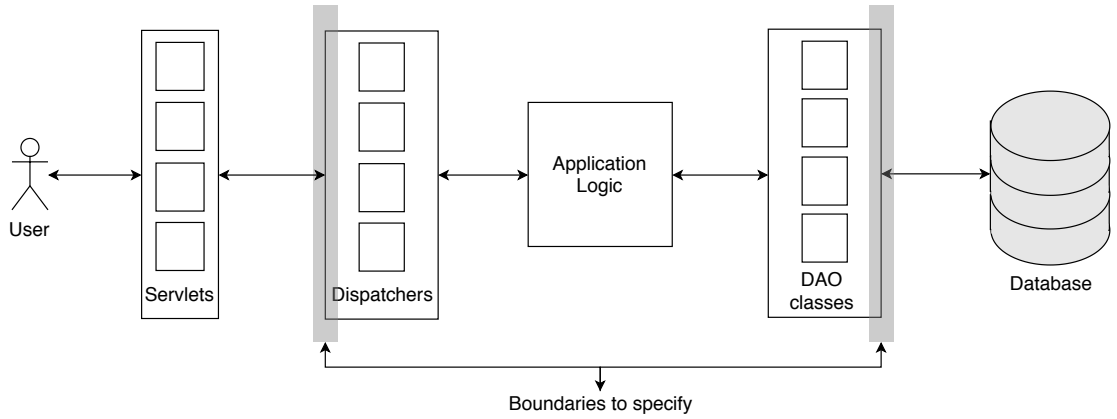

Figure 2: Typical architecture of a web application

In Figure 2 we illustrate the generalised architecture of a web application. For the sake of space and simplicity, in our evaluation example we merged the application logic and request dispatchers in a single layer. In Figure 4 we present the DAO classes to represent the company's employees. Class `Associate` represents associate employees and Class `Supervisor` represents supervisor employees. Both classes, `Associate` and `Supervisor`, extend the abstract class `Employee` that contains the information common to both kinds of employees. Class `Supervisor` is empty since it does not add any new fields to class `Employee`. We omit all class methods as there are only getters and setters.

Dependent Security Labels The instrumentation of a system starts by defining the security specification. First, it is necessary to define and implement custom security labels to extend the default security lattice provided by SNITCH which only includes two built-in labels. The label `Public`, the lowest security label of all, and the label `Secret`, the highest security label of all.

Custom security labels are implemented by extending the abstract class `SecurityLevel`, provided in a companion library, and by defining a required comparator method. By implementing the missing comparator method, we model the security lattice used by the reference monitor during the system's execution. Each label requires also a constructor which takes the same parameters as the security label. For each label parameter, the label's contructor requires one extra parameter of type `int`. For instance, a custom label `User`, parametrized by a `String` and a `long` has a constructor with the signature `User(String,int,long,int)`. The extra parameter tells the monitor if the value passed to the security label's constructor is $\bot$, $\top$, or if the corresponding parameter is to be considered as is.

In order to instrument the example, we define two custom security labels, `SupervisorSL`, and `AssociateSL`, which define security compartments for supervisors and associates, respectively. Both labels have a single parameter, an employee id, and their comparator methods define the security lattice depicted in Figure 3.

Specifications files Besides custom security labels, it is necessary to define a security layer, using a set of specification files and the security lattice (custom security labels) used to in-line the reference monitor. In Figure 2 we show the layer that needs specification, the classes that make the boundaries of the system. This layer includes all the classes that communicate with the exterior context of the system, such as service controllers and DAO classes in a typical architecture.

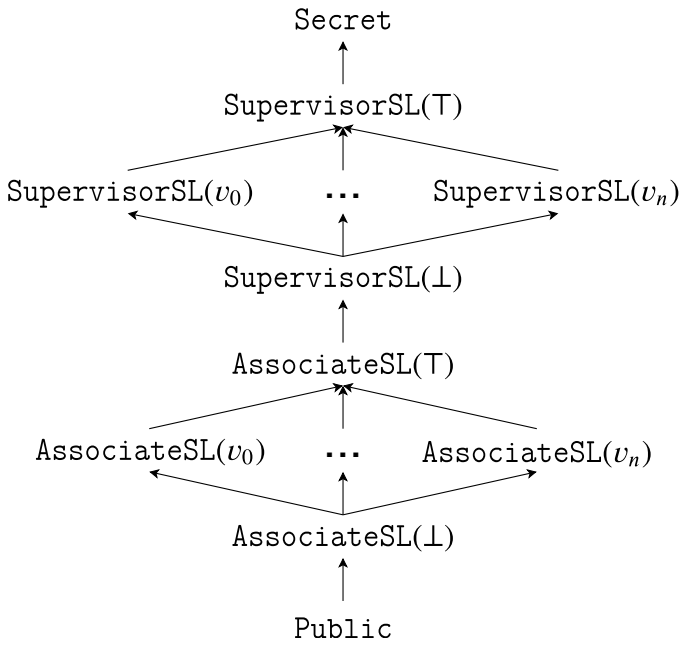

Figure 3: Security lattice used to instrument example.

```
abstract class Employee {
    long id;
    String name;
    String address;
    double salary;
    String pwd;
    ...
}
```

```
class Associate extends Employee {
    Supervisor supervisor;
    double evaluation;
    ...
}

class Supervisor extends Employee {}
```

Figure 4: Classes `Employee`, `Associate`, and `Supervisor`

Specifications files include annotations to define the security labels of class fields and method signatures. The semantics of field annotations is the following. If a security label is explicitly assigned to a class field, it will be fixed (as maximum) throughout the entire execution. Any attempt to store a value in such a field will result in one of two outcomes: if the incoming value's security label is lower than the expected security label, then the incoming value's security label is upgraded; if the incoming value's security label is higher than the expected field's security label, the monitor signals an information leak. When a field is not annotated with a security label, it changes according to the stored values. We, however, do not let assignment operations to lower the security labels of variables or fields (c.f., [5]) to avoid implicit information leaks.

When defining specifications for methods, it is possible to annotate both parameters and return values. Method annotations differ from field annotations as they include one of two modifiers, `?` or `!`. If an annotation uses the modifier `?`, the reference monitor compares the security label of the annotated value with the security label used in the annotation and, if higher, the monitor signals an information leak. If an annotation uses the modifier `!`, the monitor will associate the annotation's security label with the annotated value. The modifier `!` allows one to associate a security label with input from outside the

```
abstrcat class Employee {
    long:Public id;
    String:Public name;
    String:AssociateSL(_) address;
    String:Secret pwd;
}
```

(a)

```
class Supervisor extends Employee {
    double:SupervisorSL(id) salary;
}
```

(b)

```
class Associate extends Employee {
    double:AssociateSL(id) salary;
    Supervisor:SupervisorSL(_) supervisor;
    double:SupervisorSL(supervisor) evaluation;
}
```

(c)

Figure 5: Security specifications for classes `Employee`, `Supervisor` and `Associate`

```
public String associateDispatch(long requesterId, long queriedId) {

   Employee queried = EmployeeRepository.getInstance().getEmployeeById(queriedId);

   String response = "{"
      + "\"id\": " + "\"" + queried.getId() + "\""
      + "\"name\": \"" + queried.getName() + "\","
      + "\"address\": \"" + queried.getAddress() + "\",";

   if (requesterId == queriedId)
      response += "\"salary\": \"" + queried.getSalary() + "\",";

   return response + "}";
}
```

Figure 6: Source code for the employee information dispatcher

system and to declassify information. Since it allows for information declassification [1], it is necessary to use this modifier carefully as it may easily lead to incorrect specifications which result in undetected information leaks. If a parameter or return value does not have a security annotation, then the monitor will propagate its security label without performing any extra operation.

In the current example, we define specification files for all dispatcher classes and for classes that represent stored information about employees. The specifications for classes `Employee`, `Associate`, and `Supervisor` are depicted in Figure 5a, Figure 5b, and Figure 5c respectively. Notice that all fields in the classes are annotated with a security label from the lattice in Figure 3. Security label parameters are instantiated with fields to denote a concrete dependency, or `(_)` to represent $\bot$. The values used to instantiate security label parameters must belong to the same object and produce a security dependency between fields. For instance, in class `Supervisor` we use `SupervisorSL(id)` as a security label dependent on the value of field `id`, declared in the superclass `Employee`. The security label `AssociateSL(_)` establishes a security compartment accessible to all `Associate` employees.

The effort required to write security specifications depends on several factors: the knowledge about

```
class EmployeeInfoDispatcher {
   String:?AssociateSL(requesterId) associateDispatch(long requesterId, long queriedId);
}
```

Figure 7: Security specifications for the employee information dispatcher

the system to instrument, the constraints of the system, and the complexity of the security specification and lattice. After defining the security specifications and labels, it is possible to instrument the application.

Value tainting In the first instrumentation step, we inject shadow fields in every application class. One of them holds the security label of the class instance while the remaining ones mirror existing class fields of a primitive or library (non-instrumented) type.

Methods and Parameter passing In order to propagate the security labels of primitive or library-type arguments and return values, we add shadow fields to each class. These shadow fields help preparing method calls by allowing the caller to store and the callee to retrieve the arguments' security labels. When the called method terminates its execution, the callee stores the security labels of the arguments and return value for the caller to retrieve.

Instruction rewriting The final step of the instrumentation process consists in the instrumentation of method bodies. The body of a method consists of a graph of basic blocks, where a basic block is a sequence of instructions starting with a label and terminating in a return, a branching, or a jump instruction. The instrumentation process compositionally rewrites instructions in the SSA form [4, 26], according to the rules defined in Figure 8. Every rule for instructions that give place to information flows take into account the security label associated with the computation itself, i.e., the security label of the program counter ($\mathtt{pc}\%\ell$) [5].

The set of instructions considered is the following: load a value to a local variable $\underline{v = s}$ or $\underline{v = o.f}$; method call $\underline{v = o.g(v_1 ... v_n)}$, where $\underline{o}$ represents the target object; object instantiation $\underline{v = \mathsf{new}\ C}$; binary operations $\underline{v = op(e,e)}$; phi expressions $\underline{v = \phi(v,v)}$; conditional jumps $\underline{\mathsf{if}(e)\ \mathsf{goto}\ l}$; unconditional jumps $\underline{\mathsf{goto}\ l}$; and the return instruction $\underline{\mathsf{return}\ e}$. A phi function is a pseudo-function used in merge points to yield one of its arguments according to the control-flow path executed.
Notation: we use $\underline{C}$ to denote class names, $\underline{v}$ and $\underline{o}$ to denote local variables or registers, $\underline{k}$ to denote value literals, $\underline{e}$ ranges over local variables and constants; $\underline{f}$ to denote object fields, and $\underline{g}$ to denote method names. A variable $\underline{v_s}$ stores the security label of variable $\underline{v}$, a field $\underline{f_s}$ stores the security label of field $\underline{f_s}$, a field $\underline{g_{pi}}$ stores the security label of parameter $\underline{i}$ of the method g and a field $\underline{g_{ret}}$ stores the security label of the return value of method g.

The rules for loading operations, depicted in Figure 8, (CONST, LOCAL, and FIELD) work by combining the pc, the security label of the value, and the variable's security label. The dynamic modification of a field, rule FIELDW, potentially increases the field's security label with combination of pc and the value's security label. Rule FIELDC applies to fixed label fields, where writes are always lower or equal to the current label.

To deal with a method call, we have two rules. Rule CALL handles the call to an instrumented method. It starts by copying the arguments' security labels to the target method with the help of auxiliary fields and then calls the method. Once the method completes its execution, we retrieve the result and argument's

$$
\begin{array}{rll}
[\![v := k]\!]_\ell \triangleq & v := k;\ v_s := v_s \sqcup \mathtt{pc}\%\ell & \text{(CONST)} \\
[\![v := v']\!]_\ell \triangleq & v := v';\ v_s := v_s \sqcup v'_s \sqcup \mathtt{pc}\%\ell & \text{(LOCAL)} \\
[\![v := f]\!]_\ell \triangleq & v := f;\ v_s := v_s \sqcup f_s \sqcup \mathtt{pc}\%\ell & \text{(FIELD)} \\
[\![f := v]\!]_\ell \triangleq & \mathsf{assert}(v_s \sqcup \mathtt{pc}\%\ell \sqsubseteq f_s);\ f := v; & (f \text{ has spec.}) \quad \text{(FIELDC)} \\
[\![f := v]\!]_\ell \triangleq & f := v;\ f_s := f_s \sqcup v_s \sqcup \mathtt{pc}\%\ell & (f \text{ has no spec.}) \quad \text{(FIELDW)} \\
[\![v := \mathsf{new}\, C]\!]_\ell \triangleq & v := \mathsf{new}\, C;\ v_s := v_s \sqcup \mathtt{pc}\%\ell & \text{(NEW)} \\
[\![v := o.g(v_1,\ldots,v_n)]\!]_\ell \triangleq & o.f_{p1} := v_{1_s} \sqcup \mathtt{pc}\%\ell;\ldots; o.f_{pn} := v_{n_s} \sqcup \mathtt{pc}\%\ell; & \\
& v := o.g(v_1,\ldots,v_n); & \\
& v_{1_s} := o.f_{p1};\ldots; v_{n_s} := o.f_{pn}; & \\
& v_s := o.f_{return} & (o \text{ has spec.}) \quad \text{(CALL)} \\
[\![v := o.g(v_1,\ldots,v_n)]\!]_\ell \triangleq & v := o.g(v_1,\ldots,v_n); & \\
& v_s := v_s \sqcup o_s \sqcup v_{1_s} \sqcup \ldots \sqcup v_{n_s} \sqcup \mathtt{pc}\%\ell & (o \text{ has no spec.}) \quad \text{(CALLX)} \\
[\![v := op(e_0,e_1)]\!]_\ell \triangleq & v := op(e_0,e_1); & \\
& v_s := v_{0_s} \sqcup v_{1_s} \sqcup \mathtt{pc}\%\ell & \text{(BIN OP)} \\
[\![\mathsf{if}(v)\ \mathsf{goto}\ k]\!]_\ell \triangleq & \mathtt{pc}\%\ell_{out} := \mathtt{pc}\%\ell_{in} \sqcup v_s & \\
& \mathsf{if}(v)\ \mathsf{goto}\, k & \text{(BRANCH)} \\
[\![\mathsf{goto}\, k]\!]_\ell \triangleq & \mathsf{goto}\, k & \text{(GOTO)} \\
[\![v := \phi(v_0,v_1)]\!]_\ell \triangleq & v =: \phi(v_0,v_1); v_s := v_s \sqcup \phi(v_{0_s}, v_{1_s}) & \text{(PHI)} \\
[\![\mathsf{return}\ e]\!]_\ell \triangleq & \mathsf{this}.f_{\mathsf{return}} := e_s \sqcup \mathtt{pc}\%\ell; & \\
& o.f_{p1} := v_{1_s};\ldots; o.f_{pn} := v_{n_s}; & \\
& \mathsf{return}\ e; & \text{(RETURN)}
\end{array}
$$

Figure 8: Instrumentation rules

security label. It is necessary to collect the argument security labels after the call since they might have changed during the method's execution. Rule CALLX accounts for the use of non-instrumented methods, where the resulting security label is the combination of all operands' security labels plus the program counter and, in the case of instance methods, the callee's security label.

Notice that in the case of rule BRANCH, the value for the context's security label (pc) increases according to the security label of the branch condition. Once the execution leaves the scope started by the branch condition, it is necessary to reinstate pc's old security label. To restore the pc, we follow the rules depicted in Figure 9. The rule depicted in Figure 9a is applied in the case where there are multiple predecessors $(\ell_1,...,\ell_n)$ of the basic block ℓ but e_1 does not post-dominate a common predecessor of $\ell_1,...,\ell_n$, i.e., multiple control flows converge but the scope does not change. According to this rule, the security label of the context at beginning of the basic ℓ results from the ϕ function of the predecessors' context security labels. The second rule, the rule depicted in Figure 9b, applies when there are multiple predecessors $(\ell_1,...,\ell_n)$ of ℓ and e_1 is the first instruction to post-dominate a common predecessor, d, of $\ell_1,...,\ell_n$. In this case, the context security label at beginning of block ℓ ($\mathtt{pc}\%\ell_{in}$) is equal to the context security label at the beginning of d ($\mathtt{pc}\%d_{in}$), i.e., e_1 is the first instruction to execute outside the scope created in d and reinstates the value of pc before entering the new scope. Unconditional branches do not change any security meta-information. Rule PHI chooses the security label according to the executed predecessor. When a return instruction executes, the pc stack is placed at the same label as it was when

ℓ : $\texttt{pc\%}\ell := \phi(\texttt{pc\%}\ell_1, ..., \texttt{pc\%}\ell_n)$
$[\![e_1]\!]$
...
$[\![e_n]\!]$
$\ell_1, ..., \ell_n$ are predecessor nodes of ℓ.

(a)

ℓ : $\texttt{pc\%}\ell_{in} := \texttt{pc\%}d_{in}$
$[\![e_1]\!]$
...
$[\![e_n]\!]$
d is the post-dominated node where the current scope started.

(b)

$\texttt{pc\%}\ell_{out}$ is equal to $\texttt{pc\%}\ell_{in}$ if not stated otherwise.

Figure 9: Context security label rules for merging control flows

the function was called (because of ad-hoc returns at any point in the method body). Besides restoring the `pc`, it is also necessary to copy the returned value's security label and the argument's security labels to auxiliary fields. It is necessary to update the security labels of the arguments to deal with cases where they are objects of non-instrumented types. The objects' security label might change during the method's execution, in which case it is necessary to propagate any changes to the caller.

Testing phase Once defined the security layer and instrumented the application, we test the example for information leaks. To do so, we need to test all available operations in the instrumented application. If an operation has an information leak, the monitor halts the system's execution indicating an assertion violation. Strong guarantees about data confidentiality depend on the test coverage achieved.

In summary, our approach for the detection of illegal information flows using dependent security labels, is embodied in a tool based on the SOOT framework which instruments a target application with a reference monitor. With this approach, we believe to have improved the process of security certification for third-party systems. Despite the need for some specification effort, typically, there is a set of DAO and controller classes that are known and for which is possible to design a specification.

4 Rewriting Process Example

In this section we illustrate the rewriting process using a small example. Let us consider the Java class depicted in Figure 10b. As stated in section 3, first, we add new fields (to which we will refer as "label fields") to the application classes for storing security labels. We add one label field for the objects' security label (`secLbl$this`), one label field for every field of a non-instrumented type (`secLbl$field0`) and for every method we add label fields for parameters and return values of non-instrumented types (`secLbl$methodA$p0`, `secLbl$methodA$p1`, and `secLbl$methodA$ret`). We show the result the of field injection on class `Example` in Figure 10a.

Once injected all the necessary fields, we can proceed to rewrite the methods' body in the SSA form. To do so, we rewrite every instruction according to the rules defined in Section 3. Figure 11a depicts a possible representation of `methodA` in the static single assignment form and, Figure 11b illustrates the result of `methodA`'s body rewriting.

Lines 6-7 retrieve arguments' security labels (`secLlb$a` and `secLlb$b`) from auxiliary fields injected for the purpose (`secLlb$methodA$p0` and `secLlb$methodA$p1` respectively);

```
class Example {
  SecurityLabel secLbl$this;

  SecurityLabel secLbl$field0;
  long field0;

  Example field1;

  SecurityLabel secLbl$methodA$p0;
  SecurityLabel secLbl$methodA$p1;
  int methodA (int a, int b) {...}
  SecurityLabel secLbl$methodA$ret;

  Example methodB(Example e) {...}

}
```

(a) Class `Example` after field injection.

```
class Example {

  long field0;
  Example field1;

  int methodA (int a, int b) {
    if(a > b)
      return a;
    return b;
  }

  Example methodB(Example e) {...}

}
```

(b) Method `methodA` of Class `Example`.

Figure 10: Class `Example`

Lines 8-10 compute the condition's security label. Then, update `pc`, keeping its old value so that we can restore it when the execution leaves the branch's scope. Finally execute the branching instruction.

Lines 11-16 compute the value to return, security operations accompany every operation executed. Each branch stores the result in a different version of the same variable (`result_1` and `result_2`).

Lines 17-19 terminate the context initiated with the branching instruction (more specifically in line 9). When the execution leaves the scope of the branch instruction, it is necessary to restore `pc` to its previous value. Since there are two paths converging, it is also necessary to decide which version of the variables to consider using ϕ functions.

Lines 19-20 conclude the method's execution. They store the result's label (`secLbl$result_2`) in field `secLbl`*methodA*`ret` and update the arguments' label fields. The return only executes after storing the labels.

5 Experimental Validation

To provide some validation to our approach, we developed a prototype tool, SNITCH, and the instrumented the web application presented to provide some validation to our approach, we developed a prototype tool, SNITCH, and then used it to instrument the web application presented in Section 3.

Just as defined in the approach; SNITCH, based on a set of security specifications, instruments a system with an in-lined reference monitor. As can be seen Figure 12, which depicts SNITCH's architecture, SNITCH consists of two modules; a parser for the security specifications and an instrumentation module for bytecode rewriting. The latter component makes use of the SOOT [23] framework which, as previously stated, is a framework for Java bytecode manipulation and optimization. In an attempt to reduce the reference monitor's impact on the system's execution, SNITCH makes use of the optimization suites SOOT offers for optimizing the instrumented code.

```
1  class Example {
2
3    ...
4
5    int methodA (int a, int b) {
6      if(a > b) goto LABEL0
7      result_0 = b
8      goto LABEL1
9  LABEL0:
10     result_1 = a;
11 LABEL1:
12     result_2 = phi(result_0, result_1)
13     return result_2
14   }
15
16   ...
17
18 }
```

(a)

```
1  class Example {
2
3    ...
4
5    int methodA (int a, int b) {
6      secLbl$a = this.secLbl$methodA$p0
7      secLbl$b = this.secLbl$methodA$p1
8      secLbl$cond = combine(secLbl$a, secLbl$b)
9      secLbl$oldPC = increasePC(cond)
10     if(a > b) goto LABEL0
11     result_0 = b
12     secLbl$result_0 = secLbl$b
13     goto LABEL1
14 LABEL0:
15     result_1 = a;
16     secLbl$result_1 = secLbl$a
17 LABEL1:
18     setPC(secLbl$oldPC)
19     result_2 = phi(result_0, result_1)
20     secLbl$result_2
21        = phi(secLbl$result_0, secLbl$result_1)
22     this.secLbl$methodA$ret = secLbl$result_2
23     this.secLbl$methodA$p0 = secLbl$a
24     this.secLbl$methodA$p1 = secLbl$b
25     return result_2
26   }
27
28   ...
29
30 }
```

(b)

Figure 11: Original (left) and instrumented (right) SSA code for method `methodA` of Class `Example`

To test the approach, we introduced information leaks in the example application. The leaks resulted from implicit and explicit information flows. The leaks caused by explicit flows were bad assignments or attempts to return classified information. The in-lined reference monitor in the application was capable of detecting all the information leaks introduced in the example application.

The example's information retrieval methods' implementation was naive, returning all information available on the employees disregarding any information access restrictions. We reached the final implementation of the application through a trial and error process in which we instrumented, tested, and fixed the application multiple times until no further information leaks were detected.

The instrumentation of the example web application also the collection of for the collection of some broad measurements on the reference monitor's impact on the execution time of an application. Still, more applications need to be instrumented and tested to obtain more accurate values. We defined a set of five operations which we used to measure the total execution time and each operation's average execution time. We measured operations' execution time in both the original and instrumented applications. Figure 13 shows how the reference monitor affects the execution time of each operation. The operation for information retrieval is the one where the impact of the monitor was the greatest. An explanation for this is that this operation extracts the most information per employee; therefore, it executes data

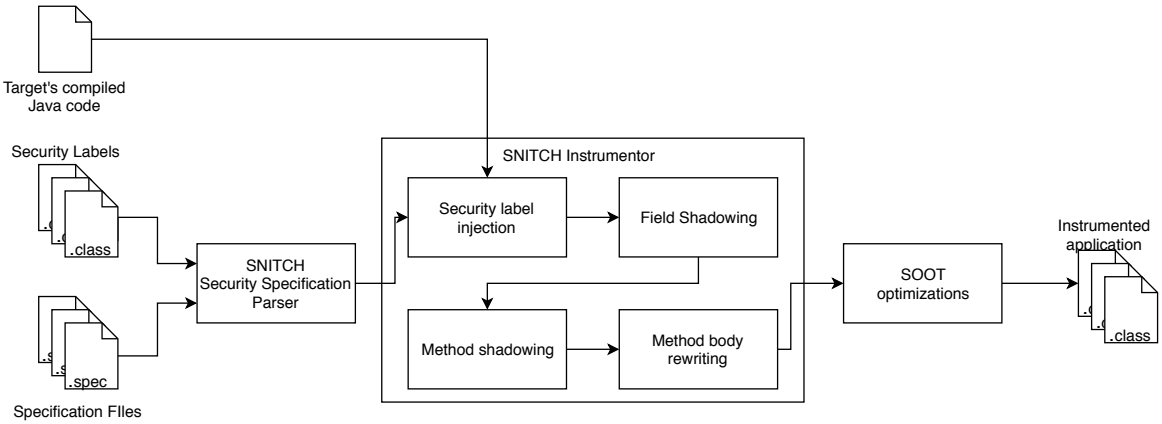

Figure 12: SNITCH internal phases

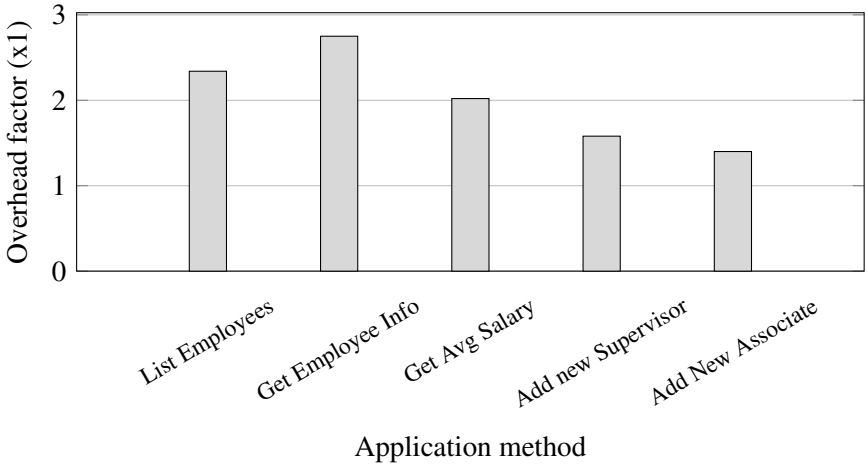

Figure 13: Runtime overhead per application method.

combination operations (e.g., string concatenation) requiring more intervention from the monitor. The overall execution time of the five hundred operations in the instrumented application is 1.79 times the original execution time. However, this estimate does not take into account the overhead of input and output operations which takes milliseconds to complete while CPU operations take microseconds.

From the instrumentation of the example, it was possible to observe the impact of the reference monitor on a system's execution time. Despite that the instrumented application is of a small dimension, we observed a significant overhead (average 1.79x) on the application's CPU time; however, there is still room for improvement; for instance, the prototype does not apply any kind of optimization specific to information flow control, only using the standard optimizations supplied by the SOOT framework.

6 Related Work

There is a considerable amount of work on information flow analysis in the literature, ranging from axiomatic approaches [3], dynamic analyses [5, **?**], programming languages and types [15, 19, 22] to instrumented virtual machines [12]. Java Information Flow [18, 19], contains embedded information flow analysis capabilities, and allows the definition of a form of dynamic matching between labels and principals which can in turn be used to parametrize classes and define richer runtime policies.

TaintDroid [12] is an approach that does not extend or create a programming language but instruments the virtual machine where the intermediate language executes. Sensitive data is tainted at its source (e.g., GPS) and the instrumented virtual machine propagates the taint along a program's execution. When tainted data reaches a sink (e.g., network interface), the information leak is logged. An advantage of the approach taken by TaintDroid over JIF is that it is not necessary to change application code.

Austin and Flanagan [5] present a dynamic approach for information flow analysis that guarantees non-interference in dynamically-typed languages. It presents and compares two approaches. *Universal Labelling*, where all values have an explicit label (security label); and *Sparse Labelling* where all values are tracked but only some are explicitly labelled. Sparse labelling is observably equivalent to universal labelling but with significantly less overhead.

Ferreira [14] introduces the use of refinement types in information flow analysis. It presents an extension of the LiveWeb/λ_{DB} [7] with type-based information flow. Security labels are expressed using first-order logic propositions dependent on runtime values. Value-dependent security labels are further developed by [16], who presents the first non-interference result for dependent information flow types.

7 Concluding Remarks

The purpose of this work is to study the applicability of information flow analysis to the certification of third-party Java-based software systems. To convey a more usable, flexible and expressive framework, we have adopted dependent information flow control as the preferred abstraction.

This paper presents work in the development of a certification tool that attaches in-lined reference monitors to existing compiled code, based on interface specifications in observable points of systems. We foresee some immediate follow-ups on this work, the challenges in dealing with label creeping and the introduction of abstract interpretation to help reduce the runtime overhead beyond the optimizations resulting from the use of SSA intermediate language. Considering the security label combination operation ($\sqcup$) that given two security labels ℓ_A and ℓ_B, yields the lowest security label that is higher or equal to both ℓ_A and ℓ_B and the security lattice shown in Figure 3, computations such as $\mathit{AssociateSL}(\bot) \sqcup \mathit{AssociateSL}(\top)$ can be removed as its result is known beforehand ($\mathit{AssociateSL}(\top)$

). By statically analysing the code, not only trivial computations could be removed, but also, it would be possible, in some instances, to detect illegal information flows statically. The introduction of such mechanisms would allow our approach to evolve from a dynamic information flow control mechanism to a hybrid one. Another possible line of work would be the introduction of our approach in software development frameworks as a development tool. This would allow software developers to test their code as they develop. There also some advantages that our approach can benefit from if integrated with software development tools like the automatic extraction of specifications based on the frameworks annotations. Frameworks like Spring and Jersey annotate classes with information relevant to the specification files; for instance spring uses the annotation `@Entity` to flag DAO classes.

Regarding the monitor's overhead presented on Section 5, we would like to highlight that the measurements made only took into account CPU time. When taking into account I/O operations, we can consider the monitor's overhead as negligible. For instance, the measurements of CPU time were of the order of the microseconds, while I/O operations took milliseconds, three decimal orders of magnitude greater and network operations even worse.

Acknowledgements This work was funded by NOVA LINCS UID/CEC/04516/2013, COST CA15123 - FC&T Project CLAY - PTDC/EEI-CTP/4293/2014

References

[1] Ana Almeida Matos & Gérard Boudol (2009): *On Declassification and the Non-disclosure Policy*. *J. Comput. Secur.* 17(5), pp. 549–597, doi:10.3233/JCS-2009-0355.

[2] Deepak Alur, Dan Malks, John Crupi, Grady Booch & Martin Fowler (2003): *Core J2EE Patterns (Core Design Series): Best Practices and Design Strategies*, 2 edition. Sun Microsystems, Inc., Mountain View, CA, USA.

[3] Gregory R. Andrews & Richard P. Reitman (1980): *An Axiomatic Approach to Information Flow in Programs*. *ACM Trans. Program. Lang. Syst.* 2(1), pp. 56–76, doi:10.1145/357084.357088.

[4] Andrew W. Appel (1998): *SSA is Functional Programming*. *SIGPLAN Not.* 33(4), pp. 17–20, doi:10.1145/278283.278285.

[5] Thomas H. Austin & Cormac Flanagan (2009): *Efficient Purely-dynamic Information Flow Analysis*. In: *Proceedings of the ACM SIGPLAN Fourth Workshop on Programming Languages and Analysis for Security*, PLAS '09, ACM, New York, NY, USA, pp. 113–124, doi:10.1145/1554339.1554353.

[6] Thomas H. Austin & Cormac Flanagan (2010): *Permissive Dynamic Information Flow Analysis*. In: *Proceedings of the 5th ACM SIGPLAN Workshop on Programming Languages and Analysis for Security*, pp. 3:1–3:12, doi:10.1145/1814217.1814220.

[7] Luís Caires, Jorge A. Pérez, João Costa Seco, Hugo Torres Vieira & Lúcio Ferrão (2011): *Type-based Access Control in Data-centric Systems*. In: *Proceedings of the 20th European Conference on Programming Languages and Systems: Part of the Joint European Conferences on Theory and Practice of Software*, ESOP'11/ETAPS'11, Springer-Verlag, Berlin, Heidelberg, pp. 136–155, doi:10.1006/inco.1994.1093.

[8] D. Chandra & M. Franz (2007): *Fine-Grained Information Flow Analysis and Enforcement in a Java Virtual Machine*. In: *Twenty-Third Annual Computer Security Applications Conference (ACSAC 2007)*, pp. 463–475, doi:10.1109/ACSAC.2007.37.

[9] Robert Daigneau (2011): *Service Design Patterns: Fundamental Design Solutions for SOAP/WSDL and RESTful Web Services*, 1 edition. Addison-Wesley Professional.

[10] Dorothy E. Denning (1976): *A Lattice Model of Secure Information Flow*. *Commun. ACM* 19(5), pp. 236–243, doi:10.1145/360051.360056.

[11] Dorothy E. Denning & Peter J. Denning (1977): *Certification of Programs for Secure Information Flow*. *Commun. ACM* 20(7), pp. 504–513, doi:10.1145/359636.359712.

[12] William Enck, Peter Gilbert, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel & Anmol N. Sheth (2014): *TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones*. *Communications of the ACM*, doi:10.1145/2494522.

[13] Úlfar Erlingsson & Fred B. Schneider (2000): *SASI Enforcement of Security Policies: A Retrospective*. In: *Proceedings of the 1999 Workshop on New Security Paradigms*, NSPW '99, ACM, New York, NY, USA, pp. 87–95, doi:10.1145/335169.335201. Available at `http://doi.acm.org/10.1145/335169.335201`.

[14] Paulo Jorge Abreu Duarte Ferreira (2012): *MSc Dissertation. Information flow analysis using data-dependent logical propositions.* Faculdade de Ciências e Tecnologia, Universidade Nova de Lisboa.

[15] Jürgen Graf, Martin Hecker & Martin Mohr (2013): *Using JOANA for Information Flow Control in Java Programs - A Practical Guide*. In: *Proceedings of the 6th Working Conference on Programming Languages (ATPS'13)*.

[16] Luísa Lourenço & Luís Caires (2015): *Dependent Information Flow Types*. *SIGPLAN Not.* 50(1), pp. 317–328, doi:10.1145/2775051.2676994.

[17] Maria Luísa Sobreira Gouveia Lourenço (2016): *A type system for value-dependent information flow analysis*. Ph.D. thesis.

[18] Andrew C. Myers & Barbara Liskov (2003): *Protecting privacy using the decentralized label model*. In: *Foundations of Intrusion Tolerant Systems, OASIS 2003*, pp. 89–116, doi:10.1145/363516.363526.

[19] Andrew C. Myers, Lantian Zheng, Steve Zdancewic, Stephen Chong & Nathaniel Nystrom (2006): *Jif 3.0: Java information flow*. Available at `http://www.cs.cornell.edu/jif`.

[20] Andrei Sabelfeld & Andrew C. Myers (2003): *Language-based information-flow security*. *IEEE Journal on Selected Areas in Communications* 21(1), pp. 5–19, doi:10.1109/JSAC.2002.806121.

[21] Fred Schneider, Greg Morrisett & Robert Harper (2001): *A Language-Based Approach to Security*.

[22] V. Simonet (2003): *The Flow Caml System (version 1.00): Documentation and user's manual*. Available at `http://www.normalesup.org/~simonet/soft/flowcaml/manual/`.

[23] R Vallée-Rai, P Co & E Gagnon (1999): *Soot-a Java bytecode optimization framework*. *CASCON*.

[24] Stephan Arthur Zdancewic (2002): *Programming Languages for Information Security*. Ph.D. thesis, Ithaca, NY, USA. AAI3063751.

[25] Steve Zdancewic (2004): *Challenges for information-flow security*. *Proceedings of the 1st International Workshop on the Programming Language Interference and Dependence (PLID'04)*.

[26] Jianzhou Zhao, Santosh Nagarakatte, Milo M.K. Martin & Steve Zdancewic (2013): *Formal Verification of SSA-based Optimizations for LLVM*. *SIGPLAN Not.* 48(6), pp. 175–186, doi:10.1145/2499370.2462164.