Static Security Certification of Programs via Dynamic Labelling

Sandip Ghosal[1], R. K. Shyamasundar[1] and N. V. Narendra Kumar[2]
[1]*Department of Computer Science and Engineering, Indian Institute of Technology Bombay, Mumbai, 400076, India*
[2]*Institute for Development and Research in Banking Technology, Hyderabad, India*

Keywords: Language-based Security, Information-flow Security, Dynamic Labelling.

Abstract: Programming languages are pivotal for building robust secure systems, and language-based security platforms are very much in demand for building secure systems. In this paper, we explore an approach for static security certification of a class of imperative programs using a hybrid of static and dynamic labelling via information flow control (IFC) models. First, we illustrate an analysis of some benchmark programs using static (or immutable) labelling approaches, and discuss possible labelling of the principals/subjects and objects using a combination of mutable and immutable labelling, and discuss their impact on the precision of the underlying certification. Then, we describe our approach of static certification of programs based on a combination of mutable and immutable (i.e., hybrid) labelling; our labelling generates labels from the given set of initial labels (some of which could be immutable) and the constraints require to be satisfied for a program to be *information-flow secure* as defined by Denning *et. al.*(Denning and Denning, 1977). Our labelling algorithm is shown to be sound with respect to non-interference, and we further establish the termination of the algorithm. Our proposed labelling approach is more security precise than the other labelling approaches in the literature. It may be pointed out that the labels are generated succinctly without unnecessarily blowing up the label space. As the method is not tied to any particular security model, it provides a sound basis for the security certification of programs for information-flow security. We compare the precision realizable by our approach with those in the literature. The comparison of our approach also brings to light an intrinsic property of our labelling algorithm that could be effectively used for non-deterministic or concurrent programs.

1 INTRODUCTION

The seminal work of Denning (Denning, 1976) on security certification of programs built on *information-flow security* led to a firm foundation for language-based security. The extension of such a theory through the proposal of the Decentralized Label Model (DLM) (Myers and Liskov, 2000) provided a momentum for language-based security. Since then there has been an enormous amount of literature on language-based security (Myers, 1999; Myers et al., 2001; Simonet and Rocquencourt, 2003; Stefan et al., 2012b). There have also been many assessments of the status of language-based security (Ryan et al., 2001; Sabelfeld and Myers, 2003; Hicks et al., 2007). With the need for security everywhere including IoT, language-based security is becoming prominent as it deals with security at various levels, and also provides various points of leaks and attacks.

Non-interference was developed after Denning's security certification as a more semantic characterization of security (Goguen and Meseguer, 1982), followed by many extensions. Informally the non-interference property says the impact on the program due to changes in *high* inputs should not be observable by the *low* outputs. (Volpano et al., 1996) have used a purely value-based interpretation of non-interference as a semantic characterization of information-flow, and derived sound typing rules to effectively capture Denning's certification semantics. Volpano *et. al.*'s notion of non-interference, and its extensions have become the de-facto standard for the semantics of information-flow in the literature on language-based security. Usually, the objective of IFC is to enforce non-interference property to ensure end-to-end flow security.

A broad spectrum of information-flow security mechanisms vary from fully dynamic ones e.g., in the form of execution-monitors (Askarov and Sabelfeld, 2009; Austin and Flanagan, 2009) to static ones e.g., in the form of type systems (Volpano et al., 1996). While one would prefer static labels as that leads to certification of programs at compile-time, it has the problem of classifying programs that would not leak

Ghosal, S., Shyamasundar, R. and Kumar, N.
Static Security Certification of Programs via Dynamic Labelling.
DOI: 10.5220/0006868602340245
In *Proceedings of the 15th International Joint Conference on e-Business and Telecommunications (ICETE 2018) - Volume 2: SECRYPT*, pages 234-245
ISBN: 978-989-758-319-3

any information under the information-flow policy at execution time due the underlying inputs that arrive at run-time. Real-world web applications often require to interact with external environment that cannot be predicted during compile-time which motivates researchers to enforce security at run-time. For instance, security settings of files and database records are updated frequently, and these changes might affect the information flow control which cannot be handled by static mechanisms. Dynamic labels are essential to capture the changes in security label and accordingly labels are changed at run-time. However, unlike static or immutable labelling implementation, dynamic mechanism has a cost that user has to pay in the form of large run-time overhead, and also *implicit flows* introduced due to uncovered flow paths not considered at run-time. Hence, an ideal label-checking mechanism should have a *hybrid labelling* that would have a nice trade off for mutable and immutable labels to realize acceptable precision and performance.

In this paper, we first discuss various labelling approaches that use a combination of attributes like mutable, immutable, static, compile-time, run-time etc., for the security certification along with the corresponding realizable precision of security. With such a comparison in hand, we propose a new hybrid (mutable and immutable) labelling approach for certifying programs for information-flow security using the standard certification of constraints as elaborated by Denning. Our dynamic labelling algorithm is established to be sound with respect to non-interference, and we further prove the termination of the algorithm. Our proposed labelling algorithm leads to certification that is more security precise than other labelling approaches in the literature. It may be pointed out that the labels are generated succinctly without unnecessarily blowing up the label space. As the method is not tied to any particular security model, it provides a sound basis for the security certification of programs for information-flow security. We further compare the precision realizable by our approach with those in the literature. The comparison of our approach also brings to light, an intrinsic property of our labelling algorithm that could be effectively used for non-deterministic or concurrent programs.

Structure of the Paper: Section 2 presents different labelling schemes, and assess the limitations of them. Section 3 describes the proposed dynamic labelling algorithm along with illustrative examples, and proofs of characteristic properties as well as soundness with respect to *non-interference*. Section 4 provides a comparison with earlier approaches. Finally, Section 5 summarizes the contributions along with the ongoing work.

2 CERTIFICATION OF PROGRAMS

According to Denning's Information Flow Model (DFM) the necessary and sufficient condition for the flow security of a program P is: if there is an information flow from object x to object y, denoted by $x \rightarrow y$, the flow is secured by P only if $\lambda(x) \leqslant \lambda(y)$. '$\lambda$' is a labelling function, responsible for binding subject/object of the program to a security class (either statically or dynamically depending on the application) from the lattice of security classes as described in Denning's lattice model (Denning, 1976). '$\leqslant$' is a binary relation on security classes that specifies permissible information flows. '$\oplus$' is a binary class-combining operator evaluates least upper bound (LUB) of two security classes in the lattice. The above condition is usually referred to as the *Information-flow policy* (IFP). A program is certified for IFP if there are no violations of the policy during program execution. The *Information Flow Secure* policy forms a basis for certifying programs for security. The crux of certification lies in assuring that all the information flows over legitimate channels or storage channels follow the specified flow-policy. The outcome of approaches could be measured in terms of *precision* defined below.

Let us suppose that F is the set of possible flows in an information flow system, and let A be the subset of F authorized by a given flow policy, and let E be the subset of F "executable" given the flow control mechanisms in operation. The system is said to be **secure** if $E \subseteq A$; that is all executable flows are authorized. A secure system is **precise** if $E = A$.

The method of binding security classes/labels to objects plays an important role in the analysis of programs. Each of the static certification and runtime enforcement algorithms proposed in the literature chooses an object labelling method, some of which may also use the label of the implicit variable, usually referred to as *Program Counter* (PC) which may be reset after every statement or keeps updating monotonically. First, let us consider the following three broad object labelling schemes:

Scheme 1: fixed labels for all the variables,

Scheme 2: labels of all the variables can be modified; for example, based on the information contained in them at any given program point, and

Scheme 3: labels of some variables are fixed while the labels of the other variables could be modified.

In this section, we argue that from the perspective of capturing the notion of security, (i) Scheme 1 is inappropriate as it is too stringent, and results in secure programs being incorrectly rejected as insecure,

(ii) Scheme 2 is also inappropriate as it allows all programs as secure programs, and (iii) Scheme 3 is the ideal candidate, if the variables whose labels can be allowed to be modified are carefully chosen.

Majority of literature on language-based security follows Scheme 1, as it has advantages in certifying tricky programs such as the one given in Table 1.

Table 1: There is implicit flow from x to y while there is no direct flow.

```
void test(int x){
    int y=0;
    int z=0;
    if(x==0)
        z=1;
    if(z==0)
        y=1;
}
```

Table 2: Need to label local variables dynamically.

```
void test(){
    int a;
    a=x;
    y=a;
    a=z;
}
```

As the program in Table 1 executes, the following information flow is observed: if the value of x is 0, the value of z becomes 1, and the second *if* block will not execute, thus the value of y remains 0. On the other hand if the value of x is 1, the second *if* block executes, and y is initialized to 1. In either cases the value of y is same as the value of x although there is no such explicit assignment e.g. $x = y$. If we consider the security labels of x and y are $\underline{x}$ and $\underline{y}$ and $\underline{x} \nleqslant \underline{y}$ then this is an example of insecure program as there is an implicit flow from x to y even though they belong to different security classes where an explicit flow is not allowed. Table 3 analyzes two different labelling approaches i.e. Scheme 1 and Scheme 2 in respect of the program in Table 1.

However, purely static labelling is too restrictive, and rejects secure programs as insecure. This is illustrated by considering the program fragment shown in Table 2, where x, y and z are global variables, and a is a local variable (we do not consider pointer variables).

If the program in Table 2 is analyzed under Scheme 1, it generates the following set of flow-constraints to be satisfied for the program to be secure: $\underline{x} \leqslant \underline{a}$, $\underline{a} \leqslant \underline{y}$, and $\underline{z} \leqslant \underline{a}$. These constraints will be satisfied only if $\underline{z} \leqslant \underline{y}$, which implies that there is an information-flow from z to y. However, from an intuitive perspective, the program never causes an information flow from z to y, and must be considered secure if $\underline{x} \leqslant \underline{y}$. Table 4 analyzes two different labelling approaches i.e. Scheme 1 and Scheme 2 in respect of the program in Table 2.

From the above examples, it follows that the use of purely static labelling is too conservative, and misses several secure programs.

2.1 Refinement Via PC Labels

Information flow is quite tricky to capture, and can happen even if a statement does not get executed. Such flows are called "implicit" (Robling Denning, 1982), and are possible due to conditional and iteration statements. To keep track of such impact, the notion of the program counter (*pc*) label is introduced that denotes the sensitivity of the current context. Traditionally, once the control exits the conditional and/or iteration statements, the pc label is reset to its previous value, thus denoting that the variables in the condition expression no longer impact the current context. Subsequently, a sequential composition $S_1; S_2$ is deemed secure if both S_1 and S_2 are individually secure.

Examples copy3 and copy4 in (Robling Denning, 1982) have highlighted that certain subtle flows cannot be captured unless the pc label is updated monotonically, and tracks the influence of all the information the program has accessed. It was noted that this may lead to a phenomenon called "label creep" (Sabelfeld and Myers, 2003) wherein the pc label rises too high resulting in rejecting any further flows. To avoid label creeping, the current literature on language-based security takes the route of resetting pc label after exiting from a control structure.

2.1.1 Tracking PC Labels(Stefan et al., 2011)

The method described for Haskell envisaged in (Stefan et al., 2011) uses a label for current control without reinitializing every time the control exits a statement. A labeled IO Haskell library unit `LIO` is built to track a single mutable *current label* (similar to *pc*) at run-time, and allows access to IO operations. The unit is responsible to ensure that the current label keeps track of all the observed data, and regulates label modifications. At each computation `LIO` keeps tracks of the *current label* and allows access to IO functionality e.g., labeled file systems. The current label is evaluated as an *upper bound* of all the labels observed during program execution.

2.2 Labelling Schemes: A Summary

Table 5 summarizes possible ways of binding labels with objects.

Some programs where ignoring the label of program point leads to incorrect certification are given in Tables 6 and 7. Table 6 shows an example with information leaks due to abnormal termination of a program. The value of x can be calculated from the value of *sum* (maximum possible integer value) and y on terminating the program due to integer overflow.

Table 3: Analyzing information flow at each line from example in Table 1 according to static and dynamic labelling.

Line No.	Static labelling (Scheme 1)	Dynamic labelling (Scheme 2)
3	Label of z would be inferred in such a way that all the flows to and from z should be secured. If z is assigned to $\underline{x}$, the flow from $z \rightarrow y$ is not permitted.	Label of z is initialized to least confidential security class e.g. public $(\perp)$
4		As there is a flow from $x \rightarrow z$ z is updated to $\underline{x}$ such that $\underline{x} \leqslant \underline{z}$
5		
6	There is no way to label z so that all the flows are safe. For this reason the program is insecure.	There is a flow from $z \rightarrow y$. But $\underline{x}$ cannot flow into $\underline{y}$. Hence the program is insecure.
7		

Table 4: Analyzing information flow at each line of example shown in Table 2 according to static and dynamic labelling.

Line No.	Flow direction	Static labelling (Scheme 1)	Dynamic labelling (Scheme 2)
5		Label of a is automatically inferred in such a way that all the flows to and from a is secure. If a is labelled as $\underline{x} \oplus \underline{z}$ due to explicit flows from x and y to a, the constraint $\underline{x} \oplus \underline{z} \leqslant \underline{y}$ will not be satisfied because $\underline{z} \nleqslant \underline{y}$. As static labelling fails to label the local variable a, the program is insecure.	Label of a is initialized to least confidential security class e.g. public $\perp$
6	$x \rightarrow a$		Label of a is updated to label of x i.e. $\underline{x}$ so that x can flow to a.
7	$a \rightarrow y$		Flow is allowed as the constraint $\underline{x} \leqslant \underline{y}$ is satisfied.
8	$z \rightarrow a$		Label of a is updated to $\underline{x} \oplus \underline{z}$ so that flow is allowed as $\underline{z} \leqslant \underline{x} \oplus \underline{z}$. Hence the program is flow-safe.

Table 5: Binding labels with objects for certification.

pc Label→ / **Labelling↓**	**Reset**	**Monotonic**
Static	P_1	P_2
Hybrid	P_3	P_4

Table 6: Information-flow through abnormal termination (cf. Copy6 from (Robling Denning, 1982)).

Program	Label constraints	
y=0; int sum=0;	$\underline{y} = high$ $\underline{x} = high$	
while(true){ sum=sum+x; y=y+1; }	$\underline{sum} \oplus \underline{x} \leqslant \underline{sum}$ $\underline{y} \oplus Low \leqslant \underline{y}$	$\underline{sum} = \underline{x}$

There is an implicit flow from x to y although the assignment to y is conditioned on *sum*.

A non-terminating flow insecure program is shown in Table 7. Let us consider the given label of the global variables x and y are given as $\underline{x}$ and $\underline{y}$ such that $\underline{x} \nleqslant \underline{y}$. It can be observed that the variable y holds the value equal to x depending on the termination of the program.

Although `LIO` follows the label binding P_2 that keeps track of program point but the run-time monitor fails to stop adversary from obtaining the *high* values by observing the termination of programs. Later in the Section 3 we illustrate examples that manifest P_4 also covers P_2. Considering all the limitations discussed above, a compile-time monitor based on the labelling Scheme 3 and binding mechanism P_4 would be a better candidate for secure certification of programs. A comparative study is given in the Section 4 where we categorize the existing prominent IFC tools and platforms based on the labelling mechanisms.

Table 7: Information-flow through non-termination (Cf. Copy5 from (Robling Denning, 1982)).

Program	Label constraints
y=0;	$\underline{y} = high$
while(x==0)	$\underline{x} = high$
skip;	
y=1;	$\underline{y} \oplus Low \leqslant \underline{y}$

From the above studies, we can infer:

1. While static labelling has advantages for the security certification of programs, over-approximately annotated static labels of local variables may lead to imprecision, and adversely impacts the soundness of these approaches.
2. For certifying iterative programs (terminating, non-terminating, abnormally terminating including exceptions), annotating local variables with improper static labels, and following the scheme that resets the label of the program counter, often miss to capture both the forward label propaga-

tion, and impact on static labels due to repeated backward information flow and leads to a loss of precision and soundness.

3. While the certification approach of Denning generates the relevant constraints, it fails to assert the existence of possible labelling for local variables/objects that satisfy the initial labels of global variables/objects.

If we can compute labels (or policies) for local variables such that the flow security is satisfied at all the program points, then we could consider such programs to be secure. Thus the question will be: is there a dynamic labelling procedure that realizes the same? A sound dynamic labelling approach that overcomes the limitations of the current techniques listed above shall be presented in Section 3.

3 OUR APPROACH TO CERTIFICATION

Our approach of certification is based on a hybrid labelling of objects in the program, that could possibly be unrolled a finite number of times. Possibility (or other wise) of labelling the objects of the program leads to certification (or other wise) of the program; the same could be used in the execution monitor for checking flow security at run-time.

In the following, we describe our Dynamic Labelling (DL) algorithm. The algorithm finds its basis in Denning's certification semantics.

3.1 Dynamic Labelling Algorithm: DL

Notation: Let G be the set of global variables/objects, L the set of local variables of a program, $(var)(e)$ the set of variables appearing in expression e, pc the program counter, and $\lambda, \lambda_0, \lambda_1, \cdots$ the labelling functions that give the security label/sensitivity-level of variables and pc.

SV is a function that takes a statement/command as input, and returns the set of source variables appearing in it as output. TV is a function that takes a statement/command as input, and returns the set of target variables appearing in it as output. DL is a dynamic labelling procedure/function that takes a command, clearance level of the subject executing the program, and a labelling function, as inputs, and returns either a labelling or UNABLE TO LABEL as output.

λ_{init} denotes the initial labelling. $\forall x \in G : \lambda_{\text{init}}(x)$ is given, $\forall x \in L : \lambda_{\text{init}}(x) = \perp$, and $\lambda_{\text{init}}(\text{pc}) = \perp$, where '$\perp$' is the least restrictive security class or *public*. Let P be a given program together with initial labelling for the global objects. Let s denote the subject trying to execute the program, and cl denote his clearance. If $\text{DL}(P, \text{cl}, \lambda_{\text{init}})$ returns a valid labelling, then the program preserves information-flow security when executed by the subject s. If $\text{DL}(P, \text{cl}, \lambda_{\text{init}})$ returns 'UNABLE TO LABEL', then information-flow security will be violated if the program is executed by the subject s, and the algorithm exits at that point without proceeding further.

Algorithm DL is described in Table 8. It is illustrated through examples followed by its' soundness in the sequel.

Illustrative Examples:
We illustrate the advantages of our dynamic labelling procedure by analyzing the example programs from Section 2. Examples clearly highlight the advantages of the dynamic labelling procedure in capturing subtle information-flows through control flow path, non-termination / abnormal termination channels etc. For each example, initial labels of the global variables are provided. We assume that the subject executing the program has the highest security label, and thus ignore the clearance field; dynamic labelling is shown in a tabular form.

We apply the proposed algorithm to the example shown in Table 2. It can be observed that the algorithm successfully labels the intermediate variable a.

Example 1. *Initial labels for global variables:* $\lambda(x) = \lambda(y) = \underline{x}$, $\lambda(z) = \underline{z}$

Consider the example shown in Table 6 where x and y are global variables having labels $\underline{x}$ and $\underline{y}$ respectively. At the point `y=y+1` the program fails to satisfy the constraint $\underline{y} \oplus \underline{pc} \leqslant \underline{y}$ because the label of pc is updated to $\underline{x}$ and $\underline{x} \nleqslant \underline{x}$. Therefore the algorithm declares the program as flow-insecure.

The algorithm if applied to the program in Table 7 identifies the flow violation as shown in Table 10: the label of pc is updated to $\underline{x}$ while testing the predicate; detects the flow violation at the statement `y=1` because the constraint $\underline{pc} \leqslant \underline{y}$ does not satisfy. Hence the algorithm fails to proceed further, and declares the program as flow-insecure, thus detects the control point where a certain object can leak information.

Example 2. *Global variable(s): x, y; No local variables.*
Initial labels for global variables: $\lambda(x) = \underline{x}$, $\lambda(y) = \underline{y}$

3.2 Soundness of Algorithm DL

In this section, we establish the termination of our dynamic labelling algorithm, and its soundness w.r.t.

Table 8: Description of Algorithm DL.

1. $S : skip$**::** SV(S)=$\{\emptyset\}$; TV(S)=$\{\emptyset\}$; DL(S,cl,λ) : return λ

```
2. S : x := e:: SV(S)=var(e); TV(S)={x};
DL=(S, cl, λ):
  tmp = $\bigoplus_{v \in \text{var}(e) \cap G} \lambda(v)$
  if (tmp $\not\leqslant$ cl) then
    exit ‘UNABLE TO LABEL’
  $\lambda_1 = \lambda$
  $\lambda_1(\text{pc}) = \lambda(\text{pc}) \oplus \text{tmp}$
  if $x \in L$ :
    $\lambda_1(x) = \lambda(x) \oplus \lambda(\text{pc}) \oplus \text{tmp}$
    return $\lambda_1$
  if $x \in G$ :
    if $\left(\left[\lambda(\text{pc}) \oplus \text{tmp} \oplus \text{cl}\right] \leqslant \lambda(x)\right)$ then
      return $\lambda_1$
    else exit ‘UNABLE TO LABEL’
```

```
3. S : if e then S_1[ else S_2]::
SV(S)=SV($S_1$) ∪ SV($S_2$) ∪ var($e$);
TV(S)=TV($S_1$) ∪ TV($S_2$)
DL(S, cl, λ):
  tmp = $\bigoplus_{v \in \text{var}(e) \cap G} \lambda(v)$
  if (tmp $\not\leqslant$ cl) then
    exit ‘UNABLE TO LABEL’
  $\lambda' = \lambda$
  $\lambda'(\text{pc}) = \lambda(\text{pc}) \oplus \text{tmp}$
  $\lambda_1 = \text{DL}(S_1, \text{cl}, \lambda')$
  $\lambda_2 = \text{DL}(S_2, \text{cl}, \lambda')$
  $\lambda_3(\text{pc}) = \lambda_1(\text{pc}) \oplus \lambda_2(\text{pc})$
  $\forall x \in L : \lambda_3(x) = \lambda_1(x) \oplus \lambda_2(x)$
  return $\lambda_3$
```

```
4. S : while e then S_1::
SV(S)=SV($S_1$) ∪ var($e$);
TV(S)=TV($S_1$);
DL(S, cl, λ):
  tmp = $\bigoplus_{v \in \text{var}(e) \cap G} \lambda(v)$
  if (tmp $\not\leqslant$ cl) then
    exit ‘UNABLE TO LABEL’
  $\lambda_1 = \lambda$
  $\lambda_1(\text{pc}) = \lambda(\text{pc}) \oplus \text{tmp}$
  $\lambda_2 = \text{DL}(S_1, \text{cl}, \lambda_1)$
  if ($\lambda_2 \neq \lambda_1$)
    $\lambda_1 = \lambda_2$
    $\lambda_2$ = DL(while e do $S_1$,
      cl, $\lambda_1$)
  return $\lambda_2$
```

```
5. S : S_1;S_2;::
SV(S)=SV($S_1$) ∪ SV($S_2$);
TV(S)=TV($S_1$) ∪ TV($S_2$);
DL(S, cl, λ):
  return DL($S_2$, cl, DL($S_1$, cl, λ));


Here, problem of “insecurity” will be
indicated by one of the recusive calls.
```

Note that “UNABLE TO LABEL” yields the control point where a certain object fails to satisfy the information flow policy.

Table 9: DL successfully labels that was no possible by static labelling.

Statement	*pc* Label	Label of local variable(a)
	$\perp$	$\perp$
int a=x;	$\underline{x}$	$\underline{x}$
y=a;	$\underline{x}$	$\underline{x}$
a=z;	$\underline{x} \oplus \underline{z}$	$\underline{x} \oplus \underline{z}$

Table 10: DL detects “insecurity” by failing to label a non-terminating flow.

Statement	*pc* Label
	$\perp$
y=0;	$\perp$
while x==0 skip;	$\underline{x}$
y=1;	UNABLE TO LABEL

non-interference.

Clearly, the procedure in Table 8 always terminates and is efficient. In fact, it is linear in the size of the program. This fact is formally established through the propositions below.

Proposition 1. *DL(S, cl, λ) always terminates for any program S not containing iteration, any label cl, and any labelling λ.*

Proof. The proof is by structural induction. For the base case, it is trivial to observe that the proposition holds for `skip`, and $x := e$.

For the inductive step, it is easy to prove that if the proposition holds for S_1 and S_2, then it also holds for `if` e `then` S_1 `else` S_2, and for $S_1;S_2$. $\square$

Proposition 2. *For any program S not containing iteration, any label cl, and any labelling λ, if DL(S, cl, λ) returns a valid labelling λ_1, then $\lambda_1(pc) = \lambda(pc) \bigoplus_{v \in SV(S) \cap G} \lambda(v)$.*

The proof of this proposition is by structural induction and is omitted for brevity.

Proposition 3. *$DL(\texttt{while}\ e\ \texttt{do}\ S, cl, \lambda)$ always terminates for any program S not containing iteration, any label cl, and any labelling λ.*

Proof. From the definition of DL, we note that the only case in which the evaluation of $DL(\texttt{while}\ e\ \texttt{do}\ S, cl, \lambda)$ does not terminate is when either the evaluation of $DL(S, cl, \lambda_1)$ does not terminate, or $DL(\texttt{while}\ e\ \texttt{do}\ S, cl, \bullet)$ goes into an infinite recursion.
The former is impossible due to Proposition 1. Impossibility of the latter is shown by considering the evaluation of $DL(\texttt{while}\ e\ \texttt{do}\ S, cl, \lambda)$:

1. $\lambda_1 = \lambda$, $\lambda_1(\mathrm{pc}) = \lambda(\mathrm{pc}) \bigoplus_{v\in \mathrm{var}(e)\cap G} \lambda(v)$
2. $\lambda_2 = \mathrm{DL}(S, \mathrm{cl}, \lambda_1)$
3. If $\lambda_2 = \lambda_1$ the evaluation terminates and there is nothing to prove. So we assume that $\lambda_2 \neq \lambda_1$. In this case $DL(\texttt{while}\ e\ \texttt{do}\ S, cl, \lambda_2)$ is invoked which proceeds as follows.
4. $\lambda_3 = \lambda_2$, $\lambda_3(\mathrm{pc}) = \lambda_2(\mathrm{pc}) \bigoplus_{v\in \mathrm{var}(e)\cap G} \lambda_2(v)$
5. $\lambda_4 = \mathrm{DL}(S, \mathrm{cl}, \lambda_3)$
6. If $\lambda_4 = \lambda_3$ the evaluation terminates and there is nothing to prove. So we assume that $\lambda_4 \neq \lambda_3$. In this case $DL(\texttt{while}\ e\ \texttt{do}\ S, cl, \lambda_4)$ is invoked which proceeds as follows.
7. $\lambda_5 = \lambda_4$, $\lambda_5(\mathrm{pc}) = \lambda_4(\mathrm{pc}) \bigoplus_{v\in \mathrm{var}(e)\cap G} \lambda_4(v)$
8. $\lambda_6 = \mathrm{DL}(S, \mathrm{cl}, \lambda_5)$

We claim that $\lambda_6 = \lambda_5$. The proof is given below.

1. First iteration: $\lambda_1(pc) = \lambda(pc) \oplus_{v\in var(e)\cap G} \lambda(v)$
$$\begin{aligned}\lambda_2(pc) &= \lambda_1(pc) \oplus_{v\in SV(S)\cap G} \lambda(v)\\ &= \lambda(pc) \oplus_{v\in var(e)\cap G} \lambda(v) \oplus_{v\in SV(S)\cap G} \lambda(v)\\ &= \lambda(pc) \oplus_{v\in (var(e)\cup SV(S))\cap G} \lambda(v)\end{aligned} \tag{1}$$
2. Second iteration:
$$\begin{aligned}\lambda_3(pc) &= \lambda_2(pc) \oplus_{v\in var(e)\cap G} \lambda(v)\\ &= \lambda(pc) \oplus_{v\in (var(e)\cup SV(S))\cap G} \lambda(v)\\ &\oplus_{v\in var(e)\cap G} \lambda(v)\\ &= \lambda(pc) \oplus_{v\in (var(e)\cup SV(S))\cap G} \lambda(v)\end{aligned} \tag{2}$$
$$\begin{aligned}\lambda_4(pc) &= \lambda_3(pc) \oplus_{v\in SV(S)\cap G} \lambda(v)\\ &= \lambda(pc) \oplus_{v\in (var(e)\cup SV(S))\cap G} \lambda(v)\\ \lambda_4(x) &= \lambda_3(x) \oplus \lambda_3(pc) \oplus_{v\in SV(S)\cap G} \lambda(v)\\ &= \lambda_3(x) \oplus \lambda_3(pc)\end{aligned}$$
This is because the label of PC is already influenced by all the global variables in S. (3)
$$= \lambda_2(x) \oplus \lambda(pc) \oplus_{v\in (var(e)\cup SV(S))\cap G} \lambda(v)$$
3. Third iteration:
$$\begin{aligned}\lambda_5(pc) &= \lambda_4(pc) \oplus_{v\in var(e)\cap G} \lambda(v)\\ &= \lambda(pc) \oplus_{v\in (var(e)\cup SV(S))\cap G} \oplus_{v\in var(e)\cap G} \lambda(v)\\ &= \lambda(pc) \oplus_{v\in (var(e)\cup SV(S))\cap G} \lambda(v)\\ \lambda_5(x) &= \lambda_4(x)\\ &= \lambda_2(x) \oplus \lambda(pc) \oplus_{v\in (var(e)\cup SV(S))\cap G} \lambda(v)\end{aligned} \tag{4}$$
$$\begin{aligned}\lambda_6(pc) &= \lambda_5(pc) \oplus_{v\in SV(S)\cap G} \lambda(v)\\ &= \lambda(pc) \oplus_{v\in (var(e)\cup SV(S))\cap G} \lambda(v)\\ \lambda_6(x) &= \lambda_5(x) \oplus \lambda_5(pc) \oplus_{v\in SV(S)\cap G} \lambda(v)\\ &= \lambda_2(x) \oplus \lambda(pc) \oplus_{v\in (var(e)\cup SV(S))\cap G} \lambda(v)\end{aligned} \tag{5}$$

It can be observed that $\lambda_6 = \lambda_5$. Thus, we can conclude that for the iteration statement, the dynamic labelling procedure terminates after a maximum of three iterations. □

Combining propositions 1 and 3 leads to the following proposition.

Proposition 4. *$DL(\texttt{while}\ e\ \texttt{do}\ S, cl, \lambda)$ always terminates for any program S, any label cl, and any labelling λ.*

Proposition 4 immediately establishes termination of DL as formalized below.

Proposition 5. *$DL(S, cl, \lambda)$ always terminates for any program S, any label cl, and any labelling λ.*

Next, we prove some results that highlight the important characteristics of our dynamic labelling procedure.

Proposition 6. *During the dynamic labelling of any program S with any clearance cl, i.e. during the evaluation of $DL(S, cl, \lambda_{init})$, $\lambda(pc) \leqslant cl$ always holds.*

Proof. In the initial state $\lambda_{init}(pc) = \bot \leqslant cl$. From the definition of DL, we note that the label of pc gets updated by taking its LUB with tmp only when $tmp \leqslant cl$. Therefore $\lambda(pc) \leqslant cl$ always holds due to simple lattice properties. □

Proposition 7. *An assignment to a global variable x is deemed safe by the dynamic labelling algorithm for a program executing with clearance cl if and only if $cl \leqslant \lambda(x)$.*

Proof. (Necessity of $cl \leqslant \lambda(x)$) From the definition of DL for an assignment statement, it is immediately clear that if the operation is deemed safe then it must be the case that $cl \leqslant \lambda(x)$.

(Sufficiency of $cl \leqslant \lambda(x)$) Note that we have $\lambda(pc) \leqslant cl$ from Proposition 6, and for the control to reach the point, we need $tmp \leqslant cl$, thus reducing the check $(\lambda(pc) \oplus tmp \oplus cl) \leqslant \lambda(x)$ to $cl \leqslant \lambda(x)$. □

Proposition 8. *During the dynamic labelling of any program S, $\lambda(pc)$ is monotonically non-decreasing.*

The proof of the above is trivially obtained by structural induction and is omitted for brevity.

Next, we prove a generalization of the result in Proposition 2.

Proposition 9. *For any program S, any label cl, and any labelling λ, if $DL(S,cl,\lambda)$ returns a valid labelling λ_1, then $\lambda_1(pc) = \lambda(pc) \bigoplus_{v \in SV(S) \cap G} \lambda(v)$.*

The proof of this proposition is by structural induction and is omitted for brevity.

Proposition 10. *During the dynamic labelling of any program S, for all $x \in L$, $\lambda(x)$ is monotonically non-decreasing.*

Proof. For $x \in L$, the label of x is updated by the dynamic labelling procedure only in the case of explicit assignment. In this case the label of x changes to $\lambda(x) \oplus \lambda(pc) \oplus tmp$. Monotonicity of $\lambda(pc)$ immediately gives us that $\lambda(x)$ is also monotonically non-decreasing. □

Proposition 11. *During the dynamic labelling of any program S i.e. $DL(S,cl,\lambda_{init})$, $\forall x \in L$ $\lambda(x) \leqslant \lambda(pc)$ always holds.*

Proof. In the initial state we have $\lambda_{init}(x) = \lambda_{init}(pc) = \perp$. We will show that every time the label of x is updated, the property holds in the new state also.

- $S :: x := e$

$$\begin{aligned} &\lambda_1(x) \leqslant \lambda(x) \oplus \lambda(pc) \oplus tmp \\ &\lambda_1(x) \leqslant \lambda(pc) \oplus tmp \text{ [by hypothesis } \lambda(x) \leqslant \lambda(pc)] \\ &\lambda_1(x) \leqslant \lambda_1(pc) \end{aligned} \tag{6}$$

- $S ::$ if e then S_1 else S_2

$$\begin{aligned} &\lambda_3(x) \leqslant \lambda_1(x) \oplus \lambda_2(x) \\ &\lambda_3(x) \leqslant \lambda_1(pc) \oplus \lambda_2(pc) \text{ [by hypothesis]} \\ &\lambda_3(x) \leqslant \lambda_3(pc) \end{aligned} \tag{7}$$

□

Relation with Non-interference:
In this section, we establish that the dynamic labelling algorithm is sound w.r.t. non-interference (Volpano et al., 1996).

A simple example illustrates the relation with non-interference. Consider the program P_1: $x := y$; - where x, y are global objects, and the labelling $\lambda(y) = l_2$, $\lambda(x) = l_3$, where l_2 and l_3 come from a total order $l_1 \leqslant l_2 \leqslant l_3 \leqslant l_4$. Consider four subjects s_1, s_2, s_3, and s_4, with clearances l_1, l_2, l_3, and l_4 respectively.

P_1 is non-interfering. Dynamic labelling of P_1 succeeds only for subjects s_2 and s_3. Dynamic labelling fails for s_1 because his clearance is below y, and therefore should not be allowed to access y. Similarly, dynamic labelling fails for s_4 because his clearance is above x and therefore should not be allowed to update x.

In the following, we shall formally establish the soundness of the dynamic labelling procedure w.r.t. non-interference. Note that globals are the only observables in this setting.

Theorem 1 (Soundness)**.** *If there exists a subject for which a program is declared secure by the dynamic labelling procedure in Table 8, then the program is non-interfering.*

Proof. For reasoning about value based non-interference, it suffices to work with the last update to a low labelled variable. From the definition of DL, we observe that the only place where a global variable is potentially updated is guarded by the condition $\lambda(pc) \oplus tmp \oplus cl \leqslant \lambda(x)$. In particular, since we are dealing with $\lambda(x) = \text{low}$, and the program is declared secure by the dynamic labelling procedure, we can immediately infer that $\lambda(pc) = tmp = cl = \text{low}$. This guarantees that no high labelled variable could have been accessed by this time in the execution. □

Traditional methods for the security certification of programs do not consider the subject labels. Let $DL_1(S,\lambda)$ be a modified dynamic labelling algorithm obtained by ignoring cl from the algorithm given in Table 8. We now prove that even this algorithm is sound w.r.t non-interference.

Theorem 2. *If a program S is declared secure by the procedure DL_1 i.e., $DL_1(S,\lambda)$ returns a valid labelling, then the program is non-interfering.*

Proof of this theorem is exactly the same as the proof of the previous theorem, and is omitted.

Finally, the set of programs declared secure by traditional certification methods that reset PC and use static labels for variables (Jif is a prominent representative of this class) is incomparable to the set of programs declared secure by our dynamical labelling algorithm as shown below.

Proposition 12. *There are programs declared secure by static labelling that cannot be dynamically labelled (insecure by our definition), and there are programs declared secure by our approach that static labelling rejects as insecure.*

Table 11: Comparison of IFC tools and platforms.

Tools and Platforms	Labelling mechanism	Flow -sensitive	Termination -sensitive
Jif	P_1	✗	✗
Paragon	P_1	✗	✗
FlowCaml	P_1	✗	✗
λ_{DSec}	P_3	✓	✗
LIO	P_2	✗	✗
$\lambda_l^{\texttt{LIO}}$	P_4	✓	✓
Aeolus	P_1	✗	✗
DL	P_4	✓	✓

Proof: Programs in Tables 6 and 7 provide an example for the former, while the program in Table 2 provides an example for the latter.

4 COMPARISON WITH RELATED WORK

In this section, we first briefly describe tools and platforms that enforce rich information flow policies, and then compare our approach with the existing approaches. Further, we discuss the applicability and limitations for certifying different classes of programs like (i) *termination-sensitive* programs, (ii) concurrent/non-deterministic programs, etc.

Information Flow Tools:
In the last decade a large number of information flow secure tools have been developed to enforce rigorous flow security policies through prevailing programming languages. For example, Jif (Myers et al., 2001), JOANA (Hammer and Snelting, 2009), Paragon (Broberg et al., 2013) for Java, FlowFox (De Groef et al., 2012), JSFlow (Hedin et al., 2014), IFC4BC (Bichhawat et al., 2014) for JavaScript, FlowCaml (Simonet and Rocquencourt, 2003) for Caml, λ_{DSec} (Zheng and Myers, 2007) for lambda calculus, LIO (Stefan et al., 2011), HLIO (Buiras et al., 2015) for Haskell and SPARK flow analysis (Barnes, 2003) for SPARK. Also flow secure platforms for instance, Jif/split (Zdancewic et al., 2002), Asbestos (Efstathopoulos et al., 2005), HiStar (Zeldovich et al., 2006), Flume (Krohn et al., 2007), Aeolus (Cheng et al., 2012) and flow checking systems that implements *sparse information labeling* (Austin and Flanagan, 2009), permissive-upgrade strategy (Austin and Flanagan, 2010) and identifies public labels and delayed exception (Hritcu et al., 2013) incorporate different label mechanisms shown in Table 11. While we have omitted some similar prominent platforms for lack of space, it may be noted that DL realizes the needed characteristics required for IFC.

The earliest attempt to capture flow-sensitive labels at run-time was observed in the work on λ_{DSec}. This was the first to propose general dynamic labels whose type system was proved to enforce non-interference. The core language λ_{DSec} is a security-typed lambda calculus that support first-class dynamic labels where labels can be checked and manipulated at run-time. Also labels can be used as a statically analyzed type annotations. The type system of λ_{DSec} prevents illegal information flows and guarantees that any well-typed program satisfies the non-interference property. In this language the label of the *pc* is a lower bound on the memory effects of the function, and an upper bound on the *pc* label of the caller but unlike DL, *pc* is not updated dynamically after executing each statement hence the language falls in the category P_3. The non-interference property discussed in λ_{DSec} is termination-insensitive, and also does not deal with timing channels. In the following, we provide a detail discussion on `LIO` that shares a common paradigm, and also subsumes the results of λ_{Dsec}.

Comparison with (Stefan et al., 2011):
Here, the authors have built a labelled `IO` Haskell library, called `LIO`, for certifying Haskell programs. `LIO` tracks a single mutable *current label* (like program-counter) at run-time and allows access to IO operations. The unit is responsible to ensure that the current label keeps track of all the observed data and regulate label modification. A type constructor `Labeled` is used to hold the restriction on a value and is mutable during run-time. At each computation `LIO` keeps tracks of *current label* and allow access to IO functionality e.g., labeled file systems. Current label is evaluated as *upper bound* of all the label observed during the program execution.

Consider reading a secret reference: `a` $\leftarrow$ `readLIORef secret`, where the value "secret" is labeled as L_S. Now to satisfy the information flow check i.e., (L_S `canFlowTo` L_C) the *current label* shall rise to (L_C `join` L_S) to read the secret value. Note that the value a is not labeled explicitly. Now let us take an example that wish to write the value of an object (a) to an output channel: `writeLIORef output a`, where the output channel is labeled as L_O (set dynamically according to the user executing the command). It is only permissible to modify or write data into the output channel when (L_C `canFlowTo` L_O) is satisfied. A second label *current clearance* (L_{cl}) provides an upper bound to *current label*. Hence, the computation cannot create, read or write to objects labeled L if L `canFlowTo` L_{cl} == `False`.

Although our approach overlaps with that of `LIO`, there are subtle differences that are briefed below:

- Unlike statically evaluating the labels in our DL algorithm, the approach in `LIO` is based on run-time *floating-label* system.

- Compared to DL algorithm, the `LIO` library provides IO actions that perform *termination-insensitive* flow analysis.
- Due to *flow-insensitive* labelling `LIO` does not provide sensitivity level of each intermediate object precisely.

We illustrate each of these points in the following.

Comparison of Labelling Mechanism:
The characterization of security labels when associated with objects is an important aspect of IFC analysis (Hunt and Sands, 2006). Security labels of subjects/objects can be mutable or immutable. *Flow-sensitive* IFC monitors allows to change the security labels over the course of the computation thus increase the permissiveness, and also alleviate the burden of explicit label annotations. Note that these monitors perform the flow analysis during execution-time or compile-time. Mutable label flow analysis during execution-time helps to precisely determine the flow-sensitivity of objects at run-time but compile-time analysis reduces the incident of *false-alarms*, and allows more programs as secure.

`LIO` keeps track of a single mutable *current* floating-label that is elevated (e.g., from *low* to *high*) at run-time to accommodate reading sensitive data, hence, `LIO` is *flow-sensitive* in *current label*. However, `LIO` is *flow-insensitive* in intermediate object labels. To allow more programs by the run-time monitor, an extension of `LIO` presented by (Buiras et al., 2014) that safely manipulates a label on the reference label. A *label on the label* describes the confidentiality of the `LIO` reference label itself. The run-time monitor upgrades a label of a reference only if that *label on the label can flow to floating-label*. Note that `LIO` follows the labelling mechanism P_2 whereas the proposed extension incorporates P_4. Another extension of `LIO` monad i.e., HLIO that provides programmers the flexibility to defer flow check of part of the program to run-time (like `LIO`) or static-time (unlike `LIO`), boosts permissiveness of the monitor.

Algorithm DL, is *flow-sensitive* in the absence of method-calls and a compile-time monitor built upon it would satisfy P_4, and hence it is more permissive than the other approaches highlighted above. DL, follows a hybrid labelling approach where the labels of global variables are assumed to be fixed and label of each intermediate object is allowed to vary, thus *flow-sensitive* dynamic labels are obtained.

***Termination-insensitive* Flow Analysis:**
Information leak depending on the termination of the program may remain undetected by the run-time monitor that extends `LIO` library unit. A program that exploits `toLabeled` function as shown by (Stefan et al., 2012a) may lead to information leak through termination channel. `toLabeled` l m executes the `LIO` operation m and encapsulates the returned value with label l. However the function does not increase the *current label*. Hence one can write an iterative program that executes a `toLabeled` function depending on a secret value or diverge otherwise. Assuming the initial *current label* as *low*, and as it remains unchanged even after executing `toLabeled`, an adversary can determine the secret value by observing termination of the program through standard output.

The DL algorithm keeps track of the sensitive labels observed by the *pc* at compile-time, therefore a program that tries to pass termination information to standard output shall abide by the information flow policy. Hence the proposed labelling approach performs an exemplary *termination-sensitive* flow analysis.

Applicability in the Concurrent Context:
As initially `LIO` was not considered for dynamic flow-sensitive concurrent settings, an extension is proposed in (Stefan et al., 2012a) that mitigates and eliminates termination and timing channels in concurrent programs. In that article a separate *current label* for each thread is mentioned that keeps track of the sensitivity of the data it has observed, and restrict the locations to which the thread can write. Hence, while termination and timing of these threads that may expose the secret values, the thread requires to raise it's *current label*. This prevents lower security threads from observing confidential information written in shared locations. Another extension of `LIO` proposed by (Buiras et al., 2014) provides the primitive of automatic upgrade that safely updates *flow-sensitive* label references. Both the extensions are shown to be equally applicable for concurrent context. However, the extensions may not stop the concurrent programs (shown in Table 12) from revealing the secret value.

An example of information leakage due to concurrent access is presented by (Le Guernic, 2007). The order of the assignments to x and y variables in VIP program depends on the secret value of h. The program in Newsmonger runs simultaneously and prints the values of x and y inside an infinite loop that are shared variables having no explicit label. If Newsmonger runs in between any of the assignments that exist in line 3 and 5 of VIP, it could reveal the value of h . Assuming the assignments as `LIO` operations, and the labels of the *current label* and h are L_C and L_h respectively, the label of y is evaluated as L_C `join` L_h when h is false. Before labelling of x is done by LIO, Newsmonger might disclose the value in x which in turn would reveal the secret value h.

Table 12: (a) VIP (left) and (b) Newsmonger (right).

```
x:=0; y:=0
if h then
  x:=1; y:=1;
else
  y:=1; x:=1;
end;
```

```
while true do
  output x;
  output y;
done;
```

It can be observed that a run-time monitor may unroll the threads execution multiple times before discovering a possible leak. A compile-time monitor also needs to take care of all possible combinations of threads executions and label the shared locations accordingly to identify any information flow violation. In our approach the DL procedure would identify the threads responsible to write the shared locations, and thus can optimize the number of threads to iterate the algorithm until the label of the locations converge in the lattice. The proof for the proposition 5 shown in the Section 3.2 for a sequential program can be extended easily for concurrent context to show that the procedure requires a finite number of unrolling rather than a full termination to converge the labels. We are working on this aspect from a proof perspective as well as from the point of of view of language platform.

Determining the Label of Intermediate Variables: As early as 1975, Dorothy Denning proposed in her thesis (Denning, 1975) a run-time source-to-source transformation to guarantee flow security of programs having selection or iteration statements. Basically, the method introduces additional code for checking possible flow violations at run-time. In a sense, her method, simulates possible information flows for each variable that has to lie between possible greatest and lowest levels. As against this, our method succinctly captures security labels of variables explicitly without introducing additional code in the program.

5 CONCLUSIONS

We have proposed a certification mechanism that is built on a hybrid labelling algorithm, DL. The algorithm DL is proved to be sound with respect to non-interference, and also we have established the termination of the algorithm. Further, it is shown to be more security precise as compared to other approaches, and also succinctly captures labels for termination-sensitive and progress-sensitive programs. Further, we have demonstrated that its characteristics make it possible to label concurrent/non-deterministic programs to monitor the flow; this property seems very useful for certifying security of such programs. Note that the labelling could be used both for static certification (that would involve bounded unrolling of loops) and for run-time monitoring; the latter will allow the execution only when the flow satisfies the flow policy or authorized thus enriching the precision.

We have been working on building a platform for certifying Python programs. So far, we have succeeded in certifying sequential programs including declassification rules. We are working on establishing the termination of the algorithm DL, in the context of concurrency with or without assertions. As mentioned already, our labelling approach is based on flow constraints, and hence, can be integrated with any security model including decentralized models that have declassification rules.

REFERENCES

Askarov, A. and Sabelfeld, A. (2009). Tight enforcement of information-release policies for dynamic languages. In *Proc. of 22nd IEEE CSF Symposium*, pages 43–59.

Austin, T. H. and Flanagan, C. (2009). Efficient purely-dynamic information flow analysis. In *Proc. of the ACM SIGPLAN 4th Workshop on PLAS*, pages 113–124.

Austin, T. H. and Flanagan, C. (2010). Permissive dynamic information flow analysis. In *Proc. of the 5th ACM SIGPLAN Workshop on PLAS*, page 3.

Barnes, J. G. P. (2003). *High integrity software: the spark approach to safety and security*. Pearson Education.

Bichhawat, A., Rajani, V., Garg, D., and Hammer, C. (2014). Information flow control in WebKit's javascript bytecode. In *Proc. of Int. Conf. on Principles of Security and Trust*, pages 159–178. Springer.

Broberg, N., van Delft, B., and Sands, D. (2013). Paragon for practical programming with information-flow control. In *APLAS*, pages 217–232. Springer.

Buiras, P., Stefan, D., and Russo, A. (2014). On dynamic flow-sensitive floating-label systems. In *Proc. of IEEE 27th CSF Symposium*, pages 65–79.

Buiras, P., Vytiniotis, D., and Russo, A. (2015). HLIO: Mixing static and dynamic typing for information-flow control in haskell. In *ACM SIGPLAN Notices*, volume 50, pages 289–301. ACM.

Cheng, W., Ports, D. R., Schultz, D. A., Popic, V., Blankstein, A., Cowling, J. A., Curtis, D., Shrira, L., and Liskov, B. (2012). Abstractions for usable information flow control in aeolus. In *USENIX Annual Technical Conference*, pages 139–151.

De Groef, W., Devriese, D., Nikiforakis, N., and Piessens, F. (2012). FlowFox: a web browser with flexible and precise information flow control. In *Proc. of ACM CCS*, pages 748–759.

Denning, D. E. (1976). A lattice model of secure information flow. *CACM*, 19(5):236–243.

Denning, D. E. and Denning, P. J. (1977). Certification of programs for secure information flow. *CACM*, 20(7):504–513.

Denning, D. E. R. (1975). Secure information flow in computer systems.

Efstathopoulos, P., Krohn, M., VanDeBogart, S., Frey, C., Ziegler, D., Kohler, E., Mazieres, D., Kaashoek, F., and Morris, R. (2005). Labels and event processes in the asbestos operating system. In *Proc. of 20th ACM SOSP*, volume 39, pages 17–30.

Goguen, J. A. and Meseguer, J. (1982). Security policies and security models. In *IEEE Symposium on SP*, pages 11–11.

Hammer, C. and Snelting, G. (2009). Flow-sensitive, context-sensitive, and object-sensitive information flow control based on program dependence graphs. *Int. Journal of Information Security*, 8(6):399–422.

Hedin, D., Birgisson, A., Bello, L., and Sabelfeld, A. (2014). JSFlow: Tracking information flow in javascript and its APIs. In *Proc. of 29th Annual ACM SAC*, pages 1663–1671.

Hicks, B., King, D., and McDaniel, P. (2007). Jifclipse: development tools for security-typed languages. In *Proc. of Workshop on PLAS*, pages 1–10.

Hritcu, C., Greenberg, M., Karel, B., Pierce, B. C., and Morrisett, G. (2013). All your IFCException are belong to us. In *IEEE Symposium on SP*, pages 3–17.

Hunt, S. and Sands, D. (2006). On flow-sensitive security types. In *ACM SIGPLAN Notices*, volume 41, pages 79–90.

Krohn, M. N., Yip, A., Brodsky, M. Z., Cliffer, N., Kaashoek, M. F., Kohler, E., and Morris, R. (2007). Information flow control for standard OS abstractions. In *Proc. of 21st ACM SOSP*, pages 321–334.

Le Guernic, G. (2007). Automaton-based confidentiality monitoring of concurrent programs. In *Proc. of 20th IEEE CSF Symposium*, pages 218–232.

Myers, A. C. (1999). JFlow: Practical mostly-static information flow control. In *Proc. 26th ACM Symposium on POPL*, pages 228–241.

Myers, A. C. and Liskov, B. (2000). Protecting privacy using the decentralized label model. *ACM TOSEM*, 9(4):410–442.

Myers, A. C., Zheng, L., Zdancewic, S., Chong, S., and Nystrom, N. (2001). Jif: Java information flow. *http://www.cs.cornell.edu/jif*.

Robling Denning, D. E. (1982). *Cryptography and data security*. Addison-Wesley Longman Publishing Co.

Ryan, P., McLean, J., Millen, J., and Gligor, V. (2001). Non-interference, who needs it? In *Proc. of 14th IEEE CSF Workshop*, pages 237–238.

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

Simonet, V. and Rocquencourt, I. (2003). Flow caml in a nutshell. In *Proc. of 1st APPSEM-II workshop*, pages 152–165.

Stefan, D., Russo, A., Buiras, P., Levy, A., Mitchell, J. C., and Maziéres, D. (2012a). Addressing covert termination and timing channels in concurrent information flow systems. In *ACM SIGPLAN Notices*, volume 47, pages 201–214.

Stefan, D., Russo, A., Mitchell, J. C., and Mazières, D. (2011). Flexible dynamic information flow control in haskell. In *ACM Sigplan Notices*, volume 46, pages 95–106.

Stefan, D., Russo, A., Mitchell, J. C., and Mazières, D. (2012b). Flexible dynamic information flow control in the presence of exceptions. *CoRR*, abs/1207.1457.

Volpano, D. M., Irvine, C. E., and Smith, G. (1996). A sound type system for secure flow analysis. *Journal of Computer Security*, 4(2/3):167–188.

Zdancewic, S., Zheng, L., Nystrom, N., and Myers, A. C. (2002). Secure program partitioning. *ACM Transactions on Computer Systems (TOCS)*, 20(3):283–328.

Zeldovich, N., Boyd-Wickizer, S., Kohler, E., and Mazières, D. (2006). Making information flow explicit in HiStar. In *Proc. of 7th Symp. on OSDI*, pages 263–278.

Zheng, L. and Myers, A. C. (2007). Dynamic security labels and static information flow control. *International Journal of Information Security*, 6(2-3):67–84.