Information Flow Security Certification for SPARK Programs

Sandip Ghosal, R. K. Shyamasundar

Information Flow Security Certification for SPARK Programs

Sandip Ghosal and R. K. Shyamasundar

Department of Computer Science and Engineering, Indian Institute of Technology, Bombay, Mumbai 400076, India
{sandipsmit,shyamasundar}@gmail.com

Abstract. SPARK 2014 (SPARK hereafter) is a programming language designed for building highly-reliable applications where safety and security are key requirements. SPARK platform performs a rigorous data/information flow analysis to ensure the safety and reliability of a program. However, the flow analysis is oriented towards establishing functional correctness and does not analyze for flow security of the program. Thus, there is a need to augment the analysis that would enable us to certify SPARK programs for security. In this paper, we propose an analysis to find information flow leaks in a SPARK program using a *Dynamic Labelling* (DL) approach for multi-level security (MLS) programs and describe an effective algorithm for detecting information leaks in SPARK programs, including classes of *termination/progress-sensitive* computations. Further, we illustrate the application of our approach for overcoming information leaks through unsanitized sensitive data. We also show how SPARK can be extended for realizing MLS systems that invariably need *declassification* through the illustration of an application of the method for security analysis of Needham-Schroeder public-key protocol.

1 Introduction

SPARK [1] is a programming language built on Ada 2012. While the SPARK data flow analysis [4,5] primarily emphasizes on establishing functional correctness, the flow security aspect remains neglected. Thus, certifying a program for flow security [8] is crucial for building reliable and secure applications.

A security property, often referred to as *information-flow policy* (IFP), governs the flow security certification mechanisms. One of the widely used policies for security certification first advocated in [8] says: if there is information flow from object x to y, denoted by $x \rightarrow y$, then the flow is secure if $\lambda(x)$ *can-flow-to* $\lambda(y)$, where λ is a labelling function that maps subjects (stakeholders of a program in execution) and objects (variables, files) of a program to the respective security *label* or *class* which describe the confidentiality and integrity of program values. The security labels together form a security lattice. In this paper, we are concerned with algorithmic techniques for security certifications of SPARK programs that comply with the IFP over a security lattice.

First, we highlight the security aspects of information flow and sanitization of sensitive data in SPARK programs following an assessment of proposed solutions in the literature.

1. Role of Implicit Flows: Two principle flows of information in a program are *direct* and *indirect* or *implicit* flows. A typical example of direct information flow is an explicit assignment of a secret value. Implicit flows arise when the control flow of a program is affected by secret values. Note that often secret information could be encoded in terms of differences in side effects of control flow. Such a property leads to various further classifications, such as termination-sensitive/insensitive or progress-sensitive/insensitive.
The work by Rafnsson *et al.* [17] was the first exploration towards the flow security in SPARK programs with the focus on termination-and progress-sensitive information leaks. We shall briefly describe the evidence of information leaks as shown in [17].
***a. Termination-insensitive flow analysis* in SPARK [17]:**
A *termination-sensitive* flow analysis can track an instance that depends on the program termination. However, it is evident from the example shown in Table 1 that SPARK follows a *termination-**in**sensitive* flow analysis. Note that the program outputs a character '!' depending on the termination of `if` block, which eventually terminates if the variable H is an odd number. If H is secret, then the program leaks one bit of sensitive information.

Table 1. SPARK program leaking information through a non-terminating loop.

```
procedure Leak (H:in out Byte) is
begin
  H:=H;
  if H mod 2 = 0 then
    while True loop
      H := H;
    end loop;
  end if;
  Write(Standard_Output,
    Character'Pos('!'));
end Leak;
```

Table 2. SPARK program leaking information progressively.

```
procedure Leak (H:in out Byte) is
  K: Byte := 0;
begin
  H:=H;
  while True loop
    Write(Standard_Output, K);
    if K >= H then
      while True loop
        H:=H;
      end loop;
    end if;
    K := K + 1;
  end loop;
end Leak;
```

***b. Progress-insensitive flow analysis* in SPARK [17]:**
A *progress-sensitive* analysis can track the completion of an instance through the continuous progress of a program. The program shown in Table 2 outputs all the numbers up to H through an intermediate variable K, thus progressively reveals the sensitive information in H. The program passes through the SPARK examiner, proving the flow analysis is *progress-**in**sensitive*.

The solution proposed in [17] identifies the terminating loops using *termination oracles* [14] and performs a graph transformation by introducing additional edges going out from potentially infinite loops to the end of the program. Therefore, it extends the dependency in the program and lets SPARK to perform dependency analysis on control flow graphs. While the approach avoids explicit exceptions, it needs to instrument the code from a global understanding, and further, there is no concrete feedback on the reasons for insecurity to the programmer. The main shortcomings of the above solutions are: (i) the transformations are not algorithmic (automatic), (ii) difficult to detect the issues of information leak. Nonetheless, the source-to-source transformation approach of [17] connects the theory of progress-sensitive non-interference with the practice, the SPARK data flow analysis together with the above approach is insufficient to enforce the classic notion of non-interference [6,11,20] while building MLS systems. Thus, it would be nice if an algorithmic strategy could be established that would enable us to overcome the above problems.

2. Sanitizing Data: Another aspect of security in SPARK, as highlighted by Chapman [7], concerns *sanitizing sensitive* local data in Ada-SPARK programs for building secure applications. The author elaborates potential leaks due to access to "unsanitized" sensitive temporary data, e.g., OS page files or core dump of a running process. For instance, in the simple decryption program shown in Table 3, the local variables N and D become sensitive and, if not sanitized, could leak secret information in S. Chapman demonstrates the issue of Ada language, where an attempt to sanitize sensitive data is suppressed by the compiler while performing optimization. In SPARK, the sanitization step is *ineffective* as it does not influence the final exported value. The discussion arises questions like:

> How do we define 'sensitive'? What objects in the program are 'sensitive'? How are they identified?

The author prescribes one possible solution, i.e., adopting coding policies to sanitize sensitive local data in the Ada-SPARK project, e.g., use of `pragma Inspection_Point`. Further, following the naming convention for the sensitive data, e.g., adding prefixes "Sensitive_", "_SAN" to the names of types, variables, would aid the programmer to handle it appropriately. Thus, it must be evident that there is a need to build a succinct definition of "sensitive" data and provide an algorithmic analysis for the compiler/runtime monitor for appropriate treatment.

Table 3. A simple decryption program written in SPARK.

```
procedure Decrypt(C: in Integer; S: in
     Private_Key; M: out Integer) is
  N,D:Integer;
begin
  N := Get_N(S);
  D := Get_D(S);
  M := (C**D) mod N;
  --Sanitize N and D
  N := 0;
  D := 0;
end Decrypt;
```

In this paper, we propose a single alternative solution to the above problems using a dynamic labelling algorithm [9,10] that not only helps to identify the sensitive local objects but also detects potential information leaks in SPARK programs, thus alleviate the burden of manual intervention in the source program. The automation receives a list of immutable security labels of global variables as input and generates mutable labels for the local variables while following the IFP throughout the computation. Primarily, objects sensitive to the outside world are considered as global, but the programmer may use her discretion to choose any variable as a global object.

Further, developing a secure application that involves objects with security labels from a multi-point general lattice often demands the need for *declassification*. We introduce the construct `Declassify` borrowing the notion of declassification from the security model `RWFM` [12,13] as briefed in the later section.

The main contributions of our work are summarized below:

1. Propose an algorithmic solution for SPARK statements using information flow policies and establish its capability to detect program points that could plausibly leak information.
2. Illustrate the efficacy of our approach in detecting information leaks primarily through termination channels [19] or progress channels [3] and discuss the advantages, such as localizing possible leaking points, identifying sensitive objects automatically.
3. Introduce "Declassify" construct for SPARK programming language based on the model proposed in [12] and illustrated its usage through an application on a cryptographic protocol.

The rest of the paper is organized as follows. Section 2 provides the necessary background for dynamic labelling algorithm and `RWFM` flow security model. Section 3 presents a single alternative solution to overcome the shortcoming of SPARK flow analysis using security labels and dynamic labelling algorithm. Section 4 discusses the necessity of *declassification* as an extension in the SPARK language and illustrate the `Declassify` construct with an application of Needham-Schroeder (NS) public key protocol. Finally, implementation highlights are given in Section 5 followed by conclusions in Section 6.

2 Background

In this section, we briefly discuss the dynamic labelling algorithm and provide an overview of a recently proposed flow security model `RWFM` that is used for labelling subjects and objects and governing information flow transitions while developing an MLS system in SPARK.

2.1 Dynamic Labelling Algorithm (DL)[9,10]

Let G and L be the sets of global and local variables of a program, and λ is a projection from subjects, objects, and program counter pc to its' respective security label from the lattice. Function var returns the set of variables appearing in expression e and SV, TV provide the set of source and target variables

respectively for a given statement S. The algorithm DL takes three parameters as inputs: basic SPARK statements such as assignment, selection, iteration, or sequence denoted by S; the highest security label cl of the executing subject referred to as *clearance*; and a labelling function λ where the global variables are mapped to their given immutable labels. If all the local variables are successfully labelled, the algorithm returns a new map λ' otherwise flags a message "UNABLE TO LABEL" on detecting a possible flow leak. Note that, initially, the mutable labels of all the local variables, including *pc* are labelled as *public* ($\bot$). The algorithm DL for basic control statements of SPARK language is described in Table 4.

Table 4. Description of Algorithm DL for basic SPARK statements such as assignment, selection, iteration and sequence

<table>
<tr><td colspan="2">1. S : null:: SV(S)=$\{\emptyset\}$; TV(S)=$\{\emptyset\}$; DL(S, cl, λ) : return λ</td></tr>
<tr><td>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 λ_1
if $x \in G$:
if $\left(\left[\lambda(\text{pc}) \oplus \text{tmp} \oplus \text{cl}\right] \leqslant \lambda(x)\right)$ then
return λ_1
else exit ‘UNABLE TO LABEL’</td><td>3. S : if e then S₁[else S₂] end if::
SV(S)=SV(S_1) $\cup$ SV(S_2) $\cup$ var(e);
TV(S)=TV(S_1) $\cup$ 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 λ_3</td></tr>
<tr><td>4. S : while e then S₁ end loop::
SV(S)=SV(S_1) $\cup$ 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 ($\lambda_2 \neq \lambda_1$)
$\lambda_1 = \lambda_2$
$\lambda_2 = \text{DL}($while e then S_1 end loop,
cl, λ_1)
return λ_2</td><td>5. S : S₁; S₂;::
SV(S)=SV(S_1) $\cup$ SV(S_2);
TV(S)=TV(S_1) $\cup$ TV(S_2);
DL(S, cl, λ):
return DL($S_2, \text{cl}, \text{DL}(S_1, \text{cl}, \lambda)$);

Here, problem of “insecurity” will be indicated by one of the recursive calls.</td></tr>
<tr><td colspan="2">Note that “UNABLE TO LABEL” yields the control point where a certain object fails to satisfy the information flow policy.</td></tr>
</table>

2.2 Readers-Writers Flow Model (RWFM)[12,13,15]: An overview

We provide a brief overview of the Readers-Writers Flow Model (RWFM) for information flow control.

Definition 1 (RWFM Label) *A RWFM label is a three-tuple (s, R, W) ($s, R, W \in$ set of principals/subjects P), where s represents the owner of the information and policy, R denotes the set of readers allowed to read the information, and W identifies the set of writers who have influenced the information so far.*

Definition 2 (can-flow-to relation ($\leqslant$)) *Given any two RWFM labels $L_1 = (s_1, R_1, W_1)$ and $L_2 = (s_2, R_2, W_2)$, the can-flow-to relation is defined as:*

$$\frac{R_1 \supseteq R_2 \quad W_1 \subseteq W_2}{L_1 \leqslant L_2}$$

Definition 3 (Join and meet for RWFM Labels) *The join ($\oplus$) and meet ($\otimes$) of any two RWFM labels $L_1 = (s_1, R_1, W_1)$ and $L_2 = (s_2, R_2, W_2)$ are respectively defined as*

$$L_1 \oplus L_2 = (-, R_1 \cap R_2, W_1 \cup W_2) \quad L_1 \otimes L_2 = (-, R_1 \cup R_2, W_1 \cap W_2)$$

The set of RWFM labels $SC = P \times 2^P \times 2^P$ forms a bounded lattice $(SC, \leqslant, \oplus, \otimes, \top, \bot)$, where $(SC, \leqslant)$ is a partially ordered set and $\top = (-, \emptyset, P)$, and $\bot = (-, P, \emptyset)$ are respectively the maximum and minimum elements.

Definition 4 (Declassification in RWFM) *The declassification of an object o from its current label (s_2, R_2, W_2) to (s_3, R_3, W_3) as performed by the subject s with label (s_1, R_1, W_1) is defined as*

$$\frac{s \in R_2 \quad s_1 = s_2 = s_3 \quad R_1 = R_2 \quad W_1 = W_2 = W_3 \quad R_2 \subseteq R_3 \quad (W_1 = \{s_1\} \vee (R_3 - R_2 \subseteq W_2))}{(s_2, R_2, W_2) \text{ may be declassified to } (s_3, R_3, W_3)}$$

This says, *the owner of an object can declassify the content to a subject(s) only if the owner is the sole writer of the information or that subject(s) had influenced the information earlier.*

3 Our Approach Using Dynamic Labelling Algorithm

Our approach relies on the application of dynamic labelling for the SPARK program. We first extend the set of rules given earlier with the labelling rules for the SPARK procedures as described below.

Extended Labelling Algorithm (DL$^+$) for SPARK Procedure:
Consider a procedure call, say $p(a_1, \ldots, a_m; b_1, \ldots, b_n)$ where, $a_1, \ldots, a_m$ are the actual input arguments and $b_1, \ldots, b_n$ are the actual input/output arguments corresponding to the formal input parameters (mode `IN`) $x_1, \ldots, x_m$ and formal input/output parameters (mode `OUT` or `IN OUT`) $y_1, \ldots, y_n$. The dynamic labelling algorithm for SPARK procedure call is shown in Table 5. Once given

a procedure call, the DL algorithm first computes the procedure body before returning the control to the caller. The algorithm adheres to the parameters passing mechanisms while transferring the control. Following are the operations the algorithm performs at entry & exit points of the procedure: (i) at the entry point it initializes the labels of formal input parameters with the corresponding labels of the actual input arguments; (ii) creates an instance *pc* local to the procedure; (iii) initializes the *pc* and local variables with the mutable label $\perp$; (iv) evaluates the procedure body; and (v) finally resets the *pc* label on exiting from the procedure and returns the final labels to the caller.

Note that, the intrinsic property of the labelling algorithm automatically enforces the required security constraints for a procedure call given in [18].

Table 5. DL Algorithm for a Procedure Call

6. $S : p(a_1, \ldots, a_m; b_1, \ldots, b_n)$ ::
DL(S, cl, λ):
  //Initialize the label of the parameters
  $\lambda' = \lambda_{\text{init}}$
  forall $i \in 1 \ldots m$, $\lambda'(x_i) = \lambda(a_i)$
  // Evaluate the body of the procedure
  λ_1 =DL$(p - body, cl, \lambda')$
  return λ_1

With the extended dynamic labelling algorithm for SPARK statements are in place (Tables 4, 5), certifying information flow security for a given SPARK program consists of the following steps:
1. Initialize the labels of global variables for the given SPARK program.
2. Apply the labelling algorithm DL$^+$, for the SPARK program.
3. If the labelling succeeds, then the program has no information leak; if the labelling algorithm outputs the message "UNABLE TO LABEL" it implies there is a possibility of information leak.

Illustration:
First, we apply the algorithm to the programs shown in Table 1, 2, and demonstrate solutions to issues 1(a)-(b) of Section 1. Consider H and `Standard_Output` are the global objects initialized with the immutable labels $\underline{H}$ and $\perp$ (i.e., *public*) respectively, such that information *cannot flow* from H to `Standard_Output`, i.e., $\lambda(H) = \underline{H} \not\leqslant \lambda(Standard\_Output) = \perp$. Also, we assume that the executing subject has the clearance label $\underline{H}$. Then the derived labels of local variables as well as *program counter* (*pc*) are shown in the Tables 6, 7.

(i) Detecting *termination-sensitive* information leaks:
In Table 6, it can be observed that since *pc* reads the variable H its label is raised to $\underline{H}$. Now execution of the procedure call `Write(Standard_Output, Character'Pos('!'))` causes a flow from *pc* to `Standard_Output`, therefore, the label of `Standard_Output` must be at least equal to the label of *pc*. Since the label of `Standard_Output` is immutable, the algorithm fails to update the label, hence exits by throwing the message "`UNABLE TO LABEL`". The point of failure detects the location and objects responsible for flow policy violation.

(ii) Detecting *progress-sensitive* information leaks:
Similarly, since the procedure `Write(Standard_Output, K)` causes information flow from both K and *pc* to `Standard_Output` it needs to satisfy the constraint

Table 6. SPARK program leaking information through a non-terminating loop. Clearance: $cl = \underline{H}$. Initial labels of global objects: $\lambda(H) = \underline{H}$, $\lambda(Standard\_Output) = \bot$.

Program	pc Label
`procedure Leak (`H`:in out Byte) is`	
`begin`	$\bot$
H`:=`H`;`	$\underline{H}$
`if` H `mod` $2 = 0$ `then`	
`while True loop`	
$H := H$`;`	
`end loop;`	
`end if;`	$\underline{H}$
`Write(Standard_Output,`	
`Character'Pos('!'));`	`UNABLE TO LABEL`
`end Leak;`	

Table 7. SPARK program leaking information progressively. Clearance: $cl = \underline{H}$. Initial labels of global objects: $\lambda(H) = \underline{H}$, $\lambda(Standard\_Output) = \bot$.

Program	Derived Labels
`procedure Leak (`H`:in out Byte) is`	
K`: Byte := 0;`	$\underline{K} = \bot$
`begin`	$\underline{pc} = \bot$
$H := H$`;`	$\underline{pc} = \underline{H}$
`while True loop`	
`Write(Standard_Output,` K`);`	`UNABLE TO LABEL`
`if` $K >= H$ `then`	
`while True loop`	
$H := H$`;`	
`end loop;`	
`end if;`	
$K := K + 1$`;`	
`end loop;`	
`end Leak;`	

$\lambda(K) \oplus \lambda(PC) \leqslant \lambda(Standard\_Output)$. But, the algorithm fails to continue as $\lambda(pc) \not\leqslant \bot$.

(iii) Identifying sensitive data for sanitization:
Here, we shall address the questions and solution related to handling "sensitive" data discussed in Section 1. Note that the labelling algorithm takes the initial classification of sensitive/non-sensitive data and transfer the sensitivity to local variables automatically during computation. Since the algorithm generates the labels of local variables from the given set of sensitive global variables, any attempt to access unsanitized sensitive local objects must satisfy the IFP, thus restrict access as required.

Consider the program shown in Table 3 where global objects C, S and M are sensitive data with the label given $\underline{H}$. Then applying the dynamic labelling algorithm would compute the labels of local variables N and D as $\underline{H}$ transferring

the sensitivity label of global variables. Thus any attempt to read the sensitive data by a less-sensitive user (or process) would indicate misuse of information flow.

Remarks: One can write statements like $H = H$ followed by $L = L$ (H and L are global and denote a *high* and *low* variables respectively) somewhere in the program. In such cases, the platform would indicate "UNABLE TO LABEL" as it fails to satisfy the constraint $\underline{H} \oplus \underline{pc} \leqslant \underline{L}$. We ignore such corner cases and leave the onus of correcting the code on the programmer.

3.1 Comparison with Rafnsson *et al.*[17] and SPARK Analysis

Table 8 provides a comparison of DL$^+$ with the SPARK analysis and approach proposed in [17] in terms of common objectives that are generally sought in the information flow analysis tools. From the comparison it is evident that our approach subsumes the advantages of other two approaches.

Table 8. Comparison of DL$^+$ with SPARK analysis and approach proposed in [17]

Objectives	SPARK Flow Analysis	Approach by [17]	DL$^+$ Algorithm
Termination-and progress-sensitive flow analysis	✗	✓	✓
Recurring backward information flow analysis in loop statements	✓	✗	✓
Precisely localize the program point violating flow policy	✗	✗	✓
Identify unauthorized access to unsanitized sensitive data	✗	✗	✓

4 Need of Declassification in MLS Systems

Quite often, in a decentralized labelling environment, it is required to relax the confidentiality level and reveal some level of information to specific stakeholders for the successful completion of the transaction. For this purpose, the notion of *Declassification* or *Downgrading* needs to be captured either implicitly or explicitly. We shall understand the context from the classic password update problem shown in Table 9.

Consider a function `Password_Update` that updates password database by a new password (*new_pwd*) only if the guess password (*guess_pwd*) provided by the user matches with the old password and updates the result accordingly. Note that there is a need to convey the *result* as an acceptance of the new password so that the user can use it further.

Table 9 shows that the *result* becomes sensitive data by the time the function returns its value. In the context of the MLS system, passing this sensitive data (i.e., *result*) to a less-sensitive entity (i.e., user) may violate the information flow policy. Therefore, it demands controlled declassification. There are two possibilities for introducing declassification at the point of returning the value: (i) Have

Table 9. Labelling password update program using DL. Initial labels of global variables: $\underline{pwd\_db} = \underline{a} \oplus \underline{b}$, $\underline{guess\_pwd} = \underline{new\_pwd} = \underline{b}$. *result* is a local variable and $cl = \underline{a} \oplus \underline{b}$.

Program	Derived Labels
`function Password_Update(`pwd_db`,`$guess\_pwd$`,`new_pwd`: Boolean)`	
`return Boolean is`	
`begin`	$\underline{pc} = \bot$
`  if` $pwd\_db = guess\_pwd$ `then`	$\underline{pc} = \underline{a} \oplus \underline{b}$
$pwd\_db := new\_pwd$;	$\underline{pc} = \underline{a} \oplus \underline{b}$
$result := True$;	$\underline{result} = \underline{a} \oplus \underline{b}$
`  else`	
$result := False$;	
`  end if;`	
`  return result;`	$\underline{result} = \underline{a} \oplus \underline{b}$
`end Password_Update;`	

an assertion that ensures declassification explicitly, or (ii) Perform declassification implicitly at the function return.

Declassification in SPARK: We adopt an explicit declassification mechanism for SPARK. The programmer may localize the program point that needs a declassification and place the construct `Declassify` to add specific readers. However, the addition needs to be robust as otherwise, the declassification may appear to be a mere discretionary that would have serious consequences in a decentralized model. For this reason, we shall borrow the "Declassification" rule from the RWFM model [12] as briefed in the Section 2.2.

Consider $p, p_1, \ldots$ range over the set of principals P, and x ranges over the set of objects/variables. Functions A, R and W have the form $f : L \to P$ which map to owner, readers and writers fields respectively for a given security label in the lattice L. Now let us assume principal p executes the statement `Declassify(x,`$\{p_1, \ldots, p_n\}$`)` to add $p_1, \ldots, p_n$ into the readers set of a variable x. Then we define the algorithm DL^+ for the `Declassify` statement as below:

$\mathbf{S}$: DL(`Declassify`$(\mathbf{x}, \{\mathbf{p_1}, \ldots, \mathbf{p_n}\}), \mathbf{cl}, \lambda)$ ::
DL(S, cl, λ) :
  if $(\lambda(x) \not\leqslant cl)$ then
    exit 'UNABLE TO LABEL'
  if $x \in G$ and $\lambda(pc) \not\leqslant \lambda(x)$:
    exit 'UNABLE TO LABEL'
  $\lambda_1 = \lambda$
  $tmp = \lambda(pc) \oplus \lambda(x)$
  $\lambda_1(pc) = tmp$
  if $A(\lambda(tmp)) = \{p\}$ and $(W(\lambda(tmp)) = \{p\}$ or $\{p_1, \ldots, p_n\} \subseteq W(\lambda(tmp)))$:
    $R(\lambda_1(x)) = R(\lambda(tmp)) \cup \{p_1, \ldots, p_n\}$
    return λ_1
  else
    exit 'UNABLE TO LABEL'

A look at the fragment of N-S public-key protocol [16]:
Table 10 shows an abstract program in SPARK demonstrating the N-S public-key protocol. The global variables are as follows: Aid represents the identity of the subject A; Na and Nb denote the fresh nonces created by the subject A and B respectively; Pub_a and Pub_b represent individual *public-key* of subjects A and B; Pri_a and Pri_b denote the *private-key* of A and B respectively. Initially, the local variables are readable by all the stakeholders (denoted by '*'), and nobody has influenced at this point. The functions `Encrypt`, `Decrypt` are executed by each of the subject A and B with the clearance level $\underline{a}$ and $\underline{b}$ respectively. Note that the program is self-explanatory, with the step numbers given in the comments depict the execution flow. Further, the generated labels for each variable are shown in the superscript.
A creates Pri_a that is accessible to A only, therefore labelled as $(A, \{A\}, \{A\})$. Similarly, Pri_b obtains a label $(B, \{B\}, \{B\})$. Further, A and B create the nonces Na, Nb respectively, that are readable by both the subjects, hence labelled as $(A, \{A, B\}, \{A\})$, $(B, \{A, B\}, \{B\})$. The public keys and identities of A and B are given identical labels as Na and Nb respectively. The clearance labels of the subjects A, B executing the programs are given as $\underline{a} = (A, \{A\}, \{A, B, S\})$ and $\underline{b} = (B, \{B\}, \{A, B, S\})$ respectively.
Note that the encrypted message, at step 2.2 obtains a label inaccessible to A, therefore, explicitly declassified by B so that A can decrypt the message for further use. Similarly, A also performs a declassification at step 3.3 so that B can access the data.

It follows from the above illustrations that declassification for MLS is essential, and the DL algorithm, along with the `RWFM` model, ensures appropriate automatic labelling to ensure security properties.

5 Implementation of DL$^+$ for SPARK

The implementation of our approach first generates RWFM labels for the global variables of a SPARK program from the given readers and writers set of respective variables. Once the labels are generated, the set of global variables annotated with the corresponding RWFM labels and the SPARK program are given as input to the dynamic labelling algorithm. The algorithm either outputs the labels of all the intermediate variables or throws the message "UNABLE TO LABEL" in the presence of possible flow leaks. Figure 1 presents a schematic diagram of the implementation. We have analyzed the programs discussed in this paper and tested in a security workbench for Python language under development using the dynamic labelling approach.

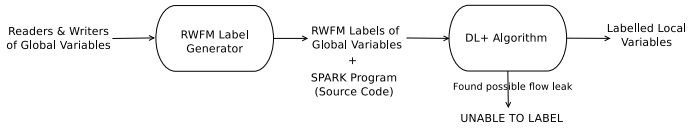

Fig. 1. A schematic diagram of our implementation.

Table 10. A prototype of a program for N-S public-key protocol [16]

Program	Derived Labels
procedure NS_Protocol_A(- - list of parameters) is	
- - Declarations of local objects Ma	$Ma^{(A,\{*\},\{\})}$
begin	$pc^{(S,\{*\},\{S\})}$
- - 1) A encrypts Na, Aid using B's key Pub_b	$pc^{(S,\{A,B\},\{A,B,S\})}$
Ma:=Encrypt(Aid,Na,Pub_b);	$Ma^{(A,\{A,B\},\{A,B,S\})}$
- - 3.1) A decrypts Mb using A's key Pri_a	$pc^{(S,\{A\},\{A,B,S\})}$
Ma:=Decrypt(Mb,Pri_a);	$Ma^{(A,\{A\},\{A,B,S\})}$
- - 3.2) A encrypts Na, Nb using B's key Pub_b	$pc^{(S,\{A\},\{A,B,S\})}$
Ma:=Encrypt(Na,Nb,Pub_b);	$Ma^{(A,\{A\},\{A,B,S\})}$
- - 3.3) A declassifies Ma	$pc^{(S,\{A\},\{A,B,S\})}$
Declassify(Ma,{B})	$Ma^{(A,\{A,B\},\{A,B,S\})}$
end NS_Protocol_A;	

procedure NS_Protocol_B(- - list of parameters) is	
- - Declarations of local objects Mb	$Mb^{(B,\{*\},\{\})}$
begin	$pc^{(S,\{*\},\{S\})}$
- - 2.1) B decrypts Ma using B's key Pri_b	$pc^{(S,\{B\},\{A,B,S\})}$
Mb:=Decrypt(Ma,Pri_b);	$Mb^{(B,\{B\},\{A,B,S\})}$
- - 2.2) B encrypts Na, Nb using A's key Pub_a	$pc^{(S,\{B\},\{A,B,S\})}$
Mb:=Encrypt(Na,Nb,Pub_a);	$Mb^{(B,\{B\},\{A,B,S\})}$
- - 2.3) B declassifies Mb	$pc^{(S,\{B\},\{A,B,S\})}$
Declassify(Mb,{A})	$Mb^{(B,\{A,B\},\{A,B,S\})}$
- - 4) B decrypts Ma using B's key Pri_b	$pc^{(S,\{B\},\{A,B,S\})}$
Mb:=Decrypt(Ma,Pri_b);	$Mb^{(B,\{B\},\{A,B,S\})}$
end NS_Protocol_B;	

6 Conclusions

In this paper, we have illustrated how an extended form of dynamic labelling algorithm integrated with RWFM provides an effective platform for flow security certification of SPARK programs, including termination- and progress-sensitive flows. The approach enables us to use an automatic compile-time labelling algorithm for data that aids in detecting information leaks due to termination and progress sensitivity, and unsanitized data. Also, the programmer gets feedback on the reasons for the misuse of information at the specific program point, enables him to refine the program only to realize flow security. These features add to the usability of the program. The approach provides a natural stepping stone for direct/implicit introduction of declassification for programming MLS systems in SPARK, thus preserve end-to-end confidentiality properties. So far, we have experimented on these aspects on our security workbench developed for Python programs. One can further develop an axiomatic proof system that follows naturally on similar lines as proposed in [2,18]. One of the distinct advantages of

our approach is keeping the SPARK analysis and security analysis orthogonal – thus, enabling technology adaptation easily to the SPARK platform.

References

1. Spark 2014. http://www.spark-2014.org/about
2. Andrews, G.R., Reitman, R.P.: An axiomatic approach to information flow in programs. ACM Transactions on Programming Languages and Systems (TOPLAS) 2(1), 56–76 (1980)
3. Askarov, A., Hunt, S., Sabelfeld, A., Sands, D.: Termination-insensitive noninterference leaks more than just a bit. In: European symposium on research in computer security. pp. 333–348. Springer (2008)
4. Barnes, J.G.P.: High integrity software: the spark approach to safety and security. Pearson Education (2003)
5. Bergeretti, J.F., Carré, B.A.: Information-flow and data-flow analysis of while-programs. ACM Transactions on Programming Languages and Systems (TOPLAS) 7(1), 37–61 (1985)
6. Boudol, G.: On typing information flow. In: International Colloquium on Theoretical Aspects of Computing. pp. 366–380. Springer (2005)
7. Chapman, R.: Sanitizing sensitive data: How to get it right (or at least less wrong...). In: Reliable Software Technologies – Ada-Europe 2017. pp. 37–52. Springer International Publishing (2017)
8. Denning, D.E., Denning, P.J.: Certification of programs for secure information flow. CACM 20(7), 504–513 (1977)
9. Ghosal, S., Shyamasundar, R.K., Kumar, N.V.N.: Static security certification of programs via dynamic labelling. In: Proceedings of the 15th International Joint Conference on e-Business and Telecommunications, ICETE 2018 - Volume 2: SECRYPT, Porto, Portugal, July 26-28, 2018. pp. 400–411 (2018)
10. Ghosal, S., Shyamasundar, R., Kumar, N.N.: Compile-time security certification of imperative programming languages. In: International Conference on E-Business and Telecommunications. pp. 159–182. Springer (2018)
11. Goguen, J.A., Meseguer, J.: Security policies and security models. In: IEEE Symposium on SP. pp. 11–11 (1982)
12. Kumar, N.V.N., Shyamasundar, R.K.: Realizing purpose-based privacy policies succinctly via information-flow labels. In: Big Data and Cloud Computing (BdCloud), IEEE 4th. Int. Conf. on. pp. 753–760 (2014)
13. Kumar, N.V.N., Shyamasundar, R.: A complete generative label model for lattice-based access control models. In: International Conference on Software Engineering and Formal Methods. pp. 35–53. Springer (2017)
14. Moore, S., Askarov, A., Chong, S.: Precise enforcement of progress-sensitive security. In: Proceedings of the 2012 ACM conference on Computer and communications security. pp. 881–893. ACM (2012)
15. Narendra Kumar, N., Shyamasundar, R.: Poster: dynamic labelling for analyzing security protocols. In: Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security. pp. 1665–1667. ACM (2015)
16. Needham, R.M., Schroeder, M.D.: Using encryption for authentication in large networks of computers. Communications of the ACM 21(12), 993–999 (1978)
17. Rafnsson, W., Garg, D., Sabelfeld, A.: Progress-sensitive security for spark. In: International Symposium on Engineering Secure Software and Systems. pp. 20–37. Springer (2016)

18. Robling Denning, D.E.: Cryptography and data security. Addison-Wesley Longman Publishing Co. (1982)
19. Volpano, D., Smith, G.: Eliminating covert flows with minimum typings. In: Proceedings 10th Computer Security Foundations Workshop. pp. 156–168. IEEE (1997)
20. Volpano, D.M., Irvine, C.E., Smith, G.: A sound type system for secure flow analysis. Journal of Computer Security 4(2/3), 167–188 (1996)