Compile-time detection of information flow in sequential programs

Jean-Pierre Banâtre
Ciarán Bryce
Daniel Le Métayer
IRISA
Campus de Beaulieu
35042 Rennes Cedex, France
e-mail: *jpbanatre/bryce/lemetayer@irisa.fr*

Abstract. We give a formal definition of the notion of information flow for a simple guarded command language. We propose an axiomatisation of security properties based on this notion of information flow and we prove its soundness with respect to the operational semantics of the language. We then identify the sources of non determinism in proofs and we derive in successive steps an inference algorithm which is both sound and complete with respect to the inference system.

Keywords: formal verification, program analysis, verification tools, computer security, information flow.

1 Introduction

The context of the work described in this paper is the application of formal methods to the verification of information flow properties in programs. In contrast with most previous contributions in this area we put emphasis on the design of mechanical tools. Rather than considering a general (and undecidable) logic in which the development of proofs requires some interaction with the user, we start with a restricted language of properties which allows us to derive an automatic proof checker. The proof checking method is akin to the program analysis techniques used in modern optimising compilers [2]. Such a tool must satisfy two crucial properties in order to be useful for checking security properties:

- Its correctness must be established.
- It must be reasonably efficient.

These goals are achieved in several stages. We first provide a formal definition of the notion of information flow embodying the intuitive idea that information does not flow from a variable x to a variable y if variations in the original value of x cannot produce any variation in the final value of y. We define an *information flow logic* and we prove its correctness with respect to the operational semantics of the language. We identify the sources of non determinism in proofs which use

this logic and we successively refine the proof system into a correct and complete algorithm for information flow analysis. The techniques used to transform the original proof system into an algorithmic version are akin to methods used to get a syntax-directed version of type inference systems including weakening rules [10].

The rest of the paper is organised in the following way. Section 2 introduces our guarded command language with its operational semantics and our definition of information flow. We propose an information flow logic SS_1 and we state its correctness with respect to the semantics of the language. In section 3 we present more deterministic versions of the original system (SS_2 and SS_3). The basic idea is that SS_2 avoids the use of a specific weakening rule and SS_3 derives at each step the most precise property provable in SS_2 (or the conjunction of all the properties derivable in SS_2). We state the soundness and a form of completeness of the new system (with respect to the information flow logic SS_1). SS_3 still contains a source of non determinism in the rule for the repetitive command. We consequently propose a fourth system SS_4 in section 4 which can be seen as a property transformer. In section 5, we show that properties can be represented as graphs for an efficient implementation. The iteration itself can be replaced by a simple graph transformation. A property can be extracted from a graph using a path finding algorithm. Section 6 provides insights on the extension to more realistic language features including parallelism and pointer manipulation and section 7 reviews related work. Space considerations prevent us from providing details about the proofs here. The interested reader can find a complete treatment in [5].

2 An inference system for security properties

We consider a simple guarded command language whose syntax is defined as follows:

Program	::= Decl $\prec$; Decl $\succ$; Stmt	*program*
Decl	::= **var** v	*declarations*
Stmt	::= (p, Comm)	*statements*
Comm	::= v := E | **skip** | Stmt; Stmt | Alt | Rep	*commands*
Alt	::= [guard $\rightarrow$ Stmt$\prec$; $\square$ guard $\rightarrow$ Stmt$\succ$]	*alternative*
Rep	::= *[guard $\rightarrow$ Stmt$\prec$; $\square$ guard $\rightarrow$ Stmt$\succ$]	*repetitive*
guard	::= B	*guard*

where $\prec\succ$ stands for zero or more repetitions of the enclosed syntactical units, 'v' stands for a variable or a list of variables, 'E' for an integer expression and 'B' for a boolean expression. Commands are associated with program points p. All program points are assumed to be different and p_0 and p_x stand for respectively the entry point and the exit point of the program. We omit program points in the text of the programs but they are used to state certain properties. The alternative and repetitive commands consist of one or more guard branch pairs. A guard is a boolean expression. A guard is *passable* if it evaluates to true.

When an alternative command is executed, a branch whose guard is passable is chosen. If more than one guard is passable, then any one of the corresponding branches can be executed. If no guard is passable then the command fails and the program terminates. On each iteration of the repetitive command, a branch whose guard is passable is executed. If more than one guard is passable, then like for the alternative, any one of the branches is chosen. When no guard is passable, the command terminates and the program continues. The structural operational semantics of this language is defined in Figure 1. The rules are expressed in terms of rewritings of configurations. A configuration is either a pair $< S, \sigma >$, where S is a statement and σ a state, or a state σ. The latter is a terminal configuration.

$$< y := exp, \sigma > \rightarrow \sigma[val(exp, \sigma)/y]$$

$$< t[i] := exp, \sigma > \rightarrow \sigma[t[i \leftarrow val(exp, \sigma)]/t]$$

$$< \mathbf{skip}, \sigma > \rightarrow \sigma$$

$$\frac{< S_1, \sigma > \rightarrow < S_1', \sigma' >}{< S_1; S_2, \sigma > \rightarrow < S_1'; S_2, \sigma' >}$$

$$\frac{< S_1, \sigma > \rightarrow \sigma'}{< S_1; S_2, \sigma > \rightarrow < S_2, \sigma' >}$$

$$\frac{< S_1, \sigma > \rightarrow abort}{< S_1; S_2, \sigma > \rightarrow abort}$$

$$\frac{< C_i, \sigma > \rightarrow true}{< [C_1 \rightarrow S_1 \Box C_2 \rightarrow S_2 \Box ..... \Box C_n \rightarrow S_n], \sigma > \rightarrow < S_i, \sigma >}$$

$$\frac{< C_i, \sigma > \rightarrow abort}{< [C_1 \rightarrow S_1 \Box C_2 \rightarrow S_2 \Box ..... \Box C_n \rightarrow S_n], \sigma > \rightarrow abort}$$

$$\frac{\forall i. < C_i, \sigma > \rightarrow false}{< [C_1 \rightarrow S_1 \Box C_2 \rightarrow S_2 \Box ..... \Box C_n \rightarrow S_n], \sigma > \rightarrow abort}$$

$$\frac{< C_i, \sigma > \rightarrow true}{\begin{array}{c} < *[C_1 \rightarrow S_1 \Box C_2 \rightarrow S_2 \Box ..... \Box C_n \rightarrow S_n], \sigma > \rightarrow \\ < S_i; *[C_1 \rightarrow S_1 \Box C_2 \rightarrow S_2 \Box ..... \Box C_n \rightarrow S_n], \sigma > \end{array}}$$

$$\frac{< C_i, \sigma > \rightarrow abort}{< *[C_1 \rightarrow S_1 \Box C_2 \rightarrow S_2 \Box ..... \Box C_n \rightarrow S_n], \sigma > \rightarrow abort}$$

$$\frac{\forall i. < C_i, \sigma > \rightarrow false}{< *[C_1 \rightarrow S_1 \Box C_2 \rightarrow S_2 \Box ..... \Box C_n \rightarrow S_n], \sigma > \rightarrow \sigma}$$

Fig. 1. Operational semantics

Let us now turn to the problem of defining the information flow for this language. There are two classes of information flows in programs. An assignment command causes a **direct** flow of information from the variables appearing on the right hand side of the (:=) operator to the variable on the left hand side. This is because the information in each of the right hand side operands can influence value of the left hand side variable [6]. The information that was in the destination variable is lost.

Conditional commands introduce a new class of flows [8]. The fact that a command is conditionally executed signals information to an observer concerning the value of the command guard. Consider the following program segment. e is some expression:

$$
\begin{array}{l}
x := e; \\
a := 0; \\
b := 0; \\
[\ x = 0 \rightarrow a := 1 \\
\quad \Box\ x \neq 0 \rightarrow b := 1\ ]
\end{array}
$$

In this program segment, the values of **both** a and b after execution indicate whether x was zero or not. This is an example of an implicit flow [8] or what we more generally refer to as an **indirect** flow.

We note IF_p the set of indirect information flow variables at a particular program point p. IF_p can be defined syntactically as the set of variables occurring in embedding guards.

We need some way of representing the set of variables which may have flown to, or influenced, a variable v. We call this set the **security variable** of v, denoted $\overline{v}$. We define $\overline{IF_p}$ as:

$$\overline{IF_p} \ = \ \{x \mid x \in \overline{v} \ and \ v \in IF_p\}$$

Our inference system for the proof of information flow properties is described in Figure 2.

An array assignment $t[i] := e$ is treated as $t := exp(t, i, e)$. The last rule in Figure 2 is called the consequence rule or the *weakening rule*. We use the notation $\vdash_1 \{P\}\ S\ \{Q\}$ to denote the fact that $\{P\}\ S\ \{Q\}$ is provable in SS_1.

We define a correspondence relation between properties and the semantics of statements and we use it to state the correctness of the information flow logic of Figure 2.

Definition

$$
\begin{array}{l}
C(P,S) = \\
\qquad (P \Rightarrow x \notin \overline{y}) \quad \Rightarrow \\
\qquad\quad \forall \sigma, v. \text{ such that } < S, \sigma > \downarrow \ and < S, \sigma[v/x] > \downarrow \\
\qquad\quad\quad \{v' \mid < S, \sigma > \stackrel{*}{\rightarrow} \sigma',\ \sigma'(y) = v'\} = \\
\qquad\quad\quad \{v'' \mid < S, \sigma[v/x] > \stackrel{*}{\rightarrow} \sigma'',\ \sigma''(y) = v''\}
\end{array}
$$

Proposition 1 (correctness of SS_1).

$$\forall S, P. \ \textbf{if} \ \vdash_1 \{Init\} S \{P\} \ \textbf{then} \ C(P,S)$$

$$\{P[\overline{y} \leftarrow \bigcup_i \overline{x_i} \cup \overline{IF_p}]\}\ y := exp(x_1, x_2, ..., x_n)\ \{P\}$$

$$\{P\}\mathbf{skip}\{P\}$$

$$\frac{\{P\}S1\{Q\}, \{Q\}S2\{R\}}{\{P\}S1; S2\{R\}}$$

$$\frac{\forall i = 1..n \{P\}S_i\{Q\}}{\{P\}[C_1 \rightarrow S_1 \Box C_2 \rightarrow S_2 \Box ..... \Box C_n \rightarrow S_n]\{Q\}}$$

$$\frac{\forall i = 1..n \{P\}S_i\{P\}}{\{P\} * [C_1 \rightarrow S_1 \Box C_2 \rightarrow S_2 \Box ..... \Box C_n \rightarrow S_n]\{P\}}$$

$$\frac{P \Rightarrow P',\ \{P'\}\ S\ \{Q'\},\ Q' \Rightarrow Q}{\{P\}\ S\ \{Q\}}$$

Fig. 2. System SS_1

Init is defined as $\forall x, y\ x \neq y.\ x \notin \overline{y}$. It represents the standard (minimal) initial property. $S[v/x]$ is the same as S except that variable x is assigned value v. $< S, \sigma >\downarrow$ stands for $\exists \sigma' \neq abort.\ < S, \sigma > \stackrel{*}{\rightarrow} \sigma'$ which means that the program may terminate successfully. The above definition characterises our notion of information flow. If $P \Rightarrow x \notin \overline{y}$ holds, then the value of x before executing S cannot have any effect on the possible values possessed by y after the execution of S. In other words, no information can flow from x to y in S. The condition $< S, \sigma >\downarrow$ *and* $< S, \sigma[v/x] >\downarrow$ is required because the execution of S may terminate or not depending on the original value of x.

The correctness of SS_1 can be proven by induction on the structure of terms as a consequence of a more general property [5].

Let us now consider, as an example, a library decryption program. The program has three inputs and two outputs. The input consists of a string of encrypted text, or *cipher*, a key for decryption and a unit rate which the user is charged for each character decrypted. The variable *cipher* is an array of characters. A character is decrypted by applying it to an expression D with the *key* parameter. To save computing resources, some characters may not have been encrypted. The user pays twice the price for every encrypted character that goes through the decryption program. The boolean expression *encrypted()* determines if the character passed is encrypted or not. The outputs are the decrypted text, or *clear*, and the charge for the decryption. We assume that *clear* is output to the user and that *charge* is output to the library owner. To be usable, the user must trust the program not to secretly leak the clear text or the key to the library owner via the charges output. Such a leakage is termed a *covert channel* in [14]. The proof system described in Figure 2 allows us to prove the following

property:

$$\vdash_1 \{Init\}\ Library\ \{(clear \notin \overline{charge})\ and\ (key \notin \overline{charge})\}$$

that is, the charge output may not receive a flow of information from the *clear* variable or from the *key* input. We show in section 5 that this property can in fact be proven mechanically.

```
var: i, charge, key, unit;
array: clear, cipher;
cipher := ≺ message to be decrypted ≻;
unit := ≺ unit rate constant ≻;
charge := unit;
i := 0;
*[ cipher[i] ≠ null_constant →
        [ encrypted(cipher[i]) → clear[i] := D(cipher[i], key);
                                 charge := charge + 2*unit;
        □ not encrypted(cipher[i]) → clear[i] := cipher[i];
                                     charge := charge + unit;
        ];
        i := i + 1
]
```

Fig. 3. Library decryption program

3 A more deterministic system

We consider now the problem of mechanising the proof of security properties. As suggested above, the sort of properties we are interested in are of the form $x \notin \overline{y}$. The language of properties is:

$$P ::= x \notin \overline{y} \mid P_1 \wedge P_2$$

where $\wedge$ represents the logical "and" connective.

The system SS_1 presented in section 2 is not suggestive of an algorithm for several reasons:

- The relationship between the input and the output property of the rule for assignment is not one to one.
- The weakening rule can be applied at any time in a proof.

The combination of the weakening rule with the rule for the repetitive command in particular requires some insight. In general this amounts to guessing the appropriate invariant for the loop. There are two possible ways of proving a property of the form $\{P\}\ Prog\ \{Q\}$: one can either start with P and try to find a postcondition implying Q or start with Q and derive a precondition implied by P. These techniques are called respectively *forwards analysis* and *backwards analysis*. The method we present here for deriving security properties belongs to the forwards analysis category. Let us note however that the inference system SS_1 is not biased towards one technique or the other and we can apply the same idea to derive a backwards analysis. In order to reduce the amount of non determinism we first distribute the weakening rule over the remaining rules, getting system SS_2 presented in Figure 4.

$$\frac{P \Rightarrow P'[\overline{y} \leftarrow \bigcup_i \overline{x_i} \cup \overline{IF_p}],\quad P' \Rightarrow Q}{\{P\}\ y := exp(x_1, x_2, ..., x_n)\ \{Q\}}$$

$$\frac{P \Rightarrow P'}{\{P\}\mathbf{skip}\{P'\}}$$

$$\frac{P \Rightarrow P',\quad \{P'\}S1\{Q'\},\ Q' \Rightarrow Q",\quad \{Q"\}S2\{R'\},\quad R' \Rightarrow R}{\{P\}S1; S2\{R\}}$$

$$\frac{P \Rightarrow P',\quad \forall i = 1..n\{P'\}S_i\{Q'\},\quad Q' \Rightarrow Q}{\{P\}[C_1 \rightarrow S_1 \Box C_2 \rightarrow S_2 \Box ..... \Box C_n \rightarrow S_n]\{Q\}}$$

$$\frac{P \Rightarrow P',\quad \forall i = 1..n\{P'\}S_i\{P'\},\quad P' \Rightarrow Q}{\{P\} * [C_1 \rightarrow S_1 \Box C_2 \rightarrow S_2 \Box ..... \Box C_n \rightarrow S_n]\{Q\}}$$

Fig. 4. System SS_2

The soundness of SS_2 is obvious and its completeness follows from the transitivity of implication:

Proposition 2 (soundness and completeness of SS_2).

$$\forall S, P, Q.\quad \vdash_1 \{P\}\ S\ \{Q\} \textbf{ if and only if } \vdash_2 \{P\}\ S\ \{Q\}$$

This first transformation still yields a highly non deterministic proof procedure but it paves the way for the next refinement. Let us first note that the new system SS_2 is syntax directed. In order to derive an algorithm from SS_2 we want

to factor out all the possible proofs of a program into a single *most precise* proof. This proof should associate with any property P the greatest property Q (in the sense of set inclusion) such that $\vdash_2 \{P\}\ S\ \{Q\}$. This requirement allows us to get rid of most of the uses of $\Rightarrow$ in the rules (but not all of them) and imposes a new rule for the assignment command. The new system SS_3 is described in Figure 5.

$$\{R\}\ y := exp(x_1, x_2, ..., x_n)\ \{T_y(R)\}$$

$$\{P\}\mathbf{skip}\{P\}$$

$$\frac{\{P\}S1\{Q\}, \{Q\}S2\{R\}}{\{P\}S1; S2\{R\}}$$

$$\frac{\forall i = 1..n \{P\}S_i\{Q_i\}}{\{P\}[C_1 \rightarrow S_1 \square C_2 \rightarrow S_2 \square ..... \square C_n \rightarrow S_n]\{\bigsqcup_i Q_i\}}$$

$$\frac{P \Rightarrow P',\quad \forall i = 1..n \{P'\}S_i\{Q_i\},\quad \bigsqcup_i Q_i \Rightarrow P'}{\{P\} * [C_1 \rightarrow S_1 \square C_2 \rightarrow S_2 \square ..... \square C_n \rightarrow S_n]\{\bigsqcup_i Q_i\}}$$

with:

$$R^y = \bigwedge_{z \neq y} \{(x \notin \overline{z}) \mid R \Rightarrow (x \notin \overline{z})\}$$
$$T_y(R) = R^y \bigwedge_i \{(y_i \notin \overline{y}) \mid \forall j \in [1, \ldots, n].\ R \Rightarrow (y_i \notin \overline{x_j})\ and\ \forall v \in IF_p.\ R \Rightarrow (y_i \notin \overline{v})\}$$
$$\bigsqcup_i Q_i = \bigwedge\{(x \notin \overline{y}) \mid \forall i \in [1, \ldots, n], Q_i \Rightarrow (x \notin \overline{y})\}$$

Fig. 5. System SS_3

The intuition behind the new rule for the assignment command is that $T_y(R)$ represents the conjunction of all the properties $x \notin \overline{z}$ derivable from the input property R. R_y is the restriction of R to properties of the form $(x \notin \overline{z})$ with $z \neq y$. $\bigsqcup$ is the approximation in our language of properties of the logical "or " connective ($\vee$). It is expressed in terms of sets as an intersection. For instance:

$$((x \notin \overline{y}) \wedge (z \notin \overline{t})) \bigsqcup ((x \notin \overline{t}) \wedge (z \notin \overline{t})) = (z \notin \overline{t})$$

It is easy to see that

$$(Q_1 \vee Q_2) \Rightarrow (Q_1 \bigsqcup Q_2)$$

This approximation is required because the "or" connective does not belong to our language of properties. The language could be extended with $\vee$ but it makes the treatment more complex and it does not seem to allow the derivation of more useful information.

We cannot get rid of the implication in a straightforward way in the rule for the repetitive command because:

$$P \Rightarrow P' \quad and \quad \{P'\}S_i\{P'\}$$

does **not** imply

$$\{P\}S_i\{P\}$$

In order to prove a property of the repetitive command an appropriate invariant P' has to be discovered. We show in the next section how the maximal invariant can be computed iteratively.

The following properties state respectively the soundness and the completeness of SS_3 with respect to SS_2.

Proposition 3 (soundness of SS_3).

$$\forall S, P, Q.\ \textbf{if}\ \vdash_3 \{P\}\ S\ \{Q\}\ \textbf{then}\ \vdash_2 \{P\}\ S\ \{Q\}$$

Proposition 4 (completeness of SS_3).

$$\forall S, P, Q.\ \textbf{if}\ \vdash_2 \{P\}\ S\ \{Q\}\ \textbf{then}\ \exists Q'.\ \vdash_3 \{P\}\ S\ \{Q'\}\quad Q' \Rightarrow Q$$

Both properties are proved by induction on the structure of commands [5].

4 Mechanical analysis of the repetitive command

In order to be able to treat the repetitive statement mechanically we must be able to compute a property P' such that

$$P \Rightarrow P'$$

and

$$\forall i = 1..n\{P'\}S_i\{Q_i\} \quad and \quad \bigsqcup_i Q_i \Rightarrow P'$$

Furthermore it must be the greatest of these properties in order to retain completeness. We compute this property using an iterative technique akin to the method used for finding least fixed points in abstract interpretation [1]. Figure 6 presents SS_4 which is a refinement of SS_3 with an effective rule for the repetitive statement.

The following properties show that SS_4 is the expression, in the form of an inference system, of a terminating, correct and complete algorithm.

$$\{R\}\ y := exp(x_1, x_2, ..., x_n)\ \{T_y(R)\}$$

$$\{P\}\mathbf{skip}\{P\}$$

$$\frac{\{P\}S1\{Q\},\ \{Q\}S2\{R\}}{\{P\}S1; S2\{R\}}$$

$$\frac{\forall i = 1..n\{P\}S_i\{Q_i\}}{\{P\}[C_1 \rightarrow S_1 \Box C_2 \rightarrow S_2 \Box ..... \Box C_n \rightarrow S_n]\{\bigsqcup_i Q_i\}}$$

$$\frac{\begin{array}{c}\forall i = 1..n\{P^0\}S_i\{Q_i^0\},\quad Q^0 = \bigsqcup_i Q_i^0,\quad Q^0 \not\Rightarrow P^0,\quad P^1 = P^0 \bigsqcup Q^0 \\ \forall i = 1..n\{P^1\}S_i\{Q_i^1\},\quad Q^1 = \bigsqcup_i Q_i^1,\quad Q^1 \not\Rightarrow P^1,\quad P^2 = P^1 \bigsqcup Q^1 \\ \vdots \\ \forall i = 1..n\{P^{n-1}\}S_i\{Q_i^{n-1}\},\quad Q^{n-1} = \bigsqcup_i Q_i^{n-1},\quad Q^{n-1} \Rightarrow P^{n-1},\quad P^n = Q^{n-1}\end{array}}{\{P^0\} * [C_1 \rightarrow S_1 \Box C_2 \rightarrow S_2 \Box ..... \Box C_n \rightarrow S_n]\{P^n\}}$$

Fig. 6. System SS_4

Proposition 5 (termination of SS_4).

$$\forall S, P,\quad \exists Q. \vdash_4 \{P\}\ S\ \{Q\}\ \textit{and } Q \textit{ is unique}$$

Proposition 6 (soundness of SS_4).

$$\forall S, P, Q.\ \textbf{if}\ \vdash_4 \{P\}\ S\ \{Q\}\ \textbf{then}\ \vdash_3 \{P\}\ S\ \{Q\}$$

Proposition 7 (completeness of SS_4).

$$\forall S, P, Q.\ \textbf{if}\ \vdash_3 \{P\}\ S\ \{Q\}\ \textbf{then}\ \exists Q'.\ \vdash_4 \{P\}\ S\ \{Q'\}\quad Q' \Rightarrow Q$$

The three properties are proven by induction on the structure of commands [5].

5 Inference as transformations on graphs

A conjunctive property P can alternatively be represented as a set of pairs of variables:

$$\{(y, x) \mid P \Rightarrow x \notin \overline{y}\}$$

For instance $z \notin \overline{t} \wedge t \notin \overline{u}$ is represented as $\{(t, z), (u, t)\}$. We present in Figure 7 a new version of SS_4 expressed in the form of an algorithm T_5 taking as arguments a property P represented as a set and a program $Prog$ and returning the property Q such that $\vdash_4 \{P\}\ Prog\ \{Q\}$.

$$T_5(P,(y := exp(x_1,x_2,.....,x_n))) \;=\; T_y(P)$$

$$T_5(P,\mathbf{skip}) \;=\; P$$

$$\frac{T_5(P,S1) \;=\; Q,\;\; T_5(Q,S2) \;=\; R}{T_5(P,S1;S2) \;=\; R}$$

$$\frac{\forall i = 1..n \;\; T_5(P,S_i) \;=\; Q_i}{T_5(P,[C_1 \rightarrow S_1 \Box C_2 \rightarrow S_2 \Box .....\Box C_n \rightarrow S_n]) \;=\; \bigcap_i Q_i}$$

$$\frac{\begin{array}{c}\forall i = 1..n \;\; T_5(P^0,S_i) \;=\; Q_i^0,\;\; Q^0 \;=\; \bigcap_i Q_i^0,\;\; Q^0 \not\supset P^0,\;\; P^1 \;=\; P^0 \bigcap Q^0 \\ \hline \forall i = 1..n \;\; T_5(P^1,S_i) \;=\; Q_i^1,\;\; Q^1 \;=\; \bigcap_i Q_i^1,\;\; Q^1 \not\supset P^1,\;\; P^2 \;=\; P^1 \bigcap Q^1 \\ \hline \vdots \\ \hline \forall i = 1..n \;\; T_5(P^{n-1},S_i) \;=\; Q_i^{n-1},\;\; Q^{n-1} = \bigcap_i Q_i^{n-1},\;\; Q^{n-1} \supset P^{n-1},\;\; P^n = Q^{n-1}\end{array}}{T_5(P^0,*[C_1 \rightarrow S_1 \Box C_2 \rightarrow S_2 \Box .....\Box C_n \rightarrow S_n]) \;=\; P^n}$$

with:

$R^y = \{(z,x) \in R \mid z \neq y\}$
$T_y(R) \;=\; R^y \bigcup \{(y,y_i) \mid \forall\, j \in [1,\ldots,n].\, (x_j,y_i) \in R$
$\qquad\qquad\qquad\qquad\qquad\qquad and\; \forall\, v \in IF_p.\, (v,y_i) \in R\}$

Fig. 7. System SS_5

The proof of the equivalence of SS_4 and SS_5 is obvious. In terms of sets, $\bigsqcup$ is implemented as set intersection($\bigcap$) and $\Rightarrow$ corresponds to the superset relation ($\supset$).

Proposition 8 (correctness and completeness of T_5).

$$\forall S,P,Q. \quad \vdash_4 \{P\}\, S\, \{Q\} \;\textbf{if and only if}\; T_5(P,S) \;=\; Q$$

The representation of properties as sets of pairs leads to a very expensive implementation of the rule for assignment involving a quadratic number of tests in sets R and IF_p. We propose instead to represent properties as accessibility graphs. We consider directed graphs defined as pairs of a set of nodes and a set of arcs:

$$\begin{array}{rcl} G & ::= & (N,A) \\ N & ::= & \{n\} \\ n & ::= & V^p \\ A & ::= & \{a\} \\ a & ::= & (n_1,n_2) \end{array}$$

V^p is the set of the variables of the program subscripted by program points. The property represented by a graph G at program point p is given by the function H defined as follows:

$$H(p, G) = \{(y, x) \mid Nopath(G, x^0, y^p)\}$$

$Nopath(G, x^0, y^p)$ returns $True$ is there is no path from node x^0 to node y^p in the graph G. We have now to show how the operations on properties required by T_5 are implemented in terms of graphs. Since the set of nodes of the graphs manipulated by our algorithm is constant it is convenient to introduce the following notation:

$$\text{if } G = (N, A) \text{ then } G + A' = (N, A \cup A')$$

Furthermore the symbol + is overloaded to operate on two graphs (no ambiguity can arise from this overloading):

$$\text{if } G = (N, A) \text{ and } G' = (N, A') \text{ then } G + G' = (N, A \cup A')$$

The final version of our algorithm is described in Figure 8 (variables v are implicitly quantified over the whole set of variables of the program).

$$T_6(G, p, (q, (y := exp(x_1, x_2, .., x_n)))) = G + \{(x_i^p, y^q) \mid i = 1..n\} + \{(z^p, z^q) \mid z \neq y\} + \{(z^r, y^q) \mid z_r \in IF_q\}$$

$$T_6(G, p(q, \textbf{skip})) = G$$

$$\frac{T_6(G, p, (q_1, S1)) = G_1, \quad T_6(G_1, q_1, (q_2, S2)) = G_2}{T_6(G, p, (q, (q_1, S1); (q_2, S2))) = G_2 + \{(v^{q_2}, v^q)\}}$$

$$\frac{\forall i = 1..n \quad T_6(G, p, (q_i, S_i)) = G_i}{T_6(G, p, (q, [C_1 \to (q_1, S_1) \square C_2 \to (q_2, S_2) \square ... \square C_n \to (q_n, S_n)])) = +_i G_i + \{(v^{q_i}, v^q)\}}$$

$$\frac{\forall i = 1..n \quad T_6(G^0, p, (q_i, S_i)) = G_i^0, \quad G^1 = +_i G_i^0 + \{(v^{q_i}, v^q)\} + \{(v^q, v^p)\}}{T_6(G^0, p, (q, *[C_1 \to (q_1, S_1) \square C_2 \to (q_2, S_2) \square ..... \square C_n \to (q_n, S_n)])) = G^1}$$

Fig. 8. System SS_6

T_6 takes three arguments: a graph G, a program point p and a statement $S \in Stmt$. The program point characterises a statement "preceding" the current statement in the (execution of) the program. The program is analysed with the input program point p^0 as argument. Program points are made explicit in the statements because they play a crucial rôle at this stage. The rule for assignment

can be explained as follows. An arc is added to the graph from each occurrence of variables x_i at the preceding program point p to y at the current program point q, and from each variable in the set of indirect flow to y. Other variables are not modified and an arc is added from their occurrence at point p to their occurrence at point q. In the rules for the alternative command and the repetitive command the operation $+$ is used to implement $\bigcap$. This comes from the fact that graphs record accessibility when sets contain negative information of the form $x \notin \overline{y}$.

The correctness of this last algorithm is stated as follows:

Proposition 9 (correctness of T_6).

$$\forall S, P, Q, G, p, q.\ H(p, G) = P \ \textit{and}\ T_5(P, S) = Q \ \Rightarrow$$

$$H(q, T_6(G, p, (q, S))) = Q$$

This property can be proved by induction on the structure of expressions [5].

It should be clear that some straightforward optimisations can be applied to this algorithm. First it is not necessary to keep one occurrence of variable per program point in the graph. As can be noticed from the rule for assignment, most of these variables would just receive one arc from the previous occurrence of the variable. All these useless arcs can be short-circuited and the only nodes kept into the graph are occurrences of x^p where p is an assignment to x or an alternative (or repetitive) statement with several assigments to x. Also a naïve implementation of the rules for the alternative and the repetitive statements would lead to duplications of the graph. The monotonicity of T_6 allows us to get rid of this duplication. Instead the graph can be constructed iteratively as follows:

$$\frac{G_0 = G,\ \ \forall i = 1..n\ \ T_6(G_{i-1}, p, (q_i, S_i)) \ = \ G_i}{T_6(G, p, (q, [C_1 \rightarrow (q_1, S_1) \square C_2 \rightarrow (q_2, S_2) \square ..... \square C_n \rightarrow (q_n, S_n)])) \ = \ G_n + \{(v^{q_n}, v^q)\}}$$

Let us now return to the library decryption program to illustrate the algorithm. Figure 9 is a new presentation of the program making some program points explicit (we do not include all of them for the sake of readability).
Figure 10 presents the main steps of the application of T_6 to this program. We note P_i the command associated with p_i and we consider only the arc component of the graph. We avoid the introduction of useless nodes and arcs as described above. As a consequence, only 14 nodes are necessary for this program. Figure 11 shows the graph returned by the algorithm. Applying the *Nopath* function to this graph, we can derive the property mentioned in section 2 (p_4 is the exit program point for *charge*):

$$(\textit{clear} \notin \overline{\textit{charge}})\ \textit{and}\ (\textit{key} \notin \overline{\textit{charge}})$$

```
var: i, charge, key, unit;
array: clear, cipher;
cipher := ≺ message to be decrypted ≻;
unit := ≺ unit rate constant ≻;
(p1,charge := unit;
i := 0);
(p2,*[ cipher[i] ≠ null_constant →
        (p3,(p4,[ encrypted(cipher[i]) → (p5,(p6,clear[i] := D(cipher[i], key));
                                                (p7,charge := charge + 2*unit));
                □ not encrypted(cipher[i]) → (p8,(p9,clear[i] := cipher[i]);
                                                (p10,charge := charge + unit));
                ]);
                (p11,i := i + 1))
])
```

Fig. 9. Library decryption program

$$
\begin{array}{lll}
T_6(\emptyset, p_0, P_1) = G_1 & G_1 & = \{(unit^0, charge^1)\} \\
T_6(G_1, p_1, P_6) = G_2 & G_2 & = G_1 + \{(cipher^0, clear^6), (key^0, clear^6), (i^1, clear^6), \\
 & & \quad (clear^0, clear^6)\} \\
T_6(G_2, p_6, P_7) = G_3 & G_3 & = G_2 + \{(charge^1, charge^7), (unit^0, charge^7), \\
 & & \quad (i^1, charge^7), (cipher^0, charge^7)\} \\
T_6(G_3, p_1, P_9) = G_4 & G_4 & = G_3 + \{(cipher^0, clear^9), (i^1, clear^9), (clear^0, clear^9)\}\} \\
T_6(G_4, p_9, P_{10}) = G_5 & G_5 & = G_4 + \{(charge^1, charge^{10}), (unit^0, charge^{10}), \\
 & & \quad (i^1, charge^{10}), (cipher^0, charge^{10})\} \\
T_6(G_5, p_1, P_4) = G_6 & G_6 & = G_5 + \{(charge^7, charge^4), (charge^{10}, charge^4), \\
 & & \quad (clear^6, clear^4), (clear^9, clear^4)\} \\
T_6(G_6, p_1, P_2) = G_7 & G_7 & = G_6 + \{(charge^4, charge^1), (clear^4, clear^0)\}
\end{array}
$$

Fig. 10. Analysis of the library decryption program

6 Extensions

There are three main directions in which we plan to extend this work in order to be able to cope with more realistic languages:

- The introduction of pointer manipulation operators.
- The treatment of less structured control flow.
- The extension to a parallel language.

We consider each in turn.

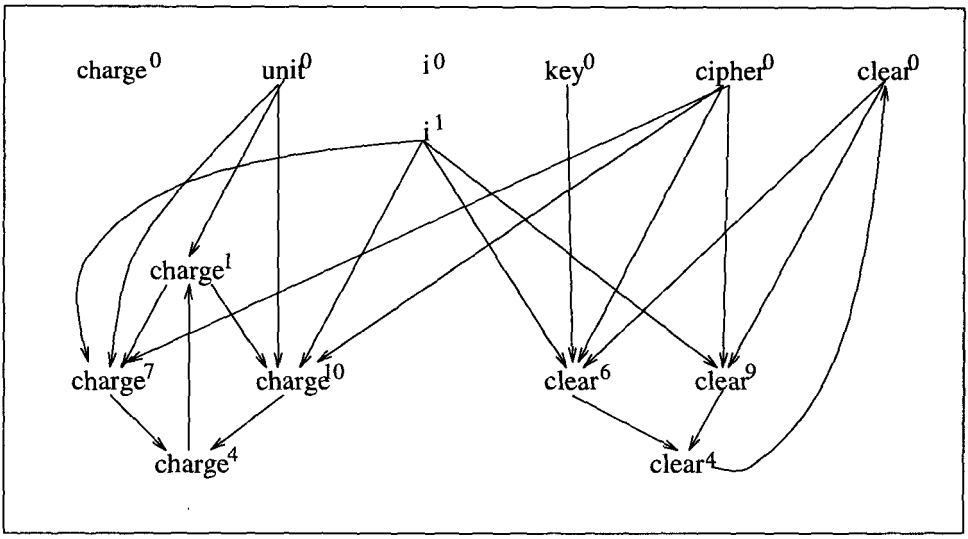

Fig. 11. Result of the analysis of the library decryption program

The addition of general pointers in a language complicates most program analyses because it introduces the well-known problem of *aliasing*. Aliasing occurs during program execution when two or more names exist for the same location [15]. Let us take a small example in the context of information flow to illustrate the problem.

$$\begin{array}{l} \textbf{int } i,\ j,\ k,\ *p \\ p := \&i; \\ i := j; \\ k := *p; \end{array}$$

The variable p is assigned the location of i. As a consequence, an information flow from i to p must be accounted for. Furthermore the assignment of j to i introduces a new (and hidden) flow of information between j and p and the assignment of $*p$ to k creates a flow from i to k. The semantics of the language and the definition of information flow (section 2) must be adapted to take into account the fact that a variable can have access to the value (or part of) of another variable through dereferences. As far as the proof checking algorithm is concerned, the first solution is to complement the method presented in this paper with a pointer aliasing analysis. Various techniques have been proposed in the literature to tackle this problem [15, 16]. These techniques are more or less accurate (and expensive) depending on the level of indirection which is considered. The rule for assignment can be enhanced to take this new form of flow into account:

$$T_6'(G, p, (q,\ (y := exp(x_1, x_2, ..., x_n)))) = G + \{(x_i^p, y^q) \mid i = 1..n\} + \{(z^p, z^q) \mid z \neq y\}$$
$$+ \{(z^r, y^q) \mid z_r \in IF_q\} + \{(y^q, z^t) \mid Alias(y^q, c.z^t)\}$$

where c is an access chain (sequence of dereferences).

A more ambitious research direction would be to integrate both analysers into a single, more efficient, algorithm. A possible solution is to consider *object names* rather than simple variables in the information flow analysis. Following [16], object names can be defined as follows:

$$object\_name \;=\; variable \mid *.object\_name \mid object\_name.field\_of\_structure$$

The introduction of less structured sequential control flow does not introduce deep technical problems into our analysis. It may however make the analyser more expensive (the same situtation occurs in traditional data flow analysis). For instance we can deal with explicit *goto* commands by adding new assignments at the join nodes of the control flow graph (very much like the ϕ-functions in SSA forms [7]). Such assignments are already implicit in the rules for the alternative and repetitive commands (see Figure 8 for instance).

The need for ensuring security properties becomes especially crucial in the context of distributed systems. We are currently studying the generalisation of our work for a full version of CSP [11]. In CSP, communication commands may occur in guards and in statements. The notion of indirect flow has to be extended to take such communications into account. The semantics of CSP introduces two main technical difficulties for a correct treatment of control flow:

- Indirect control flow can occur even in the absence of rendez-vous (when such a rendez-vous would have been made possible by a different execution of a guarded command).
- The non termination of a process can influence the values of the variables of the processes it might have communicated with.

As an example of how indirect flows can occur in the absence of a rendez-vous, consider the following program segment. Suppose that y of process $P1$ is either 1 or 0. Whatever, the value of y, at the end of process $P1$, x will equal y. The reason for this is that, if $y = 0$ in process P1, then P1 passes the value 1 to b of process P2 which then passes 0 back to x. Conversely, if y is 1 in P1, then P1 signals 0 to process P3 which signals 1 to P2's b which in turn passes this value back to x.

```
[
  P1::                          P2::                          P3::
  [var x,y;                     [var a,b;                     [var s ;
  y := e();                     [ P1 ? b → b:=b-1 □           P1 ? s;
  [ y=0 → P2 ! 1 □                P3 ? b → skip ]             P2 ! 1
   y≠0 → skip ]                 a := b;                       ]
  P3 ! 0;                       P1 ! a
  P2 ? x                        ] ||
  ] ||
]
```

Our solution consists of associating each program point p_i with a *control flow variable* c_i containing all the variables which may influence the fact that

the execution of the program reaches that point. When a communication occurs between $p_1 : P_2 \,!\, v$ and $p_2 : P_1 \,?\, x$, the control flow c_1 at point p_1 is added to the security variable $\overline{x}$. Furthermore both control flows c_1 and c_2 become $c_1 \cup c_2$. As far as algorithmic aspects are concerned, communications introduce a new source of non determinism in the proof. The traditional technique consists in carrying out the proof of each process independently before checking a *cooperation condition* on the individual rules. The first phase places little constraints on communication commands and appropriate properties have to be guessed in order to derive proofs that satisfy the cooperation conditions. Our graph algorithm can be extended in a natural way to simulate this reasoning. The set of nodes includes control flow variables and the required arcs are added between matching communication commands. The important property allowing us to retain the simplicity of the algorithm described here is the fact that we derive for each point of the program the strongest property provable at this point. As a consequence the graph can still be built incrementally avoiding the need for an iterative process.

7 Related work

Language based information flow control mechanisms have traditionally used *security levels* [8, 3]. Each variable is assigned a level denoting the sensitivity of the information it contains. After an operation, the level of the variable which received the information flow must be no less than the level of the flow source variables. However, the security level approach severely restricts the range of policies that one might like to support. A flow mechanism should log the variables that have flown to each variable rather than the level of the data. Jones and Lipton's *surveillance set* mechanism [12] is in this spirit and has some similarities with the mechanism proposed here.

In [18], McLean describes a unified framework for showing that a software module specification is *non-interfering* and that the module code satisfies this specification. Non-interference is a security property which states that a user's output cannot be affected by the input of any user with a higher security level. McLean's approach is based on the trace method for software module specification [17]. This method defines a module's semantics as the set of legal module traces (sequences of module procedure calls), the values returned by the traces terminating in a function call and a trace equivalence. Non-interference can be proved from the module's trace semantics. The author then defines a simple sequential procedural based programming language and gives the semantics of the language in trace form. This method is attractive because it allows the non-interference proof to be conducted at the abstract level of functional specifications. Program security is then established as a consequence of the functional correctness. In contrast with our approach however, no attempt is made to conduct proofs in a mechanical (or even systematic) way.

The main contribution of this paper is to provide a formally based and effective tool for checking security properties of sequential programs. To our knowl-

edge there have been surprisingly few attempts to achieve these goals so far. Most of the approaches described in the literature either lead to manual verification techniques [3] or rely on informal correctness proofs [9]. The closest work in the spirit of the contribution presented here is [19]. They derive a flow control algorithm as an abstract interpretation of the denotational semantics of the programming language. The programmer associates each variable with a security class (such as *unclassified, classified, secret, ...*). Security classes correspond to particular abstract semantics domains forming a lattice of properties and the analysis computes abstract values to check the security constraints. In contrast with this approach, we do not require security classes to be associated with variables but we check that the value of one particular variable cannot flow into another variable. We have shown in [4] that this approach provides more flexibility in the choice of a particular security policy. Our algorithm could in fact be applied to synthesise the weakest constraints on the security classes of the variables of an unannotated program. These two options can be compared with the choice between explicit typing and type synthesis in strongly typed programming languages.

References

1. Abramsky (S.) and Hankin (C. L.), "Abstract interpretation of declarative languages", Ellis Horwood, 1987.
2. Aho (A. V.), Sethi (R.) and Ullman (J. D.), "Compilers: Principles, Techniques and Tools", Addison Wesley, Reading, Mass, 1986.
3. Andrews (G.R.), Reitman (R.P.), "An Axiomatic Approach to Information Flow in Programs", in *ACM Transactions on Programming Languages and Systems*, volume 2 (1), January 1980, pages 504-513.
4. Banâtre (J.-P.) and C. Bryce, (C.), "A security proof system for networks of communicating processes", Irisa research report, no 744, June 1993.
5. Banâtre (J.-P.) and C. Bryce, (C.), and Le Métayer (D.), "Mechanical proof of security properties", Irisa research report, no 825, May 1994.
6. Cohen (E.), "Information Transmission in Computational Systems", in *Proceedings ACM Symposium on Operating System Principles*, 1977, pages 133-139.
7. Cytron (R.), Ferrante (J.), Rosen (B. K.) and Wegman (M. N.), "Efficiently computing Static Single Assignment form and the control dependence graph", in *ACM Transactions on Programming Languages and Systems*, Vol. 13, No 4, October 1991, pages 451-490.
8. Denning (D.E.), *Secure Information Flow in Computer Systems*, Phd Thesis, Purdue University, May 1975.
9. Denning (D.E.), Denning (P.J.), "Certification of Programs for Secure Information Flow", in *Communications of the ACM*, volume 20 (7), July 1977, pages 504-513.
10. Hankin (C. L.) and Le Métayer (D.), "Deriving Algorithms from Type Inference Systems: Application to Strictness Analysis", in *Proceedings ACM POPL*, 1994, pages 202-212.
11. Hoare (C.A.R.), *Communicating Sequential Processes*, Prentice-Hall London, 1985.

12. Jones (A.), Lipton (R.), "The Enforcement of Security Policies for Computations", in *Proceedings of the* 5^{th} *Symposium on Operating System Principles*, November 1975, pages 197-206.
13. Kennedy K. W., "A Survey of Data Flow Analysis Techniques", in *Program Flow Analysis*, S. S. Muchnik and N. D. Jones, Eds, Prentice-Hall, Englewood Cliffs, NJ, 1981.
14. Lampson (B.), "A note on the Confinement Problem", in *Communications of the ACM*, volume 16 (10), October 1973, pages 613-615.
15. Landi (W.) and Ryder (B. G.), "Pointer-induced aliasing: a problem classification", in *Proceedings ACM POPL*, 1991, pages 93-103.
16. Landi (W.) and Ryder (B. G.), "A safe approximate algorithm for interprocedural pointer aliasing", in *Proceedings ACM Programming Language Design and Implementation*, 1992, pages 235-248.
17. McLean (J.), "A Formal Method for the Abstract Specification of Software", in *Journal of the ACM*, 31, July 1984, pages 600-627.
18. McLean (J.), "Proving Non-interference and Functional Correctness Using Traces", in *Journal of Computer Security*, 1(1), Spring 1992, pages 37-57.
19. Mizuno (M.), Schmidt (D.), "A Security Control Flow Control Algorithm and Its Denotational Semantics Correctness Proof", *Journal on the Formal Aspects of Computing*, 4 (6A), november 1992, pages 722-754.