Modeling Secure Information Flow with Boolean Functions

Samir Genaim*, Roberto Giacobazzi, and Isabella Mastroeni

Dipartimento di Informatica
Università di Verona
Strada Le Grazie, I-37134 Verona, Italy
Fax: +39-045-8027068
{genaim,giaco,mastroeni}@sci.univr.it

Abstract. In this paper we describe two uses of Boolean functions in the context of secure information flow analysis. The first contribution concerns with modeling information flow with Boolean functions, which leads to an accurate information flow analysis that captures dependencies between possible flows. These dependencies are useful for debugging; refining the notion of secure information flow; and achieving efficient implementation using sophisticated data structures like Binary Decision Diagrams. The second contribution concerns with analyzing dynamic security policies. We describe how to construct a Boolean function, such that its models describe possible non-interference sets of program variables. This can be used to enforce security classes dynamically, rather than re-analyzing the program.

1 Introduction

A program is secure, if the information it contains can only be manipulated in a way authorized by its security policy. There are several security policies defined in the literature [9]. The security policy we are concerned with in this paper is *information flow policy*. An information flow policy can be defined as a complete lattice of security classes, where information is allowed to flow from variables of a specific security class, to variables of higher security classes [5]. This is often called non-interference secrecy. Static analysis techniques have been used to check if programs meet their security policies. These techniques range from *data/control*-flow [2, 4, 10, 13, 11] to type-inference [15, 16].

The first contribution of this paper is, modeling information flow with Boolean functions, which leads to an accurate information flow analysis, from which we can observe dependencies between information flows. For example, one can observe that in any execution information may flow from y to x or z, but not to both, or, when x flows to y then z flows to w. The basic idea is that for each program variable x we create two Boolean variables x_i and x_o, which indicate respectively if the input and output values of x may contain secrets.

* Supported by Marie Currie Fellowship number HPMF-CT-2002-01848

Using these Boolean variables, the information flow of "x:=x+y" is captured by $(x_o \leftrightarrow x_i \vee y_i) \wedge (y_o \leftrightarrow y_i)$, which means that the output value of x depends on the input values of both x and y, while the output value of y depends only on the input value of y. Checking non-interference in this context, is equivalent to checking satisfiability of a corresponding Boolean function.

From the practical side, these dependencies between information flows are useful for: (1) refining of security policies, e.g. allowing information to flow to limited amount of variables; (2) debugging of programs that violate their security policies; and (3) using sophisticated data structures in the implementation, such as Binary Decision Diagrams [3], which are known to be very efficient. From the theoretical side, they raise the issue of systematic refinement [8] of domains for information flow to capture different degrees of dependencies.

The second contribution concerns the analysis of dynamic security policies. For a given program, we describe how to construct a Boolean function (not like those of the first contribution), such that its models describe possible non-interference divisions of the program variables to *private* and *public* classes. When a variable is dynamically binded to a security class, this Boolean function can be infer the security classes of other variables, rather than re-analyzing the program.

In the related works section, we also discuss an interesting link between information flow analysis and dependency analyses of logic programs, such as groundness analysis [1]. In these analyses, Boolean function where intensively used to capture dependencies between program variables.

The structure of the paper is as follows: In Section 2 we present the idea by simple examples; In Section 3 we give notation to be used during the paper, as well as, the necessary background on Boolean functions; In Section 4 we describe the analysis and its use for verifying non-interference in the context of static and dynamic security policies. In Section 5 we overview the related works and give directions for future research; and we conclude in Section 6.

2 The Idea by Examples

Consider the single statement $C \equiv (x := y + z)$ and assume the set of program variables is $\{x, y, z, w\}$. When executing C, the secret information flow behavior is: (1) secrets may *flow* from y and z to x; and (2) the security properties of y, z and w remain the same, since they are not updated. Now let x_i, y_i, z_i and w_i (x_o, y_o, z_o and w_o) be Boolean variables, that indicate, if the corresponding program variables *may* contain secret information before (after) executing C. Using these Boolean variables, the above secret information flow behavior can be expressed as follows:

$$\varphi = \underbrace{[x_o \leftrightarrow (y_i \vee z_i)]}_{(1)} \wedge \underbrace{[(z_o \leftrightarrow z_i) \wedge (y_o \leftrightarrow y_i) \wedge (w_o \leftrightarrow w_i)]}_{(2)}$$

The models of φ, i.e. the assignments for which φ is satisfiable, describe all possible secret information flow behaviors when executing C. Given a model m

of φ, if $v_i \in m$ ($v_o \in m$) then the value of the program variable v *may* contain secret information before (after) the execution of C.

Consider for example the model $m = \{y_i, x_o, y_o\}$ of φ, it describes a situation where before executing C, only y *may* contain secret information (since $y_i \in m$), while after the execution both x and y *may* contain secret information (since $x_o, y_o \in m$). Similarly, the model $\{z_i, x_o, z_o\}$ describe the *possibility* of secret information flow from z to x. Note also that $m' = \{y_i, y_o\}$ ***is not a model*** of φ, because executing C *may* cause a flow from y to x, and this is not described by m' (since $x_o \notin m'$).

Let us consider another kind of information flow which stems from guards of conditional statements. For example, in the following statement:

$$C_1 \equiv \text{if } (w = 0) \text{ then } x := y \text{ else } z := y.$$

The information flow behavior of C_1 consists of: (1) *explicit* information flow that stems from the "*then*" or the "*else*" branches; and (2) *implicit* flow from the guard's variables, i.e. w, to all variables that *might* be updated during the executions, i.e. x and z, because by watching the values of x and z we *may* learn whether $w = 0$ or $w \neq 0$. This behavior is expressed by the following Boolean functions:

$$\varphi_3 = \underbrace{(\varphi_1 \vee \varphi_2)}_{(1)} \wedge \underbrace{(x'_o \leftrightarrow (x_o \vee w_i)) \wedge (z'_o \leftrightarrow (z_o \vee w_i))}_{(2)}$$

where φ_1 and φ_2 are respectively the information flow behaviors of the "*then*" and "*else*" branches. The idea is to define new output variables x'_o and z'_o, that consider the information from x_o and z_o (of $\varphi_1 \vee \varphi_2$) as well as from w_i.

In general for a given program P, the analysis infers a Boolean function φ such that its models describe all possible (input/output) information flows. This function can be used to verify non-interference as follows: Given that the set of *private* variables is V_H and the set of *public* variables is V_L, we can claim non-interference if $[(\bigwedge_{x \in V_L} \neg x_i) \wedge \varphi \wedge (\bigvee_{x \in V_L} x_o)]$ is not satisfiable, which means that it cannot be the case that the input values of all *public* variables do not contain secrets, and, the output values of some of them contain secrets.

3 Preliminaries

3.1 The Programming Language

The analysis in this paper is developed for the following *while* programming language [17]:

$\mathrm{S} \equiv C^{\ell}$
$\mathrm{C} \equiv x := e \mid \text{while } b \text{ do} \mid \text{if } b \text{ then } S_1 \text{ else } S_2 \mid skip \mid S_1; S_2$

Here e stands for an arithmetic expression, b for a Boolean expression and ℓ for a label name. We assume all labels are different. For more information about

this language, in particular the use of labels see [6]. A program in the *while* language is denoted by the letter P, $vars(X)$ denotes the set of variables in a syntactic object X (expression, program, etc.), $V_P = vars(P)$, and V_S denotes the program variables where a statement S occur (not only $vars(S)$). When it is clear from the context we write $v \in X$ instead of $v \in vars(X)$.

3.2 Secure Information Flow

For a given statement S, information flow from a variable x to a variable y in S, denoted by $x \overset{S}{\rightsquigarrow} y$, means that by observing the final value of y we may gain knowledge about the initial value of x. Following [5], we divide information flows into two classes: *direct* and *indirect*. The *direct* flows are: (1) *Explicit*, which stem from direct assignments, e.g. in the statement $x := y + z$ there is a flow of information from y and z to x; and (2) *Implicit*, which stem from guards of conditional statements, e.g. in the statement "*if* $(x > 0)$ *then* $y := w$ *else* $y := z$" there is a flow of information from x to y, in addition to the *explicit* flow from w and z to y. The *indirect* flows are the transitive ones, i.e. $x \overset{S}{\rightsquigarrow} y$ followed by $y \overset{S}{\rightsquigarrow} z$ implies $x \overset{S}{\rightsquigarrow} z$. In this paper we do not address information flow that stems from covert channels, e.g. termination, timing, etc.

Definition 1 (Flow pairs). *The pair* $\langle V_I, V_O \rangle$*, where* $V_I, V_O \subseteq V_S$*, is called a flow pair for* S*, if* $(\forall x{\in}V_I.\ \exists y{\in}V_O.\ x \overset{S}{\rightsquigarrow} y) \wedge (\forall y{\in}V_O.\ \exists x{\in}V_I.\ x \overset{S}{\rightsquigarrow} y)$*. The set of all flow pairs for* S *is denoted by* $\mathcal{F}_S$*.* □

Another interpretation for a flow pair $\langle V_I, V_O \rangle$ could be: if V_I is the set of all variables the input values of which may have secrets, then V_O is the set of all variables the output values of which may have secrets. We use these two interpretations interchangeably.

Example 1. Let $S \equiv x := y + z$ then $\mathcal{F}_S = \{\ \langle \emptyset, \emptyset \rangle,\ \langle \{x\}, \emptyset \rangle,\ \langle \{y\}, \{y, x\} \rangle,\ \langle \{z\}, \{x, z\} \rangle,\ \langle \{x, y, z\}, \{x, y, z\} \rangle\ \}$. □

Information flow analysis was used to ensure secure information flow according to a given security policy [12]. A security policy is defined by a complete lattice $\langle SC, \prec \rangle$, where SC is a set of security classes order by $\prec$, and each program variable is associated with one security class, denoted $type(x)$. Secure information flow is guaranteed if: $x \rightsquigarrow y$ implies $type(x) \prec type(y)$. This is known as non-interference secrecy [9, 12]. For the sake of simplicity, we consider the case of only two security classes, *high* and *low*. We denote by V_H and V_L the sets of variables of type *high* and *low* respectively, and when it is clear from the context h stands for a *high* variable and l for a *low* variable.

3.3 Boolean Functions

Let $B = \{true, false\}$. A Boolean function on $V = \{x_1, \ldots, x_n\}$ is a function $\varphi : B^n \to B$. An *interpretation* $\mu : V \to B$ is an assignment of truth values to the variables in V. An interpretation μ is a *model* for φ, denoted $\mu \models \varphi$, if

$\varphi(\mu(x_1), \ldots, \mu(x_n)) = \mathit{true}$. We write an interpretation as the set of variables which are assigned to the value *true*. The set of models of φ is thus viewed as a set of sets of variables defined by $[\![\varphi]\!]_V = \left\{ \{x \in V \mid \mu(x) = \mathit{true}\} \,\middle|\, \mu \models \varphi \right\}$. Much of the time we will omit the subscript V as it will be clear from the context. We write $\varphi_1 \models \varphi_2$ for $\forall m \in [\![\varphi_1]\!].\ m \models \varphi_2$. Let φ be a Boolean function on V, the existential quantification $\exists x.\varphi$ is defined as $\varphi[x \mapsto \mathit{true}] \vee \varphi[x \mapsto \mathit{false}]$, and, the existential quantification of a set of variable $\{x_1, \ldots, x_n\}$ is defined as $\exists x_1.(\exists x_2. \ldots (\exists x_n.\varphi))$. Also, we write $\bar{\exists} X.\varphi$ instead of $\exists\, V \backslash X.\varphi$.

4 Information Flow Dependencies Analysis

In this section, we formalize the use of Boolean functions to model *information flow dependencies*, describe a static analysis to infer these functions, and, describe its use for verifying non-interference for static and dynamic security policies.

4.1 Information Flow Dependencies

For a given statement S we define a Boolean function φ_S, from which all possible information flows can be observed. We call this function *information flow dependencies* of S (or only *dependencies* when it is clear from the context). This function is constructed from the set of flow pairs $\mathcal{F}_S$ as follows.

Definition 2 (information flow dependencies). *The information flow* dependencies *of a statement S, denoted φ_S, is a Boolean function over the variables $\left\{ v_i, v_o \,\middle|\, v \in V_S \right\}$, defined by its models as: $[\![\varphi_S]\!] = \left\{ m(V_I, V_O) \,\middle|\, \langle V_I, V_O \rangle \in \mathcal{F}_S \right\}$ where $m(V_I, V_O) = \left\{ v_i \,\middle|\, v \in V_I \right\} \cup \left\{ v_o \,\middle|\, v \in V_O \right\}$.* □

In the above definition, the Boolean variables v_i and v_o indicate if v may contain secret before and after executing S respectively.

Example 2. Consider again $S \equiv x := y + z$ from Example 1, and let $V_S = \mathit{vars}(S) = \{x, y, z\}$, then using the set of flow pairs $\mathcal{F}_S$, the *information flow dependencies* φ_S is defined by its models as follows:

$$[\![\varphi_S]\!] = \left\{ \begin{array}{l} \emptyset, \{y_i, x_o, y_o\}, \{z_i, x_o, z_o\}, \{y_i, z_i, y_o, z_o, x_o\}, \{x_i\}, \\ \{x_i, y_i, x_o, y_o\}, \{x_i, z_i, x_o, z_o\}, \{x_i, y_i, z_i, y_o, z_o, x_o\} \end{array} \right\}$$

For example, the model $\{y_i, x_o, y_o\}$ describes a scenario where only the input value of y contains secrets at, and both output values of x and y contain secrets. Note that $\varphi_S \equiv (x_o \leftrightarrow y_i \vee z_i) \wedge (y_o \leftrightarrow y_i) \wedge (z_o \leftrightarrow z_i)$. □

Since secrets introduced only at input, *information flow dependencies* φ_S satisfies: (1) $\emptyset \in [\![\varphi_S]\!]$; and (2) any $m \neq \emptyset \in [\![\varphi_S]\!]$ must include at least one v_i.

Definition 3 (The domain SD_V). *Given a set of program variables V, a Boolean function φ over the variables $\left\{ v_i, v_o \,\middle|\, v \in V \right\}$ is an element of SD_V, if and only if (1) $\emptyset \in [\![\varphi]\!]$; and (2) $\forall m \in [\![\varphi]\!].\ \exists v \in V.\ v_i \in m$.* □

Proposition 1. *The domain* $\langle \mathsf{SD}_V, \rightarrow, \vee, \wedge, \top_{\mathsf{SD}}, \bot_{\mathsf{SD}} \rangle$ *where* $\bot_{\mathsf{SD}} = \bigwedge_{v \in V} \neg v_i \wedge \neg v_o$ *and* $\top_{\mathsf{SD}} = \bot_{\mathsf{SD}} \vee (\bigvee_{v \in V} v_i)$ *is a complete lattice.* □

4.2 The analysis

Now we describe a static analysis that infer *information flow dependencies* for a given program P. The analysis is defined by mean of Boolean equations system, such that, its least solution is an *information flow dependencies* for P. The analysis is described below step-by-step, where at each step we translate one construct of the while programming language, into a corresponding *information flow dependencies* that approximate its possible information flows.

Similar to the analysis of [4], in order to handle *implicit* information flow precisely, we need the set of variables that might be updated during the execution. This information can be obtained either by a separated analysis or by refining our analysis to include it. For the sake of simplicity, we assume this information is already available, and we denote by $\mathcal{U}_\ell$ the set of variables that might be updated when executing S^ℓ.

Definition 4 (dependencies of *":="*). *The* information flow dependencies *of the statement "*$S \equiv x{:=}e$*", is* $E(x,e) = [x_o \leftrightarrow \bigvee_{y \in e} y_i] \wedge [\bigwedge_{\substack{v \in V_S \\ x \neq v}} (v_o \leftrightarrow v_i)]$ □

The above definition states that, the output value of a variable x contains secrets, if and only if, at least one of the input values of $vars(e)$ contains secrets. All other variables remain unchanged. Note that $E(x,e) \equiv x_o \leftrightarrow \mathit{false}$ when $vars(e) = \emptyset$.

Definition 5 (dependencies of *"skip"*). *The* information flow dependencies *of the statement "*$S \equiv skip$*" is* $Id_{V_S} = \bigwedge_{v \in V_S} (v_o \leftrightarrow v_i)$*. Note that we only map the input variables to the output variables.* □

Example 3. Let $S_1^{\ell_1} \equiv x := 2 * y$, $S_2^{\ell_2} \equiv z := w + x$, and $V_{S_1} = V_{S_2} = \{x, w, y, z\}$. The *information flow dependencies* of $S_1^{\ell_1}$ and $S_2^{\ell_2}$ are respectively:

$$\varphi_1 = (x_o \leftrightarrow y_i) \wedge (y_o \leftrightarrow y_i) \wedge (z_o \leftrightarrow z_i) \wedge (w_o \leftrightarrow w_i)$$
$$\varphi_2 = (z_o \leftrightarrow (w_i \vee x_i)) \wedge (y_o \leftrightarrow y_i) \wedge (x_o \leftrightarrow x_i) \wedge (w_o \leftrightarrow w_i)$$

Note that $y \overset{S_1}{\rightsquigarrow} \{x, y\}$ is described by $\{y_i, x_o, y_o\} \in [\![\varphi_1]\!]$, and $w \overset{S_2}{\rightsquigarrow} \{z, w\}$ is described by $\{w_i, z_o, w_o\} \in [\![\varphi_2]\!]$. □

Computing the *information flow dependencies* of $S_1^{\ell_1}; S_2^{\ell_2}$ require a composition operator which is able to compose the corresponding *dependencies*.

Definition 6 (Composing dependencies). *Given two* information flow dependencies $\varphi_1, \varphi_2 \in \mathsf{SD}_V$*, their composition is* $\varphi_1 \sqcap \varphi_2 = \exists V_t.\ \varphi_1[v_o/v_t] \wedge \varphi_2[v_i/v_t]$ *where* $V_t = \{v_t \,|\, v \in V\}$*,* $\varphi_1[v_o/v_t]$ *is a renaming of* φ_1 *obtained by renaming* v_o *to* v_t *for every* $v \in V$*, and* $\varphi_2[v_i/v_t]$ *is a renaming of* φ_2 *obtained by renaming* v_i *to* v_t *for every* $v \in V$*.* □

Note that composing two *dependencies* is done by making the output variables of the first one (v_o of φ_1) equal to the input variables of the second one (v_i of φ_2). In terms of Boolean functions, it is done by variables renaming, Boolean conjunction and elimination of unnecessary variables.

Example 4. Consider again the statements $S_1^{\ell_1}$ and $S_2^{\ell_2}$ from Example 3, and recall that their *information flow dependencies* are respectively φ_1 and φ_2. The *dependencies* for $S_1^{\ell_1}; S_2^{\ell_2}$ and $S_2^{\ell_2}; S_1^{\ell_1}$ are respectively:

$$\begin{aligned}
\varphi_{12} &= \exists x_t, y_t, z_t, w_t.\ \varphi_1[v_o/v_t] \wedge \varphi_2[v_i/v_t] \\
&= \exists x_t, y_t, z_t, w_t. \\
&\qquad [(x_t \leftrightarrow y_i) \wedge (y_t \leftrightarrow y_i) \wedge (z_t \leftrightarrow z_i) \wedge (w_t \leftrightarrow w_i)] \wedge \\
&\qquad [(z_o \leftrightarrow w_t \vee x_t) \wedge (y_o \leftrightarrow y_t) \wedge (x_o \leftrightarrow x_t) \wedge (w_o \leftrightarrow w_t)] \\
&= \underline{(z_o \leftrightarrow w_i \vee y_i)} \wedge (y_o \leftrightarrow y_i) \wedge \underline{(x_o \leftrightarrow y_i)} \wedge (w_o \leftrightarrow w_i)
\end{aligned}$$

$$\begin{aligned}
\varphi_{21} &= \exists x_t, y_t, z_t, w_t.\ \varphi_2[v_o/v_t] \wedge \varphi_1[v_i/v_t] \\
&= \exists x_t, y_t, z_t, w_t. \\
&\qquad [(z_t \leftrightarrow w_i \vee x_i) \wedge (y_t \leftrightarrow y_i) \wedge (x_t \leftrightarrow x_i) \wedge (w_t \leftrightarrow w_i)] \wedge \\
&\qquad [(x_o \leftrightarrow y_t) \wedge (y_o \leftrightarrow y_t) \wedge (z_o \leftrightarrow z_t) \wedge (w_o \leftrightarrow w_t)] \\
&= \underline{(z_o \leftrightarrow w_i \vee x_i)} \wedge (y_o \leftrightarrow y_i) \wedge \underline{(x_o \leftrightarrow y_i)} \wedge (w_o \leftrightarrow w_i)
\end{aligned}$$

In $S_1^{\ell_1}; S_2^{\ell_2}$ the final value of z depends on the initial values of w and y, which is expressed as $z_o \leftrightarrow (w_i \vee y_i)$ in φ_{12}, while in $S_2^{\ell_1}; S_1^{\ell_2}$ the final value of z depends on the initial values of w and x, which is expressed as $z_o \leftrightarrow (w_i \vee x_i)$ in φ_{21}. □

Information flow dependencies for the *"if"* is more complex, because of the implicit and disjunctive information flow. In order to handle these cases we need: (1) a disjunction operation for two *dependencies*, because we need to consider both branched of the *"if"*; and (2) an operation that adds an implicit flow from a variable v to the variables that might be updated during the execution of the *"if"* statement.

Definition 7 (disjunction of dependencies). *Given two* information flow dependencies $\varphi_1, \varphi_2 \in \mathsf{SD}_V$*, their disjunction is* $\varphi_1 \sqcup \varphi_2 = \varphi_1 \vee \varphi_2$ □

Definition 8 (adding implicit flow). *Adding implicit flow from a variable* $x \in V$ *to a variables* $y \in V$ *in a* dependencies $\varphi \in \mathsf{SD}_V$ *is defined as follows:* $\varphi[x \rightsquigarrow y] = (\exists y_o.\ \varphi \wedge (y'_o \leftrightarrow y_o \vee x_i))[y_{o'}/y_o]$ *where* $[y'_o/y_o]$ *mean a renaming of* y'_o *to* y_o*. This definition extends to adding implicit flows from a set of variables* X *to a set of variables* Y*, by iterating over all pairs from* X *and* Y*.* □

In the above definition, adding implicit flows is done by changing the output of y, to consider the previous output, i.e. y_o from φ, and the information from x at input, i.e. x_i. This is done by $y'_o \leftrightarrow y_o \vee x_i$. Elimination of y_o and then renaming of y'_o to y_o is done to keep the formula in the domain SD_V.

Definition 9 (dependencies of *"if"*). *Let* φ_i *be an* information flow dependencies *for* S_i*, the* dependencies *of* $S^\ell \equiv$ " if b then S_1 else S_2" *is defined by adding implicit flows from* $vars(b)$ *to* $\mathcal{U}_\ell$ *in* $\varphi_1 \sqcup \varphi_2$*, namely* $(\varphi_1 \sqcup \varphi_2)[vars(b) \rightsquigarrow \mathcal{U}_\ell]$. □

Example 5. Consider the statement $S^\ell \equiv$ *"if (w=0) then x:=y else z:=y"*. The *dependencies* for "x:=y" and "z:=y" are respectively

$$\varphi_1 = (x_o \leftrightarrow y_i) \wedge (y_o \leftrightarrow y_i) \wedge (z_o \leftrightarrow z_i) \wedge (w_o \leftrightarrow w_i)$$
$$\varphi_2 = (z_o \leftrightarrow y_i) \wedge (y_o \leftrightarrow y_i) \wedge (x_o \leftrightarrow x_i) \wedge (w_o \leftrightarrow w_i)$$

and adding an implicit flow from w to $\mathcal{U}_\ell = \{x, z\}$ in $\varphi_1 \vee \varphi_2$ results in

$$\begin{aligned}\varphi_3 = &\ (\exists x_o, z_o.\ (\varphi_1 \vee \varphi_2) \wedge (x'_o \leftrightarrow (x_o \vee w_i)) \wedge (z'_o \leftrightarrow (z_o \vee w_i)))[x'_o/x_o, z'_o/z_o] \\ = &\ ((x_o \leftrightarrow y_i \vee w_i) \wedge (y_o \leftrightarrow y_i) \wedge (z_o \leftrightarrow z_i \vee w_i) \wedge (w_o \leftrightarrow w_i)) \vee \\ &\ ((z_o \leftrightarrow y_i \vee w_i) \wedge (y_o \leftrightarrow y_i) \wedge (x_o \leftrightarrow x_i \vee w_i) \wedge (w_o \leftrightarrow w_i))\end{aligned}$$

Note that $\{y_i, x_o, y_o\}$ and $\{y_i, z_o, y_o\}$ are models of φ_3, but $\{y_i, x_o, z_o, y_o\}$ is not, which means that no execution cause a flow from y to both x and z. Moreover, for any $m \in [\![\varphi_3]\!]$, if $w_i \in m$ then $y_o, x_o \in m$, which means that in all executions w flow to both x and y. The analysis of [4] cannot provide this information. □

The *information flow dependencies* of *"while b do S_1"* also involves addition of implicit flow and disjunction of *dependencies*. It can be defined as a disjunction of all possible compositions of the statement *"if b then S_1 else skip"*.

Definition 10 (dependencies of *"while"*). *The* information flow dependencies *of* "while b do S_1" *is the least solution of $F = D \sqcup (F \sqcap D)$ where D is the* dependencies *of* "if b then S_1 else *skip*"*. Note that F is recursive and requires fix-point computation.* □

Example 6. In this example we demonstrate why fix-point computation is required for the while loop *dependencies*. Consider the following program:

$$S^\ell \equiv \text{while (w>0) (l:=l-1; w:=l; p:=p+1)}$$

The information flows are: (1) direct *explicit* flows $l \overset{S}{\rightsquigarrow} \{l, w\}$ and $p \overset{S}{\rightsquigarrow} p$; (2) direct *implicit* flow $w \overset{S}{\rightsquigarrow} \{l, w, p\}$; and (3) *indirect* flow $l \overset{S}{\rightsquigarrow} p$. Let us demonstrate how these flows are captured by Definition 10. The *dependencies* of the corresponding *"if"* statement *"if w>0 then (l:=l+1; w:=l; p:=p+1) else skip"* is

$$\begin{aligned} D \equiv \varphi = &\ [(w_o \leftrightarrow w_i) \wedge (l_o \leftrightarrow l_i \vee w_i) \wedge (p_o \leftrightarrow p_i \vee w_i)] \vee \quad (\#0) \\ &\ [(w_o \leftrightarrow w_i \vee l_i) \wedge (l_o \leftrightarrow l_i \vee w_i) \wedge (p_o \leftrightarrow p_i \vee w_i)] \quad (\#1)\end{aligned}$$

Here, (#0) and (#1) correspond to the *"then"* and *"else"* branches respectively, after the addition of the *implicit* information flow $w \rightsquigarrow \{l, w, p\}$. Note that the indirect flow $l \overset{S}{\rightsquigarrow} p$ is not described by D, it will be introduced in the fix-point computation. Using D to compute the fix-point of $F = D \sqcup (F \sqcap D)$ results in

$$\varphi' = (\#0) \vee (\#1) \vee [(w_o \leftrightarrow w_i \vee l_i) \wedge (l_o \leftrightarrow l_i \vee w_i) \wedge \underline{(p_o \leftrightarrow p_i \vee w_i \vee l_i)}]$$

Here, (#0) is the description of zero iterations; (#1) is the description of one iteration; and the rest is the description of two or more iterations, which includes the flow $l \rightsquigarrow p$. Note that, in addition to the possible information flow, one can learn that: (a) there exists an execution where $l \not\rightsquigarrow \{w, p\}$, this is due to (#0) and (#1); and (b) whenever $l \rightsquigarrow p$ then also $l \rightsquigarrow \{w, l\}$. □

Now we are in a position to define the analysis of a program P. Let us denote by F^ℓ the *information flow dependencies* of S^ℓ, then the analysis of a program P is defined as the least solution of the equations system $\mathcal{E}(P) = \{\mathcal{D}(S^\ell) | S^\ell \in P\}$ where $\mathcal{D}(S^\ell)$ is defined as follows:

$$\mathcal{D}(S^\ell) = \begin{cases} F_\ell = E(x,e) & \text{if } S^\ell \equiv (x := e)^\ell \\ F_\ell = Id_{V_P} & \text{if } S^\ell \equiv (skip)^\ell \\ F_\ell = F^{\ell_1} \sqcap F^{\ell_2} & \text{if } S^\ell \equiv (S_1^{\ell_1}; S_2^{\ell_2})^\ell \\ F_\ell = (F_{\ell_1} \sqcup F_{\ell_2})[vars(b) \rightsquigarrow \mathcal{U}_\ell] & \text{if } S^\ell \equiv (\text{if } b \text{ then } C_1^{\ell_1} \text{ else } C_2^{\ell_2})^\ell \\ F_\ell = F'_\ell \sqcup (F_\ell \sqcap F'_\ell) & \text{if } S^\ell \equiv (\text{while } b \text{ do } S_1^{\ell_1})^\ell \\ F'_\ell = (F_{\ell_1} \sqcup Id_{V_P})[vars(b) \rightsquigarrow \mathcal{U}_\ell] & \end{cases}$$

Example 7. Consider the following program P:

((while $(w > 0)$ (((l:=l-1)$^{\ell_1}$; (w:=l)$^{\ell_2}$)$^{\ell_3}$; (p:=p+1)$^{\ell_4}$)$^{\ell_5}$)$^{\ell_6}$; (l:=5)$^{\ell_7}$)$^{\ell_8}$

which is obtained by adding "l:=5" and labels to the program of Example 6. The equations system $\mathcal{E}(P)$ consists of the following equations:

$$\begin{array}{ll} F_{\ell_1} = (l_o \leftrightarrow l_i) \wedge (w_o \leftrightarrow w_i) \wedge (p_o \leftrightarrow p_i) & F_{\ell_2} = (l_o \leftrightarrow l_i) \wedge (w_o \leftrightarrow l_i) \wedge (p_o \leftrightarrow p_i) \\ F_{\ell_3} = F_{\ell_1} \sqcap F_{\ell_2} & F_{\ell_4} = (l_o \leftrightarrow l_i) \wedge (w_o \leftrightarrow w_i) \wedge (p_o \leftrightarrow p_i) \\ F_{\ell_5} = F_{\ell_3} \sqcap F_{\ell_4} & F'_{\ell_6} = (F_{\ell_5} \sqcup Id_{V_P})[w \rightsquigarrow \{l, w, p\}] \\ F_{\ell_6} = F'_{\ell_6} \sqcup (F_{\ell_6} \sqcap F'_{\ell_6}) & F_{\ell_7} = (l_o \leftrightarrow \mathit{false}) \wedge (w_o \leftrightarrow w_i) \wedge (p_o \leftrightarrow p_i) \\ F_{\ell_8} = F_{\ell_6} \sqcap F_{\ell_7} & \end{array}$$

Computing the least fix-point of $\mathcal{E}(P)$ results in the following solution for F_{ℓ_8}:

$$\begin{aligned} F_{\ell_8} = \varphi = & [(w_o \leftrightarrow w_i) \wedge (l_o \leftrightarrow \mathit{false}) \wedge (p_o \leftrightarrow p_i \vee w_i)] \vee \\ & [(w_o \leftrightarrow w_i \vee l_i) \wedge (l_o \leftrightarrow \mathit{false}) \wedge (p_o \leftrightarrow p_i \vee w_i)] \vee \\ & [(w_o \leftrightarrow w_i \vee l_i) \wedge (l_o \leftrightarrow \mathit{false}) \wedge (p_o \leftrightarrow p_i \vee w_i \vee l_i)] \end{aligned}$$

Note that there is no model $m \in [\![\varphi]\!]$ such that $l_o \in m$, because the statement $l := 5$ destruct all secrets that already leaked to l. □

Theorem 1 (Correctness). *For a given statement $S^\ell \in P$, let $\varphi_S \in \mathsf{SD}_{V_P}$ be the least solution for F^ℓ in the equations system $\mathcal{E}(P)$. If $\langle V_I, V_O \rangle \in \mathcal{F}_S$, then $\{v_i \,|v \in V_I\} \cup \{v_o \,|V \in V_O\} \in [\![\varphi_S]\!]$.* □

Information flow analysis is used to ensure non-interference. Recall that two sets of variables V_H and V_L are non-interference, if information cannot flow from $h \in V_H$ to $l \in V_L$. Using the *information flow dependencies*, non-interference can be verified by checking that whenever the input values of all *low* variables do not contain secrets, then their output values will not contain secrets.

Theorem 2 (non-interference). *Let $\varphi_S \in \mathsf{SD}_{V_S}$ be an* information flow dependencies *for S^ℓ. If $\varphi_S \models [(\bigwedge_{v \in V_L} \neg v_i) \rightarrow (\bigwedge_{v \in V_L} \neg v_o)]$ then, V_H and V_L are non-interference in S^ℓ.* □

Example 8. Consider again the program of Example 6 and let $V_L = \{w, l\}$ and $V_H = \{p\}$. You can verify that $\varphi' \models [(\neg l_i \wedge \neg w_i) \rightarrow (\neg l_o \wedge \neg w_o)]$. And indeed there is no flow from p to l or w, hence, V_H and V_L are non-interference. □

4.3 Analyzing Dynamic Security Policies

The assumption that variables are binded to their security classes statically is not always realistic [12]. In this case the security policy is enforced dynamically. In this section, we describe how information flow analysis can be used, to infer possible divisions of the program variables to V_H and V_L, such that non-interference is guaranteed. This can be applied to any information flow analysis that provides a set of possible flows, such as our analysis or the analysis of [4].

Given a statement S, we construct a Boolean function φ_{HL}, over the variables V_S, such that if $m \in [\![\varphi_{HL}]\!]$, then $V_H = m$ and $V_L = V_S \backslash V_H$ are non-interference in S. The idea is that if $x \overset{S}{\rightsquigarrow} y$, then $\varphi_{HL} \models (x \rightarrow y)$. Here, $x \rightarrow y$ means that if x is *high* then y must be *high*.

Proposition 2 (Dynamic HL-policies). *For a given statement S^ℓ, let $\varphi_{HL} = \bar{\exists} V_S.\ [\bigwedge_{x,y \in V_S} (\Psi_{x \rightsquigarrow y} \rightarrow (x \rightarrow y))]$ where $\Psi_{x \rightsquigarrow y}$ is true if $x \overset{S}{\rightsquigarrow} y$, otherwise $false$. Then, for any $m \in [\![\varphi_{HL}]\!]$, $V_H{=}m$ and $V_L{=}V_S \backslash m$ are non-interference in S^ℓ.* □

Example 9. Consider again the program of Example 5, then φ_{HL} is defined by the models: $[\![\varphi_{HL}]\!] = \{\{w,x,z\},\{y,x,z\},\{w,x,y,z\},\{x\},\{z\},\{x,z\},\emptyset\}$. Note that $\varphi_{HL} = (w \rightarrow z \wedge x) \wedge (y \rightarrow z \wedge x)$ which mean that whenever y or w are *high* then x and z must be *high*. □

Using φ_{HL}, when a variable is dynamically binded to a security class, we can infer the security classes of other variables, without re-analyzing the program. Also note that φ_{HL} provides an alternative way for verifying non-interference of V_H and V_L by verifying that $V_H \in [\![\varphi_{HL}]\!]$. Now we demonstrate how $\Psi_{x \rightsquigarrow y}$ can be constructed from *information flow dependencies*.

Proposition 3 (observing information flow). *Let φ_S be an* information flow dependencies *of S, $x, y \in V_S$, and $\Psi_{x \rightsquigarrow y} = (x_i \wedge y_o) \rightarrow \bar{\exists} x_i, y_o.\ \psi$ where $\psi = \varphi_S \wedge x_i \wedge (\bigwedge_{z \neq x \in V} \neg z_i)$. If $x \overset{S}{\rightsquigarrow} y$ then true $\models \Psi_{x \rightsquigarrow y}$, otherwise false $\models \Psi_{x \rightsquigarrow y}$.* □

The idea is: (1) we restrict φ_S to models where only x contain secret at input, i.e. to ψ; (2) we concentrate only on x_i and y_o, i.e. $\bar{\exists} x_i, y_o.\psi$; and (3) if $m = \{x_i, y_o\}$ is a model of $\bar{\exists} x_i, y_o.\ \psi$, the $x \overset{S}{\rightsquigarrow} y$, otherwise $x \overset{S}{\not\rightsquigarrow} y$.

5 Related and Future Work

In the literature, there are several techniques for checking secure information flow in software, ranging from standard *data/control*-flow analysis techniques to type inference. Our work belongs to the former.

Type-based approaches are designed such that well-typed programs do not leak secrets. A type is inductively associated at compile-time with program statements, such that, any statement showing a potential flow disclosing secrets is rejected [15, 16]. In these approaches, statements like *"(if (h=0) then l:=1 else l:=2); l:=0"* are rejected, even though there is no information flow from h to l

when the program terminates. In type-based approaches it is difficult to treat this kind of statements.

From the *data/control-flow* analysis techniques [2, 4, 10, 13, 11], the most related one to our work is of Clark *et al.* [4]. The authors described *termination-sensitive* information flow analysis (using *flow logic*) for a simple while language, as well as for Idealized Algol. The pair (y, x) is used to describe a possible flow from x to y, and sets of pairs are used to describe all possible information flows. For example, an analysis of the statement *"if (w>0) then x:=y else z:=y"* would infer {(w,w), (x,x), (y,y), (z,z), (x,w), (y,w), (z,y), (x,y)}. From this set it is *not possible* to extract *dependencies* between information flows, as we did in Examples 5 and 6. Note that, if we are interested only in the set of possible information flows, our analysis is equivalent to [4] (for the case of while language and termination-insensitive). For future work, we are interested to investigate the relation to [4] from domain point of view, we want to use domain refinement techniques [8] in order to systematically derive the domain of *information flow dependencies* from the domain of *sets of possible flows* [4]. Also, to compare it to the lifting of *sets of possible flows* to *sets of sets of possible flows*.

Boolean functions were used intensively in groundness analyses of logic programs [1]. A term is ground if it does not contain logic variables, i.e. cannot be changed under further instantiation. Several domains based on Boolean functions were suggested for groundness analysis [1], they are different in the degree of groundness dependencies they can express. In [14], the domain refinement techniques of [8] were used to systematically construct these domains. From information flow point of view, we can consider ground terms as objects to which information cannot flow. For future work, we are interested in translating while-language programs into logic programs, such that, groundness analysis of the latter provides information about secure information flow in the former.

Recently, the notion of non-interference was generalized making it parametric relatively to what an attacker can analyze about the input/output information flow [7]. For future work, we want to check if we can generalize the techniques of this paper, to model *abstract information flow dependencies*.

6 Conclusion

For static security policies, we described a technique for modeling information flow with Boolean functions, and demonstrated how it captures *information flow dependencies*. From the practical side, these *dependencies* might be useful for: (1) refining the notion of non-interference; (2) debugging; and (3) efficient implementation. From the theoretical side, they raise the issue of systematic refinement of domains for information flow analysis.

For dynamic security policies, we described a method to infer a Boolean function, such that its models define non-interference *high* and *low* sets. This can be used to infer the security classes dynamically, rather than re-analyzing the program.

For future work, we identified several possible directions: (1) systematic construction of domains for information flow; (2) study the link between information flow analysis and groundness analysis of logic programs; and (3) generalization of the techniques of this paper to the notion of abstract non-interference.

References

1. T. Armstrong, K. Marriott, P. Schachte, and H. Søndergaard. Two classes of Boolean functions for dependency analysis. *Sci. Comput. Program*, 31(1):3–45, 1998.
2. C. Bodei, P. Degano, F. Nielson, and H.R. Nielson. Static analysis for secrecy and non-interference in networks of processes. In *Proc. of PaCT'01*, volume 2127 of *Lecture Notes in Computer Science*, pages 27–41. Springer-Verlag, 2001.
3. R. Bryant. Symbolic Boolean manipulation with ordered binary-decision diagrams. *ACM Comput. Surv.*, 24(3):293–318, 1992.
4. C. Hankin D. Clark and S. Hunt. Information flow for algol-like languages. *Computer Languages*, 28(1):3–28, April 2002.
5. D. E. Denning. A lattice model of secure information flow. *Communications of the ACM*, 19(5):236–242, 1976.
6. H. R. Nielson F. Nielson and C. L. Hankin. *Principles of Programming Analysis*. Springer, 1999.
7. R. Giacobazzi and I. Mastroeni. Abstract non-interference: Parameterizing non-interference by abstract interpretation. In *The 31st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'04)*., January 2004. to appear.
8. R. Giacobazzi and F. Scozzari. A logical model for relational abstract domains. *ACM Trans. Program. Lang. Syst.*, 20(5):1067–1109, 1998.
9. J. A. Gougen and J. Meseguer. Security policies and security models. In *Proc. IEEE Symp. on Security and Privacy*, pages 11–20, 1982.
10. P. Laud. Semantics and program analysis of computationally secure information flow. In *In Programming Languages and Systems, 10th European Symposium On Programming, ESOP*, volume 2028 of *Lecture Notes in Computer Science*, pages 77–91. Springer-Verlag, 2001.
11. M. Mizuno. A least fixed point approach to inter-procedural information flow control. In *Proc. 12th NIST-NCSC National Computer Security Conference*, pages 558–570, 1989.
12. A. Sabelfeld and A.C. Myers. Language-based information-flow security. *IEEE J. on selected ares in communications*, 21(1):5–19, 2003.
13. A. Sabelfeld and D. Sands. A PER model of secure information flow in sequential programs. *Higher-Order and Symbolic Computation*, 14(1):59–91, 2001.
14. F. Scozzari. Logical optimality of groundness analysis. *Theoretical Computer Science*, 277(1-2):149–184, 2002.
15. C. Skalka and S. Smith. Static enforcement of security with types. In *ICFP'00*, pages 254–267. ACM press, 2000.
16. D. Volpano, G. Smith, and C. Irvine. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(2,3):167–187, 1996.
17. G. Winskel. *The formal semantics of programming languages: an introduction*. MIT press, 1993.