Information Flow Testing

Gurvan Le Guernic

Information Flow Testing
The Third Path towards Confidentiality Guarantee

Gurvan Le Guernic[1,2,⋆]

[1] Kansas State University - Manhattan, KS 66506 - USA
[2] IRISA - Campus universitaire de Beaulieu, 35042 Rennes - France
http://www.irisa.fr/lande/gleguern
Gurvan.Le_Guernic@irisa.fr

Abstract. Noninterference, which is an information flow property, is typically used as a baseline security policy to formalize confidentiality of secret information manipulated by a program. Noninterference verification mechanisms are usually based on static analyses and, to a lesser extent, on dynamic analyses. In contrast to those works, this paper proposes an information flow testing mechanism. This mechanism is sound from the point of view of noninterference. It is based on standard testing techniques and on a combination of dynamic and static analyses. Concretely, a semantics integrating a dynamic information flow analysis is proposed. This analysis makes use of static analyses results. This special semantics is built such that, once a path coverage property has been achieved on a program, a sound conclusion regarding the noninterfering behavior of the program can be established.

1 Introduction

With the intensification of communication in information systems, interest in security has increased. This paper deals with the problem of confidentiality, more precisely with *noninterference* in sequential programs. This notion is based on ideas from classical information theory (1) and has first been introduced by Goguen and Meseguer (2) as the absence of *strong dependency* (3).

> "information is transmitted from a source to a destination only when *variety* in the source can be conveyed to the destination" Cohen (3, Sect.1).

A sequential program, `P`, is said to be *noninterfering* if the values of its public (or low) outputs do not depend on the values of its secret (or high) inputs. Formally, noninterference is expressed as follows: a program `P` is noninterferent if and only if, given any two initial input states σ_1 and σ_2 that are indistinguishable with respect to low inputs, the executions of `P` started in states σ_1 and σ_2 are *low-indistinguishable*. *Low-indistinguishable* means that there is no observable

⋆ The author was partially supported by National Science Foundation grants CCR-0296182, ITR-0326577 and CNS-0627748.

difference between the public outputs of both executions. In the simplest form of the *low-indistinguishable* definition, public outputs include only the final values of low variables. In a more general setting, the definition may additionally involve intentional aspects such as power consumption, computation times, etc.

Static analyses for noninterference have been studied extensively and are well surveyed by Sabelfeld and Myers (4). Recently, and to a lesser extent, dynamic analyses for noninterference have been proposed (5; 6; 7). However, to be useful, those dynamic analyses must be combined with an information flow correction mechanism in order to enforce noninterference at run-time. As shown by Le Guernic and Jensen (8), in order to prevent the correction mechanism to become a new covert channel, additional constraints are put on the dynamic analysis. Those constraints limit the precision achievable by a monitor enforcing noninterference. A dynamic information flow analysis which is not used at run-time to *enforce* noninterference could therefore be more precise than its equivalent noninterference monitor.

This paper develops an information flow testing mechanism based on such a dynamic information flow analysis which is not aimed at enforcing noninterference at run-time. It is presented as a special semantics integrating a dynamic information flow analysis combined with results of a static analysis. A distinguishing feature of the dynamic information flow analysis proposed, compared to other standard run-time mechanisms, lies in the property overseen. Dynamically analyzing information flow is more complicated than, *e.g.*, monitoring divisions by zero, since it must take into account not only the current state of the program but also the execution paths *not taken* during execution. For example, executions of the following programs (a) **if** h **then** x `:=1` **else skip** and (b) **if** h **then skip else skip** in an initial state where h is `false` are equivalent concerning executed commands. In contrast, (b)'s executions are noninterfering, while (a)'s executions are not. Executions of (a), where x is not equal to `1`, do not give the same final value to x if h is `true` or `false`.

The next section starts by giving an overview of the dynamic information flow analysis at the basis of the approach. It then describes the testing technique used and, finally, presents the language studied. Before characterizing in Sect. 3.2 and 3.3 the static analyses used by the dynamic analysis and stating some properties of the proposed analysis, Section 3 presents the semantics which incorporates this dynamic analysis. Finally, Sect. 4 concludes.

2 Presentation of the approach

With regard to noninterference, a dynamic analysis suited only for the detection of information flows, and not their correction, can be used only for noninterference testing. The idea behind noninterference testing is to run enough executions of a program in order to cover a "high enough percentage" of all possible executions of the program. In cases where the dynamic analysis results enable to conclude that all the executions evaluated are safe, users gain a confidence in the "safe" behavior of the program which is *proportional to the coverage percentage.*

When dealing with the confidentiality of secret data, a percentage lower than 100% does not seem acceptable. The aim of noninterference testing is then to cover all possible executions. It is not possible to run an execution for every possible input set (as there are frequently infinitely many input values). However, the results of a dynamic information flow analysis may be the same for many executions with different inputs. Therefore, it may be possible to conclude about the noninterference behavior of any execution of a program by testing a limited, hopefully finite, number of executions. Before presenting the testing approach proposed in this paper, this section introduces some terminology, formally defines what is meant by "noninterfering execution" and gives an overview of the dynamic information flow analysis proposed in Section 3.

2.1 Overview of the noninterference analysis

A *direct flow* is a flow from the right side of an assignment to the left side. Executing "x **:=** y" creates a direct flow from y to x. An *explicit indirect flow* is a flow from the test of a conditional to the left side of an assignment in the branch executed. Executing "**if** c **then** x **:=** y **else skip end**" when c is `true` creates an explicit indirect flow from y to x. An *implicit indirect flow* is a flow from the test of a conditional to the left side of an assignment in the branch which is not executed. Executing "**if** c **then** x **:=** y **else skip end**" when c is `false` creates an implicit indirect flow from y to x.

A "safe" execution is a noninterfering execution. In this article, as commonly done, noninterference is defined as the absence of strong dependencies between the secret inputs of an execution and the final values of some variables which are considered to be publicly observable at the end of the execution.

For every program `P`, two sets of variable identifiers are defined. The set of variables corresponding to the secret inputs of the program is designated by $\mathcal{S}(\texttt{P})$. The set of variables whose final value are publicly observable at the end of the execution is designated by $\mathcal{O}(\texttt{P})$. No requirements are put on $\mathcal{S}(\texttt{P})$ and $\mathcal{O}(\texttt{P})$ other than requiring them to be subsets of $\mathbb{X}$. A variable x is even allowed to belong to both sets. In such a case, in order to be noninterfering, the program `P` would be required to, at least, reset the value of x.

In the following definitions, we consider that a program state may contain more than just a value store. This is the reason why a distinction is done between program states (ζ) and value stores (σ). Following Definition 1, two program states ζ_1, respectively ζ_2, containing the value stores σ_1, respectively σ_2, are said to be *low equivalent* with regards to a set of variables V, written $\zeta_1 \overset{V}{=} \zeta_2$, if and only if the value of any variable belonging to V is the same in σ_1 and σ_2.

Definition 1 (Low Equivalent States).
Two states ζ_1, respectively ζ_2, containing the value stores σ_1, respectively σ_2, are low equivalent *with regards to a set of variables V, written $\zeta_1 \overset{V}{=} \zeta_2$, if and only if the value of any variable belonging to V is the same in σ_1 and σ_2:*

$$\zeta_1 \overset{V}{=} \zeta_2 \quad \Longleftrightarrow \quad \forall x \in V : \sigma_1(x) = \sigma_2(x)$$

Definition 2 (Noninterfering Execution).
Let $\Downarrow_s$ denote a big-step semantics. Let $\overline{\mathcal{S}(P)}$ be the complement of $\mathcal{S}(P)$ in the set $\mathbb{X}$. For all programs P, program states ζ_1 and ζ_1', an execution with the semantics $\Downarrow_s$ of the program P in the initial state ζ_1 and yielding the final state ζ_1' is noninterfering, if and only if, for every program states ζ_2 and ζ_2' such that the execution with the semantics $\Downarrow_s$ of the program P in the initial state ζ_2 yields the final state ζ_2':

$$\zeta_1 \overset{\overline{\mathcal{S}(P)}}{=} \zeta_2 \;\Rightarrow\; \zeta_1' \overset{\mathcal{O}(P)}{=} \zeta_2'$$

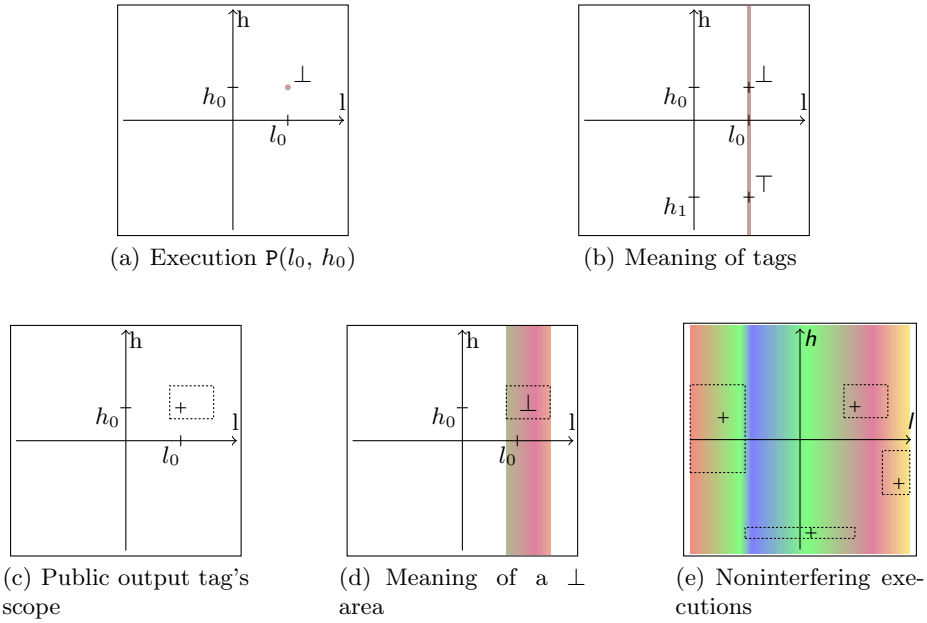
The dynamic information flow analysis uses results of static analyses. The semantics integrating the dynamic analysis, now on called *analyzing semantics*, treats directly the direct and explicit indirect flows. For implicit indirect flows, a static analysis is run on the unexecuted branch of every conditional whose test carries variety — i.e. is influenced by the secret inputs of the program.

A program state for this semantics is composed of a value store, σ, mapping variables to values, and a tag store, ρ, mapping variables to a tag. This tag reflects the level of *variety* of a variable. At any point of the execution, a variable whose tag is $\bot$ would have the exact same value for any execution started with the same public inputs. A variable whose tag is $\top$ may have a different value for an execution started with the same public inputs. In other words, the variety in the secret inputs may be carried to the variables which are tagged $\top$, and only those variables.

2.2 Noninterference Testing.

The main idea behind noninterference testing has been exposed above. Figure 1 sketches this idea. Let `P` be a program whose secret inputs are represented by h, public inputs by l, and public outputs by a color (or level of gray). `P`(l_i, h_j), where (l_i, h_j) are input values, is the public output, represented by a color, of the execution of `P` with the inputs (l_i, h_j). In the representations of Fig. 1, public input values are represented on the x-axis and secret input values are represented on the y-axis. Each point of the different graphs corresponds to the execution of `P` with, as inputs, the coordinates of this point. Whenever a point in the graph is colored, the color corresponds to the public output value of the execution of `P` with, as inputs, the coordinates of the colored point. Figure 1(a) represents the execution of `P` with inputs (l_0, h_0). Its public output value is represented by the color (or level of gray) • and its tag — result of the dynamic information flow analysis — is $\bot$ (the public output does not carry variety). Figure 1(b) shows the meaning of this tag. As the public output tag of `P`(l_0, h_0) is $\bot$, it means that for any secret inputs h_j the public output value of `P`(l_0, h_j) is the same as for `P`(l_0, h_0); it is •. Even if there exist secret inputs h_1 for which the public output tag of `P`(l_0, h_1) is $\top$, any execution of `P` with public inputs l_0 is noninterfering. It only means that the dynamic analysis is not precise enough to directly detect that the execution of `P` with inputs (l_0, h_1) is noninterfering. However, this result can be indirectly deduced from the result of the dynamic analysis of the execution of `P` with inputs (l_0, h_0).

The main challenge of noninterference testing is to develop a dynamic analysis for which it is possible to characterize a set of executions which associate the same tag to the public output as an execution which as already been tested. For example, assume that it has been proved that all executions in the dashed area in Fig. 1(c) associate the same tag to the public output as the execution of P with inputs (l_0, h_0). As this tag is $\bot$, it is possible to conclude from the single execution of P with inputs (l_0, h_0) that all colored (or grayed) executions in Fig. 1(d) are noninterfering. Therefore, with only a limited number of executions, as in Fig. 1(e), it is possible to deduce that the program is noninterfering for a wide range of inputs which can be characterized.



(a) Execution P(l_0, h_0)

(b) Meaning of tags

(c) Public output tag's scope

(d) Meaning of a $\bot$ area

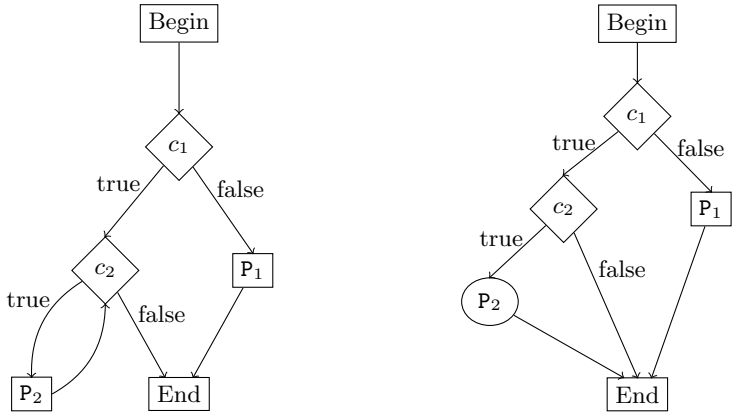
(e) Noninterfering executions

Fig. 1. Sketch of the main idea of noninterference testing.

Which executions need to be tested? As exposed above, in order to be able to conclude on the interference behavior of a program by testing it, it is necessary to be able to characterize a finite number of executions which are sufficient to conclude about all executions of this program. It is then necessary to develop a dynamic analysis which has the right balance between the number of executions covered by one test and the precision of the analysis.

The solution approached here assumes there is no recursive calls and is based on "acyclic Control Flow Graphs" (aCFG). As its name suggests an aCFG is a Control Flow Graph (CFG) without cycles. In an aCFG, there is no edge

from the last nodes of the body of a loop statement to the node corresponding to the test of this loop statement. Instead, there is an edge from every last node of the body of the loop to the node corresponding to the block following the loop statement. Figure 2(a) shows the standard CFG of the following code: "**if** c_1 **then while** c_2 **do** $\mathtt{P}_1$ **done else** $\mathtt{P}_2$ **end**". Figure 2(b) shows its aCFG. In an acyclic CFG, there is a finite number of paths. The maximum number of paths is equal to 2^b, where b is the number of branching statements (`if` and `while` statements) in the program.



(a) Standard CFG (b) Acyclic CFG

Fig. 2. CFG and aCFG of the same program

The approach to noninterference testing proposed in this section is based on a dynamic information flow analysis which returns the same result for any execution that follows the same path in the aCFG of the program analyzed. Let the acyclic CFG trace of an execution be the list of nodes of the aCFG encountered during the execution. Let $\tau[\![\sigma \vdash \mathtt{P}]\!]$ be the acyclic CFG trace of the execution of program $\mathtt{P}$ with initial value store σ. Let $\mathbb{T}[\![\zeta \vdash \mathtt{P}]\!]$ be the result of the dynamic information flow analysis of the execution of $\mathtt{P}$ in the initial program state ζ (its formal definition is given on page 8). The constraint imposed on the dynamic information flow analysis for the noninterference testing approach proposed is formalized in Hypothesis 1.

Hypothesis 1 (Usable for noninterference testing)
For all programs P, value stores σ_1 and σ_2, and tag stores ρ:

$$\tau[\![\sigma_1 \vdash P]\!] = \tau[\![\sigma_2 \vdash P]\!] \quad \Rightarrow \quad \mathbb{T}[\![(\sigma_1, \rho) \vdash P]\!] = \mathbb{T}[\![(\sigma_2, \rho) \vdash P]\!]$$

With such a dynamic analysis, the problem of verifying the noninterference behavior of any execution is reduced to the well known testing problem of achieving

100% feasible paths coverage (9; 10; 11; 12; 13). For path whose branch conditions are linear functions of inputs, Gupta et al. (14) propose a technique which finds a solution in one iteration or guarantees that the path is infeasible.

2.3 The Language: Syntax & Standard Semantics

The language used to describe programs studied in this article is an imperative language for sequential programs. Expressions in this language are deterministic — their evaluation in a given program state always results in the same value — and are free of side effects — their evaluation has no influence on the program state. The standard semantics of the language is given in Fig. 3. The evaluation symbol ($\Downarrow$) is given a subscript letter in order to distinguish between the standard semantics ($\mathcal{S}$) and the analyzing one ($\mathcal{A}$). The standard semantics is based on rules written in the format: $\sigma \vdash P \Downarrow_{\mathcal{S}} \sigma'$. Those rules means that, with the initial program state σ, the evaluation of the program P yields the final program state σ'. Let $\mathbb{X}$ be the domain of variable identifiers and $\mathbb{D}$ be the semantics domain of values. A program state is a value store σ ($\mathbb{X} \to \mathbb{D}$) mapping variable identifiers to their respective value. The definition of value stores is extended to expressions, so that $\sigma(e)$ is the value of the expression e in the program state σ.

$$\frac{}{\sigma \vdash \textbf{skip} \Downarrow_{\mathcal{S}} \sigma} \qquad \frac{\sigma(e) = \texttt{true} \qquad \sigma \vdash P^{\mathrm{l}} \,;\, \textbf{while}\ e\ \textbf{do}\ P^{\mathrm{l}}\ \textbf{done} \Downarrow_{\mathcal{S}} \sigma'}{\sigma \vdash \textbf{while}\ e\ \textbf{do}\ P^{\mathrm{l}}\ \textbf{done} \Downarrow_{\mathcal{S}} \sigma'}$$

$$\frac{}{\sigma \vdash x := e \Downarrow_{\mathcal{S}} \sigma[x \mapsto \sigma(e)]} \qquad \frac{\sigma(e) = v \qquad \sigma \vdash P^{v} \Downarrow_{\mathcal{S}} \sigma'}{\sigma \vdash \textbf{if}\ e\ \textbf{then}\ P^{\texttt{true}}\ \textbf{else}\ P^{\texttt{false}}\ \textbf{end} \Downarrow_{\mathcal{S}} \sigma'}$$

$$\frac{\sigma \vdash P^{\mathrm{h}} \Downarrow_{\mathcal{S}} \sigma^{\mathrm{h}} \qquad \sigma^{\mathrm{h}} \vdash P^{\mathrm{t}} \Downarrow_{\mathcal{S}} \sigma^{\mathrm{t}}}{\sigma \vdash P^{\mathrm{h}} \,;\, P^{\mathrm{t}} \Downarrow_{\mathcal{S}} \sigma^{\mathrm{t}}} \qquad \frac{\sigma(e) = \texttt{false}}{\sigma \vdash \textbf{while}\ e\ \textbf{do}\ P^{\mathrm{l}}\ \textbf{done} \Downarrow_{\mathcal{S}} \sigma}$$

Fig. 3. Rules of the standard semantics

3 The Analyzing Semantics

The dynamic information flow analysis and the analyzing semantics are defined together in Fig. 4. Information flows are tracked using tags. At any execution step, every variable has a tag which reflects the fact that this variable may carry variety or not.

3.1 A semantics Making Use of Static Analysis Results

Let $\mathbb{X}$ be the domain of variable identifiers, $\mathbb{D}$ be the semantics domain of values, and $\mathbb{T}$ be the domain of tags. In the remainder of this article, $\mathbb{T}$ is equal to $\{\top, \bot\}$.

Those tags form a lattice such that $\bot \sqsubset \top$. $\top$ is the tag associated to variables that *may* carry variety — i.e. whose value may be influenced by the secret inputs.

The analyzing semantics described in Fig. 4 is based on rules written in the format:

$$\zeta \vdash \mathtt{P} \Downarrow_{\mathcal{A}} \zeta' : X$$

This reads as follows: in the analyzing execution state ζ, program $\mathtt{P}$ yields the analyzing execution state ζ' and a set of variables X. An analyzing execution state ζ is a pair $(\sigma,\ \rho)$ composed of a value store σ and a tag store ρ. A value store $(\mathbb{X} \to \mathbb{D})$ maps variable identifiers to values. A tag store $(\mathbb{X} \to \mathbb{T})$ maps variable identifiers to tags. The definitions of value store and tag store are extended to expressions. $\sigma(e)$ is the value of the expression e in a program state whose value store is σ. Similarly, $\rho(e)$ is the tag of the expression e in a program state whose tag store is ρ. $\rho(e)$ is formally defined as follows, with $\mathcal{V}(e)$ being the set of free variables appearing in the expression e:

$$\rho(e) = \bigsqcup_{x \in \mathcal{V}(e)} \rho(x)$$

Definition 3 ($\mathbb{T}[\![\zeta \vdash \mathtt{P}]\!]$)**.**
$\mathbb{T}[\![\zeta \vdash \mathtt{P}]\!]$ is defined to be the final tag store of the execution of $\mathtt{P}$ with the initial state ζ. Therefore, for all programs $\mathtt{P}$, value stores σ and tag stores ρ, if the evaluation of $\mathtt{P}$ in the initial state $(\sigma,\ \rho)$ terminates then there exists a value stores σ' and a set of variables X' such that:

$$\zeta \vdash \mathtt{P} \Downarrow_{\mathcal{A}} (\sigma', \mathbb{T}[\![\zeta \vdash \mathtt{P}]\!]) : X'$$

The set of variables X contains all the variables whose value may be modified by an execution of $\mathtt{P}$ having the same trace than the current execution — i.e. all executions whose trace is $\tau[\![\sigma \vdash \mathtt{P}]\!]$.

The semantics rules make use of static analyses results. In Fig. 4, application of a static information flow analysis to the piece of code $\mathtt{P}$ is written: $[\![\rho \vdash \mathtt{P}]\!]^{\sharp_{\mathcal{G}}}$. The analysis of a program $\mathtt{P}$ must return a pair $(\mathfrak{D}, \mathfrak{X})$. $\mathfrak{D}$, which is a subset of $(\mathbb{X} \times \mathbb{X})$, is an over-approximation of the dependencies between the initial and final values of the variables created by any execution of $\mathtt{P}$. $\mathfrak{D}(x)$, which is equal to $\{y \mid (x, y) \in \mathfrak{D}\}$, is the set of variables whose initial value may influence the final value of x after an execution of $\mathtt{P}$. $\mathfrak{X}$, which is a subset of $\mathbb{X}$, is an over-approximation of the set of variables which are potentially defined — i.e. whose value may be modified — by an execution of $\mathtt{P}$. This static analysis can be any such analysis that satisfies a set of formal constraints which are stated below.

The analyzing semantics rules are straightforward. As can be expected, the execution of a **skip** statement with the semantics given in Fig. 4 yields a final state equal to the initial state. The execution of the assignment of the value of the expression e to the variable x yields an execution state $(\sigma',\ \rho')$. The final value store (σ') is equal to the initial value store (σ) except for the variable

$$\frac{}{\zeta \;\vdash\; \mathbf{skip} \;\Downarrow_{\mathcal{A}}\; \zeta : \emptyset} \qquad (\mathrm{E}_{\mathcal{A}}\text{-SKIP})$$

$$\frac{}{(\sigma,\ \rho) \;\vdash\; x := e \;\Downarrow_{\mathcal{A}}\; (\sigma[x \mapsto \sigma(e)],\ \rho[x \mapsto \rho(e)]) : \{x\}} \qquad (\mathrm{E}_{\mathcal{A}}\text{-ASSIGN})$$

$$\frac{\zeta \;\vdash\; P^{\mathrm{h}} \;\Downarrow_{\mathcal{A}}\; \zeta^{\mathrm{h}} : X^{\mathrm{h}} \qquad \zeta^{\mathrm{h}} \;\vdash\; P^{\mathrm{t}} \;\Downarrow_{\mathcal{A}}\; \zeta^{\mathrm{t}} : X^{\mathrm{t}}}{\zeta \;\vdash\; P^{\mathrm{h}}\ ;\ P^{\mathrm{t}} \;\Downarrow_{\mathcal{A}}\; \zeta^{\mathrm{t}} : X^{\mathrm{h}} \cup X^{\mathrm{t}}} \qquad (\mathrm{E}_{\mathcal{A}}\text{-SEQUENCE})$$

$$\frac{\begin{array}{c}\sigma(e) = v \qquad (\sigma,\ \rho) \;\vdash\; P^{v} \;\Downarrow_{\mathcal{A}}\; (\sigma^{v},\ \rho^{v}) : X^{v} \qquad [\![\rho \vdash P^{\neg v}]\!]^{\sharp_{\mathcal{G}}} = (\mathfrak{D}, \mathfrak{X}) \\ X^{e} = X^{v} \cup \mathfrak{X} \qquad \rho' = \rho^{v} \sqcup \left((X^{e} \times \{\rho(e)\}) \cup (\overline{X^{e}} \times \{\bot\})\right)\end{array}}{(\sigma,\ \rho) \;\vdash\; \mathbf{if}\ e\ \mathbf{then}\ P^{\mathtt{true}}\ \mathbf{else}\ P^{\mathtt{false}}\ \mathbf{end} \;\Downarrow_{\mathcal{A}}\; (\sigma^{v},\ \rho') : X^{v}} \qquad (\mathrm{E}_{\mathcal{A}}\text{-IF})$$

$$\frac{\begin{array}{c}\sigma(e) = \mathtt{false} \qquad [\![\rho \vdash P^{\mathrm{l}}\ ;\ \mathbf{while}\ e\ \mathbf{do}\ P^{\mathrm{l}}\ \mathbf{done}]\!]^{\sharp_{\mathcal{G}}} = (\mathfrak{D}, \mathfrak{X}) \\ \rho' = \rho \sqcup \left((\mathfrak{X} \times \{\rho(e)\}) \cup (\overline{\mathfrak{X}} \times \{\bot\})\right)\end{array}}{(\sigma,\ \rho) \;\vdash\; \mathbf{while}\ e\ \mathbf{do}\ P^{\mathrm{l}}\ \mathbf{done} \;\Downarrow_{\mathcal{A}}\; (\sigma,\ \rho') : \emptyset} \qquad (\mathrm{E}_{\mathcal{A}}\text{-WHILE}_{\mathtt{false}})$$

$$\frac{\begin{array}{c}\sigma(e) = \mathtt{true} \qquad \sigma \;\vdash\; P^{\mathrm{l}}\ ;\ \mathbf{while}\ e\ \mathbf{do}\ P^{\mathrm{l}}\ \mathbf{done} \;\Downarrow_{\mathcal{S}}\; \sigma^{v} \\ [\![\rho \vdash P^{\mathrm{l}}\ ;\ \mathbf{while}\ e\ \mathbf{do}\ P^{\mathrm{l}}\ \mathbf{done}]\!]^{\sharp_{\mathcal{G}}} = (\mathfrak{D}, \mathfrak{X}) \\ \rho^{\mathfrak{D}} = \left\{ \left(x\ ,\ \bigsqcup_{y \in \mathfrak{D}(x)} \rho(y)\right) \mid x \in \mathbb{X} \right\} \\ \rho^{\mathfrak{R}} = \rho^{\mathfrak{D}} \sqcup \left((\mathfrak{X} \times \{\rho(e)\}) \cup (\overline{\mathfrak{X}} \times \{\bot\})\right)\end{array}}{(\sigma,\ \rho) \;\vdash\; \mathbf{while}\ e\ \mathbf{do}\ P^{\mathrm{l}}\ \mathbf{done} \;\Downarrow_{\mathcal{A}}\; (\sigma^{v},\ \rho^{\mathfrak{R}}) : \mathfrak{X}} \qquad (\mathrm{E}_{\mathcal{A}}\text{-WHILE}_{\mathtt{true}})$$

Fig. 4. Rules of the analyzing semantics

x. The final value store maps the variable x to the value of the expression e evaluated with the initial value store ($\sigma(e)$). Similarly, the final tag store (ρ') is equal to the initial tag store (ρ) except for the variable x. The tag of x after the execution of the assignment is equal to the tag of the expression computed using the initial tag store ($\rho(e)$). $\rho(e)$ represents the level of the information flowing into x through direct flows.

For an `if` statement, the branch (P^{v}) designated by the value of e is executed and the other one ($P^{\neg v}$) is analyzed. The final value store is the one returned by the execution of P^{v}. The final tag store (ρ') is the least upper bound of the tag store returned by the execution of P^{v} and a tag store reflecting indirect flows. This latter tag store associates the tag of the branching condition to variables potentially defined by an execution having the same trace or an execution of the other branch. If the tag of the branching condition is $\bot$, the final tag store is therefore equal to the tag store returned by the execution of P^{v}.

The execution of `while` statements is similar to the execution of `if` statements. However, in order to be able to apply the testing technique exposed in Section 2.2, it is required to have the same tag store for every execution following the same path in the acyclic CFG. Therefore, the final tag store is computed from the result of a static analysis of the branch executed (**skip** if the branching condition is `false`) and not from the tag store obtained by the execution of the branch designated by the branching condition. For the same reason, the set of variables returned by the execution of a `while` statement is obtained by static analysis of the branch executed.

3.2 Hypotheses on the static analysis used

The static analysis used on unexecuted branches is not formally defined. In fact, the dynamic analysis can use any static analysis which complies with the three following hypotheses and returns a pair, whose first element is a relation between variables — i.e. a set of pairs of variables — and second element is a set of variables.

The first two hypotheses require a sound static analysis. Hypothesis 2 simply requires the static analysis used to be a *sound* analysis of defined variables. More precisely, it requires that the second element of the static analysis result ($\mathfrak{X}$) contains all the variables which may be defined by an execution of the analyzed program. This is a straightforward requirement as the result of the static analysis is used to take into account implicit indirect flows. Hypothesis 3 requires the static analysis used to be a *sound* analysis of dependencies between the final values of variables and their initial values. The last hypothesis requires only the static analysis to be deterministic.

Hypothesis 2 (Sound detection of modified variables.)
For all tag stores ρ_i, analysis results $(\mathfrak{D}, \mathfrak{X})$, testing execution states $(\sigma_i,\ \rho_i)$ and $(\sigma_f,\ \rho_f)$, programs P and sets of variables X such that:

1. $[\![\rho_i \vdash P]\!]^{\sharp_\mathcal{G}} = (\mathfrak{D}, \mathfrak{X})$
2. $(\sigma_i,\ \rho_i) \vdash P \Downarrow_\mathcal{A} (\sigma_f,\ \rho_f) : X$,

the following holds: $\forall x \notin \mathfrak{X} \,.\, \sigma_f(x) = \sigma_i(x)$.

Hypothesis 3 (Sound detection of dependencies.)
For all analysis results $(\mathfrak{D}, \mathfrak{X})$, tag stores ρ_1, testing execution states $(\sigma_1,\ \rho_1)$, $(\sigma'_1,\ \rho'_1)$, $(\sigma_2,\ \rho_2)$ and $(\sigma'_2,\ \rho'_2)$, programs P, and sets of variables X_1 and X_2 such that:

1. $[\![\rho_1 \vdash P]\!]^{\sharp_\mathcal{G}} = (\mathfrak{D}, \mathfrak{X})$
2. $(\sigma_1,\ \rho_1) \vdash P \Downarrow_\mathcal{A} (\sigma'_1,\ \rho'_1) : X_1$,
3. $(\sigma_2,\ \rho_2) \vdash P \Downarrow_\mathcal{A} (\sigma'_2,\ \rho'_2) : X_2$,

for all x in $\mathbb{X}$: $\big(\forall y \in \mathfrak{D}(x) \,.\, \sigma_1(y) = \sigma_2(y)\big) \Rightarrow \sigma'_1(x) = \sigma'_2(x)$.

Hypothesis 4 (Deterministic static analysis)
The static analysis used is a deterministic analysis. For all tag stores ρ and programs P, the following holds: $|\ range([\![\rho \vdash P]\!]^{\sharp_\mathcal{G}})\ | = 1$.

What is the reason for having a tag store in parameter of the static analysis? In fact, there is no need for the tag store which is given to the static analysis. This additional parameter to the static analysis has been added in order to be able to use existing noninterference type systems in a straightforward way.

Using this tag store, it is easy to construct an analysis satisfying the hypotheses presented above from a type inference mechanism for a sound noninterference type system. Let $\mathbb{X}\uparrow^{\rho}$ be the set of variables whose tag in ρ is $\top$. Let Γ_{ρ} be a typing environment in which variables belonging to $\mathbb{X}\uparrow^{\rho}$ are typed secret, other variables can be typed secret or public. Let $\mathbb{X}\downarrow_{\Gamma}$ be the set of variables typed public in Γ and $\mathbb{X}\uparrow^{\Gamma}$ the set of variables typed secret in Γ. Let $\mathfrak{D}_{\Gamma}$ be a relation among variables which associates every variable of $\mathbb{X}\uparrow^{\Gamma}$ to every variable ($\mathbb{X}$), and associates every variable of $\mathbb{X}\downarrow_{\Gamma}$ to every variable of $\mathbb{X}\downarrow_{\Gamma}$. If P is well-typed under Γ_{ρ} and the program "**if** h **then** P **else skip end**" is well-typed under Γ'_{ρ} with h typed secret in Γ'_{ρ}, then $(\mathfrak{D}_{\Gamma}\,,\, \mathbb{X}\uparrow^{\Gamma'})$ is a result satisfying the Hypotheses 2 and 3 if any variable tagged $\bot$ in ρ has the same value in σ_1 and σ_2.

3.3 Another Characterization of *Usable* Static Analyses

The above hypotheses define which static information flow analyses are *usable*, i.e. which static analyses can be used with the special semantics given in Fig. 4. However, Hypotheses 2 and 3 are stated using the special semantics itself. This makes it more difficult to prove that a given static analysis satisfies those hypotheses.

Figure 5 defines a set of *acceptability* rules. The result $(\mathfrak{D}, \mathfrak{X})$ of a static information flow analysis of a given program (P) is *acceptable* for the analyzing semantics only if the result satisfies those rules. This is written: $(\mathfrak{D}, \mathfrak{X}) \models$ P. In the definitions of those rules, $\mathcal{I}d$ denotes the identity relation. $\circ$ is the operation of composition of relations.

$$(S \circ R) = \bigcup_{(a,b)\in R} \{(a, c) \mid (b, c) \in S\}$$

Using the acceptability rules of Fig. 5, it is possible to characterize some static information flow analyses which are *usable* with the analyzing semantics without referring to the analyzing semantics itself. It is also possible to generate a *usable* static information flow analysis by fix-point computation on the acceptability rules; in fact, only on the rule for loop statements. However, those acceptability rules do not define a most precise usable static analysis.

As stated by Theorem 1, any *acceptable* static analysis result satisfies Hypothesis 2. Theorem 2 states that any *acceptable* static analysis result satisfies Hypothesis 3.

Theorem 1 (Acceptable imply sound detection of defined variables). *For all programs* P*, and analysis result* $(\mathfrak{D}, \mathfrak{X})$ *such that* $(\mathfrak{D}, \mathfrak{X}) \models$ P*, the Hypothesis 2 holds.*

$$(\mathfrak{D},\mathfrak{X}) \models \textbf{skip} \quad \text{iff} \quad \mathfrak{D} \supseteq \mathcal{I}d$$

$$(\mathfrak{D},\mathfrak{X}) \models x := e \quad \text{iff} \quad \mathfrak{D} \supseteq \mathcal{I}d[x \mapsto \mathcal{V}(e)] \quad \wedge \quad \mathfrak{X} \supseteq \{x\}$$

$$(\mathfrak{D},\mathfrak{X}) \models \mathtt{P}^{\mathrm{h}} \ ; \ \mathtt{P}^{\mathrm{t}}$$
iff there exist $(\mathfrak{D}^{\mathrm{h}},\mathfrak{X}^{\mathrm{h}})$ and $(\mathfrak{D}^{\mathrm{t}},\mathfrak{X}^{\mathrm{t}})$ such that:
$(\mathfrak{D}^{\mathrm{h}},\mathfrak{X}^{\mathrm{h}}) \models \mathtt{P}^{\mathrm{h}} \quad \wedge \quad (\mathfrak{D}^{\mathrm{t}},\mathfrak{X}^{\mathrm{t}}) \models \mathtt{P}^{\mathrm{t}}$
$\mathfrak{D} \supseteq (\mathfrak{D}^{\mathrm{h}} \circ \mathfrak{D}^{\mathrm{t}}) \quad \wedge \quad \mathfrak{X} \supseteq (\mathfrak{X}^{\mathrm{h}} \cup \mathfrak{X}^{\mathrm{t}})$

$$(\mathfrak{D},\mathfrak{X}) \models \textbf{if } e \textbf{ then } \mathtt{P}^{\mathtt{true}} \textbf{ else } \mathtt{P}^{\mathtt{false}} \textbf{ end}$$
iff there exist $(\mathfrak{D}^{\mathtt{true}},\mathfrak{X}^{\mathtt{true}})$ and $(\mathfrak{D}^{\mathtt{false}},\mathfrak{X}^{\mathtt{false}})$ such that:
$(\mathfrak{D}^{\mathtt{true}},\mathfrak{X}^{\mathtt{true}}) \models \mathtt{P}^{\mathtt{true}} \quad \wedge \quad (\mathfrak{D}^{\mathtt{false}},\mathfrak{X}^{\mathtt{false}}) \models \mathtt{P}^{\mathtt{false}}$
$\mathfrak{X} \supseteq (\mathfrak{X}^{\mathtt{true}} \cup \mathfrak{X}^{\mathtt{false}}) \quad \wedge \quad \mathfrak{D} \supseteq \left(\mathfrak{D}^{\mathtt{true}} \cup \mathfrak{D}^{\mathtt{false}} \cup (\mathfrak{X} \times \mathcal{V}(e))\right)$

$$(\mathfrak{D},\mathfrak{X}) \models \textbf{while } e \textbf{ do } \mathtt{P}^{\mathrm{l}} \textbf{ done}$$
iff there exists $(\mathfrak{D}^{\mathrm{l}},\mathfrak{X}^{\mathrm{l}})$ such that:$(\mathfrak{D}^{\mathrm{l}},\mathfrak{X}^{\mathrm{l}}) \models \mathtt{P}^{\mathrm{l}}$ and
$\mathfrak{D} \supseteq \left((\mathfrak{D}^{\mathrm{l}} \circ \mathfrak{D}) \cup \mathcal{I}d \cup (\mathfrak{X} \times \mathcal{V}(e))\right) \quad \wedge \quad \mathfrak{X} \supseteq \mathfrak{X}^{\mathrm{l}}$

Fig. 5. Acceptability rules for *usable* analysis results

Proof. As $\mathfrak{X}$ contains an over-approximation of variables on the left side of every assignments in P, a variable which is not in this set can not be assigned to. And therefore, its value remains unchanged.

Theorem 2 (Acceptable imply sound detection of dependencies). *For all programs P, and analysis result* $(\mathfrak{D},\mathfrak{X})$ *if* $(\mathfrak{D},\mathfrak{X}) \models P$ *then the Hypothesis 3 holds.*

Proof. The proof follows directly from the acceptability rules. The value of every assigned variables depends on the values of the variables appearing in the expression on the right side of the assignment. The rule for sequences links the dependencies created by both statements. Variables whose value can be modified in a conditional are accurately stated to depend on the values of variables appearing in the branching condition. And finally, in the rule for `while` statements, $\mathfrak{D}^{\mathrm{l}} \circ \mathfrak{D} \subseteq \mathfrak{D}$ ensures that the dependencies created by one or more iterations of the loop are contained in $\mathfrak{D}$. While $\mathcal{I}d \subseteq \mathfrak{D}$ ensures that dependencies existing in case of no iteration at all are also contained in $\mathfrak{D}$.

Therefore, a static information flow analysis, which satisfies Hypothesis 4 and whose results are acceptable ($[\![\rho \vdash \mathtt{P}]\!]^{\sharp_{\mathcal{G}}} \models \mathtt{P}$), is *usable* by the analyzing semantics — i.e. it satisfies Hypotheses 2, 3 and 4.

3.4 Properties of the Analyzing Semantics

Section 3.1 formally defined the dynamic information flow analysis proposed in this article. The soundness of this analysis with regard to the notion of noninterfering execution (Definition 2) is proved by Theorem 3. This means that, at the end of any two executions of a given program P started with the same public inputs (variables which do not belong to $\mathcal{S}(\mathtt{P})$), any variables whose final tag is $\bot$ has the same final value for both executions. Theorem 4 states that the dynamic analysis results for two executions following the same path in the acyclic CFG are identical. Therefore, the dynamic information flow analysis proposed can be used with the testing technique presented in Section 2.2.

Theorem 3 (Sound Detection of Information flows).
Assume that the analyzing semantics $\Downarrow_{\mathcal{A}}$ uses a static analysis ($[\![]\!]^{\sharp_{\mathcal{G}}}$) for which Hypotheses 2, 3 and 4 hold. For all programs P, sets of variables X_1 and X_2, and execution states $(\sigma_1,\ \rho_1)$, $(\sigma'_1,\ \rho'_1)$, $(\sigma_2,\ \rho_2)$ and $(\sigma'_2,\ \rho'_2)$ such that:

1. $\forall x \in \mathcal{S}(P).\ \rho_1(x) = \top$,
2. $\forall x \notin \mathcal{S}(P).\ \sigma_1(x) = \sigma_2(x)$,
3. $(\sigma_1,\ \rho_1) \vdash P \Downarrow_{\mathcal{A}} (\sigma'_1,\ \rho'_1) : X_1$,
4. $(\sigma_2,\ \rho_2) \vdash P \Downarrow_{\mathcal{A}} (\sigma'_2,\ \rho'_2) : X_2$,

the following holds: $\forall x \in \mathbb{X}.\quad (\rho'_1(x) = \bot)\ \Rightarrow\ (\sigma'_1(x) = \sigma'_2(x))$.

Proof. The proof goes by induction on the derivation tree of the third local hypothesis and by cases on the last evaluation rule used. For inductions, the set $\mathcal{S}(P)$ is replaced by the set of variables whose tag is $\top$, and the second local hypothesis (with $\mathcal{S}(P)$ replaced) is proved to be an invariant. The proof is straightforward for **skip** *and assignments, and goes by simple induction for sequences. For conditionals, if both executions execute the same branch then the conclusion follows from a simple induction. Otherwise, it means that the expression tested (e) does not have the same value for both execution; and therefore that $\rho_1(e)$ is $\top$. Hence, as any variables which are modified in the branch executed (X) or potentially modified by an execution of the other branch ($\mathfrak{X}$) receive the tag of e in the final tag store (ρ'_1), the desired conclusion is vacuously true for variables assigned by any of the two executions and follows directly from the second local hypothesis for the other variables.*

Theorem 4 (Identical Same Path Analysis Results).
If the analyzing semantics $\Downarrow_{\mathcal{A}}$ uses a static analysis ($[\![]\!]^{\sharp_{\mathcal{G}}}$) for which Hypotheses 2, 3 and 4 hold then Hypothesis 1 holds.

Proof. Once again, the proof goes by induction and cases on the derivation tree. The proof is direct for **skip**, *assignments and sequences. For* `if` *statements, as the trace is the same then the branch executed is the same and the proof follows by induction. For* `while` *statements, the final tag store is constructed from the result of the static analysis of the statement. Therefore, the conclusion follows directly from Hypothesis 4.*

4 Conclusion

To the best of the author knowledge, this article proposes the first information flow testing mechanism which enjoys the property of being sound with regard to noninterference. It is based on a special semantics integrating a dynamic information flow analysis which is sound from the point of view of noninterference for the tested execution, and returns the same sound result for any execution following the same path in the acyclic Control Flow Graph (aCFG) of the program. After testing once every path in the aCFG, a sound conclusion with regard to noninterference can be stated for the program under test. The dynamic analysis combines information obtained from executed statements with static analysis results of some unexecuted pieces of code. No particular static analysis is required to be used. Instead, three hypotheses on the results of the static analysis used are defined. It is proved that any static analysis respecting those hypotheses can be soundly used. A construction mechanism to obtain such a static analysis from existing noninterference type systems is given. Additionally, a set of constraints relating statements and the result of their static analysis is defined independently from anything else. This set of constraints is proved to subsume the three hypotheses stated before.

Given test cases covering all the feasible paths in the aCFG, the testing mechanism proposed returns a conclusion as strong as the conclusion returned by the static analysis used by the testing semantics. Moreover, this result is at least as precise as the result returned by the static analysis alone. The increase in precision is proportional to the number of `if` statements whose condition is not influenced by a secret.

To the author knowledge, there is no similar work. The vast majority of research on noninterference concerns static analyses and involves type systems (4). Some "real size" languages together with security type system have been developed (for example, JFlow/JIF (15) and FlowCaml (16)). A few dynamic information flow analyses have been proposed (5; 7; 17). However, those analyses are applied on final users executions and are therefore required to correct "bad" flows. In order to prevent this correction mechanism to become a new covert channel, additional constraints are applied on the dynamic analysis (8). Those additional constraints limit the precision achievable by such dynamic analyses. While testing, there is no need for a correction mechanism and therefore a higher precision can be achieved.

Noninterference testing is an interesting field of study having its own specific challenges. It may be hard, so not impossible (14), to come out with a valid set of executions in order to cover all feasible paths in the aCFG. However, in many cases, the noninterference mechanism proposed in this article is more precise than the static analyses which can be used by the testing technique proposed. Thus, noninterference testing may allow to validate some specific programs whose validation is out of reach of static analyses, or at least help find information flow bugs.

Bibliography

[1] Ashby, W.R.: An Introduction to Cybernetics. Chapman & Hall (1956)

[2] Goguen, J.A., Meseguer, J.: Security policies and security models. In: Proc. Symp. on Security and Privacy, IEEE Computer Society (1982) 11–20

[3] Cohen, E.S.: Information transmission in computational systems. ACM SIGOPS Operating Systems Review **11**(5) (1977) 133–139

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

[5] Vachharajani, N., Bridges, M.J., Chang, J., Rangan, R., Ottoni, G., Blome, J.A., Reis, G.A., Vachharajani, M., August, D.I.: Rifle: An architectural framework for user-centric information-flow security. In: Proceedings of the International Symposium on Microarchitecture (2004)

[6] Le Guernic, G., Banerjee, A., Jensen, T., Schmidt, D.: Automata-based Confidentiality Monitoring. In: Proc. Asian Computing Science Conference. LNCS (2006)

[7] Shroff, P., Smith, S.F., Thober, M.: Dynamic dependency monitoring to secure information flow. In: Proc. Computer Security Foundations Symposium, IEEE Computer Society (2007)

[8] Le Guernic, G., Jensen, T.: Monitoring Information Flow. In: Proc. Workshop on Foundations of Computer Security, DePaul University (2005) 19–30

[9] Ntafos, S.C.: A comparison of some structural testing strategies. IEEE Transactions on Software Engineering **14**(6) (1988) 868–874

[10] Beizer, B.: Software Testing Techniques. International Thomson Computer Press (1990)

[11] Williams, N., Marre, B., Mouy, P., Muriel, R.: Pathcrawler: Automatic generation of path tests by combining static and dynamic analysis. In: Proc. European Dependable Computing Conference. Volume 3463 of LNCS (2005) 281–292

[12] Godefroid, P., Klarlund, N., Sen, K.: DART: Directed Automated Random Testing. In: Proc. Programming Language Design and Implementation. Volume 40 of ACM SIGPLAN Notices. (2005) 213–223

[13] Sen, K., Agha, G.: Cute and JCute : Concolic unit testing and explicit path model-checking tools. In: Proc. Computer Aided Verification. Volume 4144 of LNCS (2006) 419–423

[14] Gupta, N., Mathur, A.P., Soffa, M.L.: Automated Test Data Generation Using an Iterative Relaxation Method. In: Proc. Symposium on Foundations of Software Engineering, ACM Press (1998) 231–244

[15] Myers, A.C.: JFlow: Practical mostly-static information flow control. In: Proc. Symp. on Principles of Programming Languages. (1999) 228–241

[16] Pottier, F., Simonet, V.: Information flow inference for ML. ACM Trans. on Programming Languages and Systems **25**(1) (2003) 117–158

[17] Le Guernic, G.: Automaton-based Confidentiality Monitoring of Concurrent Programs. In: Proc. Computer Security Foundations Symposium (2007)