Precise Dynamic Verification of Confidentiality

Gurvan Le Guernic

INRIA-MSR - Parc Orsay Universit, 91893 Orsay - France
http://www.msr-inria.inria.fr/~gleguern/
gleguern@gmail.com

Abstract. Confidentiality is maybe the most popular security property to be formally or informally verified. Noninterference is a baseline security policy to formalize confidentiality of secret information manipulated by a program. Many static analyses have been developed for the verification of noninterference. In contrast to those static analyses, this paper considers the run-time verification of the respect of confidentiality by a single execution of a program. It proposes a dynamic noninterference analysis for sequential programs based on a combination of dynamic and static analyses. The static analysis is used to analyze some unexecuted pieces of code in order to take into account all types of flows. The static analysis is sensitive to the current program state. This sensitivity allows the overall dynamic analysis to be more precise than previous work. The soundness of the overall dynamic noninterference analysis with regard to confidentiality breaches detection and correction is proved.

1 Introduction

Language-based security is an active field of research. The majority of work on confidentiality in this field focuses on static analyses [15]. Recent years have seen a resurgence of dynamic analyses aiming at enforcing confidentiality at run time [5, 8, 16, 21]. The first reason is that nowadays it is nearly impossible for consumers to prevent the execution of "bad" code on their devices — for example, in September 2007 cybercriminals introduced malicious scripts which were executed by any browser visiting webpages of a US Consulate [9]. Moreover, there are two main potential advantages of dynamic analyses over static analyses [8]. The first one is the increased knowledge of the execution environment and behavior at run time, including the knowledge of the precise control flow followed by the current execution. This increased knowledge allows the dynamic analysis to be more precise than a static analysis in some cases; as, for example, with the program on page 93. The second advantage lies in the ability of sound information flow monitors to run some "safe" executions of an "unsafe" program while still guarantying the confidentiality of secret data. In order to take into account all indirect flows (flows originating in control statements) dynamic analyses rely on static analyses of some, but not all, unexecuted pieces of code.

This paper proposes to increase the precision of such dynamic information flow analyses. This is done by taking advantage, at the static analysis level, of the dynamic nature of the overall analysis. To do so, when statically analyzing an unexecuted piece of code, the current program state is taken into account in order to reduce the program space to analyze. The following piece of code is a

motivating example for this work. It corresponds to the body of the main loop of an Instant Messaging (IM) program. This one has the appealing "movie-like" feature of displaying messages characters by characters as they are typed.

```
1 c := getCharFromKeyboard();
2 tmp := tmp + ((int) c);
3 if ( tmp > ((int) userSecretKey) ) {
4   tmp := 0;
5   if (to = "sexyPirate") {c := specialChar}
6 };
7 send(to, c)
```

This IM program is a malware developed by "sexyPirate". When someone uses this software to communicate with a user other than sexyPirate, everything goes as expected and no secret is revealed. However, if a user communicates with sexyPirate using this IM then information about the user's secret key is leaked to the pirate. When the integer value of the characters typed by the user since the last time *tmp* has been reset to 0 reaches the integer value of the user's secret key, a special character (that sexyPirate is able to distinguish) appears on the pirate's screen. Therefore, by iterating the process, sexyPirate is able to get an accurate approximation of the user's secret key. Any sound static analysis would reject this program; and therefore, all its executions. One of the advantages of dynamic information flow analysis, if it is precise enough, is to allow use of this program for communicating with users other than sexyPirate, while still guarantying the confidentiality of the secret key in any case. However, none of the previous work are precise enough. When statically analyzing lines 4 and 5, no knowledge about the value of the variable *to* is taken into account. Therefore, the overall dynamic analysis will always consider that the value of *c* may be modified; which implies a flow from *userSecretKey* to *c*. With such dynamic analyses, line 7 must then be corrected in order to prevent any potential leakage of the value of the secret key to the outside world. In the work proposed in this paper, the static analysis used for lines 4 and 5 takes into account the run time value of the variable *to*. This allows the overall dynamic analysis proposed to detect that there is no flow from *userSecretKey* to *c* whenever *to* is different from `sexyPirate`. Therefore, it allows one to use this IM program for communicating with any user different from sexyPirate; while still preserving the confidentiality of the user's secret key even when trying to use this program to communicate with sexyPirate if the correction mechanism is applied carefully [4, 7].

The next section defines various notions used in this paper and introduces the principles of the dynamic analysis proposed in this paper. Section 3 formalizes the dynamic information flow analysis whose main properties are exposed in Sect. 4. Before presenting related works in Sect. 6, the main benefits of dynamic information flow analyses are exposed in Sect. 5. Finally, Sect. 7 concludes.

2 Definitions and Principles

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.

At any execution step, a variable or expression is said to *carry variety* [2, Sect.1] if its value is not completely constrained by the public inputs of the program. In other words, a variable or expression *carries variety* if its value is influenced by the private inputs; therefore, if it may have a different value at this given execution step if the values of the private inputs were different.

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 execution of a given program `P`, two sets of variable identifiers are defined. The set of variables corresponding to the secret inputs of the program is designated by $\mathcal{S}(\mathtt{P})$. The set of variables whose final value are publicly observable at the end of the execution is designated by $\mathcal{O}(\mathtt{P})$. No requirements are put on $\mathcal{S}(\mathtt{P})$ and $\mathcal{O}(\mathtt{P})$ other than requiring them to be subsets of $\mathbb{X}$ (the domain of variables). 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 (X) and value stores (σ).

Definition 1 (V-Equivalent States).
Let V be a set of variables. Two program states X_1 and X_2, containing the value stores σ_1 and σ_2 respectively, are V-equivalent *with regards to a set of variables V, written $X_1 \overset{V}{=} X_2$, if and only if the value of any variable belonging to V is the same in σ_1 and σ_2:*

$$X_1 \overset{V}{=} X_2 \quad \Longleftrightarrow \quad \forall x \in V : \sigma_1(x) = \sigma_2(x)$$

Definition 1 states a formal relation among program states. This relation defines equivalence classes of program states with regard to a given set of variables. If two program states are V-equivalent, it means that it is impossible to distinguish them solely by looking at the value of the variables belonging to the set

V. This relation is used to define the confidentiality property which is verified by the dynamic analysis presented in this paper.

Definition 2 (Noninterfering Execution).
Let $\Downarrow_s$ denote a big-step semantics. Let $\overline{\mathcal{S}(\mathtt{P})}$ be the complement of $\mathcal{S}(\mathtt{P})$ in the set $\mathbb{X}$. For all programs $\mathtt{P}$, program states X_1 and X_1', an execution with the semantics $\Downarrow_s$ of the program $\mathtt{P}$ in the initial state X_1 and yielding the final state X_1' is noninterfering, written $ni(\mathtt{P}, s, X_1)$, if and only if, for every program states X_2 and X_2' such that the execution with the semantics $\Downarrow_s$ of the program $\mathtt{P}$ in the initial state X_2 yields the final state X_2':

$$X_1 \overset{\overline{\mathcal{S}(\mathtt{P})}}{=} X_2 \;\Rightarrow\; X_1' \overset{\mathcal{O}(\mathtt{P})}{=} X_2'$$

Definition 2 states that an execution is safe — i.e. it has the desired confidentiality property — if any other execution started with the same public (non-secret) values yields a final program state which is $\mathcal{O}(\mathtt{P})$-equivalent to the final program state of the execution analyzed. It means that, by looking only at the final values of the variables observable at the end of the execution, it is impossible to distinguish this execution from any other execution whose initial program state differs only in the values of the secret inputs. Therefore, for such an execution, it is impossible to deduce information about the secret inputs of the program by looking solely at the values of the publicly observable outputs.

The dynamic analysis is based on a flow and state sensitive static analysis. During the execution, every variable is associated a tag which reflects the fact that the variable *may* or *may not* carry variety — i.e. *may* or *may not* be influenced by the secret inputs of the program. A tag store in the program state keeps track of those associations. The dynamic analysis 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.

The static analysis is context sensitive. An unexecuted branch `P` is analyzed in the context of the program state at the time the test of the conditional, to which `P` belongs, has been evaluated. The static analysis is then aware of the exact value of the variables which do not carry variety. During the analysis, the context (value store and tag store used for the analysis) is modified to reflect loss of knowledge (in fact, only the tag store is modified). The static analysis does not compute the values of variables. Therefore, when analyzing an assignment to a variable x, the context of the static analysis is modified to reflect the fact that the static analysis does not anymore have knowledge of the precise value of the variable x. When analyzing a conditional whose test value can be computed in the current context (using only the values of the variables whose tag is $\bot$), only the branch designated by the test is analyzed. As the value of any variable which does not carry variety depends only on the public inputs, branches which are not

designated by the test value would never be executed by any execution started with the same public inputs as the monitored execution. Implicit indirect flows and explicit indirect flows must be treated with the same precision in order to prevent the creation of a new covert channel [7]. This particular point is discussed in Sect. 4. As the static analysis detects implicit indirect flows more accurately than context insensitive analyses, explicit indirect flows can also be treated more accurately.

The next section formalizes the mechanisms presented above. It presents a monitoring semantics incorporating a dynamic noninterference analysis.

3 The Monitoring Semantics

The dynamic information flow analysis and the monitoring semantics are defined together in Fig. 1. An example of the behavior of this analysis on a given execution is presented in the companion technical report [6]. Information flows are tracked using tags. At any execution step, every variable has a tag which reflects whether this variable may carry variety or not. The static analysis used for the analysis of some unexecuted branches is characterized in Fig. 2.

The language studied is an imperative language for sequential programs whose grammar follows. In this grammar, $\langle ident \rangle$ stands for a variable identifier. $\langle expr \rangle$ is an expression of values and variable identifiers. 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.

$$
\begin{aligned}
\langle prog \rangle ::= \ & \textbf{skip} \\
| \ & \langle ident \rangle \ \textbf{:=} \ \langle expr \rangle \\
| \ & \langle prog \rangle \ ; \ \langle prog \rangle \\
| \ & \textbf{if} \ \langle expr \rangle \ \textbf{then} \ \langle prog \rangle \ \textbf{else} \ \langle prog \rangle \ \textbf{end} \\
| \ & \textbf{while} \ \langle expr \rangle \ \textbf{do} \ \langle prog \rangle \ \textbf{done}
\end{aligned}
$$

A program expressed with this language is either a skip statement (**skip**) which has no effect, an assignment of the value of an expression to a variable, a sequence of programs ($\langle prog \rangle$; $\langle prog \rangle$), a conditional executing one program — out of two — depending on the value of a given expression (`if` statements), or a loop executing repetitively a given program as long as a given expression is true (`while` statements).

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 monitoring semantics described in Fig. 1 is presented as standard inference rules for sequents written in the format:

$$\zeta,\ t^{\mathrm{pc}} \ \vdash\ \mathtt{P}\ \Downarrow_{\mathcal{M}}\ \zeta'$$

This reads as follows: in the monitoring execution state ζ, with a program counter tag equal to t^{pc}, program P yields the monitoring execution state ζ'. The program counter tag (t^{pc}) is a tag which reflects the security level of the information carried by the control flow. A monitoring 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 $FV(e)$ being the set of free variables appearing in the expression e:

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

The semantics rules make use of static analyses results. In Fig. 1, application of a static information flow analysis to the piece of code P in the context ζ is written: $[\![\zeta \vdash \mathtt{P}]\!]^{\sharp_{\mathcal{G}}}$. The analysis of a program P in a monitoring execution state ζ must return a subset of $\mathbb{X}$. This set, usually written $\mathfrak{X}$, is an over-approximation of the set of variables which are potentially defined in an execution of P in the context ζ. This static information flow analysis can be any such analysis that satisfies a set of formal constraints which are stated in Sect. 3.2.

The monitoring semantics rules are straightforward. As can be expected, the execution of a **skip** statement with the semantics given in Fig. 1 yields a final state equal to the initial state. The monitored execution of the assignment of the value of the expression e to the variable x yields a monitored execution state $(\sigma',\ \rho')$. The final value store (σ') is equal to the initial value store (σ) except for the variable 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 the least upper bound of the program counter tag (t^{pc}) and the tag of the expression computed using the initial tag store ($\rho(e)$). $\rho(e)$ corresponds to the level of the information flowing into x through direct flows. t^{pc} corresponds to the level of the information flowing into x through explicit indirect flows.

$$\frac{}{\zeta,\ t^{\mathrm{pc}}\ \vdash\ \mathbf{skip}\ \Downarrow_{\mathcal{M}}\ \zeta}$$

$$\frac{}{(\sigma,\ \rho),\ t^{\mathrm{pc}}\ \vdash\ x := e\ \Downarrow_{\mathcal{M}}\ (\sigma[x \mapsto \sigma(e)],\ \rho[x \mapsto \rho(e) \sqcup t^{\mathrm{pc}}])}$$

$$\frac{\zeta,\ t^{\mathrm{pc}}\ \vdash\ P^{\mathrm{h}}\ \Downarrow_{\mathcal{M}}\ \zeta^{\mathrm{h}} \qquad \zeta^{\mathrm{h}},\ t^{\mathrm{pc}}\ \vdash\ P^{\mathrm{t}}\ \Downarrow_{\mathcal{M}}\ \zeta'}{\zeta,\ t^{\mathrm{pc}}\ \vdash\ P^{\mathrm{h}}\ ;\ P^{\mathrm{t}}\ \Downarrow_{\mathcal{M}}\ \zeta'}$$

$$\frac{\rho(e) = \bot \qquad \sigma(e) = v \qquad (\sigma,\ \rho),\ t^{\mathrm{pc}} \sqcup \bot\ \vdash\ P^{v}\ \Downarrow_{\mathcal{M}}\ \zeta'}{(\sigma,\ \rho),\ t^{\mathrm{pc}}\ \vdash\ \mathbf{if}\ e\ \mathbf{then}\ P^{\mathtt{true}}\ \mathbf{else}\ P^{\mathtt{false}}\ \mathbf{end}\ \Downarrow_{\mathcal{M}}\ \zeta'}$$

$$\frac{\begin{array}{c}\rho(e) = \top \qquad \sigma(e) = v \qquad (\sigma,\ \rho),\ t^{\mathrm{pc}} \sqcup \top\ \vdash\ P^{v}\ \Downarrow_{\mathcal{M}}\ (\sigma^{v},\ \rho^{v}) \\ [\![(\sigma,\ \rho) \vdash P^{\neg v}]\!]^{\sharp_{\mathcal{G}}} = \mathfrak{X} \qquad \rho^{\mathrm{e}} = (\mathfrak{X} \times \{\top\}) \cup (\overline{\mathfrak{X}} \times \{\bot\})\end{array}}{(\sigma,\ \rho),\ t^{\mathrm{pc}}\ \vdash\ \mathbf{if}\ e\ \mathbf{then}\ P^{\mathtt{true}}\ \mathbf{else}\ P^{\mathtt{false}}\ \mathbf{end}\ \Downarrow_{\mathcal{M}}\ (\sigma^{v},\ \rho^{v} \sqcup \rho^{\mathrm{e}})}$$

$$\frac{\rho(e) = \bot \qquad \sigma(e) = \mathtt{false}}{(\sigma,\ \rho),\ t^{\mathrm{pc}}\ \vdash\ \mathbf{while}\ e\ \mathbf{do}\ P^{\mathrm{l}}\ \mathbf{done}\ \Downarrow_{\mathcal{M}}\ (\sigma,\ \rho)}$$

$$\frac{\rho(e) = \bot \qquad \sigma(e) = \mathtt{true}}{(\sigma,\ \rho),\ t^{\mathrm{pc}} \sqcup \bot\ \vdash\ P^{\mathrm{l}}\ ;\ \mathbf{while}\ e\ \mathbf{do}\ P^{\mathrm{l}}\ \mathbf{done}\ \Downarrow_{\mathcal{M}}\ \zeta'}{(\sigma,\ \rho),\ t^{\mathrm{pc}}\ \vdash\ \mathbf{while}\ e\ \mathbf{do}\ P^{\mathrm{l}}\ \mathbf{done}\ \Downarrow_{\mathcal{M}}\ \zeta'}$$

$$\frac{\rho(e) = \top \qquad \sigma(e) = \mathtt{true}}{(\sigma,\ \rho),\ t^{\mathrm{pc}} \sqcup \top\ \vdash\ P^{\mathrm{l}}\ ;\ \mathbf{while}\ e\ \mathbf{do}\ P^{\mathrm{l}}\ \mathbf{done}\ \Downarrow_{\mathcal{M}}\ (\sigma',\ \rho^{\mathrm{e}})}{(\sigma,\ \rho),\ t^{\mathrm{pc}}\ \vdash\ \mathbf{while}\ e\ \mathbf{do}\ P^{\mathrm{l}}\ \mathbf{done}\ \Downarrow_{\mathcal{M}}\ (\sigma',\ \rho \sqcup \rho^{\mathrm{e}})}$$

$$\frac{\begin{array}{c}\rho(e) = \top \qquad \sigma(e) = \mathtt{false} \\ [\![(\sigma,\ \rho) \vdash P^{\mathrm{l}}\ ;\ \mathbf{while}\ e\ \mathbf{do}\ P^{\mathrm{l}}\ \mathbf{done}]\!]^{\sharp_{\mathcal{G}}} = \mathfrak{X} \\ \rho^{\mathrm{e}} = (\mathfrak{X} \times \{\top\}) \cup (\overline{\mathfrak{X}} \times \{\bot\})\end{array}}{(\sigma,\ \rho),\ t^{\mathrm{pc}}\ \vdash\ \mathbf{while}\ e\ \mathbf{do}\ P^{\mathrm{l}}\ \mathbf{done}\ \Downarrow_{\mathcal{M}}\ (\sigma,\ \rho \sqcup \rho^{\mathrm{e}})}$$

Fig. 1. Rules of the monitoring semantics

The monitored execution of a conditional whose test expression does not carry variety ($\rho(e) = \bot$) follows the same scheme as with a standard semantics. For a conditional whose test expression e carries variety, the branch (P^{v}) designated by the value of e (v) 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 new tag store (ρ^{e}) generated from the result of the analysis of $P^{\neg v}$ ($\mathfrak{X}$). By definition, $\rho \sqcup \rho'$ is equal to $\lambda x.\rho(x) \sqcup \rho'(x)$. The new tag store ($\rho^{e}$) reflects the implicit indirect flows between the value of the test of the conditional and the variables ($\mathfrak{X}$) which may be defined in an execution of $P^{\neg v}$. In ρ^{e}, the tag of

a variable x is equal to the initial tag of the test expression of the conditional ($\rho(e)$) if and only if x belongs to $\mathfrak{X}$; otherwise, its tag is $\bot$.

3.2 The Static Analysis Used

Fig. 2 defines some constraints characterizing a set of static analyses which can be used by the dynamic noninterference analysis. The result $\mathfrak{X}$ of a static analysis of a given program ($\mathtt{P}$) in a given context (ζ) is *acceptable* for the dynamic analysis only if the result satisfies those rules. This is written in the format: $\mathfrak{X} \models (\zeta \vdash \mathtt{P})$. In the definitions of those rules, $\{\![S^{\mathtt{true}}, S^{\mathtt{false}}]\!\}^t_v$ returns either the set $S^{\mathtt{true}}$, the set $S^{\mathtt{false}}$ or the union of both depending on the tag t and the boolean v. Its formal definition follows.

$$\{\![S^{\mathtt{true}}, S^{\mathtt{false}}]\!\}^t_v = \begin{cases} S^{\mathtt{true}} \cup S^{\mathtt{false}} & \text{iff } t = \top \\ S^v & \text{iff } t = \bot \end{cases}$$

$\emptyset \models (\ (\sigma,\ \rho) \vdash \textbf{skip}\ )$

$\{x\} \models (\ (\sigma,\ \rho) \vdash x := e\ )$

$\mathfrak{X} \models \Big(\ (\sigma,\ \rho) \vdash \mathtt{P}^{\mathrm{h}}\ ;\ \mathtt{P}^{\mathrm{t}}\ \Big)$
iff there exist $\mathfrak{X}^{\mathrm{h}}$ and $\mathfrak{X}^{\mathrm{t}}$ such that:
$\mathfrak{X}^{\mathrm{h}} \models ((\sigma,\ \rho) \vdash \mathtt{P}^{\mathrm{h}})$
let $\rho' = \rho \sqcup ((\mathfrak{X}^{\mathrm{h}} \times \top) \cup (\overline{\mathfrak{X}^{\mathrm{h}}} \times \bot))$ in $\mathfrak{X}^{\mathrm{t}} \models ((\sigma, \rho') \vdash \mathtt{P}^{\mathrm{t}})$
$\mathfrak{X} = \mathfrak{X}^{\mathrm{h}} \cup \mathfrak{X}^{\mathrm{t}}$

$\mathfrak{X} \models (\ (\sigma,\ \rho) \vdash \textbf{if}\ e\ \textbf{then}\ \mathtt{P}^{\mathtt{true}}\ \textbf{else}\ \mathtt{P}^{\mathtt{false}}\ \textbf{end}\ )$
iff there exist $\mathfrak{X}^{\mathtt{true}}$ and $\mathfrak{X}^{\mathtt{false}}$ such that:
$\mathfrak{X}^{\mathtt{true}} \models ((\sigma,\ \rho) \vdash \mathtt{P}^{\mathtt{true}})$
$\mathfrak{X}^{\mathtt{false}} \models ((\sigma,\ \rho) \vdash \mathtt{P}^{\mathtt{false}})$
$\mathfrak{X} = \{\![\mathfrak{X}^{\mathtt{true}}, \mathfrak{X}^{\mathtt{false}}]\!\}^{\rho(e)}_{\sigma(e)}$

$\mathfrak{X} \models \Big(\ (\sigma,\ \rho) \vdash \textbf{while}\ e\ \textbf{do}\ \mathtt{P}^{\mathrm{l}}\ \textbf{done}\ \Big)$
iff there exists $\mathfrak{X}^{\mathrm{l}}$ such that:
let $\rho' = \rho \sqcup ((\mathfrak{X}^{\mathrm{l}} \times \top) \cup (\overline{\mathfrak{X}^{\mathrm{l}}} \times \bot))$ in $\mathfrak{X}^{\mathrm{l}} \models ((\sigma, \rho') \vdash \mathtt{P}^{\mathrm{l}})$
$\mathfrak{X} = \{\![\mathfrak{X}^{\mathrm{l}}, \emptyset]\!\}^{\rho(e)}_{\sigma(e)}$

Fig. 2. Constraints on the static analysis results

4 Properties of the Monitoring Semantics

Section 3 formally defines the dynamic information flow analysis proposed in this article. In the current section, this dynamic noninterference analysis is proved to be sound with regard to the enforcement of the notion of noninterfering execution given in Definition 2. This means that, any monitor enforcing noninterference using this dynamic analysis would be able to ensure that: for any two monitored executions of a given program P started with the same public inputs (variables which do not belong to $\mathcal{S}(\mathtt{P})$), the final values of observable outputs (variables which belong to $\mathcal{O}(\mathtt{P})$) are the same for both executions.

Theorem 1 proves that the dynamic analysis is sound with regard to information flow *detection*. Any variable, whose tag at the end of the execution is $\bot$, has the same final value for any executions started with the same public inputs.

Theorem 1 (Detection Soundness).
For all programs $\mathtt{P}$ *and states* $(\sigma_1,\ \rho_1)$, $(\sigma'_1,\ \rho'_1)$, $(\sigma_2,\ \rho_2)$ *and* $(\sigma'_2,\ \rho'_2)$*:*

$$\left.\begin{array}{l}(\sigma_1,\ \rho_1),\ \bot\ \vdash\ \mathtt{P}\ \Downarrow_{\mathcal{M}}\ (\sigma'_1,\ \rho'_1)\\ (\sigma_2,\ \rho_2),\ \bot\ \vdash\ \mathtt{P}\ \Downarrow_{\mathcal{M}}\ (\sigma'_2,\ \rho'_2)\\ \forall x \notin \mathcal{S}(\mathtt{P}).\ \sigma_1(x) = \sigma_2(x)\\ \forall x \in \mathcal{S}(\mathtt{P}).\ \rho_1(x) = \top\end{array}\right\} \Rightarrow\ \Big(\forall x.\ (\rho'_1(x) = \bot)\ \Rightarrow\ (\sigma'_1(x) = \sigma'_2(x))\Big)$$

Proof (Proof summary). The detailed formal proof can be found in the companion technical report [6]. The proof aims at showing that the invariant *which relates the fact, for every variable, of having a* $\bot$ *tag and not carrying variety — i.e. not being influenced by the secret inputs — is preserved during the execution. The invariant preservation is obvious for* **skip** *statements. If the tag of a variable* x *after an assignment of* e *to* x *is* $\bot$*, then it means, first, that the control flow does not carry variety* ($t^{pc} = \bot$) *and therefore that, with similar public inputs, the assignment will always be executed. It also means that the expression does not carry variety and the invariant property is preserved by the execution of assignments. For sequence statements, if the invariant is preserved by the first and second statements then it is preserved by the sequential execution of both statements. If the tag of the condition of a branching statement is* $\bot$*, then any execution started with the same public inputs evaluates the same branch as the execution monitored. As, by induction, the execution of the branch preserves the invariant, the invariant is also preserved. If the condition's tag is* $\top$*, as the tag of the control flow* (t^{pc}) *is updated to* $\top$*, all the variables assigned to in the branch have a tag of* $\top$*. Additionally, any variable which may have been assigned to in the other branch are in the set returned by the static analysis, therefore their tag also becomes*$\top$*. Hence, the invariant relating* $\bot$ *and not carrying variety is preserved by the execution of a branching statement. If the tag of the condition of a loop is* $\bot$ *then, if its value is false the loop is equivalent to a* **skip** *and the*

invariant is preserved. If the tag of a true condition is $\bot$, the proof follows by induction. If the condition's tag is $\top$ then, for the same reasons as the case for branching statements, the tag of all the variables whose value may be modified by the loop becomes $\top$. Therefore, the invariant property is preserved by loops.

More informally, theorem 1 compares any 2 executions which are such that: any public inputs ($x \notin \mathcal{S}(\mathtt{P})$) have the same initial value, and any secret input have an initial tag of $\top$ in order to let the dynamic analysis know that their content is secret. Theorem 1 states that if the final tag of a variable is $\bot$ — therefore, that the analysis considers its value to be safely accessible — then the final values of this variable for both executions — any 2 executions which have the same public inputs — are the same. Therefore, an attacker, who looks at the final value of a variable whose final tag is $\bot$, sees the same value for all the executions which have different secret inputs but the same public inputs. Hence, an attacker does not learn anything about the secret inputs by observing the final values of variables whose final tag is $\bot$. Therefore, to sanitize the final state of an execution, it is sufficient to "securely" (as will be explained in the following discussion) reset the value of observable variables ($\mathcal{O}(\mathtt{P})$) whose final tag is $\top$.

As shown by Le Guernic and Jensen [7], if the correction of "bad" information flows is done without enough care, the correction mechanism itself can become a new covert channel carrying secret information. Indeed, theorem 1 states that the final value of a variable, whose final tag is $\bot$, is not influenced by the value of the secret inputs. However, if the final tag of a variable is influenced by the values of the secret inputs, then it means that for some secret input values the final tag of this variable will be $\bot$ and for other secret input values it will be $\top$. Hence, it means that, for some secret input values, the correction mechanism will reset the final value of the variable; and for other secret input values, the final value of the variable will not be reset. Therefore, by checking if the final value of this variable has been reset or not, an attacker can learn information about the secret input values. Theorem 2 proves that, for the analysis presented in this paper, the final tag of a variable does not depend on the secret inputs of the program. Therefore, any variable belonging to $\mathcal{O}(\mathtt{P})$ whose final tag is not $\bot$ can safely be reset to a default value without creating a new covert channel.

Theorem 2 (Correction Soundness).
For all programs $\mathtt{P}$ and states $(\sigma_1,\ \rho_1)$, $(\sigma'_1,\ \rho'_1)$, $(\sigma_2,\ \rho_2)$ and $(\sigma'_2,\ \rho'_2)$:

$$\left.\begin{array}{l}(\sigma_1,\ \rho_1),\ \bot\ \vdash\ \mathtt{P}\ \Downarrow_{\mathcal{M}}\ (\sigma'_1,\ \rho'_1)\\ (\sigma_2,\ \rho_2),\ \bot\ \vdash\ \mathtt{P}\ \Downarrow_{\mathcal{M}}\ (\sigma'_2,\ \rho'_2)\\ \forall x \notin \mathcal{S}(\mathtt{P}).\ \sigma_1(x) = \sigma_2(x)\\ \forall x \in \mathcal{S}(\mathtt{P}).\ \rho_1(x) = \top\\ \rho_1 = \rho_2\end{array}\right\} \Rightarrow\ \left(\rho'_1 = \rho'_2\right)$$

Proof (Proof summary). The detailed formal proof can be found in the companion technical report [6]. In order to be able to use induction, a generalization of the theorem stated above is proved with two differences in its statement: program counters (t^{pc}) must only be the same for the two executions and variables whose tag is $\bot$ must have the same value in both executions instead of the hypotheses based on $\mathcal{S}(P)$. The case for **skip** *is direct. For assignments, the only tag modified is the one of the variable assigned. As the two tag stores are initially equal, $\rho_1(e)$ is equal to $\rho_2(e)$. Therefore, both tag stores are equal after the execution of the assignment. The case for sequences goes by induction. For* `if` *statements, if the same branch is executed then by induction the theorem holds. If two different branches are executed then the tag of the program counter is $\top$. Therefore, the tag of every assigned variable becomes $\top$. Additionally, the unexecuted branch is analyzed and the tag of every variable which may have been assigned to is set to $\top$. Fig. 2 constrains the analysis to make the same choices as the execution with regard to which subbranches to ignore and which ones to analyze. Therefore, the set of variables returned by the analysis of the unexecuted branch is exactly the set of variables whose tag would have been set to $\top$ by an execution of this branch. Therefore, whatever branch is executed or analyzed, the same set of variables have their tag set to $\top$. Hence, the final tag stores are equal after the execution of the* `if` *statement. For* `while` *statements, if the condition evaluates to the same value then the inductive hypothesis implies that the theorem holds. Otherwise, it means that its tag is $\top$. Therefore the final tag store is the least upper bound of the initial tag store and a new tag store ρ^e. This new tag store is either the tag store returned by the execution of the* `while` *statement (executing the body at least once) with program counter tag (t^{pc}) equal to $\top$ or the analysis of the same statement. As for* `if` *statements, in both cases the tags of the exact same set of variables are set to $\top$. Hence, the theorem holds.*

5 Benefits of Monitoring Compared to Static Analyses

Monitoring an execution has a cost. So, what are the main benefits of noninterference monitoring compared to static analyses? The first concerns the possibility that a monitoring mechanism can be used to change the security policy for each execution. In the majority of cases, running a static analysis before every execution would be more costly than using a monitor. The second reason is that noninterference is a rather strong property. Many programs are rejected by static analyses of noninterference. In such cases it is still possible to use a monitoring mechanism with the possibility that some executions will be altered by the monitoring mechanism. However behavior alteration is an intrinsic feature of any monitoring mechanism. Monitoring noninterference ensures confidentiality while still allowing testing with regard to other specifications using unmonitored executions as perfect oracle — at least as perfect as the original program.

There are two main reasons why it is interesting to use a noninterference monitor on a program rejected by a static analysis. The first one is that a monitoring mechanism may be more precise than static analyses because during execution the monitoring mechanism gets some accurate information about the "path behavior" of the program. As an example, let us consider the following program where *h* is the only secret input and *l* the only other input (a public one).

```
1 if ( test1(l) ) then tmp := h else skip end;
2 if ( test2(l) ) then x := tmp else skip end;
3 output x
```

Without information on *test1* and *test2* (and often, even with), a static analysis would conclude that this program is unsafe because the secret input information could be carried to *x* through *tmp* and then to the output. However, if *test1* and *test2* are such that no value of *l* makes both predicates true, then any execution of the program is perfectly safe. In that case, the monitoring mechanism would allow any execution of this program. The reason is that, *l* being a public input, only executions following the same path as the current execution are taken care of by the monitoring mechanism. So, for such configurations where the branching conditions are not influenced by the secret inputs, a monitoring mechanism is at least as precise as any static analysis — and often more precise.

The second reason lies in the granularity of the noninterference property. Static analyses have to take into consideration all possible executions of the program analyzed. This implies that if a single execution is unsafe then the program (thus all its executions) is rejected. Whereas, even if some executions of a program are unsafe, a monitor still allows this program to be used. The unsafe executions, which are not useful, are altered to enforce confidentiality while the safe executions are still usable. For example, the program on page 83 being interfering, any static noninterfering analysis rejects this program. Therefore, users would be advised not to use this program at all. However, using a noninterference monitor, it is possible to safely use this program. When communicating with any user other than sexyPirate, monitored executions of this program have their normal behavior. When communicating with sexyPirate, monitored executions are safely detected as potentially interfering and can therefore be corrected to prevent any secret leakage. Of course, when attempting to communicate with sexyPirate, executions of this program are altered and it is therefore not possible to communicate with sexyPirate. However, this is the desired behavior of a noninterference monitor when confidentiality is more important than the service provided by the program.

6 Related Work

The vast majority of research on noninterference concerns static analyses and involves type systems [13, 15]. Some "real size" languages together with security type system have been developed (for example, JFlow/JIF [11] and FlowCaml [14]).

Dynamic information flow analyses [1, 3, 19, 20] are not as popular as static analyses for information flow, but there has been interesting research. For example, RIFLE [17] is a complete run-time information flow security system based on an architectural framework and a binary translator. Masri et al. [10] present a dynamic information flow analysis for structured or unstructured languages. Venkatakrishnan et al. [18] propose a program transformation for a simple deterministic procedural language that ensures a sound detection of information flows. Trishul [12] is an interesting implementation of a Java Virtual Machine integrating a dynamic information flow control mechanism. Yoshihama et al. [21] propose a Java Architecture for Web Applications with "direct" and "explicit indirect" information flow control abilities (as they acknowledge, they do not handle "implicit indirect" flows). One of the main interest of those works is the size of the language addressed. However, none of those five later works prove that the correction mechanisms of "bad" flows proposed do not create a new covert channel that can reveal secret information — see, e.g., [7] — or even, for some of them, that the detection mechanism is sound with regard to their notion of information flow. In fact, those analyses and correction mechanisms are likely to create a new covert channel. Theorem 2 proves that a correction mechanism of "bad" flows can be based on the dynamic analysis proposed in this paper as the results of the dynamic analysis are the same for every executions started with the same public inputs. More recently, Shroff, Smith, and Thober [16] proposed a dynamic information flow analysis which tracks direct flows and collects indirect flows dynamically. The information collected about indirect flows is transferred from one execution to another using a cache mechanism. After an undetermined number of executions, the analysis will know about all indirect flows in the program and thus will then be sound with regard to the detection of all information flows. This information about indirect flows can be precomputed using a static analysis at the cost of a decrease of precision. Using this approach they are able to handle a language including alias and method calls.

Contrary to common assumption, none of the related works on dynamic information flow analysis known to the author take enough context information into account to detect that the program on page 83 is noninterfering when using it to communicate with users other than sexyPirate. For example, the transformation of Venkatakrishnan et al. [18] updates the security label of c with the security label of the condition on line 3 before executing line 7. Therefore, at line 7, c is always considered as secret even if the user is not communicating

with sexyPirate. When executing the assignment of line 5, the program counter of Shroff et al.'s work [16] contains a reference to the program point of line 3 and therefore is added to the set of source of implicit flows to c. Consequently, any complete implicit dependency cache contains a reference to the implicit flow from line 3 to line 7. As the test of line 3 is always executed, Shroff et al.'s work always considers line 7 as displaying secret information. The dynamic analysis proposed in this paper *is* able to detect the noninterfering behavior of the program on page 83 when communicating with someone other than sexyPirate.

7 Conclusion

This article addresses the problem of information flow verification and correction at run time in order to enforce the confidentiality of secret data. The confidentiality property to monitor is expressed using the property of noninterference between secret inputs of the execution and its public outputs. The language taken into consideration is a sequential language with assignments and conditionals (including loops). The main difference between the monitoring mechanism proposed in this article and the ones of related works lies in the static analysis used to detect implicit indirect flows. The static information flow analyses used by the dynamic analysis proposed in this article are sensitive to the current program state. This allows the overall dynamic information flow analysis to increase the precision of the detection of implicit and explicit indirect flows. In Sect. 4, the proposed noninterference monitor is proved to be sound both with regard to the detection of information flows and with regard to their correction when necessary.

Acknowledgments: The author is grateful to Anindya Banerjee, Gérard Boudol, Thomas Jensen, Andreï Sabelfeld and David Schmidt for their helpful feedback on an earlier version of this work.

Bibliography

[1] J. Brown and T. F. Knight, Jr. A minimal trusted computing base for dynamically ensuring secure information flow. Technical Report ARIES-TM-015, MIT, 2001.

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

[3] J. S. Fenton. Memoryless subsystems. *The Computer Journal*, 17(2):143–147, 1974.

[4] G. Le Guernic. *Confidentiality Enforcement Using Dynamic Information Flow Analyses*. PhD thesis, Kansas State University, 2007.

[5] G. Le Guernic. Automaton-based Confidentiality Monitoring of Concurrent Programs. In *Proc. Computer Security Foundations Symp.*. IEEE, 2007.

[6] G. Le Guernic. Precise dynamic verification of noninterference. Technical report, INRIA, July 2008. http://hal.inria.fr/inria-00162609/fr/.

[7] G. Le Guernic and T. Jensen. Monitoring Information Flow. In *Proc. W. on Foundations of Computer Security*, pages 19–30. DePaul University, 2005.

[8] G. Le Guernic, A. Banerjee, T. Jensen, and D. Schmidt. Automata-based Confidentiality Monitoring. In *Proc. Annual Asian Computing Science Conf.*, volume 4435 of *LNCS*, 2006.

[9] J. Leyden. Trojan planted on US Consulate website. The Register, Sept. 2007.

[10] W. Masri, A. Podgurski, and D. Leon. Detecting and debugging insecure information flows. In *Proc. Int. Symp. on Software Reliability Engineering*, pages 198–209. IEEE, 2004.

[11] A. C. Myers. JFlow: Practical mostly-static information flow control. In *Proc. Symp. Principles of Programming Languages*, pages 228–241, 1999.

[12] S. K. Nair, P. N. D. Simpson, B. Crispo, and A. S. Tanenbaum. A virtual machine based information flow control system for policy enforcement. In *Proc. W. on Run Time Enforcement for Mobile and Distributed Systems*, volume 197, pages 3–16, 2007. Elsevier.

[13] F. Pottier and S. Conchon. Information flow inference for free. In *Proc. Int. Conf. on Functional Programming*, pages 46–57. ACM Press, 2000.

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

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

[16] P. Shroff, S. F. Smith, and M. Thober. Dynamic dependency monitoring to secure information flow. In *Proc. Computer Security Foundations Symp.*. IEEE, 2007.

[17] N. Vachharajani, M. J. Bridges, J. Chang, R. Rangan, G. Ottoni, J. A. Blome, G. A. Reis, M. Vachharajani, and D. I. August. RIFLE: An architectural framework for user-centric information-flow security. In *Proc. Int. Symp. on Microarchitecture*, 2004.

[18] V. N. Venkatakrishnan, W. Xu, D. C. DuVarney, and R. Sekar. Provably correct runtime enforcement of non-interference properties. In *Proc. Int. Conf. on Information and Communications Security*, volume 4307 of *LNCS*, pages 332–351. Springer-Verlag, 2006.

[19] C. Weissman. Security controls in the adept-50 timesharing system. In *Proc. AFIPS Fall Joint Computer Conf.*, volume 35, pages 119–133, 1969.

[20] J. P. L. Woodward. Exploiting the dual nature of sensitivity labels. In *Proc. Symp. Security and Privacy*, pages 23–31, 1987.

[21] S. Yoshihama, T. Yoshizawa, Y. Watanabe, M. Kudoh, and K. Oyanagi. Dynamic information flow control architecture for web applications. In *Proc. European Symp. on Research in Computer Security*, volume 4734 of *LNCS*, pages 267–282. Springer-Verlag, 2007.