A Weakest Precondition Approach to Robustness*

Musard Balliu[1] and Isabella Mastroeni[2]

[1] School of Computer Science and Communication, Royal Institute of Technology, Stockholm, Sweden
musard@kth.se
[2] Dipartimento di Informatica, Università di Verona, Strada Le Grazie 15, I-37134, Verona, Italy
isabella.mastroeni@univr.it

Abstract. With the increasing complexity of information management computer systems, security becomes a real concern. E-government, web-based financial transactions or military and health care information systems are only a few examples where large amount of information can reside on different hosts distributed worldwide. It is clear that any disclosure or corruption of confidential information in these contexts can result fatal. Information flow controls constitute an appealing and promising technology to protect both data confidentiality and data integrity. The certification of the security degree of a program that runs in untrusted environments still remains an open problem in the area of language-based security. Robustness asserts that an active attacker, who can modify program code in some fixed points (*holes*), is unable to disclose more private information than a passive attacker, who merely observes unclassified data. In this paper, we extend a method recently proposed for checking declassified non-interference in presence of passive attackers only, in order to check robustness by means of weakest precondition semantics. In particular, this semantics simulates the kind of analysis that can be performed by an attacker, i.e., from public output towards private input. The choice of semantics allows us to distinguish between different attacks models and to characterize the security of applications in different scenarios.

Our results are sound to address confidentiality and integrity of software running in untrusted environments where different actors can distrust one another. For instance, a web server can be attacked by a third party in order to steal a session cookie or hijack clients to a fake web page.

Keywords: program semantics, non-interference, robustness, declassification, active attackers, abstract interpretation, security.

1 Introduction

Security is an enabling technology, hence security means power. So to cite some examples, the correct functionality and coordination of large scale organizations, e-government, web services in general relies on confidentiality and integrity of data

* We would like to thank Mads Dam and anonymous referees for insightful suggestions and comments.

M.L. Gavrilova et al. (Eds.): Trans. on Comput. Sci. X, LNCS 6340, pp. 261–297, 2010.

exchanged between different agents. Nowadays, distributed and service oriented architectures are the first business alternative to the old fashioned client-server architectures. According to OWASP (Open Web Application Security Project) [1], the most critical security risks are due to application level attacks as injections flaws or XSS (Cross Site Scripting). Moreover, current and future trends in software engineering prognosticate mobile code technology (multi application smart cards, software for embedded systems), extensibility and platform independence. It is worth noting that all these features, almost unavoidable, become real opportunities for the attackers to exploit system bugs in order to disclose and/or corrupt valuable information. For instance, in such a context, it is easier to distribute worms or viruses that run everywhere or to embed malicious code to exploit vulnerabilities in a web server.

In many scenarios, different agents, each having their own security policy and probably not trusting each other, have to cooperate for a certain goal, for example electing the winner in an online auction. It can happen that the host used for computation violates security by either leaking information itself or causing other hosts to leak information [27,5]. In a cryptographic context, secure multi-party computation (MPC) [25] consists of computing a function between different agents, each knowing a secret they don't want to reveal to the other participating agents. It is very common that an adversary is part of such a systems by taking the control of some hosts and trying to reveal private data of the other participating hosts. As a result, it is both useful and necessary to address problems on confidential information disclosed by an adversary that can control and observe part of the system, to characterize the possible harm in case some condition is verified or to state conditions when the whole system is robust to some extent. Application level enforcement that combines programming languages and static analysis seems a promising remedy to such a problem [26,5]

Secure information flow concerns the problem of disclosing private information to an untrusted observer. This problem is indeed actual each time a program, manipulating both sensitive and public information, is executed in an untrusted environment. In this case, security is usually enforced by means of *non-interference* policies [13], stating that private information must not affect the observable public information. In the non-interference context, variables have a confidentiality level, usually public/low and private/high, and variations of the private input has not to affect the public output. In this case, we are considering attackers that can only observe the I/O behavior of programs and that, from these observations, can make some kind of *reverse engineering* in order to derive private information from the observation of public data.

Our starting idea is that of finding the program vulnerabilities by simulating the possible reasonings that an attacker can perform on programs. Indeed, we can think that the attacker can use the output observation in order to derive, *backward*, some (even partial) private input information. This is the idea of the *backward analysis* recently proposed in [3] for declassified non-interference, where declassification is modeled by means of abstract domains [8]. The ingredients of this method are: the initial declassification policy modeled as an abstraction of private input domain and the weakest liberal precondition semantics of programs [12,11], characterizing the backward analysis (i.e.,

from outputs to inputs) and the simulation of the attacker observational activity. The certification process consists in considering a possible output (public) observation and computing the weakest liberal precondition semantics of the program starting from this observation. By definition, the *weakest* precondition semantics provides the *greatest* set of possible input states leading to the given output observation. In other words, it characterizes the greatest collection of input states, and in particular of private inputs, that an attacker can identify starting from the given observation. In this way, the attacker can restrict the range of private inputs inside this collection, which corresponds to a partial release of private information. Moreover, we can note that, the fact that we compute the *weakest* precondition for the given observation, provides a characterisation of the *maximal* information released by the observation, in the lattice of abstract interpretations. Namely, starting from the results provided by the analysis, we construct an abstract domain, representing the private abstract property released, which is the most concrete one released by the program [3].

Our aim is to use these ideas also in presence of *active attackers*, namely attackers that can both observe and modify program semantics. We consider the model of active attackers proposed in [20] which can transform program semantics simply by inserting malicious code in some fixed program points (*holes*), known by the programmer. We can show that, also in presence of this kind of attackers, the weakest precondition semantics computation can be exploited for characterising the information disclosed, and therefore for revealing program vulnerabilities. This characterisation can be interpreted from two opposite points of view: the attacker and the program administrator. The attacker can be any malicious adversary trying to disclose confidential information about the system; the administrator wants to know whether the system releases private information due to particular inputs.

An important security property concerning active attackers, and related to the information disclosed, is *robustness* [26]. It "measures" the security degree of programs wrt active attackers by certifying that active attackers cannot disclose more information than what a passive attacker (a simple observer) can do.

We propose to use the weakest precondition-based analysis in order to certify also robustness of programs. The first idea we consider is to compute the maximal information disclosed both for passive attackers [3] and for active attackers, and then compare the results in the lattice of abstract interpretations for certifying robustness: if there exists at least one active attacker disclosing more than the passive one, the program fails to be robust. The problem of this technique is that it requires a program analysis for each attack, this means that it becomes unfeasible when dealing with an infinite number of possible active attacks. In order to overcome this problem, we need an analysis independent of the code of the particular active attack. For this reason, we exploit the weakest precondition computation in order to provide a *sufficient condition* that guarantees robustness independently of the attack. In particular we provide a condition that has to hold before each hole, for preventing the attacker to be successful. We initially study this condition for I/O attackers, i.e., attackers that can only observe the I/O program behavior, and afterwards we extend it to attackers able to observe also intermediate states,

i.e., trace semantics of programs. Finally, we note that, in some restricted contexts, for example where the activity of the attackers is limited by the environment, the standard notion of robustness may become too strong. For dealing with these situations we introduce a weakening of robustness, i.e., *relative robustness*, where we restrict the set of active attackers that we are checking for robustness.

There are various interesting applications where our approach is successful to capture confidential information flaws. Here we select two cases concerning API (Application Programming Interface) security and XSS attacks and apply the weakest precondition analysis to check robustness. The first case enforces the security of an API used to verify the password inserted in an ATM cash machine. The adversary is able to reveal the entire password by tampering with low integrity data prior to call API function [4]. The second example concerns a web attacks using Javascript. As we will see, a naive control of code integrity can reveal the session cookie to the adversary [23,6]. Our robustness analysis by weakest precondition semantics is sufficient to prevent attacks in both examples.

Roadmap. The rest of the paper is organized as follows. In Section 2 we give a general overview of abstract interpretation, which constitutes the underlying framework that we use to compare the information disclosed. In Section 3 we present the target security background that we address in our approach. In particular we recall notions of non-interference, robustness, declassification, decentralized label model and decentralized robustness. In Section 4 we compute (qualitatively) the maximal private information disclosed by active attackers. In particular, Section 4.1 introduces the problem of computing the maximal release by active attackers for I/O (denotational) semantics. Section 4.2 extends the analysis for attacks observing program traces. In Section 5 we discuss conditions to enforce robustness, which constitutes our main contribution. Section 5.1 presents a static analysis approach based on weakest preconditions to enforce robustness for I/O semantics; Section 5.3 extends these results for trace semantics; In Section 5.4 we compare our method with type-based methods. Section 6 introduces relative robustness which deals with restricted classes of attacks; in Section 6.1 we interpret decentralized robustness in our approach. In Section 7 we use the current approach in the context of real applications and explain how it captures the security properties we are interested in. Sect. 8 we present the most relevant related works. We conclude with Section 9 by discussing the current state of art and devising new directions for future work. This is an extended and revised version of [2].

2 Abstract Interpretation: An Informal Introduction

We use the standard framework of abstract interpretation [8,9] for modeling properties. For example, instead of computing on integers we might compute on more abstract properties, such as the sign $\{+, -, 0\}$ or parity $\{\texttt{even}, \texttt{odd}\}$. Consider the program $\texttt{sum}(x, y) = x + y$, then it is abstractly interpreted as $\texttt{sum}^*$: $\texttt{sum}^*(+, +) = +$, $\texttt{sum}^*(-, -) = -$, but $\texttt{sum}^*(+, -) =$ *"I don't know"* since we are not able to determine the sign of the sum of a negative number with a positive one (modeled by the fact that the result can be any value). Analogously, $\texttt{sum}^*(\texttt{even}, \texttt{even}) = \texttt{even}$,

$\texttt{sum}^*(\texttt{odd},\texttt{odd}) = \texttt{even}$ and $\texttt{sum}^*(\texttt{even},\texttt{odd}) = \texttt{odd}$. More formally, given a concrete domain C we choose to describe abstractions of C as upper closure operators. An *upper closure operator* (uco for short) $\rho : C \to C$ on a poset C is monotone, idempotent, and extensive: $\forall x \in C.\ x \leq_C \rho(x)$. The upper closure operator is the function that maps the concrete values with their abstract properties, namely with the best possible approximation of the concrete value in the abstract domain. For example, the operator *Sign* : $\wp(\mathbb{Z}) \to \wp(\mathbb{Z})$, on the powerset of integers, associates each set of integers S with its sign: *Sign*$(\varnothing)$ =*"none"*, *Sign*$(S) = +$ if $\forall n \in S.x > 0$, *Sign*$(0) = 0$, *Sign*$(S) = -$ if $\forall n \in S.\ n < 0$ and *Sign*(S) = *"I don't know"* otherwise. The used property names *"none"*, +,0,− and *"I don't know"* are the names of the following sets in $\wp(\mathbb{Z})$: $\varnothing$, $\{\ n \in \mathbb{Z} \mid n > 0\ \}$, $\{0\}$, $\{\ n \in \mathbb{Z} \mid n < 0\ \}$ and $\mathbb{Z}$. Namely the abstract elements, in general, correspond to the set of values with the property they represent. Analogously, we can define an operator *Par* : $\wp(\mathbb{Z}) \to \wp(\mathbb{Z})$ associating each set of integers with its parity. *Par*$(\varnothing)$ = *"none"* $= \varnothing$, *Par*$(S) = \texttt{even} = \{\ n \in \mathbb{Z} \mid n$ is even $\}$ if $\forall n \in S.\ n$ is even, *Par*$(S) = \texttt{odd} = \{\ n \in \mathbb{Z} \mid n$ is odd $\}$ if $\forall n \in S.\ n$ is odd and *Par*(S) = *"I don't know"* $= \mathbb{Z}$ otherwise. Formally, closure operators ρ are uniquely determined by the set of their fix-points $\rho(C)$, for instance *Sign* $= \{\mathbb{Z}, > 0, < 0, 0, \varnothing\}$. Abstract domains on the complete lattice $\langle C, \leq, \vee, \wedge, \top, \bot\rangle$ form a complete lattice, formally denoted $\langle \mathit{uco}(C), \sqsubseteq, \sqcup, \sqcap, \lambda x.\ \top, \lambda x.\ x\rangle$, where $\rho \sqsubseteq \eta$ means that ρ is more concrete than η, namely it is more precise, $\sqcap_i \rho_i$ is the greatest lower bound taking the most abstract domain containing all the ρ_i, $\sqcup_i \rho_i$ is the least upper bound taking the most concrete domain contained in all the ρ_i, $\lambda x.\ \top$ is the most abstract domain unable to distinguish concrete elements, the identity on C, $\lambda x.\ x$, is the most concrete abstract domain, the concrete domain itself.

3 Security Background

Information flow models of confidentiality, also called non-interference [13], are widely studied in literature [21]. Generally they consider the denotational semantics of a program P, denoted $[\![P]\!]$ and all program variables, in addition to their base type (int, float etc.), have a security type that varies between private (H) and public (L). In this paper we consider only terminating computations. Hence, there are basically two ways the program can release private information by the observation of the public outputs: due to an explicit flow corresponding to a direct assignment of a private variable to a public variable and due to an implicit flow corresponding to control structures of the program, such as the conditional **if** or the **while** loop [21].

3.1 Non-interference and Declassification

A program satisfies *standard non-interference* if for all the variations of private input data there is no variation of public output data. More formally, given a set of program states Σ, namely a set of functions mapping variables to values $\mathbb{V}$, we represent a state as a tuple $(\vec{h}, \vec{l})$ where the first component denotes the value of private variables and the second component denotes the value of public variables. Let P be a program, then P satisfies non-interference if

$$\forall l \in \mathbb{V}^{\mathtt{L}}, \forall h_1, h_2 \in \mathbb{V}^{\mathtt{H}}.[\![P]\!](h_1, l)^{\mathtt{L}} = [\![P]\!](h_2, l)^{\mathtt{L}}$$

where $v \in \mathbb{V}^{\mathbb{T}}$, $\mathbb{T} \in \{\mathtt{H}, \mathtt{L}\}$, denotes the fact that v is a possible value of a variable with security type $\mathbb{T}$ and $(h, l)^{\mathtt{L}} = l$. Declassified non-interference considers a property on private inputs which can be observed [7,3]. Consider a predicate ϕ on $\mathbb{V}^{\mathtt{H}}$, a program P satisfies *declassified* non-interference if

$$\forall l \in \mathbb{V}^{\mathtt{L}}, \forall h_1, h_2 \in \mathbb{V}^{\mathtt{H}}.$$
$$\phi(h_1) = \phi(h_2) \;\Rightarrow\; [\![P]\!](h_1, l)^{\mathtt{L}} = [\![P]\!](h_2, l)^{\mathtt{L}}$$

3.2 Robust Declassification

In language-based settings, active attackers are known for their ability to control, i.e., observe and modify, part of the information used by the program.

Security levels form a lattice whose ordering specifies the relation between different security levels. Each program variable has two security types that model, respectively, the confidentiality level and the integrity level. In our context, all the variables have only two security levels; `L` stands for *low, public, modifiable* and `H` stands for *high, private, unmodifiable*. Moreover, we assume, for each variable x, the existence of two functions, $\mathcal{C}(x)$ (confidentiality level) which shows whether the variable x is observable or not and $\mathcal{I}(x)$ (integrity level) which shows whether the variable x is modifiable or not. Definitively, each variable can have four possible security types, i.e., `LL`, `LH`, `HL`, `HH`. For example, if the variable x has type `LL` then x can be both *observed* and *modified* by the attacker, if the variable x has type `HL` then x can be *modified* by the attacker, but it cannot be *observed*, and so on.

The programs are written according to the syntax of a simple *while* language. In order to allow semantic transformations during the computation, we consider another construct, called *hole* and denoted by $[\bullet]$, which models the program locations where a potential attacker can insert some code [20].

$$c ::= \textbf{skip} \mid x := e \mid c_1; c_2 \mid \textbf{if } e \textbf{ then } c_1 \textbf{ else } c_2 \mid$$
$$\textbf{while } e \textbf{ do } c \mid [\bullet]$$

where $e ::= v \in \mathbb{V} \mid x \mid e_1 \text{ op } e_2$. The low integrity code inserted in holes models the untrusted code assumed under the control of the attacker. Hence, let $P[\vec{\bullet}]$ denote a program with holes and $\vec{a}$ (vector of fixed attacks for each program hole) an attack, $P[\vec{a}]$ denotes the program under control of the given attack. A *fair attack* is a program respecting the following syntax [20]:

$$a ::= \textbf{skip} \mid x := e \mid a_1; a_2 \mid \textbf{if } e \textbf{ then } a_1 \textbf{ else } a_2 \mid$$
$$\textbf{while } e \textbf{ do } a$$

where all variables in e and x have security type `LL`. It is worth noting that fair attackers can use in their attacks only the variables that are both observable and modifiable.

An important notion when dealing with active attackers is *robustness* [26]. Informally, a program is said to be *robust* when no active attacker, who actively controls the code in the holes, can disclose more information about private inputs than what can be disclosed by a passive attacker, who merely observes the programs I/O. Note that, by using this attacker definition it becomes possible to translate robustness into a

language-based setting. Indeed, robust declassification holds if for all attacks $\vec{a}$ whenever program $P[\vec{a}]$ cannot distinguish program behavior on some memories, any other attacker code $\vec{a}'$ cannot distinguish program behavior on these memories [20]. Thus, we can formally recall the notion of *robustness*, for terminating programs, in presence of active fair attackers [20].

$$\forall h_1, h_2 \in \mathbb{V}^{\mathtt{H}}, \forall l \in \mathbb{V}^{\mathtt{L}}, \forall \vec{a}, \vec{a}' \text{ active fair attack :}$$
$$[\![P[\vec{a}]]\!](h_1, l)^{\mathtt{L}} = [\![P[\vec{a}]]\!](h_2, l)^{\mathtt{L}} \Rightarrow [\![P[\vec{a}']]\!](h_1, l)^{\mathtt{L}} = [\![P[\vec{a}']]\!](h_2, l)^{\mathtt{L}}$$

Namely, a program is robust if any active (fair) attacker can disclose at most the same information (property of private inputs) as a passive attacker can disclose. A passive attacker is an attacker able only to observe program execution, which in this context corresponds to the active attacker $\vec{a} = \vec{\textbf{skip}}$.

3.3 Weakest Liberal Precondition Semantics

In this section we briefly present the *weakest liberal precondition* semantics (*Wlp* for short), which constitutes our basic instrument for performing static analysis. In particular, given a program c and a predicate P, $\mathit{Wlp}(c, P)$ corresponds to the greatest set of input states σ such that if (c, σ) terminates in a final state σ', then σ' satisfies the predicate P [15,14]. In our case, these predicates correspond to quantifier-free first order formulas which are transformed by the *Wlp* semantics. Below, we present the rules of the semantics.

- $\mathit{Wlp}(\textbf{skip}, \Phi) = \Phi$
- $\mathit{Wlp}(x := e, \Phi) = \Phi[e/x]$
- $\mathit{Wlp}(c_1; c_2, \Phi) = \mathit{Wlp}(c_1, \mathit{Wlp}(c_2, \Phi))$
- $\mathit{Wlp}(\textbf{if } e \textbf{ then } c_1 \textbf{ else } c_2, \Phi) = (e \wedge \mathit{Wlp}(c_1, \Phi)) \vee (\neg e \wedge \mathit{Wlp}(c_2, \Phi))$
- $\mathit{Wlp}(\textbf{while } e \textbf{ do } c, \Phi) = \bigvee_{i=0}^{n} \mathit{Wlp}_i(\textbf{while } e \textbf{ do } c, \Phi)$
  where given $(C \stackrel{\text{def}}{=} \textbf{while } B \textbf{ do } C_1)$

$$\begin{cases} \mathit{Wlp}_0(C, \Phi) \stackrel{\text{def}}{=} \neg B \wedge \Phi \\ \mathit{Wlp}_{i+1}(C, \Phi) \stackrel{\text{def}}{=} \mathit{Wlp}(\textbf{if } B \textbf{ then } C_1 \textbf{ else skip}, \mathit{Wlp}_i(C, \Phi)) \end{cases}$$

Almost all the above rules are easy to read. For instance, the weakest precondition of the conditional, given a postcondition Φ, corresponds to the disjunction of conjunctions of *Wlp* of each branch and the boolean condition of the guard. It is also worth noting that the *Wlp* of the loop requires the computation of some invariant formula. There exists techniques for doing that [16], but in this paper we don't consider them explicitly. The automatic generation of such invariants would be an interesting future direction we plan to explore more in details. Unlike weakest precondition semantics, *Wlp* defines a partial verification condition, namely only if the program does terminate the post-condition Φ should hold. In any case, for the purposes of this paper, we will be interested only in terminating programs, so we can establish the weakest liberal precondition in a finite number of iterations.

3.4 Certifying Declassification

In this section, we introduce a technique recently proposed for certifying declassification policies [3,18] in presence of passive attackers only, i.e., attackers that can only observe program execution. The method performs a backward analysis, computing the weakest precondition semantics starting from output observation, in order to derive the maximal information that an attacker can disclose from a given observation. We use abstract interpretation for modeling the declassified properties.

Certifying declassification. In [3,18] the authors present a method to compute the maximal private input information disclosed by passive attackers. They consider only terminating computations, which means that the logical language does not have expressiveness limits [24]. Their method has two main characteristics: it is a static analysis, and it performs a backward analysis from the observed outputs towards the inputs to protect. The first aspect is important since we would like to certify programs without executing them, the latter is important because non-interference aims to protect the system private *input* while attackers can observe public *outputs*. Both these characteristics are embedded in the weakest liberal precondition semantics of programs. Starting from a given observed output *Wlp* semantics computes, by definition, the *greatest* set of input states leading to the given observation. From this characterisation we can derive in particular the private input information released by observing output public variables. This corresponds exactly to the maximal private information disclosed by the program semantics. In this way, we are statically simulating the kind of analysis an attacker can perform in order to obtain initial values of (or initial relations among) private information. We can model this information by a first order predicate; the set of program states disclosed by the *Wlp* semantics are the ones which satisfy this predicate. In order to be as general as possible, we consider the public observations parametric on some symbolic value represented by some logical variable. We denote by $\vec{l} = \vec{n}$ the parametric value of each low confidentiality program variable. For instance, the formula $(l = n)$ means that the program variable l has the symbolic value n. Generally, the public output observation corresponds to a first order formula that is the conjunction of all low confidentiality variables, i.e., variables with security types `LL` or `LH`.

$$\Phi_0 \stackrel{\text{def}}{=} \{l_1 = n_1 \wedge l_2 = n_2 \wedge \cdots \wedge l_k = n_k\} = \bigwedge_{i=1}^{k} (l_i = n_i)$$

where $\forall l_i . \mathcal{C}(l_i) = \texttt{L}$. Without loss of generality, we can assume this formula to be in a disjoint normal form, namely a disjunction of conjunctions. We call *free variables* of a logical formula Φ and denote $\mathcal{FV}(\Phi)$ the set of program variables occurring in Φ, where Φ is a quantifier-free. Moreover, we assume to eliminate all possible redundancies and all subformulas that can be subsumed by others in the same formula. For instance, let $(l > 1 \wedge l > 0)$ be a logical formula. We can simply write $(l > 1)$ because this subsumes the fact that $l > 0$. From now on we'll suppose to have each logical formula in this form called *normal form*.

For instance, consider the program P with $h_1, h_2 : \mathtt{HH}$ and $l : \mathtt{LL}$.

$$P \stackrel{\text{def}}{=} \textbf{if } (h_1 = h_2) \textbf{ then } l := 0; \textbf{ else } l := 1;$$

$\mathit{Wlp}(P, l = n) = \{(h_1 = h_2 \wedge n = 0) \vee (h_1 \neq h_2 \wedge n = 1)\}$.

If we observe $l = 0$ in public output, all we can say about private inputs h_1, h_2 is $h_1 = h_2$. Otherwise, if we observe $l = 1$, we can conclude that $h_1 \neq h_2$.

In [3,18] this technique is formally justified by considering an abstract domain completeness-based [8] model of declassified non-interference. Here we avoid the formal details, and we simply show where and how we use abstract interpretation. Note that, usually *Wlp* semantics is applied to specific output states in order to derive the greatest set of input states leading to the output one. Here, the technique starts from the state $\vec{l} = \vec{n}$, which is indeed an *abstract state*, namely the state where the private variables can have any value, while the public variables $\vec{l}$ have the specific symbolic value $\vec{n}$. This corresponds to the abstraction $\mathcal{H} \in \mathit{uco}(\wp(\mathbb{V}))$ [3] modeling the fact that the attacker cannot observe private data. Formally, it associates with a generic output state $\langle h, l\rangle$ the abstract state $\langle \mathbb{V}^{\mathtt{H}}, l\rangle = \left\{ \langle h', l\rangle \,\middle|\, h' \in \mathbb{V}^{\mathtt{H}} \right\}$. As far as the input characterisation is concerned, we know that an abstract property is described by the set of all the concrete values satisfying the property. Hence, if the *Wlp* semantics characterises a set of inputs, and in particular of private inputs, then this set can be uniquely modeled as an abstract domain, i.e., the abstract property released. Consider, for instance, the trivial program fragment P above. According to the output value observed, $l = 0$ or $l = 1$, we have respectively the set of input states $\{\langle h_1, h_2, l\rangle \mid h_1 = h_2\}$ or $\{\langle h_1, h_2, l\rangle \mid h_1 \neq h_2\}$. This characterisation can be uniquely modeled by the abstract domain[1]

$$\phi = \{\top, \{\langle h_1, h_2, l\rangle \mid h_1 = h_2\}, \{\langle h_1, h_2, l\rangle \mid h_1 \neq h_2\}, \varnothing\}$$

Hence, if we declassify ϕ, the program is secure since the information released corresponds to what is declassified. While if, as in standard non-interference, nothing is declassified, modeled by the declassification policy $\phi' = \lambda x.\ \top$[2], then $\phi \sqsubseteq \phi'$, namely the policy is violated since the information released is more (concrete) than what is declassified.

3.5 Decentralized Label Model and Decentralized Robustness

Decentralized label model was proposed as a fine-grained model to enforce end-to-end security for systems with mutual distrust and decentralized authority that want to share data with each other [19]. Basically, every agent in the system defines and controls his own security policy and states which data, under his ownership, could be visible (declassified) to other agents in the system. The system itself must ensure that security policies are not circumvented and satisfy security concerns of all agents. More precisely, decentralized label model consists of two basic flavors: *principals*, whose security should be ensured in the model and *labels*, which constitute the means to enforce security policies. Principals can be users, processes, groups, roles possibly related by an *acts-for*

[1] The elements $\top$ and $\varnothing$ are necessary for obtaining an abstract domain.

[2] Since $\forall x, y$ we have $\phi'(x) = \phi'(y)$, declassified non-interference with ϕ' corresponds to standard non-interference.

relation which allows delegation of authority between them. For instance, if principal P *acts-for* principal Q, formally $P \succeq Q$, it means that P has all privileges of Q. On the other hand, labels are data annotations that express the security policy the owner sets on his data. In particular, if some data are annotated by label *owner: reader*, the policy on that data defines the owner and the set of principals that can read such data. Security labels form a security lattice where the higher an element is in the lattice, the more restrictive are the security concerns of the data it labels. Moreover, the decentralized label model supports a declassification mechanism and allows to express policies regarding both confidentiality and integrity. The model is used to perform static analysis based on type systems to enforce information flow policies.

Decentralized robustness is an approach to enforce the security condition of robustness in the decentralized label model [5]. In particular, the fact that each principal does not trust the others means that each principal may be a potential attacker. Hence, robustness is analyzed relatively to two principals: one fixes the point of view of the analysis, the other is the potential attacker. In particular, the former *fixes* which data it believes the latter can read and/or write. More formally, decentralized robustness is defined wrt a pair of principals p and q, with power $\langle R_{p\rightarrow q}, W_{p\leftarrow q}\rangle$, where $R_{p\rightarrow q}$ allows to characterise the data p believes that q can read, while $W_{p\leftarrow q}$ allows to characterise the data p believes that q can modify. A system is robust wrt all the attackers if it is robust with respect to all the pairs p, q of principals. In [5], the authors use type systems to enforce robustness against all attackers in a simple while language with holes and explicits declassification. Basically, it allows holes to occur in low confidentiality contexts and prevents attackers to influence both (explicit) declassification decision and data to be declassified as explained in [20]. Once we fix the point of view of the attacker, a safe hole insertion relation defines the admissible holes for the attacker in question together with variables he can modify and/or observe in the program.

4 Maximal Release by Active Attackers

The notion of robustness defined in Sect. 3 implicitly concerns the confidential information released by the program. Indeed, if we are able to measure the maximal release (the most concrete private property observable) in presence of active attacks, then we can compare it with the private information disclosed by passive attackers and conclude about program robustness. Thus, in this section we compute (when possible) the maximal private information disclosed by an active attacker.

The active attack model we use here is more powerful than the one defined in Sect. 3.2, i.e., fair attacks. In addition, our attackers can manipulate (use and modify) variables of security type `HL`, i.e., variables that the attacker cannot observe but can use. Indeed, `HL` is the type of those variables whose name is *visible*, i.e., usable by the attacker in his code, but whose value is not observable. Thus, in the following active attacks are programs (without holes) such that, for all the variables x occurring in the attacks code, $\mathcal{I}(x) = L$. We call them *unfair attacks*. *Unfair attacks* are more general than fair attacks because they can modify variables of security type `LL` and `HL`. For instance, suppose that a system user wants to change his password, he accesses a variable

(the password) he can write but not read (blind writing), i.e., of type HL. Now we want to compute the maximal information release in presence of unfair attacks.

4.1 Observing Input-Output

It is clear that, in order to certify the security degree of a program, also in presence of active attackers, it is important to compute which is the maximal private information released. Such information can help the programmer to understand what happens in the worst case, namely when an active attacker inserts the most harmful unfair code. Moreover, if we compute the most concrete property of private input data released by program semantics for all active attacks, we can compare it with the private information disclosed by a passive attacker and conclude about program robustness. In this section, we consider denotational semantics, namely input/output semantics. Hence, the set of program points where the attacker can observe low confidentiality data corresponds to program inputs and program outputs. Note that, the active attacker can insert code (fair or unfair) in fixed points, therefore he can change program semantics and consequently the property of confidential information released can be different in presence of different active attacks. Moreover, the number of possible unfair attacks may be infinite, thus, it becomes hard to compute the private information disclosed by all of them. The real problem is that it is impossible to characterise the maximal information released to attackers that modify program semantics, because different attacks obtain different private properties which may be incomparable if there are infinitely many such attacks.

This problem is overcome when we consider a finite number of attackers, for instance a finite class of attacks for which we want to certify our program. In this case, we can compute the maximal information disclosed by each attacker and, afterwards, we can consider the *greatest lower bound* (in the lattice of abstract domains) characterising the maximal information released for the fixed class of attackers. Let us introduce an example to illustrate the problem.

Example 1. Consider the program $P ::= l := h; [\bullet]$; with variables h : HH, l : LL and k : HL. We can have the following attacks:

- $Wlp\big(l := h; [skip], \{l = n\}\big) = \{h = n\}$
- $Wlp\big(l := h; [l := k], \{l = n\}\big) = \{k = n\}$
- $Wlp\big(l := h; [l := l + k], \{l = n\}\big) = \{h + k = n\}$

For all cases the attacker discloses different information about confidential data. In particular, in the first case the attacker obtains the exact value of variable h, in the second he obtains the exact value of variable k and in the third case he obtains a relation (the sum) between h and k. Note that if all possible active attacks were only those considered above, we can compute the greatest lower bound (*glb* for short) of private information disclosed by all of them. In this case *glb* corresponds to the identity value of confidential variables h and k.

However, as shown in the previous example, we can compute the private information disclosed by an attacker who fixes his attack and check if that particular attack

compromises program robustness. To this end, we just have to use the method introduced in [3] and verify that the method described in Sect. 3.4 holds for the transformed program.

In the previous example, we have seen that, even though we have a finite number of attackers, we have to compute a *Wlp* analysis for each active attacker. In the following, we suggest a method for computing only one analysis dealing with a (possible infinite) set of active attackers. We follow the idea proposed in [3], where, in order to avoid an analysis for each possible output observation, the authors compute the analysis parametrically on the symbolic output observation $l = n$. In particular, note that an attacker, being an imperative program, corresponds to a function manipulating low integrity variables, i.e., LL and HL variables.

Hence, we propose a *Wlp* computation parametric on the possible expressions $f(\vec{l})$ assigned by the active attacker to low integrity variables $\vec{l}$, which we call *attack schemas* (in line with program schemas [10]). In other words, the attacker can assign to all low integrity variables an expression which can possibly depend on all other low integrity variables. For instance, given a program where the only low integrity variables are l and k, all possible unfair attacks concern the variables l and k, namely $l := f(l, k)$ and $k = g(l, k)$, where f, g are expressions that can contain variables l, k free.

The confidential information released by such parametric computations can be exploited by both the programmer and the attacker. Indeed, looking at the final formula which can contain f as parameter, the former can detect vulnerabilities about the confidential information released by the program, while the latter can exploit such vulnerabilities to build the most harmful attack in order to disclose as much as possible about private input data. Let us introduce an example that shows the above technique.

Example 2. Consider the program in Ex. 1. The only low integrity variables are l : LL and k : HL. According to the method described above we have to substitute possible unfair attacks in $[\bullet]$ with attack schema $\langle l, k\rangle := \langle f(l, k), g(l, k)\rangle$. The initial formula is $\Phi_0 = \{l = n\}$ because l is the only program variable s.t. $\mathcal{C}(l) = \mathtt{L}$. Thus, the *Wlp* calculation yields the following formula:

$$\begin{gathered}\{f(h, k) = n\}\\ l := h;\\ \{f(l, k) = n\}\\ [\langle l, k\rangle := \langle f(l, k), g(l, k)\rangle;]\\ \{l = n\}\end{gathered}$$

Note that the final formula $(f(h, k) = n)$ contains information about high confidentiality variables h and k. Thus, fixing the unfair attacks as we did in the previous example, we obtain information about symbolic value of h, k or any relation between them.

It is worth pointing out that attack schemas capture pretty well the idea of classes of attacks which have a *similar* semantic effect (up to stuttering) on confidential information disclosed by an active attacker. We conjecture a close relationship between attack schemas and program schemas [10] and postpone their investigation as part of our future work.

4.2 Observing Program Traces

So far we have tried to compute the maximal private information disclosed by an active attacker which tampers with low integrity data in predefined program points (holes) and observes public input and public output of target program. In particular, the attacker could not observe low confidentiality data in any intermediate program point (random traces or holes). This condition is unrealistic in a mutual distrust scenario, where the attacker can control a compromised machine. Indeed, nothing prevents it to analyze low confidentiality data in points he is tampering code with and reveal secrets even though the overall computation has not terminated yet. This man-in-the-middle kind of attack requires to extend the analysis and consider intermediate program points as possible channels of information leakage. In many practical applications, it is common to have scenarios where a bunch of threads are running concurrently together with a malicious thread which reads the content of shared variables and dumps them in output each time the thread is scheduled to run.

In [18] the authors notice that the semantic model constitutes an important dimension for program security, the *where* dimension [22], which influences both the observation policy and the declassification policy. It seems obvious that an attacker who observes low confidentiality variables in intermediate program points is able to disclose more information than an attacker that observes only input/output. In this section, we aim to characterise the maximal information released by a program in presence of unfair attacks. In general, we can fix the set of program points where the attacker can observe low confidentiality variables (say $\mathbb{O}$) and we can denote by $\mathbb{H}$ the set of program points where there is a hole, namely where the attacker can insert malicious code. Moreover, we assume that the attacker can observe the low confidentiality variables for all program points in $\mathbb{H}$, namely $\mathbb{H} \subseteq \mathbb{O}$. In order to compute the maximal release of confidential information, an attacker can combine, at each observation point, the public information he can observe at that point together with the information he can derive by computing *Wlp* from the output to that observation point [18]. For instance, with trace semantics, an attacker can observe low confidentiality data for all intermediate program point. Let us introduce an example that presents this technique for passive attackers.

Example 3. Consider the program P with variables $l_1, l_2 : \texttt{LL}$ and $h_1, h_2 : \texttt{HH}$.

$$P ::= \begin{bmatrix} h_1 := h_2; h_2 := h_2 \bmod 2; \\ l_1 := h_2; h_2 := h_1; l_2 := h_2; l_2 := l_1; \end{bmatrix}$$

We want to compute the private information disclosed by an attacker that observes program traces. As for standard non-interference, here we want to protect private inputs h_1 and h_2. In order to make only one iteration on the program even when dealing with traces, the idea is to combine the *Wlp* semantics computed at each observable point of execution, together with the observation of public data made at the particular observation point. We denote in square brackets the value observed in that program point. The *Wlp* calculation yields the following result.

$$\{h_2 \bmod 2 = m \wedge h_2 = n \wedge l_2 = p \wedge l_1 = q\}$$
$$h_1 := h_2;$$
$$\{h_2 \bmod 2 = m \wedge h_1 = n \wedge l_2 = p \wedge l_1 = q\}$$
$$h_2 := h_2 \bmod 2;$$
$$\{h_2 = m \wedge h_1 = n \wedge l_2 = p \wedge [l_1 = q]\}$$
$$l_1 := h_2;$$
$$\{l_1 = m \wedge h_1 = n \wedge l_2 = p\}$$
$$h_2 := h_1;$$
$$\{l_1 = m \wedge h_2 = n \wedge [l_2 = p]\}$$
$$l_2 := h_2;$$
$$\{l_1 = m \wedge [l_2 = n]\}$$
$$l_2 := l_1;$$
$$\{l_1 = l_2 = m\}$$

For instance the information observed by the assignment $l_2 := l_1$ is the combination of *Wlp* calculation ($l_1 = m$) and attackers observation at that point ($[l_2 = n]$). The attacker is able to deduce the exact value of h_2. It is worth noting that this attacker is more powerful than the one who merely observes the input-output behavior; in fact, the latter can only distinguish the parity of variable h_2. This is made clear by the fact that the value of h_2's parity (m) is the value derived by the output, while the value of h_2 (n) is a value observed *during* the computation.

We would like to compute the maximal private information release in presence of unfair attacks. Here the problem is similar to the one described in the previous section. Unfair attacks, by definition, manipulate (modify and use) both variables of type LL and HL. Even though the attacker can observe low confidentiality variables in presence of holes, he cannot observe the variables of type HL. Hence, different unfair attacks cause different information releases, as it happens for attackers observing only the I/O, and in general there can be an infinite number of these attacks. However, if we fix the unfair attack we can use the method described above and compute the maximal release for that particular attack.

Things change when we consider only fair attacks, i.e., manipulating only LL variables. The following proposition shows that we can generalise all possible fair attacks to constant assignments $\vec{l} := \vec{c}$ to variables of type LL.

Proposition 1. *Let $P[\vec{\bullet}]$ be a program with holes and $\mathbb{H} \subseteq \mathbb{O}$. Then, all fair attacks can be written as $\vec{l} := \vec{n}$, where l : LL.*

Proof. In general, all fair attacks have the form $\vec{l} := f(\vec{l})$. Moreover, $\mathbb{H} \subseteq \mathbb{O}$ so the attacker can observe at least the program points where there is a hole. Thus, all the formal parameters of expression $f(\vec{l})$ are known. We conclude that $\vec{l} := \vec{n}$.

Now we are able to measure the maximal private information disclosed by an active attacker. Indeed, we can use the approach of Ex. 3 and whenever we have a program hole, we substitute it by the assignment $\vec{l} := \vec{c}$, parametric on symbolic constant values $\vec{c}$. The following example shows this method.

Example 4. Consider the program P with variables h : HH and l : LL. $\mathbb{O}$ is set of all program points.

$$P ::= l := 0; [\bullet];\ \textbf{if}\ (h > 0)\ \textbf{then skip else}\ l := 0;$$

In presence of passive attackers P does not release any information about private variable h. Indeed, the output value of variable l is always 0. An active attacker who observes each program point and injects fair attacks, discloses the following private information:

$$\begin{gathered}\{((h > 0 \wedge c = m) \vee (h \leq 0 \wedge m = 0)) \wedge c = n \wedge p = 0\}\\ l := 0;\\ \{((h > 0 \wedge c = m) \vee (h \leq 0 \wedge m = 0)) \wedge c = n \wedge [l = p]\}\\ [l := c;]\\ \{((h > 0 \wedge l = m) \vee (h \leq 0 \wedge m = 0)) \wedge [l = n]\}\\ \textbf{if}\ (h > 0)\ \textbf{then skip else}\ l := 0;\\ \{l = m\}\end{gathered}$$

Thus, an active attacker is able to disclose whether the variable h is positive or not. Hence, this is the maximal private information disclosed by an attacker who observes program traces and injects fair code in the holes.

5 Enforcing Robustness

In this section, we want to understand, by static program analysis, when an active attacker that transforms program semantics is not able to disclose *more* private information than a passive attacker, who merely observes public data. The idea is to consider *Wlp* semantics in order to find sufficient conditions which guarantee program robustness. Here we introduce a method to enforce programs which are robust in presence of active attackers.

We know [3] that declassified non-interference is a completeness problem in abstract interpretation theory and there exists systematic methods to enforce this notion. Let $P[\vec{\bullet}]$ be a program with holes and Φ a first order formula that models the declassification policy. In order to check robustness for this program, we must check the corresponding completeness problem for each possible attack a, as introduced in Sect. 3.4 where $P[\vec{a}]$ is program P under the attack $\vec{a}$. We want to characterise those situations where the semantic transformation induced by the active attack does not generate incompleteness. If there exists at least one attack a such that the program releases more confidential information than the one released by the policy, then the program is deemed not robust.

The following example shows the ability of active attackers to disclose more confidential information wrt passive attackers.

Example 5. Consider the program P with h : HH, l : LL.

$$\text{P} ::= l := 0; [\bullet]\ \textbf{if}\ (h > 0)\ \textbf{then}\ (l := 1)\ \textbf{else}\ (l := l + 1);$$

Suppose the declassification policy is $\top$, i.e., nothing has to be released. In presence of a passive attacker (the hole substituted by **skip**) program P satisfies the security policy, namely non-interference, because public output is always 1. *Wlp* semantics formalizes this fact.

$$\{(h > 0 \wedge n = 1) \vee (h \leq 0 \wedge n = 1)\} = \{n = 1\}$$
$$l := 0;$$
$$\{(h > 0 \wedge n = 1) \vee (h \leq 0 \wedge n = l + 1)\}$$
$$\textbf{if } (h > 0) \textbf{ then } (l := 1) \textbf{ else } (l := l + 1);$$
$$\{l = n\}$$

Now suppose that an active attacker inserts the code $l := 1$. In this case *Wlp* semantics shows that the attacker is able to distinguish positive values of private variable h from non positive ones. Using the *Wlp* calculation parametric on public output $\{l = n\}$ we have the following result.

$$\{(h > 0 \wedge n = 1) \vee (h \leq 0 \wedge n = 2)\}$$
$$l := 0;$$
$$\{(h > 0 \wedge n = 1) \vee (h \leq 0 \wedge n = 2)\}$$
$$[l := 1;]$$
$$\{(h > 0 \wedge n = 1) \vee (h \leq 0 \wedge n = l + 1)\}$$

The final formula shows that the adversary is able to distinguish values of h greater than 0 from values less or equal than 0 by observing, respectively, the values 1 or 2 of public output l. We can conclude that program P is not robust and the active attackers are effectively more powerful than passive ones.

If we had a method to compute the maximal private information release in presence of unfair attacks, then we could conclude about program robustness by comparing it with the information disclosed by a passive attacker. Unfortunately, in the previous section, we have seen that it is not possible to compute the maximal information released for all possible attacks, which can possibly be infinite. Hence, our aim is to look for methods enforcing robust programs without computing the maximal information released.

5.1 Robustness by *Wlp*

In this section we first distinguish between active attacks of different power and, afterwards, we present the proof of our approach to certify robust programs. The proof is organised as follows: it starts with a lemma that applies to sequential programs with one hole only, then we give a a theorem that generalizes the lemma to sequential programs with more holes and conclude with another theorem that applies the robustness condition to all terminating while programs.

Let us make some considerations about logical formulas and the set of program states they manipulate. The free variables of the output observation formula Φ_0 correspond to the set of low confidentiality variables LL and LH, namely

$$\mathcal{FV}(\Phi_0) = \{x \in \mathit{Var}(\Phi_0) | \mathcal{C}(x) = \mathtt{L}\}.$$

If a low confidentiality variable does not occur free at some program point, it means that such variable was previously, wrt backward analysis of *Wlp* semantics, substituted by an expression that does not contain that variable. This means that, it can have any value in that point. From the viewpoint of information flow, even if the variable contains some confidential information in that point this is useless for the analysis, because the variable is going to be subsequently overwritten and therefore this information can never be disclosed through public outputs.

Our aim is to generalise the most powerful active attacks and study their impact on program robustness. As a first approach one can try to represent all possible active attacks by a constant assignment to low integrity variables. Hence, the attacker observes only the input/output value of low confidentiality variables, i.e., LL and LH variables. The following example shows that this is not sufficient enough and there exist more powerful attacks that disclose more private information and break robustness.

Example 6. Consider the program P with variables $l : \texttt{LL}$, $k : \texttt{LL}$, $h : \texttt{HH}$ and declassification policy that releases nothing about private variables.

$$P ::= \begin{bmatrix} k := h; [\bullet]; \\ \textbf{if } (l = 0) \quad \textbf{then } (l := 0; k := 0) \\ \qquad\qquad\ \textbf{else } (l := 1; k := 1); \end{bmatrix}$$

First notice that P does not release private information in presence of a passive attacker who merely observes the I/O variation of public data. Indeed, the assignment of h to k is subsequently overwritten by constants 0 or 1 and depends exclusively on the variation of public input l. If it was possible to represent all active attacks by constant assignments we can see that P would be robust. In fact, if the attacker assigns constants c_1 and c_2, respectively, to variables l and k, *Wlp* calculation deems the program robust.

$$\begin{gathered} \{(c_1 = 0 \wedge m = 0 \wedge n = 0) \vee (c_1 \neq 0 \wedge m = 1 \wedge n = 1)\} \\ k := h; \\ [l := c_1; k := c_2;] \\ \{(l = 0 \wedge m = 0 \wedge n = 0) \vee (l \neq 0 \wedge m = 1 \wedge n = 1)\} \\ \textbf{if } (l = 0) \textbf{ then } (l := 0; k := 0) \textbf{ else } (l := 1; k := 1); \\ \{l = m \wedge k = n\} \end{gathered}$$

The final formula shows that such program satisfies non-interference. But if we assign to low integrity variables an expression depending on other low integrity variables, then we obtain more powerful attacks, which make P not robust. For instance, the assignment $a ::= l := k;$ makes the attacker distinguish the zeroness of private variable h.

$$\begin{gathered} \{(h = 0 \wedge m = 0 \wedge n = 0) \vee (h \neq 0 \wedge m = 1 \wedge n = 1)\} \\ k := h; \\ \{(k = 0 \wedge m = 0 \wedge n = 0) \vee (k \neq 0 \wedge m = 1 \wedge n = 1)\} \\ [l := k;] \\ \{(l = 0 \wedge m = 0 \wedge n = 0) \vee (l \neq 0 \wedge m = 1 \wedge n = 1)\} \end{gathered}$$

Definitely, program P is not robust and therefore we cannot reduce active attacks to a constant assignment to low integrity variables.

In general, an active attack is a piece of code that concerns low integrity variables, i.e., a function manipulating low integrity variables. If we assign to low integrity variables a constant value then we erase the high confidentiality information that this variables might have accumulated before reaching that point or we are not considering the possibility of assigning to that variable another one which contains some private information that possibly may be lost subsequently as shown in Ex. 6.

We can use the ideas discussed so far to present a *sufficient* condition ensuring program robustness. Remember that we represent formally the observable public output as a first order formula, Φ_0, that corresponds to the conjunction of program variables x such that $\mathcal{C}(x) = \mathtt{L}$, parametric on the observed public outputs n_i, namely

$$\Phi_0 = \bigwedge_{i=1}^{k} (l_i = n_i) \text{ and } \forall i.\mathcal{C}(l_i) = \mathtt{L}.$$

In particular, we first describe how to characterise the sufficient condition when the holes are not nested in control structures. This is obtained in two steps, the lemma shows the result for programs with only one hole, while the first theorem extends the result to programs with an arbitrary number of holes. Afterwards, we show how to exploit this result in order to characterise the sufficient condition to robustness also when holes are nested in control structures.

In the following, we denote by $\bullet_i$ the i-th hole in P and by P_i the portion of code in P after the hole $\bullet_i$ where all the following holes ($\bullet_j$, with $j \in \mathbb{H}$, $j > i$) are substituted with **skip**. Then $\Phi_i = \mathit{Wlp}(P_i, \Phi_0)$ is the formula corresponding to the execution of the subprogram P_i.

Lemma 1. *Let $P = P_2; [\bullet]; P_1$ be a program (P_1 without holes, possibly empty). Let $\Phi = \mathit{Wlp}(P_1, \Phi_0)$. Then P is robust wrt unfair attacks if $\forall v \in \mathcal{FV}(\Phi).\mathcal{I}(v) = \mathtt{H}$.*

Proof. We prove this theorem by induction on the attack's structure and on the length of its derivation. In particular, we prove that for any attack a, $\mathit{Wlp}(a, \Phi) = \Phi$, namely the formula Φ does not change, hence from the semantic point of view, the attack behaves like **skip**, namely like a passive attacker. Note that, here we consider unfair attacks, hence it can use both LL and HL variables.

- a ::= **skip**: The initial formula Φ does not change, namely $\mathit{Wlp}(\textbf{skip}, \Phi) = \Phi$, and the attacker acts as a passive one.
- a ::= l := e: By definition of active attack we have $\mathcal{I}(l) = \mathtt{L}$ and by hypothesis variable l does not occur free in Φ. Applying the *Wlp* definition for assignment, we have $\mathit{Wlp}(l := e, \Phi) = \Phi[e/l] = \Phi$.
- a ::= $c_1; c_2$: By inductive hypothesis we have $\mathit{Wlp}(c_1, \Phi) = \mathit{Wlp}(c_2, \Phi) = \Phi$ as attacks of minor length. The *Wlp* definition for sequential composition states that $\mathit{Wlp}(c_1; c_2, \Phi) = \mathit{Wlp}(c_1, \mathit{Wlp}(c_2, \Phi)) = \Phi$
- a := **if** B **then** c_1 **else** c_2: By inductive hypothesis (applied to an attack of minor length) we have $\mathit{Wlp}(c_1, \Phi) = \mathit{Wlp}(c_2, \Phi) = \Phi$. Applying the definition of *Wlp* for the conditional construct $\mathit{Wlp}(\textbf{if } B \textbf{ then } c_1 \textbf{ else } c_2, \Phi) = (B \wedge \mathit{Wlp}(c_1, \Phi)) \vee (\neg B \wedge \mathit{Wlp}(c_2, \Phi)) = (B \wedge \Phi) \vee (\neg B \wedge \Phi) = \Phi$.

– $a ::=$ **while** B **do** c: By hypothesis we consider terminating computations, so the **while** loop halts in a finite number of iterations. Applying the inductive hypothesis to command c we have $\mathit{Wlp}(c, \Phi) = \Phi$, so every iteration the formula does not change. Moreover, if the guard is *false* the formula remains unchanged too. Applying *Wlp* rule for the **while** loop and the inductive hypothesis we have:
$\mathit{Wlp}(\textbf{while } B \textbf{ do } c, \Phi) = (\neg B \wedge \Phi) \vee (B \wedge \Phi) \vee \cdots \vee (B \wedge \Phi) \vee (B \wedge \Phi) = \Phi$

Theorem 1. *Let $P[\vec{\bullet}]$ be a program. Then we say that P is robust wrt unfair attacks if* $\forall i \in \mathbb{H}. \forall v \in \mathcal{FV}(\Phi_i). \mathcal{I}(v) = \mathtt{H}$.

Proof. Suppose P has n holes:

$$P \equiv P'_{n+1}; [\bullet_n]; P'_n \dots P'_2; [\bullet_1]; P'_1$$

Let us define the following programs from $1 \leq i \leq n+1$

$$P_i \stackrel{\text{def}}{\equiv} \begin{cases} P'_1 & \text{if } i = 1 \\ P'_i P_{i-1} & \text{otherwise} \end{cases}$$

Namely P_i is the portion of code in P after the hole $\bullet_i$ where all the following holes ($\bullet_j$, with $j \in \mathbb{H}$, $j > i$) are substituted with **skip**. We prove by induction on n that $\forall 1 \leq i \leq n.\ P'_{i+1}; [\bullet_i]; P'_i; [\bullet_{i-1}]; \dots; [\bullet_1]; P'_1$ is robust wrt unfair attacks. By proving this fact, we prove the thesis since when $i = n$ we obtain exactly P.

BASE: Consider the first hole from the end of the program P, i.e., $P'_2; [\bullet_1]; P'_1$. Then by Lemma 1 we have that $P'_2; [\bullet_1]; P'_1$ is robust, being P'_1 without holes by construction. This implies that any active attacker can disclose the same information as the passive (**skip**) attacker can do, hence $\bullet_1$ can be substituted with **skip**, namely $P'_2[\bullet_1]P'_1$ can be substituted by P_2 in P without changing the robustness property of P.

INDUCTIVE STEP: Suppose, by inductive hypothesis, that $P'_i; [\bullet_{i-1}]; P'_{i-1}; \dots; P'_2; [\bullet_1]; P'_1$ is robust. This means that, exactly as we noticed in the base of the induction, the holes are useless for an attacker, therefore we can substitute all the $\bullet_j$ with **skip** obtaining a program (from the robustness point of view) equivalent to P_i. Hence, $P'_{i+1}; [\bullet_i]; P'_i; [\bullet_{i-1}]; \dots; [\bullet_1]; P'_1 \equiv P'_{i+1}; [\bullet_i]; P_i$, and robustness of this last program holds by Lemma 1, being P_i without holes by construction.

In this way we prove that $P \equiv P'_{n+1}; [\bullet_n]; P_n$ is robust.

In other words, the fact that a low integrity variable is not free in the formula means that the information in the corresponding program point cannot be exploited for revealing confidential properties. In this case we can say that a generic active attacker is not stronger than a passive one. Before showing what happens for control structures, let us introduce an example that illustrates Th. 1.

Example 7. Let us check robustness of program P with variables l : LL, h : HH and k : *HL*.

$$P ::= \begin{bmatrix} l := h + l; [\bullet]; l := 1; k := h; \\ \textbf{while } (h > 0) \textbf{ do } (l := l - 1; l := h); \end{bmatrix}$$

Analysing P from the hole $[\bullet]$ to the end we have:

$$\{(h \leq 0 \wedge n = 1) \vee (h > 0 \wedge n = 0)\}$$
$$l := 1; k := h;$$
$$\{(h \leq 0 \wedge l = n) \vee (h > 0 \wedge n = 0)\}$$
$$\textbf{while } (h > 0) \textbf{ do } (l := l - 1; l := h);$$
$$\{l = n\}$$

The formula $\Phi = (h \leq 0 \wedge n = 1) \vee (h > 0 \wedge n = 0)$ satisfies the conditions of Th. 1. We can conclude the program P is robust. Intuitively, even though the value of private input h flows to public variable l ($l := l + h$), such relation is immediately canceled when we assign the constant 1 ($l := 1$) after the hole.

The following example shows that Th. 1 is just a sufficient condition, namely there exists a robust program that violates the preconditions. This is because Th. 1 corresponds to a local condition for robustness, but one must analyze the entire program in order to have a global vision about the confidential information revealed.

Example 8. Consider the program

$$P ::= \begin{bmatrix} l := h; l := 1; [\bullet]; \\ \textbf{while } (h = 0) \textbf{ do } (h := 1; l := 0); \end{bmatrix}$$

where h : HH and l : LL. The precondition of the **while** is:

$$\mathit{Wlp}\,(\textbf{while } (h = 0) \textbf{ do } (h := 1; l := 0), \{l = n\}) =$$
$$\{(h = 0 \wedge n = 0) \vee (h \neq 0 \wedge l = n)\}$$

This formula does not satisfy the conditions of Th. 1, since it contains a free occurrence of a low integrity variable, namely $l = n$. However, we can see that program P is robust. No modification of the public variable l contains information about the private variable h because the guard of the **while** loop depends exclusively on private variables. Every terminating attack modifies the subformula $\{l = a\}$ and influences the final value of the observed public output. Moreover, the private information obtained by the assignment $l := h$ is canceled by the successive assignment $l := 1$. So the only confidential information released by P concerns the zeroness of h, the same as a passive attacker. This means that P is robust and Th. 1 is a sufficient and not necessary condition for robustness.

Let us show, now, how Theorem 1 applies to programs where the hole occurs in the branch of a conditional or in a loop. As the following theorem shows, in such cases we need to apply recursively Theorem 1 to the formula corresponding to each branch. It is worth noting that the loop can be unfolded a finite number of times until we reach the invariant formula (see the *Wlp* rule for **while** in section 3.3), as the computations we are dealing with are all terminating ones.

Theorem 2. *Let* $P_c[\vec{\bullet}] \equiv$ **if** B **then** $P_1[\vec{\bullet}]$ **else** $P_2[\vec{\bullet}]$ *and* $P_w[\vec{\bullet}] \equiv$ **while** B **do** $P[\vec{\bullet}]$ *be a program with holes and a first order formula* Φ*. Then,*

- $P_c[\vec{\bullet}]$ *is robust wrt unfair attacks iff* $P_1[\vec{\bullet}]$ *and* $P_2[\vec{\bullet}]$ *are robust wrt unfair attacks and post-condition* Φ*.*

– *$P_w[\vec{\bullet}]$ is robust wrt unfair attacks iff $P[\vec{\bullet}]$ is robust wrt unfair attacks and post-conditions $Wlp_i(P_w[\vec{\bullet}], \Phi)$*

Proof. We do induction on the structure of $P_1[\vec{\bullet}]$; the other case is symmetric. If $P_1[\vec{\bullet}]$ straight line program with holes (as in the hypothesis of Theorem 1), we apply the theorem to check robustness. Otherwise, $P_1[\vec{\bullet}]$ is a conditional and it trivially holds from the induction hypothesis.

In the case of a loop we need to apply the recursive computation as described in section 3.3. If $P[\vec{\bullet}]$ is a straight line program we apply theorem 1 as before and check at each step of *Wlp* computation whether low integrity variables occur in the formula when we reach the hole. Note that the occurrence of the loop guard B in the formula makes sure that the active attacker never influences the variables of B. In this way, we are sure that if the condition is verified the formula remains unchanged for all active attacks. Otherwise, if $P[\vec{\bullet}]$ is a loop or a conditional we apply the induction hypothesis and we are done.

The result above shows how to treat situations where the construct $[\bullet]$ may be placed in an arbitrary depth inside an **if** conditional or a **while** loop. The following example describes this situation.

Example 9. Consider the program P

$$P ::= \begin{bmatrix} k := h \bmod 3; \\ \textbf{if } (h \bmod 2 = 0) \textbf{ then}[\bullet]; l := 0; k := l; \\ \qquad\qquad\qquad\quad \textbf{else } l := 1; \end{bmatrix}$$

where h : HH, l : LL and k : LL. Applying the weakest liberal precondition rules to the initial formula $\{l = m \wedge k = n\}$ we have:

$$\left\{ \begin{array}{l} (h \bmod 2 = 0 \wedge m = 0 \wedge n = 0) \vee \\ \quad (h \bmod 2 \neq 0 \wedge m = 1 \wedge k = n) \end{array} \right\}$$
$$\textbf{if } (h \bmod 2 = 0) \textbf{ then } [\bullet]; l := 0; k := l; \textbf{ else } l := 1;$$
$$\{l = m \wedge k = n\}$$

The subformula corresponding to the **then** branch (which contains the hole $[\bullet]$) satisfies the conditions of Th.1, therefore P is robust. Every possible attack in this point manipulates the variables l, k which will immediately be substituted by constant 0 and will lose all private information they have accumulated so far.

Note that the invariant enforced by the theorems is the fact that the first order formula determined by the active attack remains inalterate compared to the formula determined by the passive attack. In particular, Theorem 1 proves our security condition, while Theorem 2 model the fact that such condition should be applied recursively in case of conditionals and loops. In the next section we present and algorithmic approach that puts all the pieces together.

5.2 An Algorithmic Approach to Robustness

In this section we present our approach algorithmically in order to make clear how the above theorems apply to terminating while programs. In particular, $Robust(P[\vec{\bullet}], \Phi, \mathcal{S})$

is the main procedure that takes in input a program with holes $P[\vec{\bullet}]$, a first order formula Φ and a set of low integrity variables S and if it returns a formula, the program is robust and such formula corresponds to the private information disclosed to both passive and active attackers, otherwise (if it returns false) we don't know whether the program is robust or not. The procedure $Check(\Phi, \mathcal{S})$ corresponds to our security condition, namely, it returns true if no low integrity variables in $\mathcal{S}$ occur in Φ as well. Moreover, we assume that we have a procedure that transforms a first order formula in the normal form in order to reduce the false alarms in our analysis. The algorithm runs recursively over the syntactical structure of while programs (with holes) and applies, at each step, the rules of Wlp semantics, as described in section 3.3. The procedure $Compute(\textbf{while}\ B\ \textbf{do}\ c, \Phi, \mathcal{S})$ checks whether the formula remains unchanged for the while loop. In particular, this corresponds to the unfolding of the loop, with a finite number of conditionals. In particular, it applies a finite number of times the security condition of Theorem 2.

$$Robust(P[\vec{\bullet}], \Phi, \mathcal{S}) :$$

$$\begin{array}{|ll|}
\hline
\textbf{case}(P[\vec{\bullet}]) : & \\
 & \\
[\bullet] : & Check(\Phi, \mathcal{S}) \\
\textbf{skip} : & \Phi \\
x := e : & \Phi[e/x] \\
P_1[\vec{\bullet}]; P_2[\vec{\bullet}] : & \Phi' := Robust(P_2[\vec{\bullet}], \Phi, \mathcal{S}) \\
 & Robust(P_1[\vec{\bullet}], \Phi', \mathcal{S}) \\
\textbf{if}\ B\ \textbf{then}\ P_1[\vec{\bullet}]\ \textbf{else}\ P_2[\vec{\bullet}] : & (B \wedge Robust(P_1[\vec{\bullet}], \Phi, \mathcal{S})) \vee \\
 & (\neg B \wedge Robust(P_2[\vec{\bullet}], \Phi, \mathcal{S})) \\
\textbf{while}\ B\ \textbf{do}\ P_1[\vec{\bullet}] : & Compute(\textbf{while}\ B\ \textbf{do}\ P_1[\vec{\bullet}], \Phi, \mathcal{S}) \\
\hline
\end{array}$$

$$Check(\Phi, \mathcal{S}) : \begin{array}{|ll|}
\hline
\text{Normalize the formula } \Phi & \\
\textbf{if}\ \mathcal{FV}(\Phi) \cap \mathcal{S} = \emptyset & \text{return true} \\
\text{otherwise} & \text{return false} \\
\hline
\end{array}$$

$$Compute(\textbf{while}\ B\ \textbf{do}\ c, \Phi, \mathcal{S}) : \begin{array}{|l|}
\hline
\Phi_{i+1} := \neg B \wedge \Phi \\
result := \Phi_{i+1} \\
\textbf{do} \\
\quad \Phi_i := \Phi_{i+1} \\
\quad \Phi_{i+1} := Robust(\textbf{if}\ B\ \textbf{then}\ c\ \textbf{else skip}, \Phi_i, \mathcal{S}) \\
\quad result \vee := \Phi_{i+1} \\
\textbf{while}\ \Phi_i \neq \Phi_{i+1} \\
\hline
\end{array}$$

5.3 Robustness on Program Traces

In this section, we want to find local conditions guaranteeing robustness also in presence of active attackers which observe trace semantics instead of I/O semantics. In other words, we want to characterise the analogous of Th. 1 when dealing with trace semantics. Note that, in this case, the problem becomes really different because the attacker is

still able to modify low integrity variables, but he can also *observe* low confidentiality variables in the holes. In this case, the problem is that the attacker can assign variables of type HL to variables of type LL, observe the corresponding trace and disclose immediately the value of HL variables. Hence, it is necessary to analyse the global program behavior in order to check robustness for all possible unfair attacks. On the other hand, if we consider fair attacks, i.e., attacks that manipulate only LL variables, the attackers capability to observe program points where the hole occurs allows us to reduce all the possible attacks to constant assignments to variables of type LL.

By using the method introduced in [18], illustrated for active attackers in Sect. 4.2, we are able to state a sufficient condition of robustness in presence of fair attacks for trace semantics. The idea is that an attacker can combine the public information he can observe at a program point together with the information he can derive by computing the *Wlp* from the output to that observation point. Moreover, he can manipulate program semantics by inserting fair code in the holes. If the formula corresponding to *Wlp* semantics of the subprogram before reaching the hole does not contain free any variables of type LL then we can conclude that the program is robust. The following example shows the robustness condition similar to Th. 1.

Example 10. Consider the program P with variables l : LH, k : LL and h_1, h_2, h_3 : HH:

$$P ::= k := h_1 + h_2; [\bullet]; k := h_3 \bmod 2; l := h_3; l := k;$$

A passive attacker who observes each program point discloses the following private information.

$$\begin{gathered}
\{h_3 \bmod 2 = m \wedge h_3 = n \wedge l = p \wedge h_1 + h_2 = q\} \\
k := h_1 + h_2; \\
[\mathit{skip};] \\
\{h_3 \bmod 2 = m \wedge h_3 = n \wedge l = p \wedge [k = q]\} \\
k := h_3 \bmod 2; \\
\{k = m \wedge h_3 = n \wedge [l = p]\} \\
l := h_3; \\
\{k = m \wedge [l = n]\} \\
l := k; \\
\{l = k = m\}
\end{gathered}$$

Hence, a passive attacker reveals the symbolic value of variable h_3 and the sum of variables h_1 and h_2. In what follows we notice that no fair attack (in our case manipulating k) can do better, because the subformula corresponding to the information disclosed by the attacker does not contain free the variable k : LL. Thus, no constant assignment influences the private information released. Indeed, if we compute the information disclosed in presence of a fair attack the final formula is the same.

$$\begin{gathered}
\{h_3 \bmod 2 = m \wedge h_3 = n \wedge l = p \wedge h_1 + h_2 = r\} \\
k := h_1 + h_2; \\
\{h_3 \bmod 2 = m \wedge h_3 = n \wedge l = p \wedge q = d_1 \wedge [k = r]\} \\
[k := d_1;] \\
\{h_3 \bmod 2 = m \wedge h_3 = n \wedge l = p \wedge [k = q]\}
\end{gathered}$$

Note that, it is useless to consider the observed value of LL variable before the hole because the attacker knows exactly what fair attack he is going to inject in.

Now we can introduce a sufficient condition for robustness for trace semantics. Basically, the idea is to propose an extension of Th. 1 to traces. We have first to note that in Th. 1 we deal with unfair attackers, which can use also variables of type HL. In the trace semantics context this may be a problem whenever attackers can observe low confidentiality data in at least one point where they can inject their code, i.e., if $\mathbb{H} \cap \mathbb{O} \neq \varnothing$. In particular what may happen is that the attacker can use variables of type HL and observe the result at the same time, possibly disclosing the value of these variables. This clearly means that the program is trivially not robust as shown in the following example.

Example 11. Consider the program

$$P := l := k \ mod\ 2; [\bullet];\ \textbf{if}\ (h = 0)\ \textbf{then}\ l := 0\ \textbf{else}\ l := 1;$$

where l : LL, k : HL and h : HH. We want to check robustness in presence of unfair attacks who observe each program point. First, we notice that a passive attacker discloses the zeroness of variable h and the parity of variable k. Now let us compute the information released in the hole.

$$\begin{gathered}\{(h = 0 \wedge n = 0) \vee (h \neq 0 \wedge n = 1)\} \\ \textbf{if}\ (h = 0)\ \textbf{then}\ l := 0\ \textbf{else}\ l := 1; \\ \{l = n\}\end{gathered}$$

This formula satisfies the conditions of Prop. 2: no low integrity variables occur free in it. But, if we attack this program with the unfair attack (e.g., $l := k$), we can see that the program releases the exact symbolic value of the private variable k.

$$\begin{gathered}\left\{\begin{array}{c}((h = 0 \wedge n = 0) \vee (h \neq 0 \wedge n = 1)) \wedge \\ k = p \wedge k \ mod\ 2 = q\end{array}\right\} \\ l := k \ mod\ 2; \\ \{((h = 0 \wedge n = 0) \vee (h \neq 0 \wedge n = 1)) \wedge k = p \wedge [l = q]\} \\ [l := k;] \\ \{((h = 0 \wedge n = 0) \vee (h \neq 0 \wedge n = 1)) \wedge [l = p]\}\end{gathered}$$

We can conclude that program P is not robust (wrt unfair attacks) even though the conditions of Th. 1 are satisfied.

At this point we can provide, in the following proposition, the robustness sufficient condition that has to hold for traces, depending on the relation between hole points $\mathbb{H}$ and observable points $\mathbb{O}$. In particular, if we consider attackers that observe low confidentiality data in at least one hole point, i.e., $\mathbb{H} \cap \mathbb{O} \neq \varnothing$, then we can prove robustness only wrt fair attacks, otherwise we can consider general unfair attackers. In fact, when $\mathbb{H} \cap \mathbb{O} = \varnothing$ the attackers cannot combine their capabilities of observing low confidentiality variables and of modifying low integrity variables, making possible to guarantee robustness.

Proposition 2. *Consider* $P[\vec{\bullet}]$ *and* $\Phi_i = Wlp(P_i, \Phi_0)$ *(where* P_i *is obtained as in Th. 1). Then we have that:*

1. *If* $\mathbb{H} \cap \mathbb{O} \neq \varnothing$ *then* P *is robust wrt fair attacks if* $\forall i \in \mathbb{H}.\forall v \in \mathcal{FV}(\Phi_i).\mathcal{I}(v) = \mathtt{H}$.
2. *If* $\mathbb{H} \cap \mathbb{O} = \varnothing$ *then* P *is robust wrt* unfair *attacks if* $\forall i \in \mathbb{H}.\forall v \in \mathcal{FV}(\Phi_i).\mathcal{I}(v) = \mathtt{H}$.

Proof. Consider the program P. First of all note that the difference between observing I/O semantics and trace semantics consists simply on the fact that the attacker can enrich the *Wlp* analysis with the observation that it can perform during the computation. Hence, we can define an enriched weakest precondition semantic function: $\mathit{Wlp}'(c, \phi) \stackrel{\text{def}}{=} \mathit{Wlp}(c, \phi \wedge \phi')$, where $\phi' = \mathit{true}$ if the corresponding program point is not in $\mathbb{O}$, ϕ' is the observable property otherwise. At this point, by using Wlp' instead of *Wlp* we can apply Th. 1 with the following restrictions:

1. If $\mathbb{H} \cap \mathbb{O} \neq \varnothing$ then the attacker can use variables of type HL and observe the result at the same time, disclosing the HL variables and violating robustness. In particular, if the program has l : LL and k : HL, then the attacker can always insert the code $l := k$, and by observing the result can directly know the value of k violating confidentiality and, obviously, robustness. This is not a problem for fair attackers, since these attackers cannot use variables of type HL.
2. If $\mathbb{H} \cap \mathbb{O} = \varnothing$ then the unfair attacker cannot observe the result of the added code and therefore robustness can again hold, at least when the sufficient condition of Th. 1 is satisfied.

5.4 *Wlp* vs. Security Type System

In [20] the authors define the notion of robustness in presence of active attackers and enforce it by using a security type system. The active attacker can replace the holes by fair attacks which manipulate variables of security type LL. The key result of the article states that typable programs satisfy robust declassification. Thus, it is important, when dealing with robustness, for the holes not to be placed into high confidentiality contexts. In particular they introduce a security environment and a program counter pc in order to trace the security contexts and avoid implicit flows. The following typing rule considers cases where the construct $[\bullet]$ is admissible.

$$\frac{\mathcal{C}(\mathrm{pc}) \in L_C}{\Gamma, \mathrm{pc} \vdash \bullet}$$

Let A be the attacker code, then $L_C \stackrel{\text{def}}{=} \{l | \mathcal{C}(l) \sqsubseteq \mathcal{C}(A)\}$, namely L_C is the set of variables whose confidentiality level is not greater than attackers confidentiality level. Hence, an active attacker that manipulates this variables does not obtain further confidential information. The type system is highly imprecise with respect to standard non-interference since it rules out all programs containing low assignments under high guards or any sub-command with an explicit assignment from high to low. Basically, the type system admits programs that *associate* low with low, high with high and do not use high expressions on guards of conditionals or loops. This corresponds to a trace-based characterization of non-interference where the attacker can observe the content of low variables in each program point. Now if we ignore the explicit declassification (*declassify(e)*) and consider only programs with holes, the typing rule for the hole requires them to occur in low confidentiality security context, namely program is robust

if there is no interaction between high and low, neither explicit nor implicit and this is quite restrictive. Getting back to explicit declassification, the rule requires it occurs in low confidentiality and high integrity program context, namely the guard of a conditional or a loop is allowed to evaluate only on variables of security type LH if we want to embed declassification. Moreover, only high integrity variables can be declassified, i.e., declassification from variables of security type HH to variables of security type LH is allowed. Putting all together, the type system approach deems robust programs that never branch on a secret value (unless each branch assigns only to high) and admit explicit flow (from high to low) in certain program points because of declassification.

Our approach, in particular Th. 2, captures exactly those situations where the hole occurs in some confidentiality context (possibly high) and, nevertheless, the fair attack does not succeed, namely where there are no low integrity variables in the corresponding first order formula. If our condition holds, we are more precise to capture the main goal of robustness, i.e., an active attacker does not disclose more private information than a passive one, as we perform a flow sensitive analysis. Indeed, if the target program has some intended global interference (the *what* dimension in [22]), the type system is unable to model it (as it considers the *where* dimension in [22]), while our approach characterizes robustness with respect to a program and a global declassification policy. Moreover, our method deals with more powerful active attacks, the unfair attacks, which can manipulate code that contains variables with security type LL and HL. However, both these approaches study program robustness as a local condition and therefore cannot provide a precise characterisation of robustness: Th. 2 provides only a sufficient condition and the type system is not complete. Anyway, we can say that, when it can be applied, namely when the hypotheses of the theorems hold, then our semantic-based method is more precise, in the sense that it generates less *false alarms*, than the type-based one. For instance, let us consider the program $P ::= [\bullet]; \textbf{if } h > 0 \textbf{ then } l := 0 \textbf{ else } l := 0$ where $h : \texttt{HH}$ and $l : \texttt{LL}$. Our method certifies this program as robust since, there are no low integrity variables in the formula corresponding to the *Wlp* semantics of the control statement **if**. If we try to type check this program by using the rules in [20] we notice that the environment before hole is a high confidentiality one. Thus, this program is deemed not robust.

We have, anyway, to note that our approach, if compared with the type-based one, loses effectiveness in order to keep precision, i.e., in order to reduce false alarms. Indeed, in the future, in order to make our certification approach systematic we will surely have to weaken the semantic precision.

6 Relative Robustness

So far, we have given only sufficient conditions to enforce robust programs. The problem is that an active attacker transforms program semantics and these transformations can be infinitely many or of infinitely many kinds. This may be an issue, first of all because it becomes hard to compute the private information released by all the active attacks (as underlined in Sect. 4), but also because, in some restricted contexts, standard robustness can be too strong a requirement.

Indeed, we can consider a restricted class of active attacks and check robustness wrt to these attacks. Namely, we aim to check whether the program, in presence of these attacks, does not release more private information than a passive attacker. Thus, we define a relaxed notion of robustness, called *relative robustness*.

Definition 1. *Let $P[\vec{\bullet}]$ be a program and $\mathcal{A}$ a set of attacks. The program is said relatively robust iff for all $\vec{a} \in \mathcal{A}$, then $P[\vec{a}]$ does not release more confidential information than $P[\overrightarrow{skip}]$.*

Recall that we model the information disclosed by the attacker by first order formulas, which we interpret by means of abstract domains in the lattice of abstract interpretations as explained in section 3.4. In particular, if the attacker a_1 discloses more private information than attacker a_2, it means that the abstract domain corresponding to the private property revealed to a_1 is contained in the abstract domain corresponding to the private property relealed to a_2.

In order to check relative robustness we can compute the confidential information released for all possible attacks, compute the greatest lower bound of all information and compare it with the confidential information released by a passive attacker. Moreover, given a program and a set of attacks we can statically certify the security degree of the program with respect to that particular finite class of active attacks. This corresponds to the *glb* of private information released by all these attacks. Hence, a programmer who wants to certify program robustness in presence of a fixed class of attacks, have to declassify at least the *glb* of private information disclosed by all attacks.

Consider Ex. 1. We noticed that different active attackers can disclose different kind of private information, for this reason the program P is not robust. Now, consider a restriction of the possible active attacks, for example we restrict to fair attacks only. This implies that the attacker can use only variable l and derive information exclusively about private variable h. In particular, P already releases in l the exact value of h and consequently no attack involving variable l can disclose more private information. Thus, we can conclude that program P satisfies *relative robustness* with respect to the class of fair attacks.

We can extend Th. 1 in order to cope with relative robustness. In particular, we recall that this theorem provides a sufficient condition to robustness requiring that the formulas before each hole do not contain *any* low integrity variable. We weaken this sufficient condition by requiring that the formulas before each hole do not contain *only* the variables modifiable and usable by the attackers in $\mathcal{A}$. It is worth noting that both Prop. 3 and Prop. 4 are easily extended to programs with more holes occuring at different depths, exactly the same way as we derived Th. 1 from Lemma 1 and Th. 2 from Th. 1. Next proposition is a rewriting of Lemma 1 for relative robustness.

Proposition 3. *Let $P = P_2; [\bullet]; P_1$ be a program (where P_1 is without holes). Let $\Phi = \mathit{Wlp}(P_1, \Phi_0)$. P is relatively robust wrt unfair attacks in $\mathcal{A}$ if $\forall a \in \mathcal{A}. \mathit{Var}(a) \cap \mathcal{FV}(\Phi) = \varnothing$.*

Proof. Note that the variables used by the active attacker do not occur free in Φ as the intersection is empty (by hypothesis). By Lemma 1 program P is robust.

It is worth noting that we can use this result also for deriving the class of *harmless* active attackers starting from the semantics of the program. Indeed, we can certify that a

program is relatively robust wrt all the active attackers that involve low integrity variables not occurring free in the formulas corresponding to the private information disclosed before reaching each hole.

6.1 Relative vs. Decentralized Robustness

In this section, we claim that, from certain viewpoints, relative robustness is a more general notion than decentralized robustness. The reasons are the same as the ones discussed in 5.4. In a nutshell, we can observe that, once the pair of principals is fixed, also the data security levels are fixed, namely we know which are the variables readable and/or modifiable by the attacker q from the point of view of a principal p. We can denote by $\mathcal{C}_{p\to q}$ the confidentiality levels and by $\mathcal{I}_{p\leftarrow q}$ the integrity levels characterised so far. For instance, for each variable x, $\mathcal{I}_{p\leftarrow q}(x) = \mathtt{L}$ if p believes that q can modify x, $\mathcal{I}_{p\leftarrow q}(x) = \mathtt{H}$ otherwise. In particular, given a program and a security policy in DLM fashion, we compute the set of readers and writers for each pair of principals p, q, as in [5], and check robustness for each pair by using Proposition 3. Hence, we have the following generalisation of relative robustness in DLM.

Proposition 4. *Let $P = P_2; [\bullet]; P_1$ be a program (where P_1 is without holes). Let $\Phi = Wlp(P_1, \Phi_0)$. P satisfies decentralized robustness wrt principals p, q if we have that $\left\{ x \,\middle|\, \mathcal{I}_{p\leftarrow q}(x) = \mathtt{L} \right\} \cap \mathcal{FV}(\Phi) = \varnothing$.*

Proof. Given a pair of principals (p, q) we compute the set of *readers* and *writers* as for decentralized robustness. Consequently, we have a static labeling of program data with respect to confidentiality and integrity. At this point we apply Lemma 1 as the hypothesis of proposition guarantees that no low integrity variable occurring free in Φ is used by the active attack. Since this holds for all possible pairs of principals, the claim is true.

This characterization suits perfectly to client-side web languages such as Javascript as it allows to prevent injection attacks or dynamically loaded third-party code. In particular, suppose we have a web page that accepts advertising adds from different sources, with different security concerns and wonder if it leaks private information to a malicious attacker. Moreover, we can assume that the web page has different trust relations with domains providing adds and this is specified in the security policy. Given this information, one can analyze the DOM (Document Object Model) tree and classify each attribute in sensitive and insensitive with respect to a possible attacker [17]. The *session cookie* might be an attribute to protect wrt all attackers, while the *history object* might be public to some trusted domains and private to others. At this point we can apply weakest precondition analysis to web server from the point it discloses information on any public channel such as the output web page or the reply information sent as response to a client request. The holes correspond to program points where the server receives adds from different clients and embeds them in its code. Analyzing the formula corresponding to the sensitive information disclosed before parsing and embedding such adds, namely using the *eval()* operation in Javascript, we can identify harmless low integrity variables and certify security modulo (relative to) programs manipulating this variables.

Example 12. Consider the following Javascript-like code (modified version of the example in [6]). Lines 3-6 correspond to an add received from a third party to be displayed on the web page. Moreover, the web site contains a simple function *login()* which authenticates users by verifying *username* and *password* inserted in a form. The function runs when the user clicks on a button, lines 7-16. Function *initSettings* corresponds to the output channel of the web page as it identifies the server used to authenticate the user, i.e., to send username and password.

```
1.  <script type="javascript">
          // 2: initialization of the output server
2.  initSettings("mysite.com/login.php", 1.0);
          // 3-4-5: definition of the add
3.  <div id="AdNode">
4.    <script src="adserver.com/display.js">
5.  </div>
6.  eval(src)

7.  var login = function() {
8.    var pwd = document.nodes.PasswordTextBox.value;
9.    var user = document.nodes.UsernameTextBox.value;
11.   var params = "u=" + user + "&p=" + pwd;
          //12: sends the parameters (params) to baseUrl
12.   post(document.settings.baseUrl, params);}
14. </script>
         //15-16:login interface
15. <text id="UsernameTextBox"> <text id="PasswordTextBox">
16. <button id="ButtonLogin" onclick="login()">
```

Now, suppose the add code corresponds to the hole and the public output is the final web page together with the result (*out* : LL) of *post* in line 12. Since formal parameters of function *initSettings* (defining variable *baseUrl* : LL) have low integrity, a malicious add could overwrite the parameters and redirect the high confidentiality part of the output of *post* (login and password, i.e., *user, pwd:* HH) to the attacker. Let us see how our approach allows to identify such security flaws. First, we compute the weakest precondition of function login and obtain the following formula:

$$
\begin{gathered}
[\bullet] \\
\{baseUrl + user + pwd = a\} \\
var\ pwd = document.nodes.PasswordTextBox.value; \\
var\ user = document.nodes.UsernameTextBox.value; \\
var\ params = "u = " + user + "\&p = " + pwd; \\
\{baseUrl + params = a\} \\
post(document.settings.baseUrl, params) \\
\{out = a\}
\end{gathered}
$$

Observing the final formula we can state that private information concerning username and password is related to the low integrity variable *baseUrl* and therefore the program does not satisfy confidentiality. Moreover, the program is not even robust since low integrity variable *baseUrl* is free before the "hole". In particular, a malicious add could

hijack such information to a malicious website and obtain username and password. However, we can deem this program robust relative to fair attacks which do not manipulate the low integrity variable *baseUrl*. In decentralized robustness, this corresponds to say that the program is robust wrt all the pairs (p, q) such that p does not believes that q can write the low integrity variable *baseUrl*.

7 Applications

In this section we present two applications where our approach captures soundly the possible security violations. The first example considers a secure API function widely used to perform PIN checking in a bank and is retrieved from [4]. The attacker is able to play with low integrity variables and reveal the real PIN by analyzing the implicit flow released by the API. The second example concerns a web application where third party code is allowed to be embedded in. Cross Site Scripting attacks (XSS) are name of the game in such contexts. In particular [23], the attacker tries to steal a session cookie and hijack the user to an evil website. In both examples our analysis is sufficient to capture the possible security violations.

7.1 Secure API Attack

This example concerns the use of secure API to authenticate and authorize a user to access an ATM cash machine. The user inserts the credit card and the PIN code at the machine. The PIN code gets encrypted and travels along the network until it reaches the issuing bank. At this point, a verifying API is executed in order to check the equality of the real user PIN and the trial PIN inserted at the cash machine. The verifying API, called PIN_V, is the one exploited by the attacker to disclose the real PIN.

The real PIN is derived through the PIN derivation key *pdk* and public data *offset, vdata, dectab*, while the trial PIN comes encrypted by key k. Of course, the two keys, pdk and k are pre-loaded in the Hardware Security Modules (HSM) of the bank server and never travel the network. Here is the description of the API, PIN_V.

```
PIN_V(EPB, len, offset, vdata,dectab) {
   x1 := enc_pdk(vdata);
   x2 := left(len, x1);
   x3 := decimalize(dectab, x2);
   x4 := sum_mod10(x3, offset);
   x5 := dec_k(len, EPB);
   if(x4 == x5) then return ("PIN correct");
                else return ("PIN wrong");
}
```

where:

- *len* is the length of real PIN obtained by the encryption of the validation data *vdata* (a kind of user profile) with the PIN derivation key *pdk* (x1), taking the *len* hexadecimal digits (x2), decimalising through *dectab* (x3), and digit-wise summing modulo 10 the offset (x4).

– EPB (Encrypted PIN Block) is the ciphertext containing the trial password encrypted with the key k. The trial PIN is recovered by decrypting EPB with key k.

The above snippet of code is insecure and there is a very nice attack used to disclose the exact PIN code just by modifying low integrity variables *offset* and *dectab* (of type LL) and observing low confidentiality output, namely by observing the I/O behavior of API method [4].

Example 13. Let $len = 4$, *offset* $= 4732$, $x1 = A47295FDE32A48B1$ and $dectab = 9753108642543210$ which is a substitution function encoding the mapping $0 \rightarrow 9, 1 \rightarrow 7, \cdots, F \rightarrow 0$. Moreover, let $EPB = enc_k(9897)$, where 9897 is the correct PIN. With these parameters PIN_V returns *PIN correct*.

Indeed, consider *x2* $=$ *left*$(4, A47295FDE32A48B1) = A472$, and consider *x3* $=$ *decimalize*(*dectab*, $A472$) $= 5165$ and *x4* $=$ *sum_mod10*$(5165, 4732) = 9897$ which is the same as the trial PIN.

Now the attacker first chooses *dectab1* $= 975311\underline{1}8642543211\underline{1}$ where the two 0's have been replaced by 1's. In this way the intruder discovers whether or not 0 appears in x3. Invoking the API with *dectab1* we obtain the same intermediate and final values, as $decimalize(dectab1, A472) = decimalize(dectab, A472) = 5165$. This means that 0 does not appear in x3.

The attacker proceeds by replacing the 1 of *dectab* by 2.

If *dectab2* $= 9753208642543220$ he obtains that *decimalize*(*dectab2*, $A472$) $= 5265 \neq$ *decimalize*(*dectab*, $A472$) $= 5165$, reflecting the presence of in the original value of x3. Then, $x4 = sum\_mod10(5265, 4732) = 9997$ instead of 9897 returning *PIN wrong*.

Now, the attacker knows that digit 1, occurs in x3 for sure. In order to discover its position and its multiplicity, he varies the *offset* so to *compensate* for the modification of *dectab*. In particular, the attacker decrements each *offset* digit by 1 until it finds the one that makes the API return *PIN correct*. For this particular instance the possible variations of the *offset* are: $\underline{3}732, 4\underline{6}32, 472\underline{2}, 473\underline{1}$ and the one that succeeds is the *offset* 4632. So, the attacker revealed that the second digit of $x3$ is 1. Given that the offset is public, he derives the second digit of user PIN as $1 + 7mod10$, where 7 is the second digit of the initial *offset*. Iterating this procedure the attacker discloses the entire value of PIN.

In the following computation we show weakest precondition approach captures the security flaws in API.

Let us observe the final formula corresponding to the weakest precondition of the API. Clearly, we can first note that the program does not satisfy confidentiality since the public output (the answer to the comparison between the real and the trial password) depends clearly on the high confidentiality variable containing the real password. From the viewpoint of robustness we can note that our sufficient condition is not satisfied since there are low integrity variables, i.e., *dectab* and *offset*, which are free before the hole (supposed to be in the input of the API, namely in the communication phase). Indeed, exactly those are the variables used by the attacker for disclosing the PIN.

$$\left\{\begin{array}{c}(sum\_mod10(decimalize(dectab, left(len, enc_p dk(vdata))), offset) = dec_k(len, EPB) \\ \wedge a = 1) \vee \\ (sum\_mod10(decimalize(dectab, left(len, enc_p dk(vdata))), offset) \neq dec_k(len, EPB) \\ \wedge a = 0)\end{array}\right\}$$

$$x1 := enc_p dk(vdata);$$

$$\left\{\begin{array}{c}(sum_m od10(decimalize(dectab, left(len, x1)), offset) = dec_k(len, EPB) \wedge a = 1) \vee \\ (sum_m od10(decimalize(dectab, left(len, x1)), offset) \neq dec_k(len, EPB) \wedge a = 0)\end{array}\right\}$$

$$x2 := left(len, x1);$$

$$\left\{\begin{array}{c}(sum\_mod10(decimalize(dectab, x2), offset) = dec_k(len, EPB) \wedge a = 1) \vee \\ (sum\_mod10(decimalize(dectab, x2), offset) \neq dec_k(len, EPB) \wedge a = 0)\end{array}\right\}$$

$$x3 := decimalize(dectab, x2);$$

$$\left\{\begin{array}{c}(sum\_mod10(x3, offset) = dec_k(len, EPB) \wedge a = 1) \vee \\ (sum\_mod10(x3, offset) \neq dec_k(len, EPB) \wedge a = 0)\end{array}\right\}$$

$$x4 := sum\_mod10(x3, offset);$$

$$\{(x_4 = dec_k(len, EPB) \wedge a = 1) \vee (x_4 \neq dec_k(len, EPB) \wedge a = 0)\}$$

$$x5 := dec_k(len, EPB);$$

$$\{(x_4 = x_5 \wedge a = 1) \vee (x_4 \neq x_5 \wedge a = 0)\}$$

$$if\ (x_4 == x_5)\ then\ (return\ 1)\ else\ (return\ 0)$$

$$\{l = a\}$$

The authors [4] fix this problem by using a MAC (Message Authentication Code) security primitive. In particular, MACs are used to guarantee the integrity of information received from an untrusted source, namely any modification of data before calling the API is prevented by MAC. Semantically, this means that the variables *dectab* and *offset* can be modified only by authorised agents. In our approach, this can be modelled by assigning the security level `LH` to *dectab* and *offset*, i.e., by considering them as high integrity. In this way, we are done, because our weakest precondition approach yields a formula containing free only high integrity variables. Hence the robustness condition is satisfied.

7.2 Cross Site Scripting Attack

Javascript is a very flexible dynamic object-based scripting language running in almost all modern web browsers. The language allows to transfer, parse and run code sent over the network between different web-based applications. While very useful and user-friendly, such flexibility comes at a great price as the underlying applications become vulnerable to code injection attacks. These attacks circumvent the security enforcement mechanism of Javascript, namely *the same-origin* policy which prevents a document or script loaded from one origin from getting or setting properties of a document from another origin [17]. Indeed, when the browser receives a compromised web page, it is executed in the context of the website hosting it, therefore, the same-origin policy deems the operation secure. Afterwards, the malicious code can establish a connection to the attacker server and transfer sensitive information such as cookie sessions for instance. The following example shows that language-based security techniques can be used to prevent this kind of attacks.

Suppose a user visits a untrusted web site in order to download a picture, where an attacker has inserted his own malicious Javascript code (Fig. 1), and execute it on the clients browser [23].

In the following we described a simplified version. The Javascript code snippet in Fig. 1 can be used by the attacker to send users cookie[3] to a web server under the attackers control.

```
var cookie = document.cookie;
         /*initialisation of the cookie by the server*/
var dut;
if (dut == undefined) {dut = "";}
while(i<cookie.length) {
    switch(cookie[i]) {
      case 'a': dut += 'a'; break;
      case 'b': dut += 'b'; break;
      ...
  }
}
        /* dut contains now copy of cookie*/
document.images[0].src =  "http://badsite/cookie?" + dut;
        /* when the user click on the image dut is sent
        to the web server under the attackers control*/
```

Fig. 1. Code creating a XSS vulnerability

One can easily see that the variable *dut* contains a copy of users cookie. This attack circumvents same-origin policy in client browser as it is correctly received after a request to some server where the attacker injected the malicious code. Now lets apply our analysis to the above Javascript snippet. In particular, suppose that variable *cookie* has security type HL and variable *dut* has security type LL. Moreover, imagine we emulate the switch-case operator by a chain of if-then-else constructs and *cookie.length* has security type LL .

$$
\begin{array}{c}
[\bullet] \\
\{cookie + dut = res\} \\
while(i < cookie.length)\{ \\
switch(cookie[i])\{ \\
case' a' : dut+ =' a'; break; \\
case' b' : dut+ =' b'; break; \\
...\}\} \\
\{dut = res\}
\end{array}
$$

[3] A cookie is a text string stored by a user's web browser. A cookie consists of one or more name-value pairs containing bits of information, sent as an HTTP header by a web server to a web browser (client) and then sent back unchanged by the browser each time it accesses that server. It can be used, for example, for authentication.

By observing the final formula we can notice that confidentiality is violated since there is a (implicit) flow of information from private variable *cookie* towards the public variable *dut*. However, this is the sensitive information disclosed by a passive attacker when *dut* is initialised in the code to the empty string. Nevertheless, *dut* is free before the hole, i.e., where the attacker can insert other malicious code, therefore the (active) attacker can exploit *dut* for disclosing other user confidential information. Suppose, for instance, the attacker to be interested in the *history* object (with security type `HL`) together with its attributes[4]. In this case, an active attack could loop over the elements of the *history* object and pass through variable *dut* all the web pages the client has had access to. Consider for example the injection of the code in Fig. 2.

```
<script language="JavaScript">
var dut = "";
for (i=0; i<history.length; i++){
    dut = dut + history.previous;
}
</script>
```

Fig. 2. Malicious code exploiting XSS vulnerability

Hence, in this case the program violates the robustness condition since the attacker can exploit the low integrity variable *dut*, which occurs free in the formula before the hole, in order to disclose more confidential information. Moreover we have shown that the attacker can exploit this vulnerability by inserting the code in Fig. 2 just before the malicious code (Fig. 1) in the untrusted web page, getting both *history* and *cookie* through the variable *dut*.

It is worth noting that our approach provides a theoretical model for the existing techniques used in practice for protecting code from XSS attacks [23].

8 Related Work

Prior work on robustness, in the language-based setting, has been addressed in [26,20]. In these papers the authors give a trace-based definition of robustness and enforce it with a flow-insensitive type system. They consider a simple while language, as we do in the this paper, but, in addition they consider an additional construct for declassifying the security of variables in fixed program points (the *where* dimension in [22]). Therefore, a program is robust is an active attacker is unable to manipulate program semantics and declassify more information than a passive attacker does. The security type system enforces both non-interference and robustness so a program is ruled out if neither of the two security properties holds. On the other hand, our semantic approach is different as we model global declassification policies (the *what* dimension in [22]). Moreover, we capture a cleaner characterization of robustness, namely the active attacker does

[4] The history object allows to navigate through the history of websites that a browser has visited.

not disclose more private information than a passive one, even though the program under passive attacker does not satisfy non-interference. Other differences between two approaches are shown in section 5.4.

The idea of considering the weakest liberal precondition semantics for static certification of program security is borrowed from [18]. The authors define declassified non interference as a completeness problem in abstract interpretation and the semantic function corresponds to the *Wlp* semantics. However this paper considers only passive attackers and moreover the idea of computing *Wlp* wrt first order formulas is novel in our approach.

Decentralized robustness [5] expresses robustness in the context of the decentralized label model and enforces it statically by a type system. In this paper we showed that the approach can be characterized by our notion of relative robustness. Section 6.1 compares the two approaches.

Language-based techniques for security are more and more being applied to client-side web languages such as Javascript to prevent different attacks [6,23]. Basically, they combine static and dynamic analysis to enforce information flow properties such as non interference. However, our idea of interpreting robustness for Javascript, to the best of our knowledge, is novel and could nicely fit in as a good security model for such language. In particular, the security type `HL` can model the code injected by an attacker, which knows a certain variable exists (password for instance), but doesn't know its value.

9 Conclusions

In this paper, we addressed an important notion in language-based security called robustness [26,20]. In general a program can run in any distributed environment in presence of untrusted components. This fact is modeled by fixed program points called *holes*, namely program points where the attacker can insert untrusted code. At this point, the program is robust if an active attacker cannot disclose more private information than a passive one. We noted that an active attacker can transform program semantics and control private information released by the program. Moreover, different active attacks can release different properties of private data. Hence, the total number of attacks may be infinite so it is impossible to find the most harmful attack for a given program. Here we characterised a sufficient condition that enforces robustness for unfair attacks (using `LL` and `HL` variables). Moreover, we have considered robustness in two different semantic models, I/O and trace semantics. Then we introduced the notion of *relative robustness* which is a relaxation of robustness dealing with a restricted class of attacks. Finally, we conclude with two real application: the analysis of the API for PIN verification and the analysis of code vulnerable to XSS attacks.

The analysis we performed in this paper results very interesting from both theoretical and practical point of view. On the one hand the semantic condition of robustness addresses the issue of systematic transformations of program code that preserve interesting extensional properties, robustness for instance. Indeed abstract interpretation is a possible framework to play with in order to guarantee such properties. On the other hand, we saw that our approach is a good remedy to the lack of precise static analysis approaches in real application domains concerning security.

However, this is just the beginning and there is much more work to do. First, we need to implement the algorithm for static certification of robust programs. Hence, given a program we need to effectively compute when it happens to be robust. It would be important to characterize classes of attacks that induce the same semantic transformation, namely disclose the same property of private inputs. In this way, we can hope for finding a finite number of such attack classes. Second, our work can be generalised to deal with abstract active attackers. Namely, as it happens for abstract non-interference, one can consider attackers modifying *properties* of low integrity data. Third, we plan to extend our approach to different attacker models such as concurrent attackers or attackers able to erase parts of program code. Off we go.

References

1. `http://www.owasp.org`
2. Balliu, M., Mastroeni, I.: A weakest precondition approach to active attacks analysis. In: PLAS, pp. 59–71 (2009)
3. Banerjee, A., Giacobazzi, R., Mastroeni, I.: What you lose is what you leak: Information leakage in declassification policies. In: Proc. of the 23th Internat. Symp. on Mathematical Foundations of Programming Semantics MFPS 2007. Electronic Notes in Theoretical Computer Science, vol. 1514. Elsevier, Amsterdam (2007)
4. Centenaro, M., Focardi, R., Luccio, F.L., Steel, G.: Type-based analysis of pin processing apis. In: Backes, M., Ning, P. (eds.) ESORICS 2009. LNCS, vol. 5789, pp. 53–68. Springer, Heidelberg (2009)
5. Chong, S., Myers, A.C.: Decentralized robustness. In: Proc. the IEEE Computer Security Foundations Workshop (CSFW-19), Washington, DC, USA, pp. 242–256. IEEE Computer Society, Los Alamitos (2006)
6. Chugh, R., Meister, J.A., Jhala, R., Lerner, S.: Staged information flow for javascript. In: PLDI, pp. 50–62 (2009)
7. Cohen, E.S.: Information transmission in sequential programs. In: DeMillo, et al. (eds.) Foundations of Secure Computation, pp. 297–335. Academic Press, New York (1978)
8. Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Proc. of Conf. Record of the 4th ACM Symp. on Principles of Programming Languages POPL 1977, pp. 238–252. ACM Press, New York (1977)
9. Cousot, P., Cousot, R.: Systematic design of program analysis frameworks. In: Proc. of Conf. Record of the 6th ACM Symp. on Principles of Programming Languages POPL 1979, pp. 269–282. ACM Press, New York (1979)
10. Danicic, S., Harman, M., Hierons, R., Howroyd, J., Laurence, M.: Applications of linear program schematology in dependence analysis. In: PLID (2004)
11. Dijkstra, E.W.: A discipline of programming. Series in automatic computation. Prentice Hall, Englewood Cliffs (1976)
12. Dijkstra, E.W.: Guarded commands, nondeterminism and formal derivation of programs. Comm. of The ACM 18(8), 453–457 (1975)
13. Goguen, J.A., Meseguer, J.: Security policies and security models. In: Proc. IEEE Symp. on Security and Privacy, pp. 11–20. IEEE Comp. Soc. Press, Los Alamitos (1982)
14. Gries, D.: The Science of Programming. Springer, Heidelberg (1981)
15. Hehner, E.C.R.: The Logic of Programming. In: Hoare, C.A.R. (ed.) Series in Computer Science. Prentice Hall, Englewood Cliffs (1984)

16. Rustan, K., Leino, M.: Efficient weakest preconditions. Inf. Process. Lett. 93(6), 281–288 (2005)
17. Ingo Lutkebohle. Same origin policy for javascript
18. Mastroeni, I., Banerjee, A.: Modelling declassification policies using abstract domain completeness. Technical Report RR 61/2008, Department of Computer Science, University of Verona (May 2008)
19. Myers, A.C., Liskov, B.: Protecting privacy using the decentralized label model. ACM Trans. Softw. Eng. Methodol. 9(4), 410–442 (2000)
20. Myers, A.C., Sabelfeld, A., Zdancewic, S.: Enforcing robust declassification. In: Proc. IEEE Symp. on Security and Privacy, pp. 21–34. IEEE Comp. Soc. Press, Los Alamitos (2004)
21. Sabelfeld, A., Myers, A.C.: Language-based information-flow security. IEEE J. on Selected Areas in Communications 21(1), 5–19 (2003)
22. Sabelfeld, A., Sands, D.: Declassification: Dimensions and principles. J. of Computer Security (2007)
23. Vogt, P., Nentwich, F., Jovanovic, N., Kirda, E., Krügel, C., Vigna, G.: Cross site scripting prevention with dynamic data tainting and static analysis. In: NDSS (2007)
24. Winskel, G.: The formal semantics of programming languages: an introduction. MIT Press, Cambridge (1993)
25. Yao, A.C.-C.: Protocols for secure computations (extended abstract). In: FOCS, pp. 160–164 (1982)
26. Zdancewic, S., Myers, A.C.: Robust declassification. In: Proc. of the IEEE Computer Security Foundations Workshop, pp. 15–23. IEEE Comp. Soc. Press, Los Alamitos (2001)
27. Zdancewic, S., Zheng, L., Nystrom, N., Myers, A.C.: Untrusted hosts and confidentiality: Secure program partitioning. In: SOSP, pp. 1–14 (2001)