A weakest precondition approach to active attacks analysis

Musard Balliu
School of Computer Science and Communication
Royal Institute of Technology (KTH)
Stockholm, Sweden
musard.balliu@gmail.com

Isabella Mastroeni
Dipartimento di Informatica
Università di Verona
Strada Le Grazie 15, I-37134 Verona, Italy
isabella.mastroeni@univr.it

Abstract

Information flow controls can be used 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 language-based security. The notion of robustness asserts that an active attacker, who can modify program code in some fixed points (*holes*), is not able to disclose more private information than a passive attacker, who merely observes public 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 the weakest precondition semantics. In particular, this semantics simulates the kind of analysis that can be performed by an attacker, i.e., from the public output towards the private input. The choice of the semantics lets us distinguish between different attacks models. In this paper, we also introduce relative robustness that is a relaxed notion of robustness for restricted classes of attacks.

Categories and Subject Descriptors D.3.1 [*Programming Languages*]: Formal Definitions and TheorySemantics; F.3.2 [*Logics and Meanings of Programs*]: Semantics of Programming Languages—Program Analysis; K.6.5 [*Management of Computing and Information Systems*]: Security and Protection

General Terms Languages, Security, Theory, Verification

Keywords Program semantics, Non-interference, Robustness, Declassification, Active attackers

1. Introduction

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 [Goguen and Meseguer 1982], stating that private information has not to affect the observable public data. 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 behaviour of programs and that, from these observations, can make some kind of *reverse engineering* in order to derive private information from the observation of the public data.

The idea from which we start 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 [Banerjee et al. 2007] for declassified non-interference, where declassification is modelled by means of abstract domains [Cousot and Cousot 1977]. The ingredients of this method are: the initial declassification policy modelled as an abstraction of the private input domain and the weakest liberal precondition semantics of programs [Dijkstra 1975, 1976], characterising the backward analysis (i.e., from outputs to inputs) simulating 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 characterises 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 the private inputs inside this collection, and this 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 observa-

PLAS '09 June 15, Dublin, Ireland.

tion, 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 [Banerjee et al. 2007].

Our aim is to use these ideas also in presence of *active attackers*, namely attackers that can both observe and modify the program semantics. We consider the model of active attackers proposed in [Myers et al. 2004] 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 the 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 with the information disclosed, is *robustness* [Zdancewic and Myers 2001]. 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 I/O 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 [Banerjee et al. 2007] 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 attacker, this means that it becomes unfeasible when dealing with an infinite number of possible active attackers. In order to overcome this problem, we need an analysis independent of the code of the particular active attacker. For this reason, we exploit the weakest precondition computation in order to provide a *sufficient condition* that guarantees robustness independently from 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 behaviour, and afterwards we extend 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 with respect to we are certifying robustness.

Roadmap. The rest of the paper is organized as follows. In Sect. 2 we present a general overview of non-interference and robust declassification. In Sect. 3 we introduce abstract interpretation and recall a technique for certifying declassification by means of weakest liberal precondition semantics. In Sect. 4 we compute (qualitatively) the maximal private information disclosed by active attackers. In particular, Sect. 4.1 introduces the problem of computing the maximal release by active attackers for I/O semantics. Sect. 4.2 extends the analysis for attacks observing traces. In Sect. 5 we discuss sufficient conditions to enforce robustness. Sect. 5.1 presents the main contribution of the paper, namely, a static analysis based on weakest preconditions to enforce robustness for I/O semantics; Sect. 5.2 compares our method with the type system-based one; Sect. 5.3 extends these results for trace semantics. Sect. 5.4 introduces relative robustness which deals with restricted classes of attacks. In Sect. 6 we interpret decentralized robustness in our approach. Sect. 7 concludes the paper and states new directions for future work.

2. Security Background

Information flow models of confidentiality, also called non-interference [Goguen and Meseguer 1982], are widely studied in literature [Sabelfeld and Myers 2003]. 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 ($\mathtt{H}$) and public ($\mathtt{L}$). In this paper we consider only terminating computations. Hence, there are basically two ways the program can release private information by observing the public outputs; first because of an explicit flow corresponding to a direct assignment of a private variable to a public variable and second because of an implicit flow corresponding to control structures of the program like the conditional **if** or the **while** loop [Sabelfeld and Myers 2003].

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 [Cohen 1978, Banerjee et al.

2007]. Consider a predicate ϕ on $\mathbb{V}^{\mathtt{H}}$, a program P satisfies declassified non-interference if

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

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 just 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., $\mathtt{LL}, \mathtt{LH}, \mathtt{HL}, \mathtt{HH}$. 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 [Myers et al. 2004].

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

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 supposed under control of the attacker. Hence, let $P[\vec{\bullet}]$ be 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 [Myers et al. 2004]:

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

where all variables in e and x have type LL. Note that fair attacks can observe and modify in any possible way only the variables that are both observable and modifiable.

An important notion when dealing with active attackers is *robustness* [Zdancewic and Myers 2001]. Informally, a program is said to be *robust* if every active attacker who actively controls the code in the holes, does not 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 [Myers et al. 2004]. Thus, we can formally define the notion of *robustness*, for terminating programs, in presence of active fair attackers [Myers et al. 2004].

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

Namely, a program is robust if any active (fair) attacker can disclose at most the same information that 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. Certifying Declassification

In this section, we introduce a technique recently proposed for certifying declassification policies [Banerjee et al. 2007, Mastroeni and Banerjee 2008] 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 the attacker can disclose from the given observation. We use abstract interpretation for modelling the declassified properties.

Abstract interpretation: an informal introduction. We use the standard framework of abstract interpretation [Cousot and Cousot 1977, 1979] for modelling properties. For example, instead of computing on integers we might compute on more abstract properties, such as the sign $\{+, -, 0\}$ or parity $\{\mathsf{even}, \mathsf{odd}\}$. Consider the program $\mathsf{sum}(x, y) = x + y$, then it is abstractly interpreted as sum^*: $\mathsf{sum}^*(+, +) = +$, $\mathsf{sum}^*(-, -) = -$, but $\mathsf{sum}^*(+, -) =$ *"I don't know"* since we are not able to determine the sign of the sum of a negative number with a positive one (modelled by the fact that the result can be any value). Analogously, $\mathsf{sum}^*(\mathsf{even}, \mathsf{even}) = \mathsf{even}$, $\mathsf{sum}^*(\mathsf{odd}, \mathsf{odd}) = \mathsf{even}$ and $\mathsf{sum}^*(\mathsf{even}, \mathsf{odd}) = \mathsf{odd}$. 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*$(S) = \mathsf{even} = \{\ n \in \mathbb{Z} \mid\ n$ is even $\}$ if $\forall n \in S.\ n$ is even, *Par*$(S) = \mathsf{odd} = \{\ n \in \mathbb{Z} \mid\ n$ 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 $\langle uco(C), \sqsubseteq, \sqcup, \sqcap, \lambda x.\top, \lambda x.x\rangle$, 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.

Certifying declassification. In [Banerjee et al. 2007, Mastroeni and Banerjee 2008] 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 [Winskel 1993]. 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 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 (*Wlp* for short) of programs. Starting from a given observed output *Wlp* semantics computes, by definition, the *greatest* set of inputs 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 $\vec{l} = \vec{a}$ the parametric value of each low confidentiality program variable. For instance, the formula $(l = a)$ means that the program variable l has the symbolic value a. 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.

$$\Phi_0 \stackrel{\text{def}}{=} \{l_1 = a_1 \wedge l_2 = a_2 \wedge \cdots \wedge l_n = a_n\} = \bigwedge_{i=1}^{n} (l_i = a_i)$$

where $\forall l_i.\mathcal{C}(l_i) = \mathtt{L}$. The *Wlp* semantic rules transform such formula for every construct of the while language. In particular, the assignment induces modification of program variables, the **if** and **while** constructs induce state partition represented by the logical connective *or* ($\vee$). Finally, the **skip** command leaves the formula unchanged while the sequential composition models the composition of *Wlp* semantics. 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, being Φ without quantifiers. 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 $l : \mathtt{LL}$.

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

$Wlp(P, l = a) = (h_1 = h_2 \wedge a = 0) \vee (h_1 \neq h_2 \wedge a = 1)$
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 [Banerjee et al. 2007, Mastroeni and Banerjee 2008] this technique is formally justified by considering an abstract domain completeness-based [Cousot and Cousot 1977] 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 $\vec{l} = \vec{a}$, 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 $\vec{a}$. This corresponds to the abstraction $\mathcal{H} \in uco(\wp(\mathbb{V}))$ [Banerjee et al. 2007] modelling the fact that the attacker cannot observe private data. Formally, it associates with a generic output state $\langle h, l\rangle$ the abstract state $\langle \mathbb{V}^{\mathtt{H}}, l\rangle = \{\ \langle h', l\rangle \mid\ h' \in \mathbb{V}^{\mathtt{H}}\ \}$. 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 modelled 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 modelled 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, modelled 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.

4. Maximal release by active attackers

The notion of robustness defined in Sect. 2 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, than 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. 2, i.e., fair attacks. In particular, our attackers can also 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.

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, afterwords, 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 4.1. *Consider the program* $P ::= l := h; [\bullet]$; *with variables* h : HH, l : LL *and* k : HL. *We can have the following attacks:*

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

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 (*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 [Banerjee et al. 2007] and verify that the method described in Sect. 3 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

[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.

the idea proposed in [Banerjee et al. 2007], where, in order to avoid an analysis for each possible output observation, the authors compute the analysis parametrically on the symbolic output observation $l = a$. 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 introduce a *Wlp* computation parametric even on the possible expressions $f(\vec{l})$ assigned by the active attacker to low integrity variables $\vec{l}$. 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 4.2. *Consider the program in Ex. 4.1. The only low integrity variables are* l : LL *and* k : HL. *According to the method described above we have to substitute the possible unfair attacks in* $[\bullet]$ *with* $\langle l,k\rangle := \langle f(l,k), g(l,k)\rangle$. *The initial formula is* $\Phi_0 = \{l = a\}$ *because* l *is the only program variable s.t.* $\mathcal{C}(l) =$ L. *Thus, the Wlp calculation yields the following formula:*

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

Note that the final formula $(f(h,k) = a)$ *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.*

4.2 Observing program traces

In [Mastroeni and Banerjee 2008] the authors notice that the semantic model constitutes an important dimension for program security, the *where* dimension [Sabelfeld and Sands 2007], 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 the 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 ($\mathbb{O}$) and we can denote by $\mathbb{H}$ the set of program points where there is a hole, namely where the attacker can insert his code. In this case, we assume that the attacker can observe the low integrity 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 [Mastroeni and Banerjee 2008]. 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 4.3. *Consider the program* P *with variables* l_1, l_2 : LL *and* h_1, h_2 : HH.

$$P ::= \left[\begin{array}{l} h_1 := h_2; h_2 := h_2 \bmod 2;\\ l_1 := h_2; h_2 := h_1; l_2 := h_2; l_2 := l_1; \end{array}\right.$$

We want to compute the private information disclosed by an attacker that observes program traces. As for standard noninterference, 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.*

$$\begin{gathered}\{h_2 \bmod 2 = a \wedge h_2 = b \wedge l_2 = c \wedge l_1 = d\}\\ h_1 := h_2;\\ \{h_2 \bmod 2 = a \wedge h_1 = b \wedge l_2 = c \wedge l_1 = d\}\\ h_2 := h_2 \bmod 2;\\ \{h_2 = a \wedge h_1 = b \wedge l_2 = c \wedge [l_1 = d]\}\\ l_1 := h_2;\\ \{l_1 = a \wedge h_1 = b \wedge l_2 = c\}\\ h_2 := h_1;\\ \{l_1 = a \wedge h_2 = b \wedge [l_2 = c]\}\\ l_2 := h_2;\\ \{l_1 = a \wedge [l_2 = b]\}\\ l_2 := l_1;\\ \{l_1 = l_2 = a\}\end{gathered}$$

For instance the information observed by the assignment $l_2 := l_1$ *is the combination of Wlp calculation* $(l_1 = a)$ *and attackers observation at that point* $([l_2 = b])$. *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 observing input-output, who can only distinguish the parity of variable* h_2.

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 4.4. *Let $P[\vec{\bullet}]$ be a program vulnerable to fair attacks and $\mathbb{H} \subseteq \mathbb{O}$. Then, all fair attacks can be written as $\vec{l} := \vec{a}$, 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 $\vec{l} := \vec{a}$. □

Now we are able to measure the maximal private information disclosed by an active attacker. Indeed, we can use the approach of Ex. 4.3 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 4.5. *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 doesn't 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 inserts fair attacks, discloses the following private information:

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

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 inserts 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 guaranteeing program robustness. Here we introduce a method to enforce programs which are robust in presence of active attackers.

We know [Banerjee et al. 2007] that declassified noninterference 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 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 gain more confidential information wrt passive attackers.

EXAMPLE 5.1. *Consider the program P with h : HH, l : LL.*

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

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

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

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

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

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 effectively may be 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 the 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*

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 5.2. *Consider the program P with variables l : LL, k : LL, h : HH and declassification policy that releases nothing about private variables.*

$$P ::= \left[\begin{array}{ll} k := h; [\bullet]; & \\ \textbf{if } (l = 0) & \textbf{then } (l := 0; k := 0) \\ & \textbf{else } (l := 1; k := 1); \end{array}\right.$$

First notice that P does not release any 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 the constants c_1 and c_2, respectively, to variables l and k, the Wlp calculation deems the program robust.

$$\{(c_1 = 0 \wedge a = 0 \wedge b = 0) \vee (c_1 \neq 0 \wedge a = 1 \wedge b = 1)\}$$
$$k := h;$$
$$[l := c_1; k := c_2;]$$
$$\{(l = 0 \wedge a = 0 \wedge b = 0) \vee (l \neq 0 \wedge a = 1 \wedge b = 1)\}$$
$$\textbf{if } (l = 0) \textbf{ then } (l := 0; k := 0) \textbf{ else } (l := 1; k := 1);$$
$$\{l = a \wedge k = b\}$$

The final formula shows that such program satisfies noninterference. But if we assign to low integrity variables an expression depending on other low integrity variables, then we obtain some 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.

$$\{(h = 0 \wedge a = 0 \wedge b = 0) \vee (h \neq 0 \wedge a = 1 \wedge b = 1)\}$$
$$k := h;$$
$$\{(k = 0 \wedge a = 0 \wedge b = 0) \vee (k \neq 0 \wedge a = 1 \wedge b = 1)\}$$
$$[l := k;]$$
$$\{(l = 0 \wedge a = 0 \wedge b = 0) \vee (l \neq 0 \wedge a = 1 \wedge b = 1)\}$$

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 concerning low integrity variables. If we assign to low integrity variables a constant value in the domain it means that we are erasing the high confidentiality information that such variable might have accumulated before reaching that point or we are not considering the possibility of assigning to such variable another one which contains some private information that may possibly be lost subsequently as shown in Ex. 5.2.

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 output, namely, $\Phi_0 = \bigwedge_{i=1}^{n}(l_i = a_i)$ and $\forall i.\mathcal{C}(l_i) = \mathtt{L}$. We apply recursively the *Wlp* semantic rules and at each step we suppose the formula is in the normal form described before. The following theorem states that a program with holes $P[\bullet]$ is robust when the post condition of each hole does not contain low integrity free variables. 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. Next lemma, shows the result for programs with only one hole, while the theorem extends the result to programs with an arbitrary number of holes.

LEMMA 5.3. *Let $P = P_2[\bullet]P_1$ be a program (P_1 without holes). 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 will prove that for any attack a, 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**: Trivial. The initial formula Φ doesn't change, namely *Wlp*(**skip**, Φ) = Φ, and the attacker acts as a passive one.
- a ::= l := e: By definition of the active attack we have $\mathcal{I}(l) = \texttt{L}$ and by hypothesis the variable l does not occur free in Φ. Finally we apply *Wlp* definition for the assignment, so *Wlp*$(l := e, \Phi) = \Phi[e/l] = \Phi$.
- a ::= $c_1; c_2$: Applying the inductive hypothesis we have *Wlp*(c_1, Φ) = *Wlp*(c_2, Φ) = Φ. The *Wlp* definition for sequential composition states that *Wlp*$(c_1; c_2, \Phi)$ = *Wlp*$(c_1,$ *Wlp*$(c_2, \Phi))$ = Φ
- a := **if** B **then** c_1 **else** c_2: By inductive hypothesis (applied to a construct of minor length) we have *Wlp*(c_1, Φ) = *Wlp*(c_2, Φ) = Φ. Applying the definition of *Wlp* for the conditional construct *Wlp*(**if** B **then** c_1 **else** c_2, Φ) = $(B \wedge$ *Wlp*$(c_1, \Phi)) \vee (\neg B \wedge$ *Wlp*$(c_2, \Phi))$ = $(B \wedge \Phi) \vee (\neg B \wedge \Phi)$ = Φ.
- a ::= **while** B **do** c: By hypothesis we consider terminating computations and the **while** loop halts in a finite number of iterations. Applying the inductive hypothesis to command c we have *Wlp*$(c, \Phi) = \Phi$, so every iteration the formula doesn't change. Moreover, if the guard is *false* the formula remains unchanged too. Applying the *Wlp* rule for the **while** loop and the inductive hypothesis we have:
  Wlp(**while** B **do** $c, \Phi) = (\neg B \wedge \Phi) \vee (B \wedge \Phi) \vee \cdots \vee (B \wedge \Phi) = \Phi$

□

THEOREM 5.4. *Let $P[\bullet]$ be a program. P is robust wrt unfair attacks if $\forall i \in \mathbb{H}. \forall v \in \mathcal{FV}(\Phi_i). \mathcal{I}(v) = \texttt{H}$.*

In other words, if a low integrity variable in not free in the formula, this means its information in that program point is useless for the analysis to reveal confidential properties. Hence, an active attacker is not stronger than a passive one. Let us introduce an example that illustrates Th. 5.4.

EXAMPLE 5.5. *Let us check robustness of program P with variables l : LL, h : HH and k : HL.*

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

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

$$\begin{array}{c} \{(h \leq 0 \wedge a = 1) \vee (h > 0 \wedge a = 0)\} \\ l := 1; k := h; \\ \{(h \leq 0 \wedge l = a) \vee (h > 0 \wedge a = 0)\} \\ \textbf{while } (h > 0) \textbf{ do } (l := l - 1; l := h); \\ \{l = a\} \end{array}$$

The formula $\Phi = (h \leq 0 \wedge a = 1) \vee (h > 0 \wedge a = 0)$ satisfies the conditions of Th. 5.4. We can conclude the program P is robust. Intuitively, even though the value of private input h flows to the public variable l ($l := l + h$), such relation is immediately canceled after the hole when we assign the constant 1 ($l := 1$).

The following example shows that Th. 5.4 is just a sufficient condition, namely there exists a robust program that violates the preconditions. This is because Th. 5.4 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 5.6. *Consider the program*

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

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

$$\begin{array}{c} \textit{Wlp}\,(\textbf{while } (h = 0) \textbf{ do } (h := 1; l := 0), \{l = a\}) = \\ \{(h = 0 \wedge a = 0) \vee (h \neq 0 \wedge l = a)\} \end{array}$$

This formula does not satisfy the conditions of Th. 5.4, since it contains a free occurrence of a low integrity variable, namely $l = a$. However, we can see that program P is robust. No modification of 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. 5.4 is a sufficient and not necessary condition for robustness.*

More generally, the construct $[\bullet]$ may be placed in an arbitrary depth inside an **if** conditional or a **while** loop. It is sufficient to consider the subformula corresponding to the branch where the hole is placed in and apply the Th. 5.4 to that particular part. The following example describes this situation.

EXAMPLE 5.7. *Consider the program P*

$$P ::= \left[\begin{array}{ll} k := h \textit{ mod } 3; & \\ \textbf{if } (h \textit{ mod } 2 = 0) & \textbf{then}[\bullet]; l := 0; k := l \\ & \textbf{else } l := 1; \end{array}\right.$$

where h : HH, l : LL and k : LL. Applying the weakest liberal precondition rules to the initial formula $(l = a \wedge k = b)$ we

have:

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

The subformula corresponding to the **then** *branch (which contains the hole* $[\bullet]$*) satisfies the conditions of Th.5.4, 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 any accumulated private information.*

5.2 *Wlp* vs the security type system

In [Myers et al. 2004] the authors define the notion of robustness in presence of active attackers and enforce it 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 the cases where the construct $[\bullet]$ is admissible.

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

By definition $L_C = \{l | \mathcal{C}(l) \sqsubseteq \mathcal{C}(A)\}$, namely L_C is the set of variables whose confidentiality level is not greater than the attackers confidentiality level. Hence, an active attacker that manipulates this variables does not obtain further confidential information.

Our approach, in particular Th. 5.4, captures exactly those situations where we have a low confidentiality context L_C and, nevertheless, the fair attack does not succeed, namely where there are no low integrity variables in the corresponding first order formula. 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. 5.4 provides only a sufficient condition and the type system is not complete. Anyway, we can say that our semantic-based method is more precise, in the sense that it generates less *false alarms*, than the type-based one. Namely, let us consider the program $P ::= [\bullet]; \textbf{if } h > 0 \textbf{ then } l := 0 \textbf{ else } l := 0$ where h : HH and l : 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 typecheck this program by using the rules in [Myers et al. 2004] we notice that the environment before hole is a high confidentiality one. Thus, this program is deemed not robust.

5.3 Robustness on program traces

In this section, we want to find local conditions guaranteeing robustness also in presence of active attackers observing the trace semantics instead of the I/O one. In other words, we want to characterise the analogous of Th. 5.4 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 all 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 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 capability to observe the hole program points allows us to reduce all the possible attacks to constant assignments to variables of type LL.

Using the method introduced in [Mastroeni and Banerjee 2008], and 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 it 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 doesn't 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. 5.4.

EXAMPLE 5.8. *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.

$$\{h_3 \bmod 2 = a \wedge h_3 = b \wedge l = c \wedge h_1 + h_2 = d\}$$
$$k := h_1 + h_2;$$
$$[skip;]$$
$$\{h_3 \bmod 2 = a \wedge h_3 = b \wedge l = c \wedge [k = d]\}$$
$$k := h_3 \bmod 2;$$
$$\{k = a \wedge h_3 = b \wedge [l = c]\}$$
$$l := h_3;$$
$$\{k = a \wedge [l = b]\}$$
$$l := k;$$
$$\{l = k = a\}$$

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'll notice that no fair attack (in our case manipulating* k*) can do better, because the subformula corresponding to the information disclosed by the attacker before reaching the*

hole 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.*

$$\{h_3 \bmod 2 = a \wedge h_3 = b \wedge l = c \wedge h_1 + h_2 = e\}$$
$$k := h_1 + h_2;$$
$$\{h_3 \bmod 2 = a \wedge h_3 = b \wedge l = c \wedge d = d_1 \wedge [k = e]\}$$
$$[k := d_1;]$$
$$\{h_3 \bmod 2 = a \wedge h_3 = b \wedge l = c \wedge [k = d]\}$$

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

Now we can introduce a sufficient condition for robustness in presence of fair attacks for trace semantics. Basically, it is a extension of Th. 5.4 to traces.

PROPOSITION 5.9. *Consider* $P[\vec{\bullet}]$ *with* $\mathbb{H} \subseteq \mathbb{O}$, *and* $\Phi_i = Wlp(P_i, \Phi_0)$ *(where* P_i *is obtained as in Th. 5.4).* P *is robust wrt fair attacks if* $\forall i \in \mathbb{H}.\forall v \in \mathcal{FV}(\Phi_i).\mathcal{I}(v) = \mathtt{H}$.

Note that, in this proposition we consider attackers that observe low confidentiality data at least in the points they can modify, i.e., $\mathbb{H} \subseteq \mathbb{O}$.

It is worth noting that, in Prop. 5.9 differently from Th. 5.4, we consider fair attacks and, indeed, the proposition is not true for unfair attacks. The main reason is that an active attacker can combine the information released by the *Wlp* semantics with the value of low confidentiality variables observed in that point. Thus, an unfair attacks, for instance, can assign a variable of type HL to a variable of type LL and hence such information is disclosed by the observation in that program point. The following example shows this situation.

EXAMPLE 5.10. *Consider the program*

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

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.*

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

This formula satisfies the conditions of Prop. 5.9: 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.

$$\left\{ \begin{array}{c} ((h = 0 \wedge a = 0) \vee (h \neq 0 \wedge a = 1)) \wedge \\ k = b \wedge k \bmod 2 = c \end{array} \right\}$$
$$l := k \bmod 2;$$
$$\{((h = 0 \wedge a = 0) \vee (h \neq 0 \wedge a = 1)) \wedge k = b \wedge [l = c]\}$$
$$[l := k;]$$
$$\{((h = 0 \wedge a = 0) \vee (h \neq 0 \wedge a = 1)) \wedge [l = b]\}$$

We can conclude that program P *is not robust (wrt unfair attacks) even though the conditions of Prop. 5.9 are satisfied.*

The problem, in the example above, 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. Hence, if we consider a model where the attacker can observe all program points except the ones where there is a hole, then Prop. 5.9 is true even for unfair attacks.

PROPOSITION 5.11. *Consider* $P[\vec{\bullet}]$ *with* $\mathbb{H} \cap \mathbb{O} = \varnothing$, *and* $\Phi_i = Wlp(P_i, \Phi_0)$ *(*P_i *obtained as in Th. 5.4).* P *is robust wrt* unfair *attacks if* $\forall i \in \mathbb{H}.\forall v \in \mathcal{FV}(\Phi_i).\mathcal{I}(v) = \mathtt{H}$.

This proposition tells us that we can ensure robustness in presence of unfair attacks also on the trace semantics only when the attackers cannot combine their capabilities of observing low confidentiality variables and of modifying low integrity variables, i.e., when $\mathbb{H} \cap \mathbb{O} = \varnothing$.

5.4 Relative Robustness

So far, we have characterised 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 a problem, 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 5.12. *Let* $P[\bullet]$ *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* $P[\overrightarrow{skip}]$.

In order to check relative robustness we can compute the confidential information released for all the 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 the 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 the attacks.
Consider Ex. 4.1. 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 the variable l and derive information exclusively about the 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. 5.4 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 *all* the low integrity variables. 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 $\mathcal{A}$.

PROPOSITION 5.13. *Let $P = P_2[\bullet]P_1$ be a program (P_1 without holes). Let $\Phi = Wlp(P_1, \Phi_0)$. P is relatively robust wrt the unfair attacks in $\mathcal{A}$ if $\forall a \in \mathcal{A}. Var(a) \cap \mathcal{FV}(\Phi) = \varnothing$.*

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. Relative vs Decentralized Robustness

Recently robustness has been considered also for mutual distrust and in particular in the decentralized label model (DLM) [Myers and Liskov 2000].
This model expresses information security policies in terms of principals, which do not trust each other and which can express and retain ownership over information security policies regarding confidentiality and integrity [Chong and Myers 2006]. In particular, the fact that each principal does not trust the others means that each principal may be a potential attacker. Hence, robustness is analysed 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 $\langle R_{p\to q}, W_{p\leftarrow q}\rangle$, where $R_{p\to q}$ 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 [Chong and Myers 2006].
At this point, 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 given attacker q from the point of view of p. We can denote by $\mathcal{C}_{p\to q}$ the confidentiality levels and by $\mathcal{I}_{p\leftarrow q}$ the integrity levels so far characterised. 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. Hence, we have the following generalisation of relative robustness in the DLM.

PROPOSITION 6.1. *Let $P = P_2[\bullet]P_1$ be a program (P_1 without holes). Let $\Phi = Wlp(P_1, \Phi_0)$. P satisfies decentralized robustness wrt the principals p, q if we have that $\left\{ x \,\middle|\, \mathcal{I}_{p\to q}(x) = \mathtt{L} \right\} \cap \mathcal{FV}(\Phi) = \varnothing$.*

7. Conclusions

In this paper, we studied an important notion in language-based security called robustness [Zdancewic and Myers 2001, Myers et al. 2004]. 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 noticed that an active attacker can transform program semantics and control the 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 characterise 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. Finally, we introduced the notion of *relative robustness* which is a relaxation of robustness dealing with a restricted class of attacks.

However, this is just the beginning and there is much more work to do. First, we need an algorithm for static certification of robust programs. Hence, given a program we need to compute the most harmful attacker able to disclose the maximal information release. 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. Finally we would like to deepen the relation between our approach and the decentralized model. In particular we aim to characterise the robustness wrt all the attackers by using our weakest precondition analysis.

Acknowledgements. This paper was partially supported by the PRIN projects “SOFT”.

References

A. Banerjee, R. Giacobazzi, and I. Mastroeni. What you lose is what you leak: Information leakage in declassifivation policies. In *Proc. of the 23th Internat. Symp. on Mathematical Foundations of Programming Semantics* (*MFPS '07*), volume 1514 of *Electronic Notes in Theoretical Computer Science*, Amsterdam, 2007. Elsevier.

S. Chong and A. C. Myers. Decentralized robustness. In *Proc. the IEEE Computer Security Foundations Workshop (CSFW-19)*, pages 242–256, Washington, DC, USA, 2006. IEEE Computer Society.

E. S. Cohen. Information transmission in sequential programs. In DeMillo et al., editor, *Foundations of Secure Computation*, pages 297–335, New York, 1978. Academic Press.

P. Cousot and R. Cousot. 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 '77*), pages 238–252, New York, 1977. ACM Press.

P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Proc. of Conf. Record of the 6th ACM Symp. on Principles of Programming Languages* (*POPL '79*), pages 269–282, New York, 1979. ACM Press.

E. W. Dijkstra. *A discipline of programming*. Series in automatic computation. Prentice-Hall, 1976.

E.W. Dijkstra. Guarded commands, nondeterminism and formal derivation of programs. *Comm. of The ACM*, 18(8):453–457, 1975.

J. A. Goguen and J. Meseguer. Security policies and security models. In *Proc. IEEE Symp. on Security and Privacy*, pages 11–20, Los Alamitos, Calif., 1982. IEEE Comp. Soc. Press.

I. Mastroeni and A. Banerjee. Modelling declassification policies using abstract domain completeness. Technical Report RR 61/2008, Department of Computer Science, University of Verona, May 2008.

A. C. Myers and B. Liskov. Protecting privacy using the decentralized label model. *ACM Trans. Softw. Eng. Methodol.*, 9(4): 410–442, 2000.

A.C. Myers, A. Sabelfeld, and S. Zdancewic. Enforcing robust declassification. In *Proc. IEEE Symp. on Security and Privacy*, pages 21–34, Los Alamitos, Calif., 2004. IEEE Comp. Soc. Press.

A. Sabelfeld and A.C. Myers. Language-based information-flow security. *IEEE J. on selected ares in communications*, 21(1): 5–19, 2003.

A. Sabelfeld and D. Sands. Declassification: Dimensions and principles. *J. of Computer Security*, 2007.

G. Winskel. *The formal semantics of programming languages: an introduction*. MIT press, Cambridge, Mass., 1993.

S. Zdancewic and A. C. Myers. Robust declassification. In *Proc. of the IEEE Computer Security Foundations Workshop*, pages 15–23, Los Alamitos, Calif., 2001. IEEE Comp. Soc. Press.