Defining an Information Flow Control Policy with Declassification and Countermeasures

Llorenç Garcia Cases

Télécom Bretagne

Département Informatique

Master Recherche en Informatique

Defining an Information Flow Control Policy with Declassification and Countermeasures

Author:
Llorenç Garcia Cases

Supervisor:
Julien Mallet

August 16, 2012

Abstract

Information systems manage different types of data, commonly classified between public and secret data. The origin of some programs are not known by the final user, having the risk of executing a program and leaking some of the secret data. Information flow control serves to track how are the variables accessed and used in order to know if a program may leak a secret.

Information flow policies define how information can flow, allowing no flow from secret to public variables in the case of the noninterference property or accepting certain flows in the case of information declassification. However, countermeasures are not considered at the level of defining a policy.

This work intends to give a framework to accept countermeasures during the definition of a policy, improving its expressiveness. The proposals are formally defined in order to allow a future systematization of the security analysis.

Keywords: Computer security, information flow, noninterference, declassification, countermeasures, security policies.

Contents

Acknowledgements

First of all I would like to thank my family, specially my parents and sister, for the way they educated me. It is the challenging way of life and the perseverance they taught me that helped me bringing to fruition this stage of my life.

I will always thank Julien Mallet for his initial vote of confidence to do this internship. I'm also thankful for the patience he had towards me, knowing the difficulties that appear when changing the language to work with. Thanks for your help, I have tried to keep learning every day from you.

I want to express thanks to my school, ETSETB, for supporting this experience. And also to Telecom Bretagne, specially to Antoine Beugnard and Fabien Dagnat for giving me the chance to be in the Master Recherche en Informatique.

In a more general way, I am thankful to all my friends (both from here and there) for making it easier for me to go through this adventure.

1 Introduction

Keeping information secure is a challenge humanity has dealt with since a long time ago. Historically, information has been accessed while it is being transmitted. With the spread of information systems, new ways appeared for accessing certain secrets, extracting the information directly from the system. Analyzing such access is known as information flow control.

Information systems manage different types of data. The basic distinction is between secret and public data. While a user name is information that can be published (public), spreading the value of a password (secret) is dangerous for the user. Information flow control takes this distinction into account while tracking how information is used during the execution of a program.

In several widely used contexts nowadays (web 2.0, JavaScript web applications, etc.) [JJLS10] the origin of the programs to execute are not known by the final user. Such programs may access private information and send it to a possible attacker. In this context, information flow control serves to know if a program may leak any secret or not. In order to define how information can flow, some security policies have been proposed.

Some properties ensure no information flow from secret to public data, a property known as noninterference [VSI96]. However, this property does not accept most applications that need to access some secret data in order to work properly. In trying to characterize real applications, some developments have been proposed starting from the property of noninterference, but they are still too restrictive for widespread use [Zda04].

Other proposals consider declassifying some private information, that is, explicitly defining which private information may be used [SS05]. This development allows a certain flow from private information, defined by a property, hence ensuring a certain secure information flow in actual systems.

The security properties define information flows that can be considered secure. However, if the program does not ensure the property, then it is not known which leaks it could generate. Therefore it is desirable to avoid or limit the leaks a program may generate. Security policies usually don't define what to do in the case of privacy-violating information flows, which may entail the leak of secret data. Such reactions are known as countermeasures.

There are different available countermeasures for reacting to these privacy-violating information flows. It should be possible to choose which countermeasures have to be applied. Defining the countermeasures at the policy level allows one to formalize them at a high abstraction level.

Existing security policies are not flexible enough for managing existing applications that are vastly used. Security properties are not expressive enough to characterize the accepted information flows in the existing programs. Additionally, no reactions are specified when a program causes certain leaks. Consequently, it is imperative to define more expressive security policies.

This work presents security policies as a formal combination of security properties and countermeasures. On one hand, security properties have to characterize in more detail how information is accessed, accepting more complex information flows than noninterference. On the other hand, countermeasures have to be properly defined in order to understand the possible reactions to information

leakage. The proposition is formally defined, allowing a systematization of the security analysis and eventually proving its correctness.

Section 2, below, presents the related work in the context of information flow control. Then, Section 3 proposes an extension for the semantics to track the different information flows, studying their properties. Section 4 presents the different countermeasures available in the present context. Then, Section 5 presents an expressive policy considering both declassification of information and countermeasures. Finally, Section 6 concludes this dissertation and presents some perspectives for the future.

2 Related Work

There are several works referring to the noninterference property, presented by J. A. Goguen and J. Meseguer [GM82]. It usually differentiates information between secret and public, but also accepts multiple levels [Den76]. While looking to use information in a less restrictive way [Zda04, Ros99], some works allow the release of certain secrets, thereby, declassifying this information. Such considerations define if a program is secure or not, but in the negative case it presents the need to define the possible reactions for a program to become secure.

Subsection 2.1, below, presents the noninterference property and some remarks about information flows that can cause leaks of private information. Then, Subsection 2.2 presents the different ways the private information can be accepted to then be published, or rather, the declassification dimensions. Subsection 2.3 presents the different countermeasures proposed in the context of network protocols and a review of the existing countermeasures implementations in the context of information flow control. Finally, Subsection 2.4 summarizes the ideas presented, leading to the proposition of a more expressive policy of the present work.

2.1 Noninterference Property

The noninterference property has been long used in the context of information flow control [GM82]. In short, noninterference states that secret data does not affect the public data.

The noninterference property is satisfied if the public data is not affected by the secret data after applying the semantics of a program. That is, a public observer cannot see any difference between all the executions with the same value of the public data but different values on the secret data. This property can be semantically defined as:

$$\forall s_1, s_2 \in S.(s_1 =_L s_2) \wedge (\llbracket C \rrbracket s_1 \Downarrow, \llbracket C \rrbracket s_2 \Downarrow) \implies \llbracket C \rrbracket s_1 =_L \llbracket C \rrbracket s_2 \qquad (1)$$

Where $\llbracket C \rrbracket$ is the semantics of a program that takes a memory state s to a resulting memory state $s_f = \llbracket C \rrbracket s$ or $\perp$ if it does not terminate. In a more detailed way, let us define the configuration $\langle c, s \rangle$ consisting on the command c and a memory s. It generates the transition $\langle c, s \rangle \longrightarrow \langle c', s' \rangle$, and finishes with $\langle c, s \rangle \longrightarrow^* s_f$, which can be expressed as $\langle c, s \rangle \Downarrow s_f$ or $\langle c, s \rangle \Downarrow$ if the resulting state is not important (but it does terminate).

The presented property means that given two initial states s_1 and s_2 equivalent at the low level (defined with $=_L$), the resulting states after the computation $\llbracket C \rrbracket s_i$ will be also equivalent in the low level (i.e., all the public variables). The state s is usually defined by a configuration of high and low variables, $s = (s_H, s_L)$ where *high* corresponds to the *secret* data and *low* to *public* data.

In order to statically check that a program ensures noninterference, Volpano *et al.* defined in [VSI96] a set of typing rules. If a program can be typed according to such rules, it then ensures the noninterference property.

For detecting whether a program ensures the noninterference property it is necessary to monitor how the information flows. Such flows can be classified

into two categories: *direct flow* and *indirect flow*. The language *While* [NN92] is considered in this work.

Direct flow appears when there is an assignment of a low variable affected by one or more high variables (see Figure 1a). *Indirect flow* appears when conditionals or loops contain high variables in the guard and any of the branches have assignments onto low variables (see Figure 1b). The distinction between *explicit* or *implicit* indirect flow is done if the assignments onto low variables take place or not respectively. In Figure 1b, the value of h can be learned by observing the value of l through different *indirect flows*. Two possibilities can be differentiated, if $h = true$ then the assignment of $l := 1$ is executed and hence the value of h flows through *explicit indirect flow*. On the other hand, if $h = false$, no assignment takes place and it can be learned that $h \neq true$ through an *implicit indirect flow*.

```
l := h;
```

(a) Direct Flow

```
l := 0;
if(h){ l := 1;}
else{skip;}
```

(b) Indirect Flow

Figure 1: Examples of programs leaking high information into low variables

If the attacker takes into consideration the time a program takes to be executed, then it is possible to learn private information through *timing leaks*. The work of J. Agat presents them in detail and presents also a possible transformation to avoid such timing leaks [Aga00]. Consequently, timing leaks are out of the scope of the present work.

Yet, forbidding all the high variables to flow into the low ones is, in some cases, too restrictive. For example, one program checking the password (a secret variable) does not ensure noninterference because the program behavior depends on the password, but it is widely used. Another example is online buying programs, which usually need to print some of the numbers of the user's bank account (a secret variable). In such cases it is necessary to allow some private information to flow into low variables, knowing that this release is not dangerous for the user. This information release is referred as declassification.

2.2 Information Declassification

There is a need to allow the release of private information for lots of programs. However, there are different works defining this information release in some specific way, depending on how the information is accepted to be released. In [SS05], A. Sabelfeld and D. Sands propose four different dimensions of declassification, taking into account the goals of each one: "*what* information is released, *who* releases information, *where* in the system information is released and *when* information can be released.". Below the main characteristics of each one are discussed.

What. This dimension states *what* information can be released, which is presented by Sabelfeld and Myers as *delimited release* [SM04]. In their work the information release has to be done through some *escape hatch* expressions (a declassify operator, namely $declassify(h)$). The policy defines which expressions are permitted to cause some high information to flow into a lower level.

Some examples of this dimension can be to show the last four numbers of a bank account or to transform the data (as with the RSA algorithm), preventing an attacker from learning the whole secret. Other possibilities are to accept publishing the average salary of an enterprise. Then, any observer can obtain the average salary without learning the individual salaries of each employee. Therefore, the only accepted release of the individual salaries sal_i is through $average(sal_1, sal_2, ..., sal_n) = (sal_1 + sal_2 + ... + sal_n)/n$. This way, the salaries of each employee become indistinguishable.

In the Partial Equivalent Relation model of information [SS98] it is shown that for certain transformations of the secret, the attacker is in fact unable to distinguish two different executions with certain different secret values; for example, downgrading only the parity of the data gives same values for two different executions, if the parity of the secret values agrees.

Who. It is necessary to specify *who* can release information [LM09]. If it is not specified, then an attacker can hijack the execution and release secret data. The first approach is to declare the owner of a secret and who else can modify it with a label in the code; a practical use of that can be found in the Jif Compiler [MZZ+04]. An ownership-based model relies on the certainty that an attacker cannot distinguish an information release in the memory; in this context the robust declassification is proposed [ZM01, MSZ04]. The robust declassification has the property that no more information than originally intended can be released by an attacker (if a particular secret is declassified, it is not possible that another secret is leaked).

Where. *Where* the information is released can be understood as where in the code the declassification is placed. It can be considered that the security violation can take place only where the function $declassify$ appears in the program. In a broader sense, the *where* declassification can consider a whole function or a part of the code that the level locality policy has to consider safe.

The common approach is the *intransitive noninterference* [Ros99, MS04]. This approach defines how information can flow from one security level to a lower one, allowing, for example, a flow from *high* to *declassifier* and from *declassifier* to *low*. Hence, the security lattice has the following order: $L \sqsubseteq declassifier \sqsubseteq H$. In this case, the security is granted if an attacker with a low view of the system cannot distinguish the secret.

The other possible approach can be seen as an instance of the *intransitive noninterference*, where all the information flow from *high* to *low* has to be done through declassification constructs in the code (i.e., $declassify(h)$). An example of the dimension *Where* could be a program which declassifies a certain variable after doing a big computation, but not before.

When. The authors consider three different cases on the *when* dimension of declassification. In the first case, *time-complexity* based, it is supposed that the secret can be leaked in a non-polynomial time while an attacker has polynomial computational power. Under these assumptions the system satisfies the noninterference property. In the second case, *probabilistic*, a program is considered secure if an attacker has a small probability of distinguishing a secret. In the third case, *relative*, the downgrade of secure information will be done after some stated conditions are verified [SM04]; for example, declassifying a secret after receiving a payment confirmation or after some variables have been written.

It is important to notice that declassification is not always specifically placed in one of these four dimensions. For example, S. Chong and A.C. Myers [CM04] propose a combination of both *what* and *when*, while A. Askarov and A. Sabelfeld [AS07] propose a combination of *what* and *where*. This shows that these four dimensions are not exclusive.

Some of the works about declassification also define a semantic property that a secure program ensures [MR07, LM09, AS07]. However, not all of them define explicitly this security property. Such properties allow a certain abstraction of how information is declassified. Additionally, the formalization of the properties allows the systematization of the security analysis and to prove its correctness.

2.3 Countermeasures

There is a need to find which countermeasures can be applied in the information flow control context. However, there are not too many works implementing countermeasures in this context. Nonetheless, in the context of network protocols, S. Zander *et al.* [ZAB07] present some possible reactions in order to eliminate or limit the possible leaks that take place due to *covert channels*. Presented below are the four main ideas [ZAB07] present as countermeasures for the covert channels (that is, the hiding of information).

Eliminate the channel. Some channels can be detected and eliminated during the design phase. It is noted that the deletion of all the covert channels leads to inefficient systems.

Limit the bandwidth of the channel. Reducing the channel capacity leads to a reduction on the amount of information that can be leaked. The main idea is that the capacity becomes so small that information will be outdated in the moment when the attacker finally has access to the whole data.

Audit the channel. If it is not possible to eliminate a covert channel it is necessary to monitor it. Monitoring the channel state is considered as discouraging for the possible attackers.

Document the channel. When it is not possible to eliminate, reduce the capacity or monitor the covert channel, the only option available is to document the existence of the mentioned covert channel. This way all the users know about its existence and hence the leaking menace decreases because the users are aware of the risks.

In [ZAB07] there are several specific ways for developing the ideas presented above, but given their specificity, this work only presents the general procedures of each one. It is remarked that in general covert channels cannot be totally eliminated. Furthermore limiting those covert channels usually has an impact on the network performance, degrading its characteristics.

To our knowledge, there are two works defining countermeasures in the information flow control context, one by S. Cavadini and D. Cheda [CC08] and the other by G. Le Guernic *et al.* [LGBJS06]. [CC08] eliminates the leak of a program during its execution by modifying the semantics of the program when needed. They monitor the execution with Dynamic Dependence Graphs tracking the changes on the security levels. [LGBJS06] also eliminates the leak, but in this case a monitor tracks how the information flows and it executes some modifications on the semantics at the end of the execution. A detailed study of

the implicit flows while monitoring information flow can be found in the work of G. Le Guernic and T. Jensen [GJ05].

2.4 Conclusion

Noninterference is a property well studied but too restrictive for many widely used programs. In order to escape from the restrictions of noninterference, the different declassification dimensions accept certain flows from high variables to low ones. This accepted release of information can be defined by other semantic properties according to the semantics of their traces. These properties define more complex information flows than the noninterference property. Nonetheless, not all the works define explicitly the property ensured by the declassification. However, such semantic properties allow to abstract the way declassification is done.

The two works implementing countermeasures studied for the present work, [LGBJS06, CC08], use them at a low level, implementing them in order to ensure the noninterference property. As opposed to the proposition of [CC08], [LGBJS06] defines formally the countermeasures and proves them to ensure noninterference. In both works it is not possible to choose the countermeasure to apply.

To provide a more expressive policy, our approach is to define more detailed security properties while allowing the choice of which countermeasures to apply. It seems possible to mix countermeasures with declassification, even when the works studied only apply the countermeasures into noninterfering systems. Countermeasures are meant to be defined at a high level of abstraction, not to specify how they are implemented but how they have to alter the information flow.

When considering different countermeasures, as in [ZAB07], each countermeasure will ensure a different security property. This depends on the limitation of the leak to which the countermeasure applies. If a leak takes place, the resulting system will ensure a weaker security property than the original one, since more information is finally allowed to flow (even if the leak is limited).

On the other hand, countermeasures can make some modifications into the semantics of the system. Such modifications can be formally modeled as a semantic property similar to the ones for declassification or noninterference. Following this idea, a policy might be characterized by the expected semantic property and the properties of the countermeasures to apply. The countermeasures are applied whenever a flow violates the expected property. Both groups of properties ought to be formally defined, allowing a systematization of the security analysis and eventually proving its correctness.

3 Extending the Information Flow Control Properties

Tracking how the information flows through the semantics of a program is imperative in order to assure that the program respects a security property. This section presents an extension to the semantics of the *While* language [NN92] tracking how information flows. Such extension should consider noninterference but also different declassification dimensions, accepting more complex information flows. Studying how the information flows in each case allows to define accurately the semantic properties of the systems under the different declassification dimensions.

Subsection 3.1, below, presents the extended semantics tracking noninterference. Then, Subsection 3.2 defines more complex semantic properties (dimensions *what* and *where*), defining also the necessary modifications over the semantics. Finally, Subsection 3.3 concludes discussing about the implications of a program not ensuring a certain security property.

3.1 The Extended Semantics

The noninterference property of a certain program C with a configuration of some low (public) and high (secret) variables (set_L, set_H) can be formally defined as:

$$\begin{aligned} NI[\![C]\!](set_L, set_H) \quad &\triangleq \forall s_1, s_2 \in S.\ (s_1 =_L s_2) \wedge ([\![C]\!]s_1 \Downarrow, [\![C]\!]s_2 \Downarrow) \\ &\implies [\![C]\!]s_1 =_L [\![C]\!]s_2 \end{aligned} \tag{2}$$

This property ensures that no initial values of high variables affects the final value of the low variables. Below we present an extension of the semantics of the *While* language [NN92] in order to track how information flows (see Figure 3 and 4). In order to define them, it is necessary to extend the configuration containing the command C and memory state s (in $\langle C, s\rangle$) with a tuple $\langle C, s, \sigma_F, \omega\rangle$. σ_F contains the variables which value depends on the initial values of the high variables. ω tracks the context of the execution in a stack (being usually high or low, represented by the variables affecting the context and $\bot$ respectively).

The environment σ_F ($\sigma_F : var \to \{var\}$) keeps track of which variables have information about the initial values of the high variables. For doing so, the initial value of σ_F is $(v \to \{v\})$ for each variable $v \in set_H$. Additionally, the following functions are defined: $dom(\sigma_F)$, $varset(v)$, $add(v, \{var\})$ and $remove(\sigma_F, v)$. The domain $dom(\sigma_F)$ returns a set with the variables contained in σ_F ($dom(\sigma_F) : \sigma_F \to \{var\}$). To obtain the image of a certain variable, $varset(v)$ returns the set of variables from which v is affected ($varset(v) = \sigma_F v : var \to \{var\}$). In order to add or remove some elements of the environment, the functions *add* and *remove* are included. To add an element with its set of variables, $add(v, \{var\})$ is used ($add(v, \{var\}) \equiv \sigma_F[v \to \{var\}] : var, \{var\} \to \sigma_F$), indicating the different variables (in the set $\{var\}$) that affected the value of v. $remove(\sigma_F, v)$ removes the variable v from the environment ($remove(\sigma_F, v) \equiv \sigma_F \setminus [v \to -] : var \to \sigma_F$).

As it can be seen in Figure 3 and 4, an assignment is not leaking information (the rule [ASSIGN-L]) if the context of execution is low ($\bot$) and all the variables

on the expression do not have high information (by checking $dom(\sigma_F)$). In this case, the affected variable is removed from σ_F. The same occurs with the conditionals and loops ([IF-T-L], [IF-F-L], [WHILE-T-L] and [WHILE-F-L]). It is required to add the statement *stop* at the end of the code (corresponding to the rule [STOP]) to terminate the execution.

If the context is not low or some variables contain high information (the rule [ASSIGN-H]), then σ_F is updated because the variable affected contains the information from both sources (the variables in ω and $dom(\sigma_F)$). Conditionals and loops are resolved similarly. If the context of execution contains high information then the modified variables contain information from the context of execution (marked with [H-Context]). If the guard contains high information (marked with [H-Exp]) then the context of execution is updated with this high information and the environment is updated with the non-executed branches.

In order to keep track of the context of execution, the statement **end** is used in the conditionals and loops. It indicates the ending of a branch, hence the exit of the last context of execution. This statement is added in the semantics (see the rules [IF] and [WHILE]) in order to avoid the need of declaring such statement in the code, not modifying the grammar the user can use.

While it is recommended to avoid loops with high guards (to avoid leaking information through termination channels [SM03]), the semantics are not meant to restrict this and hence this possibility is considered by the semantics. On the other hand, $modified(S)$ returns the set of variables modified by the statements in S. This exists in order to detect implicit flows. $modified(S)$ works as described in Figure 2.

$$
\begin{array}{l}
\text{modified}(x := e) = x \\
\text{modified}(skip) = \emptyset \\
\text{modified}(S_1; S_2) = \text{modified}(S_1) \cup \text{modified}(S_2) \\
\text{modified}(\text{if } e \text{ then } S_1 \text{ else } S_2) = \text{modified}(S_1) \cup \text{modified}(S_2) \\
\text{modified}(\text{while } e \text{ do } S) = \text{modified}(S)
\end{array}
$$

Figure 2: Description of how *modified* is composed

The function $var(exp)$ returns the set of variables appearing in the expression exp ($var : exp \to \{var\}$), while the function $modified(C)$ returns the set of variables in the statements of C ($modified : statement \to \{var\}$). We also define the function $vars(\omega)$ ($vars : \omega \to \{var\}$) which returns the set of variables contained in ω (hence, discarding the low contexts, $\bot$).

With this extension over the semantics of the *While* language, at the end of the execution the environment σ_F contains the set of variables affected by the initial values of the high variables. If no variable in this environment belongs to set_L ($set_L \cap dom(\sigma_F) = \emptyset$), then the execution satisfies noninterference.

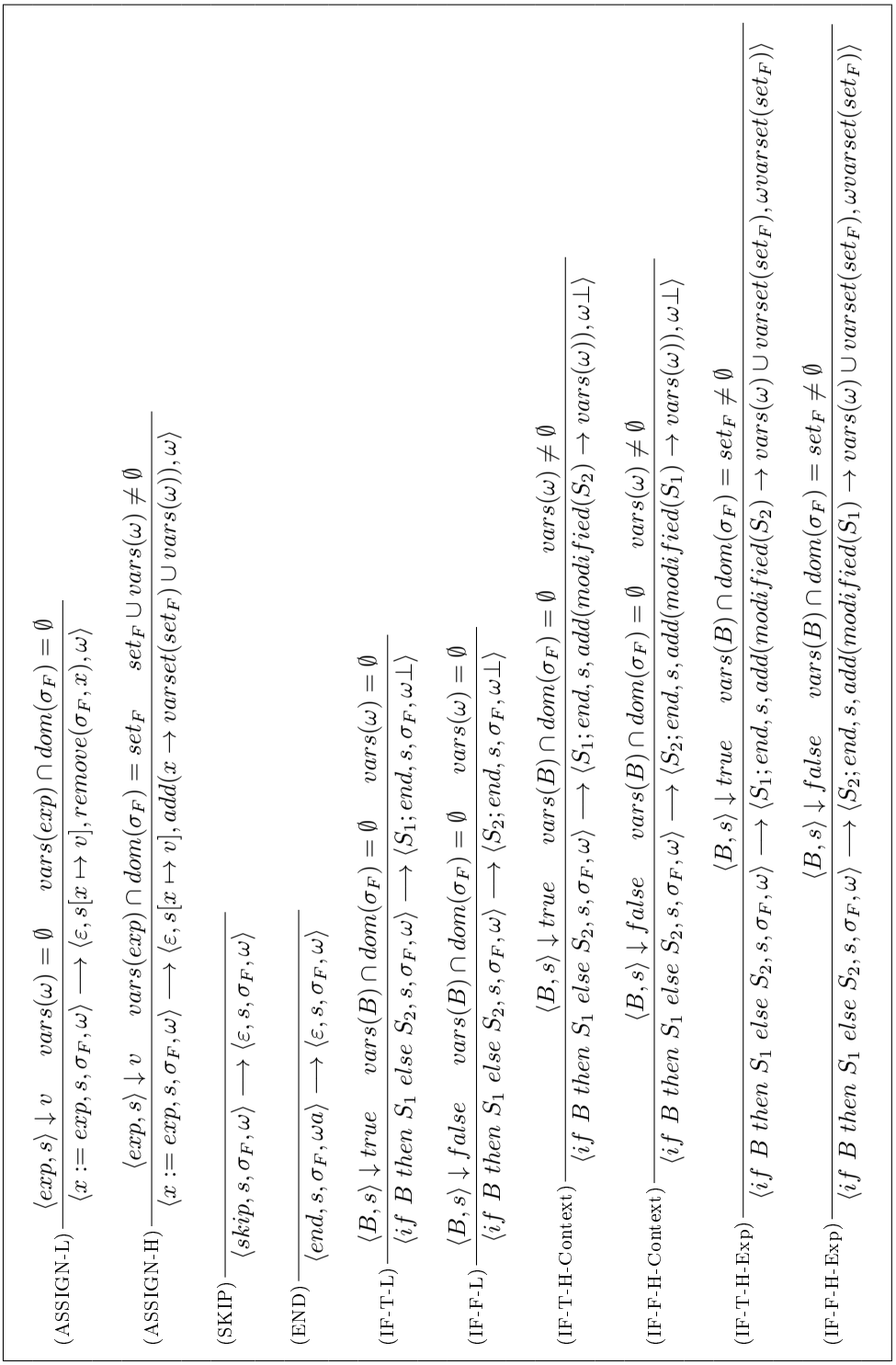


Figure 3: Extended small-step semantics tracking information flow (A)

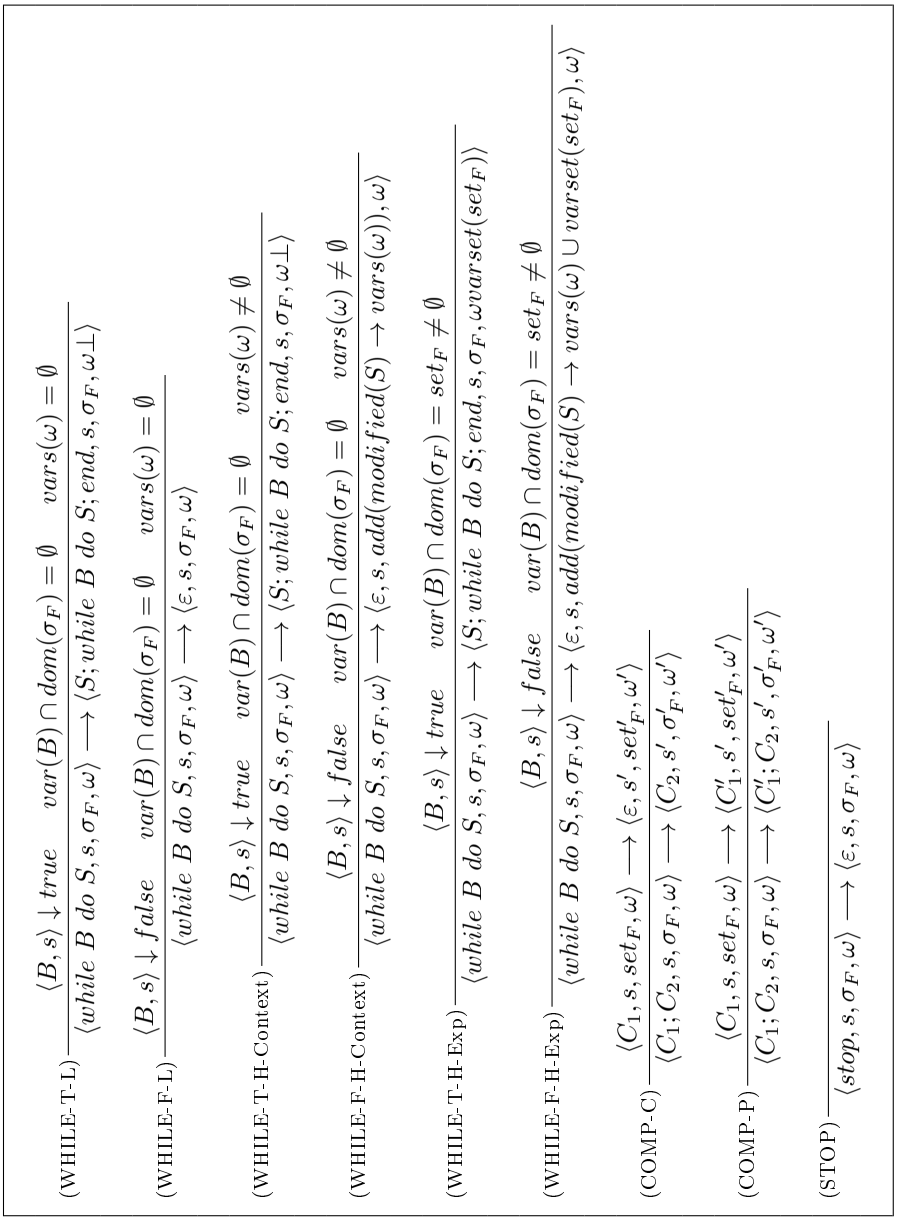


Figure 4: Extended small-step semantics tracking information flow (B)

In order to show how the semantics track the information flow, we present in Table 1 a code as an example and analyze how the semantics affect the resulting configuration. This example intends to show the different flows a program generates and how the environment σ_F tracks it. The statement **end** is added in the code (it is added by the semantics) in order to clearly show the updating of the context of execution.

On line 2, h_2 contains the information of h_1 and h_2. On line 3, the variables in the guard of the conditional (h_1) are added to the context of execution, while the modified variables in the non-executed branch are added to σ_F (on line 5, l_1, depending on h_1). On line 7, l_1 is removed from σ_F. On line 10 h_1 is also reset and hence removed from σ_F. On the other hand, on line 11 the assignment makes l_1 depend on the variables which h_2 depends on (in this case, h_1 and h_2).

	Program	Elements in σ_F	Elements in ω
		$h_1 \to \{h_1\}\ h_2 \to \{h_2\}$	$\{\epsilon\}$
1	**if** $l_1 > 5$	$h_1 \to \{h_1\}\ h_2 \to \{h_2\}$	$\{\bot\}$
2	$h_2 := h_1 + h_2$	$h_1 \to \{h_1\}\ h_2 \to \{h_1, h_2\}$	$\{\bot\}$
3	**if** $h_1 = 10$	$h_1 \to \{h_1\}\ h_2 \to \{h_1, h_2\}\ l_1 \to \{h_1\}$	$\{\bot\{h_1\}\}$
4	$l_2 := 7$	$h_1 \to \{h_1\}\ h_2 \to \{h_1, h_2\}$ $l_1 \to \{h_1\}\ l_2 \to \{h_1\}$	$\{\bot\{h_1\}\}$
5	**else** $l_1 := 3$		
6	**end**	$h_1 \to \{h_1\}\ h_2 \to \{h_1, h_2\}$ $l_1 \to \{h_1\}\ l_2 \to \{h_1\}$	$\{\bot\}$
7	$l_1 := 0$	$h_1 \to \{h_1\}\ h_2 \to \{h_1, h_2\}\ l_2 \to \{h_1\}$	$\{\bot\}$
8	**else** $l_1 := 3$		
9	**end**	$h_1 \to \{h_1\}\ h_2 \to \{h_1, h_2\}\ l_2 \to \{h_1\}$	$\{\epsilon\}$
10	$h_1 := 0$	$h_2 \to \{h_1, h_2\}\ l_2 \to \{h_1\}$	$\{\epsilon\}$
11	$l_1 := h_2$	$h_2 \to \{h_1, h_2\}\ l_2 \to \{h_1\}\ l_1 \to \{h_1, h_2\}$	$\{\epsilon\}$
12	**stop**	$h_2 \to \{h_1, h_2\}\ l_2 \to \{h_1\}\ l_1 \to \{h_1, h_2\}$	$\{\epsilon\}$

Table 1: Example of the functioning of the extended semantics.

3.2 Properties of each Declassification Dimension

This Subsection presents the semantic properties ensured by the dimensions *what* and *where*. These properties define more complex information flows than noninterference, taking into account a certain policy for releasing information. The dimension *What* is affected by the initial values of the memory state. On the other hand, the dimension *Where* takes into account the trace generated during the execution, requiring a more detailed analysis of the resulting trace.

Since the semantics consider the policy, this subsection also presents some modifications into the semantics for appropriately tracking the information flow. Defining both semantic properties allows to semantically define more expressive information flows than noninterference.

3.2.1 Dimension What

This dimension states *what* information can be released (like the *delimited release* in [SM04]). The present work considers declaring in the property which are the

accepted uses of the secret information. The semantics do not update σ_F when a flow from high to low is accepted by this dimension.

Let us consider the set $EDecl$ in the property, which contains the information of which expressions are accepted upon which variables. That is, $EDecl$ is formed by a set of $quant_n(v*)exp$, stating which expressions are accepted but also upon which specific variables. On example would be, accepting to publish the *hash* of a variable, with $hash(var)$. In the case where different variables are affected, as in the average salary example from subsection 2.2, $average(s_1, s_2, ..., s_n)$. Then, the semantics accept such release by not updating σ_F if the used variables and the function applied correspond to the information in $EDecl$. For more details, see Appendix A.1.

With this formalization of the accepted information release, it is possible to express the security property of this dimension. Recalling the property defined in the *delimited release* [SM04]: given two initial states s_1 and s_2 equivalent at the low level, if the resulting values of the accepted information release are the same, the final state s_1' and s_2' will be indistinguishable at low level. In the present case, it is necessary to compute the resulting value of each expression $(quant_n(v^*)\ exp)$. Then, for all $quant_n(v^*)\ exp_i \in EDecl$, the resulting values for both initial states have to be the same. This constraint takes into account the memory state at the beginning of the program execution, not considering modifications on the declassified variables over the execution. Therefore the property can be formally defined as.

Definition 1 (Property *What*)**.**

$$
\begin{aligned}
&WHAT[\![C]\!](set_L, set_H, EDecl)\\
&\triangleq \forall s_1, s_2.\ \{s_1 =_L s_2 \wedge ([\![C]\!]s_1 \Downarrow, [\![C]\!]s_2 \Downarrow)\\
&\wedge \forall exp \in Edecl\ (\langle quant_n(v^*)\ exp, s_1\rangle = \langle quant_n(v^*)\ exp, s_2\rangle)\}\\
&\implies [\![C]\!]s_1 =_L [\![C]\!]s_2
\end{aligned}
$$

This property is satisfied whenever a program C obtains the same values on the low variables after applying the semantics into any memory state which has the same values on the low variables and the application of the expressions in $EDecl$ also entail the same value. In the absence of declassification, this property is equivalent to the noninterference property. In the presence of declassification, the property states that the low level resulting state is also affected by the results of the declared expressions in the property.

Recalling the example of the average salary from subsection 2.2, an observer cannot see a difference between, for example, the following two collection of values: 1) $sal_1 = 25, sal_2 = 75, sal_3 = 50$ and 2) $sal_1 = 50, sal_2 = 50, sal_3 = 50$, because the only constraint on the high level data is about the result of the expression, in this case, $(sal_1 + sal_2 + sal_3)/3$. It has been stated that the constraints take into account the initial values. This constraints cancels the possibility of leaking a certain value by modifying the values of some of the variables. For example, executing $sal_2 = sal_1;\ sal_3 = sal_1$ before declassifying the average salary, consequently declassifying directly the value of sal_1.

3.2.2 Dimension Where

This dimension states *where* the information is released, usually treated as where in the code the declassification is placed. The security violation takes place if the values of high variables are leaked into low ones in a non specified statement of the code.

The information that can be released is declared in the set $WDecl$, which also contains the code line where it can be released. The semantics have to indicate the accepted information release by adding a subscript I in these transitions (that is, $\longrightarrow_I$). The other transitions are not affected (either they are not releasing information, or the information is flowing in a non-declared statement of the code).

By indicating the accepted information release like this, a program trace presents some regular transitions ($\longrightarrow$) and other interfering transitions ($\longrightarrow_I$). For more details about the semantics, see Appendix A.2. The property expected for a safe system with the dimension *Where* has to compare the values of the released variables. That is, the regular transitions ($\longrightarrow$) are not to be analyzed; instead, the interfering transitions are analyzed. To do this, we propose a transformation over the resulting trace by joining a whole subset of regular transitions into a sole one. The resulting trace is defined by $\mathcal{T}(\langle C, s_1\rangle)$.

The transformation only considers the initial and final configurations of a subset of regular transitions, while the interfering transitions remain unaltered. For a detailed description of such transformation, see Appendix B. Over the resulting trace, containing some regular and some interfering transitions, the property constraints are located into the interfering ones. For defining the security property expected for this dimension, let us recall the definition of strong bisimulation (as in [MS04], for a sequential program):

Definition 2 (Strong Low-Bisimulation)**.** *The strong low-bisimulation is the union of all symetric relations $R \subseteq S \times S$ such that:*

$$\forall s_1, s_2, s_1' : \forall c_1' : \{(\langle c_1, s_1\rangle R \langle c_2, s_2\rangle) \wedge (s_1 =_L s_2) \wedge (\langle c_1, s_1\rangle \longrightarrow \langle c_1', s_1'\rangle)\}$$
$$\Longrightarrow \exists c_2', s_2' : \{(\langle c_2, s_2\rangle \longrightarrow \langle c_2', s_2'\rangle) \wedge (\langle c_1', s_1'\rangle R \langle c_2', s_2'\rangle) \wedge (s_1' =_L s_2')\}$$

In the case presented here, having two possible transitions, it is necessary to distinguish between the noninterfering transition or the one releasing high information. Additionally, we consider no constraints over the low variables, hence:

Definition 3 (Strong Bisimulation with Declassification)**.** *The strong bisimulation with declassification (marked $\approxeq$) is the union of all symmetric relations R such that:*

$$\begin{array}{ll} & \forall s_1, s_2, s_1' : \forall c_1' : \\ (a) & \{(\langle c_1, s_1\rangle R \langle c_2, s_2\rangle) \wedge (\langle c_1, s_1\rangle \longrightarrow \langle c_1', s_1'\rangle)\} \\ & \quad \Longrightarrow \exists c_2', s_2' : \{(\langle c_2, s_2\rangle \longrightarrow \langle c_2', s_2'\rangle) \wedge (\langle c_1', s_1'\rangle R \langle c_2', s_2'\rangle)\} \\ & \wedge \\ (b) & \{(\langle c_1, s_1\rangle R \langle c_2, s_2\rangle) \wedge (\langle c_1, s_1\rangle \longrightarrow_I \langle c_1', s_1'\rangle) \\ & \quad \wedge (\forall v \in (vars(c_1) \cup vars(c_2)) \cap set_H.\ s_1(v) = s_2(v))\} \\ & \quad \Longrightarrow \exists c_2', s_2' : \{(\langle c_2, s_2\rangle \longrightarrow_I \langle c_2', s_2'\rangle) \wedge (\langle c_1', s_1'\rangle R \langle c_2', s_2'\rangle)\} \end{array}$$

Where (a) stands for the set of transitions without information release declared by the policy and (b) stands for the accepted information release. It means that the accepted transition for the configurations are equivalent and the released variables from set_H (when there are any) have to have the same value. Let us define $\langle c_1, s_1\rangle \cong \langle c_2, s_2\rangle$ when two configurations $\langle c_1, s_1\rangle$ and $\langle c_2, s_2\rangle$ ensure the strong bisimulation with declassification.

As opposed to the Strong Low-Bisimulation (Definition 2), Definition 3 does not restrict the low values of the system. The constraint over the initial and final variables is located in the general equation (and considering the transformed traces $\mathcal{T}(\langle C, s_1\rangle)$):

Definition 4 (Property *Where*)**.**

$$\begin{aligned} &WHERE[\![C]\!](set_L, set_H, WDecl) \\ &\triangleq \forall s_1, s_2.\ \{s_1 =_L s_2 \wedge ([\![C]\!]s_1 \Downarrow, [\![C]\!]s_2 \Downarrow) \\ &\wedge \mathcal{T}(\langle C, s_1\rangle) \cong \mathcal{T}(\langle C, s_2\rangle)\} \\ &\implies [\![C]\!]s_1 =_L [\![C]\!]s_2 \end{aligned}$$

This property defines that for any memory states with the same values on the low variables, and the same values on the released high variables in the moment they are released, the resulting low variables have the same value. In the absence of declassification, $\langle C, s_1\rangle \cong \langle C, s_2\rangle$ stands and hence this property entails the noninterference property. The property considers that any release of information not declared on the policy is not allowed. This is because the infringement of the policy is tracked through the small-step semantics.

3.3 Conclusion

This section presents the extended semantics for tracking how information flows. The main development is done over the basic extension tracking noninterference, then adding some modifications in order to track each different dimension since the accepted flows are more complex.

The semantic properties presented take into account if a program respects or not a certain policy, and therefore stating if they can be considered *secure* under such property. For one sole execution, it is possible to see if it ensures the property by checking the environment σ_F. If $\sigma_F \cap set_L = \emptyset$ then the execution ensures the security property. Considering a sole execution to ensure a security property is introduced by G. Le Guernic *et al.* in [LGBJS06], rather than asserting if the program ensures the security property for all the possible executions.

The *What* property takes into account the initial memory states, being the semantic modifications necessary to track the leaks into σ_F (but not to define the semantic property). On the other hand, the *Where* property entails a more complex information flow, hence the distinction between regular transitions and interfering transitions. Additionally, the present work presents the transformation over the resulting trace in order to define a less restrictive bisimulation. In [MS04] the bisimulation also forces the low variables to have the same values for each transition, which we consider too restrictive with regard to the noninter-

ference property. As a result, the two formally defined properties express more complex information flows than noninterference.

We consider that the information flows generated by the *relative* case of the *When* dimension are similar to those generated by the dimension *Where*. Hence, it is not defined in detail in the present work. For a more detailed explanation, see Appendix A.3.

These properties are ensured if the system respects the policy, but makes no modification if the policy is violated. Since all the unaccepted flows are tracked in σ_F, it is possible to apply a certain reaction in order to make the execution ensure some security property. This subject is presented in the next section.

4 Defining the Countermeasures

Countermeasures are the reactions to the leakage of private information. An expressive security policy should specify different alternatives for reacting to leaks in order to avoid or at least minimize such leaks. Allowing to choose the countermeasures improves the expressiveness of the resulting policy. This work presents several different propositions and discusses their possible use in the context of information flow control.

Such possible reactions are inspired on the work of S. Zander *et al.* focused on Network Protocols [ZAB07]. Their work considers the leaks through *covert channels*. In the context of information flow control, leaks take place when an attacker obtains private information he is not supposed to access through the low variables.

Subsection 4.1, below, presents some available countermeasures. Subsection 4.2 discusses the possibility of applying the countermeasures at different moments of the execution of a program. Finally, Subsection 4.3 sums up the use of countermeasures in the information flow control context and proposes two principles for the definition of other countermeasures.

4.1 Possible Countermeasures

A security policy should present different reactions in order to avoid or at least minimize the leaks a program can create. While this work proposes eight different methodologies to respond to a leak, they can be classified into the following five different categories:

- Eliminate the leak
- Limit the bandwidth of the leak
- Limit the leak along the time
- Monitor the leak
- Document the leak

4.1.1 Eliminate the Leak

The first option is to avoid completely the leak. The possible ways to do so depend on the type of flow that leaks the secret and the moment they are applied. Below are presented the different countermeasures eliminating the leak.

Stop the execution. Stopping the execution of the program avoids the leak and impede the program to continue. This proceeding drastically changes the semantics of the program. Nevertheless, it would affect two low-equivalent traces in the same way. This countermeasure can only be applied during the program execution, right before an assignment may leak a secret.

Replace by skip. Replacing the operation (or operations) by a skip is another possible option. This countermeasure, as well as *stopping the execution*, can only be applied during the program execution because there is no statement to skip at the termination of the execution.

Assign default value. Assigning a default value avoids the leak without changing drastically the semantics. This countermeasure can be applied both during the program execution and at the termination of the same.

Assign random value. Assigning a random value is a weakening of *assigning default value*. Unlike the previous one, it assigns some randomly chosen values to the variables containing information about high variables.

4.1.2 Limit the Bandwidth of the Leak

The second option is to allow only part of the leak. That is, leaking only certain information of the variable instead of the whole secret. Similarly to the declassification dimension *What*, applying a function to the leaked variable forbids a possible attacker to learn the whole secret. In more detail, when a certain variable h is about to be leaked (or flows into a low variable), the value that is published is a function of its value, for example, $parity(h)$. This way some information is given but the possible attacker is not able to learn the whole secret. It would be necessary to specify for each policy which function should be used when applying this countermeasure.

4.1.3 Limit the Leak Along the Time

Another possible option is to allow the leak, but increase the time it takes to leak. This way, a brute-force attack is more time-demanding and decreases its efficiency. Furthermore, it may be possible that when the secret is finally learned it is already outdated. This countermeasure has a more visible effect when it is applied during the execution of the program, adding the delay for each forbidden access to certain variables. On the other hand, if it is applied when the program terminates, the delay is applied once, at the end of the execution.

4.1.4 Monitor the Leak

Monitoring the leak means that the system keeps track of the leaked variables (according to the extended semantics in subsection 3.1, in σ_F). This way, even when the secret is leaked and the program does not ensure the security property, it is possible to know which specific variables have been leaked.

This countermeasure is supposed to analyze the program during its execution and informing about the leaked variables, yet it has to be applied at the end of the execution when the leaks can be stated for sure.

4.1.5 Document the Leak

The last countermeasure considered in this work is to inform of the leaks a certain program can generate. This way, the leaks take place and no limitation is done. Nevertheless, this procedure is less accurate than *monitor the leak* because it informs of the leaks that *can take place*.

This countermeasure analyzes the whole program and consider all the possible executions. However, this procedure is not considered in this work because

the approach is different than for the other possible reactions, considering all the possible executions on a program instead of a certain execution.

This subsection presents a summary of the different countermeasures proposed. Not all of the countermeasures are supposed to enforce the original security property (usually noninterference). Some possibilities aim to hinder a possible attacker to learn some secrets. As a last option, other possibilities keep track of the secrets leaked by the program (leaked or potentially leaked).

4.2 When are Countermeasures Applied

Countermeasures can be applied at different moments. On the one hand, according to when a leak is considered to take place, on the other hand, according to the moment it is desired to avoid or limit such leak. In the work of G. Le Guernic et al. [LGBJS06] countermeasures are applied at the latest moment possible, when a variable is about to be output and the variable contains high information. On the other hand, in the work of S. Cavadini and D. Cheda [CC08] the countermeasures are applied as soon as a forbidden flow is taking place, whether it is really a threat or not (it might be possible that the flow is lately avoided with an assignment).

Depending on how the countermeasures are applied, some countermeasures do not react to the leak as they are supposed to. This subsection intends to point out such details. We make a distinction between applying the countermeasures at the very first moment of the potentially forbidden flow (referring to it as *when the information flows*) and applying them when the program is about to end (referring to it as *when the leak is about to take place*).

Applying countermeasures when the information flows. In this case all the countermeasures previously presented can be applied. The option of *limiting the leak along the time* is specially useful in this case, where each access slows down the execution time. However, these countermeasures might be applied unnecessarily. If it occurs a flow from *high* to *low* and then the *low* variable is reset, the leak is not taking place at all, yet the countermeasure has been applied.

Applying countermeasures when the leak is about to take place. In this case not all the countermeasures can be applied because they don't have the expected effect. Neither *replace by skip* nor *stop the execution* would avoid the leak as they are supposed to. On the other hand, as it can be noticed, the option of *limiting the leak along the time* would not have the same effectiveness (slowing down the execution) since the delay would be applied only once at the end of the execution. However, proceeding this way assures that countermeasures are applied only when leaks are really occurring, as opposed to the previous option.

The present work considers the latter development (applying the countermeasures at the end of the execution) in order not to alter the program semantics if it directly ensures a certain security property. Nevertheless, all the countermeasures ought to be presented, specifying which ones can be used with each choice. Such differences are summarized in Table 2, showing for each countermeasure if it can be used *when information flows* or *when leak takes place*.

Countermeasures		When information flows	When leak takes place
Eliminate the leak	Stop the execution	✓	-
	Replace by skip	✓	-
	Assign default value	✓	✓
	Assign random value	✓	✓
Limit the bandwidth		✓	✓
Limit along the time		✓	✓
Monitor the leak		-	✓

Table 2: Available countermeasures according to *when* they are applied.

4.3 Conclusion

This section discusses some possible countermeasures and their application. Depending on the moment they are applied, some countermeasures cannot be applied at all, or the way they are applied has to change. This work applies the countermeasures at the end of the execution in order not to modify the semantics of programs that are not really leaking information. The semantics presented in Section 3 track the information flow and the environment σ_F contains the variables to which countermeasures ought to be applied.

The countermeasures presented in this section are not the only possible ones. It is possible to combine them but also to define new ones. One example can be to limit the bandwidth of a leak while also limiting it along the time, applying both countermeasures.

On the other hand, the present section does not specify how each countermeasure is to be applied. For example, when having an indirect flow from *high* to *low* with $if(h)\ then\ l := 2\ else\ skip$, the countermeasure limiting the bandwidth could be applied either to the result of the guard (h in this case) or to the assignment to l. During the implementation of the countermeasures such details have to be resolved, as shown in the following section.

Since other countermeasures can be proposed, or the application mechanisms could change, it is crucial to have some checkpoints to help in the countermeasures choice. This work proposes the *non-altering* and *active* principles.

Active. For all programs not ensuring the security property of the policy, an active countermeasure makes the resulting execution to ensure the security property.

Non-altering. For all programs ensuring a certain security property, a non-altering countermeasure makes the resulting execution to keep ensuring the security property.

The *active* principle intends to check if the countermeasure makes the execution to satisfy the security property or another (weaker) one. On the other hand, the *non-altering* principle intends to ensure that a countermeasure does not turn a secure program into an insecure one.

The executions which are affected by countermeasures ensure a certain security property. To formally define it, it is imperative to analyze how countermeasures interact with the execution when considering a certain security policy. This interaction is presented in more detail in the following section.

5 Defining a Formal Expressive Policy

This section presents a new approach to the information flow control policies by considering countermeasures in addition to a certain security property. The security property specifies the permitted information flow, which can be more complex than noninterference. Countermeasures improve the expressiveness of the policy. This section also analyzes the resulting semantic properties of executions considering the policy.

Subsection 5.1, below, specifies how the policy is defined and presents the semantics of the countermeasures. Subsection 5.2 studies the resulting properties of a policy by combining the security property with the different countermeasures. Finally, Subsection 5.3 concludes summing up the presented policy and the resulting properties.

5.1 Specifying a Policy

The security policy, according to our approach, is formed by an information flow property and some associated countermeasures. The countermeasures can be specified to each variable of set_H individually. This way, a certain variable h_1 can have assigned the countermeasure *assign default value* (referred as Φ) while h_2 can have assigned the countermeasure *limit the bandwidth of the leak*, with a certain function (referred as $BW(function)$, for example, $BW(parity)$).

The property defines the accepted information flow, hence the semantics behavior, and whenever the property is not ensured (if $set_L \cap dom(\sigma_F) \neq \emptyset$) the countermeasures are triggered and execute certain modifications on the resulting low variables of the system.

Combining both elements (information flow property and countermeasures), we define a security policy as a property and the associated countermeasures. For example, with the noninterference property that results in:

$$Pol = NI(set_L, set_H) \triangleright CM(set_\Phi, set_{rnd}, set_{BW(f)}, set_{t_0}, set_{track})$$

where other properties than NI can be declared, as they have been presented in Subsection 3.2. Each set for countermeasures contains high variables, and each high variable has only one countermeasure assigned. Additionally, the whole set_H has a countermeasure assigned (that is, $\bigcup set_i = set_H$).

The following subsection analyzes the resulting properties of this policy. For doing so we define the semantics for the countermeasures in Figure 5. We consider that a program finishes with the statement *stop*. While applying the countermeasures, *stop* is called recursively. The program finishes when there are no variables to apply the countermeasures (when $dom(\sigma_F) \cap set_L = \emptyset$), as it can be seen in the case [nothing].

Each rule is applied if there are some low variables in the environment σ_F and the high variables they depend on are defined in each set_i. The rule executes some modifications into the low variables (in [default], [random] and [BW(f)]) and the environment is updated by removing the low variable from its domain. In the case of [time] and [track], there is no modification on the low variables. Instead, [time] applies a certain delay (noted with $wait(t_0)$) and [track] informs

$$\text{default}\ \frac{dom(\sigma_F) \cap set_L = \{x\} \neq \emptyset \qquad varset(x) \cap set_\Phi \neq \emptyset}{\langle stop, s, \sigma_F, \omega\rangle \longrightarrow \langle stop, s[x \mapsto \Phi], remove(\sigma_F, x), \omega\rangle}$$

$$\text{random}\ \frac{dom(\sigma_F) \cap set_L = \{x\} \neq \emptyset \qquad varset(x) \cap set_{rnd} \neq \emptyset}{\langle stop, s, \sigma_F, \omega\rangle \longrightarrow \langle stop, s[x \mapsto rnd], remove(\sigma_F, x), \omega\rangle}$$

$$\text{BW(f)}\ \frac{dom(\sigma_F) \cap set_L = \{x\} \neq \emptyset \qquad varset(x) \cap set_{BW} \neq \emptyset \qquad f(x) = v}{\langle stop, s, \sigma_F, \omega\rangle \longrightarrow \langle stop, s[x \mapsto v], remove(\sigma_F, x), \omega\rangle}$$

$$\text{time}\ \frac{dom(\sigma_F) \cap set_L = \{x\} \neq \emptyset \qquad varset(x) \cap set_{t_0} \neq \emptyset}{\langle stop, s, \sigma_F, \omega\rangle \longrightarrow \langle wait(t_0); stop, s, remove(\sigma_F, x), \omega\rangle}$$

$$\text{track}\ \frac{dom(\sigma_F) \cap set_L = \{x\} \neq \emptyset \qquad vars = varset(x) \cap set_{track} \neq \emptyset}{\langle stop, s, \sigma_F, \omega\rangle \longrightarrow \langle alert(vars); stop, s, remove(\sigma_F, x), \omega\rangle}$$

$$\text{nothing}\ \frac{dom(\sigma_F) \cap set_L = \emptyset}{\langle stop, s, \sigma_F, \omega\rangle \longrightarrow \langle \varepsilon, s, \sigma_F, \omega\rangle}$$

Figure 5: Countermeasures in the extended semantics

of the leaked variables (noted with $alert(vars)$). When no low variables are present in σ_F, the program terminates.

In the case where different variables have different countermeasures assigned, it is necessary to state a hierarchy (from more restrictive to less restrictive) for applying the countermeasures. Let us imagine a low variable l_1. After the execution of a program the environment σ_F indicates that l_1 depends on the values of both h_1 and h_2. The countermeasure assigned to h_1 is *default* because it is a *top secret* variable while h_2 has assigned the *limit the leak along the time* because it is a variable that is updated regularly. In this case it is imperative to apply the *default* countermeasure in order to avoid the leak from h_1, even if h_2 does not require it. This example shows the need of such hierarchy.

We present in Table 3 an example of how are the countermeasures applied, retaking the example from Subsection 3.1. Let h_1 have the countermeasure $BW(parity)$ (limit the bandwidth) and h_2 the countermeasure Φ (assign default value). Table 3 shows the semantics modifications after the statement *stop* in the column *Actions*. On line 12 the first countermeasure applied is to l_1, assigning a default value because of h_2 as noted above. On line 13 l_2 is modified applying the function *parity*. On line 14 $dom(\sigma_F) \cap set_L = \emptyset$ and then the program terminates.

	Statement	Elements in σ_F	Actions
12	**stop**	$h_2 \to \{h_1, h_2\}\ l_2 \to \{h_1\}$	$s[l_1 \mapsto \Phi]$
13	**stop**	$h_2 \to \{h_1, h_2\}$	$s[l_2 \mapsto v]$ where $v = parity(l_2)$
14	ϵ	$h_2 \to \{h_1, h_2\}$	$\{\epsilon\}$

Table 3: Example of the functioning of the countermeasures.

5.2 Defining the Resulting Properties of the Policy

Defining a security policy with property and countermeasures allows to define a richer policy. By specifying the policy with

$$Pol = PROP \triangleright CM(set_\Phi, set_{rnd}, set_{BW}, set_{t_0}, set_{track})$$

where $PROP$ stands for a security property, a certain user can state how information can flow during the execution of a program. When the program generates some unaccepted flows by the property, the countermeasures execute certain modifications on the semantics.

Defining the resulting property of a certain policy is a complex task because different cases can be considered. This subsection presents the properties ensured by a policy where all the high variables in set_H have the same assigned countermeasure. In here we distinguish between the semantics of a program $[\![C]\!]$ and the extended semantics $[\![C]\!]_{ext}$. Additionally, the application of a certain countermeasure at the end of the execution is indicated with $[\![C]\!]^{CM}_{ext}$.

Below we prove the resulting properties of a noninterfering policy with a sole set of countermeasures: $Pol = NI(set_L, set_H) \triangleright CM(set_H)$. Analyzing one countermeasure at a time defines how each countermeasure affects the policy. When having a combination of countermeasures, the properties become a combination or relaxation of the presented ones. The properties of a policy combining countermeasures is not studied due to the many possible combinations.

5.2.1 Assign Default Value (CM_Φ)

Theorem 5.2.1 (Assign default value)**.**

$$\forall C, s_1, s_2.\ (s_1 =_L s_2) \wedge ([\![C]\!]s_1 \Downarrow, [\![C]\!]s_2 \Downarrow) \implies [\![C]\!]^{CM_\Phi}_{ext} s_1 =_L [\![C]\!]^{CM_\Phi}_{ext} s_2$$

That is, for all program C and the configuration (set_L, set_H), any execution under the extended semantics and applying the *default* countermeasure ensures the noninterference with the same configuration.

Proof. The theorem follows directly from Lemmas 5.1 and 5.2. Since the equality stands for the two subsets $set_L \setminus dom(\sigma_F)$ (low variables not containing high information, in Lemma 5.1) and $set_L \cap dom(\sigma_F)$ (low variables containing high information, in Lemma 5.2), it is also true for the union, obtaining the whole $set_L = (set_L \setminus dom(\sigma_F)) \cup (set_L \cap dom(\sigma_F))$. □

Lemma 5.1 (Low variables not affected by high variables)**.**

$$\begin{aligned}\forall C, s_1, s_2. \quad & (s_1 =_L s_2) \wedge ([\![C]\!]_{ext} s_1 \Downarrow s_1', [\![C]\!]_{ext} s_2 \Downarrow s_2') \\ & \implies s_1'(set_L \setminus dom(\sigma_F)) = s_2'(set_L \setminus dom(\sigma_F))\end{aligned}$$

The resulting states of $s_1, s_2 \mid s_1 =_L s_2$ are equivalent in $set_L \setminus dom(\sigma_F)$ (that is, the low variables not containing high information) at the end of the execution with the extended semantics.

Proof (sketch). Proved by induction. Assuming that the n state ensures the equation, let us prove the $n+1$ state also ensures it for every possible transition. Let us prove that, given a certain pair of states s_1, s_2 such that $s_1(set_L \setminus dom(\sigma_F)) = s_2(set_L \setminus dom(\sigma_F))$, after any possible transition the resulting states also ensure $s_1'(set_L \setminus dom(\sigma_F')) = s_2'(set_L \setminus dom(\sigma_F'))$. Below the proofs are shown for some of the possible transitions:

[SKIP] then we have:
The semantics maintain the memory and environment $s_i' = s_i$ and $\sigma_F' = \sigma_F$.
Since $s_1(set_L \setminus dom(\sigma_F)) = s_2(set_L \setminus dom(\sigma_F))$ stands
then $s_1'(set_L \setminus dom(\sigma_F')) = s_2'(set_L \setminus dom(\sigma_F'))$ also stands true.

[ASSIGN-L] then we have:
$\langle exp, s_1\rangle \downarrow v_1$ and $\langle exp, s_2\rangle \downarrow v_2$
$vars(exp) \cap dom(\sigma_F) = \emptyset \implies vars(exp) \subseteq (set_L \setminus dom(\sigma_F)) \implies v_1 = v_2$
Then $s_1' = s_1[x \mapsto v_1]$ and $s_2' = s_2[x \mapsto v_2]$ with $v_1 = v_2$
and σ_T' is modified with $dom(\sigma_F') = dom(\sigma_F) \setminus \{x\}$
Depending on the variable x the resulting set can be:
- case $x \in set_L$: $set_L \cap dom(\sigma_F') \subset set_L \cap dom(\sigma_F)$
- case $x \in set_H$: $set_L \cap dom(\sigma_F') = set_L \cap dom(\sigma_F)$
Then $set_L \cap dom(\sigma_F') \subseteq set_L \cap dom(\sigma_F)$
and $s_1'(set_L \setminus dom(\sigma_F')) = s_2'(set_L \setminus dom(\sigma_F'))$ stands true.

[ASSIGN-H] then we have:
$\langle exp, s_1\rangle \downarrow v_1$ and $\langle exp, s_2\rangle \downarrow v_2$
and since $set_F \cup vars(\omega) \neq \emptyset$, $\sigma_F' \subset \sigma_F$
and $dom(\sigma_F') = dom(\sigma_F) \cup \{x\}$
Then $set_L \cap dom(\sigma_F') = set_L \cap (dom(\sigma_F) \cup \{x\})$
since only x is modified, $s_1'(set_L \setminus dom(\sigma_F')) = s_2'(set_L \setminus dom(\sigma_F'))$ stands true. □

Lemma 5.2 (Low variables affected by default value)**.**

$$\forall C, s_1, s_2. \quad (s_1 =_L s_2) \wedge ([\![C]\!]^{CM_\Phi}_{ext} s_1 \Downarrow s_1', [\![C]\!]^{CM_\Phi}_{ext} s_2 \Downarrow s_2') \implies s_1'(set_L \cap dom(\sigma_F)) = s_2'(set_L \cap dom(\sigma_F))$$

The resulting states of $s_1, s_2 \mid s_1 =_L s_2$ are equivalent in $set_L \cap dom(\sigma_F)$ (that is, the low variables containing high information) after the application of the *default* countermeasure.

Proof. All the variables in $set_L \cap dom(\sigma_F)$ are affected by the *default* countermeasure by assigning them the same value Φ. It directly follows that $s_1'(set_L \cap dom(\sigma_F)) = s_2'(set_L \cap dom(\sigma_F))$. □

5.2.2 Assign Random Value (CM_{rnd})

Theorem 5.2.2 (Assign random value)**.**

$$\forall C, s_1, s_2. \quad (s_1 =_L s_2) \wedge ([\![C]\!]s_1 \Downarrow, [\![C]\!]s_2 \Downarrow) \wedge (\{rnd_1\} = \{rnd_2\}) \implies [\![C]\!]^{CM_{rnd}}_{ext} s_1 =_L [\![C]\!]^{CM_{rnd}}_{ext} s_2$$

For all program C and the configuration (set_L, set_H), any execution under the extended semantics and applying the *random* countermeasure ensures noninterference with the same configuration if the assigned values match (characterized with the set $\{rnd_i\}$).

Proof. The theorem follows directly from Lemmas 5.1 and 5.3. Since the equality stands for the subset $set_L \setminus dom(\sigma_F)$ (in Lemma 5.1) and $set_L \cap dom(\sigma_F)$ (in Lemma 5.3), it is also true for the union, obtaining the whole set_L. The condition $\{rnd_1\} = \{rnd_2\}$ affects only Lemma 5.3. □

Lemma 5.3 (Low variables affected by random value)**.**

$$\begin{aligned}\forall C, s_1, s_2. \quad & (s_1 =_L s_2) \wedge ([\![C]\!]^{CM_{rnd}}_{ext} s_1 \Downarrow s_1', [\![C]\!]^{CM_{rnd}}_{ext} s_2 \Downarrow s_2') \\ & \implies s_1'(set_L \cap dom(\sigma_F)) = s_2'(set_L \cap dom(\sigma_F))\end{aligned}$$

The resulting states of $s_1, s_2 \mid s_1 =_L s_2$ are equivalent in $set_L \cap dom(\sigma_F)$ (that is, the low variables containing high information) after the application of the *random* countermeasure if the random values assigned are equivalent (modeled as the set of values rnd_i).

Proof. All the variables in $set_L \cap dom(\sigma_F)$ are affected by the *random* countermeasure by assigning them a random value that can be modeled as a set of values rnd. It directly follows that, in this case, $s_1'(set_L \cap dom(\sigma_F)) = s_2'(set_L \cap dom(\sigma_F))$. □

5.2.3 Limit the Bandwidth (CM_{BW})

Theorem 5.2.3 (Limit the bandwidth)**.**

$$\begin{aligned}\forall C, s_1, s_2. \quad & (s_1 =_L s_2) \wedge ([\![C]\!] s_1 \Downarrow s_1', [\![C]\!] s_2 \Downarrow s_2') \wedge \forall v \in set_{CM} \\ & f(s_1'(v)) = f(s_2'(v)) \implies [\![C]\!]^{CM_{BW}}_{ext} s_1 =_L [\![C]\!]^{CM_{BW}}_{ext} s_2\end{aligned}$$

For all program C and the configuration (set_L, set_H), any execution under the extended semantics and applying the *limit the bandwidth* countermeasure ensures a property similar to the $WHAT$ with a set of low variables in $EDecl$. Such low variables are the ones affected by high variables, that is, $set_{CM} = varset(set_L \cap dom(\sigma_F))$.

Proof. The theorem follows directly from Lemmas 5.1 and 5.4. Since the equality stands for the subset $set_L \setminus dom(\sigma_F)$ (in Lemma 5.1) and $set_L \cap dom(\sigma_F)$ (in Lemma 5.4), it is also true for the union, obtaining the whole set_L. □

Lemma 5.4 (Low variables affected by Limit the bandwidth)**.**

$$\begin{aligned}\forall C, s_1, s_2. \quad & (s_1 =_L s_2) \wedge ([\![C]\!]^{CM_{BW}}_{ext} s_1 \Downarrow s_1', [\![C]\!]^{CM_{BW}}_{ext} s_2 \Downarrow s_2') \wedge \forall v \in set_{CM} \\ & f(s_1'(v)) = f(s_2'(v)) \implies s_1'(set_L \cap dom(\sigma_F)) = s_2'(set_L \cap dom(\sigma_F))\end{aligned}$$

The resulting states of $s_1, s_2 \mid s_1 =_L s_2$ are equivalent in $set_L \cap dom(\sigma_F)$ after the application of the *limit the bandwidth* countermeasure if after applying the specified functions the resulting values of the variables in set_{CM} are equivalent.

Proof. All the variables in $set_L \cap dom(\sigma_F)$ are affected by the *limit the bandwidth* countermeasure by applying a certain function to their value. It directly follows that, if applying the function to the variable into both memories returns the same value, $s'_1(set_L \cap dom(\sigma_F)) = s'_2(set_L \cap dom(\sigma_F))$. □

5.2.4 Limit the Leak Along the Time (CM_{t_0}) and Monitor the Leak (CM_{track})

Theorem 5.2.4 (Limit the leak along the time *and* Monitor the leak)**.**

$$\forall s_1, s_2 \in S.\ (s_1 =_L s_2) \wedge ([\![C]\!]s_1 \Downarrow, [\![C]\!]s_2 \Downarrow)$$
$$\Rightarrow [\![C]\!]^{CM} s_1(set_L \setminus set_{CM}) = [\![C]\!]^{CM} s_2(set_L \setminus set_{CM})$$

For all program C and the configuration (set_L, set_H), any execution under the extended semantics and applying the *monitor the leak* countermeasure ensures a weaker property considering only the variables which do not contain high information. Such low variables are the ones that ensure noninterference, that is, $set_{CM} = set_L \setminus dom(\sigma_F))$.

Proof. The theorem follows directly from the Lemma 5.1. The other variables are affected by the leaked high variables and hence not considered in the property. □

5.3 Conclusion

This section presents the specification of a policy considering the countermeasures. By formally defining the countermeasures, it has been possible to define the properties ensured by a program with noninterference and the different countermeasures (one at a time). The *assign a default value* makes a program ensure the noninterference property, whilst the *assign random value* adds a constraint to the property over the random values. The other countermeasures ensure weaker properties, depending on how they affect the semantics. Hence countermeasures, when mixed with a certain security property, make the execution to ensure the same, or another (weaker), security property.

It is worth noting that countermeasures are applied whenever a leak takes place. Hence, if a program is not leaking information then no countermeasures have to be applied. That is, countermeasures modify the property only if the declared property in the policy is not ensured (and hence countermeasures take place).

It is possible to combine the different countermeasures on a certain policy. However, depending on the used countermeasures, the properties defining a program become more complex. Considering more complex information flows is another challenge entailing more complex properties. Due to the complexity of the analysis, only the main points have been presented. Namely the variables depending on different *high* variables after the execution or the property assigned to both countermeasures and the information flow properties.

The proposed formalization of the policy considers the information in two different classes or sets, *high* and *low*. Other works manage several different

levels [MS04, CM04, TCBC11], for example, *top secret*, *secret* and *public*. Each level can have different information flows assigned, in this case, *top secret* cannot flow into the *public*, but *secret* can flow under certain conditions. For our proposal, the different treatments on the secret information is done assigning different countermeasures. This way, a variable considered *top secret* will have a more restraining countermeasure than a variable considered *secret*.

The work of G. Le Guernic *et al.* [LGBJS06] can be expressed in the terms of the present policy. Their work defines the noninterference property with the *default* countermeasure. One difference between this formalization and their development is that eventually they use the countermeasure *replace by skip*. This is due to the consideration of the function $output(var)$, and in the present work this situation does not apply.

The proposition of the present section is to specify different countermeasures for different variables. Another possibility would be to specify different countermeasures for a variable for each program execution. An example of this idea would be to allow the leak of a *secret* with $parity(secret)$ only a certain amount of times, and then apply the countermeasure *assign default value*. It is out of the scope of the present work to analyze how to develop this proposal and what the implications are on the resulting properties.

6 Conclusion and Perspectives

A security property can define if a program can be securely executed or not. However, it is necessary to specify what to do in the event a program does not ensure the property. Hence the need for countermeasures; so programs not ensuring the security property can be executed without risking the private information. If countermeasures are directly implemented at low level, it is not possible to choose between the different countermeasures. Consequently, adding countermeasures at a higher level increases the expressiveness of the policy. To our knowledge, considering the countermeasures into the policy definition is a new approach which improves the security policies expressiveness.

This work therefore develops the idea of adding the countermeasures in the policy and tries to enumerate the different possibilities available. Since this approach is new, there are not existing examples about it. However, as it is presented in subsection 5.3, other works can be expressed in the terms of our policy, giving a general idea of its expressiveness.

This work presents a new perspective for defining more expressive security policies by considering countermeasures. Considering countermeasures at the definition of the policy allows the choice between different options. It allows the execution of programs that do not ensure a certain security property, by modifying their semantics according to the policy.

This work also presents an extension for the semantics formally defined, which track how information flows. The countermeasures to apply are also formally defined. This allows one to prove which semantic properties are ensured by a policy considering noninterference with a countermeasure.

Moreover, this work presents an environment for tracking the information flow through the program semantics. However, the way to track the information flow is not unique. Other ways of tracking the information flow would lead into other ways to treat the leaks. The presented environment tracks, for each variable, which high variables affected its value. It is possible to imagine an environment for tracking the other way around, tracking for each high variable which variables have been affected by its value. In this case the semantics for the countermeasures would be defined differently.

As a future perspective, it would be desirable to test the soundness and validity of the presented semantics (including the one regarding declassification) and evaluate the resulting properties of different policies. The proposal of the present work has been formally defined taking into account this perspective.

Furthermore, it can be helpful to map the proposed principles over the different defined countermeasures. The definition of new countermeasures should take into account such principles, while maybe some others can be proposed.

Additionally other countermeasures could be defined, on one hand by defining new ones but on the other hand by combining the existing ones. Such combinations of countermeasures results in other semantic properties.

It is also possible to define other properties while declassifying information. While this work is concerned with the dimensions *what* and *where* and formalizes them in a certain way, it is possible to define their properties differently. Additionally, declassification dimensions can be combined, leading to other different properties entailing more complex permitted information flows.

Bibliography

[Aga00] Johan Agat. Transforming out timing leaks. In *Proceedings of the 27th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '00, pages 40–53, New York, NY, USA, 2000. ACM.

[AS07] Aslan Askarov and Andrei Sabelfeld. Localized delimited release: combining the what and where dimensions of information release. In *PLAS*, pages 53–60, 2007.

[CC08] S. Cavadini and D. Cheda. Run-time information flow monitoring based on dynamic dependence graphs. In *Availability, Reliability and Security, 2008. ARES 08. Third International Conference on*, pages 586 – 591, march 2008.

[CM04] S. Chong and A. C. Myers. Security policies for downgrading. In *Proceedings of the 11th ACM conference on Computer and communications security*, CCS '04, pages 198–209, New York, NY, USA, 2004. ACM.

[Den76] D.E. Denning. A lattice model of secure information flow. *Communications of the ACM*, 19(5):236 – 243, 1976.

[GJ05] G. Le Guernic and T. Jensen. Monitoring information flow. In Andrei Sabelfeld, editor, *Proceedings of the Workshop on Foundations of Computer Security*, pages 19–30. DePaul University, June 2005.

[GM82] Joseph A. Goguen and José Meseguer. Security policies and security models. In *IEEE Symposium on Security and Privacy*, pages 11–20, 1982.

[JJLS10] D. Jang, R. Jhala, S. Lerner, and H. Shacham. An empirical study of privacy-violating information flows in javascript web applications. In *Proc. of the 17th ACM conference on Computer and communications security*, 2010.

[LGBJS06] G. Le Guernic, A. Banerjee, T. Jensen, and D. A. Schmidt. Automata-based confidentiality monitoring. In *ASIAN'06: the 11th Asian Computing Science Conference on Secure Software*, 2006.

[LM09] Alexander Lux and Heiko Mantel. Formal aspects in security and trust. chapter Who Can Declassify?, pages 35–49. Springer-Verlag, Berlin, Heidelberg, 2009.

[MR07] Heiko Mantel and Alexander Reinhard. Controlling the what and where of declassification in language-based security. In *Proceedings of the 16th European conference on Programming*, ESOP'07, pages 141–156, Berlin, Heidelberg, 2007. Springer-Verlag.

[MS04] Heiko Mantel and David Sands. Controlled declassification based on intransitive noninterference. In Wei-Ngan Chin, editor, *APLAS*, volume 3302 of *Lecture Notes in Computer Science*, pages 129–145. Springer, 2004.

[MSZ04] Andrew C. Myers, Andrei Sabelfeld, and Steve Zdancewic. Enforcing robust declassification, 2004.

[MZZ+04] A. C. Myers, L. Zheng, S. Zdancewic, S. Chong, and N. Nystrom. Jif: Java information flow. Software release. Located at http://www.cs.cornell.edu/jif, 2001-2004.

[NN92] Hanne Riis Nielson and Flemming Nielson. *Semantics with applications - a formal introduction*. Wiley professional computing. Wiley, 1992.

[Ros99] A. W. Roscoe. What is intransitive noninterference. In *Proc. of the 12th IEEE Computer Security Foundations Workshop*, pages 228–238, 1999.

[SM03] A. Sabelfeld and A.C. Myers. Language-based information-flow security. *Selected Areas in Communications, IEEE Journal on*, 21(1):5 – 19, jan 2003.

[SM04] A. Sabelfeld and A. C. Myers. A model for delimited information release. In *Proc. International Symp. on Software Security (ISSS'03), volume 3233 of LNCS*, pages 174 – 191. Springer-Verlag, 2004.

[SS98] Andrei Sabelfeld and David Sands. A per model of secure information flow in sequential programs. *Higher-Order and Symbolic Computation*, 14:40–58, 1998.

[SS05] A. Sabelfeld and D. Sands. Dimensions and principles of declassification. In *Computer Security Foundations, 2005. CSFW-18 2005. 18th IEEE Workshop*, pages 255 – 269, june 2005.

[TCBC11] J.A. Thomas, N. Cuppens-Boulahia, and F. Cuppens. Declassification policy management in dynamic information systems. In *Availability, Reliability and Security (ARES), 2011 Sixth International Conference on*, pages 143 – 152, aug. 2011.

[VSI96] D. Volpano, G. Smith, and C. Irvine. A sound type system for secure flow analysis, 1996.

[ZAB07] S. Zander, G. Armitage, and P. Branch. A survey of covert channels and countermeasures in computer network protocols. *Communications Surveys Tutorials, IEEE*, 9(3):44 – 57, quarter 2007.

[Zda04] S. Zdancewic. Challenges for information-flow security. In *Proc. Programming Language Interference and Dependence (PLID)*, 2004.

[ZM01] Steve Zdancewic and Andrew C. Myers. Robust declassification. In *in Proc. IEEE Computer Security Foundations Workshop*, pages 15–23. IEEE Computer Society Press, 2001.

Appendices

A Modifications Over the Semantics

In order to facilitate the reading of this report, the modifications on the semantics for each declassification dimension are presented in this section of the Appendices. Below the reader can find the modified functions of both assignments and conditionals for each dimension. The other functions remain with the same semantics as presented in Section 3. The loops are not accepted to declassify information because of the possible leaks through termination and timing channels [SM03]. The reader can also find some remarks about the dimension *When*.

A.1 Dimension What

This dimension accepts the declassification of certain expressions exp contained in $EDecl$. The semantics take into account the expression of both assignments or guards in the conditionals and check if they are contained in $EDecl$. If they are not, the statement is executed and the environment σ_F is updated.

Figures 6 and 7 present the rules replacing the assignments and conditionals on the extended semantics tracking noninterference. It is considered that the whole expression of exp or B has to be defined in $EDecl$ in order to accept the declassification.

A.2 Dimension Where

This dimension accepts the declassification of certain variables in a certain part of the code. In other parts of the code, declassification of the variable is not accepted by the policy.

Figures 8 and 9 present the rules replacing the assignments and conditionals on the extended semantics tracking noninterference. Such rules consider the interfering transitions whenever the information declassification is accepted by the policy. The procedure is similar to the extended semantics tracking noninterference, but considering the policy forces to distinguish among several possibilities. The composition rules for the interfering transitions are also specified.

In the semantics the following elements are used. To specify the variables that can be released, $WDecl$ contains a variable (belonging to set_H) and the code line where it can be declassified. $i(var)$ returns $true$ or $false$ depending on the code line of the statement to be executed and the code line defined by the policy. If $i(var)$ returns $true$ then the declassification can be done (signaling the transition with $\longrightarrow_I$), and if $i(var)$ returns $false$ then the transition is not signaled and the environment σ_F is updated. Additionally, declassification can only be done if the context of execution ω is all formed by $\perp$ (that is, $vars(\omega) = \emptyset$). If the context of execution is different ($vars(\omega) \neq \emptyset$) then declassification is not accepted by the policy. For the conditionals under $vars(\omega) \neq \emptyset$ there are two possibilities, depending on the variables in the guard, hence modifying or not the context of execution ω.

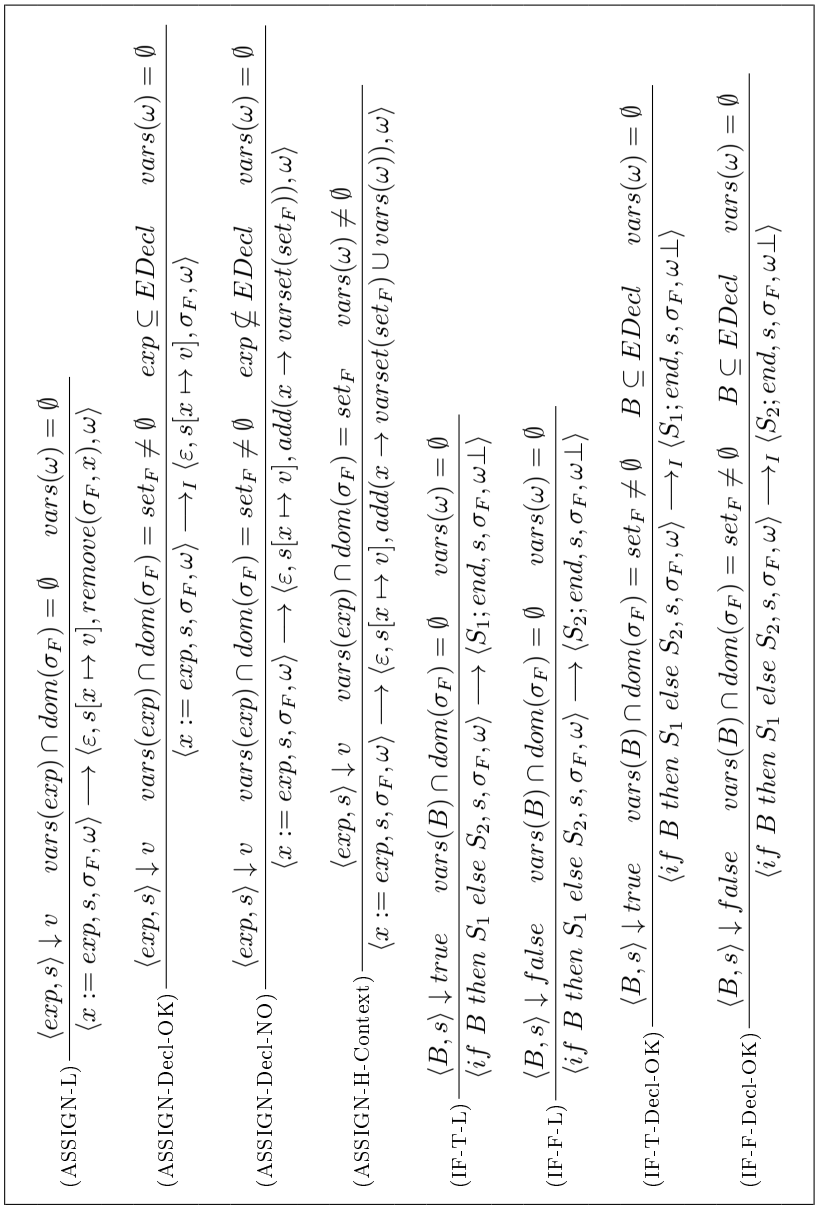
A.3 Dimension When

This dimension (more specifically, case *relative* of the dimension *When*) states that the variables can be released under certain conditions. Each variable has some stated conditions that can be updated during the execution of the program. If the variable flows into the low variables and the assigned condition is true, the declassification is accepted by the policy.

We understand this type of declassification as a generalization of the dimension *Where*, where not only the code line is checked but some other conditions. Some possible conditions can be related to the updating of variables (declassifying a variable if other variable has been modified), to the timing (after a certain amount of time), etc.

Considering this, dimension *When* can be defined by some regular transitions ($\longrightarrow$) and some interfering transitions ($\longrightarrow_I$, when the policy accepts the flow). Hence, the security property is equivalent to the *Where* one, yet adapting the semantics checking the stated conditions instead of the code lines.

With all this, we consider that the information flow defined by this property is similar to the one of the dimension *Where*. Hence it is not developed in this work.



$$\text{(ASSIGN-L)}\ \frac{\langle exp, s\rangle \downarrow v \quad vars(exp) \cap dom(\sigma_F) = \emptyset \quad vars(\omega) = \emptyset}{\langle x := exp, s, \sigma_F, \omega\rangle \longrightarrow \langle \varepsilon, s[x \mapsto v], remove(\sigma_F, x), \omega\rangle}$$

$$\text{(ASSIGN-Decl-OK)}\ \frac{\langle exp, s\rangle \downarrow v \quad vars(exp) \cap dom(\sigma_F) = set_F \neq \emptyset \quad exp \subseteq EDecl \quad vars(\omega) = \emptyset}{\langle x := exp, s, \sigma_F, \omega\rangle \longrightarrow_I \langle \varepsilon, s[x \mapsto v], \sigma_F, \omega\rangle}$$

$$\text{(ASSIGN-Decl-NO)}\ \frac{\langle exp, s\rangle \downarrow v \quad vars(exp) \cap dom(\sigma_F) = set_F \neq \emptyset \quad exp \nsubseteq EDecl \quad vars(\omega) = \emptyset}{\langle x := exp, s, \sigma_F, \omega\rangle \longrightarrow \langle \varepsilon, s[x \mapsto v], add(x \to varset(set_F)), \omega\rangle}$$

$$\text{(ASSIGN-H-Context)}\ \frac{\langle exp, s\rangle \downarrow v \quad vars(exp) \cap dom(\sigma_F) = set_F \quad vars(\omega) \neq \emptyset}{\langle x := exp, s, \sigma_F, \omega\rangle \longrightarrow \langle \varepsilon, s[x \mapsto v], add(x \to varset(set_F) \cup vars(\omega)), \omega\rangle}$$

$$\text{(IF-T-L)}\ \frac{\langle B, s\rangle \downarrow true \quad vars(B) \cap dom(\sigma_F) = \emptyset \quad vars(\omega) = \emptyset}{\langle if\ B\ then\ S_1\ else\ S_2, s, \sigma_F, \omega\rangle \longrightarrow \langle S_1; end, s, \sigma_F, \omega\perp\rangle}$$

$$\text{(IF-F-L)}\ \frac{\langle B, s\rangle \downarrow false \quad vars(B) \cap dom(\sigma_F) = \emptyset \quad vars(\omega) = \emptyset}{\langle if\ B\ then\ S_1\ else\ S_2, s, \sigma_F, \omega\rangle \longrightarrow \langle S_2; end, s, \sigma_F, \omega\perp\rangle}$$

$$\text{(IF-T-Decl-OK)}\ \frac{\langle B, s\rangle \downarrow true \quad vars(B) \cap dom(\sigma_F) = set_F \neq \emptyset \quad B \subseteq EDecl \quad vars(\omega) = \emptyset}{\langle if\ B\ then\ S_1\ else\ S_2, s, \sigma_F, \omega\rangle \longrightarrow_I \langle S_1; end, s, \sigma_F, \omega\perp\rangle}$$

$$\text{(IF-F-Decl-OK)}\ \frac{\langle B, s\rangle \downarrow false \quad vars(B) \cap dom(\sigma_F) = set_F \neq \emptyset \quad B \subseteq EDecl \quad vars(\omega) = \emptyset}{\langle if\ B\ then\ S_1\ else\ S_2, s, \sigma_F, \omega\rangle \longrightarrow_I \langle S_2; end, s, \sigma_F, \omega\perp\rangle}$$

Figure 6: Extended small-step semantics for the dimension *What* (A)

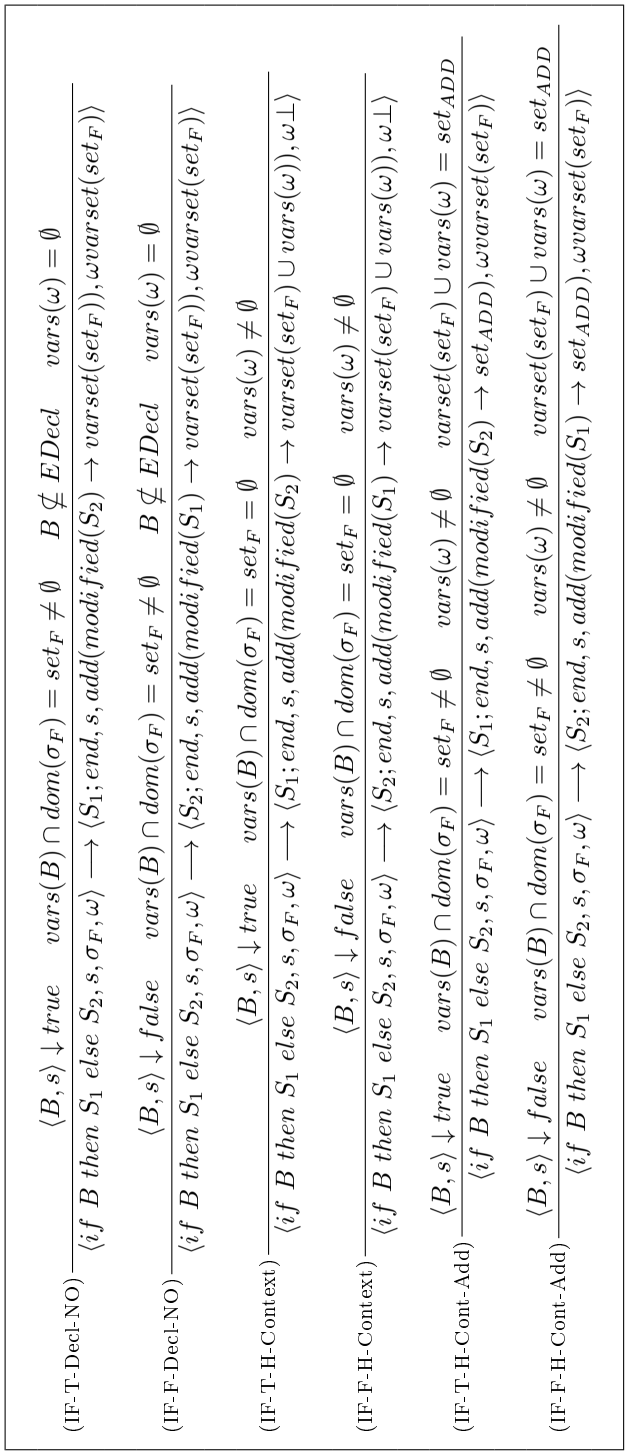


Figure 7: Extended small-step semantics for the dimension *What* (B)

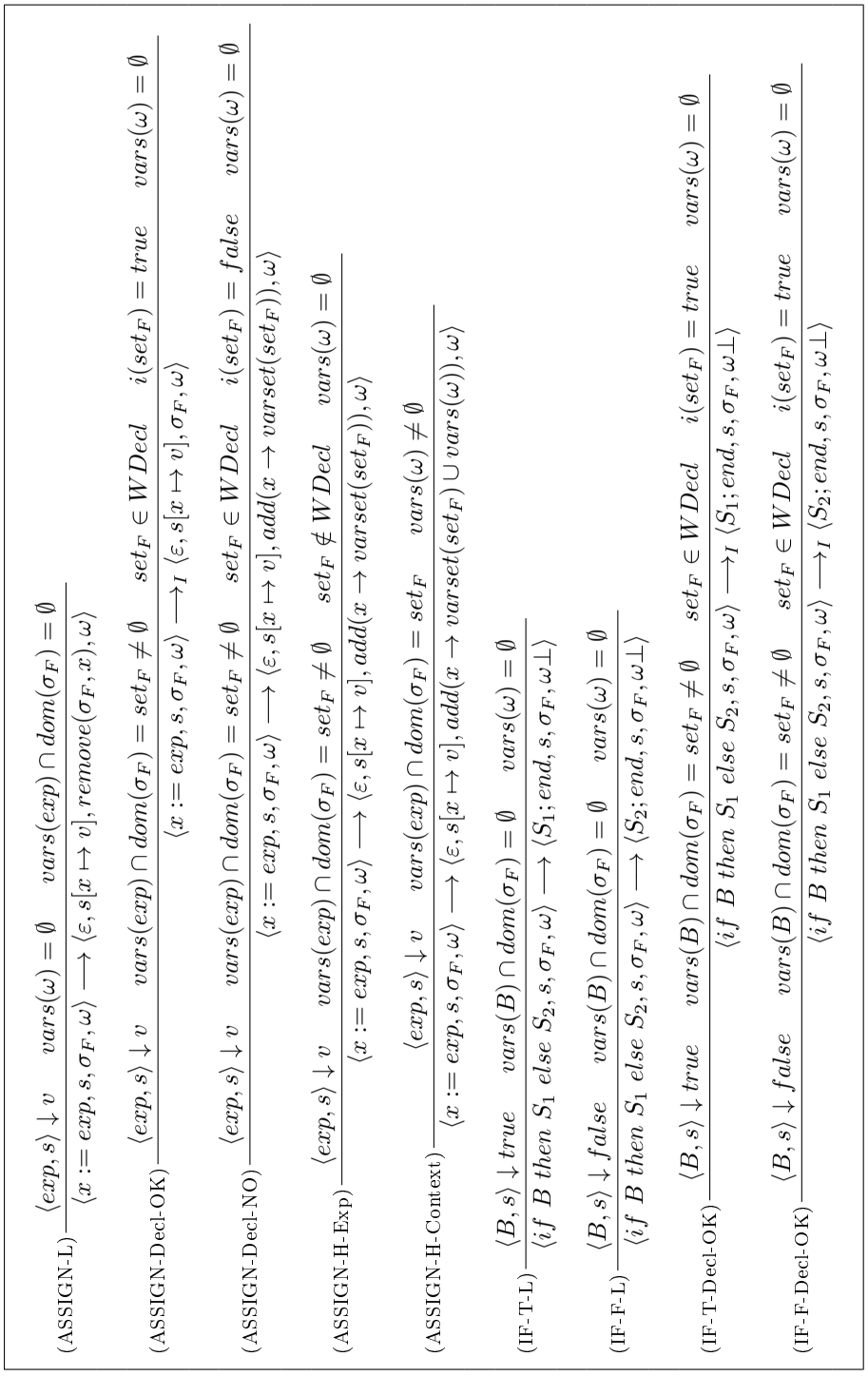


Figure 8: Extended small-step semantics for the dimension *Where* (A)

$$\text{(IF-T-Decl-NO)}\ \frac{\langle B, s\rangle \downarrow true \quad vars(B) \cap dom(\sigma_F) = set_F \neq \emptyset \quad set_F \in WDecl \quad i(set_F) = false \quad vars(\omega) = \emptyset}{\langle if\ B\ then\ S_1\ else\ S_2, s, \sigma_F, \omega\rangle \longrightarrow \langle S_1; end, s, add(modified(S_2) \rightarrow varset(set_F)), \omega varset(set_F)\rangle}$$

$$\text{(IF-F-Decl-NO)}\ \frac{\langle B, s\rangle \downarrow false \quad vars(B) \cap dom(\sigma_F) = set_F \neq \emptyset \quad set_F \in WDecl \quad i(set_F) = false \quad vars(\omega) = \emptyset}{\langle if\ B\ then\ S_1\ else\ S_2, s, \sigma_F, \omega\rangle \longrightarrow \langle S_2; end, s, add(modified(S_1) \rightarrow varset(set_F)), \omega varset(set_F)\rangle}$$

$$\text{(IF-T-H-Exp)}\ \frac{\langle B, s\rangle \downarrow true \quad vars(B) \cap dom(\sigma_F) = set_F \neq \emptyset \quad set_F \notin WDecl \quad vars(\omega) = \emptyset}{\langle if\ B\ then\ S_1\ else\ S_2, s, \sigma_F, \omega\rangle \longrightarrow \langle S_1; end, s, add(modified(S_2) \rightarrow varset(set_F)), \omega varset(set_F)\rangle}$$

$$\text{(IF-F-H-Exp)}\ \frac{\langle B, s\rangle \downarrow false \quad vars(B) \cap dom(\sigma_F) = set_F \neq \emptyset \quad set_F \notin WDecl \quad vars(\omega) = \emptyset}{\langle if\ B\ then\ S_1\ else\ S_2, s, \sigma_F, \omega\rangle \longrightarrow \langle S_2; end, s, add(modified(S_1) \rightarrow varset(set_F)), \omega varset(set_F)\rangle}$$

$$\text{(IF-T-H-Context)}\ \frac{\langle B, s\rangle \downarrow true \quad vars(B) \cap dom(\sigma_F) = set_F = \emptyset \quad vars(\omega) \neq \emptyset}{\langle if\ B\ then\ S_1\ else\ S_2, s, \sigma_F, \omega\rangle \longrightarrow \langle S_1; end, s, add(modified(S_2) \rightarrow varset(set_F) \cup vars(\omega)), \omega\bot\rangle}$$

$$\text{(IF-F-H-Context)}\ \frac{\langle B, s\rangle \downarrow false \quad vars(B) \cap dom(\sigma_F) = set_F = \emptyset \quad vars(\omega) \neq \emptyset}{\langle if\ B\ then\ S_1\ else\ S_2, s, \sigma_F, \omega\rangle \longrightarrow \langle S_2; end, s, add(modified(S_1) \rightarrow varset(set_F) \cup vars(\omega)), \omega\bot\rangle}$$

$$\text{(IF-T-H-Cont-Add)}\ \frac{\langle B, s\rangle \downarrow true \quad vars(B) \cap dom(\sigma_F) = set_F \neq \emptyset \quad vars(\omega) \neq \emptyset \quad varset(set_F) \cup vars(\omega) = set_{add}}{\langle if\ B\ then\ S_1\ else\ S_2, s, \sigma_F, \omega\rangle \longrightarrow \langle S_1; end, s, add(modified(S_2) \rightarrow set_{add}), \omega varset(set_F)\rangle}$$

$$\text{(IF-F-H-Cont-Add)}\ \frac{\langle B, s\rangle \downarrow false \quad vars(B) \cap dom(\sigma_F) = set_F \neq \emptyset \quad vars(\omega) \neq \emptyset \quad varset(set_F) \cup vars(\omega) = set_{add}}{\langle if\ B\ then\ S_1\ else\ S_2, s, \sigma_F, \omega\rangle \longrightarrow \langle S_2; end, s, add(modified(S_1) \rightarrow set_{add}), \omega varset(set_F)\rangle}$$

$$\text{(COMP-C-Decl)}\ \frac{\langle C_1, s, set_F, \omega\rangle \longrightarrow_I \langle \varepsilon, s', set'_F, \omega'\rangle}{\langle C_1; C_2, s, \sigma_F, \omega\rangle \longrightarrow_I \langle C_2, s', \sigma'_F, \omega'\rangle}$$

$$\text{(COMP-P-Decl)}\ \frac{\langle C_1, s, set_F, \omega\rangle \longrightarrow_I \langle C'_1, s', set'_F, \omega'\rangle}{\langle C_1; C_2, s, \sigma_F, \omega\rangle \longrightarrow_I \langle C'_1; C_2, s', \sigma'_F, \omega'\rangle}$$

Figure 9: Extended small-step semantics for the dimension *Where* (B)

B Formal Transformation of the Semantics

For accepting declassification of information, it is necessary to make a distinction between regular transitions and interfering ones. Interfering transitions are the ones that allow a flow from high variables into low ones. For each dimension the semantics of such transitions are defined according to the policy. In short, this work makes a distinction between regular transitions (defined by $\longrightarrow$) and interfering (defined by $\longrightarrow_I$). Then a certain configuration $C = \langle c, s\rangle$ generates a certain trace trc with both transitions $\longrightarrow$ and $\longrightarrow_I$.

In order to simplify the resulting trace, let us define a transformation for this system of transitions. To the purpose of this work, a subprogram with no release of information can be considered as a sole transition. To formalize the transformation it is necessary to consider a trace trc_1 and certain configurations C_1, C_2.

- Basic transformations:
  1. $\mathcal{T}(C_1 \longrightarrow C_2 \longrightarrow trc_1) = \mathcal{T}(C_1 \longrightarrow trc_1)$
  2. $\mathcal{T}(C_1 \longrightarrow_I C_2 \longrightarrow trc_1) = C_1 \longrightarrow_I \mathcal{T}(C_2 \longrightarrow trc_1)$
  3. $\mathcal{T}(C_1 \longrightarrow C_2 \longrightarrow_I trc_1) = C_1 \longrightarrow C_2 \longrightarrow_I \mathcal{T}(trc_1)$
  4. $\mathcal{T}(C_1 \longrightarrow_I C_2 \longrightarrow_I trc_1) = C_1 \longrightarrow_I C_2 \longrightarrow_I \mathcal{T}(trc_1)$
- Ending traces:
  1. $\mathcal{T}(C_1) = C_1$
  2. $\mathcal{T}(C_1 \longrightarrow C_2) = C_1 \longrightarrow C_2$
  3. $\mathcal{T}(C_1 \longrightarrow_I C_2) = C_1 \longrightarrow_I C_2$

Figure 10: Transformation normalisation of the traces

The transformation intends to simplify any subtrace with no release of information ($\longrightarrow$) into a sole transition, keeping track of both initial and final configurations of the subtrace ($C_o \longrightarrow C_f$). Under this transformation, the resulting transitions can be a) equivalent to a whole noninterferent subprogram or b) the information release.

Let us define formally a trace before the transformation:
$T ::= C \longrightarrow T \mid C \longrightarrow_I T \mid C$
as it can be noted, there can be any number of transitions. After the transformation, instead, the trace keeps a certain structure.

Theorem B.1. *A trace after the transformation have the following structure:*
$T ::= T_a \mid T_b \quad T_a ::= C \longrightarrow T_b \quad T_b ::= C \longrightarrow_I T_a \mid C \longrightarrow_I T_b \mid C$

Proof (sketch). It is proved by recurrence for all the possible transitions. Assuming different possible combinations of traces, the resulting transformed traces ensure the structure. □

This indicates that after a regular transition ($\longrightarrow$) it is not possible to have another regular transition, while the interfering transitions ($\longrightarrow_I$) have no such

constraint.

To see how this transformation works, let us transform the trace below:
$M = C_1 \longrightarrow C_2 \longrightarrow C_3 \longrightarrow C_4 \longrightarrow_I C_5 \longrightarrow_I C_6 \longrightarrow C_7 \longrightarrow C_8 \longrightarrow_I C_9$

$$
\begin{aligned}
\mathcal{T}(M) &= \mathcal{T}(C_1 \longrightarrow C_2 \longrightarrow C_3 \longrightarrow_I C_4 \longrightarrow_I C_5 \longrightarrow C_6 \longrightarrow C_7 \longrightarrow_I C_8) \\
&= \mathcal{T}(C_1 \longrightarrow C_3 \longrightarrow_I C_4 \longrightarrow_I C_5 \longrightarrow C_6 \longrightarrow C_7 \longrightarrow_I C_8) \\
&= C_1 \longrightarrow C_3 \longrightarrow_I \mathcal{T}(C_4 \longrightarrow_I C_5 \longrightarrow C_6 \longrightarrow C_7 \longrightarrow_I C_8) \\
&= C_1 \longrightarrow C_3 \longrightarrow_I C_4 \longrightarrow_I \mathcal{T}(C_5 \longrightarrow C_6 \longrightarrow C_7 \longrightarrow_I C_8) \\
&= C_1 \longrightarrow C_3 \longrightarrow_I C_4 \longrightarrow_I \mathcal{T}(C_5 \longrightarrow C_6 \longrightarrow_I C_8) \\
&= C_1 \longrightarrow C_3 \longrightarrow_I C_4 \longrightarrow_I C_5 \longrightarrow C_7 \longrightarrow_I C_8
\end{aligned}
$$

As it can be seen, this transformation keeps record of both initial and final subset states of regular transitions ($\longrightarrow$) and the whole set of interfering transitions ($\longrightarrow_I$). This transformation of the transitions system will be considered while defining the security properties of the declassification dimensions. This way, the constraints among the properties take into account only the initial and final states of the noninterfering subset of transitions.