Who Can Declassify?

Alexander Lux and Heiko Mantel

Department of Computer Science, TU Darmstadt
Hochschulstraße 10, 64289 Darmstadt, Germany
{lux,mantel}@cs.tu-darmstadt.de

Abstract. Noninterference provides reliable guarantees for the confidentiality of sensitive information, but it is too restrictive if exceptions shall be permitted. Although many approaches to permitting and controlling exceptional information release have been proposed, the problem of declassification is not yet satisfactorily solved. The aim of our project is to provide adequate control for declassification in language-based security. The main contribution of this article is a novel approach for controlling who can initiate a declassification. Our contributions include a formal security condition and a sound approach to statically enforcing this condition. This article complements our earlier work on controlling where declassification can occur and what can be declassified.

1 Introduction

Before private data is given as input to an application, one would like a guarantee that the program is sufficiently trustworthy. The desired guarantee can be formalized by the noninterference property, which ensures that there is no danger of undesired information leakage. This is expressed by requiring that the program's output to untrusted sinks must be completely independent from any confidential input. While noninterference constitutes a reliable guarantee about the flow of information, it is a too restrictive requirement for some domains. For instance, the output of a password-based authentication mechanism differs for a given input depending on the stored password, i.e. on a secret. Therefore, such a mechanism necessarily cannot satisfy the noninterference property.

It is clear that the noninterference property can be relaxed in order to permit such exceptional information leakage. However, the problem of controlling such exceptions is not yet satisfactorily solved. To clarify the aims and virtues of previous approaches, three dimensions of controlling declassification were identified in [1], namely, *what* can be declassified, *where* declassification can occur, and *who* can initiate declassification. A recent classification of existing approaches to controlling declassification [2] shows that we do not yet have an integrated approach that provides adequate control in all of these dimensions.

In this article, we propose a novel approach to controlling the third dimension of declassification, i.e. who can initiate declassification. This work complements our earlier work on controlling the first two dimensions [3]. In addition, we

Appeared in P. Degano et al. (Ed.): Preproceedings of FAST 2008

present prudent principles of declassification that can be used as a sanity check for new security conditions. Our principles extend and refine the ones proposed in [2]. The second novel contribution is the security condition WHO that we integrate with our earlier condition WHERE to WHERE&WHO, in order to control who can initiate which declassifications. We prove that WHERE&WHO satisfies all prudent principles of declassification, the novel as well as the established ones. Interestingly, we could show that, in some cases, it is possible to refine the security policy such that WHERE&WHO can be enforced by the simpler condition WHERE, which we developed for controlling where declassification can occur. We also present an approach to statically enforcing WHERE&WHO by refining the policy and applying a type system for WHERE.

2 A Motivating Example

We consider a program that is used by a video store to control the delivery of movies to customers. After a customer decides to buy a movie, his payment data is fetched, and it is forwarded to a bank. The movie is delivered only after the payment has been confirmed by the bank. Movies can be ordered either via a web interface or at a vending machine in the store. Regular customers may become preferred customers, who may obtain a movie also without confirmation of their payment by the bank. However, this preferred treatment is limited to orders at the vending machine because the vendor does not have sufficient trust in the authentication mechanism of the web interface.

The example program is written in a simple imperative language with explicit I/O-instructions. Execution of an instruction x <- *in* sets the value of the variable x to a value read from the input channel *in*. Execution of x -> *out* writes the current value of x to the output channel *out*. As a convention, names of input and output channels end with I or O, respectively. Instructions in brackets (like, e.g., $[\mathsf{public{:=}movie}]_1$) mark assignments that are intended as declassifications. For now such declassification statements should be read as usual assignments.

```
if byMachine then                          % branch on whether purchase at machine
   paydatvd <- machineI;                   % get payment data from vending machine
   paydatvd -> bankO;                      % pass payment data to bank
   payOK <- bankI;                         % get confirmation of payment from bank
   if (payOK or isPreferred(paydatvd)) then
      [public:=movie]_1;                   % copy movie to public variable
      public -> machineO fi                % pass movie to machine
else
   paydatweb <- webI;                      % get payment data from web interface
   paydatweb -> bankO;                     % pass payment data to bank
   payOK <- bankI;                         % get confirmation of payment from bank
   if payOK then
      [public:=movie]_2;                   % copy movie to public variable
      public -> webO fi                    % pass movie to web interface
fi
```

For the store, it is essential that a movie is not leaked accidently. That is, a movie is a secret that must be protected from the customer until his credentials have been confirmed. As a movie is a secret, it must be explicitly declassified before it can be delivered to a customer. A preferred customer can initiate this declassification also without the bank's confirmation by declaring his special status. However, exceptions should be limited to purchases at the vending machine. It is the vendor's policy that a customer's input at the web interface cannot initiate a declassification. While our first example program satisfies this security requirement, the following program is vulnerable to attacks via the web interface. The problem with this program is that the check isPreferred(paydat) can depend on the input from the web interface, which violates the vendor's policy.

```
if byMachine
   then paydat <- machineI
   else paydat <- webI fi;
paydat -> bankO;                      % pass payment data to bank
payOK <- bankI;                       % get confirmation of payment from bank
if (payOK or isPreferred(paydat)) then
   [public:=movie]1;                  % copy movie to public variable
   if byMachine
      then public -> machineO
      else public -> webO fi
fi
```

The objective of this article is to develop a security condition that adequately controls who can initiate a declassification. In particular, it should reject vulnerable programs like our second example, and it should accept secure programs like the first example. The subscripts at declassification statements (e.g., 1 and 2 in the first example and 1 in the second example) will be used to specify in a policy which declassification statements may be initiated by whom.

3 Adequate Control of Declassification

We aim for security conditions that formalize the intuitive notion of secure information flow on a semantic level and that are suitable points of reference for a soundness argument of a given syntactic security analysis. However, defining a security condition that adequately captures the security of information flow becomes non-trivial if exceptional information release shall be permitted. There is an inherent trade-off between relaxing information flow control in order to permit declassification and reliably ensuring security by rigorous information flow control. In the following, we present prudent principles of declassification that can be used as a sanity check for security conditions. The principles extend and refine the ones proposed by Sabelfeld and Sands [2]. The principles are presented in Section 3.1. In Section 3.2, we introduce the model of computation and the programming language used in the rest of the article. The prudent principles are formalized and specialized to this setting in Section 3.3.

3.1 Prudent Principles of Controlling Declassification

In the following, we use the term *noninterference* as a place-holder for a security condition that adequately characterizes information flow security in a setting without declassification. In order to apply the principles as a sanity check, this place-holder must be instantiated with a suitable security condition.

Semantic consistency [2] The (in)security of a program is invariant under semantic-preserving transformations of declassification-free subprograms.

Whether a program is secure depends on its behavior. Semantic consistency ensures that the classification of a program is not effected by syntactic modifications that do not change the program's behavior. This principle is desirable for security definitions, in general, including ones that permit declassification.

Relaxation Every program that satisfies noninterference also satisfies the given security condition.

Monotonicity of release [2] Adding further declassifications to a secure program cannot render it insecure.

These principle are reasonable, because the whole purpose of introducing declassification is to accept more programs as secure. The principles *relaxation* and *monotonicity* impose a lower bound on the set of programs that are accepted by security conditions. This distinguishes them from the principles below, which impose upper bounds on the set of acceptable programs.

Non-occlusion [2] The presence of a declassification operation cannot mask other covert information leaks.

Non-occlusion is crucial, because it summarizes the goal of controlling declassification. However, a bootstrapping problem occurs when formalizing this principle because such a formalization itself would constitute a characterization of secure information flow, which would need to be checked for non-occlusion. We introduce further prudent principles that can be formalized in the following.

Noninterference up-to Every program that satisfies the given security condition also satisfies noninterference if it were executed in an environment that terminates the program when it is about to perform a declassification.

Persistence For every program that satisfies the given security condition, all programs that are reachable also satisfy the security condition. If this only holds for programs that are reached by an execution where the last step is a declassification, then the given security condition is called *weakly persistent*.

The principle *noninterference up-to* ensures that the security condition is not more permissible than noninterference as long as no declassification occurs. Persistence and weak persistence, both ensure, after a declassification occurred, that one again obtains the original security guarantee for the resulting configuration.

The fourth principle introduced in [2], *conservativity*, is subsumed by *noninterference up-to* and *relaxation*. *Conservativity* requires that a security condition must be equivalent to noninterference for programs without declassification. One direction of the equivalence is implied by *relaxation* and the other by *noninterference up-to*. Note, however, that *noninterference up-to* also establishes guarantees for programs with declassification while conservativity does not.

While the previous principles provide a check of adequacy for security conditions with declassification, in general, the following principle is especially intended to check the adequacy of the control of *who* can initiate declassification.

Protection A security property complies with *protection*, if for all programs satisfying this property, an attacker from whom declassification should be protected, cannot effect declassification by his behavior.

3.2 Policies, Programs, and a Definition of Noninterference

We capture the intended security guarantees by flow policies:

Definition 1. *An* MLS policy with exceptions *is a triple* $(\mathcal{D}, \leq, \rightsquigarrow)$*, where* $\mathcal{D}$ *is a finite set of security domains,* $\leq \subseteq \mathcal{D}\times\mathcal{D}$ *is a partial order and* $\rightsquigarrow \subseteq \mathcal{D}\times\mathcal{D}$*.*

The relation $\leq$ determines, between which domains information may flow normally. The relation $\rightsquigarrow$ determines, between which domains information may flow exceptionally, i.e. by declassification. An example is the two-level flow policy $(\{low, high\}, \{(low, high), (low, low), (high, high)\}, \rightsquigarrow)$, which permits information flow from *low* to *high* but not from *high* to *low*. Declassification from *high* to *low* is permitted or not depending on whether $high \rightsquigarrow low$ or $high \not\rightsquigarrow low$.

We assume a set of programs *Com*, a set of variables *Var* and a set of values *Val*. A *memory* assigns values to variables $s : Var \to Val$. A *domain assignment* is a function $dom : Var \to \mathcal{D}$. It establishes a connection between memories and a flow policy by assigning a domain to each variable. We say that *an observer has a security domain* D if he can see the values of all variables x with $dom(x) \leq D$, but not of other variables. Hence, a D-observer can distinguish memories, if and only if they differ in the value of at least one variable x with $dom(x) \leq D$.

Definition 2. *For a given domain* $D \in \mathcal{D}$*, two memories* s *and* s' *are* D-equal*, denoted by* $s =_D s'$*, if* $\forall x \in Var.\ (dom(x) \leq D \Rightarrow s(x) = s'(x))$*.*

We define the set of *configurations Conf* as the set of all pairs of a program C (or of the special symbol ϵ) and a memory s, denoted by $\langle C, s\rangle$ or $\langle \epsilon, s\rangle$, respectively. The operational semantics are given by a deterministic step relation $\twoheadrightarrow$ between configurations. We partition $\twoheadrightarrow$ into disjoint sub-relations $\twoheadrightarrow_k^{D_1\to D_2}$ and $\twoheadrightarrow_{\mathrm{O}}$ where $k \in \mathbb{N}$ and $D_1, D_2 \in \mathcal{D}$. A $\twoheadrightarrow_k^{D_1\to D_2}$-step models the execution of a declassification instruction with label k, source domain D_1, and destination domain D_2. We call such steps *declassification steps* and $\twoheadrightarrow_{\mathrm{O}}$-steps *ordinary steps*.

In the following, we assume a flow policy $(\mathcal{D}, \leq, \rightsquigarrow)$ and domain assignment *dom*. As notational convention we denote elements of $\mathcal{D}$ by D, of *Com* by C, of *Var* by x and y, of *Val* by v, of memories by s and t, of *Conf* by *cnf*, and of instruction labels in $\mathbb{N}$ by k, all possibly with indices or primes.

In Sect. 3.1, we used the term *noninterference* as a place-holder for a security condition that characterizes the absence of unintended information flow in a setting without declassification. We instantiate this place-holder with the *strong*

security condition, which was originally introduced in [4] for multi-threaded programs. This is an established definition of security for which there already exist variants that permit and control declassification [1, 3]. Strong security is based on the PER-approach [5], i.e. information flow security is characterized based on non-reflexive indistinguishability relations on programs. Two programs are indistinguishable for a D-observer, if they do not reveal information to D, when started in D-equal memories. As strong security does not permit declassification, the relation $\rightsquigarrow$ does not occur in the following definition.

Definition 3 (Strong Security for Sequential Programs). *A* strong D-bisimulation *is a symmetric relation R on programs that satisfies*

$$\forall C_1, C_1'.\ \forall s, s', t.\ \forall C_2.\ \left[\begin{array}{r} (C_1\ R\ C_1' \wedge \langle\!\langle C_1, s\rangle\!\rangle \twoheadrightarrow \langle\!\langle C_2, t\rangle\!\rangle \wedge s =_D s') \\ \Rightarrow \exists C_2', t' : (C_2\ R\ C_2' \wedge \langle\!\langle C_1', s'\rangle\!\rangle \twoheadrightarrow \langle\!\langle C_2', t'\rangle\!\rangle \wedge t =_D t') \end{array}\right]$$

The relation $\approxeq_D$ is defined as the union of all strong D-bisimulations. A program C is strongly secure *if $C \approxeq_D C$ holds for all $D \in \mathcal{D}$.*

For two programs being strongly D-bisimilar means that individual computation steps from D-equal memories can be simulated, such that the resulting memories also are D-equal and the resulting programs also are strongly D-bisimilar.

We instantiate programs and the operational semantics with a simple while language (WL), augmented with a declassifying assignment. The set *Com* is defined by the following grammar.

$$C ::= \mathsf{skip} \mid x{:=}Exp \mid C_1; C_2 \mid \mathsf{if}\ B\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2\ \mathsf{fi} \mid \mathsf{while}\ B\ \mathsf{do}\ C\ \mathsf{od} \mid [x{:=}y]_k$$

As further condition we require that no two declassification assignments with the same instruction label may appear in a given program. That is, an instruction label uniquely determines the occurrence of a declassification in the program code. To denote expressions from a language $\mathcal{E}$ we use B or *Exp*. That expression *Exp* evaluates to value v in memory s is denoted by $\langle\!\langle Exp, s\rangle\!\rangle \downarrow v$. Here, we do not fully define the language $\mathcal{E}$, but only assume that the evaluation of expressions is total, atomic, and unambiguous. Moreover, we assume a function $vars : \mathcal{E} \to \mathfrak{P}(Var)$ such that

$$\forall Exp, s, s'.\ [(\forall x \in vars(Exp).\ s(x) = s'(x)) \Rightarrow \forall v.\ (\langle\!\langle Exp, s\rangle\!\rangle \downarrow v \Rightarrow \langle\!\langle Exp, s'\rangle\!\rangle \downarrow v)]$$

For instance, $vars(Exp)$ could be the set of variables appearing in *Exp*.

The instantiation of the step relations is given by the rules in Fig. 1. Most rules are standard. Exceptions are the rules for declassifying assignments $[x{:=}y]_k$, which result in $\twoheadrightarrow_k^{D_1 \to D_2}$ steps, if the domains are not $\leq$-related. Declassifying assignments with $\leq$-related domains result in ordinary steps, because the direct information flow conducted by such instructions intuitively complies with $\leq$.

For simplicity, we require the right-hand side of declassifying assignments to be a variable. The language WL will be extended with statements for input and output in Sect. 4.1.

$$\frac{}{\langle \mathsf{skip}, s\rangle \rightarrow_{\mathrm{O}} \langle \epsilon, s\rangle} \qquad \frac{\langle Exp, s\rangle \downarrow v}{\langle x{:=}Exp, s\rangle \rightarrow_{\mathrm{O}} \langle \epsilon, [x=v]s\rangle}$$

$$\frac{\langle B, s\rangle \downarrow v \quad v \neq 0}{\langle \mathsf{if}\ B\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2\ \mathsf{fi}, s\rangle \rightarrow_{\mathrm{O}} \langle C_1, s\rangle} \qquad \frac{\langle B, s\rangle \downarrow 0}{\langle \mathsf{if}\ B\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2\ \mathsf{fi}, s\rangle \rightarrow_{\mathrm{O}} \langle C_2, s\rangle}$$

$$\frac{\langle B, s\rangle \downarrow v \quad v \neq 0}{\langle \mathsf{while}\ B\ \mathsf{do}\ C\ \mathsf{od}, s\rangle \rightarrow_{\mathrm{O}} \langle C; \mathsf{while}\ B\ \mathsf{do}\ C\ \mathsf{od}, s\rangle} \qquad \frac{\langle B, s\rangle \downarrow 0}{\langle \mathsf{while}\ B\ \mathsf{do}\ C\ \mathsf{od}, s\rangle \rightarrow_{\mathrm{O}} \langle \epsilon, s\rangle}$$

$$\frac{D_1 = dom(y) \quad D_2 = dom(x) \quad D_1 \not\leq D_2}{\langle [x{:=}y]_k, s\rangle \rightarrow_k^{D_1 \rightarrow D_2} \langle \epsilon, [x = s(y)]s\rangle} \qquad \frac{dom(y) \leq dom(x)}{\langle [x{:=}y]_k, s\rangle \rightarrow_{\mathrm{O}} \langle \epsilon, [x = s(y)]s\rangle}$$

$$\frac{\langle C_1, s\rangle \rightarrow_{\mathrm{O}} \langle \epsilon, s'\rangle}{\langle C_1; C_2, s\rangle \rightarrow_{\mathrm{O}} \langle C_2, s'\rangle} \qquad \frac{\langle C_1, s\rangle \rightarrow_{\mathrm{O}} \langle C_1', s'\rangle}{\langle C_1; C_2, s\rangle \rightarrow_{\mathrm{O}} \langle C_1'; C_2, s'\rangle}$$

$$\frac{\langle C_1, s\rangle \rightarrow_k^{D_1 \rightarrow D_2} \langle \epsilon, s'\rangle}{\langle C_1; C_2, s\rangle \rightarrow_k^{D_1 \rightarrow D_2} \langle C_2, s'\rangle} \qquad \frac{\langle C_1, s\rangle \rightarrow_k^{D_1 \rightarrow D_2} \langle C_1', s'\rangle}{\langle C_1; C_2, s\rangle \rightarrow_k^{D_1 \rightarrow D_2} \langle C_1'; C_2, s'\rangle}$$

Fig. 1. Operational semantics of WL

3.3 Formalization of the Principles

The purpose of a security condition is to formally characterize which programs obey a given flow policy. Hence, we can view a security condition as a function from an MLS-policy and a domain assignment to a set of WL-programs. As a notational convention, we write $PROP$ instead of $PROP((\mathcal{D}, \leq, \rightsquigarrow), dom)$ if $(\mathcal{D}, \leq, \rightsquigarrow)$ and dom are determined by the context.

We are now ready to formalize all prudent principles from Sect. 3.1 (with the exception of *non-occlusion* as explained before, and of the *who*-specific principle *protection* whose formalization is deferred to Sect. 4.1) by meta-properties of security conditions. To formalize *monotonicity* and *semantic consistency*, we define a *context* as a program $\mathcal{C}$, where the hole $\bullet$ may occur as an atomic subprogram. We use $\mathcal{C}\langle C\rangle$ to denote the program that one obtains by replacing each occurrence of $\bullet$ with C. As suggested in [2], we define semantic equivalence between programs by $\approx = \approx_{high}$, where $\approx_{high}$ is the strong *high*-bisimulation for the single-domain policy $(\{high\}, \{(high, high)\}, \emptyset)$.

Definition 4 (Semantic consistency). *A security property PROP is* semantically consistent*, if $C' \approx C$ and $\mathcal{C}\langle C\rangle \in PROP$ imply $\mathcal{C}\langle C'\rangle \in PROP$ for all commands C, C' without declassification instructions and for all contexts $\mathcal{C}$.*

Definition 5 (Relaxation). *A security property PROP is* relaxing*, if C is strongly secure implies $C \in PROP$.*

Definition 6 (Monotonicity). *A security property PROP complies with* monotonicity of release*, if*

1. *$\mathcal{C}\langle x{:=}y\rangle \in PROP$ implies $\mathcal{C}\langle [x{:=}y]_k\rangle \in PROP$ for all $\mathcal{C}$, x, y, and k and*
2. *$\rightsquigarrow \subseteq \rightsquigarrow'$ and $C \in PROP((\mathcal{D}, \leq, \rightsquigarrow), dom)$ imply $C \in PROP((\mathcal{D}, \leq, \rightsquigarrow'), dom)$.*

The intuition of Definition 6 with respect to declassifying assignments is the following. If a program C_2 is obtained from a given program C_1 by replacing a declassifying assignment with an ordinary assignment and C_2 is accepted already, then C_1, (i.e. the same program with additional brackets indicating that declassification is permissible) should certainly also be accepted.

To formalize *noninterference up-to* we have to consider executions of programs under a monitor. Whenever a program is about to execute a declassification step $\rightarrowtriangle_k^{D_1 \rightarrow D_2}$, the monitor terminates the execution. This is similar to changing the operational semantics by removing the rule for declassification steps with the condition $D_1 \not\leq D_2$, because in the operational semantics an execution is stopped, if no further transition from the current configuration is possible.

Definition 7 (Noninterference up-to). *A security property PROP is* noninterferent up-to*, if $C \in PROP$ implies that the program C is is strongly secure if it were executed with a declassification-prohibiting monitor.*

Definition 8 (Persistence). *A security property PROP is* persistent*, if $C' \in PROP$ holds for all C' that are reachable from some $C \in PROP$, i.e. if $C \in PROP$ and $\langle C, s\rangle \rightarrowtriangle^* \langle C', s'\rangle$ for some C, s , and s' implies $C' \in PROP$.*

A property PROP is weakly persistent*, if $C \in PROP$, $\langle C, s\rangle \rightarrowtriangle^* cnf$, and $cnf \rightarrowtriangle_k^{D_1 \rightarrow D_2} \langle C', s'\rangle$ for some C, s, s', cnf, D_1, D_2, and k implies $C' \in PROP$.*

The formalizations of the prudent principles in this section will serve as a sanity check for our new security condition in the next section.

4 Characterization of Security

In this section we define a novel security property to adequately control *who* may influence declassification, by his input on a given channel.

4.1 Input and Output

We extend the notions of programs and security with input and output. We assume two disjunctive sets, a set of *input channels* $\mathcal{I}$, and a set of *output channels* $\mathcal{O}$. Now, the *domain assignments* assign security domains to channels, too: $dom : (Var \cup \mathcal{I} \cup \mathcal{O}) \rightarrow \mathcal{D}$. A D-observer knows the input of channels *in* with $dom(in) \leq D$ and observes the output of channels *out* with $dom(out) \leq D$. The step relation $\rightarrowtriangle$ additionally has the disjoint sub-relations $\rightarrowtriangle_{chan,v}$, where $chan \in \mathcal{I} \cup \mathcal{O}$ and $v \in Val$. A $\rightarrowtriangle_{in,v}$-step models the input of the value v on the channel $in \in \mathcal{I}$. A $\rightarrowtriangle_{out,v}$-step models the output of the value v on the channel $out \in \mathcal{O}$. As convention we denote elements of $\mathcal{I}$ by *in*, of $\mathcal{O}$ by *out*, and of $\mathcal{I} \cup \mathcal{O}$ by *chan*. We extend WL by atomic programs for input x <- *in* and for output *Exp* -> *out*. The operational semantics contains the following new rules in addition to the ones in Fig. 1.

$$\frac{}{\langle x \texttt{ <- } in, s\rangle \rightarrowtriangle_{in,v} \langle \epsilon, [x = v]s\rangle} \qquad \frac{\langle Exp, s\rangle \downarrow v}{\langle Exp \texttt{ -> } out, s\rangle \rightarrowtriangle_{out,v} \langle \epsilon, s\rangle}$$

$$\frac{\langle C_1, s\rangle \rightarrowtriangle_{chan,v} \langle \epsilon, s'\rangle}{\langle C_1; C_2, s\rangle \rightarrowtriangle_{chan,v} \langle C_2, s'\rangle} \qquad \frac{\langle C_1, s\rangle \rightarrowtriangle_{chan,v} \langle C_1', s'\rangle}{\langle C_1; C_2, s\rangle \rightarrowtriangle_{chan,v} \langle C_1'; C_2, s'\rangle}$$

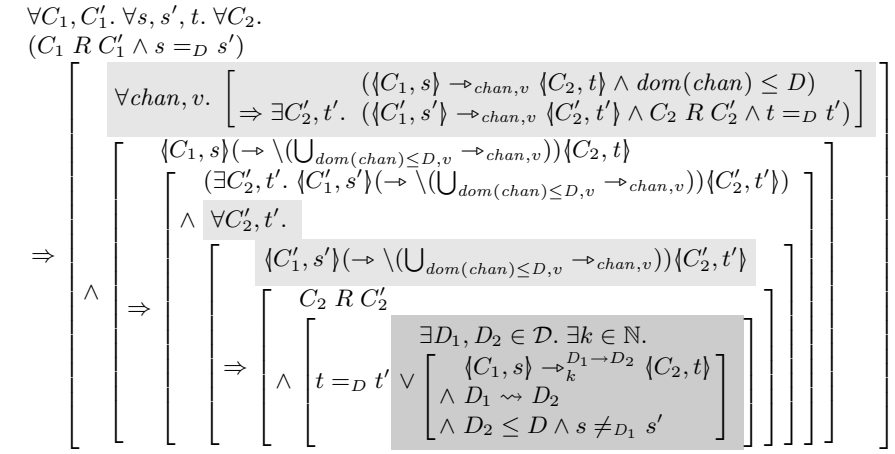

Fig. 2. Strong Bisimulation Relations with I/O

Unlike the rest of the operational semantics of WL, the input value v in the annotation of input steps is not deterministic. To account for that, we need to adapt the definition of a strong D-bisimulation accordingly. We define strong security as before (see Definition 3), however, now we define strong D-bisimulations as symmetric relations satisfying the sub-formula in Figure 2 without the box in dark-gray background. The boxes with light-gray background mark the new elements of the formula compared to Definition 3. Now, strong D-bisimulations give different guarantees depending on whether a step is an I/O-step with a D-visible channel or not. If it is an I/O-step with a D-visible channel (first box with light-gray background), then the simulating step needs to be an I/O-step on the same channel and with the same value. However, it is important, that for D-visible input steps only the step with the same value needs to satisfy the requirements $C_2\ R\ C_2'$ and $t =_D t'$. This captures the assumption, that a D-observer knows the input values of D-visible channels. For steps that are not I/O-steps with D-visible channels, the guarantees are required for all possible step results (last two boxes with light-gray background). This is necessary, because for input steps with non-D-visible channels the step result depends on the input value, which is not known to the observer and which should not be revealed to him.

To specify the input channels that must not effect a declassification step with a given instruction label, we assign sets of input channels to instruction labels.

Definition 9. *A protection labeling is a function* $prot : \mathbb{N} \to \mathfrak{P}(\mathcal{I})$.

Now we formalize the principle *protection*. An attacker cannot effect declassification k in a program by his behavior if the occurrence of a declassification k is invariant under change of the attacker's behavior. Hence, for the definition we fix the behavior of everybody else. Here behavior of everybody else means, the input provided by channels that are not in $prot(k)$. In the following for each

$P \subseteq \mathcal{I}$ we define $\overline{P} = \mathcal{I} \backslash P$. We define the sets $\mathcal{B}_k$ of behaviors of channels that are not in $prot(k)$ as lists of input events (in, v) and no-input events $\bot$: $\mathcal{B}_k = ((\overline{prot(k)} \times Val) \cup \{\bot\})^*$. For $b \in \mathcal{B}_k$ we define $\rightarrow_{b,k} \subseteq \twoheadrightarrow^*$ inductively by $cnf \rightarrow_{b,k} cnf$, if b has length 0, and $cnf_1 \rightarrow_{b,k} cnf_2$, if b consists of the prefix b' and the last element a, and there is a cnf' such that $cnf_1 \rightarrow_{b',k} cnf'$ and either $a = (in, v) \wedge cnf' \twoheadrightarrow_{in,v} cnf_2$, or $a = \bot \wedge cnf'(\twoheadrightarrow \setminus (\bigcup_{in \in \overline{prot(k)}, v} \twoheadrightarrow_{in,v}))cnf_2$. The relation $(\twoheadrightarrow \setminus (\bigcup_{in \in \overline{prot(k)}, v} \twoheadrightarrow_{in,v}))$ contains any step, that is not an input step of a channel, that may effect declassification k. I.e. a behavior b determines when an input step of a channel in $\overline{prot(k)}$ occurs and what value is read. The inputs of channels in $prot(k)$ are not determined by a $b \in \mathcal{B}_k$. As abbreviation we define $\twoheadrightarrow_k := \bigcup_{D_1, D_2} \twoheadrightarrow_k^{D_1 \to D_2}$ and $\twoheadrightarrow_{\bar{k}} := \twoheadrightarrow \setminus \twoheadrightarrow_k$ for all k.

Definition 10 (Protection). *A security property PROP is* protecting*, if, given $C \in PROP$, it holds that*

$$\forall k, b \in \mathcal{B}_k, s, cnf_1, cnf_2, cnf_1'. \left[\begin{pmatrix} \langle C, s \rangle \rightarrow_{b,k} cnf_1 \wedge cnf_1 \twoheadrightarrow_{\bar{k}} cnf_2 \\ \wedge \langle C, s \rangle \rightarrow_{b,k} cnf_1' \end{pmatrix} \Rightarrow \neg \exists cnf_2'.\ (cnf_1' \twoheadrightarrow_k cnf_2') \right]$$

The intuition is, that whether a given k-labeled execution step occurs or not is independent from the inputs of all channels from which k should be protected.

4.2 The Security Property for *Who*

First, to ensure that exceptional information flow only can occur by declassification steps, we define a supporting security property characterizing control of *where*. The property is defined similar to strong security, however, it permits declassification by declassification steps, if the exceptional flow complies with $\rightsquigarrow$. The property is an adaption of WHERE in [3] to the language with I/O.

Definition 11 (WHERE with I/O). *A* strong $(D, \rightsquigarrow)$-bisimulation *is a symmetric relation R on programs that satisfies the whole formula in Figure 2. The relation $\cong_D^{\rightsquigarrow}$ is the union of all strong $(D, \rightsquigarrow)$-bisimulations. A program C has secure information flow while complying with the restrictions where declassification can occur if $C \cong_D^{\rightsquigarrow} C$ holds for all $D \in \mathcal{D}$ (brief: C is* where*-secure or $C \in WHERE((\mathcal{D}, \leq, \rightsquigarrow), dom)$).*

Declassification is possible, since strong $(D, \rightsquigarrow)$-bisimulations do not always require the memory states after bisimulation steps to be D-indistinguishable. However, such exceptions are restricted: they may only occur after declassification steps $\twoheadrightarrow_k^{D_1 \to D_2}$, where the declassification target is visible to D ($D_2 \leq D$), the flow complies with the exceptional flow relation ($D_1 \rightsquigarrow D_2$), and the declassified information is D_1-visible ($s =_{D_1} s'$). The restrictions of WHERE on exceptional information flow offer the possibility to control *who* may effect declassification by only restricting further the occurrence of declassification steps.

Definition 12. *Let $P \subseteq \mathcal{I}$ and $k \in \mathbb{N}$. A* (P, k)-protecting bisimulation *is a symmetric relation $R \subseteq \text{Conf} \times \text{Conf}$ such that for all $cnf_1\ R\ cnf_1'$ it is*

- $\forall cnf_2.\ (cnf_1 \twoheadrightarrow^* cnf_2 \Rightarrow \neg \exists cnf_3\, cnf_2 \twoheadrightarrow_k cnf_3)$ *or*

– *for all* cnf_2 *with* $cnf_1 \rightarrowtriangle cnf_2$ *it is*

$$\exists cnf'_2.\ cnf'_1 \rightarrowtriangle cnf'_2$$
$$\wedge\ \forall cnf'_2.$$
$$\left[\begin{array}{l} [(cnf_1 \rightarrowtriangle_k cnf_2 \wedge cnf'_1 \rightarrowtriangle cnf'_2) \Rightarrow cnf'_1 \rightarrowtriangle_k cnf'_2] \\ \wedge\ [(cnf_1(\rightarrowtriangle \setminus (\bigcup_{in \in \overline{P}, v} \rightarrowtriangle_{in,v}))cnf_2 \wedge cnf'_1 \rightarrowtriangle cnf'_2) \Rightarrow cnf_2\ R\ cnf'_2] \\ \wedge\ \forall in \in \overline{P}, v.\ \left[\begin{array}{l} \left(\begin{array}{l} cnf_1 \rightarrowtriangle_{in,v} cnf_2 \\ \wedge\ cnf'_1(\rightarrowtriangle \setminus (\bigcup_{v' \neq v} \rightarrowtriangle_{in,v'}))cnf'_2 \end{array}\right) \\ \Rightarrow cnf_2\ R\ cnf'_2 \end{array}\right] \end{array}\right]$$

Given C*, a protection labeling* $prot : \mathbb{N} \to \mathfrak{P}(\mathcal{I})$*, and* k*,* $WHO_{C,prot}(k)$ *holds, iff for all from* C *reachable programs* C' *there is a* $(prot(k), k)$*-protecting bisimulation* R *such that* $\forall s.\ \langle C', s\rangle\ R\ \langle C', s\rangle$*. A program* C *is* who-protected *if* $WHO_{C,prot}(k)$ *holds for all* k *(brief:* $C \in WHO$*).*

Configurations are related by a $(prot(k), k)$-protecting bisimulation, if there is no reachable declassification step k, because in this case such a step cannot be effected by any channel input. Else, if such a step occurs, it has to be simulated by a step with the same annotation. The results of bisimulation steps also need to be in the bisimulation relation, except, when the steps are input steps of channels in $\overline{prot(k)}$ and have differing values. This exception captures that these channels may effect declassification. The predicate $WHO_{C,prot}(k)$ initially only requires configurations with equal memories to be related by a bisimulation. This captures that there are no restrictions on the influence of initial values of variables on declassification. Hence, any difference in occurrence of a declassification step k is not caused by input channels that must not effect declassification.

We define the security property for control of *who* may effect declassification.

Definition 13 (WHERE&WHO)**.** *A program* C *has secure information flow while complying with the restrictions where declassification can occur and who may effect declassification if* C *is* where-*secure and* who-*protected. (brief:* C *is* where&who-*secure or* $C \in \text{WHERE\&WHO}((\mathcal{D}, \leq, \rightsquigarrow), dom)$*).*

Example 1. We consider the example from Sect. 2 with the two-level flow policy where *high* $\rightsquigarrow$ *low*, a domain assignment *dom* assigning *high* to movie and *low* to every other variable or channel, and $prot(1) = prot(2) = \{\mathsf{webI}\}$. We first consider the first program. The variable movie is only read by the declassifying assignments. The channel webI either is not read at all, or, if the input of bankI is fixed, the execution of declassification does not depend on the input from webI. Hence, the program is *where*&*who*-secure. Now we consider the second program. Consider two configurations, both consisting of the branching instruction with the branch condition (payOK or isPreferred(paydat)), and of memories, where in both payOK is 0 and in one isPreferred(paydat) is 1 and in the other 0. These configurations, are not $(\{\mathsf{webI}, 1\})$-protecting bisimilar. However, their bisimilarity is required by *WHO*-protection of 1, when we consider an initial memory that assigns to byMachine the value 0. Hence, this program is not *where*&*who*-secure. This classification of the two programs is according to our intuition.

The property WHERE&WHO complies with all the principles from Sect. 3.

Theorem 1 (Compliance to Principles). WHERE&WHO *is*

1. semantically consistent*.*
2. relaxing*,*
3. monotonic*,*
4. noninterferent up-to*,*
5. persistent*, and*
6. protecting*.*

5 Enforcing Who Control by a Type System for WHERE

There are some cases, where WHERE is equivalent to WHERE&WHO. These are not only the trivial cases, but also cases where restrictions on *who* are imposed. We capture these cases by the following theorem.

Theorem 2. *Let C be given. Let $range(\rightsquigarrow) := \{D \in \mathcal{D} | \exists D' \in \mathcal{D}.\ D' \rightsquigarrow D\}$. If*

1. *C is* where-*secure and*
2. *$\forall in \in \bigcup_{k\in\mathbb{N}} prot(k).\ \forall D \in range(\rightsquigarrow).\ \neg \big(dom(in)(\leq \cup \rightsquigarrow)^* D\big)$,*

then C satisfies WHERE&WHO*.*

Since WHERE is parameterized with multi-level flow policies, which can be used to express integrity aspects, and since WHERE already restricts declassification within this policy, satisfaction of WHERE with a suitable flow policy can ensure WHERE&WHO. Inspired by this result, given a protection labeling *prot*, an MLS-policy $(\mathcal{D}, \leq, \rightsquigarrow)$, and a domain assign *dom*, we call *prot flow-enforced by* $(\mathcal{D}, \leq, \rightsquigarrow)$ *and dom*, whenever the second hypothesis of Theorem 2 is satisfied. By this theorem, if we have given a policy such that *prot* is flow-enforced, it just remains to check that a program is *where*-secure to check *where*&*who*-security.

5.1 Refining Flow Policies

To apply Theorem 2 to a given program and security policy, it might be necessary to refine the MLS-policy and the domain assignment, in order to capture the desired integrity aspect.

Definition 14. *Given MLS-policies $(\mathcal{D}_1, \leq_1, \rightsquigarrow_1)$, $(\mathcal{D}_2, \leq_2, \rightsquigarrow_2)$ and domain assignments $dom_1 : (Var \cup \mathcal{I} \cup \mathcal{O}) \to \mathcal{D}_1$, $dom_2 : (Var \cup \mathcal{I} \cup \mathcal{O}) \to \mathcal{D}_2$, we call a function $abs : \mathcal{D}_1 \to \mathcal{D}_2$* abstracting*, iff*

1. *abs is surjective,*
2. *$\forall D_1, D_1' \in \mathcal{D}_1.\ (D_1 \leq_1 D_1' \Rightarrow abs(D_1) \leq_2 abs(D_1'))$,*
3. *$\forall D_1, D_1' \in \mathcal{D}_1.\ (D_1 \rightsquigarrow_1 D_1' \Rightarrow abs(D_1)(\rightsquigarrow_2 \cup \leq_2) abs(D_1'))$,*
4. *$\forall a \in (Var \cup \mathcal{I} \cup \mathcal{O}).\ dom_2(a) = abs(dom_1(a))$.*

We call $(\mathcal{D}_1, \leq_1, \rightsquigarrow_1)$ and dom_1 a policy refinement *of $(\mathcal{D}_2, \leq_2, \rightsquigarrow_2)$ and dom_2, iff there is an abstracting function $abs : \mathcal{D}_1 \to \mathcal{D}_2$.*

In a refinement of a given policy, security domains may be split up and the flow relations may impose additional restrictions. However, a refinement must not relax the restrictions on the flow of information between variables and channels.

Example 2. For instance, we consider the first program in Sect. 2 with the policy and domain assignment we present for the program in Sect. 4. The security domain of the channel webI is *low* and *low* is in the range of $\rightsquigarrow$, i.e. *prot* is not flow-enforced. However, there is a more restrictive policy, such that *prot* is flow-enforced and the program is *where*-secure: we add a security domain *web*, extend the flow relation to $\leq' = \leq \cup \{(low, web), (web, web)\}$, and we assign *web* to webI, paydatweb and bankO. The function *abs* defined by $abs(low) = low$, $abs(web) = low$, and $abs(high) = high$ is abstracting, i.e. the new MLS-policy and domain assignment are a policy refinement.

Lemma 1. *Let* $(\mathcal{D}_1, \leq_1, \rightsquigarrow_1)$*,* $dom_1 : (Var \cup \mathcal{I} \cup \mathcal{O}) \to \mathcal{D}_1$*,* $(\mathcal{D}_2, \leq_2, \rightsquigarrow_2)$*,* $dom_2 : (Var \cup \mathcal{I} \cup \mathcal{O}) \to \mathcal{D}_2$*, and* $prot : \mathbb{N} \to \mathfrak{P}(\mathcal{I})$ *be given. If* $abs : \mathcal{D}_1 \to \mathcal{D}_2$ *is abstracting then*

1. $\forall D_2.\, \forall s, s'.\, \left[(\forall D_1 \in \mathcal{D}_1.\, (abs(D_1) \leq_2 D_2 \Rightarrow s =_{D_1,1} s')) \Leftrightarrow s =_{D_2,2} s'\right]$*, and*
2. $\forall D_2.\, \forall C, C'.\, \left[(\forall D_1 \in \mathcal{D}_1.\, (abs(D_1) \leq_2 D_2 \Rightarrow C \approx^{\rightsquigarrow_1}_{D_1} C')) \Rightarrow C \approx^{\rightsquigarrow_2}_{D_2} C'\right]$,

where $=_{D_1,1}$ *is the* D_1*-equality with respect to* $\leq_1$ *and* dom_1 *for all* $D_1 \in \mathcal{D}_1$*, and* $=_{D_2,2}$ *is the* D_2*-equality with respect to* $\leq_2$ *and* dom_2 *for all* $D_2 \in \mathcal{D}_2$*.*

For our purpose, only flow-enforcing refinements are relevant.

Theorem 3. *Let* C*,* $(\mathcal{D}_2, \leq_2, \rightsquigarrow_2)$*,* dom_2 *and* $prot$ *be given. If there is a refinement* $(\mathcal{D}_1, \leq_1, \rightsquigarrow_1)$ *and* dom_1 *of* $(\mathcal{D}_2, \leq_2, \rightsquigarrow_2)$ *and* dom_2 *such that*

- $C \in WHERE((\mathcal{D}_1, \leq_1, \rightsquigarrow_1), dom_1)$ *and*
- $prot$ *is flow-enforced by* $(\mathcal{D}_1, \leq_1, \rightsquigarrow_1)$ *and* dom_1*,*

then $C \in$ WHERE&WHO$((\mathcal{D}_2, \leq_2, \rightsquigarrow_2), dom_2)$*.*

Theorem 3 shows, that even if policies are not beforehand designed to represent the integrity aspect with respect to declassification, flow-enforced protection labelings can be exploited.

5.2 Static Enforcement of WHERE&WHO

We propose an enforcement mechanism for WHERE&WHO in two steps. The first step is to find a refinement of the given flow policy and domain assignment such that *prot* is flow-enforced. The second step is to apply a type system enforcing WHERE with respect to the policy refinement, and to apply Theorem 3.

To find a suitable refinement for a given policy, we split up security domains from that information may flow to security domains in the range of $\rightsquigarrow$ into two security domains, with the intuition, that one has high integrity and one has low integrity with respect to the input channels. We construct the normal flow relation such that it relates domains of high integrity with the respective domains of low integrity, however, not the other way round. We construct the exceptional flow relation such that it has only security domains of high integrity as source. To input channels, from that declassification should be protected, we assign security domains of low integrity. To determine for each variable and output channel, whether it needs to be assigned to the respective domain of low or high integrity, a type inference based on the type system has to be pursued, which is out of the scope of this paper. The type system to enforce WHERE is identical to the one of [3]. A program C is *typable*, which we denote by $\vdash C$, if $\vdash C$ can be derived by the rules in Fig. 3.

$$\frac{\forall x \in vars(Exp).\ dom(x) \leq D}{\vdash Exp : D} \qquad \frac{}{\vdash \mathsf{skip}} \qquad \frac{\vdash Exp : D \quad D \leq dom(x)}{\vdash x{:=}Exp} \qquad \frac{dom(y) \rightsquigarrow dom(x)}{\vdash [x{:=}y]_k}$$

$$\frac{\vdash C_1 \quad \vdash C_2}{\vdash C_1\,;\,C_2} \qquad \frac{\vdash B : low \quad \vdash C}{\vdash \mathsf{while}\ B\ \mathsf{do}\ C\ \mathsf{od}} \qquad \frac{\vdash C_1 \quad \vdash C_2 \quad \vdash B : D \quad \forall D' \not\geq D : C_1 \cong^{\rightsquigarrow}_{D'} C_2}{\vdash \mathsf{if}\ B\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2\ \mathsf{fi}}$$

$$\frac{dom(in) \leq dom(x)}{\vdash x \texttt{ <- } in} \qquad \frac{\vdash Exp : D \quad D \leq dom(out)}{\vdash Exp \texttt{ -> } out}$$

Fig. 3. Rules of the Security Type System enforcing WHERE

Theorem 4 (Soundness of Security Type System). *Let* $\vdash C$.

1. *C is* where-*secure.*
2. *If prot is flow-enforced by* $(\mathcal{D}, \leq, \rightsquigarrow)$ *and dom then C is* where&who-*secure.*

Note that *flow-enforced* is a property of just the MLS policy, the domain assignment, and the protection labeling, that can be checked by checking whether security domains are related by the transitive closure of $\leq$ and $\rightsquigarrow$.

The rule for conditional branches contains a semantic side condition ($\forall D' \not\geq D : C_1 \cong^{\rightsquigarrow}_{D'} C_2$). To be able to fully automatize the analysis, we additionally need a syntactic approximation of this side condition. A simple solution is to require $\vdash B : low$. Examples for less restrictive approaches to syntactic approximations for similar side conditions can be found in [1, 6, 3].

6 Related Work

The development of adequate control for noninterference-like conditions is an active research area. In the following discussion, we focus on related work that targets the control of *who* can initiate declassification. For other dimensions of declassification, we refer to the overview on declassification in [2].

Approaches, based on *robustness* [7–9, 8] permit any information to leak, as long as the leak appears for all possible behaviors of attackers. Hence, *robust declassification* does not comply to *noninterference up-to*. Possible behaviors of an attacker are explicitly defined as programs with limited capability to write [8]. Different to WHERE&WHO, *robust declassification* does not differentiate which channels may influence which declassifications.

A different kind of control of *who* can be conducted on the basis of authorization. The decentralized label model [10] explicitly defines the flow policy using ownership labels, that state which principal permits reading to which other principals for each information. Here, declassification is restricted in that each principal may only relax the requirements imposed by his label. Abstract noninterference [11] is also claimed to control the dimension *who*. However, here *who* is not used in the sense of who may influence, but in the sense of attackers with different observational capabilities.

Our prudent principles of declassification extend and, in some cases, refine the ones in [2]. Interestingly, the conjunction of *noninterference up-to* and *weak persistence* has similarities to *noninterference unless* in [12].

7 Conclusion

We presented a novel approach to controlling who can initiate declassification. Our security condition WHO permits to control who can effect a given declassification in a program. We integrated WHO with the previously defined condition WHERE, which controls where in the program and where in the flow policy declassification may occur. We argued for the adequacy of the combined condition WHERE&WHO with the help of prudent principles of controlling declassification. We showed that WHERE&WHO can be reduced to WHERE for some flow policies. Based on this result, we developed a technique for enforcing WHERE&WHO by, firstly, refining a given flow policy and, secondly, applying an existing type system for WHERE.

Acknowledgments. We thank the anonymous reviewers for their suggestions. This work was funded by the DFG in the Computer Science Action Program and by the Information Society Technologies program of the European Commission, Future and Emerging Technologies under the IST-2005-015905 MOBIUS project. This article reflects only the authors' views, and the Commission, the DFG, and the authors are not liable for any use that may be made of the information contained therein.

References

1. Mantel, H., Sands, D.: Controlled Declassification based on Intransitive Noninterference. In: APLAS 2004. Volume 3303 of LNCS., Springer (2004) 129–145
2. Sabelfeld, A., Sands, D.: Dimensions and Principles of Declassification. In: Proc. of the 18th IEEE Computer Security Foundations Workshop, IEEE (2005) 255–269
3. Mantel, H., Reinhard, A.: Controlling the What and Where of Declassification in Language-Based Security. In: ESOP 2007. Volume 4421 of LNCS., Springer (2007) 141–156
4. Sabelfeld, A., Sands, D.: Probabilistic Noninterference for Multi-threaded Programs. In: Proc. of the 13th IEEE Computer Security Foundations Workshop, IEEE (2000) 200–215
5. Sabelfeld, A., Sands, D.: A Per Model of Secure Information Flow in Sequential Programs. In: ESOP 1999. Volume 1576 of LNCS., Springer (1999) 50–59
6. Köpf, B., Mantel, H.: Transformational typing and unification for automatically correcting insecure programs. International Journal of Information Security (IJIS) **6**(2–3) (2007) 107–131
7. Zdancewic, S., Myers, A.: Robust declassification. In: Proc. of IEEE Computer Security Foundations Workshop, IEEE (2001) 15–26
8. Myers, A.C., Sabelfeld, A., Zdancewic, S.: Enforcing Robust Declassification and Qualified Robustness. Journal of Computer Security **14** (2006) 157 – 196
9. Chong, S., Myers, A.C.: Decentralized robustness. In: Proc. of the 19th IEEE workshop on Computer Security Foundations, IEEE (2006) 242–256
10. Myers, A.C., Liskov, B.: Protecting Privacy using the Decentralized Label Model. ACM Trans. Softw. Eng. Methodol. **9**(4) (2000) 410–442
11. Mastroeni, I.: On the role of abstract non-interference in language-based security. In: APLAS 2005. Volume 3780 of LNCS., Springer (2005) 418–433
12. Goguen, J.A., Meseguer, J.: Unwinding and Inference Control. In: Proceedings of the IEEE Symposium on Security and Privacy, IEEE (1984) 75–86