Declassification with Explicit Reference Points

Alexander Lux and Heiko Mantel

Computer Science, TU Darmstadt, Germany
{lux,mantel}@cs.tu-darmstadt.de

Abstract. Noninterference requires that public outputs of a program must be completely independent from secrets. While this ensures that secrets cannot be leaked, it is too restrictive for many applications. For instance, the output of a knowledge-based authentication mechanism needs to reveal whether an input matches the secret password. The research problem is to allow such exceptions without giving up too much. Though a number of solutions has been developed, the problem is not yet satisfactorily solved. In this article, we propose a framework to control what information is declassified. Our contributions include a policy language, a semantic characterization of information flow security, and a sound security type system. The main technical novelty is the explicit treatment of so called reference points, which allows us to offer substantially more flexible control of what is released than in existing approaches.

1 Introduction

Information systems process a wide range of secrets, including national secrets, private data, and electronic goods. Confidentiality requirements may also originate from security mechanisms, e.g., the confidentiality of passwords (for authentication mechanisms), of cryptographic keys (for encryption), of random challenges (for security protocols), or of capabilities (for access controls).

Static program analysis can be applied to ensure that secrets cannot be leaked during program execution or, in other words, that the flow of information in a program is secure. The resulting security guarantee is usually captured as a lack-of-dependency property, which states that the output to untrusted observers is independent from all data that they are not authorized to obtain.

While strict lack-of-dependency properties like *noninterference* [1] are rather attractive from a theoretical point of view, they become impractical if secrets shall be deliberately released. For instance, an electronic good (initially a secret) should be released to a customer after it has been paid for and an authentication attempt necessarily reveals some information about the stored password. In these cases, it is necessary to relax strict lack of dependency to some extent – but how much? The research community has been actively searching for solutions and proposed a number of approaches in recent years. However, the problem of controlled declassification is not yet satisfactorily solved.

Mantel and Sands proposed in [2] to distinguish carefully whether a given approach controls *what* can be declassified, *where* declassification can occur, and

M. Backes and P. Ning (Eds.): ESORICS 2009, LNCS 5789, pp. 69–85, 2009.

who can initiate declassification. Based on these *dimensions of declassification*, a taxonomy of known approaches to control declassification was developed in [3]. In this article, we focus on what information may be declassified.

When reviewing existing approaches to controlling the *what* dimension with similar syntax, we found significant differences on the semantic level.

For instance, *delimited release* [4] uses so called *escape hatches* to indicate what may be declassified by a program. An escape hatch has the syntax $\mathsf{declassify}(exp, d)$, where exp is an expression and d is a security domain in the given flow policy. Semantically, the escape hatch specifies that the value of exp in the initial state (i.e. before program execution begins) may be revealed to the security domain d. This permission dominates all restrictions that are defined by a given flow relation $\rightsquigarrow$. That is, if the policy contains an escape hatch $\mathsf{declassify}(exp, d)$ then the initial value of exp may be revealed to d – even if exp incorporates variables from a security domain d' such that $d' \not\rightsquigarrow d$.

Delimited non-disclosure [5] indicates that the expression exp may be declassified in the program c by commands of the form $\mathsf{declassify}\ (exp)\ \mathsf{in}\ \{c\}$. Security domains are not explicitly mentioned in declassification commands because implicitly a flow policy with only two domains, public and secret, is assumed. Under this flow policy, declassification always constitutes an exception to the restriction that information must not flow from secret to public. Interestingly, delimited non-disclosure permits declassification of the value exp in *any* state in which exp is evaluated during the execution of the command c. This local view is different from permitting the declassification of the initial value of exp or of the value of exp in the state before the execution of the command c starts.

That is, despite the syntactic similarities between delimited release and delimited non-disclosure, these approaches differ significantly in their semantics. The implicit assumptions of initial and local reference points can also be observed in further approaches, e.g., in [6,7,8] and [9], respectively.

In this article, we propose *explicit reference points* as a concept to support a flexible specification of what secrets may be declassified. A *declassification guard* $\mathsf{dguard}(r, exp, d)$ specifies the values that may be declassified by an expression exp *and* by a reference point r. The reference point determines a set of states with the intention that the value of exp in any of these states may be declassified to domain d. Unlike in earlier approaches, our framework allows one to make explicit in which states exp is evaluated. Delimited release and delimited non-disclosure can be simulated by placing reference points at the beginning of a program or at all points where exp is evaluated, respectively. However, our framework goes far beyond providing a uniform view on initial and local reference points. Rather, explicit reference points can be placed at *any* point in a program, and this is adequately supported by our semantic characterization of security.

In Section 2, we elaborate the limitations of leaving reference points implicit and sketch the use of our declassification framework. Our novel technical contributions are presented in Section 3 (policy language), Section 4 (security condition), and Section 5 (security type system and soundness result). We conclude with a presentation of further examples and a comparison to related work.

2 From Implicit to Explicit Reference Points

Many approaches to controlling what is declassified implicitly assume that reference points are either always initial or always local (see Section 1). We point out the limitations of this assumption in Section 2.1 and offer a first glance at the explicit treatment of reference points in our framework in Section 2.2.

2.1 Initial versus Local Reference Points

As a running example, we consider a program that calculates the average of 100 salaries. We assume that the individual salaries (which constitute inputs to the program) must be kept secret, but that the resulting average may be published. We capture this requirement by a two-level flow policy forbidding that information flows from a security domain secret to a security domain public (i.e., $\mathsf{secret} \not\rightsquigarrow \mathsf{public}$). The domain assignment associates the program variables $\mathsf{sal}_1, \ldots, \mathsf{sal}_{100}$ (storing the individual salaries) with the domain secret and the program variable avg (storing the resulting average) with the domain public.

The desired control of what is declassified can be expressed with delimited release (see P_1 below) as well as with delimited non-disclosure (see P_2 below):

$$P_1 = \quad \mathsf{avg} := \mathsf{declassify}((\mathsf{sal}_1 + \mathsf{sal}_2 + \ldots + \mathsf{sal}_{100}) \;/\; 100, \mathsf{public})$$

$$P_2 = \quad \mathsf{declassify}((\mathsf{sal}_1 + \mathsf{sal}_2 + \ldots + \mathsf{sal}_{100}) \;/\; 100) \;\; \mathsf{in}\; \{\mathsf{avg}{:=}(\mathsf{sal}_1 + \mathsf{sal}_2 + \ldots + \mathsf{sal}_{100}) \;/\; 100\}$$

So far, we do not observe any significant differences between the two implicit assumptions of initial versus local reference points. However, differences become apparent if we place the program fragments into a larger context. For instance,

$$P_3 = \quad \mathsf{sal}_1 := \mathsf{sal}_1;\; \mathsf{sal}_2 := \mathsf{sal}_1;\; \ldots;\; \mathsf{sal}_{100} := \mathsf{sal}_1;\; P_1$$

effectively assigns sal_1 to avg. Intuitively, this clearly is a breach of security because the policy permits only to declassify the average value of all salaries, but not the value of any individual salary. In this case, delimited release is, indeed, a suitable characterization because P_3 violates this security condition.

In contrast, delimited non-disclosure is not suitable to detect such an information leak. Each of the following two programs (where $Avg = (\mathsf{sal}_1 + \mathsf{sal}_2 + \ldots + \mathsf{sal}_{100}) \;/\; 100$) satisfies delimited non-disclosure although P_4 as well as P_5 intuitively incorporate the same insecurity as P_3:

$$P_4 = \quad \mathsf{sal}_1 := \mathsf{sal}_1;\; \mathsf{sal}_2 := \mathsf{sal}_1;\; \ldots;\; \mathsf{sal}_{100} := \mathsf{sal}_1;\; P_2$$

$$P_5 = \quad \mathsf{declassify}(Avg) \;\; \mathsf{in}\; \{\; \mathsf{sal}_1 := \mathsf{sal}_1;\; \mathsf{sal}_2 := \mathsf{sal}_1;\; \ldots;\; \mathsf{sal}_{100} := \mathsf{sal}_1;\; \mathsf{avg}{:=}Avg\}$$

However, this does not mean that delimited release is fully satisfactory. There are programs for which delimited release is too restrictive. Consider, for instance,

$$P_6 = \quad \mathsf{sal}_1 \;\texttt{<-}\; \mathsf{input};\; \mathsf{sal}_2 \;\texttt{<-}\; \mathsf{input};\; \ldots;\; \mathsf{sal}_{100} \;\texttt{<-}\; \mathsf{input};\; P_1$$

where input is an input channel that supplies the ith salary for the ith assignment in the first line of the program. This program would be rejected by delimited

release[1] although, intuitively, the program is secure given that inputs are indeed delivered as specified. The underlying reason is that delimited release implicitly assumes initial reference points, which are not adequate in this scenario. Interestingly, delimited non-disclosure is fulfilled by the following program:

$$P_7 = \quad \mathsf{declassify}(Avg) \\ \mathsf{in}\ \{\ \mathsf{sal}_1\ \texttt{<-}\ \mathsf{input};\ \mathsf{sal}_2\ \texttt{<-}\ \mathsf{input};\ \ldots;\ \mathsf{sal}_{100}\ \texttt{<-}\ \mathsf{input};\ \mathsf{avg}{:=}Avg\}$$

Hence, the implicit assumptions of initial reference points (delimited release) and of local reference points (delimited non-disclosure) both have their limits.

2.2 Towards Explicit Reference Points

We propose *declassification guards* as a means to indicate more explicitly what may be declassified. A declassification guard has the form $\mathsf{dguard}(r, exp, d)$, where dguard is a keyword, r is a reference label, exp is an expression, and d is a security domain. Reference labels are also used to annotate selected commands in a given program. Hereby, each reference label specifies a set of *r-labeled program configurations*, namely those configurations that can be reached in a run of the program such that the next command to be executed is annotated with r. Intuitively, a declassification guard $\mathsf{dguard}(r, exp, d)$ specifies that if an r-labeled configuration occurs in a given run, then the value of exp in this configuration may be released to domain d afterwards in this run.

We illustrate declassification guards at our running example for two scenarios with different security requirements. In the first scenario, the initial values of the program variables $\mathsf{sal}_1, \ldots, \mathsf{sal}_{100}$ must be kept secret, while the average of these initial values may be declassified. In the second scenario, the values read from an input channel into $\mathsf{sal}_1, \ldots, \mathsf{sal}_{100}$ must be kept secret, while the average of these inputs may be declassified. To make things concrete, we consider the following variants of the programs P_3 and P_6:

$$P_3' = \quad \mathsf{ref}_1\ :\ \mathsf{sal}_1 := \mathsf{sal}_1;\ \mathsf{sal}_2 := \mathsf{sal}_1;\ \ldots;\ \mathsf{sal}_{100} := \mathsf{sal}_1; \\ \mathsf{ref}_{101}\ :\ \mathsf{avg} := Avg$$

$$P_6' = \quad \mathsf{ref}_1\ :\ \mathsf{sal}_1\ \texttt{<-}\ \mathsf{input};\ \mathsf{sal}_2\ \texttt{<-}\ \mathsf{input};\ \ldots;\ \mathsf{sal}_{100}\ \texttt{<-}\ \mathsf{input}; \\ \mathsf{ref}_{101}\ :\ \mathsf{avg} := Avg$$

The intended control of declassification in the first scenario can be captured by the declassification guard $\mathsf{dguard}(\mathsf{ref}_1, Avg, \mathsf{public})$ for P_3'. In the second scenario, the intended control can be captured by $\mathsf{dguard}(\mathsf{ref}_{101}, Avg, \mathsf{public})$. For these declassification guards, our security condition (to be presented in Section 4) is violated by P_3' but satisfied by P_6', which is exactly as desired because P_3' is intuitively insecure (the initial value of sal_1 is revealed to public), while P_6' intuitively is secure. Note that the explicit treatment of reference points is crucial to achieve this. While an initial reference point is appropriate in the first scenario, a local reference point is needed for P_6' in the second scenario. Therefore,

[1] The programming languages in [4] and [5] lack explicit I/O-commands. We assume here a straightforward extension of delimited release and delimited non-disclosure that treats input commands like non-deterministic assignments.

implicitly assuming that reference points are either always initial or always local (as assumed by delimited release and by delimited non-disclosure, respectively), is not satisfactory (also recall the examples in Section 2.1).

Our framework is not restricted to initial and local reference points. This feature is helpful if a value may be declassified that originates in some intermediate state of a run without being immediately released. In fact, a declassification guard should always contain the earliest point in a program where the value to be declassified originates. This helps to detect insecurities of the following kind:

$$\begin{aligned} P_8 = \quad & \mathsf{ref}_1 \ : \ \mathsf{sal}_1 \ \texttt{<-} \ \mathsf{input}; \ \mathsf{sal}_2 \ \texttt{<-} \ \mathsf{input}; \ \ldots; \ \mathsf{sal}_{100} \ \texttt{<-} \ \mathsf{input}; \\ & \mathsf{ref}_{101} \ : \ \mathsf{sal}_1 := \mathsf{sal}_1; \ \mathsf{sal}_2 := \mathsf{sal}_1; \ \ldots; \ \mathsf{sal}_{100} := \mathsf{sal}_1; \\ & \mathsf{ref}_{201} \ : \ \mathsf{avg} := Avg \end{aligned}$$

For the second scenario, one should choose $\mathsf{dguard}(\mathsf{ref}_{101}, exp, \mathsf{public})$ and *not* $\mathsf{dguard}(\mathsf{ref}_{201}, exp, \mathsf{public})$ because the second declassification guard would occlude that the intermediate computation leaks an individual input (sal_1). Our security condition (to be presented in Section 4) is, indeed, violated by P_8 for the first declassification guard (but not for the second).

3 Security Policies with Explicit Reference Points

The specification of a security policy comprises four parts: a specification of the security domains, of the assignment of security domains to program variables and to communication channels, of the regular flow between security domains, and of the exceptional flow by a list of declassification guards.

Definition 1. *Let $\mathcal{D}$ be a set of security domains, Var be a set of program variables, I and O be two disjoint sets of input and output channels, respectively, $\mathcal{E}$ be a set of expressions, and $\mathcal{R}$ be a set of reference labels.*

A policy specification *has the following form:*

$$\begin{aligned} \mathit{Spec} ::= \ & \textbf{SecurityDomains} \ \mathit{DomSpec} \ \textbf{DomainAssign} \ \mathit{DomA} \\ & \textbf{RegularFlow} \ \mathit{Flow} \ \textbf{ExceptionalFlow} \ \mathit{DGuards} \ \textbf{EndPolicy} \end{aligned}$$

The sub-specifications are defined by the following grammar (where $d \in \mathcal{D}$, $x \in Var$, $ch \in I \cup O$, $exp \in \mathcal{E}$, and $r \in \mathcal{R}$):

$$\begin{aligned} \mathit{DomSpec} ::= \ & ; \mid d; \ \mathit{DomSpec} \\ \mathit{DomA} ::= \ & ; \mid x : d; \ \mathit{DomA} \mid ch : d; \ \mathit{DomA} \\ \mathit{Flow} ::= \ & ; \mid d \texttt{=>} d; \ \mathit{Flow} \\ \mathit{DGuards} ::= \ & ; \mid \mathsf{dguard}(r, exp, d); \ \mathit{DGuards} \end{aligned}$$

In order to make policy specifications more concise, we introduce two assumptions. Firstly, we assume that there is a domain $\mathsf{public} \in \mathcal{D}$ and that all program variables and communication channels are associated with public by default, i.e., unless otherwise explicitly specified. Secondly, we consider the flow relation modulo reflexivity and transitivity. If a flow relation is given by a policy specification then the reflexive and transitive closure is implicitly computed.

If a security domain is listed more than once in *DomSpec*, if a domain is used in *DomA*, *Flow* or *DGuards* that is not listed in *DomSpec*, or if *DomA* contains

multiple declarations for the same program variable or communication channel, then we call the policy specification *inconsistent*. We also call it inconsistent if it induces a flow relation that is not an ordering (violation of anti-symmetry). Otherwise, a policy specification is *consistent*.

Semantically, a policy specification corresponds to a quadruple.

Definition 2. *A* security policy *Pol is a tuple* $(D, dom, \leq, G)$*, where* D *is a finite set of security domains,* $dom : (Var \cup I \cup O) \to D$ *is a domain assignment,* $\leq \subseteq D \times D$ *is a partial order expressing the permitted flow between domains (analogously to* $\rightsquigarrow$ *in Section 1), and* $G \subseteq (\mathcal{R} \times \mathcal{E} \times D)$ *is a set of guards.*

The semantic of a policy specification *Spec* is a quadruple $(D, dom, \leq, G)$ that is defined as follows. The set D equals the union of the set of all domains listed in *DomSpec* with $\{\mathsf{public}\}$. The function *dom* returns domain d for a variable x or for a channel ch if $x : d$ or $ch : d$, respectively, appears in *DomA*. Otherwise, *dom* returns public. The relation $\leq$ relates d_1 and d_2 if $d_1 = d_2$, if $d_1 => d_2$, or if there is a sequence of domains $d_3, \ldots, d_n$ such that *Flow* contains $d_1 => d_3$, $d_n => d_2$, and $d_i => d_{i+1}$ for all $i \in \{3, \ldots, n-1\}$. The set G contains (r, exp, d) iff $\mathsf{dguard}(r, exp, d)$ appears in *DGuards*. Note that this construction reflects our two assumptions from above and ensures that consistent policy specifications induce quadruples that are security policies according to Definition 2.

4 Characterization of Security

We are now ready to formalize under which conditions a given policy is fulfilled. The main innovation of our security condition is that explicit reference points are adequately supported. The key difficulty we faced when defining this condition was to collect the values that may be declassified on the fly during a run.

Our exposition in this section focuses on the semantic level. That is, we use a semantic model of program execution to define when a given program model satisfies a given security policy. We lift security to the syntactic level, by defining that a policy specification is fulfilled by a program if and only if the corresponding security policy is fulfilled by the semantic model of the program.

4.1 A Semantic Model of Program Execution

In the rest of the article, we assume sets $\mathcal{C}$ (programs), *Var* (program variables), I (input channels), O (output channels), and *Val* (values). Snapshots of a program in execution are modeled by *configurations* $\langle c, s \rangle$ which consist of a program $c \in \mathcal{C}$ (or the special symbol ϵ modeling termination) and a memory state $s : Var \to Val$ (assigning a value to each variable in *Var*). The set of all configurations is denoted by $Conf = \mathcal{C} \times (Var \to Val)$.

Program execution is modeled by a transition relation on configurations: $\twoheadrightarrow \subseteq Conf \times Conf$. This transition relation is split into the sub-relations: $\twoheadrightarrow_O$ and $(\twoheadrightarrow_{ch,v})_{ch \in I \cup O,\ v \in Val}$, i.e., $\twoheadrightarrow = \twoheadrightarrow_O \ \cup \bigcup_{ch,v} \twoheadrightarrow_{ch,v}$. A relation $\twoheadrightarrow_{ch,v} \subseteq Conf \times Conf$ specifies the steps with input or output of value v on channel

ch. The relation $\rightarrow_{\mathrm{O}} \subseteq \mathit{Conf} \times \mathit{Conf}$ specifies execution steps without I/O, i.e., *ordinary steps*. If the special symbol ϵ occurs in a configuration instead of a program then this is a final configuration. We assume that all transition relations are deterministic. The only exceptions to this assumption are the values of input steps because the environment chooses the value v. In contrast, the channel ch is completely determined by the source configuration.

As notational conventions for the rest of the article, we denote meta-variables for elements of $\mathcal{D}$ by d, of $\mathcal{R}$ by r, of $\mathcal{C}$ by c, of Var by x, of $\mathit{Var} \to \mathit{Val}$ by s and t, of $\mathcal{E}$ by exp and b, of I by in, of O by out, of $I \cup O$ by ch, and of Val by v, all possibly with indices or primes. We also assume a policy $(D, \mathit{dom}, \leq, G)$.

4.2 A Novel Security Condition for Explicit Reference Points

We define the security condition based on the idea underlying *non-interference*, i.e., that the observations of an attacker must not depend on secret data.

Attacker Model. For each security domain $d \in D$, we assume a d-observer who can see the values of variables x with $\mathit{dom}(x) \leq d$. Hence, he can distinguish two memory states if they differ in the value of at least one d-observable variable.

Definition 3. *For a given domain d, two memory states s and s' are* d-equal, *denoted by $s =_d s'$, iff* $\forall x \in \mathit{Var}.\ (\mathit{dom}(x) \leq d \Rightarrow s(x) = s'(x))$.

Accordingly, we assume that a d-observer can see which values are input and output on channels ch with $\mathit{dom}(ch) \leq d$, i.e, he can distinguish two communication steps on ch if different values are transmitted. He can also distinguish communication steps on such a *d-observable* channel from communication steps on other channels as well as from ordinary steps. Otherwise, he can distinguish two computation steps only if he can distinguish the two corresponding memory states before or after the step. We define a sub-relation of $\rightarrow$ capturing the computation steps that do not communicate on d-observable channels by $\rightarrow_{\not\leq d=} (\rightarrow \setminus (\bigcup_{\mathit{dom}(ch) \leq d, v} \rightarrow_{ch,v}))$. This relation and the assumptions about d-observability of steps will be relevant when defining the security condition.

In addition to values of variables that he can directly observe, a d-observer may learn further information about memory states due to permissible declassifications. We represent what may be declassified by hatches. A *hatch* is a pair (exp, d) consisting of an expression exp and a security domain d. A hatch (exp, d') with $d' \leq d$ gives any d-attacker the possibility to peek at the value of exp through this hatch. Given a set $H \subseteq \mathcal{E} \times D$, a d-observer may distinguish two d-equal memory states only if they differ in the value of at least one expression exp for which there is a hatch $(\mathit{exp}, d') \in H$ with $d' \leq d$.

Definition 4. *For a given domain d and a given set of hatches $H \subseteq \mathcal{E} \times D$, two memory states s and s' are* (d, H)-equal, *denoted by $s =_d^H s'$, iff*

- $s =_d s'$ *and*
- $\forall (\mathit{exp}, d') \in H.\ [d' \leq d \Rightarrow \forall v \in \mathit{Val}.\ (\langle\!\langle \mathit{exp}, s \rangle\!\rangle \downarrow v \Leftrightarrow \langle\!\langle \mathit{exp}, s' \rangle\!\rangle \downarrow v)]$.

Here $\langle\!\langle \mathit{exp}, s \rangle\!\rangle \downarrow v$ denotes that exp evaluates to v in the memory state s, where we assume that the evaluation of expressions is total, atomic, and unambiguous.

From Guards to Hatches. Note that d-equality (Definition 3) captures what a d-attacker cannot observe while (d, H)-equality (Definition 4) captures what must be kept secret from him. The two notions of equality coincide if H is empty. If program execution starts with an empty set, then this means that attackers must not learn more than what they can directly see. This requirement is relaxed, whenever the run reaches a point referred to by a reference label r. For each guard $(r, exp, d') \in G$ in the policy, a hatch (exp, d') is added to the current set of hatches. This means that a d-observer may learn from now on the value of exp given that $d' \leq d$ holds. This is exactly what we had in mind when we introduced the notion of declassification guards with explicit reference points.

We use a function $ah : \mathcal{C} \times \mathfrak{P}(\mathcal{R} \times \mathcal{E} \times D) \to \mathfrak{P}(\mathcal{E} \times D)$ to formalize which hatches are added for a given step. If the program c has no top-level reference label, then we have $ah(c, G) = \emptyset$. Otherwise, if r is the top-level reference label of c, then $ah(c, G)$ contains each hatch (exp, d) for which $(r, exp, d) \in G$. We will define a concrete instance of ah in Section 5.1.

Maintaining Hatches. In order to obtain an adequate security condition, it does not suffice to merely add the right hatches whenever a reference point is reached. It is also necessary to identify all hatches that are invalidated by a computation step. For instance, if the current set of hatches is $H = \{((h_1 + h_2), \mathsf{public})\}$, then the assignment $h_2{:=}0$ invalidates the hatch $((h_1 + h_2), \mathsf{public})$ because any subsequent evaluation of $h_1 + h_2$ could reveal the value of h_1. This is not a permissible declassification, unless (h_2, public) were also in the set of hatches.

We use a function $ih : \mathcal{C} \times \mathfrak{P}(\mathcal{E} \times D) \to \mathfrak{P}(\mathcal{E} \times D)$ to capture the invalidation of hatches. For a program c and a set of hatches H, $ih(c, H)$ is the subset of all hatches in H that are not invalidated by the next computation step of c. We will define a concrete instance of ih in Section 5.1.

Capturing Secure Flow. Intuitively, a program c has secure information flow for a policy $(D, dom, \leq, G)$ if attackers cannot learn information about the initial state and about inputs that they are not authorized to obtain. That is,

> if c is run in d-equal states s_0 and s_0' then a d-observer must see the same values on d-observable output channels and in d-observable variables, given that no declassification occurs (e.g., $G = \emptyset$) and that, in the two runs, the same values are provided on all d-observable input channels.

If declassification can occur, then the setting is somewhat more complicated because permitting declassification means to relax the indistinguishability requirement to some extent. In particular, when a reference point r is passed in a run, some previously secret values might become declassifiable (as determined by the guards in G with reference point r). Given a set of hatches H (determining what may be revealed in addition to what can be observed), the requirement is

> if c is run in two (d, H)-equal states s_0 and s_0' and if the same values are provided on d-observable input channels, then, by observing d-observable variables and output channels, a d-observer must not learn any information beyond what he can already observe and beyond what he is permitted to learn by H and by hatches originating during the run.

$$F_{\text{obs}} \equiv \quad \forall ch, v.\ \left[\begin{array}{l}(\langle c_1, s\rangle \rightarrow_{ch,v} \langle c_2, t\rangle \wedge dom(ch) \leq d)\\ \implies \exists c_2', t'.\ (\langle c_1', s'\rangle \rightarrow_{ch,v} \langle c_2', t'\rangle \wedge F_{\text{concl}})\end{array}\right]$$

$$F_{\text{noobs}} \equiv \langle c_1, s\rangle \rightarrow_{\not\leq d} \langle c_2, t\rangle \Rightarrow \left[\begin{array}{l}\exists c_2', t'.\ \langle c_1', s'\rangle \rightarrow_{\not\leq d} \langle c_2', t'\rangle\\ \wedge\ \forall c_2', t'.\ (\langle c_1', s'\rangle \rightarrow_{\not\leq d} \langle c_2', t'\rangle \Rightarrow F_{\text{concl}})\end{array}\right]$$

$$F_{\text{concl}} \equiv \forall H_{\text{new}}.\ \left[\begin{array}{l}H_{\text{new}} = ih(c_1, H \cup (ah(c_1, G) \cap ah(c_1', G)))\\ \qquad\quad \cap ih(c_1', H \cup (ah(c_1, G) \cap ah(c_1', G)))\\ \implies (c_2\ R^{H_{\text{new}}}\ c_2' \wedge t =_d^{H_{\text{new}}} t')\end{array}\right]$$

Fig. 1. The Subformulas used in Definition 5

For the definition of our security condition, we use the PER-approach [10]. We define indistinguishability relations on configurations as products of partial equivalence relations on programs and the (d, H)-equality on memory states. More precisely, we characterize a family $(R^H)_{H \subseteq \mathcal{E} \times D}$ of partial equivalence relations (PERs) on programs. If two programs are related by some R^H in such a family, then running these programs in two (d, H)-equal memory states does not reveal any information to a d-observer that he is not authorized to obtain. Note that a relation R^H might not be reflexive, because programs that leak secrets cannot be related to themselves. Given that the requirements for the family of partial equivalence relations are properly defined, one obtains a definition of security by saying that a program c is secure if $c\ R^{\emptyset}\ c$ holds.

Partial Equivalences on Programs. The parameter H captures which values have been declassified in the past. If H is the current set of hatches and $c_1\ R^H\ c_1'$ holds, then performing a computation step in two (d, H)-equal memory states, respectively, must not leak any secrets. However, the two steps may reveal information that has been termed declassifiable in the past (captured by H) and about values that may be declassified due to guards that point to c_1 as well as to c_1' (captured by $ah(c_1, G)$ and $ah(c_1', G)$, respectively).

Definition 5. *A* strong (d, G)-bisimulation *is a family $(R^H)_{H \subseteq \mathcal{E} \times D}$ of relations R^H on $\mathcal{C}$ that are symmetric such that the following formula is satisfied for all $H \subseteq \mathcal{E} \times D$ (where F_{obs} and F_{noobs} are defined in Figure 1):*

$$F_{\text{main}} \equiv \begin{array}{l}\forall c_1, c_1', c_2.\\ \forall s, s', t.\end{array}\ \left[\begin{array}{l}\left(c_1\ R^H\ c_1' \wedge s =_d^{H \cup (ah(c_1, G) \cap ah(c_1', G))} s'\right)\\ \implies \left(\begin{array}{l}F_{\text{obs}} \wedge F_{\text{noobs}} \wedge ah(c_1, G) = ah(c_1', G)\\ \wedge\ ih(c_1, H \cup (ah(c_1, G) \cap ah(c_1', G)))\\ \quad = ih(c_1', H \cup (ah(c_1, G) \cap ah(c_1', G)))\end{array}\right)\end{array}\right]$$

The left hand side of the implication in F_{main} restricts c_1, c_1', s, and s' by $c_1\ R^H\ c_1'$ and $s =_d^{H \cup (ah(c_1, G) \cap ah(c_1', G))} s'$. The rest of F_{main} captures that a computation step in $\langle c_1, s\rangle$ cannot lead to undesired information leakage. Within F_{main}, the sub-formula F_{concl} occurs only on the right hand side of implications (within F_{obs} and F_{noobs} that will be explained below). Within F_{concl}, the set H_{new} captures which values may be declassified in future steps. The set H_{new} results

from H by adding new hatches (determined by ah) and by deleting invalidated hatches (according to ih). The propositions $c_2\ R^{H_{\text{new}}}\ c_2'$ and $t =_d^{H_{\text{new}}} t'$ ensure that no information will be leaked to d-observers in the future, other than what they can already see in the current state or what may be declassified to them (as specified by H_{new}). Naturally, it is crucial that the functions ah and ih are defined with care. In particular, H_{new} must not be too large.

Had we restricted ourselves to programs without I/O in this article, then it would suffice to use F_{concl} as the right hand side of the implication in F_{main}. However, we decided to tackle a more realistic program model, which supports I/O operations. Consequently, the definition of indistinguishability must additionally ensure (1) that inputs on channels that are not d-observable are kept secret from d-observers and (2) that transmissions on d-observable channels do not reveal any secrets. This is the purpose of the formulas F_{noobs} and F_{obs}. If the step $\langle c_1, s\rangle \rightarrowtriangle \langle c_2, t\rangle$ causes the transmission of a value v on a d-observable channel ch, then the step from $\langle c_1', s'\rangle$ must also transmit v on ch (captured by F_{obs}). Formula F_{noobs} is slightly more involved. If the step $\langle c_1, s\rangle \rightarrowtriangle \langle c_2, t\rangle$ does not cause any d-observable transmission, then the step from $\langle c_1', s'\rangle$ must not cause any d-observable transmission either. Note that it would not suffice to require only that F_{concl} holds for at least one $\langle c_2', t'\rangle$ with $\langle c_1', s'\rangle \rightarrowtriangle_{\not\approx d} \langle c_2', t'\rangle$, because this requirement would be too weak. Different steps are possible in $\langle c_1', s'\rangle$ if input is expected on some channel because the environment chooses the value (recall Section 4.1). Hence, the quantification over all possible steps is needed. Note also, that d-observable input is covered by F_{obs}, while F_{noobs} covers input on channels that are not d-observable.

A Novel Security Condition. As usual for the PER-approach, we define the security condition via the largest reflexive sub-relation on programs.

Theorem 1. *The set of all (d, G)-bisimulations has a maximal element under the point-wise subset ordering. We denote this maximal element by $(\approx_d^H)_{H \subseteq \mathcal{E} \times D}$.*

Definition 6. *A program c has secure information flow for a security policy $(D, dom, \leq, G)$ if $c \approx_d^{\emptyset} c$ holds for all $d \in D$. For a given policy, we also say that c* is secure while respecting explicit reference points *(brief: c* is WERP*).*

Note that $\emptyset$ occurs as super-script of $\approx_d$ in Definition 6. This reflects that, before program execution begins, no values are declassifiable. The set of hatches becomes non-empty as soon as a state is reached that is referred to by some guard in the security policy. In particular, it is possible that a guard refers to the top-level program, i.e., initial reference points are supported.

In the PER-approach, the adequacy of a security condition follows directly from the adequacy of the strong bisimulation relation on programs. In our presentation, we have derived the definition of strong (d, G)-bisimulations in a stepwise manner and argued in detail for the various elements in formula F_{main} in Definition 5. The following theorem shall provide further confidence in our novel security condition WERP (formalized by Definition 6).

Intuitively, the theorem states that, if a program is WERP and contains no output commands, then running this program cannot reveal any differences

between d-equal memory states or about input on d-invisible channels, unless a reference point is passed and some guard $(r, exp, d') \in G$ allows a d-observer to distinguish the corresponding intermediate states in the two runs.

Theorem 2. *Let* $m, n,\ c_0, c_1, \ldots, c_n,\ c'_1, \ldots, c'_n,\ s_0, s_1, \ldots, s_n,\ s'_0, s'_1, \ldots, s'_n,$ $in_1, in_2, \ldots, in_m,$ *and* $v_1, v_2, \ldots, v_m,\ v'_1, v'_2, \ldots, v'_m$ *such that* $m < n$ *and*

$$\langle c_0, s_0\rangle \twoheadrightarrow_{\mathrm{O}} \langle c_1, s_1\rangle \twoheadrightarrow_{\mathrm{O}} \ldots \twoheadrightarrow_{\mathrm{O}} \langle c_{i_1}, s_{i_1}\rangle \twoheadrightarrow_{in_1,v_1} \ldots \twoheadrightarrow_{\mathrm{O}} \langle c_{i_2}, s_{i_2}\rangle \twoheadrightarrow_{in_2,v_2} \ldots \twoheadrightarrow_{\mathrm{O}} \langle c_n, s_n\rangle$$
$$\langle c_0, s'_0\rangle \twoheadrightarrow_{\mathrm{O}} \langle c'_1, s'_1\rangle \twoheadrightarrow_{\mathrm{O}} \ldots \twoheadrightarrow_{\mathrm{O}} \langle c'_{i_1}, s'_{i_1}\rangle \twoheadrightarrow_{in_1,v'_1} \ldots \twoheadrightarrow_{\mathrm{O}} \langle c'_{i_2}, s'_{i_2}\rangle \twoheadrightarrow_{in_2,v'_2} \ldots \twoheadrightarrow_{\mathrm{O}} \langle c'_n, s'_n\rangle$$

If c_0 *is WERP,* $s_0 =_d s'_0$, *and* $\forall j \in 1, \ldots, m.\ (dom(in_j) \leq d \Rightarrow v_j = v'_j)$, *but* $s_n \neq_d s'_n$, *then there are* $i \in \{0, \ldots, n\}$, $d' \leq d$ *and* $(exp, d') \in ah(c_i, G)$ *such that the value of* exp *in* s_i *differs from the one in* s'_i.

Theorem 2 can be generalized to programs with output.

5 Security Type System and Soundness

Security type systems provide a suitable basis for automating an information flow analysis. We illustrate how a sound security type system for WERP can be derived for an exemplary programming language with I/O.

5.1 Exemplary Programming Language

We investigate a simple while-language (WL). Below, we present a grammar for three sub-languages: *uc* (the commands that may be annotated with reference labels), *lc* (annotated and non-annotated commands), and *c* (the entire WL).

$$\begin{aligned} uc &::= \mathsf{skip} \mid x{:=}exp \mid exp \text{ -> } out \mid x \text{ <- } in \mid \mathsf{if}\ b\ \mathsf{then}\ c\ \mathsf{else}\ c\ \mathsf{fi} \mid \mathsf{while}\ b\ \mathsf{do}\ c\ \mathsf{od} \\ lc &::= r\ :\ uc \mid uc \\ c &::= lc \mid c\ ;\ c \end{aligned}$$

The operational semantics of WL instantiates the step relations. Output commands *exp* -> *out* result in an output step $\twoheadrightarrow_{out,v}$, where v is the value of *exp* in the current memory state. Input commands x <- *in* result in an input step $\twoheadrightarrow_{in,v}$, where v can be any value. Reference labels are irrelevant for the operational semantics, i.e., $\langle r : c, s\rangle \twoheadrightarrow_{lab} \langle c', s'\rangle$ if $\langle c, s\rangle \twoheadrightarrow_{lab} \langle c', s'\rangle$, where *lab* is O or ch, v. We omit the formal definition of the operational semantics which is similar to the one in [11].

Instantiation of ah and ih for WL. We instantiate the functions *ah* and *ih* from Section 4.2 for our language WL. We inductively define the function $ah : \mathcal{C} \times \mathfrak{P}(\mathcal{R} \times \mathcal{E} \times D) \to \mathfrak{P}(\mathcal{E} \times D)$ for a given set G. Firstly, $ah(uc, G) = \emptyset$ and $ah(r : uc, G) = \{(exp, d) \in \mathcal{E} \times D \mid (r, exp, d) \in G\}$ for all *uc*. Secondly, $ah(c_1;c_2, G) = ah(c_1, G)$ for all c_1, c_2, because the first execution step of a sequentially composed command corresponds to the first step of its first component. Therefore, the first component determines the set of additional hatches.

We assume a function $vars : \mathcal{E} \to \mathfrak{P}(Var)$, such that $vars(exp)$ contains all variables on which the value of *exp* depends. The instantiation of $ih : \mathcal{C} \times \mathfrak{P}(\mathcal{E} \times$

$$\frac{\forall x \in vars(exp).\ dom(x) \leq d}{H' \vdash\ exp : d} \qquad \frac{(exp, d) \in H'}{H' \vdash\ exp : d} \qquad \frac{H' \vdash exp : d \quad d \leq dom(x)}{H' \vdash\ x{:=}exp : ih(x{:=}exp, H')}$$

$$\frac{}{H' \vdash \mathsf{skip} : H'} \qquad \frac{dom(in) \leq dom(x)}{H' \vdash\ x \texttt{ <- } in : ih(x \texttt{ <- } in, H')} \qquad \frac{H' \vdash exp : d \quad d \leq dom(out)}{H' \vdash\ exp \texttt{ -> } out : H'}$$

$$\frac{H' \cup \{(exp, d) \in \mathcal{E} \times D | (r, exp, d) \in G\} \vdash\ c : H_\epsilon}{H' \vdash\ (r\ : c) : H_\epsilon} \qquad \frac{H' \vdash\ c_1 : H'' \quad H'' \vdash\ c_2 : H_\epsilon}{H' \vdash\ c_1\,;c_2 : H_\epsilon}$$

$$\frac{H' \vdash\ B : low \quad H' \vdash\ c : H'}{H' \vdash \mathsf{while}\ b\ \mathsf{do}\ c\ \mathsf{od} : H'} \qquad \frac{H' \vdash\ c_1 : H_\epsilon \quad H' \vdash\ c_2 : H_\epsilon \quad H' \vdash B : low}{H' \vdash \mathsf{if}\ b\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2\ \mathsf{fi} : H_\epsilon}$$

Fig. 2. Rules of the Security Type System

$D) \to \mathfrak{P}(\mathcal{E} \times D)$ for WL invalidates a hatch (exp, d) if some variable in $vars(exp)$ might be modified by the next execution step. We inductively define ih by

$$\begin{array}{ll}
ih(c_1;c_2, H) = ih(c_1, H), & \\
ih(r : uc, H) = ih(uc, H), & \\
ih(uc, H) = \{(exp', d) \in H \mid x \notin vars(exp')\} & \text{if } uc = x{:=}exp \text{ or } uc = x \texttt{ <- } in, \\
ih(uc, H) = H & \text{otherwise.}
\end{array}$$

5.2 Security Type System

With respect to declassification the two main objectives of the type system are, firstly, to identify at which subprogram which set of hatches represents information that may be declassified, and, secondly, to ensure that each command has secure information flow, given the set of hatches for this command.

The type system defines judgments $H' \vdash\ c : H_\epsilon$ for commands. The sets of hatches help to achieve the first objective. The set H_ϵ is the set of declassifiable hatches when c stops, if we assume H' is the set of declassifiable hatches when c starts. Hence, H_ϵ is the set for the direct successor of c, if the successor exists.

The type system defines judgments $H' \vdash\ exp : d$ for expressions. A judgment $H' \vdash\ exp : d$ guarantees that the value of exp only depends on variables that are visible to d, or that H' specifies that the value may be learned by the d-observer. We exploit the guarantees to ensure secure information flow from exp by comparing d to the security domains of potential targets of information flow.

The rules to derive the judgments are defined in Figure 2. We assume $low \in D$ such that $\forall d.\ low \leq d$. In judgments $H' \vdash\ c : H_\epsilon$ for labeled commands and commands that write to variables, i.e. input commands and assignments, H_ϵ is modified in comparison to H'. In the first case hatches are added as determined by guards with the label of the respective command. In the latter case the function ih is applied. The type rules for loops and conditionals require the branching condition to be typable with security domain low, because this means the value of the condition may be learned by anyone. How to define a more fine-grained syntactic requirement that takes into account a comparison of the branches is demonstrated in [2]. We omit such a requirement due to space limitations.

Theorem 3 (Soundness). *If* $\emptyset \vdash c : H_\epsilon$ *for some* H_ϵ *then* c *is WERP.*

This is the soundness result for the security type system.

6 Applying the Security Type System

We illustrate the capabilities of our novel framework by applying the security type system to example programs.

Example 1. We revisit the example about the average of 100 salaries, i.e. the programs P_3', P_6', and P_8. Let $(D, dom, \leq, G)$ be the security policy denoted by

SecurityDomains secret; **DomainAssign** sal_1:secret; ... ; sal_{100}:secret; input:secret; **RegularFlow** public=>secret; **ExceptionalFlow** dguard(r, Avg, public); **EndPolicy**,

where $Avg = (\mathsf{sal}_1 + \ldots + \mathsf{sal}_{100})/100$. The reference label r is chosen differently for the programs as argued in Section 2.2. Let $H = \{(Avg, \mathsf{public})\}$.

First we consider the program P_6', which reads the salary values from the channel input, and, intuitively, is secure. Here $r = \mathsf{ref}_{101}$. We derive $H \vdash Avg : \mathsf{public}$, $H \vdash \mathsf{avg}{:=}Avg : H$, and $\emptyset \vdash (\mathsf{ref}_{101} : \mathsf{avg} := Avg) : H$ by the rules for expressions, assignments, and labeled commands. We derive $\emptyset \vdash \mathsf{sal}_i \text{ <- input} : \emptyset$ for all $i \in \{1 \ldots 100\}$ by the rule for input commands. We derive $\emptyset \vdash P_6' : H$ by the rule for sequential composition. That is the type system accepts P_6'.

Now we consider P_3' with $r = \mathsf{ref}_1$. The judgment $\emptyset \vdash Avg : \mathsf{public}$ is not derivable by any of the two rules for expressions. Hence, $\emptyset \vdash \mathsf{avg}{:=}Avg : H_\epsilon$ is not derivable for any H_ϵ. However, this is a precondition to derive $\emptyset \vdash \mathsf{ref}_{101} : \mathsf{avg}{:=}Avg : H_\epsilon$, because $\{(r', exp, d) \in G | r' = \mathsf{ref}_{101}\} = \emptyset$. For any derivation of $H' \vdash \mathsf{sal}_{100} \text{ <- input} : H_\epsilon$ we have $(Avg, \mathsf{public}) \notin H_\epsilon$, because $\mathsf{sal}_{100} \in vars(Avg)$. Hence, for P_3' the rule for sequential composition does not apply. For P_8, where $r = \mathsf{ref}_{101}$, we argue as for P_3'. The type system does not accept P_3' or P_8.

The novel type system classifies the three programs exactly as we intended.

Example 2. In an example from [4] an electronic wallet stores an amount of money in the wallet (variable h), and an amount spent so far (variable l). An amount to spend (variable k) is moved from the wallet to the money spent so far if enough money is in the wallet. We consider an interactive variant with an input channel toSpend for the amount to spend, and an output channel loyalty to a customer loyalty application. The amount of money in the wallet is a secret, i.e., h must not leak to loyalty. However, it is inherent in the functionality of the program that the loyalty application obtains whether money is spent, i.e., whether the amount in the wallet is enough for the amount to spend. Hence, one decides to exceptionally permit this flow.

```
Spec = SecurityDomains secret; DomainAssign h:secret; RegularFlow public=>secret;
       ExceptionalFlow dguard(ref, (h >= k), public); EndPolicy
P9   = while True do
         k <- toSpend;
         ref : if (h >= k) then h:=h-k ; l:=l+k ; l -> loyalty else skip fi
       od
```

We set the reference point ref right behind the input of the amount to spend, because here the value to be declassified originates. Since the reference label is at the conditional and the condition is the expression of the guard, P_9 is typable.

The example demonstrates the typability in cases, where the declassifiable information depends on fresh input in each iteration of a loop and on calculations.

7 Related Work

Comparison of Analysis. We compare the application of our security type system in Section 6 with the application of security type systems from the literature in order to compare our framework to frameworks that implicitly assume initial (delimited release [4]) or local reference points (delimited non-disclosure [5]).

The security type system in [4] accepts declassification of expression that are escape hatches. In order to detect expressions that are declassified after they are updated, for each command two sets of variables are collected, one for variables that are updated and one for variables that appear in declassifiable expressions.

The security type system in [5] associates declassifiable information with the security domain *low* at program points within the declassification statement.

Both security type systems are defined for languages without explicit I/O-commands. In order to enable a comparison, without any claims on the soundness, we assume a straightforward extension of the security type systems from [4,5] that treats I/O-commands similar to assignments.

We consider the programs P_3, P_5, P_6, P_7, and the following programs:

$$\begin{aligned} P_8' &= \mathsf{sal}_1 \text{ <- input; } \ldots; \mathsf{sal}_{100} \text{ <- input;} \\ &\quad \mathsf{sal}_1{:=}\mathsf{sal}_1; \ldots; \mathsf{sal}_{100}{:=}\mathsf{sal}_1; \mathsf{avg}{:=}\mathsf{declassify}(Avg,\mathsf{public}) \\ P_8'' &= \mathsf{declassify}\ (Avg)\ \mathsf{in}\ \{\mathsf{sal}_1 \text{ <- input; } \ldots; \mathsf{sal}_{100} \text{ <- input;} \\ &\quad \mathsf{sal}_1{:=}\mathsf{sal}_1; \ldots; \mathsf{sal}_{100}{:=}\mathsf{sal}_1; \mathsf{avg}{:=}Avg\} \end{aligned}$$

Let the program P_9' be P_9 with an escape hatch as condition of the conditional. Let P_9'' be P_9 with a declassification command around the conditional.

We list the results of applying the security type systems in Table 1. The rows represent the programs and the columns represent the security type systems for the security conditions named at the head. Our security type system is more precise than the security type system for delimited release [4], for instance in the wallet example where some input is not available at the start of the program. Our security type system is stricter than the security type system for delimited non-disclosure [5], for instance for the programs that leak sal_1. Hence this strictness is desirable for these programs.

Table 1. Typability of Programs ("x" means "typable", "-" "not typable")

	WERP	delimited release	delimited non-disclosure	Remark
avg. with copying sal_1	- (P_3')	- (P_3)	x (P_5)	leaks secret
avg. with input	x (P_6')	- (P_6)	x (P_7)	-
avg. with both	- (P_8)	- (P_8')	x (P_8'')	leaks secret
wallet	x (P_9)	- (P_9')	x (P_9'')	-

Related Approaches. Many prior frameworks for controlling what is declassified implicitly assume that reference points are either always local or always

Localized delimited release [8] requires that an expression is only declassified after a declassification expression for this expression has appeared in the code. The information that may be declassified is the initial value of the expression. Policies for the security conditions WHAT_1, WHAT_2 [7] specify a set of pairs of hatches externally to the program. For $SIMP^*_D$ [6] just a set of expressions is specified. These three conditions are based on step-wise bisimulations. Unlike for WERP, the set of declassifiable expressions is fixed over steps, i.e. only the initial values of the expressions may be declassified. In *conditioned gradual release* [9], commands can be annotated with a *flowspec*, which is a triple of a formula on program variables, a set of expressions, and a variable. Declassification is only permitted if the current command is annotated with a *flowspec* whose formula is satisfied. Moreover, only the local value of an expression in the specified set may be declassified, and only to the variable specified in the *flowspec*.

Complementing control of *what*, control of other *dimensions of declassification* [2,3] has been developed. Examples for control of where declassification can occur are *intransitive noninterference* [2], WHERE [7], *gradual release* [12], *nondisclosure* [13], and *flow locks* [14]. Examples for control of who can declassify are the *decentralized label model* [15], WHERE&WHO [11], and *robustness* [16].

The security conditions *gradual release* and *conditioned gradual release* are based on characterizations of deducible knowledge. Given the observation about a run up to some point, the set of initial states that might have caused this observation becomes known. In contrast, in a PER-based condition like WERP, the indistinguishability of memory states represents the deducible information. The PER-approach facilitates the collection of values that may be declassified on the fly. Once information is represented in the indistinguishability relation, the origin, i.e. some intermediate state, does not matter anymore. It remains to be investigated how this would be done best in a knowledge-based approach. In a recent article [17], input from channels is treated in a knowledge-based fashion by modelling input as streams that are part of the initial state. Still, declassification of values that are calculated on the fly, like, e.g., in Example 2, is not captured by this approach.

8 Conclusion

We developed a framework that permits to control what information may be declassified by declassification guards with *explicit* reference points. The framework comprises a policy language, a security condition, and a sound security type system. We illustrated the benefits of our framework with several concrete example programs. In comparison to earlier approaches, our framework allows one to characterize more precisely what may be declassified.

Explicit reference points clarify the implicit differences between the aforementioned existing approaches to controlling the what dimension. However, our framework goes beyond providing a uniform view and a straightforward

combination of prior approaches. Reference points can be placed anywhere in a program. In particular, they can refer to where declassifiable information originates, even if the information is not immediately released.

We expect that an integration of WERP will be feasible with approaches to controlling other dimensions of declassification, in particular if the respective other security condition is also defined based on a step-wise bisimulation relation with the PER-approach (like, e.g., WHERE [7] or WHERE&WHO [11]).

Acknowledgments. We thank Henning Sudbrock for helpful comments and 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, and supported by CASED (www.cased.de). This article reflects only the authors' views, and CASED, the Commission, the DFG, and the authors are not liable for any use that may be made of the information contained therein.

References

1. Goguen, J.A., Meseguer, J.: Security Policies and Security Models. In: 3rd IEEE Symposium on Security and Privacy, pp. 11–20. IEEE Computer Society Press, Los Alamitos (1982)
2. Mantel, H., Sands, D.: Controlled Declassification based on Intransitive Noninterference. In: Chin, W.-N. (ed.) APLAS 2004. LNCS, vol. 3302, pp. 129–145. Springer, Heidelberg (2004)
3. Sabelfeld, A., Sands, D.: Dimensions and Principles of Declassification. In: 18th IEEE Computer Security Foundations Workshop, pp. 255–269. IEEE Computer Society Press, Los Alamitos (2005)
4. Sabelfeld, A., Myers, A.C.: A Model for Delimited Information Release. In: ISSS 2004, pp. 174–191. Springer, Heidelberg (2004)
5. Barthe, G., Cavadini, S., Rezk, T.: Tractable Enforcement of Declassification Policies. In: 21st IEEE Computer Security Foundations Symposium, pp. 83–97. IEEE, Los Alamitos (2008)
6. Bossi, A., Piazza, C., Rossi, S.: Compositional Information Flow Security for Concurrent Programs. Journal of Computer Security 15(3), 373–416 (2007)
7. Mantel, H., Reinhard, A.: Controlling the What and Where of Declassification in Language-Based Security. In: De Nicola, R. (ed.) ESOP 2007. LNCS, vol. 4421, pp. 141–156. Springer, Heidelberg (2007)
8. Askarov, A., Sabelfeld, A.: Localized Delimited Release: Combining the What and Where Dimensions of Information Release. In: Workshop on Programming Languages and Analysis for Security, pp. 53–60. ACM Press, New York (2007)
9. Banerjee, A., Naumann, D.A., Rosenberg, S.: Expressive Declassification Policies and Modular Static Enforcement. In: 29th IEEE Symposium on Security and Privacy, pp. 339–353. IEEE Computer Society Press, Los Alamitos (2008)
10. Sabelfeld, A., Sands, D.: A Per Model of Secure Information Flow in Sequential Programs. In: Swierstra, S.D. (ed.) ESOP 1999. LNCS, vol. 1576, pp. 50–59. Springer, Heidelberg (1999)

11. Lux, A., Mantel, H.: Who Can Declassify? In: Degano, P., Guttman, J., Martinelli, F. (eds.) FAST 2008. LNCS, vol. 5491, pp. 35–49. Springer, Heidelberg (2009)
12. Askarov, A., Sabelfeld, A.: Gradual Release: Unifying Declassification, Encryption and Key Release Policies. In: 28th IEEE Symposium on Security and Privacy, pp. 207–221. IEEE Computer Society Press, Los Alamitos (2007)
13. Almeida Matos, A., Boudol, G.: On Declassification and the Non-Disclosure Policy. In: 18th IEEE Computer Security Foundations Workshop, pp. 226–240. IEEE Computer Society Press, Los Alamitos (2005)
14. Broberg, N., Sands, D.: Flow Locks: Towards a Core Calculus for Dynamic Flow Policies. In: Sestoft, P. (ed.) ESOP 2006. LNCS, vol. 3924, pp. 180–196. Springer, Heidelberg (2006)
15. Myers, A.C., Liskov, B.: Protecting Privacy using the Decentralized Label Model. ACM Transactions on Software Engineering and Methodology 9(4), 410–442 (2000)
16. Zdancewic, S., Myers, A.C.: Robust Declassification. In: 14th IEEE Computer Security Foundations Workshop, pp. 15–23. IEEE Computer Society Press, Los Alamitos (2001)
17. Askarov, A., Sabelfeld, A.: Tight Enforcement of Information-Release Policies for Dynamic Languages. In: 22nd IEEE Computer Security Foundations Symposium. IEEE Computer Society Press, Los Alamitos (2009)