Controlled Declassification based on Intransitive Noninterference

Heiko Mantel[1] and David Sands[2]

[1] Information Security, ETH Zürich, Switzerland, Heiko.Mantel@inf.ethz.ch
[2] Chalmers University of Technology, Göteborg, Sweden, www.cs.chalmers.se/~dave

Abstract. Traditional noninterference cannot cope with common features of secure systems like channel control, information filtering, or explicit downgrading. Recent research has addressed the derivation and use of weaker security conditions that could support such features in a language-based setting. However, a fully satisfactory solution to the problem has yet to be found. A key problem is to permit exceptions to a given security policy without permitting too much. In this article, we propose an approach that draws its underlying ideas from *intransitive noninterference*, a concept usually used on a more abstract specification level. Our results include a new bisimulation-based security condition that controls tightly where downgrading can occur and a sound security type system for checking this condition.

1 Introduction

Research on secure information flow in a programming-language setting has flourished in recent years, with advances in both theory and practice [SM03b]. The basic problem is to determine: When can a program be trusted to access confidential data even when some of its actions can be observed by untrusted subjects? That is, information flow security goes beyond access control as it not only restricts the program's access to data but also the propagation of data within the program. In the simplest case there are two kinds of data: confidential (high) and public (low) data, where the only permitted flow of information is from low to high. In the general case there is a partial ordering of security levels (often a lattice [Den76]), representing different levels of security where the ordering relation expresses where information may legitimately flow.

Despite many recent advances in the theory and practice of secure information flow, some serious practical concerns remain. One problem is that secure information flow, although a worthy goal, is often an unrealistic one. The problem is that many programs must inevitably leak certain amounts of data.

For instance, a bank's IT system stores data about the financial transactions of its customers. Information about a transaction should be kept confidential to the parties involved and to authorized bank personnel. However, in the process of a criminal prosecution it might become necessary and allowable for a prosecutor to analyze the data about a particular customer in the bank's IT system. Hence, the system must be able to reveal the secret data about a customer to the prosecutor, which can be seen as a form of declassification. Naturally, this does not

Appeared in W.-N. Chin (Ed.): APLAS 2004, LNCS 3302, pp. 129–145, 2004.

mean that arbitrary parts of the system may be able to perform declassification. For example, there should be no danger that a procedure responsible for printing account statements causes declassification neither with a malicious intent nor due to a bug in the program. Rather, declassification should be limited to designated parts of a program whose execution can then be protected by an unmistakable request for a confirmation of the declassification or a request for another password (e.g. to implement a two-person rule). Moreover, permitting declassification does not mean that arbitrary data may be declassified. For example, it would not be allowable if data about transactions that are completely unrelated to the prosecuted customer were revealed to the prosecutor. In summary, one wants to tightly control *where* classification can occur in a program and *where* exceptions to the information flow ordering are permitted in the security policy. This is what *intransitive noninterference* [Rus92,Pin95,RG99,Man01,BPR04] provides.

Our goal here is to adapt intransitive noninterference to a programming-language setting. To our knowledge, this has not been achieved before. Prior work on language-based information flow security has addressed other aspects of declassification, namely controlling *what* or *how much* information is declassified [Coh78,VS00,SS01,CHM02,Low02,BP02,DHW02,SM03a,SM03b] and controlling *who* initiates the act of declassification [ZM01,Zda03,MSZ04]. These aspects are also important, but orthogonal to the localization of declassification in a program and in a security policy, i.e. the aspects that we investigate here.

Rather than re-inventing a new program analysis from scratch, we want to illustrate how an existing analysis technique (namely, the one developed in [SS00]) can be adapted for dealing with intransitive information flow. The specific contributions of our work are: (1) A state-based definition of security for simple concurrent programming languages that is suitable for information flow policies permitting forms of declassification and that exhibits good compositionality properties (Section 4). (2) A security type system for a toy language illustrating the utility of our security definition (Section 6). We illustrate our security definition also with several examples (in Section 5) and compare it to related work (in Section 7). Due to space restrictions, proofs of the main results are presented in an extended version of the paper.

2 Preliminaries

The definition of security to be introduced in the next section is formulated in terms of a "small-step" transition relation on commands and states. The specification of the transition relation is stratified into a deterministic layer for individual commands and a nondeterministic layer for command vectors.

Deterministic judgments have the form $\langle C, s\rangle \rightharpoonup \langle \vec{W}, t\rangle$ expressing that a thread with program C performs a computation step in state s, yielding a state t and a vector of programs $\vec{W}$ that has length zero if C terminated, length n if $n > 1$ threads were spawned, and length one, otherwise. That is, a program vector of length n can be viewed as a *pool of n threads* that run concurrently. *Nondeterministic judgments* have the form $\langle \vec{V}, s\rangle \rightarrow\!\!\!\!\!\rightharpoonup \langle \vec{W}, t\rangle$ expressing that some

thread C_i in the thread pool $\vec{V}$ performs a step in state s resulting in the state t and some thread pool $\vec{W}'$. The global thread pool $\vec{W}$ results then by replacing C_i with $\vec{W}'$.

We abstract from the details of scheduling and work with a purely nondeterministic semantics for thread pools. This has shortcomings in terms of security modeling as the way in which nondeterminism is resolved can be exploited to create additional covert channels (see [SS00]). We adopt this simplification to reduce technical overhead. However, we believe that it is not a fundamental limitation of our approach and that most of our results would carry over to a more scheduler-specific setting, i.e. one in which the scheduler is under the control of the attacker [SS00].

For the sake of concreteness, we introduce a minimalistic programming language that we will use for illustrating the basic principles. The language that we adopt is the multi-threaded while language (short: MWL) from [SS00]. It includes commands for assignment, conditional branching, looping, and dynamic thread creation. The complete set of commands $C \in \mathit{Com}$ is:

$$C ::= \mathsf{skip} \mid \mathit{Id} := \mathit{Exp} \mid C_1; C_2 \mid \mathsf{if}\ B\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2 \mid \mathsf{while}\ B\ \mathsf{do}\ C \mid \mathsf{fork}(C\vec{V})$$

In the following (and above), the metavariable C denotes an individual command, and $\vec{V}$, $\vec{W}$ denote command vectors. The set of all command vectors is defined by $\vec{\mathit{Com}} = \bigcup_{n\in\mathbb{N}} \mathit{Com}^n$. We assume a set *Var* of program variables and a (not further specified) set *Val* of values. *States* are mappings from variables to values being denoted by s or t. A *configuration* is a pair $\langle \vec{V}, s\rangle$ where $\vec{V}$ specifies the threads that are currently active and s defines the current *state* of the memory. *Expressions* are simply variables, constants, or binary operators applied to expressions. Our treatment of expressions is rather informal; we assume the existence of a subset of expressions that yield boolean results. These boolean expressions are ranged over by B. We use a judgement $\langle \mathit{Exp}, s\rangle \downarrow n$ for specifying that the expression *Exp* evaluates to the value n in state s. Expression evaluation is assumed to be total and to occur atomically. For reasons of space, the transition relation for MWL is omitted in this version of the paper.

The operational semantics require an implementation that executes small-step transitions atomically.

3 Strong Security for Multi-level Policies

The *strong security condition* has been defined in [SS00] for a security policy with a high and a low domain. In the following, we generalize this definition to multi-level security policies with an arbitrary number of security domains.

Definition 1 (MLS Policy). *A* multi-level security policy *is a pair* $(\mathcal{D}, \leq)$ *where* $\mathcal{D}$ *is a set of security domains and* $\leq \subseteq \mathcal{D} \times \mathcal{D}$ *is a partial ordering.*

Definition 2 (Domain Assignment). *A* domain assignment *is a function* $\mathit{dom} : \mathit{Var} \to \mathcal{D}$ *that assigns a security domain to each program variable.*

In examples, we adopt the convention that names of variables reflect their domains. For example, in the context of a two-level security policy ($L \leq H$), l and h denote typical variables of the domains L and H, respectively.

Security will be defined with reference to the way that an observer with a given clearance (i.e. a security domain) observes the behavior of the system. We assume, in a standard manner, that an observer at domain D can see the values of variables at domain D, and at all domains below D. However, we focus not on this projection, but rather on the equivalence relation that it induces on states:

Definition 3 (D-equality). *Let $D \in \mathcal{D}$. Two states s_1 and s_2 are D-equal (denoted by $s_1 =_D s_2$) iff $\forall var \in Var : dom(var) \leq D \implies s_1(var) = s_2(var)$.*

The intuition is: if $s_1 =_D s_2$ then the states s_1 and s_2 are indistinguishable for an observer with clearance D.

In [SS00] a *strong low-bisimulation* relation captures when two command vectors are indistinguishable (from the low observer's perspective). This is an auxiliary definition that is used to define when a program is secure. Here we generalize the definition to the multi-level case:

Definition 4 (Strong D-bisimulation). *The* strong *D-bisimulation $\cong_D$ (for $D \in \mathcal{D}$) is the union of all symmetric relations R on command vectors $\vec{V}, \vec{V'} \in \vec{Com}$ of equal size, i.e. $\vec{V} = (C_1, \ldots, C_n)$ and $\vec{V'} = (C'_1, \ldots, C'_n)$, such that*

$$\begin{aligned}\forall s, s', t : \forall i \in \{1 \ldots n\} : \forall \vec{W} : \vec{V}\, R\, \vec{V'} \wedge s =_D s' \wedge \langle C_i, s\rangle \rightarrow \langle \vec{W}, t\rangle \\ \Rightarrow \exists \vec{W'}, t' : \langle C'_i, s'\rangle \rightarrow \langle \vec{W'}, t'\rangle \wedge \vec{W}\, R\, \vec{W'} \wedge t =_D t'\end{aligned}$$

Intuitively, two programs C and C' are strongly D-bisimilar ($C \cong_D C'$) iff their behavior on D-equivalent states is indistinguishable by an observer with clearance D. The definition is "strong" in a number of senses:

1. The programs must be indistinguishable for D not only if being run in identical starting states, but also if being run in different states that are D-equal.
2. The programs must not only be indistinguishable for D in their complete runs, but they must be strongly D-bisimilar after each computation step.
3. The relationship requires that threads match one-for-one and step-by-step.

The first property relates to the standard noninterference view: Being strongly D-bisimilar to itself means for a program that an observer can infer the starting state at most up to D-equality. That is, such a program cannot leak to domain D any information that D is not permitted to access. The second property is crucial for making the security condition compositional. This is related to the well-known fact that a (standard) compositional semantics cannot be given to the language if we only model the uninterrupted traces of computation. The third property is justified by the fact that the semantics remains sound for particular choices of the scheduler (see [SS00] for the details.) Note that $\cong_D$ is not reflexive as some programs yield different observations if run in D-equal starting states:

Example 1. For the two-level security policy $L \leq H$, the program $l := h$ is not L-bisimilar to itself. For instance, the states s (defined by $s(l) = 0$, $s(h) = 0$) and t (defined by $t(l) = 0$, $t(h) = 1$) are L-equal, but the states s' and t' resulting after $l := h$ is run in s and t, respectively, are not L-equal ($s'(l) = 0 \neq 1 = t'(l)$).

The security of a given program is then defined by using self-bisimilarity.

Definition 5 (Strong D-Security). *Let $Pol = (\mathcal{D}, \leq)$ be an MLS policy and $D \in \mathcal{D}$ be a security domain. A program $\vec{V}$ is* strongly secure for D *iff $\vec{V} \cong_D \vec{V}$.*

Strong D-security says that the program does not leak information to an observer with clearance D. Strong security is then defined by requiring this at all levels:

Definition 6 (Strong Security). *Let $Pol = (\mathcal{D}, \leq)$ be an MLS policy. A program $\vec{V}$ is* strongly secure *iff it is strongly secure for all $D \in \mathcal{D}$.*

4 Permitting Controlled Declassification

Declassification features are needed for many applications, but, unfortunately, they cause a fundamental problem for the security analysis: On the one hand, it shall be enforced that the flow of information complies with the given MLS policy, but on the other hand, violations of the policy should be tolerated. Obviously, these goals are conflicting. There is a wide spectrum of possible solutions, each of which involving some compromise. For instance, the strong security condition in the previous section is at the one end of the spectrum as it does not tolerate any exceptions to the MLS policy. A solution at the other end of the spectrum is employed in the Bell/La Padula model [BL76] where processes are considered as *trusted processes* if they involve downgrading and to exempt such processes from the formal security analysis. Such an analysis investigates the system *without* trusted processes although the system will always run *with* these processes.[1] Since the system being formally analyzed differs, it is not entirely clear which guarantees such a security analysis can provide for the system in operation.

Our goal here is to move away from such extreme solutions. We are aiming for a security condition that provides *formal* guarantees regarding the security of the system as it is in operation, but, nevertheless, also permits declassification under certain conditions. That is, we do not want to permit arbitrary downgrading but rather want to control tightly where downgrading can occur. In comparison to the trusted-processes approach, our aims differ as follows:

- Rather than localizing declassification in a program at the level of entire processes, we want to localize it at the level of individual commands.
- Rather than granting rights for arbitrary exceptions to a program (or parts thereof), we want to restrict exceptions to certain parts of the security lattice.

[1] The traditional terminology is somewhat confusing. The term *trusted processes* expresses that one *needs to trust* these processes for the overall system to be secure although they have only been informally inspected.

In other words, we aim for a finer-grained localization of declassification in the program as well as for a localization of declassification in the security policy. This is what we mean by controlling tightly *where* declassification can occur.

To this end, we modify the definition of a security policy such that it becomes possible to specify permissible exceptions, enrich the programming language with designated downgrading commands, and relax the strong security condition such that it becomes compatible with the extended language and the modified definition of security policies (and such that it meets our above goals).

4.1 Permitting Exceptions to MLS-Policies

We introduce a relation $\rightsquigarrow$ between security domains for expressing where exceptions to the information flow ordering $\leq$ are permitted: $D_1 \rightsquigarrow D_2$ means that information may flow from D_1 to D_2 even if $D_1 \leq D_2$ does not hold.

Definition 7 (MLS Policy with exceptions). *A* multi-level security policy with exceptions *is a triple* $(\mathcal{D}, \leq, \rightsquigarrow)$ *where* $\mathcal{D}$ *is a set of security domains,* $\leq \subseteq \mathcal{D} \times \mathcal{D}$ *is a partial ordering, and* $\rightsquigarrow \subseteq \mathcal{D} \times \mathcal{D}$ *is a relation.*

Adding the *information flow relation* $\rightsquigarrow$ to an MLS policy changes where information flow is permitted, but does not affect visibility. An observer can still only see the values of variables at his clearance and below (according to $\leq$). In particular, the definition of D-equality remains unchanged.

Note that the flow of information permitted by an MLS policy with exceptions does not constitute a transitive relation, i.e. neither $\leq \cup \rightsquigarrow$ nor $\rightsquigarrow$ is transitive, in general. For example, a standard example of such a policy is information flow via a trusted downgrader. Suppose we wish to allow information to flow from levels A to B, but only if it is vetted by a trusted downgrader (representing a physical audit, or a mechanical procedure such as encryption). This could be represented by the intransitive information flow relation $A \rightsquigarrow D \rightsquigarrow B$. As a second example, suppose we have a three level policy $\mathit{www} \leq \mathit{Employee} \leq \mathit{Webmaster}$ with a downgrading relation: Webmaster $\rightsquigarrow$ www; in this case it is the relation $\leq \cup \rightsquigarrow$ which is intransitive. Ultimately, the webmaster chooses what information is released to the web, which means that employee-level information can only be published to the web if the information passes through the webmaster.

4.2 Downgrading Commands

We extend our programming language with a downgrading command $[Id := Id']$. This command has the same effect as the usual assignment command $Id := Id'$, but it is handled differently in the security analysis. It may violate the information flow ordering $\leq$ as long as it complies with the information flow relation $\rightsquigarrow$. Differentiating between downgrading commands and assignment on the syntactic level allows a programmer to make explicit where he intends to downgrade information in a program. For the analysis, this makes it possible to distinguish intentional downgrading from accidental information leakage.

$$\frac{\langle\!\langle\, Id', s \,\rangle\!\rangle \downarrow n \quad dom(Id') = D_1 \quad dom(Id) = D_2}{\langle\!\langle\, [Id := Id'], s \,\rangle\!\rangle \rightarrow_d^{D_1 \to D_2} \langle\!\langle\, \langle\rangle, [Id = n]s \,\rangle\!\rangle}$$

Fig. 1. Downgrading transitions

We deliberately extend the language with a rather simple command for declassification. The motivation for this choice will be discussed in Section 5 where we will also illustrate how to program with this downgrading command.

Transitions that involve downgrading are highlighted in the operational semantics: The transition relation $\rightarrow$ is split into a relation $\rightarrow_o$ (the *ordinary transitions*) and a family $\rightarrow_d^{D_1 \to D_2}$ of relations (the *downgrading transitions*). The commands in Section 2 cause ordinary transitions, which means that $\rightarrow_o$ corresponds to $\rightarrow$ in the standard semantics. In contrast, $[Id := Id']$ causes downgrading transitions (as specified in Figure 1). The operational semantics are also extended with variants of the rules for sequential composition and for thread pools that propagate the annotations of downgrading transitions from the premise to the conclusion in the obvious way. From now on, transitions are either ordinary or downgrading transitions, i.e. $\rightarrow = \rightarrow_o \cup \bigcup_{D_1, D_2 \in \mathcal{D}} \rightarrow_d^{D_1 \to D_2}$.

4.3 A Modified Strong Security Condition

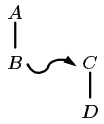
The strong security condition enforces that the flow of information complies with the information flow ordering and does not tolerate any exceptions. In order to cope with downgrading, we need to weaken this condition. To this end, we modify the notion of a strong D-bisimulation that underlies the strong security condition, and then use the modified notion to define a new security condition.

Intuitively, we want this condition to provide the following guarantees:

1. The strong security condition holds for programs without downgrading.
2. Downgrading commands may violate $\leq$, but the programs resulting after downgrading again satisfy the modified strong security condition.
3. Downgrading obeys the information flow relation $\rightsquigarrow$:
   (a) If $D_1 \not\rightsquigarrow D_2$ then no declassification from D_1 to D_2 occurs.
   (b) A command that supposedly downgrades information to D_2 may only affect observations at D_2 or above D_2 in the security lattice.
   (c) A downgrading command that supposedly downgrades information from D_1 must not leak information from other domains.

Note that the first item above subsumes the guarantees provided by the trusted processes approach. The other items above go beyond what the trusted processes approach can achieve. Also note that (3c) is not a trivial requirement because executing $[Id := Id']$ might leak information beyond Id'. There is a danger of information leakage from other domains than $dom(Id')$ via the control flow:

Example 2. Consider [c1:=b1]; [c2:=b2]; if a==0 then [c0:=b1] else [c0:=b2] and the policy represented to the right assuming that variable names follow their security classification. If the program is run in a state where b1≠b2 then an observer at domain C can determine at the end of the run whether a is zero or not although the program was meant to downgrade only from B to C (and not from A to C). Intuitively, such a program is insecure due to the information leakage via the control flow and this should be captured by a sensible security definition.



Let us now try to formalize each of the three aspects. Formalizing property (1) is somewhat straightforward because this is the strong security condition restricted to ordinary transitions ($\vec{V}$ and $\vec{V}'$ shall be as in Definition 4):

$$\forall s, s', t\colon \forall i \in \{1 \ldots n\}\colon \forall \vec{W}\colon \vec{V}\, R\, \vec{V}' \wedge s =_D s' \wedge \langle C_i, s\rangle \rightarrow_o \langle \vec{W}, t\rangle \qquad (1)$$
$$\Rightarrow \exists \vec{W}', t'\colon \langle C_i', s'\rangle \rightarrow_o \langle \vec{W}', t'\rangle \wedge \vec{W}\, R\, \vec{W}' \wedge t =_D t'$$

The remaining properties are concerned with downgrading transitions. Obviously, the analog of the strong security condition cannot hold for such transitions, in general. In particular, it cannot be guaranteed, in general, that the resulting states are D_2-equal if the starting states are D_2-equal (due to downgrading information from some domain D_1 to D_2). However, it can be guaranteed that downgrading commands are executed in lock step and that the resulting programs are in relation. Demanding that downgrading commands are executed in lock step prevents downgrading commands occurring in one branch of a conditional but not in the other. Otherwise, there would be a danger of information leakage in a branch without a downgrading command, which would violate the requirement that declassification in a given program shall be localized to its downgrading commands. That the programs resulting after downgrading are in relation is precisely what property (2) demands. We arrive at the following condition (leaving a place holder for property (3)).

$$\forall s, s', t\colon \forall i \in \{1 \ldots n\}\colon \forall \vec{W}\colon \vec{V}\, R\, \vec{V}' \wedge s =_D s' \wedge \langle C_i, s\rangle \rightarrow_d^{D_1 \to D_2} \langle \vec{W}, t\rangle \qquad (2)$$
$$\Rightarrow \exists \vec{W}', t'\colon \langle C_i', s'\rangle \rightarrow_d^{D_1 \to D_2} \langle \vec{W}', t'\rangle \wedge \vec{W}\, R\, \vec{W}' \wedge (3)$$

Properties (3a)–(3c) express special cases where the resulting states must be D-equal although a downgrading command has occurred. Property (3a) says that downgrading must not have any effect if the information flow relation would be violated, i.e. $D_1 \not\rightsquigarrow D_2 \implies t =_D t'$. Property (3b) says that downgrading to D_2 may only affect the observations at D_2 and at domains being above D_2 in the lattice, i.e. $D_2 \not\leq D \implies t =_D t'$. Finally, property (3c) says that downgrading information from D_1 to D_2 in two states that are indistinguishable for D_1 should result in states that are also indistinguishable, i.e. $s =_{D_1} s' \implies t =_D t'$. In summary, we arrive at the following condition:

$$(D_1 \not\rightsquigarrow D_2 \vee D_2 \not\leq D \vee s =_{D_1} s') \implies t =_D t' \qquad (3)$$

Let us summarize our variant of a strong D-bisimulation. The remaining definitions proceed along the same lines as those in Section 3.

Definition 8 (Strong D-bisimulation). *The* strong *D-bisimulation $\cong_D$ (for $D \in \mathcal{D}$) is the union of all symmetric relations R on command vectors $\vec{V}, \vec{V'} \in \vec{Com}$ of equal size, i.e. $\vec{V} = (C_1, \ldots, C_n)$ and $\vec{V'} = (C'_1, \ldots, C'_n)$, such that*

$$\begin{array}{l}
\forall s, s', t : \forall i \in \{1, \ldots, n\} : \forall \vec{W} : \\
\left[\begin{array}{l} \vec{V}\ R\ \vec{V'} \wedge s =_D s' \wedge \langle C_i, s\rangle \rightarrow_o \langle \vec{W}, t\rangle \\ \Longrightarrow \exists \vec{W'}, t' : \langle C'_i, s'\rangle \rightarrow_o \langle \vec{W'}, t'\rangle \wedge \vec{W}\ R\ \vec{W'} \wedge t =_D t' \end{array}\right] \\
\wedge \\
\left[\begin{array}{l} \vec{V}\ R\ \vec{V'} \wedge s =_D s' \wedge \langle C_i, s\rangle \rightarrow_d^{D_1 \rightarrow D_2} \langle \vec{W}, t\rangle \\ \Longrightarrow [\exists \vec{W'}, t' : \langle C'_i, s'\rangle \rightarrow_d^{D_1 \rightarrow D_2} \langle \vec{W'}, t'\rangle \wedge \vec{W}\ R\ \vec{W'} \\ \quad \wedge ((D_1 \not\rightsquigarrow D_2 \vee D_2 \not\leq D \vee s =_{D_1} s') \implies t =_D t')] \end{array}\right]
\end{array}$$

Definition 9 (Strong D-Security). *Let $Pol = (\mathcal{D}, \leq, \rightsquigarrow)$ be an MLS policy with exceptions and $D \in \mathcal{D}$. A program $\vec{C}$ is* strongly secure for D *iff $\vec{C} \cong_D \vec{C}$.*

Definition 10 (Strong Security). *Let $Pol = (\mathcal{D}, \leq, \rightsquigarrow)$. A program $\vec{V}$ is* strongly secure *iff it is strongly secure for all $D \in \mathcal{D}$.*

The following two theorems justify our re-use of terminology. They show that our new definition of strong security is equivalent to the original one for programs without downgrading and also for policies that do not permit declassification.

Theorem 1. *Let $Pol = (\mathcal{D}, \leq, \rightsquigarrow)$. A program C without downgrading commands is strongly secure under Definition 10 if and only if it is strongly secure under Definition 6.*

The proofs of this and all subsequently presented results are in the extended version of this paper.

Theorem 2. *Let $Pol = (\mathcal{D}, \leq, \rightsquigarrow)$ with $\rightsquigarrow = \emptyset$. Let C be an arbitrary program and C' be the program that results from C by replacing each downgrading command with the corresponding assignment statement. Then C is strongly secure under Definition 10 if and only if C' is strongly secure under Definition 6.*

From now on, we will refer by the terms "strong D-bisimulation", "strong D-security", and "strong security" to Definitions 8, 9, and 10, respectively (rather than to Definitions 4, 5, and 6).

5 Applying the Strong Security Condition

Let us illustrate the strong security condition with some simple example programs that do or do not satisfy the modified strong security condition.

Example 3. For the policy in Example 2, the program `c:=b1` is intuitively insecure because it leaks information outside a downgrading command. This information leakage is ruled out by the strong security condition because running the program in any two C-equal starting states that differ in the value of `b1` results in states that are not C-equal. Hence, condition (1) is violated. The intuitively secure program `[c:=b0]`, however, satisfies the strong security condition.

Composing these two programs sequentially results in [c:=b0]; c:=b1, an intuitively insecure program. This is detected by our strong security condition. In other words, the program is not strongly secure: for any two C-equal starting states that differ in the value of b1 the resulting states are not C-equal. Note that the proposition $\vec{W}\ R\ \vec{W}'$ in condition (2) ensures that the first downgrading command does not mask the information leakage in the second command.

The security condition also rules out information leakage via the control flow.

Example 4. The program if a==0 then [c:=b1] else [c:=b2] is intuitively insecure because it leaks information about the value of a to C (see Example 2). The program is not strongly secure: Take two C-equal starting states s_1 and s_2 with s_1(a)=0, s_2(a)=1, and s_1(b1)$\neq s_2$(b2). Less obviously, the program if b0 then [c:=b1] else [c:=b2] is also not strongly secure: Take two C-equal starting states that differ in their value of b0. Condition (1) demands that the two downgrading commands are C-bisimilar to each other. Running these commands in the same starting state (taking any state with b1$\neq$b2) results in two states that are not C-equal. This contradicts condition (3c).

We have limited the declassification capabilities in our programming language to one very simple downgrading command, i.e. assignment of variables to variables. This is a deliberate choice that shall prevent the programmer from accidentally creating information leaks. Let us illustrate this aspect in the following example where we assume a language with more powerful declassification capabilities. In this extended language, any program fragment can be marked as a trusted piece of code that may declassify information from some domain to another one.

Example 5. RSA encryption is based on computing $a^k \ mod\ n$, where a represents the plaintext and k the encryption key. To efficiently compute $r := a^k \ mod\ n$ without first computing a^k the following exponentiation algorithm can be used:

```
w:=length(k); r:=1; while w > 0 do r:=r*r mod n
                                   if k[w]==1 then r:=(r*a) mod n
                                   w:=w-1
```

Here we have assumed a primitive function length that returns the number of bits needed to encode the key, array indexing k[w] returning the wth bit of k (where k[1] is the least significant bit), and an if-statement without else-branch with the obvious semantics. Let us assume a two-level policy. In a typical context of use, we expect the key k and the data a to be *high*. Therein lies the problem. If w is secret, then the program is not strongly secure, since it loops over w, and the duration of this loop may be indirectly observed by other threads.

We might want to mark the entire algorithm as a trusted piece of code, e.g. by surrounding it with brackets (assuming we had such a powerful declassification capability in our language). However, this is a rather course-grained solution and one might not want to trust a piece of code of this size (as we will see, mistrust is justified). Alternatively, one might only modify the assignment w:=length(k) into a downgrading command and change the security level of w into *low*. Interestingly, it turns out for this second solution that the resulting program is not

strongly secure (take two *low*-equal starting states that differ in the number of ones in the value of k). The problem is that the number of small-step transitions not only differs depending on how often the loop is executed, but also depending on whether the body of the conditional is executed or not. Hence, observing the runtime may reveal to a *low*-level observer more information about the key k than only its length (namely, the number of ones in the key). This is a well-known problem for real implementations [Koc96].

This problem can be solved without significantly changing the algorithm by adding an else-branch with a skip-command, which results in an identical number of atomic computation steps no matter whether the guard of the conditional is true or false. Agat [Aga00] discusses variants for a finer-grained time model.

Downgrading commands where an expression is assigned to a variable are less troublesome than the ones we started out with at the beginning of the example. Though there still is a danger of accidental information leakage, in particular, if the expressions are complex. Therefore, we limit downgrading to even simpler commands that only permit the assignment of variables to variables. In order to meet this constraint, the above program fragment can be modified by replacing [w:=length(k)] with h:=length(k);[w:=h] (where h is a *high*-level variable).

In summary, we arrive at the following strongly secure program:

```
h:=length(k); [w:=h]; r:=1; while w > 0 do r:=r*r mod n
                                           if k[w]==1 then r:=(r*a) mod n
                                                      else skip
                                           w:=w-1
```

6 Mechanizing the Analysis

Proving the strong security condition for a given program is rather tedious. Here, we develop a security type system that can be used to mechanize the information flow analysis. The type system is sound in the sense that any type correct program is also strongly secure. This soundness result provides the basis for mechanically analyzing that a program is strongly secure. Before presenting the (syntactic) type system, we derive some compositionality results that will be helpful for proving soundness.

6.1 Compositionality Properties

The strong security condition with downgrading satisfies a number of compositionality properties. The main compositionality properties are established based on the following reasoning principles for $\approx_D$:

Lemma 1. *If* $C_1 \approx_D C_1'$, $C_2 \approx_D C_2'$ *and* $\vec{V} \approx_D \vec{V'}$, *then*

1. $C_1; C_2 \approx_D C_1'; C_2'$
2. $\mathit{fork}(C_1 \vec{V}) \approx_D \mathit{fork}(C_1' \vec{V'})$
3. *If* $\forall s =_D s' : \langle B, s\rangle \downarrow n \iff \langle B, s'\rangle \downarrow n$ *then*
   (a) *if* B *then* C_1 *else* $C_2 \approx_D$ *if* B *then* C_1' *else* C_2'
   (b) *while* B *do* $C_1 \approx_D$ *while* B *do* C_1'

$$\text{[Var]}\quad Id : dom(Id) \qquad \text{[Const]}\quad n : D \qquad \text{[Skip]}\quad \vdash \mathsf{skip}$$

$$\text{[Arithm]}\quad \frac{Exp_1 : D_1 \quad Exp_2 : D_2 \quad D_1 \leq D \quad D_2 \leq D}{op(Exp_1, Exp_2) : D} \qquad \text{[Seq]}\quad \frac{\vdash C_1 \quad \vdash C_2}{\vdash C_1; C_2}$$

$$\text{[Assign]}\quad \frac{Exp : D \quad D \leq dom(Id)}{\vdash Id := Exp} \qquad \text{[Fork]}\quad \frac{\vdash C \quad \vdash \vec{V}}{\vdash \mathsf{fork}(C\vec{V})}$$

$$\text{[DG]}\quad \frac{dom(Id') \rightsquigarrow dom(Id)}{\vdash [Id := Id']} \qquad \text{[While]}\quad \frac{B : low \quad \vdash C}{\vdash \mathsf{while}\ B\ \mathsf{do}\ C}$$

$$\text{[If]}\quad \frac{B : D \quad \vdash C_1 \quad \vdash C_2 \quad \forall D' \not\geq D : C_1 \cong_{D'} C_2}{\vdash \mathsf{if}\ B\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2}$$

Fig. 2. The Proto Type System

From here it is a small step to the following "hook-up" properties:

Theorem 3. *If C_1, C_2 and $\vec{V}$ are strongly secure then so are*

1. $C_1; C_2$
2. $\mathsf{fork}(C_1\vec{V})$
3. *if B then C_1 else C_2 and while B do C_1 given that there is a least security domain, low, and* $\forall s =_{low} s' : \langle B, s\rangle \downarrow n \iff \langle B, s'\rangle \downarrow n$

6.2 A Security Type System

We begin with the "proto" type system given in Figure 2. The typing rules can be used to deduce when a program is secure. The rules for expressions are quite simple: the security level of variables is determined by the domain assignment, constants may have any level, and the level of compound expressions must be an upper bound of the security levels of each subexpression. The statement $Exp : D$ implies that Exp only depends on the part of the state at or below level D. This is captured by the following standard noninterference property for expressions.

Lemma 2. *If $Exp : D$ then $s =_D s' \implies (\langle Exp, s\rangle \downarrow n \iff \langle Exp, s'\rangle \downarrow n)$.*

The rules for commands largely follow the compositionality properties for the strong security condition. The only point of note for the downgrading rule itself is that there is no "subtyping" permitted. This reflects the tight control of intransitive information flow enforced by the semantic condition. The thrust of the "type" system is concentrated in the conditional rule. This rule has a semantic side condition – so the system is not decidable – but it serves as a starting point for a variety of refinements (to be presented later in this section), and helps to modularize the correctness argument. The rule says that a conditional is secure if each branch is typeable and if the branches have identical behavior for all observers who are not permitted to see the value of the guard. The latter condition

ensures that we do not indirectly leak information via the control flow. Note that this also includes the leaking of the conditional test via a downgrading operation in the branches. We obtain the following soundness result:

Theorem 4. *If* $\vdash C$ *then* C *is strongly secure.*

6.3 A Syntactic Approximation of the Semantic Side Condition

It remains to mechanize the check of the semantic condition $\forall D' \not\geq D : C_1 \cong_{D'} C_2$. Here, we provide a couple of approximations that can be easily mechanized. As the condition only occurs as a premise of the [If] rule, we may safely assume that the programs being compared, i.e. C_1 and C_2, are each strongly secure.

"Minimal" Typing Approximation. Assume the existence of a least security level, *low*. A simple approximation of the semantic rule is the conditional rule

$$\frac{B : \mathit{low} \quad \vdash C_1 \quad \vdash C_2}{\vdash \mathsf{if}\ B\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2}$$

This is indeed a safe approximation of the semantic rule as there are no levels $D' \not\geq \mathit{low}$ when $B : \mathit{low}$ and, hence, the semantic side condition is vacuous.

If we restrict our system to this special case, then the rules correspond, in essence, to the approach taken in the type system of [VS97] (ignoring other language features considered there). As pointed out in [Aga00], such a restriction is rather severe, since programs are not allowed to branch on other data than low data. In the presence of downgrading, however, the restriction is not nearly as severe because one can downgrade the data to the lowest level *before* branching.

Approximation Relations. In order to be more permissive than only permitting branching on *low* guards, we need to find a computable approximation of the condition $\forall D' \not\geq D : C \cong_{D'} C'$ for an arbitrary level D. Before looking at a particular instance, let us clarify the requirements of a safe approximation.

Definition 11 (Safe Approximation Relation). *A family* $\{R_D\}_{D \in \mathcal{D}}$ *of relations on commands is a* safe approximation relation *if given that* C *and* C' *are strongly secure and* $C(R_D)C'$ *holds for some level* D *then* $\forall D' \not\geq D : C \cong_{D'} C'$.

Theorem 5. *Let* $\{R_D\}$ *be a safe approximation relation. Let* $\vdash_R C$ *be the predicate on commands obtained by replacing the [If] rule in Figure 2 with the rule*

$$\mathit{If}_R\ \frac{B : D \quad \vdash C_1 \quad \vdash C_2 \quad C_1(R_D)C_2}{\vdash \mathit{if}\ B\ \mathit{then}\ C_1\ \mathit{else}\ C_2}$$

Whenever $\vdash_R C$ *then* C *is strongly secure.*

A Safe Approximation Relation. The starting point for a variety of approximations is the syntactic identity relation, i.e. $C(R_D)C' \iff C = C'$. This is a safe approximation, though not a too useful one as it amounts to the typing rule

$$\frac{B : D \quad \vdash C}{\vdash \mathsf{if}\ B\ \mathsf{then}\ C\ \mathsf{else}\ C}$$

However, we can derive more useful approximation relations by carefully relaxing syntactic equality while retaining D'-bisimilarity of the branches for all levels D' that are not permitted to see D (i.e. $D' \not\geq D$). For instance, an observer at level D' cannot directly observe the changes caused by an assignment $Id := Exp$ when $D' \not\geq dom(Id)$ (due to transitivity of $\geq$). This fact motivates a larger relation than the syntactic identity, which additionally relates $Id := Exp$ to skip and to assignments $Id' := Exp'$ with $dom(Id') = dom(Id)$.

Definition 12 (Non k-Visible Equality). *Let $\sim_k$ be the least pre-congruence relation on commands (transitive, reflexive and symmetric relation closed under the constructs of the language) satisfying the following rule:*

$$\frac{dom(Id) \geq D}{(Id := Exp) \sim_D skip}$$

Theorem 6. *$\{\sim_k\}$ is a safe approximation relation*

Example 6. The modular exponentiation algorithm from the end of Example 5 is typeable using the above safe approximation relation, since to any non-high observer, the two branches look identical, i.e., r := (r * a) mod n $\sim_{high}$ skip.

7 Related Work

Prior work on intransitive noninterference has focused on more abstract specifications of systems in terms of state machines [Rus92,Pin95,Ohe04], event systems [Man01], or process algebras [RG99,BPR04]. In this article, we have demonstrated how the underlying ideas can be adapted to a programming-language setting. The main objective in the derivation of our novel security condition has been to provide tight control of *where* declassification can occur in a program and *where* exceptions to the information flow ordering are permitted in a security policy. Prior work on controlling declassification in a language-based setting has focused on other aspects and can be classified into the following two categories: *What is downgraded?* and *Who can influence the downgrading decision?*

What? Using Cohen's notion of *selective dependency* [Coh78], one would show, e.g., that a program leaks no more than the least significant bit of a secret by establishing the standard information flow property, firstly, for the case where the secret is even and then for the case where the secret is odd, thereby proving that no *more* than the least significant bit is leaked. A more compact formulation of this idea can be made using *equivalence relations* to model partial information

flow [SS01]. Sabelfeld and Myers [SM03a] have considered a condition which permits *expressions* in a program to be declassified if, roughly speaking, the entire program leaks no more than the declassified expressions would in isolation. An alternative to specifying exactly what is leaked, is to focus on the *amount* of information that is leaked [CHM02,Low02], or on the *rate* at which it is leaked. Complexity-theoretic and probabilistic arguments have been used to argue that leaks are "sufficiently small" or "sufficiently slow" (e.g. [VS00,BP02,DHW02]). An alternative is to modify the model of the attacker, making him less sensitive to small/slow information leaks [MRST01,Lau03].

Who? Zdancewic and Myers [ZM01,Zda03] introduce *robust declassification*, a security condition focusing on the integrity of downgrading decisions, thereby limiting *who* can control downgrading. For example, in a two-level setting, a program exhibits robust declassification if decisions about downgrading of high-level data cannot be influenced by an attacker who controls and observes low-level data. A recent extension includes, amongst other things, also an account of endorsement, i.e. the controlled upgrading of low-integrity data [MSZ04].

Each the three broad approaches (*What?*, *Who?*, *Where?*) has its merits – but also its limitations. Therefore, it would be desirable to combine these approaches with each other. However, this is outside the scope of the current paper.

8 Conclusion

Our main objective has been to localize possibilities for permitted declassification, both in the program and in the security policy. The basis for this has been the introduction of a designated downgrading command and of the information flow relation $\rightsquigarrow$ that co-exists with the usual information flow ordering $\leq$. These concepts allowed us to tightly restrict by our security condition where downgrading can occur. For checking the security condition mechanically, we have presented a security type system that is pleasingly simple, given that it can deal with controlled downgrading. Both, richer language features (from the point of view of the type system) and weaker definitions of security (e.g. without threads or termination sensitivity) deserve further investigation.

It would be desirable to integrate the different approaches for controlling declassification (*What?*, *Who?*, *Where?*). As pointed out before, these approaches are largely orthogonal to each other and, e.g., an analysis of *what* is leaked in a given program can be performed independently from an analysis of *where* information is leaked. The benefit of a tighter integration of these analysis techniques would be that more complicated questions could be investigated like, e.g., *What information is leaked where?* or *Who can leak what information where?*

Acknowledgments. Thanks to Andrei Sabelfeld for useful discussions and to Daniel Hedin and Boris Köpf for helpful comments. The first author thanks Chalmers/Göteborg University for providing an inspiring working environment during his research stay. The work is partially funded by SSF and Vinnova.

References

[Aga00] J. Agat. Transforming out Timing Leaks. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, pages 40–53, 2000.

[BL76] D. E. Bell and L. LaPadula. Secure Computer Systems: Unified Exposition and Multics Interpretation. Technical Report MTR-2997, MITRE, 1976.

[BP02] M. Backes and B. Pfitzmann. Computational Probabilistic Non-interference. In *Proceedings of ESORICS*, LNCS 2502, pages 1–23, 2002.

[BPR04] A. Bossi, C. Piazza, and S. Rossi. Modelling Downgrading in Information Flow Security. In *Proc. of IEEE CSFW*, 2004. to appear.

[CHM02] D. Clark, S. Hunt, and P. Malacaria. Quantitative Analysis of the Leakage of Confidential Data. In *Quantitative Aspects of Programming Languages—Selected papers from QAPL 2001*, volume 59 of *ENTCS*, 2002.

[Coh78] E. S. Cohen. Information Transmission in Sequential Programs. In *Foundations of Secure Computation*, pages 297–335. Academic Press, 1978.

[Den76] D. E. Denning. A Lattice Model of Secure Information Flow. *Communications of the ACM*, 19(5):236–243, 1976.

[DHW02] A. Di Pierro, C. Hankin, and H. Wiklicky. Approximate Non-Interference. In *Proceedings of IEEE CSFW*, pages 1–17, 2002.

[Koc96] P. C. Kocher. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In Neal Koblitz, editor, *Advances in Cryptology – CRYPTO'96*, volume 1109 of *LNCS*, pages 104–113. Springer-Verlag, 1996.

[Lau03] P. Laud. Handling Encryption in an Analysis for Secure Information Flow. In *Proceedings of ESOP*, LNCS 2618, pages 159–173. Springer-Verlag, 2003.

[Low02] G. Lowe. Quantifying Information Flow. In *Proceedings of IEEE CSFW*, pages 18–31, 2002.

[Man01] H. Mantel. Information Flow Control and Applications – Bridging a Gap. In *Proceedings of Formal Methods Europe*, LNCS 2021, pages 153–172, 2001.

[MRST01] J. Mitchell, A. Ramanathan, A. Scedrov, and V. Teague. A Probabilistic Polynomial-Time Calculus for Analysis of Cryptographic Protocols (Preliminary report). In *Proc. of the Conf. on the Math. Foundations of Programming Semantics*, volume 45 of *ENTCS*, 2001.

[MSZ04] A. C. Myers, A. Sabelfeld, and S. Zdancewic. Enforcing Robust Declassification. In *Proc. of IEEE CSFW*, 2004. to appear.

[Ohe04] David von Oheimb. Information flow control revisited: Noninfluence = Noninterference + Nonleakage. In *Proc. of the* 9^{th} *European Symposium on Research in Computer Security*, LNCS. Springer, 2004. to appear.

[Pin95] S. Pinsky. Absorbing Covers and Intransitive Non-Interference. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 102–113, Oakland, CA, USA, 1995.

[RG99] A. W. Roscoe and M. H. Goldsmith. What is Intransitive Noninterference? In *Proceedings of IEEE CSFW*, pages 228–238, 1999.

[Rus92] J. M. Rushby. Noninterference, Transitivity, and Channel-Control Security Policies. Technical Report CSL-92-02, SRI International, 1992.

[SM03a] A. Sabelfeld and A. C. Myers. A Model for Delimited Information Release. In *International Symposium on Software Security*, 2003.

[SM03b] A. Sabelfeld and A. C. Myers. Language-Based Information-Flow Security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, 2003.

[SS00] A. Sabelfeld and D. Sands. Probabilistic Noninterference for Multi-threaded Programs. In *Proceedings of IEEE CSFW*, pages 200–214, 2000.

[SS01] A. Sabelfeld and D. Sands. A Per Model of Secure Information Flow in Sequential Programs. *HOSC*, 14(1):59–91, 2001.

[VS97] D. Volpano and G. Smith. Eliminating Covert Flows with Minimum Typings. In *Proceedings of IEEE CSFW*, pages 156–168, 1997.

[VS00] D. M. Volpano and G. Smith. Verifying Secrets and Relative Secrecy. In *Proceedings of POPL*, pages 268–276, 2000.

[Zda03] S. Zdancewic. A Type System for Robust Declassification. In *Proc. of the Conf. on the Math. Foundations of Programming Semantics*, ENTCS, 2003.

[ZM01] S. Zdancewic and A. C. Myers. Robust Declassification. In *Proceedings of IEEE CSFW*, pages 15–23, 2001.