Master Thesis

Master of Science Informatik

Controlled Declassification under Semantics with Schedulers

Matthias Perner

TU Darmstadt
Fachbereich Informatik
Fachgebiet Modeling and Analysis of Information Systems

Prüfer: Prof. Dr.-Ing. Heiko Mantel
Betreuer: Dipl.-Inform. Alexander Lux

Abgabetermin: 10. Januar 2011

Erklärung:

Hiermit versichere ich, dass ich die Arbeit selbstständig verfasst und keine anderen als die angegebenen Hilfsmittel und Quellen verwendet, sowie Zitate kenntlich gemacht habe.

Darmstadt, den 10. Januar 2011

Matthias Perner

Danksagung

Ich möchte meinem Betreuer Dipl.-Inf. Alexander Lux danken, der mir während dieser Arbeit immer mit Rat zur Seite stand. Seine Betreuung war mir sowohl fachlich als auch methodisch eine große Hilfe, ins Besondere da er sich viel Zeit für Gespräche und Diskussionen mit mir nahm.

Weiterhin möchte ich Herrn Prof. Dr.-Ing Heiko Mantel für die Möglichkeit danken, dieses interessante Forschungsgebiet bearbeiten zu können und dafür, dass er mir mit seinem Feedback geholfen hat über die Formalisierungen und deren Rechtfertigung wiederholt nachzudenken und diese so zu verfeinern.

Außerdem danke ich meiner Familie und meinen Freunden, die meinen Ausführungen über dieses ihnen fremde Thema Gehör geschenkt haben und mir dabei oft unbewusst geholfen haben, die Ansätze und Ideen zu reflektieren.

ABSTRACT

Information flow analysis and control for concurrent programs is a very important topic in information security. The security properties that describe what an analysis enforces play an important role, because they describe the semantics of security. Many of these properties are based on non-interference, where public information must be independent from secret information. In many scenarios, for example password checks, a complete independence is too restrictive. In order to be able to analyze such programs intentional information leaks called declassifications are necessary. These declassification should be controlled in order to prevent the program from leaking just any secret information. In this thesis we give a detailed look on two scheduler-independent properties that control the declassification with respect to where in the program a declassification occurs and what is declassified. Furthermore, we introduce a novel, scheduler-independent property that allows an association of the information what is declassified where in the program and present a sound type system that enforces this property. Additionally, we discuss how the security properties capture our intuition in several small code examples and an application scenario from the view of the software developers.

Contents

1. Introduction

1.1. Motivation

Confidentiality of information is an important security goal in many cases. For example if the passwords for logging in to the operating system would not be kept confidential, an attacker could impersonate the user that is associated with the password and thus render the other security mechanisms of the operating system useless. Since it is such an important goal it is reasonable that a program should be checked for the preservation of this property. On the other hand, many modern software systems that deal with confidential information are complex programs. A web browser, for instance, transfers information over the network for different purposes, like loading a web site the user has requested or checking for updates or registration information or supplying the developers statistics. Additionally the browser might store passwords or be extensible with plugins, maybe even use multi-threading for loading different web sites concurrently.

In a sequential program the information may get leaked explicitly, for example in an assignment, or implicit, for example hidden in the control flow of a branching command like if. The presence of multi-threading introduces even more possibilities for information leaks, for example the interactions of the threads via the scheduler or shared memory. Many of these leaks are very subtle and therefore hard to detect. Additionally, the complexity of many systems makes the detection even harder, because there is much code with possibly subtle interactions that need to be checked. Furthermore, the information leakage could be accidental, for example due to programming errors, or even intentional, for example in a trojan horse. In the second case the developer of the trojan horse may put much effort into leaking the information as subtly as possible.

Information flow analyses are a common solution for this problem. Such an analysis may be used to check whether confidential information flows to some public location or output. For developing meaningful program analyses it is necessary to explicate the property that it checks, because the property is the description of the purpose, in our case confidentiality. Consequently, the properties play an important role in describing and enforcing confidentiality.

Non-interference-like properties follow the basic idea that independence of public data from confidential data is required. In some situations complete independence is too restrictive. If we wanted to write, for example, a login with a password check, the stored password should be kept secret, while all the user input, as well as the success of the login and therefore the password check, is visible for the public user. The password check would reveal the information about the equivalence of the user input and the stored password and thereby reveal confidential information. This information leakage is intended, since it is necessary for the functionality of the login. Such an information leakage that is intended and assumed to be secure is called a declassification.

Programs that use declassifications can not be categorized as secure with pure non-interference properties. Much research effort is put into the question how to control declassification such that declassifications that are necessary for the functionality are allowed, but the intended confidentiality is preserved [SS05]. The authors of [MS04]

introduced the three W-aspects to categorize the control of declassification. These three aspects are: **where** declassification may occur, **what** information may get declassified and **who** may initiate a declassification.

In [SS00] the authors develop STRONG-Security, a security property that uses the basic idea of pure non-interference for multi-threaded programs. This property is scheduler-independent for a wide class of schedulers. The scheduler-independence of the property is beneficial, because in many cases the scheduler is not known before run time or different instances of the program may even be run with different schedulers. Since this property follows pure non-interference, no declassifications are allowed.

The authors of [MR07] use the concept introduced with STRONG-Security to create novel properties that allow declassifications that are controlled with respect to the aspects **where** and **what** in multi-threaded programs. The intuition behind building the novel properties on the foundation of STRONG-Security is that the novel properties should be scheduler-independent, too.

1.2. Challenge

Since the properties play such an important role for defining what security means and for analyzing security of programs, great confidence in the properties and in their adequacy is desirable. One of the great challenges when developing security properties is to gain the confidence in the properties for example by explaining the intuition behind their definition and showing that they capture it. In the presence of declassification the argumentation about adequacy is even harder, if the intuition behind those properties is not clearly explained, because of the subtle possibilities for violations.

The three W-aspects from [MS04] make it possible to categorize the controls with respect to these important aspects and thus make it easier to argue about the intuition and the goal of properties.

Another challenge is to show that the properties are adequate in the sense that they allow declassifications as they are needed for the desired functionality, but preserve the intuitive understanding of confidentiality.

1.3. Contribution

In this work we examine the properties WHERE-Security and WHAT-Security from [MR07] to strengthen the confidence in these properties. Furthermore, we show that these properties are scheduler-independent and thus preserve their faithfulness under a wide class of schedulers.

On the foundation of these properties we present an integration of the controls for the aspects **where** and **what** that allows a more exact control of those aspects. We show that this property is scheduler-independent with respect to the same class of schedulers as WHERE-Security and WHAT-Security. Furthermore, we look at some example programs to compare the intuition, adequacy and applicability for real world programs.

1.4. Structure

In Chapter 2 we introduce a small language for example programs and explain the basic idea behind STRONG-Security from [SS00] as this property is on the one side the inspiration for the properties WHERE-Security and WHAT-Security that control declassification and on the other side is the baseline for comparing the declassification controls in those properties.

Chapter 3 explains the properties WHERE-Security and WHAT-Security from [MR07], especially how the control of declassifications is realized in those properties. Furthermore, we show that the properties are scheduler-independent with respect to the same class of schedulers as STRONG-Security to show that these properties preserve their faithfulness with many different schedulers and are therefore adequate for multi-threaded settings.

In Chapter 4 we present a tighter integration of the two aspects **where** and **what** into a single property. This integration builds on the foundation of the properties WHERE-Security and WHAT-Security. Additionally, we present some pitfalls that can occur when controlling declassification and show that the novel property is scheduler-independent. Furthermore, we present a sound type system that analyzes programs in the example language for confidential information flow with respect to the integrated property.

In Chapter 5 we review the properties and show their impact on some program fragments. Furthermore, we examine the applicability for real world examples with some small example programs.

Finally, Chapter 6 concludes the work with a short summary of the results and puts them in context with other related work. Furthermore, a short overview of possible future work is given.

2. Preliminaries

In this work we want to develop and analyze information flow properties that depend on the formal semantics of programs. Hence, we start with defining a language for our examples that supports multi-threading with dynamic thread creation. Since we want our properties to be scheduler-independent we introduce a scheduler model that can be used to describe a wide class of schedulers.

The next step is to specify the rules for the information flow in the form of security policies. After that we define the capabilities of an attacker in a reasonable attacker model that describes what an attacker can see.

Since STRONG-Security from [SS00] is the inspirational foundation and the baseline for comparing the other properties in this work, we give a short overview over the intuition and idea behind it.

2.1. Terminology and Notation

Before developing properties that describe what security means in the sense of obeying specific rules, we want to introduce some terminology and notation.

A bisimulation is a symmetric relation R on states where the occurrence of $(a, b) \in R$ means that the states a and b have an equivalent behavior. That means that for all states c if a can make a transition to c then b can make a transition to c, too.

A partial equivalence relation is a relation that is symmetric and transitive, but not necessarily reflexive. The part of the partial equivalence relation that is reflexive partitions the set in equivalence classes just like a normal equivalence relation. An element of the set that is not related to itself under the partial equivalence relation is not element of any equivalence class.

For any (partial) equivalence relation R we will write aRb to denote that $(a, b) \in R$ and A/R to denote the set of all equivalence classes of A induced by R. Furthermore, we will write $[a]_R$ to denote the equivalence class induced by R that contains a. Therefore $b \in [a]_R$ denotes that b is in the same equivalence class induced by R as a which means aRb. If a is not related to itself under the relation R, denoted by $(a, a) \notin R$, then a is not in any equivalence class denoted by $\emptyset \in [a]_R$ and especially $a \notin [a]_R$.

The security properties that we discuss in this work are based on bisimulations that are partial equivalence relations.

2.2. Multi-threaded While Language

In order to be able to define the language we introduce a program model with a memory, expressions and commands.

We model the set of all possible values in a program with the set Val and leave this set underspecified, but assume that it contains at least the boolean values $True$ and $False$, as well as the the set of integers $\mathbb{Z}$.

The set Var models the set of variables in the program. A variable identifies a location in the memory.

Definition 1 ***Memory State:***
A Memory State *is a function s of the type $Mem : Var \to Val$, that maps values to variables.*

If the value val of a variable var is defined in a memory state s we write $s(var) = val$ and say that var has the value val under memory state s. We assume shared variables. This is important when we introduce multi-threading since the shared variables are a communication medium between threads. We use $s \otimes \{var = val\}$ to denote the state that is identical to s, but val is mapped to var.

The set Op denotes the operations that can be used to combine the values of variables. We leave this set underspecified, but assume that the set contains at least the common boolean operations $\{and, not, or\}$ and the common arithmetic operations $\{+, -, *, /, =\}$ with their intuitive meaning. Although we usually use the infix notation for operations in program examples, we restrict ourselves to a prefix notation with brackets around the operands in the definitions for simplicity and clearity.

Definition 2 ***Expressions:***
Let expr be an Expression*, then expr can be deduced with the rules from Figure 2.2.*

$$\frac{expr \in Val}{expr \in Expr} \quad \frac{expr \in Var}{expr \in Expr} \quad \frac{expr_1 \in Expr \ldots expr_n \in expr \quad op \in Op}{op(expr_1, \ldots, expr_n)}$$

Figure 1: Structure of an Expression in the MWL

We leave the set $Expr$ underspecified, since we did not specify possible operations on variables completely. Furthermore, we are not interested in the expressions per se, but in some information that relies on the expressions. We use $vars(expr)$ to denote the set of all variables in the expression and $subexpressions(expr)$ to denote the set of all subexpressions of the expression.

Definition 3 ***Evaluation of an Expression:***
An Evaluation of an Expression $\langle expr, s \rangle \downarrow val$ *reduces the expression $expr \in Exprs$ to a value $val \in Val$ with respect to the memory state $s : Var \to Val$.*

Since the set of expressions is underspecified, we can not define the exact semantics of the reduction. Instead we assume that boolean and arithmetic operations are evaluated according to their common, intuitive meaning.

On these preliminaries we want to define the Multi-threaded While Language, or short MWL. MWL is a language that supports common features of imperative languages, like assignments, expressions, control flow branches and loops. Additionally, it supports dynamic thread creation.

A single thread in MWL is a command $thread \in Com$ where Com is defined by the grammar in Figure 2. We use the empty command $\langle\rangle$ to denote that no command

remains to be executed for a thread. A thread pool in the MWL is a list of threads denoted by $\overrightarrow{C} = \langle C_0 \dots C_{n-1} \rangle$ where n is the length of the thread pool. We call the thread pool before executing any command program, since it represents the code that we want to analyze. Normally the program is a thread pool of length 1, since this is the initial thread that spawns all other threads. Sometimes we will refer to a thread pool with a length that is not 1 as program for simplicity, because our focus is not on the initial thread creation of a program, but on the information flow properties during the execution.

$$
\begin{array}{rcl}
Com & :== & Com; Com \mid \textbf{skip} \\
 & \mid & \textbf{if } (\ Expr\ ) \textbf{ then } Com \textbf{ else } Com \textbf{ fi} \\
 & \mid & \textbf{while } (\ Expr\ ) \textbf{ do } Com \textbf{ od} \\
 & \mid & Var := Expr \\
 & \mid & \textbf{fork } (\ Com\ \overrightarrow{Com}\ )
\end{array}
$$

Figure 2: Grammar for the Multi-threaded While Language

This small language supports the common concepts of imperative languages. Sequences Com;Com are a combination of other commands that are executed sequentially one after another. **skip** is a command that does neither change the memory, nor influence the control flow in a thread. The command **if** results in a branching of the control flow with respect to the evaluation of the expression. The looping command **while** executes the command between **do** and **od** as long as the expression in the brackets evaluates to $True$. Memory changes are made with assignments of the form $Var := Expr$. The command **fork** executes the first command in the brackets and adds new threads, according to the right command in the brackets, to the thread pool.

Definition 4 ***Thread Configuration:***
A Thread Configuration *is a tuple* $\langle\!|C, s|\!\rangle$ *of a thread* $C \in Com$ *and a memory state* $s : Var \to Val$.

A thread configuration captures the state of a thread at a given time. The commands in C are the commands that remain to be executed and the memory state s captures the memory at the time of the observation.

We use an operational small step semantics that transforms the thread C into C' by removing the command that has been executed from C. We formalize the intuition of the commands with the semantics in Figure 3. We call the transition from one thread configuration to another thread configuration according to the semantics an execution step.

2.3. Scheduler Model

In a multi-threaded environment, the scheduler determines which thread performs an execution step. Hence, the scheduler interacts with single threads and can be used as a

$$\frac{\langle\!|C_1, s|\!\rangle \rightarrow_o \langle\!|\langle\rangle, s'|\!\rangle}{\langle\!|C_1; C_2, s|\!\rangle \rightarrow_o \langle\!|C_2, s'|\!\rangle} \quad \frac{}{\langle\!|\textbf{skip}, s|\!\rangle \rightarrow_o \langle\!|\langle\rangle, s|\!\rangle}$$

$$\frac{\langle\!|B, s|\!\rangle \downarrow True}{\langle\!|\textbf{if } B \textbf{ then } C_1 \textbf{ else } C_2 \textbf{ fi}, s|\!\rangle \rightarrow_o \langle\!|C_1, s|\!\rangle} \quad \frac{\langle\!|B, s|\!\rangle \downarrow False}{\langle\!|\textbf{if } B \textbf{ then } C_1 \textbf{ else } C_2 \textbf{ fi}, s|\!\rangle \rightarrow_o \langle\!|C_2, s|\!\rangle}$$

$$\frac{\langle\!|B, s|\!\rangle \downarrow True}{\langle\!|\textbf{while } B \textbf{ do } C \textbf{ od}, s|\!\rangle \rightarrow_o \langle\!|C; \textbf{while } B \textbf{ do } C \textbf{ od}, s|\!\rangle} \quad \frac{\langle\!|B, s|\!\rangle \downarrow False}{\langle\!|\textbf{while } B \textbf{ do } C \textbf{ od}, s|\!\rangle \rightarrow_o \langle\!|\langle\rangle, s|\!\rangle}$$

$$\frac{}{\langle\!|\textbf{fork}(C\ \overrightarrow{V}), s|\!\rangle \rightarrow_o \langle\!|C\ \overrightarrow{V}, s|\!\rangle} \quad \frac{\langle\!|C_1, s|\!\rangle \rightarrow_o \langle\!|C_1\ \overrightarrow{V}, s'|\!\rangle}{\langle\!|C_1; C_2, s|\!\rangle \rightarrow_o \langle\!|(C_1; C_2)\ \overrightarrow{V}, s'|\!\rangle}$$

$$\frac{\langle\!|exp, s|\!\rangle \downarrow n}{\langle\!|var{:=}exp, s|\!\rangle \rightarrow_o \langle\!|\langle\rangle, s \otimes \{var = n\}|\!\rangle}$$

Figure 3: Operational semantics for threads in the Multi-threaded While Language

communication medium, since it determines the execution order of commands and the execution order has an influence on the memory state. Often the scheduler is unknown before executing a program or different instances of a program may be run with different schedulers. Therefore it is reasonable to aim for a scheduler model that allows the representation of a wide class of schedulers. The authors of [SS00] introduced a scheduler model that we want to adapt.

Many scheduling algorithms rely on information about which threads have been executed in the past. In [SS00], this is addressed by introducing histories of previously scheduled threads. A history is a list of pairs of the type $\mathbb{N} \times \mathbb{N}$, where the first component denotes the last scheduled thread and the second component denotes the amount of live threads. We write $Hist$ for the set of all histories and write ϵ for the empty history.

The function $live(H)$ returns the amount of live threads of the given history $H \in Hist$. It returns 1 for the empty history and the second component of the last element, if the history is not empty. In our language where threads can not be blocked $live(H)$ equals the length of the thread pool which is the amount of existing threads.

According to [SS00] some scheduling algorithms use information about the memory state of a program to determine the scheduling order.

Definition 5 ***σ-scheduler:***
A σ-scheduler *is a function that given a history H and a part of the memory mempart returns a probability distribution on live threads.*

$$\sigma(H, mem) \in distribution(\{0 \ldots live((h) - 1)\})$$

In order to be able to capture a wide class of schedulers, we model the scheduler as a function that returns a probability distribution on live threads. This distribution

represents the probabilities for the threads to execute a step. We write $\sigma(H, mem)[i]$ to denote the probability that thread i is scheduled under the history H and memory state mem.

With this model it is possible to represent deterministic schedulers, where the probability is 0 for all threads but one, as well as probabilistic schedulers as the authors of [SS00] show. We call the class of schedulers that can be represented with this model σ-schedulers and each time we walk about schedulers implicitly refer to the class of σ-schedulers.

Definition 6 ***System Configuration:***
A System Configuration *is a tuple* $\langle\!|H, \overrightarrow{C}, s|\!\rangle$ *of a history* $H \in Hist$*, a thread pool* $\overrightarrow{C} \in \overrightarrow{Com}$ *and a memory state* $s : Var \rightharpoonup Val$*.*

The system configurations capture the state of the complete system. The history H is the history of all previously scheduled threads, the thread pool $\overrightarrow{C}$ is the vector that describes the remaining computations for the threads and the memory state s describes the memory shared between all threads.

While the transitions between thread configurations describe the changes to the remaining commands of a single thread and the global state, we also need to keep track of the histories and the state of the thread pools. We model this with transitions between system configurations and require that these transitions may only occur if a thread executed a step. As in [SS00], the transitions between system configurations are described by a semantics rule in Figure 4.

$$\frac{\langle\!|C_i, s|\!\rangle \rightarrow \langle\!|\overrightarrow{W}, s'|\!\rangle}{\langle\!|H, \langle C_0 \dots C_i \dots C_{n-1}\rangle, s|\!\rangle \xrightarrow{p} \langle\!|H(i, n + |\overrightarrow{W}| - 1), \langle C_0 \dots C_{i-1} \overrightarrow{W} C_{i+1} \dots C_{n-1}\rangle, s'|\!\rangle}$$

Figure 4: Semantics of Thread Pools

According to the semantics for single threads the execution of a step changes the command that must be executed in the thread itself and possibly spawns new threads. We capture this by the resulting command vector $\overrightarrow{W}$ in the execution steps and update the Thread Pool by replacing C_i with $\overrightarrow{W}$.

The semantics of thread pools capture the updating of histories according to our intuition by extending the list that represents the history with a new tuple $(i, n+|\overrightarrow{W}|-1)$. Since the update of the system configuration is triggered by an execution step of thread i, the thread i is remembered as the last scheduled thread in the history. Since C_i is replaced with the resulting command vector $\overrightarrow{W}$, the amount of live threads is now equal to the amount of previously live threads n plus the amount of newly created threads $|\overrightarrow{W}|$ minus one, since C_i was replaced.

At last we propagate the resulting memory state from the execution step to the new system configuration.

We call a step from one system configuration to another a system step or scheduler step.

2.4. Security Policies

We want a program to preserve confidentiality. For this purpose we want to analyze the program for secure information flow. In order to do this, we need to specify the meaning of confidentiality or security.

We use a set $\mathcal{D}$ of security domains to classify information and information containers. In the following, we use variables as information containers and let the set of all variables be denoted by Var. Furthermore, the level of information that is stored in a variable is in the same domain as the variable itself.

Definition 7 ***Multilevel Security Policy:***
A Multilevel Security Policy *(brief: mls policy) is a tuple* $(\mathcal{D}, \leq, dom)$*, where* $\mathcal{D}$ *is a set of security domains,* $\leq \subseteq \mathcal{D} \times \mathcal{D}$ *is a partial order that has a minimum and* $dom : Var \to \mathcal{D}$ *is a domain assignment.*

The minimum of $\mathcal{D}$ *with respect to* $\leq$ *is called low and the maximum, if it exists, is called high.*

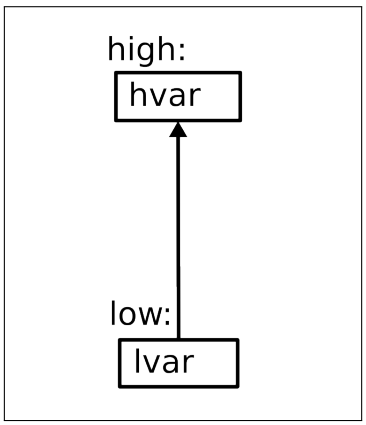
In a given flow policy, the partial order $\leq$ describes the regular information flow that may occur in the program. Regular information flow from one domain $D_1 \in \mathcal{D}$ to another domain $D_2 \in \mathcal{D}$ is allowed, if $D_1 \leq D_2$ holds. When we define a mls policy in our examples, we will not explicitly write down the reflexive and transitive parts of $\leq$, since they can be derived by generating the transitive and reflexive closure of the pairs we define.

The domain assignment maps security domains to variables to capture the intuition that the variables may only hold information that may flow to their security domain.

We call an information flow from D_1 to D_2 where $D_1, D_2 \in \mathcal{D}$ legal, if $D_1 \leq D_2$. Otherwise the information flow is called illegal.

Example:

Let *pol* be the following mls policy:
$\mathcal{D} = \{low, high\}$
$low \leq low$, $low \leq high$, $high \leq high$
$dom(lvar) = low$
$dom(hvar) = high$.
This policy describes that information may flow regularly from the security domain *low* to the security domain *high*. Information flow is not allowed from *high* to *low*. That means that a program that contains for example lvar := hvar violates the mls policy, since the information from the *high* variable *hvar* is assigned to the *low* variable *lvar*, but $lvar \leq hvar$.



Definition 8 ***Source Domains of an Expression:***
Source Domains of an Expression *is a function* $sources : Expr \to \mathcal{P}(\mathcal{D})$ *that returns the set of domains of all variables in the expression.*

This function is related to an expression as the domain assignment is to the variables. Since we specified that an expression is a combination of variables with operations, the source domains for the combined information of the expression are all the domains of the variables. This intuition is captured by the source domains of an expression. We will write $source(expr)$ when we mean the source domains of the expression $expr$.

2.5. Attacker Model

When talking about security one must make some assumptions about the abilities of the attacker. These assumptions have an influence on the security properties by defining what an attacker is able to see and therefore which states an attacker can distinguish. In general it is better to assume an attacker which has more capabilities, because that leads to stricter security properties and in consequence prevents more security leaks.

One possibility to formalize non-interference is to capture the capability of the attacker to distinguish different states or behaviors of the program. We capture these capabilities of the attacker with equivalence relations that describe which states look equivalent for an attacker and hence which states are indistinguishable for him.

Definition 9 ***D-equality:***
Two states s_1 *and* s_2 *are* D-equal *with respect to a security domain* D *(denoted:* $s_1 =_D s_2$*), if all variables, that are visible to an observer of the domain* D*, have identical values.*

$$s_1 =_D s_2 \iff \forall var \in Var : (dom(var) \leq D \implies s_1(var) = s_2(var))$$

We assume that the variables that are of a level lower than D are visible for an observer of the security level D. In consequence, this definition captures the intuition that an attacker can not distinguish memory states that are equal in the parts of the memory he can observe.

Furthermore, we assume that the attacker can observe or at least approximate the scheduling behavior. In the scheduler model from Section 2.3 this means that the attacker can learn or approximate the probabilities of a given thread to be scheduled. We explain this with the fact that an attacker may run the same program several times with the same initial memory state and therefore can gain information about the scheduling.

Since the scheduler function is parametrized with the memory state, the probability distribution of the thread selection can reveal information about the memory state. In consequence, we assume that the scheduler itself must fulfill some kind of security restriction.

Definition 10 *low-**Secure** σ-**Scheduler:***
*A low-*Secure σ-Scheduler *is a σ-scheduler that fulfills the following formula:*

$$\forall s_1, s_2 \in Mem : \forall H \in Hist : s_1 =_{low} s_2 \Longrightarrow \sigma(H, s_1) = \sigma(H, s_2)$$

The intuition behind this definition is that all *low*-secure σ-schedulers do not reveal information about those parts of the memory that a *low*-observer may not learn. This is realized by requiring that the probability distribution returned by the scheduler function are equal for two states that are *low*-indistinguishable. Since *low* is defined as the minimum of $\mathcal{D}$ with respect to $\leq$ all *low*-secure σ-schedulers are only allowed to reveal information that every observer is allowed to know or learn.

During the rest of the work we will only write σ-scheduler when we mean *low*-secure σ-scheduler since the class of *low*-secure σ-schedulers is the class of schedulers that is used as σ in [SS00].

The second parameter of the scheduler function are the histories of previously scheduled threads. Since not every scheduler uses the whole history for the calculation of the probabilities many histories will result in equivalent probability distributions.

Definition 11 *σ-**Equivalence of Histories:***
Two histories $H_1, H_2 \in Hist$ are σ-equivalent *(denoted: $H_1 =_\sigma H_2$), if the scheduler σ returns the same probability distribution for both histories under a given memory state s and adding equal elements to the histories does not render them distinguishable.*

The intuition behind the definition of σ-equivalence of histories is that we can partition the set of all histories in equivalence classes for a given scheduler σ. Since the scheduler can not distinguish the histories in one equivalence class, the probability distribution for histories in the same equivalence class must be equal under the same memory state. We can capture the first important consequence of this definition with the following formula:

$$\frac{H_1 =_\sigma H_2}{\forall H_1, H_2 \in Hist : \forall s \in Mem : \sigma(H_1, s) = \sigma(H_2, s)}$$

Additionally, if the scheduler can not distinguish the histories in one equivalence class, adding the same pair to the elements of an equivalence class should not render them distinguishable, since the pairs are indistinguishable and intuitively the indistinguishability should be preserved under sequential composition of histories. We can capture this consequence of the definition with the formula:

$$\frac{H_1 =_\sigma H_2}{H_1(i, m) =_\sigma H_2(i, m)}$$

Since we assume that the attacker may approximate or even learn the probability distributions by running the program repeatedly with the same initial memory state, the σ-equivalence is an important relation, because it describes which scheduling histories the attacker can not distinguish.

Theorem 1 ***Probability Equivalence for Equivalent Histories and States:*** *Given a low-secure σ-scheduler, two σ-equivalent histories $H_1 =_\sigma H_2$ and two low-equivalent states $s_1 =_{low} s_2$, the probability distributions returned by the scheduler function σ are equal.*

$$\forall \sigma, H_1, H_2, s_1, s_2 : H_1 =_\sigma H_2 \wedge s_1 =_{low} s_2 \Longrightarrow \sigma(H_1, s_1) = \sigma(H_2, s_2)$$

Proof Probability Equivalence for Equivalent Histories and States: This theorem follows from the definition of *low*-secure σ-schedulers and the definition of σ-equivalent histories. *low*-secure σ-schedulers require that for two memories that are *low*-equivalent, the scheduler function returns equivalent probability distributions. The definition of σ-equivalent histories requires that for σ-equivalent histories the returned probability distribution is equal for σ-equivalent histories under a fixed memory state.

If $H_1 =_\sigma H_2$ holds, then probability distributions returned by σ must be equivalent under a fixed memory according to the definition of σ-equivalence and in consequence the following holds, too:

$$\begin{aligned} &\forall s \in Mem : \sigma(H_1, s) = \sigma(H_2, s) \\ &\Longrightarrow \sigma(H_1, s_1) = \sigma(H_2, s_1) \wedge \sigma(H_1, s_2) = \sigma(H_2, s_2) \end{aligned}$$

According to the definition of *low*-secure σ-schedulers, the probability distributions returned for s_1 and s_2 must be equal under fixed history, if $s_1 =_{low} s_2$ holds and in consequence the following holds, too:

$$\begin{aligned} &\forall H \in Hist : \sigma(H, s_1) = \sigma(H, s_2) \\ &\Longrightarrow \sigma(H_1, s_1) = \sigma(H_1, s_2) \wedge \sigma(H_2, s_1) = \sigma(H_2, s_2) \end{aligned}$$

The combination of those two facts gives us as result:

$$\sigma(H_1, s_1) = \sigma(H_2, s_1) = \sigma(H_2, s_2)$$

■

This theorem is an interesting statement about the capabilities of the scheduler to reveal information via the probability distribution, since the probability distributions are equal and therefore indistinguishable, if we have histories that are indistinguishable for the scheduler and memory states that are indistinguishable for an *low*-observer.

2.6. Strong security

In [SS00], a program is defined as strongly secure, if a partial equivalence relation exists, that fulfills the characterization formula for strong *low*-bisimulations, and the program is related to itself under this relation. Strong security is the inspirational foundation and baseline for WHERE-Security and WHAT-Security, which in turn are the foundation for the integration we want to introduce in this work. Therefore, we want to explain strong security briefly to give a better understanding of the intuition behind it.

Definition 12 ***Strong*** *low*-***Bisimulation:***
A strong low-bisimulation $\approx_L$ *is the union of all symmetric relations* R *on thread pools of equal size, such that whenever* $\langle C1_0 \dots C1_{n-1}\rangle R \langle C2_0 \dots C2_{n-1}\rangle$ *then*

$$\begin{aligned}&\forall s_1, s_2 : \forall i \in \{0 \dots n-1\} : (\!|C1_i, s_1|\!) \rightarrow (\!|\overrightarrow{C1'}, s_1'|\!) \wedge s_1 =_{low} s_2 \\ &\implies \exists \overrightarrow{C2'} : \exists s_2' : ((\!|C2_i, s_2|\!) \rightarrow (\!|\overrightarrow{C2'}, s_2'|\!) \wedge \overrightarrow{C1'} R \overrightarrow{C2'} \wedge s_1' =_{low} s_2')\end{aligned}$$

Strong *low*-bisimulations use the idea that two configurations are indistinguishable for an attacker, if in both configurations the memory states are indistinguishable and the behavior of the thread pools, which means the possible transitions to new configurations, are indistinguishable.

It captures the idea of non-interference by requiring that from two indistinguishable configurations, it is possible to perform an execution step in the same thread in both thread pools and the resulting configurations are again indistinguishable. This captures the idea of non-interference, because whenever $s_1 =_{low} s_2$, but $s_1 \neq s_2$, then an information flow of the information that causes $s_1 \neq s_2$ to the domain *low* results in $s_1 =_{low} s_2$ and thus a program that contains an instruction that results in such an information flow can not be related to itself under a strong *low*-bisimulation and independence from *low* information of *high* information is required by the strong *low*-bisimulation.

The authors of [SS00] specify strong security on the basis of strong *low*-bisimulations as follows:

$$\overrightarrow{C} \text{ is } \textit{strongly secure} \iff \overrightarrow{C} \approx_L \overrightarrow{C}$$

In other words, a program is *strongly secure*, if and only if a program can be related to itself under a relation that is a strong *low*-bisimulation. On the other hand, a program is considered insecure, if no such relation exists. This uses the fact that partial equivalence relations are not reflexive to distinguish between a secure program, that can be related to itself under a relation that fulfills the characterization formula, and an insecure program, that can not be related to itself under a relation that fulfills the characterization formula.

Example:
Let *pol* be a mls policy with:
$\mathcal{D} = \{low, high\}$
$low \leq low$, $low \leq high$, $high \leq high$
$dom(lvar) = low$
$dom(hvar) = high$.

```
if (hvar)
then
  lvar := 1
else
  lvar := 0
fi
```

is not strongly secure. We can construct a counter example that violates the formula for *low*-bisimulations by choosing two states $s_1(lvar) = s_2(lvar)$, $s_1(hvar) = 0$ and $s_2(hvar) = 1$. Since $s_1(lvar) = s_2(lvar)$, $s_1 =_{low} s_2$ holds. After executing the if command with s_1 the remaining command vector is $lvar := 0$, but after executing the if command with s_2 the remaining command vector is $lvar := 1$. If we now perform the step in both cases the new mappings for $lvar$ are $s_1'(lvar) = 0$ and $s_2'(lvar) = 1$. In consequence $s_1' \neq_{low} s_2'$ and therefore no strong *low*-bisimulation can exist that relates the program to itself, which means the program is not strongly secure.

While STRONG-Security is the inspirational foundation for the other bisimulaion based properties in this work, it does not support multi-level security policies, but the other properties support them. This issue was addressed in [MS04] where the authors also introduce a property for controlling **where** declassification occurs. Basically, the idea is to lift the *low*-bisimulation to multi-level security policies by parameterizing the equivalence relation of memory states with security domains and quantify over all security domains. We will use the approach with the mls policies and quantify over all security domains in the policies.

2.7. Information Leaks and Declassification

An information leak is a violation of the information flow as described in a security policy. As we have already informally showed, STRONG-Security from [SS00] is a property that detects such information leaks. In some situations, the information leaks may be intended for the functionality and in such a situation a property that requires complete independence as STRONG-Security is overly restrictive.

Example:
Let *pol* be a mls policy with:
$\mathcal{D} = \{low, high\}$
$low \leq low$, $low \leq high$, $high \leq high$
$dom(invar) = dom(outvar) = low$
$dom(password) = high$.
Let $\overrightarrow{C}$ be a simple password check:

```
if ( invar = password ) then
  outvar := 1
else
  outvar := 0
fi
```

This simple password check can not be modified to fulfill STRONG-Security, because the *low*-observer can observe the success of the action that is guarded by the password check and the success of the password check depends on the *high* information stored in *password*. It is reasonable that *password* should be *high* data, because intuitively we do not want that anybody can learn the password. Furthermore, it is reasonable that *outvar* is *low*, since there are situations in which the success of the password check needs

to be revealed, for example when a login depends on the password check. In consequence, we need the dependency of *outvar* from $invar = password$ and in consequence of the *low* information from the *high* information for the functionality.

The intentional release of information from a higher domain to a lower domain is called declassification. In many cases different declassifications may occur in different parts of the program. In consequence it is hard to check manually if some guidelines for declassification are obeyed in the whole program. The presence of multi-threading makes the problem of manual checking even harder, because the order of the commands executed does not only depend on the program, but on the scheduling algorithm. Even in the case, where declassifications may only occur in a specific part of the program, it would not be very wise to trust a part of the program without a proper analysis to declassify information as intended, because very subtle information leaks via transitive assignments or even the scheduling order could occur.

Therefore it would be good to have security policies that specify **where** declassifications occur and **what** is declassified. On the foundation of such policies it would be possible to define security properties that should capture the intuition of non-interference, but allow controlled exceptions as needed for the functionality. With proper formalizations of those properties it would then be possible to develop automatic analyses that check programs for preservation of these properties.

3. Scheduler-independence of WHERE-Security and WHAT-Security

As we have already seen in Section 2.7 properties that require complete independence of public information from secret information are too restrictive in many situations. The simple example with the password check could not be classified as secure with respect to STRONG-Security, because this example requires declassification. Since we still want to have guarantees about confidentiality, it is reasonable that an exact specification of **where** declassifications may occur and **what** may be declassified is possible and that the security properties include adequate controls for these properties. The properties WHERE-Security and WHAT-Security from [MR07] achieve such controls.

Additionally, in a multi-threaded setting the real sequence of execution steps relies not only on the programs, but also on the scheduler, because the scheduler decides which thread may perform an execution step next. This can lead to very subtle information leaks. Furthermore, the scheduler is in many cases unknown before executing the program or different instances of the program may be executed with different schedulers and in consequence minimum assumptions should be made about the scheduler. Scheduler-independence of the security properties is therefore a reasonable goal. The property STRONG-Security from [SS00] achieves scheduler-independence with respect to the wide class of *low*-secure σ-schedulers.

In this section we will give an insight on the intuition of the properties WHERE-Security and WHAT-Security to strengthen the confidence in these properties and furthermore show that these properties are scheduler-independent with respect to the class of *low*-secure σ-schedulers and therefore are scheduler-independent in the same sense as STRONG-Security.

3.1. WHERE-Security

The first property controls the aspect **where** declassification may occur and is therefore called WHERE-Security. The foundation for this property is laid in [MS04] and the property was defined in [MR07]. Like the inspirational foundation STRONG-Security the property uses partial equivalence relations that describe bisimulations to describe what security means.

3.1.1. Integration of where into the Policies and the Language

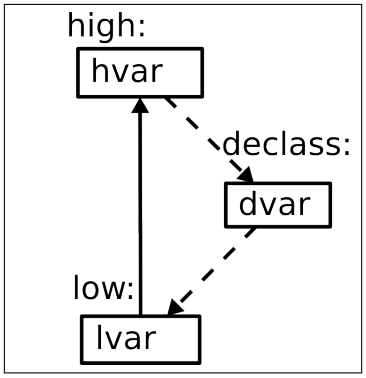
Definition 13 ***Multilevel Security Policy with* where *Exceptions:***
A Multilevel Security Policy with **where** Exceptions *(brief: mls-$\rightsquigarrow$ policy is a tuple $(\mathcal{D}, \leq, \rightsquigarrow, dom)$, where $(\mathcal{D}, \leq, dom)$ is a multilevel security policy and $\rightsquigarrow \subseteq \mathcal{D} \times \mathcal{D}$ is a binary relation on security domains.*

This definition of security domains captures the intuition that the information flow rules can be described as we have already seen in Section 2.4. We will refer to information flow that obeys the partial order $\leq$ as regular information flow.

Additionally, the mls-$\rightsquigarrow$ policies use the relation $\rightsquigarrow$ between which security domains, in other words **where** in the policy, declassifications may occur. In opposite to the partial order $\leq$, this relation is not required to be reflexive or transitive. The intuition behind this is that information may only be declassified from domain $D_1 \in \mathcal{D}$ to domain $D_2 \in \mathcal{D}$ if $(D_1, D_2) \in \rightsquigarrow$. Since the relation is not transitive and not reflexive this allows an exact localization of declassifications in the policy and follows the idea of intransitive noninterference from [MS04]. We will refer to information flow that obeys $\rightsquigarrow$ as exceptional information flow. Furthermore, we will use the infix notation $D_1 \rightsquigarrow D_2$ to denote that $(D_1, D_2) \in \rightsquigarrow$.

Example:

Let *pol* be the following mls-$\rightsquigarrow$ policy:
$\mathcal{D} = \{low, high, declass\}$
$low \leq low$, $low \leq high$, $high \leq high$
$high \rightsquigarrow declass$, $declass \rightsquigarrow low$
$dom(hvar) = high$
$dom(lvar) = low$
$dom(dvar) = declass$.
This policy describes that regular information flow may only occur from *low* to *high*. Regular information flow from *high* to *low*, as well as any regular information flow from or to *declass* is not allowed. The relation $\rightsquigarrow$ allows exceptional information flow from *high* to *declass*, but not to *low*, and from *declass* to *low*. While a direct assignment of *high* information to a *low* variable is not allowed, it is allowed that *high* information is assigned to a *declass* variable and that the information stored in this variable is assigned to a variable of the domain *low*.



In addition to the localization of the declassifications in the policy, a localization in the program code is desirable. In order to achieve this, the authors of [MR07] introduced square brackets as a syntactic construct that encloses an assignment to denote that this assignment may be used for declassification. These brackets are an extension of the security policy into the program code, because the enclosing of an assignment has the meaning that declassifications are allowed in this assignment. We will call such an assignment declassification assignment in the rest of the work. With this new construct the syntax of the MWL changes to the syntax in Figure 5.

Furthermore, the authors of [MR07] extended the semantics of the MWL by adding rules to capture the new syntactic construct. Figure 6 shows these new semantics rules.

The first novelty of those semantics rules is that the executions steps of declassifications use a d as index to distinguish them from the other execution steps in order to be able to localize the declassifications in the semantics. We will call executions steps of declassifications declassification steps and all other executions steps ordinary steps. Since the semantics rule is bound to the syntax this distinction between ordinary steps and declassification steps describes **where** the declassification occurs in the program code.

$$
\begin{array}{rcl}
Com & :== & Com; Com \mid \textbf{skip} \\
 & \mid & \textbf{if } (\ Expr\ ) \textbf{ then } Com \textbf{ else } Com \textbf{ fi} \\
 & \mid & \textbf{while } (\ Expr\ ) \textbf{ do } Com \textbf{ od} \\
 & \mid & Var := Expr \\
 & \mid & [\ Var := Expr\ ] \\
 & \mid & \textbf{fork } (\ Com\ \overrightarrow{Com}\ )
\end{array}
$$

Figure 5: Grammar for the Multi-threaded While Language with **where** Exceptions

$$
\frac{\langle\!|exp, s|\!\rangle \downarrow n \quad sources(expr) = \mathcal{D}_1 \quad dom(var) = D_2}{\langle\!|[var{:=}expr], s|\!\rangle \rightarrow_{\mathrm{d}}^{\mathcal{D}_1 \to D_2} \langle\!|\langle\rangle, s \otimes \{var = n\}|\!\rangle} \qquad \frac{\langle\!|C_1, s|\!\rangle \rightarrow_{\mathrm{d}}^{\mathcal{D}_1 \to D_2} \langle\!|\langle\rangle, s'|\!\rangle}{\langle\!|C_1; C_2, s|\!\rangle \rightarrow_{\mathrm{d}}^{\mathcal{D}_1 \to D_2} \langle\!|C_2, s'|\!\rangle}
$$

Figure 6: Operational semantics for threads in the Multi-threaded While Language with **where** Exceptions

The rule for declassification assignments says that the memory state should be updated exactly as in the normal assignments, but additionally we track the domain of the target variable and the source domains of the expression. Since the information that is assigned to the target variable depends on the source domains of the expression this describes **where** the declassification occurs in the policy.

The rule for sequences on the right side allows us to use declassification steps in sequences and keeps track of the information that we generated with the help of the left rule.

3.1.2. Definition and Intuition of WHERE-Security

The definition of WHERE-Security in [MR07] follows the partial equivalence approach used in STRONG-Security in [SS00].

Definition 14 ***WHERE-Security:***
A strong $(D, \rightsquigarrow)$-bisimulation *is a symmetric relation R on thread pools of equal size that satisfies the entire formula in Figure 7. The relation $\approx_D^{\rightsquigarrow}$ is the union of all strong $(D, \rightsquigarrow)$-bisimulations. A program $\overrightarrow{C}$ has secure information flow while complying with the restrictions* where *declassification can occur if $\overrightarrow{C} \approx_D^{\rightsquigarrow} \overrightarrow{C}$ holds for all $D \in \mathcal{D}$ (brief: $\overrightarrow{C}$ is* WHERE-secure *or $\overrightarrow{V} \in WHERE$).*

If we remove the part after $\vee$ from the characterization formula in Figure 7 we get a formula similar to the formula in the definition of STRONG-Security. Due to the disjunction on the right side of the implication, the requirement that the resulting memory states must be indistinguishable from STRONG-Security is weakened and thereby declassifications are made possible.

The intuition behind the disjunction is that the execution step either does not reveal information, which is expressed by requiring $s_1' =_D s_2'$ as in STRONG-Security, or a

$$
\begin{aligned}
&\forall s_1, s_2, s_1' : \forall \overrightarrow{W}_1 : \forall i \in \{1 \dots n\} : \\
&(\overrightarrow{C}_1 R \overrightarrow{C}_2 \wedge \langle\!| C_{1,i}, s_1 |\!\rangle \twoheadrightarrow \langle\!| \overrightarrow{W}_1, s_1' |\!\rangle \wedge s_1 =_D s_2) \\
&\Longrightarrow \exists \overrightarrow{W}_2 : \exists s_2' : \overrightarrow{W}_1 R \overrightarrow{W}_2 \wedge \langle\!| C_{2,i}, s_2 |\!\rangle \twoheadrightarrow \langle\!| \overrightarrow{W}_2, s_2' |\!\rangle \\
&\qquad \wedge \left[\begin{array}{l} s_1' =_D s_2' \\ \vee \left[\begin{array}{l} \exists \mathcal{D}_1, \{D_2\} \subseteq \mathcal{D} : \\ \left[\begin{array}{l} \langle\!| C_{1,i}, s_1 |\!\rangle \twoheadrightarrow_{\mathrm{d}}^{\mathcal{D}_1 \to D_2} \langle\!| \overrightarrow{W}_1, s_1' |\!\rangle \\ \wedge \forall D_1 \in \mathcal{D}_1 : (D_1 \leq D_2 \vee D_1 \rightsquigarrow D_2) \\ \wedge D_2 \leq D \wedge \exists D' \in \mathcal{D}_1 : s_1 \neq_{D'} s_2 \end{array} \right] \end{array} \right] \end{array} \right]
\end{aligned}
$$

Figure 7: Characterization of WHERE-Security with $\overrightarrow{C}_i = \langle C_{i,1}, \dots, C_{i,n} \rangle$ for $i \in \{1, 2\}$

declassification step occurred. In the second case the declassification is controlled with respect to the security policy by the right part of the disjunction in the last four lines of the formula.

As we have already mentioned, declassifications should only occur in declassification assignments. According to our program semantics, a declassification assignment is resolved with a declassification step in the semantics. Hence $\langle\!| C_{1,i}, s_1 |\!\rangle \twoheadrightarrow_{\mathrm{d}}^{\mathcal{D}_1 \to D_2} \langle\!| \overrightarrow{W}_1, s_1' |\!\rangle$ in line 6 of the formula requires that the declassification occurs in a declassification assignment and thus the declassification occurs in a location in the code where declassifications are allowed. This restriction of declassifications to declassification assignments results in a control of **where** in the program code the declassifications may occur.

Example:
Let *pol* be a mls-$\rightsquigarrow$ policy with:
$low \leq low$, $low \leq high$, $high \leq high$
$high \rightsquigarrow low$
$dom(lvar) = low$
$dom(hvar) = high$.

Let $\overrightarrow{C}$ be a program that contains:

```
lvar := hvar
```

While the security policy allows declassification from security domain *high* to domain *low*, the program is not WHERE-secure, because the declassification does not occur in a special declassification assignment. Since it is not a declassification assignment, we can always obtain a counter example by using two states $s_1 =_{low} s_2$, but $s_1(hvar) \neq s_2(hvar)$ and in consequence $s_1' \neq_{low} s_2'$ after executing the assignment. Furthermore, since the assignment is no special declassification assignment
$\exists \mathcal{D}_1, \{D_2\} \subseteq \mathcal{D} : \langle\!| C_{1,i}, s_1 |\!\rangle \twoheadrightarrow_{\mathrm{d}}^{\mathcal{D}_1 \to D_2} \langle\!| C_{1,i}', s_1' |\!\rangle$ is not fulfilled and therefore no strong $(D, \rightsquigarrow)$-bisimulation can exist that relates this program to itself.

The declassifications should not only be restricted with respect to **where** in the program code, but also **where** in the policy the declassifications occur. The semantics for the declassification steps tell us that $\mathcal{D}_1$ must be $sources(expr)$ and therefore contains all the domains that information in the expression may originate from and that D_2 must be the domain of the target variable $dom(var)$. In consequence, we can say that information flows from the source domains $\mathcal{D}_1$ to the target domain D_2. According to our intuition of the mls-$\rightsquigarrow$ policies we want to allow exceptional information flow only from a domain D_1 to a domain D_2, if $D_1 \rightsquigarrow D_2$. Intuitively, information flow that is allowed in ordinary steps by regular information flow should also be allowed in declassification steps. The requirement $\forall D_1 \in \mathcal{D}_1 : (D_1 \leq D_2 \vee D_1 \rightsquigarrow D_2)$ in line 7 of the characterization formula captures the intuition that in a declassification the information flow should obey the rules of exceptional and regular information flow and therefore controls **where** in the security policy declassifications may occur.

Example:
Let *pol* be a mls-$\rightsquigarrow$ policy with:
$low \leq low$, $low \leq high$, $high \leq high$
$\rightsquigarrow = \emptyset$
$dom(lvar) = low$
$dom(hvar) = high$

Let $\overrightarrow{C}$ be a program that contains:

```
[ lvar := hvar ]
```

This policy does not allow any exceptional information flow. The declassification now occurs in a declassification assignment, but the program is not WHERE-secure, because the information is declassified from *high* to *low* and that is not allowed by the policy. We can construct a counter example similar to the previous counter example. This time the declassification occurs in a declassification step and thus $\exists \mathcal{D}_1, \{D_2\} \subseteq \mathcal{D} : \langle\!| C_{1,i}, s_1 |\!\rangle \rightarrow_{\mathrm{d}}^{\mathcal{D}_1 \to D_2} \langle\!| C'_{1,i}, s'_1 |\!\rangle$ can be fulfilled with $\mathcal{D}_1 = \{high\}$ and $D_2 = low$, but $\forall D_1 \in \mathcal{D}_1 : (D_1 \leq D_2 \vee D_1 \rightsquigarrow D_2)$ can not be fulfilled, since $low \leq high$ and $\rightsquigarrow = \emptyset$ and, in consequence, no strong $(D, \rightsquigarrow)$-bisimulation can exist that relates this program to itself.

The last line of the **where** control is not necessary to capture our intuition, but instead is an additional check for the annotations of the declassification steps. In the language that we use it is easy to determine the target and the source domains, since the left hand expression of an assignment is just a variable and the right hand side is just a combination of variables, but in other languages the determination of these levels could be harder.

The first part of the last line, $D_2 \leq D$, requires that the target domain is indeed visible and in consequence the assignment to this domain can render two states distinguishable. This is an additional check for the annotation that describes the target domain.

The second part of the last line, $\exists D' \in \mathcal{D}_1 : s_1 \neq_{D'} s_2$, requires that the two states before executing the declassification already were distinguishable for at least one domain

in the source domains. Since the states were already distinguishable for a source domain it is reasonable that an information flow from this domain to the target domain can render the states indistinguishable. This is an additional check for the annotation that describes the source domains.

This property captures our intuition of security and the control of declassification with respect to the aspect **where** very closely and therefore we are confident that the property is adequate and faithful to our intuition.

3.1.3. Scheduler-independence of WHERE-Security

After the detailed reflection of WHERE-Security we want to show that the property is scheduler-independent with respect to the class of *low*-secure σ-schedulers in the next section and therefore is adequate in a multi-threaded setting. The authors of [SS00] presented an inspiring scheduler-independence proof for STRONG-Security that we can adapt for WHERE-Security.

We extend the thread pool semantics to include the information about declassifications to be able to distinguish between ordinary scheduler steps and declassification scheduler steps as we already did in the MWL semantics. The new thread pool semantics can be found in Figure 8.

$$\frac{\langle\!|C_i, s|\!\rangle \rightarrow\!\!\triangleright_{\mathrm{o}} \langle\!|\overrightarrow{W}, s'|\!\rangle}{\langle\!|H, \langle C_0 \ldots C_i \ldots C_{n-1}\rangle, s|\!\rangle \xrightarrow{p}_{\mathrm{o}} \langle\!|H(i, n+|\overrightarrow{W}|-1), \langle C_0 \ldots C_{i-1}\overrightarrow{W}C_{i+1} \ldots C_{n-1}\rangle, s'|\!\rangle}$$

$$\frac{\langle\!|C_i, s|\!\rangle \rightarrow\!\!\triangleright_{\mathrm{d}}^{\mathcal{D}_1 \rightarrow D_2} \langle\!|\overrightarrow{W}, s'|\!\rangle}{\langle\!|H, \langle C_0 \ldots C_i \ldots C_{n-1}\rangle, s|\!\rangle \xrightarrow{p}_{\mathrm{d}}^{\mathcal{D}_1 \rightarrow D_2} \langle\!|H(i, n+|\overrightarrow{W}|-1), \langle C_0 \ldots C_{i-1}\overrightarrow{W}C_{i+1} \ldots C_{n-1}\rangle, s'|\!\rangle}$$

Figure 8: Semantics of Thread Pools with Declassification

Definition 15 ***σ-specific WHERE-Security:***
A σ-specific strong $(D, \rightsquigarrow)$-bisimulation *is a symmetric relation R on command vectors of equal size that satisfies the entire formula in Figure 9.The relation $\cong^{\rightsquigarrow}_{D,\sigma}$ is the union of all strong σ-specific $(D, \rightsquigarrow)$-bisimulations. A program $\overrightarrow{C}$ has secure information flow for a given scheduler σ while complying with the restrictions* where *declassification can occur if $\overrightarrow{C} \cong^{\rightsquigarrow}_{D,\sigma} \overrightarrow{C}$ holds for all $D \in \mathcal{D}$ (brief: $\overrightarrow{C}$ is σ-WHERE-secure or $\overrightarrow{C} \in \sigma - WHERE$).*

The characterization formula of σ-specific WHERE-Security uses the system configurations instead of the thread configurations that are used in WHERE-Security to include the scheduling information into the property. The additional information that we gain by using the system configurations are the scheduling histories and the probabilities for a specific thread to be scheduled.

$$
\begin{aligned}
&\forall H_1, H_2, H_1' : \forall \overrightarrow{C}_1' : \forall s_1, s_2, s_1' : \\
&\langle\!\langle H_1, \overrightarrow{C}_1, s_1 \rangle\!\rangle \to \langle\!\langle H_1', \overrightarrow{C}_1', s_1' \rangle\!\rangle \wedge H_1 =_\sigma H_2 \wedge \overrightarrow{C}_1 R \overrightarrow{C}_2 \wedge s_1 =_D s_2 \\
&\Longrightarrow \exists H_2' : \exists \overrightarrow{C}_2' : \exists s_2' : \\
&\quad \langle\!\langle H_2, \overrightarrow{C}_2, s_2 \rangle\!\rangle \to \langle\!\langle H_2', \overrightarrow{C}_2', s_2' \rangle\!\rangle \\
&\quad (i) \wedge \left[\begin{array}{l} H_1' =_\sigma H_2' \wedge \overrightarrow{C}_1' R \overrightarrow{C}_2' \\ \wedge \left[\begin{array}{l} s_1' =_D s_2' \vee \left[\begin{array}{l} \exists \mathcal{D}_1, \{D_2\} \subseteq \mathcal{D} : \\ \left[\begin{array}{l} \langle\!\langle H_1, \overrightarrow{C}_1, s_1 \rangle\!\rangle \twoheadrightarrow_{\mathrm{d}}^{\mathcal{D}_1 \to D_2} \langle\!\langle H_1', \overrightarrow{C}_1', s_1' \rangle\!\rangle \\ \wedge \forall D_1 \in \mathcal{D}_1 : (D_1 \leq D_2 \vee D_1 \rightsquigarrow D_2) \\ \wedge D_2 \leq D \wedge \exists D' \in \mathcal{D}_1 : s_1 \neq_{D'} s_2 \end{array} \right] \end{array} \right] \end{array} \right] \end{array} \right] \\
&\quad (ii) \wedge \left[\begin{array}{l} \sum \{\!|p | \langle\!\langle H_1, \overrightarrow{C}_1, s_1 \rangle\!\rangle \to_p \langle\!\langle H, \overrightarrow{C}, s \rangle\!\rangle, H \in [H_1']_{=_\sigma}, \overrightarrow{C} \in [\overrightarrow{C}_1']_R, s \in [s_1']_{=_D} |\!\} \\ = \sum \{\!|p | \langle\!\langle H_2, \overrightarrow{C}_2, s_2 \rangle\!\rangle \to_p \langle\!\langle H, \overrightarrow{C}, s \rangle\!\rangle, H \in [H_2']_{=_\sigma}, \overrightarrow{C} \in [\overrightarrow{C}_2']_R, s \in [s_2']_{=_D} |\!\} \end{array} \right]
\end{aligned}
$$

Figure 9: Characterization of σ-specific WHERE-Security

We preserve all the requirements of WHERE-Security and transfer them to the system configurations. Our intuition was that the behavior of the program in indistinguishable configurations should be indistinguishable. Since we switched to the system configurations, we require σ-equivalence of the histories for the indistinguishability of states, additionally to the thread pool being in relation with respect to the characterization formula and D-equivalence of the memory states.

We stick to our intuition that an execution step should preserve the indistinguishability of the configurations or a controlled declassification occurred, but replace the execution steps with scheduler steps.

Additionally, to the preservation of the intuition that WHERE-Security introduced, we require that the scheduling does not reveal any further information. Since the scheduler is modeled as a function that returns a probability distribution for the threads to be scheduled, we want that these probabilities do not reveal any additional information. In order to achieve this, we require that a transition from the configuration $\langle\!\langle H_2, \overrightarrow{C}_2, s_2 \rangle\!\rangle$ into the equivalence class of configurations that contains $\langle\!\langle H_2', \overrightarrow{C}_2', s_2' \rangle\!\rangle$ is as probable as a transition from $\langle\!\langle H_1, \overrightarrow{C}_1, s_1 \rangle\!\rangle$ into the equivalence class of configurations that contains $\langle\!\langle H_1', \overrightarrow{C}_1', s_1' \rangle\!\rangle$. In the case where no declassification occurs, the equivalence class that contains $\langle\!\langle H_2', \overrightarrow{C}_2', s_2' \rangle\!\rangle$ is the same equivalence class as $\langle\!\langle H_1', \overrightarrow{C}_1', s_1' \rangle\!\rangle$, since we require that $H_1' =_\sigma H_2'$, $\overrightarrow{C}_1' R \overrightarrow{C}_2'$ and $s_1' =_D s_2'$. In the case where a declassification occurred $s_1' \neq_D s_2'$ holds and therefore the transitions are made into two different equivalence classes.

Since the characterization formula is recursive, the thread pools in the resulting states

must be related under R again and the characterization formula must be fulfilled again. This in turn leads to the restriction that whenever a declassification step is possible in $\langle H_1, \overrightarrow{C}_1, s_1 \rangle$ and therefore removed from $\overrightarrow{C}_1$ then a declassification step must be equally likely in $\langle H_2, \overrightarrow{C}_2, s_2 \rangle$ or otherwise a declassification step would be possible in $\overrightarrow{C}'_2$, but not in $\overrightarrow{C}'_1$ and therefore $\overrightarrow{C}'_1 R \overrightarrow{C}'_2$ would not be be possible.

Since it is possible that different steps lead to indistinguishable configurations from the configurations before the step, we use the sum of the probabilities that lead to the equivalence class of configurations and argue that this sum must be equal for both configurations.

This definition gives us a scheduler-specific property that must be fulfilled in order to be called secure with respect to this property. We can call WHERE-Security scheduler-independent, if every program that is WHERE-secure is σ-WHERE-secure for all *low*-secure σ-schedulers, too.

Theorem 2 ***Scheduler-independence of WHERE-Security:*** *If $\overrightarrow{C}$ is WHERE-secure, then $\overrightarrow{C}$ is σ-WHERE-secure for all σ.*

$$\overrightarrow{C} \approx^{\sim}_{D} \overrightarrow{C} \Longrightarrow \forall \sigma : \overrightarrow{C} \approx^{\sim}_{D,\sigma} \overrightarrow{C}$$

Proof Scheduler-independence of WHERE-Security: In order to proof that WHERE-Security is scheduler-independent, we show that a relation R that fulfills the characterization formula of WHERE-Security also fulfills the characterization formula of σ-specific WHERE-Security for all *low*-secure schedulers σ. So, if we assume that R exists, such that $\overrightarrow{C} R \overrightarrow{C}$ and R fulfills the characterization formula of WHERE-Security given in Figure 7, then the following must hold for R all *low*-secure schedulers σ, too:

$$\begin{aligned}
&\forall H_1, H_2, H'_1 : \forall \overrightarrow{C}'_1 : \forall s_1, s_2, s'_1 : \\
&\langle H_1, \overrightarrow{C}_1, s_1 \rangle \rightarrow \langle H'_1, \overrightarrow{C}'_1, s'_1 \rangle \wedge H_1 =_\sigma H_2 \wedge \overrightarrow{C}_1 R \overrightarrow{C}_2 \wedge s_1 =_D s_2 \\
&\Longrightarrow \exists H'_2 : \exists \overrightarrow{C}'_2 : \exists s'_2 : \\
&\quad \langle H_2, \overrightarrow{C}_2, s_2 \rangle \rightarrow \langle H'_2, \overrightarrow{C}'_2, s'_2 \rangle \\
&\quad (i) \wedge \left[\begin{array}{l} H'_1 =_\sigma H'_2 \wedge \overrightarrow{C}'_1 R \overrightarrow{C}'_2 \\ \wedge \left[\begin{array}{l} s'_1 =_D s'_2 \vee \left[\begin{array}{l} \exists \mathcal{D}_1, \{D_2\} \subseteq \mathcal{D} : \\ \left[\begin{array}{l} \langle H_1, \overrightarrow{C_1}, s_1 \rangle \twoheadrightarrow_{\mathrm{d}}^{\mathcal{D}_1 \to D_2} \langle H'_1, \overrightarrow{C}'_1, s'_1 \rangle \\ \wedge \forall D_1 \in \mathcal{D}_1 : (D_1 \leq D_2 \vee D_1 \rightsquigarrow D_2) \\ \wedge D_2 \leq D \wedge \exists D' \in \mathcal{D}_1 : s_1 \neq_{D'} s_2 \end{array} \right] \end{array} \right] \end{array} \right] \end{array} \right] \\
&\quad (ii) \wedge \left[\begin{array}{l} \sum \{\!|p | \langle H_1, \overrightarrow{C}_1, s_1 \rangle \rightarrow_p \langle H, \overrightarrow{C}, s \rangle, H \in [H'_1]_{=_\sigma}, \overrightarrow{C} \in [\overrightarrow{C}'_1]_R, s \in [s'_1]_{=_D} |\!\} \\ = \sum \{\!|p | \langle H_2, \overrightarrow{C}_2, s_2 \rangle \rightarrow_p \langle H, \overrightarrow{C}, s \rangle, H \in [H'_2]_{=_\sigma}, \overrightarrow{C} \in [\overrightarrow{C}'_2]_R, s \in [s'_2]_{=_D} |\!\} \end{array} \right]
\end{aligned}$$

First, we want to show that (ii) holds. If the left side of the implication is fulfilled we know that $s_1 =_D s_2$ and can deduce $s_1 =_{low} s_2$, because $low \leq D$ holds,

since *low* is the minimum of $\mathcal{D}$ with respect to $\leq$ per definition. Using Theorem 1 about the probability equivalence for equivalent histories and states we can deduce that $\sigma(H_1, s_1) =_\sigma \sigma(H_2, s_2)$. In combination with the quantification over all the threads in the thread pools and the requirements on the execution steps in the characterization formula of WHERE-Security we can deduce that a one-to-one correspondence exists between the elements of the multi sets and in consequence the sum of the elements of the multi sets is equal and (ii) is fulfilled.

Second, we want to show that (i) holds. We know from the thread pool semantics in Figure 8 that the execution step

$$\langle\!|C_{1,i}, s_1|\!\rangle \twoheadrightarrow \langle\!|\overrightarrow{W}_1, s_1'|\!\rangle$$

triggers a scheduler step

$$\langle\!|H_1, \overrightarrow{C}_1, s_1|\!\rangle \twoheadrightarrow \langle\!|H_1(i, n + |\overrightarrow{W}_1| - 1, \langle C_{1,0} \dots C_{1,i-1} \overrightarrow{W}_1 C_{1,i+1} \dots C_{1,n-1}\rangle, s_1'|\!\rangle$$

and deduce that $H_1' = H_1(i, n+|\overrightarrow{W}_1|-1)$ and $\overrightarrow{C}_1' = \langle C_{1,0} \dots C_{1,i-1} \overrightarrow{W}_1 C_{1,i+1} \dots C_{1,n-1}\rangle$. From the characterization formula of WHERE-Security we know that

$$\langle\!|C_{2,i}, s_2|\!\rangle \twoheadrightarrow \langle\!|\overrightarrow{W}_2, s_2'|\!\rangle$$

exists and can use the thread pool semantics again to deduce that this execution step triggers a scheduler step

$$\langle\!|H_2, \overrightarrow{C}_2, s_2|\!\rangle \to \langle\!|H_2(i, n + |\overrightarrow{W}_2| - 1), \langle C_{2,0} \dots C_{2,i-1} \overrightarrow{W}_2 C_{2,i+1} \dots C_{2,n-1}\rangle, s_2'|\!\rangle$$

and in consequence $H_2' = H_2(i, n + |\overrightarrow{W}_2| - 1)$ and
$\overrightarrow{C}_2' = \langle C_{2,0} \dots C_{2,i-1} \overrightarrow{W}_2 C_{2,i+1} \dots C_{2,n-1}\rangle$.
From the definition of WHERE-Security we know that $|\overrightarrow{C}_1| = |\overrightarrow{C}_2| = n$ and $|\overrightarrow{W}_1| = |\overrightarrow{W}_2|$. In consequence, $n+|\overrightarrow{W}_1|-1 = n+|\overrightarrow{W}_2|-1$. From the left side of the implication we know that $H_1 =_\sigma H_2$ and with the Definition 11 of σ-equivalence, we can conclude that

$$H_1' = H_1(i, n + |\overrightarrow{W}_1| - 1) =_\sigma H_2(i, n + |\overrightarrow{W}_2| - 1) = H_2'$$

We can use the quantification over all threads in the thread pool in the characterization formula of WHERE-Security to deduce that all execution steps that can be made by a single thread fulfill the characterization formula of WHERE-Security point-wise and therefore $C_{1,i} R C_{2,i}$ for all i. Furthermore, we know from the the characterization formula that $\overrightarrow{W}_1 R \overrightarrow{W}_2$. Using the quantification over the threads again, we can deduce that $\langle C_{1,0} \dots C_{1,i-1} \overrightarrow{W}_1 C_{1,i+1} \dots C_{1,n-1}\rangle R \langle C_{2,0} \dots C_{2,i-1} \overrightarrow{W}_2 C_{2,i+1} \dots C_{2,n-1}\rangle$ which in turn means $\overrightarrow{C}_1' R \overrightarrow{C}_2'$.

In the case where the resulting memory states are indistinguishable, $s_1' =_D s_2'$ we have the same requirement in WHERE-Security and σ-specific WHERE-Security. In the case where the resulting memory states are distinguishable, we require the existence of

an execution step that belongs to a declassification assignment and have the restrictions defined using the information that is calculated in the semantics of the declassification assignment. According to the thread pool semantics, this declassification step triggers a scheduler step

$$\langle\!\langle H_1, \overrightarrow{C}_1, s_1 \rangle\!\rangle \rightarrow_{\mathrm{d}}^{\mathcal{D}_1 \to D_2} \langle\!\langle H_1(i, n + |\overrightarrow{W}_1| - 1), \langle C_{1,0} \ldots C_{1,i-1} \overrightarrow{W}_1 C_{1,i+1} \ldots C_{1,n-1} \rangle, s_1' \rangle\!\rangle$$

Since $\mathcal{D}_1$, D_2 are the same as in the execution step and the requirements defined on them are equivalent, the requirements in the characterization formula of σ-specific WHERE-Security are fulfilled, too.

Since the characterization formula of σ-specific WHERE-Security is fulfilled for any arbitrary scheduler, WHERE-Security is scheduler-independent.

■

This is an important result, because we know now that WHERE-Security guarantees that an attacker may not learn any additional information depending on the scheduler choice as long as it is a *low*-secure σ-scheduler. Furthermore, to our knowledge this is the first scheduler-independence result for a security property that allows declassification. This strengthens our confidence in the adequacy of the property for multi-threaded settings and therefore to be a good foundation for our integrated property.

3.2. WHAT-Security

The second property controls declassifications with respect to the aspect **what** may get declassified. In fact WHAT-Security are two properties that are very closely related, since they use the same characterization formula. The two properties have subtle, but very important differences in their definition. The properties use the bisimulations with partial equivalence relations approach and look very similar to STRONG-Security with the exception of declassifications being possible.

3.2.1. Integration of what into the Policies and Attacker Model

Definition 16 ***Multilevel Security Policy with* what *Exceptions:***
A Multilevel Security Policy with **what** Exceptions *(brief: mls-$\mathcal{H}$ policy) is a tuple $(\mathcal{D}, \leq, dom, \mathcal{H})$, where $(\mathcal{D}, \leq, dom)$ is a multilevel security policy and $\mathcal{H} \subseteq \mathcal{D} \times Expr$ is a set of pairs of security domains and expressions.*

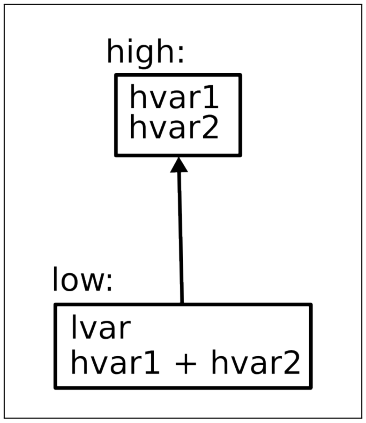
This definition preserves the possibilities to describe the information flow rules as we have already seen in Section 2.4. We will refer to information flow that obeys $\leq$ as regular information flow.

Additionally, the mls-$\mathcal{H}$ policies use the set $\mathcal{H}$ to describe **what** may get declassified. We call an element of $(D, expr) \in \mathcal{H}$ an escape hatch. The intuition of an escape hatch is to denote that an observer is allowed to learn the information that the expression evaluates to, if the domain D is visible for him. More formally, the observer is allowed

to learn n where $\langle\!|expr, s|\!\rangle \Downarrow n$ for a given s, if the domain D is visible for him. The formal definition of an evaluation of an expression shows us explicitly that the evaluation does not only depend on the expression, but also on the memory state. In consequence, the information that an attacker learns also relies on the memory state. It is possible to restrict **what** not only to the expression, but also to the memory state of a specific execution point. We call those execution points reference points and distinguish between the initial reference point that is given by the initial memory state and local reference points that are memory states somewhere during the execution of the program. We call information flow that is allowed due to an escape hatch exceptional.

Example:

Let *pol* be the following mls-$\mathcal{H}$ policy:
$\mathcal{D} = \{low, high\}$
$low \leq low$, $low \leq high$, $high \leq high$
$dom(lvar) = low$
$dom(hvar1) = dom(hvar2) = high$
$\mathcal{H} = \{(low, hvar1 + hvar2)\}$
This policy describes that regular information flow may occur from the domain *low* to the domains *low* and *high* and from the domain *high* to *high*. Additionally, the information $n = hvar1 + hvar2$ may get declassified to domain *low* and therefore this information may flow exceptionally. The information stored in $hvar1$ or $hvar2$ on the other hand may not flow the security domain *low* individually, since only the combined information occurs in an escape hatch in $\mathcal{H}$.



Since we explicitly describe **what** we allow an observer to learn, we extend the attacker model to capture this additional information.

Definition 17 (D, H)***-Equality:***
Two states s_1 and s_2 are (D, H)-equal *with respect to a security domain D and a set of escape hatches H (denoted: $s1 =_D^H s2$), if the states are D-equal and all expression in escape hatches that have a lower domain than D evaluate to equal values.*

$$s_1 =_D^H s_2 \iff s_1 =_D s_2 \wedge \forall (D', expr) \in H : ((\langle\!|expr, s_1|\!\rangle \Downarrow m \wedge \langle\!|expr, s_2|\!\rangle \Downarrow n) \implies (m = n))$$

Since the intuition of the mls-$\mathcal{H}$ policies is that an observer is allowed to learn the values the expressions in escape hatches evaluate to, if the domain of the escape hatch is visible, an observer is allowed to distinguish D-equal states due to the evaluation of the expressions in visible escape hatches. In consequence, we partition the set of memory states not only on the basis of the domains of variables, but also on the evaluated expressions of visible escape hatches to capture this capability of the attacker.

3.2.2. Definition and Intuition of WHAT-Security

As already mentioned, the authors of [MR07] introduced two different properties WHAT$_1$-Security and WHAT$_2$-Security that use the partial equivalence relation approach we already know from STRONG-Security and WHERE-Security.

Definition 18 ***WHAT$_1$-Security [MR07]:***
A strong $(D, \mathcal{H})$-bisimulation *is a symmetric relation R on command vectors of equal size that satisfies the formula in Figure 10. The relation $\approx_D^{\mathcal{H}}$ is the union of all strong $(D, \mathcal{H})$-bisimulations. A program $\overrightarrow{V}$ has secure information flow while complying with the restrictions* **what** *can be declassified if* $\forall D : \overrightarrow{V} \approx_D^{\mathcal{H}} \overrightarrow{V}$
(brief: $\overrightarrow{V}$ is WHAT$_1$-secure or $\overrightarrow{V} \in$ WHAT$_1$).

Definition 19 ***WHAT$_2$-Security [MR07]:***
A program $\overrightarrow{V}$ has secure information flow while complying with the restrictions **what** *can be declassified if* $\forall D : \exists \mathcal{H}' \subseteq \mathcal{H} : \overrightarrow{V} \approx_D^{\mathcal{H}'} \overrightarrow{V}$
(brief: $\overrightarrow{V}$ is WHAT$_2$-secure or $\overrightarrow{V} \in$ WHAT$_2$).

$$
\begin{aligned}
&\forall s_1, s_2, s_1' : \forall \overrightarrow{W}_1 : \forall i \in \{1 \dots n\} : \\
&(\overrightarrow{C}_1 R \overrightarrow{C}_2 \wedge \langle\!| C_{1,i}, s_1 |\!\rangle \twoheadrightarrow \langle\!| \overrightarrow{W}_1, s_1' |\!\rangle \wedge s_1 =_D^{\mathcal{H}} s_2) \\
&\Longrightarrow \exists \overrightarrow{W}_2 : \exists s_2' : \langle\!| C_{2,i}, s_2 |\!\rangle \twoheadrightarrow \langle\!| \overrightarrow{W}_2, s_2' |\!\rangle \wedge \overrightarrow{W}_1 R \overrightarrow{W}_2 \wedge s_1' =_D^{\mathcal{H}} s_2'
\end{aligned}
$$

Figure 10: Characterization of WHAT-Security with $\overrightarrow{C}_i = \langle C_{i,1}, \dots, C_{i,n} \rangle$ for $i \in \{1, 2\}$

The characterization formula that both properties use looks very similar to STRONG-Security. In fact, the only difference are the equivalence classes of memory states that are used. Where STRONG-Security uses the equivalence classes induced by D-equivalence, WHAT$_1$-Security and WHAT$_2$-Security use the equivalence classes induced by (D, H)-equivalence. This captures our intuition, that an observer is allowed to learn the values the expressions in the escape hatches evaluate to and therefore can distinguish the states on the basis of this knowledge.

The replacement on the left side of the implication $s_1 =_D^{\mathcal{H}} s_2$ enables the use of declassification, because (D, H)-equality is stricter and therefore less pairs of resulting states are constrained by the right side of the implication. The use of the stricter relation on the left side captures the intuition that an D-observer is allowed to distinguish the states s_1 and s_2 either due to the states being not D-equal, or due to the existence of at least one visible (formally: $D' \leq D$) escape hatch $(D', expr)$ that does not evaluate to the same value under both memory states (formally: $\langle\!| expr, s_1 |\!\rangle \Downarrow m \wedge \langle\!| expr, s_2 |\!\rangle \Downarrow n \wedge m \neq n$).

Example:
Let *pol* be the following mls-$\mathcal{H}$ policy:
$\mathcal{D} = \{low, high\}$
$low \leq low$, $low \leq high$, $high \leq high$
$dom(lvar) = low$
$dom(hvar1) = dom(hvar2) = high$
$\mathcal{H} = \{(low, hvar1 + hvar2)\}$

Let lvar := hvar1 + hvar2 be part of the program $\overrightarrow{C}$. Intuitively, this program line should not violate the security property, since the declassification of $hvar1 + hvar2$ to low is allowed by the policy. In fact, it is not possible to construct a counter example for this line, because we would require that $s_1 =_D^{\mathcal{H}} s_2$, but $s_1' \neq_D^{\mathcal{H}} s_2'$. If $s_1 =_D^{\mathcal{H}} s_2$, we know that all expressions in escape hatches that may be learned by a D-observer must evaluate to equal values under s_1 and s_2 and therefore the assignment in this line would lead to a state where only the mapping for lvar is changed, but it is changed such that $s_1'(lvar) = s_2'(lvar)$ and thus $s_1' =_D^{\mathcal{H}} s_2'$ is fulfilled.

The replacement on the right side of the implication $s_1' =_D^{\mathcal{H}} s_2'$ introduces a control of information flow into the escape hatches. By doing so it restricts the changes in the memory state such that the equivalence of values from evaluating expressions in escape hatches is preserved. This captures the intuition that an attacker is not allowed to distinguish the states after performing an execution step with the knowledge he is allowed to have, which are the values of the visible variables and the evaluations of the visible escape hatches, if he could not distinguish the states before the execution step on this basis. The consequence is that the properties WHAT_1-Security and WHAT_2-Security implicitly use an initial reference point.

Example:
Let *pol* be the following mls-$\mathcal{H}$ policy:
$\mathcal{D} = \{low, high\}$
$low \leq low$, $low \leq high$, $high \leq high$
$dom(lvar) = low$
$dom(hvar1) = dom(hvar2) = high$
$\mathcal{H} = \{(low, hvar1 + hvar2)\}$

Let hvar1 := hvar2 be part of the program $\overrightarrow{C}$. Intuitively, this program line should violate the security property, if we assume initial reference points. A D-observer is only allowed to learn $hvar1 + hvar2$ via a declassification, but after the assignment, the attacker could learn $hvar2$ instead via a legal declassification $lvar := hvar1 + hvar2$, since $hvar1 = hvar2$ and therefore $hvar1 + hvar2 = hvar2 + hvar2$. Indeed, this program line violates WHAT_1-Security and WHAT_2-Security, if a declassification of the form lvar1 := hvar1 + hvar2 is part of the program. We can construct a counter example by choosing $s_1 =_D s_2$ and $s_1(hvar1) = s_2(hvar2) = 0$ and $s_1(hvar2) = s_2(hvar1) = 1$. In this case, $s_1 =_{low}^{\mathcal{H}} s_2$ is fulfilled, since $s_1(hvar1) + s_1(hvar2) = 0 + 1 = 1 + 0 = s_2(hvar1) + s_2(hvar2)$, but after executing the assignment hvar1 := hvar2 , we would

have $s_1'(hvar1)+s_1'(hvar2) = 1+1 \neq 0+0 = s_2'(hvar1)+s_2'(hvar2)$ and in consequence $s_1' \neq^{\mathcal{H}}_{low} s_2'$.

The properties WHAT$_1$-Security and WHAT$_2$-Security do not differ in the characterization formula, but in the set of escape hatches that is used for inducing the equivalence classes of memory states. While in WHAT$_1$-Security the whole set of escape hatches is used, which probably contains escape hatches that never occur in the program as declassifications, WHAT$_2$-Security uses the subset of $\mathcal{H}$ that contains only escape hatches with expressions that actually occur as declassifications.

Example:
Let *pol* be the following mls-$\mathcal{H}$ policy:
$low \leq low$, $low \leq high$, $high \leq high$
$dom(lowvar) = low$
$dom(hvar1) = dom(hvar2) = high$
$\mathcal{H} = \{(low, hvar1 + hvar2), (low, hvar1 + hvar3)\}$.

Let $\overrightarrow{C}$ be the following program:

```
hvar3 := hvar1
lowvar := hvar1 + hvar2
```

This program does not fulfill WHAT$_1$-Security. Assume two states s_1 where $s_1(lowvar) = s_2(lowvar)$, $s_1(hvar1) = 0 = s_2(hvar2) = s_2(hvar3)$ and $s_1(hvar2) = s_1(hvar3) = 1 = s_2(hvar1)$. Since $s_1(hvar1) + s_1(hvar2) = 0 + 1 = s_2(hvar2) + s_2(hvar1)$ and $s_1(hvar1) + s_1(hvar3) = 0 + 1 = s_2(hvar3) + s_2(hvar1)$ holds, $s_1 =^{\mathcal{H}}_{low} s_2$ is fulfilled. After executing hvar3 := hvar1 we have $s_1'(hvar3) = 0$ and $s_2'(hvar3) = 1$. In consequence, $s_1'(hvar1) + s_1'(hvar3) = 0 + 0 \neq 1 + 1 = s_2'(hvar3) + s_2'(hvar1)$ and as a result $s_1' \neq^{\mathcal{H}}_{low} s_2'$.
However the program fulfills WHAT$_2$-Security, because the definition requires only $\exists \mathcal{H}' \subseteq \mathcal{H}$ such that the characterization formula is fulfilled. For our counter example $\mathcal{H}'$ would be $\mathcal{H}' = \{(low, hvar1 + hvar2)\}$.

There are two fundamental differences that result from the different use of the escape hatch sets. The first difference is that WHAT$_1$-Security is compositional, but WHAT$_2$-Security is not, as the we can show by using the previous example as starting point and define a second program lowvar := hvar1 + hvar3. Both programs individually would fulfill WHAT$_2$-Security, but neither a sequential, nor a parallel composition of the programs would be WHAT$_2$-Security, because we would have to choose $\mathcal{H}' = \mathcal{H}$ and therefore the program would violate WHAT$_2$-Security in the same way it violates WHAT$_1$-Security. Hence, WHAT$_2$-Security is not compositional.
The second difference is that WHAT$_2$-Security obeys the monotonicity of release principle from [SS05], which requires that adding a declassification can not render a secure program insecure, but WHAT$_1$-Security breaks this principle. We can show this with the previous example, if we chose $\mathcal{H}$ to be $\{(low, hvar1 + hvar2)\}$ in the first place, the program would fulfill WHAT$_1$-Security, but if we then added $(low, hvar1 + hvar3)$ as

an escape hatch and therefore $hvar1 + hvar3$ as a legal declassification, the program is insecure as we have seen in the example, thus WHAT_1-Security breaks the monotonicity of release principle.

Since the properties WHAT_1-Security and WHAT_2-Security are very similar and under policies that fulfill the constraint that only escape hatches that are used in the program are in the set $\mathcal{H}$ are even equal, we will talk about WHAT-Security when we mean WHAT_1-Security and WHAT_2-Security during the rest of the work.

3.2.3. Scheduler-independence of WHAT-Security

After the detailed reflection of the intuition and definition of WHAT-Security we want to show that the properties are scheduler-independent with respect to the class of *low*-secure σ-schedulers. We will use the scheduler-independence proof for WHERE-Security from Section 3.1.3 and STRONG-Security as a guideline.

It is possible to use the same scheduler-independence proof for WHAT_1-Security and WHAT_2-Security, because the characterization formula of the properties is identical and in consequence, the constraints on the relation are identical. The difference in the escape hatches can be ignored for the proof since the sets $\mathcal{H}$ for WHAT_1-Security and $\mathcal{H}' \subseteq \mathcal{H}$ are determined only by the program code and the security policies and not by the properties.

Definition 20 ***σ-specific WHAT-Security:***
A σ-specific $(D, \mathcal{H})$-bisimulation *is a symmetric relation R on command vectors of equal size that satisfies the entire formula in Figure 11. The relation $\approx^{\mathcal{H}}_{D,\sigma}$ is the union of all strong σ-specific $(D, \mathcal{H})$-bisimulations. A program $\overrightarrow{C}$ has secure information flow for a given scheduler σ while complying with the restrictions* **what** *may get declassified if $\overrightarrow{C} \approx^{\mathcal{H}}_{D,\sigma} \overrightarrow{C}$ holds for all $D \in \mathcal{D}$*
(brief: $\overrightarrow{C}$ is σ-WHERE-secure or $\overrightarrow{C} \in \sigma - WHAT$).

$$
\begin{aligned}
&\forall H_1, H_2, H_1' : \forall s_1, s_2, s_1' : \forall \overrightarrow{C}_1' : \\
&(\overrightarrow{C}_1 R \overrightarrow{C}_2 \wedge \langle\!| H_1, \overrightarrow{C}_1, s_1 |\!\rangle \rightarrow \langle\!| H_1', \overrightarrow{C}_1', s_1' |\!\rangle \wedge H_1 =_\sigma H_2 \wedge s_1 =^{\mathcal{H}}_D s_2) \\
&\Longrightarrow \exists H_2' : \exists \overrightarrow{C}_2' : \exists s_2' : \langle\!| \overrightarrow{C}_2, s_2 |\!\rangle \rightarrow \langle\!| \overrightarrow{C}_2', s_2' |\!\rangle \\
&\qquad (i) \wedge H_1' =_\sigma H_2' \wedge \overrightarrow{C}_1' R \overrightarrow{C}_2' \wedge s_1' =^{\mathcal{H}}_D s_2' \\
&\qquad (ii) \wedge \left[\begin{array}{l} \sum \{\!| p | \langle\!| H_1, \overrightarrow{C}_1, s_1 |\!\rangle \rightarrow_p \langle\!| H, \overrightarrow{C}, s |\!\rangle, H \in [H_1']_{=_\sigma}, \overrightarrow{C} \in [\overrightarrow{C}_1']_R, s \in [s_1']_{=^{\mathcal{H}}_D} |\!\} \\ = \sum \{\!| p | \langle\!| H_2, \overrightarrow{C}_2, s_2 |\!\rangle \rightarrow_p \langle\!| H, \overrightarrow{C}, s |\!\rangle, H \in [H_2']_{=_\sigma}, \overrightarrow{C} \in [\overrightarrow{C}_2']_R, s \in [s_2']_{=^{\mathcal{H}}_D} |\!\} \end{array} \right]
\end{aligned}
$$

Figure 11: Characterization of σ-specific WHAT-Security

In order to be able to argue about the scheduling we use the system configurations and, in consequence, the scheduler steps in the characterization formula of σ-specific WHAT-Security. We preserve the requirements of WHAT-Security, but transfer them to the system configurations. Since system configurations contain the scheduling histories, we additionally use σ-equivalence of the histories to describe the indistinguishability of two configurations.

Since we assume that the attacker can estimate the scheduling-properties, for example by running the program several times with the same initial memory, we want the scheduling not to reveal any further information. In order to achieve this, we require that a transition from $\langle\!|H_2, \overrightarrow{C}_2, s_2|\!\rangle$ into the class of configurations that is indistinguishable from the configuration $\langle\!|H_2', \overrightarrow{C}_2', s_2'|\!\rangle$ is equally likely as a transition from $\langle\!|H_1, \overrightarrow{C}_1, s_1|\!\rangle$ into the class of configurations that is indistinguishable from the configuration $\langle\!|H_1', \overrightarrow{C}_1', s_1'|\!\rangle$. Since we require $H_1' =_\sigma H_2'$, $\overrightarrow{C}_1' R \overrightarrow{C}_2'$ and $s_1' =_D^{\mathcal{H}} s_2'$, the the configurations after executing the step are in the same equivalence class, just like the states before executing the step where. In consequence, the indistinguishability of the configurations is preserved and furthermore the probability of the execution of the step is equally likely in both configurations. Since the probabilities are equal in both configurations the attacker can not learn any additional information by estimating or learning the probabilities.

This definition gives us a scheduler-specific property. We call WHAT-Security scheduler-independent, if every program that is WHAT-secure is σ-WHAT-secure for all *low*-secure σ-schedulers, too.

Theorem 3 ***Scheduler-independence of WHAT-Security:***
If $\overrightarrow{C}$ is WHAT-secure, then $\overrightarrow{C}$ is σ-WHAT-secure for all σ.

$$\overrightarrow{C} \approx_D^{\mathcal{H}} \overrightarrow{C} \Longrightarrow \forall \sigma : \overrightarrow{C} \approx_{D,\sigma}^{\mathcal{H}} \overrightarrow{C}$$

Proof Scheduler-independence of WHAT-Security: In order to proof that WHAT-Security is scheduler-independent, we show that a relation R that fulfills the characterization formula of WHAT-Security also fulfills the characterization formula of σ-specific WHAT-Security for all *low*-secure schedulers σ. So, if we assume that R exists, such that $\overrightarrow{C} R \overrightarrow{C}$ and R fulfills the characterization formula of WHAT-Security given in Figure 10, then the following must hold for all *low*-secure schedulers σ, too:

$$\begin{aligned}
&\forall H_1, H_2, H_1' : \forall s_1, s_2, s_1' : \forall \overrightarrow{C}_1' : \\
&(\overrightarrow{C}_1 R \overrightarrow{C}_2 \wedge \langle\!|H_1, \overrightarrow{C}_1, s_1|\!\rangle \to \langle\!|H_1', \overrightarrow{C}_1', s_1'|\!\rangle \wedge H_1 =_\sigma H_2 \wedge s_1 =_D^{\mathcal{H}} s_2) \\
&\Longrightarrow \exists H_2' : \exists \overrightarrow{C}_2' : \exists s_2' : \langle\!|\overrightarrow{C}_2, s_2|\!\rangle \to \langle\!|\overrightarrow{C}_2', s_2'|\!\rangle \\
&\quad (i) \wedge H_1' =_\sigma H_2' \wedge \overrightarrow{C}_1' R \overrightarrow{C}_2' \wedge s_1' =_D^{\mathcal{H}} s_2' \\
&\quad (ii) \wedge \left[\begin{array}{l} \sum \{\!|p | \langle\!|H_1, \overrightarrow{C}_1, s_1|\!\rangle \to_p \langle\!|H, \overrightarrow{C}, s|\!\rangle, H \in [H_1']_{=_\sigma}, \overrightarrow{C} \in [\overrightarrow{C}_1']_R, s \in [s_1']_{=_D^{\mathcal{H}}} |\!\} \\ = \sum \{\!|p | \langle\!|H_2, \overrightarrow{C}_2, s_2|\!\rangle \to_p \langle\!|H, \overrightarrow{C}, s|\!\rangle, H \in [H_2']_{=_\sigma}, \overrightarrow{C} \in [\overrightarrow{C}_2']_R, s \in [s_2']_{=_D^{\mathcal{H}}} |\!\} \end{array} \right]
\end{aligned}$$

First, we want to show that (ii) holds. If the left side of the implication is fulfilled we know that $s_1 =_D^{\mathcal{H}} s_2$ and can deduce from the definition of (D,H)-equality that $s_1 =_D s_2$. In consequence, $s_1 =_{low} s_2$, because $low \leq D$ holds, since low is the minimum of $\mathcal{D}$ with respect to $\leq$ per definition. Using Theorem 1 about the probability equivalence for equivalent histories and states we can deduce that $\sigma(H_1, s_1) =_\sigma \sigma(H_2, s_2)$. In combination with the quantification over all the threads in the thread pools and the requirements on the execution steps in the characterization formula of WHAT-Security we can deduce that a one-to-one correspondence exists between the elements of the multi sets and in consequence the sum of the elements of the multi sets is equal and (ii) is fulfilled.

Second, we want to show that (i) holds. We know from the thread pool semantics in Figure 4 that the execution step

$$\langle\!|C_{1,i}, s_1|\!\rangle \twoheadrightarrow \langle\!|\overrightarrow{W}_1, s_1'|\!\rangle$$

triggers a scheduler step

$$\langle\!|H_1, \overrightarrow{C}_1, s_1|\!\rangle \twoheadrightarrow \langle\!|H_1(i, n + |\overrightarrow{W}_1| - 1, \langle C_{1,0} \dots C_{1,i-1} \overrightarrow{W}_1 C_{1,i+1} \dots C_{1,n-1}\rangle, s_1'|\!\rangle$$

and deduce that $H_1' = H_1(i, n+|\overrightarrow{W}_1|-1)$ and $\overrightarrow{C}_1' = \langle C_{1,0} \dots C_{1,i-1} \overrightarrow{W}_1 C_{1,i+1} \dots C_{1,n-1}\rangle$. From the characterization formula of WHAT-Security we know that

$$\langle\!|C_{2,i}, s_2|\!\rangle \twoheadrightarrow \langle\!|\overrightarrow{W}_2, s_2'|\!\rangle$$

exists and can use the thread pool semantics again to deduce that this execution step triggers a scheduler step

$$\langle\!|H_2, \overrightarrow{C}_2, s_2|\!\rangle \to \langle\!|H_2(i, n + |\overrightarrow{W}_2| - 1), \langle C_{2,0} \dots C_{2,i-1} \overrightarrow{W}_2 C_{2,i+1} \dots C_{2,n-1}\rangle, s_2'|\!\rangle$$

and in consequence $H_2' = H_2(i, n + |\overrightarrow{W}_2| - 1)$ and
$\overrightarrow{C}_2' = \langle C_{2,0} \dots C_{2,i-1} \overrightarrow{W}_2 C_{2,i+1} \dots C_{2,n-1}\rangle$.
From the definition of WHAT-Security we know that $|\overrightarrow{C}_1| = |\overrightarrow{C}_2| = n$ and $|\overrightarrow{W}_1| = |\overrightarrow{W}_2|$. In consequence, $n+|\overrightarrow{W}_1|-1 = n+|\overrightarrow{W}_2|-1$. From the left side of the implication we know that $H_1 =_\sigma H_2$ and with the Definition 11 of σ-equivalence, we can conclude that

$$H_1' = H_1(i, n + |\overrightarrow{W}_1| - 1) =_\sigma H_2(i, n + |\overrightarrow{W}_2| - 1) = H_2'$$

We can use the quantification over all threads in the thread pool in the characterization formula of WHAT-Security to deduce that all execution steps that can be made by a single thread fulfill the characterization formula of WHAT-Security point-wise and therefore $C_{1,i} R C_{2,i}$ for all i. Furthermore, we know from the the characterization formula that $\overrightarrow{W}_1 R \overrightarrow{W}_2$. Using the quantification over the threads again, we can deduce that $\langle C_{1,0} \dots C_{1,i-1} \overrightarrow{W}_1 C_{1,i+1} \dots C_{1,n-1}\rangle R \langle C_{2,0} \dots C_{2,i-1} \overrightarrow{W}_2 C_{2,i+1} \dots C_{2,n-1}\rangle$ which in turn means $\overrightarrow{C}_1' R \overrightarrow{C}_2'$.

The last requirement $s_1' =_D^{\mathcal{H}} s_2'$ is identical in WHAT-Security and in σ-specific WHAT-Security and therefore is fulfilled in σ-specific WHAT-Security, whenever it is fulfilled in WHAT-Security.

Since the characterization formula of σ-specific WHAT-Security is fulfilled for any arbitrary scheduler, WHAT-Security is scheduler-independent.

■

The scheduler-independence of WHAT-Security is important, because it means that WHAT-Security guarantees our intuition of security with controlled declassification with respect to the aspect **what** for all *low*-secure σ-scheduler. This strengthens our confidence in the adequacy of the property for multi-threaded settings and therefore WHAT-Security is a good foundation for our integrated property.

3.3. Combined WHERE-Security and WHAT-Security

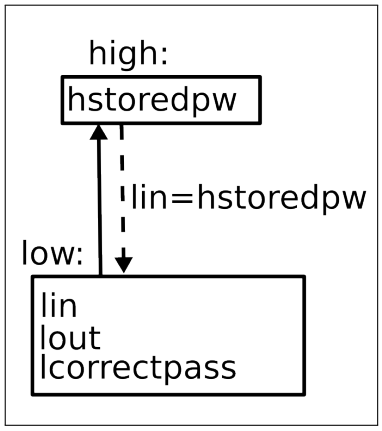
The conjunction of WHERE-Security and WHAT-Security results in a property that **where** in the program something is declassified and **what** is declassified somewhere in the program. It is now possible to analyze a program with WHERE-Security and WHAT-Security. Since both properties are scheduler-independent, we can use them to classify multi-threaded programs as secure that use declassification in a controlled way and therefore can not be classified as secure with STRONG-Security, since it does not allow declassifications.

Definition 21 ***Multilevel Security Policy with* where *and* what *Exceptions:*** *A* Multilevel Security Policy with **where** and **what** Exceptions *(brief: mls-*$(\rightsquigarrow, \mathcal{H})$ *policy) is a tuple* $(\mathcal{D}, \leq, \rightsquigarrow, dom, \mathcal{H})$ *where* $(\mathcal{D}, \leq, \rightsquigarrow, dom)$ *is a mls-*$\rightsquigarrow$ *policy and* $(\mathcal{D}, \leq, dom, \mathcal{H})$ *is a mls-*$\mathcal{H}$ *policy.*

The mls-$(\rightsquigarrow, \mathcal{H})$ policies combine the intuitions of the multilevel security policies with **where** exceptions and **what** exceptions.

Example:

Let *pol* be the following mls-$(\rightsquigarrow, \mathcal{H})$ policy:
$\mathcal{D} = \{low, high\}$ $low \leq low$
$low \leq high$
$high \leq high$
$high \rightsquigarrow low$
$dom(lin) = dom(lout) = low$
$dom(lcorrectpass) = low$
$dom(hstoredpw) = high$
$\mathcal{H} = \{(low, (lin = hstoredpw))\}$



Let $\overrightarrow{C}$ be the following program:

```
[ lcorrectpass := ( lin = hstoredpw ) ]1
if (lorrectpass)
then
  lout := 1
else
  lout := 0
fi
```

The program is WHERE-secure, since every declassification occurs in a declassification assignment and every declassification obeys the rules for information flow given by $\leq$ and $\rightsquigarrow$. The program is WHAT-secure, too, since $(low, (lin = hstoredpw))$ is in the set of escape hatches $\mathcal{H}$.

This little example program is a simple password check, where the input and output is modeled with *lin* and *lout*. Clearly these variables must be *low* since an attacker can directly see the value of the variables. On the other hand the stored password, represented by *hstoredpw* should not leak to the attacker and therefore is classified as *high*.

The program does not fulfill STRONG-Security since *lcorrectpass* is in the domain *low*, but depends on *hstoredpw*. On the other hand the value of *hstoredpw* is not leaked to the domain *low*, but only the value of the expression $(linout = hstoredpw)$ and therefore we intuitively would consider this program secure.

This example shows that the combination of WHERE-Security and WHAT-Security allows us to use declassifications in a controlled way and captures our intuition of security better than STRONG-Security in this example program. It seems reasonable to use WHERE-Security and WHAT-Security as a foundation for developing an integrated property, in order to be able to determine **where** in the program **what** gets declassified.

4. Integration of the aspects where and what

After the detailed look on WHERE-Security and WHAT-Security we are confident that these properties capture our intuition very closely. They are scheduler-independent. These facts make them a good foundation for a novel security property that integrates the aspects **where** and **what** into a single property that is adequate for multi-threaded settings.

4.1. Motivation for the Integration

Before we introduce the novel property, we want to motivate the decision to integrate both both aspects into a single property with the help of the following example:

Example:
Let *pol* be the following mls-$(\rightsquigarrow, \mathcal{H})$ policy:
$low \leq low$, $low \leq high$, $high \leq high$
$high \rightsquigarrow low$
$dom(h1) = dom(h2) = high$
$dom(l1) = low$
$\mathcal{H} = \{(low, h1 + h2), (low, h1)\}$.

Let P be the following program, where S is a sequence of commands that contains some output of variable $l1$:

$[\,\mathrm{l1{=}h1{+}h2}\,]_1$
S
$[\,\mathrm{l1{=}h1}\,]_2$

If we want to know, if an attacker that only sees the output that occurs somewhere in S can learn the secret value $h1$, neither the use of WHERE-Security, nor the use of WHAT-Security can help to answer this question. Even a combination of both properties does not allow us to restrict a specific declassification, in this case $(low, h1)$, to a specific location or set of locations, in this case $[\mathrm{l1{=}h1}]_2$, because WHERE-Security has no information about **what** gets declassified and WHAT-Security has no information **where** the declassification occurs.

Furthermore, it would not be possible to change the security policy such that it is possible to use the properties to determine if output of $h1$ via $l1$ occurs in S, because $h1$ is assigned to $l1$, but only after S.

As this example shows, in some situations it is useful to know exactly **where** in the program **what** information is declassified, instead of **where** in the program something is declassified and additionally know **what** is declassified somewhere in the program.

Another useful scenario for a more exact localization of declassifications would be in the process of software development, where we assume that the information leak is not intentionally created, but is introduced due to a programming error. In a large project, there may exist many declassifications in many different threads and over time

some more might get introduced during development. In the case, where the program must be considered insecure after the extension of the program code, one only knows that either something is declassified, which should not be declassified or somewhere a declassification occurs, where no declassification should occur.

An integration of both aspects into one single property on the other side would allow us to specify **what** information gets declassified **where**. Thus it would be possible to point to the location and to the expression that causes the new problem. In consequence, it is easier to find the problematic code and handle the problem by either changing the code or, if the developer decides that the information flow is secure, just change the problematic part of the policy.

4.2. Integration of where and what into the Policies and the Language

In order to get more detailed information about **what** is declassified **where**, we introduce identifiers for sets of escape hatches. An identifier is a natural number $n \in \mathbb{N}$. We use $Loc \subseteq \mathbb{N}$ to describe the subset of the natural numbers that contains all used identifiers.

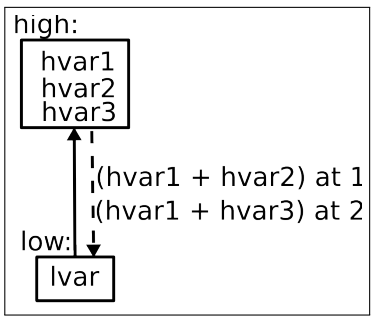
Definition 22 ***Multilevel Security Policy with localized* what *Exceptions:***
A Multilevel Security Policy with localized **what** Exceptions *(brief: mls-$(\rightsquigarrow, \mathcal{L})$-policy) is a tuple $(\mathcal{D}, \leq, \rightsquigarrow, dom, \mathcal{L})$ where $(\mathcal{D}, \leq, \rightsquigarrow, dom)$ is a mls-$\rightsquigarrow$ policy and $\mathcal{L}$ is a function that maps pairs of domains and expressions to identifiers $\mathcal{L} : Loc \to \mathcal{P}(\mathcal{D} \times Expr)$.*

The main difference between mls-$(\rightsquigarrow, \mathcal{H})$ policies and mls-$(\rightsquigarrow, \mathcal{L})$ policies is that we use a function instead of the set of escape hatches. We call this function localized hatches, because the intuition behind this function is that we can use the identifier to determine a set of escape hatches that represents the knowledge an observer is allowed to learn at a specific declassification in the program.

While the mls-$(\rightsquigarrow, \mathcal{L})$ policies do not explicitly contain a set $\mathcal{H}$ of all escape hatches, such a set can be constructed with the conjunction over all sets of localized hatches. We use $\mathcal{H}$ in the rest of the work to denote this set.

Example:

Let *pol* be the following mls-$(\rightsquigarrow, \mathcal{L})$ policy:
$\mathcal{D} = \{low, high\}$
$low \leq low$, $low \leq high$, $high \leq high$
$high \rightsquigarrow low$
$dom(lvar) = low$
$dom(hvar1) = high$
$dom(hvar2) = high$
$dom(hvar3) = high$
$\mathcal{L}(1) = \{(low, hvar1 + hvar2)\}$
$\mathcal{L}(2) = \{(low, hvar1 + hvar3)\}$.



The intuition of this policy is that information may flow regularly from *low* to *high*, but the information flow from *high* to *low* is exceptional and is restricted to $hvar1 + hvar2$ at location 1 and $hvar1 + hvar3$ at location 2.

Let $\overrightarrow{C}$ be the following program:

```
if ( lvar ) then
  [ lvar := hvar1 + hvar2 ]1
else
  [ lvar := hvar1 + hvar3 ]2
fi
```

Intuitively, this program is secure. While the assignments in line 2 and 4 violate the regular information flow rules given by $\leq$, they occur in a declassification assignment and obey the exceptional information flow rules given by $\rightsquigarrow$ and $\mathcal{L}$, because $high \rightsquigarrow low$ and $(low, hvar1 + hvar2) \in \mathcal{L}(1)$, respectively $(low, hvar1 + hvar3) \in \mathcal{L}(2)$.

This policy does not allow [lvar := hvar1 + hvar2]$_2$, because $(low, hvar1{+}hvar2) \notin \mathcal{L}(2)$.

In order to be able to use the identifiers to determine special locations in the program we extend the syntax of the declassification assignments and annotate the closing bracket of a declassification with a location identifier $loc \in Loc$. The complete MWL syntax with localized declassifications can be found in Figure 12.

$$
\begin{array}{rcl}
Com & :== & Com; Com \mid \textbf{skip} \\
 & \mid & \textbf{if } (\ Expr\ ) \textbf{ then } Com \textbf{ else } Com \textbf{ fi} \\
 & \mid & \textbf{while } (\ Expr\ ) \textbf{ do } Com \textbf{ od} \\
 & \mid & Var := Expr \\
 & \mid & [\ Var := Expr\ ]_{Loc} \\
 & \mid & \textbf{fork } (\ Com\ \overrightarrow{Com}\ )
\end{array}
$$

Figure 12: Grammar for the Multi-threaded While Language with localized Exceptions

We require the identifiers $loc \in Loc$ to point to a unique declassification assignment in the program. It is important to note that the transformation of the program with the operational semantics may break the uniqueness of the identifiers, because the unwinding in the transformation triggered by *while* can duplicate a declassification assignment. Since we want a static analysis this does not violate our intuition, because the locations should be unique in the program code, but not necessarily in the intermediary command vectors.

In order to be able to use the information of the identifiers in the security property, we change the semantics for the declassification assignments to keep track of the identifiers. Figure 13 shows the updated MWL semantics.

It is possible to transform a program that uses the declassification assignments without identifiers into a program with identifiers by adding identifiers to every declassification

$$\frac{\langle\!|exp, s|\!\rangle \downarrow n \quad sources(Exp) = \mathcal{D}_1 \quad dom(var) = D_2}{\langle\!|[var{:=}exp]_{loc}, s|\!\rangle \rightarrow_{loc}^{\mathcal{D}_1 \to D_2} \langle\!|\langle\rangle, s \otimes \{var = n\}|\!\rangle} \qquad \frac{\langle\!|C_1, s|\!\rangle \rightarrow_{loc}^{\mathcal{D}_1 \to D_2} \langle\!|\langle\rangle, s'|\!\rangle}{\langle\!|C_1; C_2, s|\!\rangle \rightarrow_{loc}^{\mathcal{D}_1 \to D_2} \langle\!|C_2, s'|\!\rangle}$$

Figure 13: Operational semantics for threads in the Multi-threaded While Language with localized Exceptions

assignment, such that no identifier occurs twice. Additionally, we can construct a mls-$(\rightsquigarrow, \mathcal{L})$ policy from a mls-$(\rightsquigarrow, \mathcal{H})$ policy by defining $\forall loc \in Loc : \mathcal{L}(loc) = \mathcal{H}$.

It is possible to construct a program without identifiers at declassification assignments by just removing the identifiers at the cost of omitting localization information. This transformation is necessary in order to use WHERE-Security and WHAT-Security. Furthermore, if we omit the localization information in the policy, we can construct a mls-$(\rightsquigarrow, \mathcal{H})$ policy from a mls-$(\rightsquigarrow, \mathcal{L})$ policy by using the conjunction of all localized hatch sets that we already called $\mathcal{H}$ earlier in this section.

4.3. Definition and Intuition of WHERE&WHAT$_{local}$-Security

The confidence in WHERE-Security and WHAT-Security makes those properties a good foundation for a property that aims at a tighter integration of both aspects **where** and **what**. Since the relation $=_D^H$, that occurs on the left side of the implication in WHAT-Security, is stricter than the relation $=_D$, that occurs on the right side of the implication in WHERE-Security, it is reasonable to follow the basic idea of WHERE-Security and integrate the control for **what** into this property.

Definition 23 ***WHERE&WHAT$_{local}$-Security:***
A strong $(D, \rightsquigarrow, \mathcal{L})$-local-bisimulation *is a symmetric relation R on command vectors of equal size that satisfies the entire formula in Figure 14. The relation $(loc \approx_D^{\rightsquigarrow, \mathcal{L}})$ is the union of all strong $(D, \rightsquigarrow, \mathcal{L})$-local-bisimulations. A program $\overrightarrow{C}$ has secure information flow while complying with the restrictions where declassification can occur and what information may get declassified if $\overrightarrow{C}(loc \approx_D^{\rightsquigarrow, \mathcal{L}})\overrightarrow{C}$ holds for all $D \in \mathcal{D}$*
(brief: $\overrightarrow{C}$ is WHERE&WHAT$_{local}$-secure or $\overrightarrow{C} \in$ WHERE&WHAT$_{local}$).

The basic intuition of WHERE-Security is preserved by using the characterization formula of WHERE-Security. The only change is the additional requirement $s_1 \neq_D^{\mathcal{L}(loc)} s_2$ in the case where a declassification occurs. The intuition behind this novel requirement is that the attacker is allowed to learn the information that we get by evaluating the expressions in the escape hatches. On basis of this information he could have distinguished the states before the execution step. This requirement basically allows the same declassifications as adding its dual, $s_1 =_D^{\mathcal{L}(loc)} s_2$, to the left side of the implication, which captures the basic idea from WHAT-Security, but adding $s_1 =_D^{\mathcal{L}(loc)} s_2$ to the left side of the implication would lead to a formula, where the restrictions that come from WHERE-Security would not have any impact.

$$
\begin{aligned}
&\forall s_1, s_2, s_1' : \forall \overrightarrow{W}_1 : \forall i \in \{1 \dots n\} : \\
&(\overrightarrow{C}_1 R \overrightarrow{C}_2 \wedge \langle\!| C_{1,i}, s_1 |\!\rangle \rightarrowtriangle \langle\!| \overrightarrow{W}_1, s_1' |\!\rangle \wedge s_1 =_D s_2) \\
&\Longrightarrow \exists \overrightarrow{W}_2 : \exists s_2' : \overrightarrow{W}_1 R \overrightarrow{W}_2 \wedge \langle\!| C_{2,i}, s_2 |\!\rangle \rightarrowtriangle \langle\!| \overrightarrow{W}_2, s_2' |\!\rangle \\
&\qquad \wedge \left[\begin{array}{l} s_1' =_D s_2' \\ \quad \vee \left[\begin{array}{l} \exists \mathcal{D}_1, \{D_2\} \subseteq \mathcal{D} : \exists loc \in Loc \\ \quad \left[\begin{array}{l} \langle\!| C_{1,i}, s_1 |\!\rangle \rightarrowtriangle_{loc}^{\mathcal{D}_1 \to D_2} \langle\!| \overrightarrow{W}_1, s_1' |\!\rangle \\ \quad \wedge \forall D_1 \in \mathcal{D}_1 : (D_1 \leq D_2 \vee D_1 \rightsquigarrow D_2) \\ \quad \wedge D_2 \leq D \wedge \exists D_1 \in \mathcal{D} : s_1 \neq_{D_1} s_2 \\ \quad \wedge s_1 \neq_D^{\mathcal{L}(loc)} s_2 \end{array} \right] \end{array} \right] \end{array} \right]
\end{aligned}
$$

Figure 14: Characterization of WHERE&WHAT$_{local}$-Security with $\overrightarrow{C}_i = \langle C_{i,1} \dots C_{i,n} \rangle$ for $i \in \{1, 2\}$

Example:
Let *pol* be the following mls-$(\rightsquigarrow, \mathcal{L})$ policy:
$\mathcal{D} = \{low, high\}$
$low \leq low$, $low \leq high$, $high \leq high$
$\rightsquigarrow = \emptyset$
$dom(lvar) = low$
$dom(hvar1) = dom(hvar2) = high$
$\mathcal{L}(1) = \{(low, hvar1 + hvar2)\}$.

Let $\overrightarrow{C}$ be the following program:

```
[ lvar := hvar1 + hvar2 ]1
```

The security policy *pol* disallows declassifications from *high* to *low*, because $\rightsquigarrow = \emptyset$. The program on the other hand uses a declassification of $hvar1 + hvar2$. The property WHERE&WHAT$_{local}$-Security is not fulfilled by $\overrightarrow{C}$. We construct a counter example by choosing $s_1(lvar) = s_2(lvar) = 0$, $s_1(hvar1) = s_1(hvar2) = s_2(hvar1) = 0$ and $s_2(hvar2) = 1$. When looking at the assignment $s_1 =_{low} s_2$ holds, but $s_1' =_{low} s_2'$ does not and in consequence a declassification occurred. The restriction $\forall D_1 \in \mathcal{D}_1 : (D_1 \leq D_2 \wedge D_1 \rightsquigarrow D_2)$ is not fulfilled, because neither $\leq$ nor $\rightsquigarrow$ allow an information flow from *high* to *low*.

If the property required $s_1 =_D^{\mathcal{L}(loc)} s_2$ on the left side of the implication instead of $s_1 =_D s_2$ the program would be considered secure, because whenever $hvar1 + hvar2$ would evaluate to different values under s_1 and s_2 the precondition of the formula would not be fulfilled and therefore the complete formula would be fulfilled despite the fact that $\rightsquigarrow$ does not allow any declassification.

Since the semantics of the declassification assignments keep track of the identifier, we can be sure that the correct set of localized hatches is used. Furthermore, the right side of the implication requires that $s_1 =_D s_2$ and in consequence, $s_1 \neq_D^{\mathcal{L}(loc)} s_2$ must be caused by an expression from an escape hatch in $\mathcal{L}(loc)$ and an observer is allowed to distinguish the resulting states due to the expression evaluation which we wanted to declassify intentionally in the first place.

Unlike WHAT-Security, this property has no control of information flow into the escape hatches. Since there is no control of information flow into the escape hatches, the reference point of what may get declassified is the memory state directly before executing the declassification step. This weakens the security guarantees this property can give.

Example:
Let *pol* be the following mls-$(\rightsquigarrow, \mathcal{L})$ policy:
$\mathcal{D} = \{low, high\}$
$low \leq low$, $low \leq high$, $high \leq high$
$high \rightsquigarrow low$
$dom(lvar) = low$
$dom(hvar1) = dom(hvar2) = high$
$\mathcal{L}(1) = \{(low, hvar1)\}$.

Let $\overrightarrow{C}$ be the following program:

```
hvar1 := hvar2
[ lvar := hvar1 ]1
```

Intuitively, this program should be considered insecure, because we assign the value of $hvar2$ to $hvar1$ and then declassify $hvar1$ which contains the value of $hvar2$. Against the intuition the program is secure with respect to WHERE&WHAT$_{local}$-Security, since the declassification occurs in a declassification assignment, it obeys the security policy and when reaching the declassification assignment $s_1 \neq_D^{\mathcal{L}} (1) s_2$ for every run that started with two initial memory states where $s_1(hvar2) \neq s_2(hvar2)$.

This simple example shows how important the reference points for the memory states are. On the other hand it is to restrictive to require the reference points always to be the initial memory states, because that would make the handling of run time user input very cumbersome. The authors of [LM09a] address this problem by introducing explicit reference points. We assume that it is possible to include the concept of explicit reference points into this property instead of just restricting the information flow into the escape hatches like in WHAT-Security, but this is out of the scope of this work and therefore left for future work. In this work we stick to the control of information flow into the escape hatches.

4.4. Controlling the Information Flow into the Escape Hatches

In the last section we have seen that the reference points for the memory states play an important role. While explicit reference points as in [LM09a] look very promising, the integration of the explicit reference points is out of the scope of this work and therefore we stick to the concept of controlling the information flow into the escape hatches and leave the reference points implicit. We assume the integration of explicit reference points can be a achieved by replacing the control of information flow into the escape hatches in the characterization formula and therefore this section is not only a presentation of a property with controlled information flow into the hatches, but also a guideline for refinement in later work.

Definition 24 ***WHERE&WHAT$_{initial}$-Security:***
A strong $(D, \rightsquigarrow, \mathcal{L})$-initial-bisimulation *is a symmetric relation R on command vectors of equal size that satisfies the entire formula in 15. The relation $\approx_D^{\rightsquigarrow,\mathcal{L}}$ is the union of all strong $(D, \rightsquigarrow, \mathcal{L})$-initial-bisimulations. A program $\overrightarrow{C}$ has secure information flow while complying with the restrictions where declassification can occur and what information may get declassified if $\overrightarrow{C} \approx_D^{\rightsquigarrow,\mathcal{L}} \overrightarrow{C}$ holds for all $D \in \mathcal{D}$ (brief: $\overrightarrow{C}$ is WHERE&WHAT$_{initial}$-secure or $\overrightarrow{C} \in WHERE\&WHAT_{initial}$).*

$$
\begin{aligned}
&\forall s_1, s_2, s_1' : \forall \overrightarrow{W}_1 : \forall i \in \{1 \dots n\} : \\
&(\langle\!| C_{1,i}, s_1 |\!\rangle \twoheadrightarrow \langle\!| \overrightarrow{W}_1, s_1' |\!\rangle \wedge \overrightarrow{C}_1 R \overrightarrow{C}_2 \wedge s_1 =_D s_2) \\
&\Longrightarrow \exists \overrightarrow{W}_2 : \exists s_2' : \overrightarrow{W}_1 R \overrightarrow{W}_2 \wedge \langle\!| C_{2,i}, s_2 |\!\rangle \twoheadrightarrow \langle\!| \overrightarrow{W}_2, s_2' |\!\rangle \\
&\qquad \wedge \left[s_1' =_D s_2' \vee \left[\begin{array}{l} \exists \mathcal{D}_1, \{D_2\} \subseteq \mathcal{D} : \exists loc \in Loc \\ \left[\begin{array}{l} \langle\!| C_{1,i}, s_1 |\!\rangle \twoheadrightarrow_{loc}^{\mathcal{D}_1 \to D_2} \langle\!| \overrightarrow{W}_1, s_1' |\!\rangle \\ \wedge \forall D_1 \in \mathcal{D}_1 : (D_1 \leq D_2 \vee D_1 \rightsquigarrow D_2) \\ \wedge D_2 \leq D \wedge \exists D_1 \in \mathcal{D} : s_1 \neq_{D_1} s_2 \\ \wedge s_1 \neq_D^{\mathcal{L}(loc)} s_2 \end{array} \right] \end{array} \right] \right] \\
&\wedge \forall loc \in Loc : (s_1 =_D^{\mathcal{L}(loc)} s_2 \Longrightarrow s_1' =_D^{\mathcal{L}(loc)} s_2')
\end{aligned}
$$

Figure 15: Characterization of WHERE&WHAT$_{initial}$-Security with $\overrightarrow{C}_i = \langle C_{i,1} \dots C_{i,n} \rangle$ for $i \in \{1, 2\}$

We use the characterization formula of WHERE&WHAT$_{local}$-Security to control the regular information flow and the declassifications. Additionally, we want to forbid information flow into the escape hatches that makes indistinguishable states distinguishable by the evaluation of the expressions in the hatches like it is achieved in WHAT-Security. As a result, we require that states that where indistinguishable lead to indistinguishable states with an execution step. We have already explained that we can not just

add $s_1 =_{\mathcal{L}(loc)} s_2$ on the left side of the implication in Section 4.3 and therefore we can not use the same structure for the control of information flow into the hatches as in WHAT-Security, but we can take the implication that describes the requirement on the indistinguishability of the states and shift it to the right side of the implication. The formalization captures the intuition that an observer may not learn any information that helps him to distinguish the states after the execution step, if he could not distinguish the states before the step with the same knowledge. Neither the memory states, nor the escape hatches of every single location in the program should reveal more information than they did before executing the step.

Instead of using all escape hatches as in WHAT-Security where the formula requires $s_1 =_D^{\mathcal{H}} s_2 \Longrightarrow s_1' =_D^{\mathcal{H}} s_2'$, the characterization formula of WHERE&WHAT$_{initial}$-Security requires $\forall loc \in Loc : s_1 =_D^{\mathcal{L}(loc)} s_2 \Longrightarrow s_1' =_D^{\mathcal{L}(loc)} s_2'$ for two reasons. The first reason is that we drop $\mathcal{H}$ as set of all escape hatches from the policies. More important, the second reason is that that the use of all escape hatches limits the ability to localize where information gets declassified, for example in the case where two single variables get declassified and there exists an assignment at which the information from one variable flows to the other.

Example:
Let *pol* be the following mls-$(\rightsquigarrow, \mathcal{L})$ policy:
$\mathcal{D} = \{low, high\}$
$low \leq high$
$high \rightsquigarrow low$
$dom(lvar1) = dom(lvar2) = low$
$dom(hvar1) = dom(hvar2) = high$
$\mathcal{L}(1) = \{(low, hvar1)\}$
$\mathcal{L}(2) = \{(low, hvar2)\}$.

Let $\overrightarrow{C}$ be a thread pool with the following two threads:

```
if (bvar)                  |
then                       |
   [ lvar1 := hvar1 ]1     | hvar2 := hvar1
else                       |
   [ lvar2 := hvar2 ]2     |
fi                         |
```

It is possible that the information from $hvar1$ gets declassified at the declassification assignment with the identifier 2, since the information from $hvar1$ is copied to $hvar2$. Intuitively, this should render the program insecure, since the policy does only allow the declassification of $hvar2$ at declassification assignments with the identifier 2. The definition of WHERE&WHAT$_{initial}$-Security classifies the program as insecure. We can construct a counter example by choosing $s_1(lvar1) = s_1(lvar2) = s_2(lvar1) = s_2(lvar2) = s_1(hvar2) = s_2(hvar2) = s_1(hvar1) = 0$ and $s_2(hvar1) = 1$. These two memory states s_1 and s_2 fulfill $s_1 =_{low}^{\mathcal{L}(2)} s_2$. The execution step for the assignment hvar2 := hvar1 leads to the two memory states s_1' and s_2' with $s_1'(hvar2) = 0 \neq 1 = s_2'(hvar2)$ and in consequence $s_1' =_{low}^{\mathcal{L}(2)} s_2'$ does not hold and $\forall loc \in Loc : s_1 =_D^{\mathcal{L}(loc)} s_2 \Longrightarrow s_1' =_D^{\mathcal{L}(loc)} s_2'$ is

violated.

If the definition used the whole set of escape hatches like WHAT-Security does, the program would be considered secure, because whenever two states s_1 and s_2 would differ in $hvar1$ the states would not fulfill $s_1 =^{\mathcal{H}}_{low} s_2$ and in consequence the right side of the implication is not needed to hold in order to fulfill the property.

As this example shows the property does not only allow a localization of the declassifications, but even reveals the transitive information flow that can be caused by assignments to variables that occur in escape hatches.

A closer look at the characterization formulas of WHERE&WHAT$_{initial}$-Security and WHAT-Security raises the question why we did not use the restriction of the information flow as a second property and then use the conjunction of both properties as it is done with WHERE-Security and WHAT-Security. The splitting of both requirements into two properties leaves the problem that required information that may be dynamically updated would be required to be propagated to both properties which is cumbersome. While the restriction of information flow into the escape hatches does not require the calculation of dynamic information, the explicit reference points from [LM09a] use such information and as we have already mentioned, we think of this section not only as a introduction of the novel property, but as a guideline how to introduce controls for the information flow into the escape hatches in general and think that the integration of the explicit reference points is a good starting point for future work.

Furthermore, we think that the integration of the control on the right side of the implication is a good choice, because it keeps all the requirements that an execution step must fulfill on the right side of the implication and therefore we assume that it is easier to understand the intuition behind the property.

The property WHERE&WHAT$_{initial}$-Security achieves the same control of declassifications as WHERE&WHAT$_{local}$-Security, but additionally it controls the information flow into the escape hatches in a way that captures our intuition for secure information flow.

We have seen that WHAT$_1$-Security violates the principle of Monotonicity of release. The same holds for WHERE&WHAT$_{initial}$-Security. It would be possible to create a property in the fashion of WHAT$_2$-Security to circumvent this problem. In order to do this we need a quantification over all sets of escape hatches and can then adapt the approach from the definition of WHAT$_2$-Security. We want to leave this open for future work.

4.5. Scheduler-independence of WHERE&WHAT$_{initial}$-Security

The property WHERE&WHAT$_{initial}$-Security captures our intuition of confidentiality very closely. In order to show that the property is adequate for a multi-threaded setting we show now that the property is scheduler-independent with respect to the class of the *low*-secure σ-schedulers. We follow the same approach as we did with in the scheduler-independence proof for WHERE-Security and WHAT-Security.

The first step is to extend the thread pool semantics to keep track of the identifiers for

$$\frac{\langle\!|C_i, s|\!\rangle \twoheadrightarrow_{\mathrm{o}} \langle\!|\overrightarrow{W}, s'|\!\rangle}{\langle\!|H, \langle C_0 \dots C_i \dots C_{n-1}\rangle, s|\!\rangle \xrightarrow{p}_{\mathrm{o}} \langle\!|H(i, n+|\overrightarrow{W}|-1), \langle C_0 \dots C_{i-1}\overrightarrow{W}C_{i+1} \dots C_{n-1}\rangle, s'|\!\rangle}$$

$$\frac{\langle\!|C_i, s|\!\rangle \twoheadrightarrow_{loc}^{\mathcal{D}_1 \to D_2} \langle\!|\overrightarrow{W}, s'|\!\rangle}{\langle\!|H, \langle C_0 \dots C_i \dots C_{n-1}\rangle, s|\!\rangle \xrightarrow{p}_{loc}^{\mathcal{D}_1 \to D_2} \langle\!|H(i, n+|\overrightarrow{W}|-1), \langle C_0 \dots C_{i-1}\overrightarrow{W}C_{i+1} \dots C_{n-1}\rangle, s'|\!\rangle}$$

Figure 16: Semantics of Thread Pools with Identifiers for Localized Hatches

the localized hatches. Since this information is already present in the MWL semantics we can just propagate this information to the thread pool semantics. Figure 16 shows the updated thread pool semantics.

Definition 25 ***σ-specific WHERE&WHAT$_{initial}$-Security:***
A σ-specific strong $(D, \rightsquigarrow, \mathcal{L})$-bisimulation *is a symmetric relation R on command vectors of equal size that satisfies the entire formula in Figure 17.The relation $\approx^{\rightsquigarrow,\mathcal{L}}_{D,\sigma}$ is the union of all strong σ-specific $(D, \rightsquigarrow, \mathcal{L})$-bisimulations. A program $\overrightarrow{C}$ has secure information flow for a given scheduler σ while complying with the restrictions* where *declassification can occur if $\overrightarrow{C} \approx^{\rightsquigarrow,\mathcal{L}}_{D,\sigma} \overrightarrow{C}$ holds for all $D \in \mathcal{D}$ (brief: $\overrightarrow{C}$ is σ-WHERE&WHAT$_{initial}$-secure or $\overrightarrow{C} \in \sigma$-WHERE&WHAT$_{initial}$).*

The characterization formula uses all the requirements introduced with WHERE&WHAT$_{initial}$-Security, but uses the system configurations instead of the thread configurations. Due to this change we can now argue about the scheduling. In order to capture the additional information of the scheduling histories, we require σ-equivalence for the indistinguishability of configurations.

The intuition behind this property is that either the indistinguishability of the states based only on the D-visible information is preserved, or a declassification occurs. An occurring declassification must fulfill the same restriction as in WHERE&WHAT$_{initial}$-Security. This means that the declassification obeys the rules for exceptional information flow and the two memory states were already distinguishable before performing the step for at least one domain in the sources and for the domain D with the knowledge of the evaluated expressions that may get declassified at this declassification assignment due to the localized hatches.

Additionally, the probability of reaching the class of system configurations that is indistinguishable from the resulting configurations is equal for both configurations before the step. It is important to recognize that the equivalence classes of system configurations that are reached with the execution step from $\langle\!|H_1, \overrightarrow{C}_1, s_1|\!\rangle$ and from $\langle\!|H_2, \overrightarrow{C}_2, s_2|\!\rangle$ are not the same equivalence class in the case where a declassification occurs, just like in σ-specific WHERE-Security. This is captured in (ii) and captures the intuition that an observer should not be able to distinguish the configurations by approximating the probabilities of an execution step to be performed.

$$
\begin{aligned}
&\forall H_1, H_2, H_1' : \forall \overrightarrow{C}_1' : \forall s_1, s_2, s_1' : \\
&\langle\!\langle H_1, \overrightarrow{C}_1, s_1 \rangle\!\rangle \rightarrow \langle\!\langle H_1', \overrightarrow{C}_1', s_1' \rangle\!\rangle \wedge H_1 =_\sigma H_2 \wedge \overrightarrow{C}_1 R \overrightarrow{C}_2 \wedge s_1 =_D s_2 \\
&\Longrightarrow \exists H_2' : \exists \overrightarrow{C}_2' : \exists s_2' : \\
&\quad \langle\!\langle H_2, \overrightarrow{C}_2, s_2 \rangle\!\rangle \rightarrow \langle\!\langle H_2', \overrightarrow{C}_2', s_2' \rangle\!\rangle \\
&\quad (i) \wedge \left[\begin{array}{l}
H_1' =_\sigma H_2' \wedge \overrightarrow{C}_1' R \overrightarrow{C}_2' \\
\wedge \left[\begin{array}{l}
s_1' =_D s_2' \vee \left[\begin{array}{l}
\exists \mathcal{D}_1, \{D_2\} \subseteq \mathcal{D} : \exists loc \in Loc : \\
\left[\begin{array}{l}
\langle\!\langle H_1, \overrightarrow{C}_1, s_1 \rangle\!\rangle \rightarrow_{loc}^{\mathcal{D}_1 \rightarrow D_2} \langle\!\langle H_1', \overrightarrow{C}_1', s_1' \rangle\!\rangle \\
\wedge \forall D_1 \in \mathcal{D}_1 : (D_1 \leq D_2 \vee D_1 \rightsquigarrow D_2) \\
\wedge D_2 \leq D \wedge \exists D_1 \in \mathcal{D} : s_1 \neq_{D_1} s_2 \\
\wedge s_1 \neq_D^{\mathcal{L}(loc)} s_2
\end{array}\right]
\end{array}\right]
\end{array}\right] \\
\wedge \forall loc \in Loc : (s_1 =_D^{\mathcal{L}(loc)} s_2 \Longrightarrow s_1' =_D^{\mathcal{L}(loc)} s_2')
\end{array}\right] \\
&\quad (ii) \wedge \left[\begin{array}{l}
\sum \{\!|p| \langle\!\langle H_1, \overrightarrow{C}_1, s_1 \rangle\!\rangle \rightarrow_p \langle\!\langle H, \overrightarrow{C}, s \rangle\!\rangle, H \in [H_1']_{=_\sigma}, \overrightarrow{C} \in [\overrightarrow{C}_1']_R, s \in [s_1']_{=_D} |\!\} \\
= \sum \{\!|p| \langle\!\langle H_2, \overrightarrow{C}_2, s_2 \rangle\!\rangle \rightarrow_p \langle\!\langle H, \overrightarrow{C}, s \rangle\!\rangle, H \in [H_2']_{=_\sigma}, \overrightarrow{C} \in [\overrightarrow{C}_2']_R, s \in [s_2']_{=_D} |\!\}
\end{array}\right]
\end{aligned}
$$

Figure 17: Characterization of σ-specific WHERE&WHAT$_{initial}$-Security

Like in the previous sections we call a property scheduler-independent with respect to the class of *low*-secure σ-schedulers, if the property implies that a σ-specific property holds for all *low*-secure schedulers σ. The WHERE&WHAT$_{initial}$-Security is scheduler-independent, because it implies σ-specific WHERE&WHAT$_{initial}$-Security for all schedulers σ.

Theorem 4 ***Scheduler-independence of WHERE&WHAT$_{initial}$-Security:*** *If $\overrightarrow{C}$ is WHERE&WHAT$_{initial}$-secure, then $\overrightarrow{C}$ is σ-WHERE&WHAT$_{initial}$-secure for all σ.*

$$\overrightarrow{C} \approx_D^{\rightsquigarrow,\mathcal{L}} \overrightarrow{C} \Longrightarrow \forall \sigma : \overrightarrow{C} \approx_{D,\sigma}^{\rightsquigarrow,\mathcal{L}} \overrightarrow{C}$$

Proof Scheduler-independence of WHERE&WHAT$_{initial}$-Security: In order to show that WHERE&WHAT$_{initial}$-Security is scheduler-independent, we show that a relation R that fulfills the characterization formula for WHERE&WHAT$_{initial}$-Security also fulfills the characterization formula of σ-specific WHERE&WHAT$_{initial}$-Security for all schedulers σ. So, if we assume that R exists, such that R fulfills the characterization formula of WHERE&WHAT$_{initial}$-Security from Figure 15, then the relation R also fulfills the characterization formula of σ-specific WHERE&WHAT$_{initial}$-Security from Figure 17. Let σ be an arbitrary *low*-secure σ-scheduler. If R fulfills the characterization formula of WHERE&WHAT$_{initial}$-Security, the following must hold:

$$
\begin{aligned}
&\forall H_1, H_2, H_1' : \forall \overrightarrow{C}_1' : \forall s_1, s_2, s_1' : \\
&\langle\!| H_1, \overrightarrow{C}_1, s_1 |\!\rangle \to \langle\!| H_1', \overrightarrow{C}_1', s_1' |\!\rangle \wedge H_1 =_\sigma H_2 \wedge \overrightarrow{C}_1 R \overrightarrow{C}_2 \wedge s_1 =_D s_2 \\
&\Longrightarrow \exists H_2' : \exists \overrightarrow{C}_2' : \exists s_2' : \\
&\quad \langle\!| H_2, \overrightarrow{C}_2, s_2 |\!\rangle \to \langle\!| H_2', \overrightarrow{C}_2', s_2' |\!\rangle \\
&\quad (i) \wedge \left[\begin{array}{l}
H_1' =_\sigma H_2' \wedge \overrightarrow{C}_1' R \overrightarrow{C}_2' \\
\wedge \left[\begin{array}{l}
s_1' =_D s_2' \vee \left[\begin{array}{l}
\exists \mathcal{D}_1, \{D_2\} \subseteq \mathcal{D} : \exists loc \in Loc : \\
\left[\begin{array}{l}
\langle\!| H_1, \overrightarrow{C}_1, s_1 |\!\rangle \twoheadrightarrow_{loc}^{\mathcal{D}_1 \to D_2} \langle\!| H_1', \overrightarrow{C}_1', s_1' |\!\rangle \\
\wedge \forall D_1 \in \mathcal{D}_1 : (D_1 \leq D_2 \vee D_1 \rightsquigarrow D_2) \\
\wedge D_2 \leq D \wedge \exists D_1 \in \mathcal{D} : s_1 \neq_{D_1} s_2 \\
\wedge s_1 \neq_D^{\mathcal{L}(loc)} s_2
\end{array}\right]
\end{array}\right]
\end{array}\right] \\
\wedge \forall loc \in Loc : (s_1 =_D^{\mathcal{L}(loc)} s_2 \Longrightarrow s_1' =_D^{\mathcal{L}(loc)} s_2')
\end{array}\right] \\
&\quad (ii) \wedge \left[\begin{array}{l}
\sum \{\!\{ p | \langle\!| H_1, \overrightarrow{C}_1, s_1 |\!\rangle \to_p \langle\!| H, \overrightarrow{C}, s |\!\rangle, H \in [H_1']_{=_\sigma}, \overrightarrow{C} \in [\overrightarrow{C}_1']_R, s \in [s_1']_{=_D} \}\!\} \\
= \sum \{\!\{ p | \langle\!| H_2, \overrightarrow{C}_2, s_2 |\!\rangle \to_p \langle\!| H, \overrightarrow{C}, s |\!\rangle, H \in [H_2']_{=_\sigma}, \overrightarrow{C} \in [\overrightarrow{C}_2']_R, s \in [s_2']_{=_D} \}\!\}
\end{array}\right]
\end{aligned}
$$

First, we want to show that (ii) holds. If the left side of the implication is fulfilled, we know that $s_1 =_{low} s_2$, because low is defined as the minimum of $\mathcal{D}$ with respect to $\leq$, and $H_1 =_\sigma H_2$. Using Theorem 1 about probability equivalence for equivalent histories and states we can deduce that $\sigma(H_1, s_1) = \sigma(H_2, s_2)$. In combination with the quantification over all the threads in the thread pools and the requirement on the execution steps in the characterization formula of WHERE&WHAT$_{initial}$-Security we can deduce that a one-to-one correspondence exists between the elements of the multi sets and in consequence the sum of the elements of the multi sets is equal and (ii) is fulfilled.

Second, we want to show that (i) holds. We know from the thread pool semantics in Figure 16 that the execution step

$$\langle\!| C_{1,i}, s_1 |\!\rangle \twoheadrightarrow \langle\!| \overrightarrow{W}_1, s_1' |\!\rangle$$

triggers a scheduler step

$$\langle\!| H_1, \overrightarrow{C}_1, s_1 |\!\rangle \to \langle\!| H_1(i, n + |\overrightarrow{W}_1| - 1), \langle C_{1,0} \dots C_{1,i-1} \overrightarrow{W}_1 C_{1,i+1} \dots C_{1,n-1} \rangle, s_1' |\!\rangle$$

and in consequence we can deduce that $H_1' = H_1(i, n + |\overrightarrow{W}_1| - 1)$ and $\overrightarrow{C}_1' = \langle C_{1,0} \dots C_{1,i-1} \overrightarrow{W}_1 C_{1,i+1} \dots C_{1,n-1} \rangle$.

From the characterization formula of WHERE&WHAT$_{initial}$-Security we know that

$$\langle\!| C_{2,i}, s_2 |\!\rangle \twoheadrightarrow \langle\!| \overrightarrow{W}_2, s_2' |\!\rangle$$

exists and can use the thread pool semantics again to deduce that this execution step triggers a scheduler step

$$\langle\!\langle H_2, \overrightarrow{C}_2, s_2 \rangle\!\rangle \rightarrow \langle\!\langle H_2(i, n + |\overrightarrow{W}_2| - 1), \langle C_{2,0} \dots C_{2,i-1} \overrightarrow{W}_2 C_{2,i+1} \dots C_{2,n-1} \rangle, s_2' \rangle\!\rangle$$

and in consequence $H_2' = H_2(i, n + |\overrightarrow{W}_2| - 1)$ and
$\overrightarrow{C}_2' = \langle C_{2,0} \dots C_{2,i-1} \overrightarrow{W}_2 C_{2,i+1} \dots C_{2,n-1} \rangle$.

From the definition of WHERE&WHAT$_{initial}$-Security we know that $|\overrightarrow{C}_1| = |\overrightarrow{C}_2| = n$ and $|\overrightarrow{W}_1| = |\overrightarrow{W}_2|$. In consequence, $n + |\overrightarrow{W}_1| - 1 = n + |\overrightarrow{W}_2| - 1$. From the left side of the implication we know that $H_1 =_\sigma H_2$ and with the Definition 11 of σ-equivalence, we can conclude that

$$H_1' = H_1(i, n + |\overrightarrow{W}_1| - 1) =_\sigma H_2(i, n + |\overrightarrow{W}_2| - 1) = H_2'$$

We can use the quantification over all threads in the thread pool in the characterization formula of WHERE&WHAT$_{initial}$-Security to deduce that all execution steps that can be made by a single thread fulfill the characterization formula point-wise and therefore $C_{1,i} R C_{2,i}$ for all i. Furthermore, we know from the the characterization formula that $\overrightarrow{W}_1 R \overrightarrow{W}_2$. Using the quantification over the threads again, we can deduce that $\langle C_{1,0} \dots C_{1,i-1} \overrightarrow{W}_1 C_{1,i+1} \dots C_{1,n-1} \rangle R \langle C_{2,0} \dots C_{2,i-1} \overrightarrow{W}_2 C_{2,i+1} \dots C_{2,n-1} \rangle$ which in turn means

$$\overrightarrow{C}_1' R \overrightarrow{C}_2'$$

In the case where the resulting memory states are indistinguishable, $s_1' =_D s_2'$ we have the same requirement in WHERE&WHAT$_{initial}$-Security and σ-specific WHERE&WHAT$_{initial}$-Security. In the case where the resulting memory states are distinguishable, we require the existence of an execution step that belongs to a declassification assignment and have the restrictions defined using the information that is calculated in the semantics of the declassification assignment. According to the thread pool semantics, this declassification step triggers a scheduler step

$$\langle\!\langle H_1, \overrightarrow{C}_1, s_1 \rangle\!\rangle \rightarrow_{loc}^{\mathcal{D}_1 \rightarrow D_2} \langle\!\langle H_1(i, n + |\overrightarrow{W}_1| - 1), \langle C_{1,0} \dots C_{1,i-1} \overrightarrow{W}_1 C_{1,i+1} \dots C_{1,n-1} \rangle, s_1' \rangle\!\rangle$$

Since $\mathcal{D}_1$, D_2 and loc are the same as in the execution step and the requirements defined on them are equivalent, the requirements in the characterization formula of σ-specific WHERE&WHAT$_{initial}$-Security are fulfilled, too.

The last part of (i), $\forall loc \in Loc : (s_1 =_D^{\mathcal{L}(loc)} s_2 \implies s_1' =_D^{\mathcal{L}(loc)} s_2')$ is equivalent in both characterization formulas and therefore, if it is fulfilled in WHERE&WHAT$_{initial}$-Security, then it is fulfilled in σ-specific WHERE&WHAT$_{initial}$-Security, too.

Since a relation that fulfills the characterization formula of WHERE&WHAT$_{initial}$-Security also fulfills the characterization formula of σ-specific WHERE&WHAT$_{initial}$-Security for all schedulers σ, $\overrightarrow{C} \approx_D^{\rightsquigarrow,\mathcal{L}} \overrightarrow{C} \implies \forall \sigma : \overrightarrow{C} \approx_{D,\sigma}^{\rightsquigarrow,\mathcal{L}} \overrightarrow{C}$ holds.

■

The scheduler-independence of WHERE&WHAT$_{initial}$-Security makes the property adequate for multi-threaded settings even in the case where the scheduler is unknown, as long as it is safe to assume that the scheduler is in the class of *low*-secure σ-schedulers. To our knowledge this is the first scheduler-independence result for a property that integrates the control of the aspects **where** and **what** that tightly.

4.6. A Sound Type System

A great benefit of formal security properties is the possibility to show the soundness of a well found analysis with respect to the security property. Security type systems are a possible and well understood method to enforce information flow properties. In this section we want to present a type system that enforces WHERE&WHAT$_{initial}$-Security. Since the basic ideas for controlling information flow and declassifications are similar to the controls in WHERE-Security and WHAT-Security, we can adapt the type system from [MR07] to suit the novel WHERE&WHAT$_{initial}$-Security. The type system in this section is a safe approximation of WHERE&WHAT$_{initial}$-Security. That means that it is stricter than the property and therefore may reject some programs that fulfill the property, but it must reject any program that does not fulfill the property.

According to the MWL semantics, memory state changes are caused by assignments and declassification assignments. More specific, the change in the memory state results in the mapping of the evaluated value of the expression on the right side of the assignment to the variable on the right side of the assignment. In consequence, expressions must be type able in the type system to capture the security requirements of the memory state changes in the security types. A security type of an expression is a set of security domains. We require the expressions to be in a prefix notation for simplicity. Figure 18 shows the type rules for expressions in prefix notation.

$$\frac{}{\vdash val : \{\}} \qquad \frac{dom(var) = D}{\vdash var : \{D\}} \qquad \frac{\vdash expr_1 : \mathcal{D}_1 \cdots \vdash expr_n : \mathcal{D}_n}{\vdash op(expr_1, \ldots, expr_n) : \bigcup_{i \in \{1,\ldots,n\}} \mathcal{D}_i}$$

Figure 18: Type Rules for Expressions

The intuition behind the typing rules for expressions is to capture the security domains that the evaluation of the expression relies on. The evaluation of a constant value $val \in Val$ does not depend on any security domain, because the value is a constant in the program code. The evaluation of a variable $var \in Var$ relies only on the security domain of the variable and in consequence the security type of a variable as expression is the set that contains only the security domain of the variable $dom(var)$. The evaluation of an expression that combines sub expressions with an operator $op \in Op$ depends on all the security domains of the sub expressions $\bigcup_{i \in 1,\ldots,n} \mathcal{D}_i$. These type rules are equivalent to the rules in [MR07].

The type rules for expressions capture the intuition behind the function $sources(expr)$. In consequence, the set returned by the function $sources(expr)$ for a given expression

expr equals the security type of the expression *expr* and vice versa.

Definition 26 ***H*-Escaped Sources of an Expression:**
The function H-escaped sources of an expression *(brief:* $escaped(H, expr)$*) returns a set of security domains for an expression expr with respect to a set of escape hatches* H *that is calculated as follows:*

$$escaped(H, expr)) = \begin{cases} \{D\} & (D, expr) \in H \\ \bigcup_{i \in \{1,\ldots,n\}} escaped(H, expr_i) & expr = op(expr_1, \ldots, expr_n) \\ \{\} & expr \in Val \\ \{dom(var)\} & expr \in Var \end{cases}$$

Since we assume that an observer can see information in a domain lower than or equal to his own domain, this function captures the intuition that escape hatches allow an attacker to learn this information, even if it relies on information from a domain higher than his own by removing the source domains for expressions from escape hatches and replacing them with the domain from the escape hatch.

It is noteworthy that an escape hatch may remove lower domains from the sources and replace it with the domain from the escape hatch, but the intuition is that an observer may only learn an expression if all the domains it relies on are lower than or equal to his own and so this is only a problem if all domains of the escape hatch are lower than the domain in the escape hatch and thus an observer might learn the information of the expression regularly, but not with the escape hatch. This is a problem of an inconsistent security policy, since the purpose of the escape hatches was to allow an observer to learn additional information, or in other words release the information to domain that is lower or at least not higher than the source domain.

Definition 27 ***Restricted Information Flow into Escape Hatches:***
An assignment $var := expr$ *has* restricted information flow into escape hatches *(brief:* $\mathcal{L} \swarrow_{var} expr$*), if the following holds:*

$$\forall loc \in Loc : \forall (D', expr') \in \mathcal{L}(loc) : var \in vars(expr') \Longrightarrow (D', expr) \in \mathcal{L}(loc)$$

This definition captures the intuition that a subsequent declassification of an expression that relies on a variable that some information is assigned to must explicitly be allowed to reveal the information assigned to the variable to capture the transitive information flow via the variable.

This restriction of the information flow into escape hatches requires that the occurrence of a variable *var* in an escape hatch that may be used at a declassification location *loc* can only be used on the left side of an assignment, if the expression on the ride side of the assignment is allowed to be declassified to the same domain as the expression that contains the variable from the left side of the assignment. This restriction of the information flow into escape hatches is a safe over approximation of

the restriction $\forall loc : s_1 =_D^{\mathcal{L}(loc)} s_2 \Longrightarrow s_1' =_D^{\mathcal{L}(loc)} s_2'$ in the characterization formula of WHERE&WHAT$_{initial}$-Security, because it requires that the localized hatches explicitly contain the expressions that are assigned to a variable which occurs in a subsequent declassification and therefore, if two memory states result in different evaluations of the expressions the localized hatch set that uses the variable the expression is assigned two renders both states distinguishable and the left side of the implication is not fulfilled for this location.

Theorem 5 $\mathcal{L} \swarrow_{var} expr$ ***is a Sound Approximation for an Initial Reference Point:*** *Let pol be a mls-$(\rightsquigarrow, \mathcal{L})$ policy. Every assignment var:=expr or declassification $[var{:=}expr]_{loc'}$ that fulfills*

$$\forall loc \in Loc : \forall (D', expr') \in \mathcal{L}(loc) : var \in vars(expr') \Longrightarrow (D', expr) \in \mathcal{L}(loc)$$

also fulfills the restriction of information flow into escape hatches

$$\forall loc \in Loc : s_1 =_D^{\mathcal{L}(loc)} s_2 \Longrightarrow s_1' =_D^{\mathcal{L}(loc)} s_2'$$

in the characterization formula.

Proof $\mathcal{L} \swarrow_{var} expr$ is a Sound Approximation for an Initial Reference Point: Let *var* be the left side, *expr* be the right side of the assignment of the declassification, let D be arbitrary and let $s_1 =_D s_2$. According to the semantics of assignments and declassifications $s_1' = s_1 \otimes \{var = m\}$ and $s_2' = s_2 \otimes \{var = n\}$ where $\langle\!|expr, s_1|\!\rangle \downarrow m$ and $\langle\!|expr, s_2|\!\rangle \downarrow n$

For every declassification location *loc* where no escape hatch $(D', expr') \in \mathcal{L}(loc)$ exists such that the expressions relies on the variable $var \in vars(expr')$, we can use the definition of (D, H)-equality to directly conclude that

$$s_1 =_D^{\mathcal{L}(loc)} s_2 \Longrightarrow s_1' =_D^{\mathcal{L}(loc)} s_2'$$

since no expression that may get declassified at those locations is influenced by *var*.

For every declassification location *loc* where an escape hatch relies on *var*, $var \in vars(expr')$ holds for the escape hatch that contains expression that rely on *var*. From the definition of $\mathcal{L} \swarrow_{var} expr$, we know that

$$\forall loc \in Loc : \forall (D', expr') \in \mathcal{L} : var \in vars(expr') \Longrightarrow (D', expr) \in \mathcal{L}(loc)$$

and in consequence at such locations *loc* an escape hatch $(D', expr) \in \mathcal{L}(loc)$ must exist. We distinguish two cases, where D' is invisible for D and where D' is visible for D.

Case 1 (Invisible Escape Hatch): $D' \not\leq D$
Since $D' \not\leq D$, due to the premise, the evaluations of the escape hatches may not get declassified to D and the escape hatch does not fulfill the requirement $D' \leq D$ on the left side of the implication in the formula for (D, H)-equality. Thus, if $s_1 =_D^{\mathcal{L}(loc)} s_2$ is fulfilled, then also $s_1 \otimes \{var = m\} =_D^{\mathcal{L}(loc)} s_2 \otimes \{var = n\}$ holds.

Case 2 (Visible Escape Hatch): $D' \leq D$
If $s_1 =_D^{\mathcal{L}(loc)} s_2$ holds for this location and $D' \leq D$, then $m = n$ according to the definition of (D, H)-equality and in consequence $s_1 \otimes \{var = m\} =_D^{\mathcal{L}(loc)} s_2 \otimes \{var = n\}$.

Since the information flow restriction is fulfilled in both cases

$$\forall loc \in Loc : s_1 =_D^{\mathcal{L}(loc)} s_2 \Longrightarrow s_1 \otimes \{var = m\} =_D^{\mathcal{L}(loc)} s_2 \otimes \{var = n\}$$

holds and $\mathcal{L} \swarrow_{var} expr$ is a safe approximation of the information flow restriction into escape hatches for an initial reference point.

■

We can now use these definitions to create a type system that is a safe over approximation of WHERE&WHAT$_{initial}$-Security.

$$\frac{}{\vdash \textbf{skip}} \quad \frac{\vdash C \quad \vdash \overrightarrow{V}}{\vdash \textbf{fork}(\ C\ \overrightarrow{V}\ )} \quad \frac{\vdash C_0 \cdots \vdash C_{n-1}}{\vdash \langle C_0 \ldots C_{n-1} \rangle} \quad \frac{\vdash C_1 \quad \vdash C_2}{\vdash C_1; C_2} \quad \frac{\vdash B : \{low\} \quad \vdash C}{\vdash \textbf{while}\ B\ \textbf{do}\ C\ \textbf{od}}$$

$$\frac{\vdash C_1 \quad \vdash C_2 \quad \vdash B : \mathcal{D}' \quad \forall D \in \mathcal{D} : (\exists D' \in \mathcal{D}' : D' \not\leq D) \Longrightarrow C_1 \approx_D^{\rightsquigarrow, \mathcal{L}} C_2}{\vdash \textbf{if}\ B\ \textbf{then}\ C_1\ \textbf{else}\ C_2}$$

$$\frac{\vdash expr : \mathcal{D}' \quad \forall D' \in D : D \leq dom(var) \quad \mathcal{L} \swarrow_{var} expr}{\vdash var := expr}$$

$$\frac{\begin{array}{ll} \vdash expr : \mathcal{D}' & \forall D' \in \mathcal{D}' : D(\leq \cup \rightsquigarrow) dom(var) \\ \mathcal{L} \swarrow_{var} expr & \forall D'' \in escaped(\mathcal{L}(loc), expr) : D'' \leq dom(var) \end{array}}{[var := expr]_{loc}}$$

Figure 19: Type Rules for MWL Command Vectors

The command **skip** does not reveal any information, because it does neither have an influence on the memory state nor on the control flow of the program and therefore it is always type able.

The command **fork**($C\ \overrightarrow{V}$), as well as parallel and sequential composition of commands do only influence the memory state and the program vectors by the commands they contain and in consequence are type able, if the commands they contain are type able.

The **while** loops influence the information flow via the control flow of the program. The expression guarding the **while** loop is restricted to *low*, because even if no assignment or declassification occurs in the loop, the termination of the loop might still leak information about the guarding expression.

Example:
Let *pol* be the following mls-$(\rightsquigarrow, \mathcal{L})$ policy:

$\mathcal{D} = \{low, high\}$
$low \leq low$, $low \leq high$, $high \leq high$
$\rightsquigarrow = \emptyset$
$dom(hvar) = high$
$\mathcal{L}(1) = \emptyset$

Let $\overrightarrow{C}$ be the following program:

```
while (hvar > 0)
do
  skip
od
```

Even with no assignment in the loop and therefore no change of the memory state in the loop, this program is intuitively insecure, because it only terminates for $hvar \leq 0$ and in consequence reveals this *high* information to every observer. The resulting command vectors reveal this problem. Executing a step in a state s_1 where $s_1(hvar) = 0$ results in the command vector $\langle\rangle$, whereas executing a step in a state s_2 where $s_2(hvar) = 1$ results in the command vector **skip**; **while** (hvar > 0) **do skip od**. In these two vectors no bisimulation of the execution steps is possible.

An assignment is type able, if the expression that is assigned to the variable consists only of variables which may flow regularly to the domain of the variable. Additionally the information flow into the escape hatches must be restricted as given by the Definition 27.

A declassification is type able, if the expression that is assigned to the variable consists only of information that may flow either regularly or exceptionally to the domain of the variable. Furthermore, the information flow into the escape hatches must be restricted as given by the Definition 27 just like in regular assignments. Additionally, the expression may not reveal any information that may not be revealed at this point. This requirement is formalized with the condition that all the domains in $escaped(\mathcal{L}(loc), expr)$ must be lower than or equal to the domain of the variable.

The type rule for **if** branchings is a special case. The information flow that it causes is not explicit like in the assignments and declassifications, but implicit in the control flow. If the evaluation of the expression relies on information that an observer might not see, both branches must look equal with respect to WHERE&WHAT$_{initial}$-Security to him, because otherwise the observer can learn information about the guard which he is not allowed to learn. In consequence, we require that for any observer that is no allowed to see some of the information the guarding expression relies on both branches look equal. We formalize this requirement with the help of the type of the expression and the help of $\approx_D^{\rightsquigarrow,\mathcal{L}}$.

Theorem 6 ***Soundness of the Type System:*** *Let $\overrightarrow{C}$ be a MWL program. If $\overrightarrow{C}$ is type able with respect to the type rules, then the program $\overrightarrow{C}$ is WHERE&WHAT$_{initial}$-secure.*

Proof Soundness of the Type System: In order to show that the type ability of

the program $\overrightarrow{C}$ implies that $\overrightarrow{C}$ is WHERE&WHAT$_{initial}$-secure, we need to show that $\forall D \in \mathcal{D} : \overrightarrow{C} \approx_D^{\rightsquigarrow,\mathcal{L}} \overrightarrow{C}$ holds whenever $\overrightarrow{C}$ is type able. We can use an induction over the structure of MWL command vectors.

We distinguish the cases by the last applied rule, where the cases for **skip**, assignments and declassifications form the induction basis. Since the characterization formula has the requirement $s_1 =_D s_2$ on the left side of the implication, all cases where $s_1 \neq_D s_2$ automatically fulfill the characterization formula and need no further analysis and therefore we assume in the reminder $s_1 =_D s_2$ if not mentioned otherwise.

skip:

$$\frac{}{\vdash \textbf{skip}}$$

A skip is always type able. Let $s_1 =_D s_2$ be arbitrary. According to the semantics of **skip** the execution steps have the form:

$$\langle\!|\textbf{skip}, s_1|\!\rangle \rightarrowtriangle_o \langle\!|\langle\rangle, s_1|\!\rangle$$

$$\langle\!|\textbf{skip}, s_2|\!\rangle \rightarrowtriangle_o \langle\!|\langle\rangle, s_2|\!\rangle$$

Since no change is made to the memory state, we can directly conclude that the restriction of information flow into escape hatches $\forall loc \in Loc : s_1 =_D^{\mathcal{L}(loc)} s_2 \Longrightarrow s_1' =_D^{\mathcal{L}(loc)} s_2'$ is fulfilled. In addition $\langle\rangle \approx_D^{\rightsquigarrow,\mathcal{L}} \langle\rangle$ holds and the right side of the implication in the characterization formula is fulfilled. Hence **skip** $\approx_D^{\rightsquigarrow,\mathcal{L}}$ **skip** holds.

Assignments:

$$\frac{\vdash expr : \mathcal{D}' \quad \forall D' \in D : D \leq dom(var) \quad \mathcal{L} \swarrow_{var} expr}{\vdash var := expr}$$

Let $s_1 =_D s_2$ be arbitrary. According to the semantics of assignments the execution steps have the form:

$$\langle\!|var := expr, s_1|\!\rangle \rightarrowtriangle_o \langle\!|\langle\rangle, s_1 \otimes \{var = m\}|\!\rangle$$

$$\langle\!|var := expr, s_2|\!\rangle \rightarrowtriangle_o \langle\!|\langle\rangle, s_2 \otimes \{var = n\}|\!\rangle$$

Furthermore, from the premise of the semantic rule, we know that $\langle\!|expr, s_1|\!\rangle \downarrow m$ and $\langle\!|expr, s_2|\!\rangle \downarrow n$.

We must distinguish two cases. Either the assignment is to an invisible variable or it is to a visible variable.

Case 1 (Invisible Assignment): $dom(var) \not\leq D$
Since $dom(var) \not\leq D$ we can conclude from the definition of D-equality that $s_1 =_D s_1 \otimes \{var = m\}$ and $s_2 =_D s_2 \otimes \{var = n\}$. Using the transitivity of D-equality, we can conclude that $s_1 \otimes \{var = m\} =_D s_2 \otimes \{var = n\}$.

Since $\mathcal{L} \swarrow_{var} expr$ is a safe approximation for the information flow into the escape hatches and since $\langle\rangle \approx_D^{\rightsquigarrow,\mathcal{L}} \langle\rangle$, we get as a result that $var := expr \approx_D^{\rightsquigarrow,\mathcal{L}} var := expr$.

Case 2 (Visible Assignment): $dom(var) \leq D$
From the premise of the type rule we know that the type of the expression is $\mathcal{D}'$ and in combination with the premise $\forall D' \in \mathcal{D} : D' \leq dom(var)$ and the type rules for expressions we know that $dom(var') \leq dom(var)$ for every $var' \in vars(expr)$. Using the transitivity of $\leq$ we can conclude with the premise of this case that $dom(var') \leq D$ for every $var' \in vars(expr)$. With that knowledge we can conclude that $\langle\!|expr, s_1|\!\rangle \downarrow m$ and $\langle\!|expr, s_2|\!\rangle \downarrow n$ implies that $m = n$ for every two states that fulfill $s_1 =_D s_2$. Using the definition of D-equality, we can conclude that $s_1 \otimes \{var = m\} =_D s_2 \otimes \{var = n\}$.

Since $\mathcal{L} \swarrow_{var} expr$ is a safe approximation for the information flow into the escape hatches and since $\langle\rangle \approx_D^{\rightsquigarrow,\mathcal{L}} \langle\rangle$, we get as a result that $var := expr \approx_D^{\rightsquigarrow,\mathcal{L}} var := expr$.

Declassifications:

$$\frac{\vdash expr : \mathcal{D}' \qquad \forall D' \in \mathcal{D}' : D(\leq \cup \rightsquigarrow) dom(var) \\ \mathcal{L} \swarrow_{var} expr \qquad \forall D'' \in escaped(\mathcal{L}(loc), expr) : D'' \leq dom(var)}{[var := expr]_{loc}}$$

Let $s_1 =_D s_2$ be arbitrary. According to the semantics of declassification the execution steps have the form:

$$\langle\!|[var{:=}expr]_{loc}, s_1|\!\rangle \rightarrow_{loc}^{\mathcal{D}_1 \to D_2} \langle\!|\langle\rangle, s_1 \otimes \{var = m\}|\!\rangle$$

$$\langle\!|[var{:=}expr]_{loc}, s_2|\!\rangle \rightarrow_{loc}^{\mathcal{D}_1 \to D_2} \langle\!|\langle\rangle, s_2 \otimes \{var = n\}|\!\rangle$$

where $\langle\!|expr, s_1|\!\rangle \downarrow m$ and $\langle\!|expr, s_2|\!\rangle \downarrow n$.

We must distinguish two cases. Either the declassification is to an invisible variable or it is to a visible variable.

Case 1 (Invisible Declassification): $dom(var) \not\leq D$
This case is analogous to the case for invisible assignments.

Case 2 (Visible Declassification): $dom(var) \leq D$
In this case we must distinguish two sub cases, where the evaluation of the expression is either equal in both states, or the evaluation differs in both states.

Case 2.1 (Equal Evaluation): $\forall s_1 =_D s_2 : \langle\!|expr, s_1|\!\rangle \downarrow m \wedge \langle\!|expr, s_2|\!\rangle \downarrow n \wedge m = n$
Due to the premise of this case $m = n$ and with the definition of D-equality $s_1 \otimes \{var = m\} =_D s_2 \otimes \{var = n\}$.

Since $\langle\rangle \approx_D^{\rightsquigarrow,\mathcal{L}} \langle\rangle$ and $\mathcal{L} \swarrow_{var} expr$ is a safe approximation for the information flow into the escape hatches $\lfloor\ var := expr\ \rfloor_{loc} \approx_D^{\rightsquigarrow,\mathcal{L}} \lfloor\ var := expr\ \rfloor_{loc}$ holds.

Case 2.2 (Different Evaluation): $\exists s_1 =_D s_2 : \langle\!|expr, s_1|\!\rangle \downarrow m \wedge \langle\!|expr, s_2|\!\rangle \downarrow n \wedge m \neq n$
According to the semantics $\exists \mathcal{D}_1, \{D_2\} \in \mathcal{D} : \exists loc \in Loc : \langle\!|C_{1,i}, s_1|\!\rangle \rightarrow_{loc}^{\mathcal{D}_1 \to D_2} \langle\!|\overrightarrow{W}, s_1'|\!\rangle$ is fulfilled with $\overrightarrow{W} = \langle\rangle$ and $s_1' = s_1 \otimes \{var = m\}$.

Furthermore, according to the semantics $\mathcal{D}_1 = sources(expr)$, which is captured by the type of the expression, and $D_2 = dom(var)$. Due to the restriction of the type rule $\forall D' \in \mathcal{D}' : D'(\leq \cup \rightsquigarrow) dom(var)$, $\forall D_1 \in \mathcal{D}_1 : (D_1 \leq D_2 \vee D_1 \rightsquigarrow D_2$ from the characterization formula is fulfilled.

Since it is a visible declassification we know that $dom(var) \leq D$ and conclude from $D_2 = dom(var)$ that $D_2 \leq D$ is fulfilled.

From the premise of this case we know that $\langle\!|expr, s_1|\!\rangle \downarrow m \wedge \langle\!|expr, s_2|\!\rangle \downarrow n$ and thus there must be a $var' \in vars(expr)$, such that $s_1(var') \neq s_2(var')$ and due to the definition of the expression types $dom(var') \in \mathcal{D}_1$ and in consequence $\exists D_1 \in \mathcal{D}^1 : s_1 \neq_{D'} s_2$ holds.

Since $s_1 =_D s_2$, but $\exists D_1 \in \mathcal{D}_1 : s_1 \neq_{D_1} s_2$, we can conclude that there exists an $D_1 \in \mathcal{D}_1$ where $D_1 \not\leq D$, but from the premise of the type rule $\forall D'' \in escaped(\mathcal{L}(loc), expr) : D'' \leq dom(var)$ and the definition of $escape(\mathcal{L}(loc), expr)$ that for every $var' \in vars(expr)$ which fulfills $dom(var') \not\leq dom(var)$ and escape hatch $(D'', expr')$ must exist in $\mathcal{L}(loc)$, such that $D'' \leq dom(var)$ and $expr' \in subexpressions(expr)$. In combination with the fact that there exists at least one variable $var' \in vars(expr)$ with $dom(var') \not\leq D$ and in consequence $dom(var') \not\leq dom(var)$ due to the premise of the visible declassification, there must be an escape hatch with an $expr'$ that relies on this variable and in consequence $\langle\!|expr', s_1|\!\rangle \downarrow o$ and $\langle\!|expr', s_2|\!\rangle \downarrow p$ with $o \neq p$. With the definition of (D, H)-equality $s_1 \neq_D^{\mathcal{L}} s_2$ is fulfilled.

Since $\mathcal{L} \swarrow_{var} expr$ is a safe approximation for the information flow into the escape hatches and $\langle\rangle \approx_D^{\rightsquigarrow, \mathcal{L}} \langle\rangle$ holds, $[\ var := expr\ ]_{loc} \approx_D^{\rightsquigarrow, \mathcal{L}} [\ var := expr\ ]_{loc}$ holds, too.

if Branching:

$$\frac{\vdash C_1 \quad \vdash C_2 \quad \vdash B : \mathcal{D}' \quad \forall D \in \mathcal{D} : (\exists D' \in \mathcal{D}' : D' \not\leq D) \Longrightarrow C_1 \approx_D^{\rightsquigarrow, \mathcal{L}} C_2}{\vdash \textbf{if } B \textbf{ then } C_1 \textbf{ else } C_2}$$

We have to distinguish two cases for **if** branchings. In the first case the guarding expression evaluates to equal values for all states D-equal states and in the second case the guarding expression may evaluate to different values for D-equal states.

Case 1 (Equal Evaluation): $\forall s_1 =_D s_2 : \langle\!|B, s_1|\!\rangle \downarrow m \wedge \langle\!|B, s_2|\!\rangle \downarrow n \wedge m = n$
Let $\langle\!|B, s_1|\!\rangle \downarrow True$. From the premise of this case we know that $\langle\!|B, s_2|\!\rangle \downarrow True$, too. According to the semantics of **if**, the execution steps have the form

$$\langle\!|\textbf{if } B \textbf{ then } C_1 \textbf{ else } C_2 \textbf{ fi}, s_1|\!\rangle \rightarrow_o \langle\!|C_1, s_1|\!\rangle$$

$$\langle\!|\textbf{if } B \textbf{ then } C_1 \textbf{ else } C_2 \textbf{ fi}, s_2|\!\rangle \rightarrow_o \langle\!|C_1, s_2|\!\rangle$$

Since the memory state does not change, $s_1 =_D s_2$ still holds and the restriction of information flow into the escape hatches is fulfilled. Furthermore, from the premise

of the type rule, we know that $\overrightarrow{C}_1$ is type able and conclude with the induction hypothesis that $\overrightarrow{C} \approx_D^{\rightsquigarrow,\mathcal{L}} \overrightarrow{C}$ holds and in consequence **if** B **then** C_1 **else** C_2 **fi** $\approx_D^{\rightsquigarrow,\mathcal{L}}$ **if** B **then** C_1 **else** C_2 **fi** holds. The argument is analogous for $\langle\!|B, s_1|\!\rangle \downarrow False$.

Case 2 (Different Evaluation): $\exists s_1 =_D s_2 : \langle\!|B, s_1|\!\rangle \downarrow m \wedge \langle\!|B, s_2|\!\rangle \downarrow n \wedge m \neq n$
Let $\langle\!|B, s_1|\!\rangle \downarrow True$. From the premise of this case we know that $\langle\!|B, s_2|\!\rangle \downarrow False$. According to the semantics of **if**, the execution steps have the form:

$$\langle\!|\textbf{if } B \textbf{ then } C_1 \textbf{ else } C_2 \textbf{ fi}, s_1|\!\rangle \rightarrow_o \langle\!|C_1, s_1|\!\rangle$$

$$\langle\!|\textbf{if } B \textbf{ then } C_1 \textbf{ else } C_2 \textbf{ fi}, s_2|\!\rangle \rightarrow_o \langle\!|C_2, s_2|\!\rangle$$

Since $s_1 =_D s_2$, but $\langle\!|B, s_1|\!\rangle \downarrow True$ and $\langle\!|B, s_2|\!\rangle \downarrow False$, we know that there must be at least one $var \in vars(B)$ with $dom(var) \not\leq D$. According to the premise $\forall D \in \mathcal{D} : \exists D' \in \mathcal{D}' : D' \not\leq D \Longrightarrow C_1 \approx_D^{\rightsquigarrow,\mathcal{L}} C_2$ of the type rule, $C_1 \approx_D^{\rightsquigarrow,\mathcal{L}} C_2$ must hold, because D fulfills the left side of the implication of this requirement. Since the executions step did not change the memory state $s_1 =_D s_2$ still holds and the restriction of information flow into escape hatches is fulfilled. As result **if** B **then** C_1 **else** C_2 **fi** $\approx_D^{\rightsquigarrow,\mathcal{L}}$ **if** B **then** C_1 **else** C_2 **fi** holds. The argument is analogous for $\langle\!|B, s_1|\!\rangle \downarrow False$.

while Loops:

$$\frac{\vdash B : \{low\} \qquad \vdash C}{\vdash \textbf{ while } B \textbf{ do } C \textbf{ od}}$$

According to the premise of the type rule the type of the guarding expression B must be $\{low\}$ and in consequence B can only rely on variables from the domain low. Since low is the minimum of $\mathcal{D}$ with respect to $\leq$, $\forall D \in \mathcal{D} : low \leq D$. In consequence, for all memory states s_1 and s_2 that fulfill $s_1 =_D s_2$ the guarding expression evaluates to equal values. According to the premise of the type rule, C must be type able and in consequence $C \approx_D^{\rightsquigarrow,\mathcal{L}} C$.

We must distinguish two cases. In the first case the guard evaluates to $False$ and in the second case the guard evaluates to $True$.

Case 1 (Guard evaluates to False): $\langle\!|B, s_1|\!\rangle \downarrow False \wedge \langle\!|B, s_2|\!\rangle \downarrow False$
According to the semantics, the execution steps have the following form:

$$\langle\!|\textbf{while } B \textbf{ do } C \textbf{ od}, s_1|\!\rangle \rightarrow_o \langle\!|\langle\rangle, s_1|\!\rangle$$

$$\langle\!|\textbf{while } B \textbf{ do } C \textbf{ od}, s_2|\!\rangle \rightarrow_o \langle\!|\langle\rangle, s_2|\!\rangle$$

Since the memory states are not changed, the information flow into the escape hatches is fulfilled and $s_1 =_D s_2$ still holds. In addition $\langle\rangle \approx_D^{\rightsquigarrow,\mathcal{L}} \langle\rangle$ and in consequence **while** B **do** C **od** $\approx_D^{\rightsquigarrow,\mathcal{L}}$ **while** B **do** C **od** holds.

Case 2 (Guard evaluates to True): $\langle\!|B, s_1|\!\rangle \downarrow True \wedge \langle\!|B, s_2|\!\rangle \downarrow True$
According to the semantics, the execution steps have the following form:

$$\langle\!|\textbf{while } B \textbf{ do } C \textbf{ od}, s_1|\!\rangle \rightarrowtriangle_{\mathrm{o}} \langle\!|C; \textbf{while } B \textbf{ do } C \textbf{ od}, s_1|\!\rangle$$

$$\langle\!|\textbf{while } B \textbf{ do } C \textbf{ od}, s_2|\!\rangle \rightarrowtriangle_{\mathrm{o}} \langle\!|C; \textbf{while } B \textbf{ do } C \textbf{ od}, s_2|\!\rangle$$

The resulting command is a sequence. From the premise of the type rule we know that C is type able and with the induction hypothesis can conclude that $C \approx_D^{\rightsquigarrow,\mathcal{L}} C$ holds.

According to the semantics of sequences the execution step for the sequence is

$$\langle\!|C; \textbf{while } B \textbf{ do } C \textbf{ od}, s_1|\!\rangle\langle\!|\textbf{while } B \textbf{ do } C \textbf{ od}, s_1'|\!\rangle$$

$$\langle\!|C; \textbf{while } B \textbf{ do } C \textbf{ od}, s_2|\!\rangle\langle\!|\textbf{while } B \textbf{ do } C \textbf{ od}, s_2'|\!\rangle$$

Since $C \approx_D^{\rightsquigarrow,\mathcal{L}} C$, we can conclude that the restriction of information flow into escape hatches is preserved in this command and that s_1' and s_2' fulfill the requirements for WHERE&WHAT$_{initial}$-Security.

Since the loop body C remains the same in all iteration of the loop, $C \approx_D^{\rightsquigarrow,\mathcal{L}} C$ always holds and if the loop finally terminates case 1 is fulfilled. Thus **while** B **do** C **od** $\approx_D^{\rightsquigarrow,\mathcal{L}}$ **while** B **do** C **od** holds.

fork:

$$\frac{\vdash C \qquad \vdash \overrightarrow{V}}{\vdash \textbf{fork}(\ C\ \overrightarrow{V}\ )}$$

According to the semantics of fork the execution steps have the form:

$$\langle\!|\textbf{fork}(\ C\ \overrightarrow{V}\ ), s_1|\!\rangle \rightarrowtriangle_{\mathrm{o}} \langle\!|C\ \overrightarrow{V}, s_1|\!\rangle$$

$$\langle\!|\textbf{fork}(\ C\ \overrightarrow{V}\ ), s_2|\!\rangle \rightarrowtriangle_{\mathrm{o}} \langle\!|C\ \overrightarrow{V}, s_2|\!\rangle$$

The premise of the type rule require that C is type able and that $\overrightarrow{V}$ is type able. With the induction hypothesis we can conclude that $C \approx_D^{\rightsquigarrow,\mathcal{L}} C$ and $\overrightarrow{V} \approx_D^{\rightsquigarrow,\mathcal{L}} \overrightarrow{V}$ hold. Using the unwinding of command vectors in the characterization formula we can conclude that $\overrightarrow{V} \approx_D^{\rightsquigarrow,\mathcal{L}} \overrightarrow{V}$ holds point-wise, too. Using this unwinding again, we can conclude that $C\overrightarrow{V} \approx_D^{\rightsquigarrow,\mathcal{L}} C\overrightarrow{V}$.

Since the memory state is unchanged the information flow into the escape hatches is fulfilled and $s_1 =_D s_2$ still holds. In consequence, **fork**$(C\ \overrightarrow{V}) \approx_D^{\rightsquigarrow,\mathcal{L}}$ **fork**$(C\ \overrightarrow{V})$ holds.

Sequence of Commands:

$$\frac{\vdash C_1 \qquad \vdash C_2}{\vdash C_1; C_2}$$

According to the semantics of sequences the execution steps have the form:

$$\langle\!|C_1; C_2, s_1|\!\rangle \rightarrowtriangle \langle\!|C_2, s_1'|\!\rangle$$

$$\langle\!|C_1; C_2, s_1|\!\rangle \rightarrowtriangle \langle\!|C_2, s_1'|\!\rangle$$

where $\rightarrowtriangle$ is either $\rightarrowtriangle_{\mathrm{o}}$ or $\rightarrowtriangle_{loc}^{\mathcal{D}_1 \rightarrow D_2}$. The premise of the type rule requires that C_1 and C_2 to be type able and with the induction hypothesis we can conclude that $C_1 \approx_D^{\rightsquigarrow,\mathcal{L}} C_1$ and $C_2 \approx_D^{\rightsquigarrow,\mathcal{L}} C_2$.

Since $C_1 \approx_D^{\rightsquigarrow,\mathcal{L}} C_1$ holds, we can conclude that the restriction of information flow is fulfilled in this execution step and that either $s_1' =_D s_2'$ or the restrictions for declassification are fulfilled. In combination with $C_2 \approx_D^{\rightsquigarrow,\mathcal{L}} C_2$, the characterization formula is fulfilled in any case and thus $C_1; C_2 \approx_D^{\rightsquigarrow,\mathcal{L}} C_1; C_2$ holds.

Parallel Commands:

$$\frac{\vdash C_0 \cdots \vdash C_{n-1}}{\vdash \langle C_0 \ldots C_{n-1} \rangle}$$

The premise of the type rule for parallel commands requires that each of the parallel commands is type able and in combination with the induction hypothesis $C_i \approx_D^{\rightsquigarrow,\mathcal{L}} C_i$ for all $i \in \{0, \ldots, n-1\}$. Using the unwinding of command vectors in the characterization formula we can conclude that $\langle C_0 \ldots C_{n-1} \rangle \approx_D^{\rightsquigarrow,\mathcal{L}} \langle C_0 \ldots C_{n-1} \rangle$.

■

Now we have seen that the type system is sound with respect to WHERE&WHAT$_{initial}$-Security, but one open problem remains. The type rule for **if** branchings has a semantic side condition that relies on the definition of WHERE&WHAT$_{initial}$-Security in order to prevent information about the guard to be leaked via the different commands of the **then** branch and the **else** branch. This semantic side condition renders the type system undecidable. The authors of [MS04] use safe approximation relations based on the syntax of the programs to solve the problem of undecidability of such semantic side conditions. We use this approach to make the type system decidable and adapt the Definition of Non k-Visible Equality to take the escape hatches into account.

Definition 28 ***Safe Approximation Relation:***
A family $\{R_D^{\mathcal{L}}\}_{D \in \mathcal{D}, \mathcal{L} \text{ is a localized hatches function}}$ *of relations on commands is a safe approximation relation if whenever two WHERE&WHAT$_{initial}$-secure commands* C_1 *and* C_2 *are related to each other* $C_1 (R_D^{\mathcal{L}}) C_2$ *with respect to the security domain* $D \in \mathcal{D}$ *and a function of localized hatches* $\mathcal{L}$ *then* $\forall D' \in \mathcal{D} : D \not\leq D' \Longrightarrow C_1 \approx_{D'}^{\rightsquigarrow,\mathcal{L}} C_2$ *holds.*

Definition 29 ***Non D-Visible Equality with Localized Hatches $\mathcal{L}$:*** *Let $D \in \mathcal{D}$ be a security domain and $\mathcal{L}$ be a function of localized hatches.* Non D-visible equality with localized hatches $\mathcal{L}$ *(brief: $\sim_D^{\mathcal{L}}$) is a congruence relation (transitive, reflexive and symmetric) on commands that fulfills the following rules:*

$$\frac{D \leq dom(var) \qquad \nexists loc \in Loc : \exists (D', expr') \in \mathcal{L}(loc) : var \in vars(expr') \wedge D' \leq D}{var := expr \sim_D^{\mathcal{L}} \textbf{\textit{skip}}}$$

$$\frac{D \leq dom(var) \qquad \nexists loc \in Loc : \exists (D', expr') \in \mathcal{L}(loc) : var \in vars(expr') \wedge D' \leq D}{[\ var := expr\ ]_{loc'} \sim_D^{\mathcal{L}} \textbf{\textit{skip}}}$$

Theorem 7 ***Non D-Visible Equality with $\mathcal{L}$ is a Safe Approximation Relation:*** *The family of relations $\sim_D^{\mathcal{L}}$ is a safe approximation relation of $\cong_D^{\rightsquigarrow,\mathcal{L}}$.*

Proof Non D-Visible Equality with $\mathcal{L}$ is a Safe Approximation Relation: This proof is almost similar to the proof for Non k-Visible Equality [MS04] and was inspired by the proof in [Rei06]. The differences are in those execution steps that change the memory state and therefore may violate the restriction of information flow into escape hatches. Let C_1 and C_2 be two WHERE&WHAT$_{initial}$-secure commands. We will use an induction over the smallest number of steps to deduce $C_1 \sim_D^{\mathcal{L}} C_2$ with the rules that we explicitly present in the proof to show that
$\forall D' \in \mathcal{D} : D' \leq D \Longrightarrow \overrightarrow{C}_1 \cong_{D'}^{\rightsquigarrow,\mathcal{L}} \overrightarrow{C}_2$ holds.

Let $D' \leq D$ be arbitrary security domains. We distinguish the cases by the last applied rule, where the rules for invisible assignments, declassifications and reflexivity form the induction basis.

Invisible Assignments:

$$\frac{D \leq dom(var) \qquad \nexists loc \in Loc : \exists (D', expr') \in \mathcal{L}(loc) : var \in vars(expr') \wedge D' \leq D}{\text{var} := \text{expr} \sim_D^{\mathcal{L}} \textbf{skip}}$$

Let $s_1 =_{D'} s_2$ be arbitrary memory states. Let $\langle\!\langle \overrightarrow{C}'_1, s'_1 \rangle\!\rangle$ be the thread configuration after performing the execution step of the assignment

$$\langle\!\langle \text{var} := \text{expr}, s_1 \rangle\!\rangle \rightarrow_{\text{o}} \langle\!\langle \overrightarrow{C}'_1, s'_1 \rangle\!\rangle$$

According to the semantics, $\overrightarrow{C}'_1 = \langle\rangle$ and $s'_1 = s_1 \otimes \{var = n\}$.

Since $D \leq dom(var)$ and $D \not\leq D'$, we know that $dom(var) \not\leq D'$ and in consequence $s_1 =_{D'} s'_1$ and using $s_1 =_{D'} s_2$ and the transitivity of $=_{D'}$ we can conclude that $s'_1 =_D s_2$.

The semantics of the execution step for **skip** is

$$\langle\!\langle \textbf{skip}, s_2 \rangle\!\rangle \rightarrow_{\text{o}} \langle\!\langle \langle\rangle, s_2 \rangle\!\rangle$$

Since $\langle\rangle \cong_D^{\rightsquigarrow,\mathcal{L}} \langle\rangle$ and $s'_1 =_D s_2$, we only need to show that
$\forall loc \in Loc : s_1 =_{D'}^{\mathcal{L}(loc)} s_2 \Longrightarrow s'_1 =_{D'}^{\mathcal{L}(loc)} s'_2$ holds. This follows from the second premise of the rule, because no escape hatch with an expression that contains the variable *var*

may exist that allows a declassification to a domain $D'' \leq D$ and thus $D'' \leq D'$. In consequence, every escape hatch that relies on *var* has no influence on the (D', H)-equivalence of memory states, even if the escape hatch is in the set H.

As result, var := expr $\approx^{\rightsquigarrow,\mathcal{L}}_{D'}$ **skip** holds.

Invisible Declassification:

$$\frac{D \leq dom(var) \qquad \nexists loc \in Loc : \exists (D', expr') \in \mathcal{L}(loc) : var \in vars(expr') \wedge D' \leq D}{[\ \text{var} := \text{expr}\ ]_{loc'} \sim^{\mathcal{H}}_{D} \textbf{skip}}$$

The proof is analog to the proof for invisible assignments.

Reflexivity:

$$\frac{C_1 = C_2}{C_1 \sim^{\mathcal{L}}_{D} C_2}$$

Since we require that C_1 is WHERE&WHAT$_{initial}$-secure, $C_1 \cong^{\rightsquigarrow,\mathcal{L}}_{D} C_2$ follows directly from the definition of WHERE&WHAT$_{initial}$-Security.

Symmetry:

$$\frac{C_1 \sim^{\mathcal{L}}_{D} C_2}{C_2 \sim^{\mathcal{L}}_{D} C_1}$$

According to the induction hypothesis we can deduce from $C_1 \sim^{\mathcal{L}}_{D} C_2$ that $C_1 \cong^{\rightsquigarrow,\mathcal{L}}_{D'} C_2$ and with the symmetry of $\cong^{\rightsquigarrow,\mathcal{L}}_{D'}$ also $C_2 \cong^{\rightsquigarrow,\mathcal{L}}_{D'} C_1$.

Transitivity:

$$\frac{C_1 \sim^{\mathcal{L}}_{D} C_3 \qquad C_3 \sim^{\mathcal{L}}_{D} C_2}{C_1 \sim^{\mathcal{L}}_{D} C_2}$$

According to the induction hypothesis we can deduce from $C_1 \sim^{\mathcal{L}}_{D} C_3$ that $C_1 \cong^{\rightsquigarrow,\mathcal{L}}_{D'} C_3$ and from $C_3 \sim^{\mathcal{L}}_{D} C_2$ that $C_3 \cong^{\rightsquigarrow,\mathcal{L}}_{D'} C_2$. With the transitivity of $\cong^{\rightsquigarrow,\mathcal{L}}_{D'}$ we can deduce that $C_1 \cong^{\rightsquigarrow,\mathcal{L}}_{D'} C_2$.

Sequences:

$$\frac{C_1 \sim^{\mathcal{L}}_{D} C_1' \qquad C_2 \sim^{\mathcal{L}}_{D} C_2'}{C_1; C_2 \sim^{\mathcal{L}}_{D} C_1'; C_2'}$$

According to the induction hypothesis we can deduce from the premise of the rule that $C_1 \cong^{\rightsquigarrow,\mathcal{L}}_{D'} C_1'$ and $C_2 \cong^{\rightsquigarrow,\mathcal{L}}_{D'} C_2'$. According to the semantics of sequences

$$\langle\!| C_1; C_2, s_1 |\!\rangle \rightarrowtriangle \langle\!| \overrightarrow{C}, s |\!\rangle$$

we must distinguish three different cases.

Case 1:

$$\langle\!| C_1, s_1 |\!\rangle \rightarrowtriangle_{\text{o}} \langle\!| \langle\rangle, s_2 |\!\rangle$$

Due to the premise of this case: $\overrightarrow{C} = C_2$.
Since $C_1 \cong^{\rightsquigarrow,\mathcal{L}}_{D'} C_1'$, we know that a memory state s' exists, such that $\langle\!| C_1', s_1' |\!\rangle \rightarrow_{\text{o}} \langle\!| \langle\rangle, s' |\!\rangle$

and $s =_{D'} s'$ as well as the localized hatch sets do not reveal more information than before the transition due to $\forall loc \in Loc : s_1 =_{D'}^{\mathcal{L}(loc)} s_2 \implies s =_{D'}^{\mathcal{L}(loc)} s'$. Using the semantics of sequences again,

$$\langle\!| C_1'; C_2', s_1' |\!\rangle \rightarrow_{\mathrm{o}} \langle\!| C_2', s' |\!\rangle$$

and due to the premise for sequences $C_2 \approx_{D'}^{\sim,\mathcal{L}} C_2'$. Since the intermediary commands and the intermediary states fulfill the condition for $\approx_{D'}^{\sim,\mathcal{L}}$, $C_1; C_2 \approx_{D'}^{\sim,\mathcal{L}} C_1'; C_2'$ holds.

Case 2:

$$\langle\!| C_1, s_1 |\!\rangle \rightarrow_{loc}^{\mathcal{D}_1 \rightarrow D_2} \langle\!| \langle\rangle, s_2 |\!\rangle$$

Due to the premise of this case $\overrightarrow{C} = C_2$.
Since $C_1 \approx_{D'}^{\sim,\mathcal{L}} C_1'$, we know that a memory state s' exists, such that

$$\langle\!| C_1', s_1' |\!\rangle \rightarrow_{loc}^{\mathcal{D}_1 \rightarrow D_2} \langle\!| \langle\rangle, s' |\!\rangle$$

and that either $s_1 =_{D'} s_2$ or the restrictions for declassification are fulfilled. The localized hatch sets do not reveal more information analog to the argument in case 1.

Using the semantics of sequences again,

$$\langle\!| C_1'; C_2', s_1' |\!\rangle \rightarrow_{loc}^{\mathcal{D}_1 \rightarrow D_2} \langle\!| C_2', s' |\!\rangle$$

and due to the premise for sequences $C_2 \approx_{D'}^{\sim,\mathcal{L}} C_2'$. Since the intermediary commands and the intermediary states fulfill the condition for $\approx_{D'}^{\sim,\mathcal{L}}$, $C_1; C_2 \approx_{D'}^{\sim,\mathcal{L}} C_1'; C_2'$ holds.

Case 3:

$$\langle\!| C_1, s_1 |\!\rangle \rightarrow_{\mathrm{o}} \langle\!| C\overrightarrow{V}, s |\!\rangle$$

Due to the premise of the case $\overrightarrow{C} = (C; C_2)\overrightarrow{V}$. Since $C_1 \approx_{D'}^{\sim,\mathcal{L}} C_1'$, we know that a command C', a command vector $\overrightarrow{D}'$ and s' exist, such that

$$\langle\!| C_1', s_1' |\!\rangle \rightarrow_{\mathrm{o}} \langle\!| C'\overrightarrow{V}', s' |\!\rangle$$

where $s =_{D'} s'$ and $C\overrightarrow{V} \approx_{D'}^{\sim,\mathcal{L}} C'\overrightarrow{V}$. Using the unwinding of command vectors in the characterization formula we can conclude that $C \approx_{D'}^{\sim,\mathcal{L}} C'$ and $\overrightarrow{V} \approx_{D'}^{\sim,\mathcal{L}} \overrightarrow{V}'$.

Using the semantics for sequences again,

$$\langle\!| C_1'; C_2', s_1' |\!\rangle \rightarrow_{\mathrm{o}} \langle\!| (C'; C_2')\overrightarrow{C}', s' |\!\rangle$$

Since $C \approx_{D'}^{\sim,\mathcal{L}} C'$ and $C_2 \approx_{D'}^{\sim,\mathcal{L}} C_2'$ we can conclude with the help of case 1 that $C; C_2 \approx_{D'}^{\sim,\mathcal{L}} C'; C_2'$. Using the unwinding in the characterization formula again and $\overrightarrow{V} \approx_{D'}^{\sim,\mathcal{L}} \overrightarrow{V}'$, we can conclude that $(C; C_2)\overrightarrow{V} \approx_{D'}^{\sim,\mathcal{L}} (C'; C_2')\overrightarrow{V}'$. Since the intermediary commands and the intermediary states fulfill the condition for $\approx_{D'}^{\sim,\mathcal{L}}$, $C_1; C_2 \approx_{D'}^{\sim,\mathcal{L}} C_1'; C_2'$ holds.

Forks:

$$\frac{C_1 \sim_D^{\mathcal{L}} C_2 \quad \overrightarrow{C}_1 \sim_D^{\mathcal{L}} \overrightarrow{C}_2}{\textbf{fork}(C_1 \overrightarrow{C}_1) \sim_D^{\mathcal{L}} \textbf{fork}(C_2 \overrightarrow{C}_2)}$$

According to the induction hypothesis we can deduce from the premise of the rule that $C_1 \approx_{D'}^{\rightsquigarrow,\mathcal{L}} C_2$ and $\overrightarrow{C}_1 \approx_{D'}^{\rightsquigarrow,\mathcal{L}} \overrightarrow{C}_2$. Note that we use $\overrightarrow{C}_1 \sim_D^{\mathcal{L}} \overrightarrow{C}_2$ to denote the point-wise application of $\sim_D^{\mathcal{L}}$ on the command vectors and the unwinding of command vectors in the characterization formula in the previous conclusion. According to the semantics for fork

$$\langle\!|\textbf{fork}(\ C_1\ \overrightarrow{C}_1\ ), s_1|\!\rangle \rightarrowtriangle_o \langle\!|C_1\ \overrightarrow{C_1}, s_1|\!\rangle$$

$$\langle\!|\textbf{fork}(\ C_2\ \overrightarrow{C}_2\ ), s_2|\!\rangle \rightarrowtriangle_o \langle\!|C_2\ \overrightarrow{C_2}, s_2|\!\rangle$$

Using the unwinding of command vectors in the characterization formula with $C_1 \approx_{D'}^{\rightsquigarrow,\mathcal{L}} C_2$ and $\overrightarrow{C}_1 \approx_{D'}^{\rightsquigarrow,\mathcal{L}} \overrightarrow{C}_2$, we can conclude that $C_1\overrightarrow{C}_1 \approx_{D'}^{\rightsquigarrow,\mathcal{L}} C_2\overrightarrow{C}_2$ and since the memory state remains untouched by the execution step $\textbf{fork}(\ C_1\ \overrightarrow{C}_1\ ) \approx_{D'}^{\rightsquigarrow,\mathcal{L}} \textbf{fork}(\ C_2\ \overrightarrow{C}_2\ )$.

Loops:

$$\frac{C_1 \sim_D^{\mathcal{L}} C_2}{\textbf{while}\ B\ \textbf{do}\ C_1\ \textbf{od} \sim_D^{\mathcal{L}} \textbf{while}\ B\ \textbf{do}\ C_2\ \textbf{od}}$$

According to the induction hypothesis we can deduce from the premise of the rule that $C_1 \approx_{D'}^{\rightsquigarrow,\mathcal{L}} C_2$. We must distinguish two cases. In the first case the guarding expression B depends only on information from domains that are lower than or equal to D' and thus B evaluates to equal values under all D'-equal states. In the second case, the expression depends on information from at least one domain that is neither lower than, nor equal to D' and thus B may evaluate to different values for some D'-equal states.

Case 1:

$$\forall s_1 =_{D'} s_1 : \langle\!|B, s_1|\!\rangle \downarrow m \wedge \langle\!|B, s_2|\!\rangle \downarrow n \wedge m = n$$

From the premise of this case, we know that B evaluates to equal values in all D'-equivalent states. Let $\langle\!|B, s_1|\!\rangle \downarrow True$, then $\langle\!|B, s_2|\!\rangle \downarrow True$, too. According to the semantics of **while**, this means

$$\langle\!|\textbf{while}\ B\ \textbf{do}\ C_1\ \textbf{od}, s_1|\!\rangle \rightarrowtriangle_o \langle\!|C_1; \textbf{while}\ B\ \textbf{do}\ C_1\ \textbf{od}, s_1|\!\rangle$$

$$\langle\!|\textbf{while}\ B\ \textbf{do}\ C_2\ \textbf{od}, s_2|\!\rangle \rightarrowtriangle_o \langle\!|C_2; \textbf{while}\ B\ \textbf{do}\ C_2\ \textbf{od}, s_2|\!\rangle$$

We can use the same argumentation as for sequences with the exception that $\sim_D^{\mathcal{L}}$ holds for the second command, which is the next iteration for unwinding the **while** loop and apply the rule for while again.

Let $\langle\!|B, s|\!\rangle \downarrow False$, then $\langle\!|B, s'|\!\rangle \downarrow False$, too. According to the semantics of **while**, this means

$$\langle\!|\textbf{while}\ B\ \textbf{do}\ C_1\ \textbf{od}, s_1|\!\rangle \rightarrowtriangle_o \langle\!|\langle\rangle, s_1|\!\rangle$$

$$\langle\!|\textbf{while}\ B\ \textbf{do}\ C_2\ \textbf{od}, s_2|\!\rangle \rightarrowtriangle_o \langle\!|\langle\rangle, s_2|\!\rangle$$

Since $s_1 =_{D'} s_2$ and $\langle\rangle \approx^{\rightsquigarrow,\mathcal{L}}_{D'} \langle\rangle$ hold, and the memory states are unchanged, which means that the information flow into the states is fulfilled,
while B **do** C_1 **od** $\approx^{\rightsquigarrow,\mathcal{L}}_{D'}$ **while** B **do** C_2 **od** holds.

Case 2:

$$\exists s_1 =_{D'} s_2 : \langle\!|B, s_1|\!\rangle \downarrow m \wedge \langle\!|B, s_2|\!\rangle \downarrow n \wedge m \neq n$$

Let s_1 and s_2 be the memory states that are D'-equal, but lead to different evaluations of B. Let $\langle\!|B, s_1|\!\rangle \downarrow \mathit{False}$, then $\langle\!|B, s_2|\!\rangle \downarrow \mathit{True}$ due to the premise of this case. According to the semantics,

$$\langle\!|\textbf{while } B \textbf{ do } C_1 \textbf{ od}, s_1|\!\rangle \rightarrow_o \langle\!|\langle\rangle, s_1|\!\rangle$$

$$\langle\!|\textbf{while } B \textbf{ do } C_1 \textbf{ od}, s_2|\!\rangle \rightarrow_o \langle\!|C_1; \textbf{while } B \textbf{ do } C_1 \textbf{ od}, s_2|\!\rangle$$

$\langle\rangle \approx^{\rightsquigarrow,\mathcal{L}}_{D'} C_1;$ **while** B **do** C_1 **od** does not hold, because both command vectors differ in the amount of threads. Since $C =$ **while** B **do** C_1 **od** must be WHERE&WHAT$_{\mathit{initial}}$-secure, this is a contradiction and thus the premise for this case can not be fulfilled for WHERE&WHAT$_{\mathit{initial}}$-secure commands. For $\langle\!|B, s_1|\!\rangle \downarrow \mathit{True}$ the argument is symmetric.

Branches:

$$\frac{C_1 \sim^{\mathcal{L}}_D C_3 \quad C_2 \sim^{\mathcal{L}}_D C_4}{\textbf{if } B \textbf{ then } C_1 \textbf{ else } C_2 \textbf{ fi} \sim^{\mathcal{L}}_D \textbf{if } B \textbf{ then } C_3 \textbf{ else } C_4 \textbf{ fi}}$$

According to the induction hypothesis we can deduce from the premise of the rule that $C_1 \approx^{\rightsquigarrow,\mathcal{L}}_{D'} C_3$ and $C_2 \approx^{\rightsquigarrow,\mathcal{L}}_{D'} C_4$. We must distinguish two cases similar to those for loops.

Case 1:

$$\forall s_1 =_{D'} s_1 : \langle\!|B, s_1|\!\rangle \downarrow m \wedge \langle\!|B, s_2|\!\rangle \downarrow n \wedge m = n$$

From the premise of this case, we know that B evaluates to equal values in all D'-equivalent states. Let $\langle\!|B, s_1|\!\rangle \downarrow \mathit{True}$, the $\langle\!|B, s_2|\!\rangle \downarrow \mathit{True}$, too. According to the semantics of **if**, this means

$$\langle\!|\textbf{if} B \textbf{then} C_1 \textbf{else} C_2 \textbf{fi}, s_1|\!\rangle \rightarrow_o \langle\!|C_1, s_1|\!\rangle$$

$$\langle\!|\textbf{if} B \textbf{then} C_3 \textbf{else} C_4 \textbf{fi}, s_2|\!\rangle \rightarrow_o \langle\!|C_3, s_2|\!\rangle$$

Since $C_1 \approx_{D'} C_3$ and $s_1 =_{D'} s_2$ hold,
if B **then** C_1 **else** C_2 **fi** $\approx^{\rightsquigarrow,\mathcal{L}}_{D'}$ **if** B **then** C_3 **else** C_4 **fi** holds. The argument for $\langle\!|B, s_1|\!\rangle \downarrow \mathit{False}$ is analog to this.

Case 2:

$$\exists s_1 =_{D'} s_2 : \langle\!\langle B, s_1 \rangle\!\rangle \downarrow m \wedge \langle\!\langle B, s_2 \rangle\!\rangle \downarrow n \wedge m \neq n$$

Let s_1 and s_2 be the memory states that are D'-equal, but lead to different evaluations of B. Let $\langle\!\langle B, s_1 \rangle\!\rangle \downarrow \mathit{False}$, then $\langle\!\langle B, s_2 \rangle\!\rangle \downarrow \mathit{True}$ due to the premise of this case. According to the semantics,

$$\langle\!\langle \textbf{if } B \textbf{ then } C_1 \textbf{ else } C_2 \textbf{ fi}, s_1 \rangle\!\rangle \rightarrow_o \langle\!\langle C_2, s_1 \rangle\!\rangle$$

$$\langle\!\langle \textbf{if } B \textbf{ then } C_1 \textbf{ else } C_2 \textbf{ fi}, s_2 \rangle\!\rangle \rightarrow_o \langle\!\langle C_1, s_2 \rangle\!\rangle$$

Since $C = \textbf{if } B \textbf{ then } C_1 \textbf{ else } C_2 \textbf{ fi}$ must be WHERE&WHAT$_{initial}$-secure and thus $C \approx^{\rightsquigarrow,\mathcal{L}}_{D} C$ must hold. We can conclude from the determinism of execution steps that $C_1 \approx^{\rightsquigarrow,\mathcal{L}}_{D'} C_2$. The same argument holds for $C_3 \approx^{\rightsquigarrow,\mathcal{L}}_{D'} C_4$.

Using the transitivity of $\approx^{\rightsquigarrow,\mathcal{L}}_{D'}$ we conclude that $C_1 \approx^{\rightsquigarrow,\mathcal{L}}_{D'} C_4$ and $C_2 \approx^{\rightsquigarrow,\mathcal{L}}_{D'} C_3$ from the conclusions that we have drawn from the premise of the rule with the help of the induction hypothesis.

Let now $s_1' =_{D'} s_2'$ be arbitrary. According to the semantics we know that

$$\langle\!\langle \textbf{if } B \textbf{ then } C_1 \textbf{ else } C_2 \textbf{ fi}, s_1' \rangle\!\rangle \rightarrow_o \langle\!\langle D_1, s_1' \rangle\!\rangle$$

$$\langle\!\langle \textbf{if } B \textbf{ then } C_3 \textbf{ else } C_4 \textbf{ fi}, s_2' \rangle\!\rangle \rightarrow_o \langle\!\langle D_2, s_2' \rangle\!\rangle$$

where $D_1 \in \{C_1, C_2\}$ and $D_2 \in \{C_3, C_4\}$. Since $C_1 \approx^{\rightsquigarrow,\mathcal{L}}_{D'} C_3$, $C_1 \approx^{\rightsquigarrow,\mathcal{L}}_{D'} C_4$, $C_2 \approx^{\rightsquigarrow,\mathcal{L}}_{D'} C_3$ and $C_2 \approx^{\rightsquigarrow,\mathcal{L}}_{D'} C_4$ hold, $D_1 \approx^{\rightsquigarrow,\mathcal{L}}_{D'} D_2$ holds for any $D_1 \in \{C_1, C_2\}$ and $D_2 \in \{C_3, C_4\}$. Since $s_1' =_{D'} s_2'$, the characterization formula is fulfilled and
$\textbf{if } B \textbf{ then } C_1 \textbf{ else } C_2 \textbf{ fi} \approx^{\rightsquigarrow,\mathcal{L}}_{D'} \textbf{if } B \textbf{ then } C_3 \textbf{ else } C_4 \textbf{ fi}$ holds.
For $\langle\!\langle B, s_1 \rangle\!\rangle \downarrow \mathit{True}$ the argument is symmetric.

■

With non D-visible equality with localized hatches $\mathcal{L}$ we have now a safe approximation relation that we can use and thus can solve the problem of the semantic side condition of **if** branchings. With this problem solved, we have now a type system that is a safe over approximation for WHERE&WHAT$_{initial}$-Security and thus we can automatically analyze programs with respect to this property. Since we already argued that the property captures our intuition so closely, we have now a sound analysis to check if a program captures our intuition of security. Since it is a safe approximation, the analysis may reject some programs that fulfill the property, but it will not accept any program, that does not fulfill the property.

5. Exemplary Analysis of Several Programs

We have now seen that the properties presented in this work capture our intuition very closely and that it is possible to automatically analyze programs with respect to the properties, but in order to further strengthen the confidence in the properties that we have presented in this work and show how those properties compare when analyzing program code that could occur in real world programs we want to analyze some code fragments and later in this section embed them into a small application scenario.

5.1. Explicit Assignments and Declassifications

The information flow that is introduced by an explicit assignment var := expr that is not executed in a loop or a branching command, which means it is not executed under a guard, is simply an information flow from $sources(expr)$ to $dom(var)$.
Let $pol1$ be the following mls-$(\rightsquigarrow, \mathcal{L})$ policy:
$\mathcal{D} = \{low, high\}$
$low \leq low$, $low \leq high$, $high \leq high$
$high \rightsquigarrow low$
$dom(lvar1) = dom(lvar2) = low$
$dom(hvar1) = dom(hvar2) = high$
$\mathcal{L}(1) = \{(low, hvar1)\}$
$\mathcal{L}(2) = \{(low, hvar2)\}$.

Let $\overrightarrow{C}_1$ be the following program:

```
lvar1 := hvar1
```

This program violates all properties discussed in this work, except WHAT-Security, because the $high$ information of $hvar1$ is assigned to the low variable $lvar1$ and $low \leq high$ and we can construct a counter example with $s_1 =_{low} s_2$, but $s_1(hvar1) = 0 \neq 1 = s_2(hvar1)$. After the execution of the assignment $s_1'(lvar1) = 0 \neq 1 = s_2'(lvar1)$ and in consequence $s_1' \neq_{low} s_2'$, which renders STRONG-Security unfulfilled. Since the execution step of the assignment is an ordinary step the restriction for **where** declassifications may occur can not be fulfilled and the program does not fulfill WHERE-Security, WHERE&WHAT$_{local}$-Security and WHERE&WHAT$_{initial}$-Security. On the other hand, the program fulfills WHAT-Security, because $\mathcal{H} = \{h | h \in \mathcal{D} \times Expr : \exists loc : h \in \mathcal{L}(loc)\}$ contains $(low, hvar1)$. If the assignment was an intended declassification and therefore the program should be considered secure, then there should be a declassification assignment with square brackets and an identifier that points to a set of escape hatches that contains a hatch that allows the declassification of $hvar1$ to low:

```
[ lvar1 := hvar1 ]1
```

This program fulfills all security properties in this work, except STRONG-Security which does not allow any declassification.

In the presence of branching commands and loops the requirement that the resulting thread pools can be related to each other in combination with the unwinding in the transformational semantics results in the preservation of indistinguishability of the states.

5.2. Subsequent Assignments for Input Handling

When dealing with user input subsequent assignments and declassifications are necessary to be able to fulfill the restrictions of the security policy or the functionality of the input. Subsequent declassifications are a problem, because an assignment to a variable can easily break the information flow into escape hatches.

Let *pol* be the following mls-$(\rightsquigarrow, \mathcal{L})$ policy:
$\mathcal{D} = \{low, high\}$
$low \leq low$, $low \leq high$, $high \leq high$
$high \rightsquigarrow low$
$dom(hvar) = dom(input) = high$
$dom(lvar) = low$
$\mathcal{L}(1) = \{(low, hvar)\}$

Let $\overrightarrow{C}$ be the following program:

```
hvar := hvar + input
[ lvar := hvar ]1
```

This program is intuitively insecure with respect to the security policy, because we declassify *hvar+input* at location 1, but the set of localized hatches allows only the declassification of *hvar*. This intuition is captured by WHAT-Security and WHERE&WHAT$_{initial}$-Security, but not by WHERE-Security, because this property does not have any control for **what** is declassified, and not by WHERE&WHAT$_{local}$-Security, because this property uses the state right before the declassification as implicit reference point and therefore has no control of information flow into escape hatches.

If the declassification of $hvar + input$ at location 1 is intentional, the security policy must allow this declassification explicitly. We can adapt the security policy and change only the set of localized hatches accordingly: $\mathcal{L}(1) = \{(low, hvar), (low, hvar + input)\}$. This program fulfills WHERE&WHAT$_{initial}$-Security and WHAT-Security, because a *low* observer is allowed to learn *hvar+input* and therefore can distinguish any two states where the evaluation of the expressions differs. In consequence, it is not possible to construct two states s_1, s_2 such that $s_1 =_{low}^{\mathcal{L}(1)} s_2$ and $s_1' \neq_{low}^{\mathcal{L}(1)}$ and therefore WHAT-Security and the restriction of information flow into the escape hatches in WHERE&WHAT$_{initial}$-Security are always fulfilled.

In the program model without input and output channels the input behavior can be modeled with variables in the memory states. The initial reference points of WHAT-Security and WHERE&WHAT$_{initial}$-Security require that distinguishability of memory states can not be altered by changes in the sets of localized hatches or escape hatches. This combines well with the model of input behavior with variables, because the input

is known before running the program and as a result can be used in the escape hatches to express that an observer might learn this information without special handling.

5.3. Declassification with Intransitive Security Policies

Subsequent assignments and declassifications can lead to an unintended information flow due to the transitivity of assignments. The subsequent assignments can be necessary, for example to require the declassification of information to be performed with the use of a specific declassification domain to be able to enforce a run time monitoring of the declassifications.

Let $pol1$ be the following mls-$(\rightsquigarrow, \mathcal{L})$ policy:
$\mathcal{D} = \{low, high, monitor\}$
$low \leq low$, $low \leq high$, $high \leq high$
$high \rightsquigarrow monitor \rightsquigarrow low$
$dom(hvar1) = dom(hvar2) = high$
$dom(lvar) = low$
$dom(mvar) = monitor$
$\mathcal{L}(1) = \{(low, hvar1)\}$.

Let $\overrightarrow{C}_1$ be the following program:
$[$ lvar := hvar1 $]_1$

This is intuitively insecure, because the security policy does not allow a direct declassification from security domain *high* to the security domain *low*. All properties in this work capture this intuition, except WHAT-Security which does not take $\rightsquigarrow$ into account.

The security policy requires a declassification to be made with the intermediate domain *monitor*, hence we need to declassify $hvar1$ to *monitor* and from there we can declassify the information to *low*. Furthermore, we need to change the set of escape hatches such that it captures our requirement of the intermediary assignment to a variable of the domain *monitor*.

Let $pol2$ be a policy equivalent to $pol1$, except:
$\mathcal{L}(1) = \{(monitor, hvar1)\}$
$\mathcal{L}(2) = \{(low, mvar)\}$.

Let $\overrightarrow{C}_2$ be the program:
$[$ mvar := hvar1 $]_1$
$[$ lvar1 := mvar $]_2$

This program is considered secure with respect to the policy $pol2$ and the properties WHERE-Security, and WHERE&WHAT$_{local}$-Security. Intuitively, we would assume this program not to be secure, because $hvar1$ may only get declassified to *monitor*, but in fact is declassified to *low* due to the subsequent declassifications. The properties WHAT-Security and WHERE&WHAT$_{initial}$-Security capture this intuition and we can construct a counter example by choosing $s_1 =_{low} s_2$ and $s_1 =_{monitor} s_2$, but $s_1(hvar1) = 0 \neq 1 = s_2(hvar1)$. Before executing the declassification in the first line $s_1 =_{low}^{\mathcal{L}(2)} s_2$

holds, but after executing the declassification $s_1'(mvar) = 0 \neq 1 = s_2'(mvar)$ and in consequence $s_1' =_{low}^{\mathcal{L}(2)} s_2'$ does not hold and $\forall loc \in Loc : (s_1 =_D^{\mathcal{L}(loc)} s_2 \Longrightarrow s_1' =_D^{\mathcal{L}(loc)} s_2')$ is not fulfilled.

Albeit this declassification is intended, it would be better if the exact information about the declassification is explicit in the policy. Let $pol3$ be equivalent to $pol2$, except:
$\mathcal{L}(1) = \{(monitor, hvar1)\}$
$\mathcal{L}(2) = \{(low, mvar), (low, hvar1)\}$.
With $pol3$ the program $\overrightarrow{C}_2$ is secure with respect to the WHERE&WHAT$_{local}$-Security. While the policy still does not allow the direct declassification of $high$ information to the domain low, the localized hatches $\mathcal{L}(2)$ now describe the transitive information flow from $hvar1$ via $mvar$ to the domain low. The counter example with $pol2$ does not work anymore, since $s_1 =_{low}^{\mathcal{L}(2)} s_2$ does not hold in the first place.

5.4. Localization of Different Declassifications

The main goal of the integration of the control of the aspects **where** and **what** is to enable an improved localization between both aspects in order to be able to determine **where** in the program **what** gets declassified. The intuition is that an observer is allowed to learn some additional information, but he is allowed to learn this information only at a specific location, for example he might be allowed to learn if his input equals the stored password, but he is only allowed to learn it by the means of the success of a login and not anywhere else in the program. Another possible application for the localization is when in a sequential program the declassification should not happen before or after a specific program point.

Let pol be the following mls-$(\rightsquigarrow, \mathcal{L})$ policy:
$\mathcal{D} = \{low, high\}$
$low \leq low$, $low \leq high$, $high \leq high$
$high \rightsquigarrow low$
$dom(lvar) = low$
$dom(hvar1) = dom(hvar2) = high$
$\mathcal{L}(1) = \{(low, hvar1 + hvar2)\}$
$\mathcal{L}(2) = \{(low, hvar1)\}$

Let $\overrightarrow{C}$ be the following program, where Seq1 is a sequence of commands that outputs $lvar$ and Seq1 is a sequence of commands that requires the knowledge of $hvar1$ in the domain low:

```
[ lvar := hvar1 + hvar2 ]1
Seq1
[ lvar := hvar1 ]2
Seq2
```

The security policy describes that $hvar1+hvar2$ may get declassified at location 1 and $hvar1$ at location 2. Intuitively, this program is secure and it fulfills WHERE-Security, WHAT-Security and WHERE&WHAT$_{initial}$-Security.

Assuming a programming error or the intention to leak $hvar1$, the author of the program could have written $[\ \text{lvar} := \text{hvar1}\ ]_1$ and declassify $hvar1$ in the first line instead of $hvar1 + hvar2$. This leak would not be revealed with an analysis that is sound with respect to WHERE-Security and WHAT-Security, because the integration of the aspects **where** and **what** is very loose and the analysis would only reveal if a declassification occurs outside of a declassification assignment or something gets declassified that is not allowed to be declassified at all. WHERE&WHAT$_{initial}$-Security captures this intuition and an analysis that is sound with respect to this property would reveal that at the declassification location 1 $hvar1$ gets declassified instead of $hvar1 + hvar2$.

With the exact localization of the declassified information in the security policies, the security policies do not only get more expressive, but the developers get more insight into the declassifications of the program and can use these insights to find problems in the software more easily.

5.5. Example Scenario: Development of an Online Market Place

Previously we have seen how small fragments of program code compare under the different security properties. In this section we want to show how the different properties compare when applied during software development. Since we think compositionality is more important than the monotonicity of release we will only look at WHAT$_1$-Security.

5.5.1. Data Structures in this Example

In order to model complexer data structures in our memory model we use a lookup table with the operation select(identifier , database) to retrieve the data associated with $identifier$ in the data structure stored in the variable *database* or False, if no data is associated with the identifier or the data stored in the variable is not a lookup table, and the operation update(database, identifier , data) to create a data structure that contains all associations from the variable *database*, but associates *data* with $identifier$ either by updating the association, if it exists, or adding the association to the table.

Furthermore, we assume an operation preview(digitalproduct) that can automatically calculate a preview of a digital product.

5.5.2. Initial Specification and Development

A software company wants to develop an online market place, where users can buy and sell digital goods. The initial specification requires that every user has a distinct login with a secret password and that the digital goods should be secret, except the user is the owner of the good or the user has paid the fee for this good. No user should be able to learn information about the other users.

According to this specification we can identify the requirement of at least two security domains and will use *high* and *low* and want to allow an information flow from *low* to *high*, but not vice versa. Furthermore, we require the *userdb*, the *productdb* and *product* to be *high*, since these are the secrets in the system. From the experience the developers

already have, they know that they must release the information about the equivalence of the password, as well as the product into the security domain *low*. Furthermore, they decide that the user information stored in an entry in *userdb* must not be kept secret, if it is the information of the logged in user.

They decide to use the following security policy:
$\mathcal{D} = \{low, high\}$
$low \leq low, low \leq high, high \leq high$
$high \rightsquigarrow low$
$dom(userdb) = dom(productdb) = dom(product) = high$
All other variables are low.
$\mathcal{L}(1) = \{(low, (select(\text{"}password\text{"}, select(username, userdb)) = inputpassword))\}$
$\mathcal{L}(2) = \{(low, select(username, userdb))\}$

The developers start by writing a login. The login requires a user to give his username as input and asks for the password for this user.

```
username := inputusername
[ correctpassword := ( select( "password",
                              select( username,
                                      userdb ) )
                       = inputpassword ) ]1
if ( correctpassword )
then
  [ activeuser := select( username, userdb ) ]2
  DispatcherForAuthenticatedActions
else
  activeuser := False
  username := False
  DispatcherForNonAuthenticatedActions
fi
```

Intuitively, this program should be considered secure with respect to the security policy. In fact, the program is secure with respect to WHERE-Security, WHAT_2-Security and $\text{WHERE\&WHAT}_{initial}$-Security. WHERE-Security and WHAT-Security use a different security policy with a set of all escape hatches $\mathcal{H}$ instead of a function of locations to escape hatches $\mathcal{L}$. In consequence, someone looking at the security alone would only see that the equivalence of the password
$(select(\text{"}password\text{"}, select(username, userdb)) = inputpassword))$ as well as all the user information associated with the name *username* $select(username, userdb)$ gets declassified in the case of WHERE-Security and WHAT_2-Security, whereas someone looking at the security policy with the localized hatches in combination with $\text{WHERE\&WHAT}_{initial}$-Security could see immediately that the equivalence of the password may get declassified at another location than the user information and that the user information can only be declassified, if the correct password has been entered, due

to the branching condition of the **if** command. Furthermore, observe that select(username, userdb) depends on *inputusername*, but it does not occur as an allowed declassification in $\mathcal{L}(1)$ or $\mathcal{L}(2)$ and still the program fulfills our security properties. The properties are fulfilled, because *username* and *inputusername* are both *low* and in consequence every observer can distinguish states that differ in those variables and the left side of the implications in the characterization formulas in each of the properties is not fulfilled which renders the whole formula fulfilled. The two commands DispatcherForAuthenticatedActions and DispatcherForNonAuthenticatedActions are placeholders for dispatchers that dispatch the single actions available for authenticated users and not authenticated users. Since they do not require declassifications, we omit the code for these dispatchers for simplicity.

In the next step the developers decide to write the code for browsing, buying and retrieving the digital goods. Remember that the product database *productdb* and the single products *product* should be a secret, unless the active user *activeuser* is the owner of the product or has paid the fee for the product. If the user is not logged in, he can just browse the previews of products, but should not be able to retrieve the full product or buy the product.

```
productid := inputproductid
product := select( productid, productdb )

[ isOwner := ( username = select( "owner",
                                  select( productid,
                                          productdb ) ) ) ]3

paidFee := ( not( select( productid,
                          select( "paidproducts",
                                  activeuser ) ) ) )

if ( isOwner or paidFee )
then
  [ output := product ]4
else
  [ output := select( "preview",
                      select( productid,
                              productdb ) )]5
  if ( not( username ) and  inputpurchase = "buy" )
  then
    paidproducts := select( "paidproducts", activeuser )
    paidproducts := update( paidproducts, productid, True )
    activeuser := update( activeuser,
                          "paidproducts",
                          paidproducts )
    userdb := update( userdb, username, activeuser )
```

```
    [ output := product ]6
  else
  fi
fi
```

This program is intuitively secure, because the program reveals the product only, if the active user is either the owner or has paid the fee for the product. If neither is the case, the user can only learn a preview of the product and only after buying the product may learn the whole product. The security properties WHERE-Security, WHAT_2-Security and $\text{WHERE\&WHAT}_{initial}$-Security capture this intuition, if we extend $\mathcal{L}$ to allow the new declassifications by adding
$\mathcal{L}(3) = \{(low, select(\text{"}owner\text{"}, select(productid, productdb)))\}$,
$\mathcal{L}(4) = \mathcal{L}(6) = \{(low, product), (low, select(productid, productdb))\}$ and
$\mathcal{L}(5) = \{(low, select(preview, select(productid, productdb)))\}$. It is important to notice that we intentionally used $select(\text{"}preview\text{"}, select(productid, productdb))$ at the declassification assignment 5, instead of $select(\text{"}preview\text{"}, product)$. If we used the latter, we hat to allow $select(productid, productdb)$ as declassification at 5 due to the transitive information flow and a look at the security policy alone would not reveal if only the preview or the whole product is declassified at this location anymore. This would limit the expressiveness of the security policy and a look at the program code would be necessary to reveal what is declassified at this location just like it is necessary with WHERE-Security and WHAT_2-Security. What we can learn from this is the fact that transitive declassifications, while they can be handled, still should be avoided, if possible, when it comes to declassification in order to take full advantage of the localization of declassifications.

The program fragment for registering new users and the fragment for adding new products do not require declassifications. We just present the code adding new products as an example.

```
product := inputproduct
productid := inputproductid

product := update( product, "owner", username )
productdb := update( productdb,
                     productid,
                     product )
```

This program is intuitively secure, since every information flow between two security domains is from the lower domain *low* to the higher domain *high*. This code fragment does not only fulfill WHERE-Security, WHAT_2-Security and $\text{WHERE\&WHAT}_{initial}$-Security, but also fulfills STRONG-Security. It is the only code fragment in that we presented that can be classified as secure with respect to STRONG-Security, because it is the only fragment that does not require any declassifications.

5.5.3. Adding a Reputation System

After the initial development, the company decides that the market place requires a reputation system, where customers can use the reputation of another customer in order to decide, if they want to buy a product from him and rate the other customer after purchasing the product. The reputation system should be very easy and just distinguish between the two judgments *positive* and *negative* and the average as well as the count of votes is released as the reputation. The developers immediately recognize the problem that users that have only a few sales could guess who gave them a *negative* judgment, if they have only few votes from other users, and fear that this could lead to a biased judgment, because the customers fear a bad vote in return. So they decide that the reputation is set to 0.5, if the customer has less than 10 judgments or the average of the judgments, if he has 10 or more.

In order to capture this additional information they introduce a new lookup table *reputationdb* that should be kept secret. This table associates user names with reputations. A reputation is a tuple consisting of the count of all judgments and the count of positive judgments. Initially both values are 0.

An adaption of the security policy is necessary to include the new variables *reputationdb* and *reputation* as secrets, which means they must be added to the domain assignment and must be in the domain *high*.

For the handling of the reputation system, we need to change the code for the user registration in order to capture the initial reputation and the code for browsing and purchasing products. We will omit the code for the user registration, since this code still does not require any declassification.

The new code for browsing and purchasing products has the additional requirement, that depending on the number of judgments for the user, either the reputation is fixed to 0.5 or it is the average of the judgments.

```
productid := inputproductid
product := select( productid, productdb )

[ owner := select( "owner",
                   select( productid,
                           productdb ) ) ]3

isOwner := ( username = owner )

paidFee := ( not( select( productid,
select( "paidproducts",
activeuser ) ) ) )

if ( isOwner or paidFee )
then
  [ output := product ]4
```

```
else
  [ output := select( "preview",
                      select( productid,
                              productdb ) )]5
  [ toFewJudgements := ( 10 > select( owner,
                                      reputationdb ) ) ]7

  if ( toFewJudgements )
  then
    output := 0.5
  else
    reputation := select( owner, reputationdb )
    [ output := select( "positive",
                        reputation )
                / select( "count",
                          reputation ) ]8

  if ( not( username ) and  inputpurchase = "buy" )
  then
    paidproducts := select( "paidproducts", activeuser )
    paidproducts := update( paidproducts, productid, True )
    activeuser := update( activeuser,
                          "paidproducts",
                          paidproducts )
    userdb := update( userdb, username, activeuser )
    [ output := product ]6
  fi
fi
```

In order to classify this program as secure with respect to the security properties WHERE-Security, WHAT_2-Security and $\text{WHERE\&WHAT}_{initial}$-Security, it is necessary to adapt the security policy to allow the additional declassifications 7 and 8, as well as adapt the changed declassification 3. The following security policy captures these new requirements:
$\mathcal{D} = \{low, high\}$
$low \leq low, low \leq high, high \leq high$
$high \rightsquigarrow low$
$dom(userdb) = high$
$dom(productdb) = dom(product) = high$
$dom(reputationdb) = dom(reputation) = high$
All other variables are low.
$\mathcal{L}(1) = \{(low, (select(\text{"}password\text{"}, select(username, userdb)) = inputpassword))\}$
$\mathcal{L}(2) = \{(low, select(username, userdb))\}$
$\mathcal{L}(3) = \{(low, select(\text{"}owner\text{"}, select(productid, productdb)))\}$

$\mathcal{L}(4) = \mathcal{L}(6) = \{(low, product), (low, select(productid, productdb))\}$
$\mathcal{L}(5) = \{(low, select(preview, select(productid, productdb)))\}$
$\mathcal{L}(7) = \{(low, (10 > select(owner, reputationdb)))\}$
$\mathcal{L}(8) = \{(low, select(\text{"}positive\text{"}, reputation)/select(\text{"}count\text{"}, reputation),$
$(low, select(owner, reputationdb)\}$.

This example shows the advantages of an exact localization of the declassification very well. Due to the amount of declassifications and the size of the set of all escape hatches, it is very cumbersome to determine what expressions are declassified at which location, because one must read the whole sequence of commands to track the assignments. The initial reference point used in WHERE&WHAT$_{initial}$-Security requires the localized hatches to capture these transitive information flows and the automated analyses supports the developer in specifying the localized hatches, because it reveals information flow into the set of escape hatches of single locations.

Additionally, a developer easily could have used a declassification of the form [$output := product$] at location 6 accidentally and reveal the product instead of the preview. With WHERE-Security and WHAT$_2$-Security this would not render the program insecure and the program would be classified as secure, even though it would not be secure with respect to our intuition. WHERE&WHAT$_{initial}$-Security captures this intuition and would reveal this false declassification due to the localized hatches.

This makes the mls-$(\rightsquigarrow, \mathcal{L})$ policies more expressive than mls-$(\rightsquigarrow, \mathcal{H})$ policies and therefore supports the developers better in finding information flow problems and understanding the declassifications occurring in a security policy.

On the other side, this example reveals two problems. It shows the problem with transitive information flow and the loss of accuracy it introduces we already mentioned. We could remedy this in two ways, either by not using the assignment to *owner* in the third line and instead explicitly requesting this information or we could use a different hatch set for location 8, where the second escape hatch resembles the first one, but *owner* is replaced with the expression of the assignment in the third line. The latter solution would still render the program secure with respect to the security properties, but it would be less close to the code and in consequence it would be harder to find the location that leads to the requirement of the second escape hatch, which is the assignment in line 3.

The other problem the example reveals is that the security policies require already require some restricted insight of the information flow in order to capture the transitive flows, but an automated analysis supports the development of this insight. Still, the mls-$(\rightsquigarrow, \mathcal{L})$ policies are more complex than mls-$(\rightsquigarrow, \mathcal{H})$ policies, due to the localized hatches, and the complexity of the function $\mathcal{L}$ increases with the amount of declassifications, but so does the amount of escape hatches in $\mathcal{H}$.

5.6. Benefits and Costs of the Localization of Declassifications

After the example code fragments and the example scenario, we want to reflect some benefits and oppose them with the costs the localization of the declassification have. As a baseline we use the combined property of WHERE-Security and WHAT-Security.

In order to handle transitive information flow due to subsequent assignments and declassifications, it is necessary to explicitly include those transitive flows in the sets of escape hatches. In the case of WHAT-Security, we must only insert additional escape hatches to the set of all escape hatches, for expressions in escape hatches that contain variables that occur on the left side of an assignment, while in the case of WHERE&WHAT$_{initial}$-Security, we must insert additional escape hatches to every location that allows a declassification of an expression that contains a variable that is on the left side of an assignment. On the other side, the insight on the information flow necessary to construct the hatch set is not made explicit in $\mathcal{H}$, because it only allows additional declassifications, whereas in the case of the localized hatches in $\mathcal{L}$ the information is made explicit in the form of declaring that additionally to the the expression on the right side, the information from evaluating the expressions in the localized hatches flows into the variable on the left side of the declassification, either directly or combined with other values or variables.

Adding a new declassification assignment is quite similar in both policies. When a new declassification assignment is added, the mls-$(\rightsquigarrow, \mathcal{H})$ policies must be changed by adding the necessary expressions for the declassification and possible transitive information flow to the set of all escape hatches $\mathcal{H}$, while in the case of the mls-$(\rightsquigarrow, \mathcal{L})$ policies a new location is introduced and the necessary expressions for the declassification and transitive information flows form the set of localized hatches associated with the new location.

While the mls-$(\rightsquigarrow, \mathcal{L})$ policies seem to be more complex than mls-$(\rightsquigarrow, \mathcal{H})$ policies, because escape hatches that may be used at several declassification assignments occur only once in $\mathcal{H}$, but for every occurrence of this expression at a declassification an escape hatch must be added to the set of localized hatches for this location and therefore occur more than once in $\mathcal{L}$. This is only a minor increase to the complexity of the policies, since locations that declassify the same expression may use equivalent sets of localized hatches and the real complexity lies in the determination of the correct escape hatches. Furthermore, the localized hatches again have the benefit that a look in the security policy reveals meaningful information about the escape hatches possibly used at a location in $\mathcal{L}$.

The control of information flow into the escape hatches in $WHAT-Security$ is done directly by the bisimulation of the preservation of (D, H)-equality with $H = \mathcal{H}$ during execution steps. It seems that the additional requirement that results in the restriction of information flow into escape hatches in WHERE&WHAT$_{initial}$-Security adds further complexity, but a comparison of the requirements in the type system reveals that the only additional complexity lies in the quantification over all declassification locations to capture the localized hatch sets.

All in all, the additional costs for the localization are basically restricted to the additional complexity of the mls-$(\rightsquigarrow, \mathcal{L})$ policies. On the other hand, the additional complexity of the policies is necessary in order to write down the localization information about the escape hatches we were interested in. Furthermore, the localized hatches reveal insights about the transitive information flows of the program the developer of the policy gained while analyzing the program. This information is a great support for either removing such transitive information flows were possible or showing the developers

which assignments must be handled with special care.

The result of this short overview is that the benefits of the localized hatches outweigh the additional costs, especially in the case where the analysis is used during development. In software development the additional information in the policies guides the developers in thinking about what information they really want to declassify and under which conditions the declassification should be allowed. This information could be of great use in combination with an analysis of control flow conditions. This combination could allow a reasoning that some information can only be declassified under certains conditions, for example that the **then** branch was used and therefore the entered password was correct.

6. Summary

6.1. Conclusion of the Results

We started this work with a short overview about STRONG-Security from [SS00] and showed that this property does not allow declassifications and in consequence is not suitable for all real world applications.

After that we took a detailed look at WHERE-Security and WHAT-Security from [MR07] and showed that these properties capture our intuition of security, or more specific confidentiality, closely even in the presence of declassifications. Furthermore, we showed that these properties are scheduler-independent with respect to a broad class of schedulers and to our knowledge were these the first scheduler-independence results for information flow properties with declassification. On the other hand, the separation of the aspect **where** and the aspect **what** into two individual properties prevented us from a more detailed view about **what** gets declassified **where** in the program.

We faced this problem with the introduction of WHERE&WHAT$_{local}$-Security, but showed that this property suffered from the problem of implicit, local reference points and therefore did not capture our intuition of security in the presence of subsequent assignments and declassifications. With WHERE&WHAT$_{initial}$-Security we presented a solution to this problem by restricting the information flow into the escape hatches. This novel property captures our intuition of confidentiality in the presence of declassification now very closely and allows a more detailed insight about **what** gets declassified **where** in the program. Furthermore, this property is still scheduler-independent with respect to the same class of schedulers as STRONG-Security, WHERE-Security and WHAT-Security. In consequence the property is suitable for application in multi-threaded settings. Additionally, we presented a type system for WHERE&WHAT$_{initial}$-Security and showed that the property is automatically enforceable.

In the last section we presented some example code fragments and showed how the different properties that allow declassification compare with respect to these. Furthermore, we presented a small application scenario in which we showed why the more detailed information about **what** gets **where** declassified in the program is useful, especially in the application of the analysis during software development. We have seen that the benefit of this properties lies in more detailed security policies that help to understand the information flow from declassifications easier and that the cost of a more complex policy is out weight by the benefit.

6.2. Future Work

While WHERE&WHAT$_{initial}$-Security can handle subsequent assignments and declassifications, a model for the input and output would still be desirable. As we have already mentioned, we think that the explicit reference points in [LM09a] are very promising and think that this approach could be adapted and integrated or combined with WHERE&WHAT$_{initial}$-Security.

Furthermore, the class of σ schedulers is very wide. It would be reasonable to look for

a smaller, but still practical class of schedulers, in order to weaken the strict bisimulation approach or find less strict safe approximation relations for the different **if** branches.

Another interesting topic which is completely untouched in this work is synchronization. The presence of synchronization has a huge impact on scheduling behavior and therefore it is only reasonable that in future work the impact of different synchronization primitives on WHERE&WHAT$_{initial}$-Security could be an interesting topic.

The security type system in this work is rather strict when it comes to the implicit information flow induced by the control flow of the program. We assume this to capture our intuition very closely, but a detailed look on the requirements for conditionals in branches, maybe even loops would give more insight, if a less strict approximation may be possible or desirable.

Another interesting topic for future research could be a combination of the localization of the declassifications with path conditions to capture the intuition of the developers that some information might only get declassified under special conditions which can be found implicitly in the control flow of branchins or loops.

Finally, the property was build using MWL, an exemplary toy language. It would be reasonable to compare this language with real world languages and to determine what other constructs may impose problems for an information flow analysis and to determine how we can handle those constructs.

6.3. Related Work

Information flow is a prominent and recent research topic as [SM03] suggests. Much research effort is put into controlling the declassifications in such scenarios [SS05].

The authors of [Smi07] give a short overview what language constructs may lead to an information flow and present type rules how to check a program for these information flows and by doing so give a good foundation for further work in the area of information flow analyses.

In [SS00] STRONG-Security uses bisimulations to formalize non-interference. Furthermore, this work introduces the σ schedulers and presents a proof that STRONG-Security is scheduler-independent. In this proof they use a scheduler-dependent property and probabilistic bisimulations to show even the scheduler can not leak information. In [MS04] the authors show how to adapt this approach for multi level security policies and presented an approach how to control declassification based on intransitive non-interference.

In [SM04] the authors introduce a control of the aspect **what**. They use a special declassification command that takes an expression and security domains as an argument. This command declassifies the expression to the given security domains and therefore allows the declassification of the expression. The intuition is very similar to the intuition of the escape hatches. Furthermore, they present a type system that is already sound with respect to another property from [AS07] that combines the **what** with the aspect **where**. This result is not too surprising, since the special declassification commands already capture the intuition of locality in the code. The language used in those works

does not have multi-threading and to our knowledge no results about their applicability in multi-threaded settings exist.

In *abstract non-interference* abstract interpretations, basically equivalence relations, are used to model the knowledge of the attacker. In [HM05] they review abstract non-interference by comparing it with the per model from [SS98]. Abstract non-interference differs from delimited release and our approach with the escape hatches, since they use relations to capture the knowledge of an attacker while the special command of delimited release and the escape hatches approximate this syntactically in order to make a syntactic analysis easier.

[MR07] introduces the properties WHERE-Security and WHAT-Security in order to control the aspect **where** and the aspect **what**. These properties use [SS00] and [MS04] as a foundation. As we have already seen in this work, those properties are scheduler-independent and therefore suitable for multi-threaded settings.

The authors of [LM09b] focus on the aspect **who** may initiate a declassification and integrate the control into WHERE-Security. In consequence, they achieve a control of the aspects **where** and **who**. Furthermore, the authors use input and output of a program, which is not present in the model in this work.

In [LM09a] the authors present an approach where reference points for distinguishable memory states are made explicit. While in our work the reference points where implicitly local in WHERE&WHAT$_{local}$-Security and implicitly initial in WHERE&WHAT$_{initial}$-Security, the security property presented in [LM09a] allows an explicit specification where in the program the reference points are set and which of the reference points are used.

As we have already mentioned a smaller set of schedulers could guide in the development of a weaker property that still captures our intuition of security. The authors of [MS10] present a class of schedulers they call robust schedulers that is smaller than the set of σ schedulers, but still very natural.

Other approaches exist that quantify the information flow. In [Low04] the authors present the idea to take the amount of the distinguishable program runs from indistinguishable states to classify how much information is leaked. The authors use a process algebra, but we assume that the result could be transferred to imperative programming languages.

References

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

[HM05] Sebastian Hunt and Isabella Mastroeni. The per model of abstract non-interference. In *Proc. of The 12th Internat. Static Analysis Symp. (SAS âĂŹ05), volume 3672 of Lecture Notes in Computer Science*, pages 171–185. Springer-Verlag, 2005.

[LM09a] Alexander Lux and Heiko Mantel. Declassification with explicit reference points. In Michael Backes and Peng Ning, editors, *14th European Symposium on Research in Computer Security*, volume 5789 of *LNCS*, pages 69–85. Springer, 2009.

[LM09b] Alexander Lux and Heiko Mantel. Who can declassify? In P. Degano, J. Guttman, and F. Martinelli, editors, *Proceedings of the Workshop on Formal Aspects in Security and Trust, FAST 2008*, volume 5491 of *LNCS*, pages 35–49. Springer, 2009.

[Low04] Gavin Lowe. Defining information flow quantity. *J. Comput. Secur.*, 12:619–653, May 2004.

[MR07] Heiko Mantel and Alexander Reinhard. Controlling the what and where of declassification in language-based security. In Rocco De Nicola, editor, *European Symposium on Programming (ESOP)*, volume 4421 of *LNCS*, pages 141–156. Springer, 2007.

[MS04] Heiko Mantel and David Sands. Controlled Declassification based on Intransitive Noninterference. In *Proceedings of the 2nd ASIAN Symposium on Programming Languages and Systems, APLAS 2004*, LNCS 3302, pages 129–145, Taipei, Taiwan, November 4–6 2004. Springer-Verlag.

[MS10] Heiko Mantel and Henning Sudbrock. Flexible scheduler-independent security. In *European Symposium on Research in Computer Security (ESORICS)*, LNCS 6345, pages 116–133. Springer, 2010.

[Rei06] Alexander Reinhard. Analyse nebenläufiger programme unter intransitiven sicherheitspolitiken. Master's thesis, RWTH Aachen, May 2006.

[SM03] Andrei Sabelfeld and Andrew C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21:2003, 2003.

[SM04] Andrei Sabelfeld and Andrew C. Myers. A model for delimited information release. In Kokichi Futatsugi, Fumio Mizoguchi, and Naoki Yonezaki, editors, *Software Security - Theories and Systems*, volume 3233 of *Lecture Notes in Computer Science*, pages 174–191. Springer Berlin / Heidelberg, 2004.

[Smi07] Geoffrey Smith. Principles of secure information flow analysis. In Mihai Christodorescu, Somesh Jha, Douglas Maughan, Dawn Song, and Cliff Wang, editors, *Malware Detection*, volume 27 of *Advances in Information Security*, pages 291–307. Springer US, 2007.

[SS98] Andrei Sabelfeld and David Sands. A per model of secure information flow in sequential programs. *HIGHER-ORDER AND SYMBOLIC COMPUTATION*, 14:40–58, 1998.

[SS00] Andrei Sabelfeld and David Sands. Probabilistic noninterference for multi-threaded programs. In *CSFW '00: Proceedings of the 13th IEEE workshop on Computer Security Foundations*, page 200, Washington, DC, USA, 2000. IEEE Computer Society.

[SS05] Andrei Sabelfeld and David Sands. Declassification: Dimensions and principles. In *In Proceedings of the 18th IEEE Workshop on Computer Security Foundations (CSFWâĂŹ05)*, pages 255–269, 2005.

A. Proposal

Department of Computer Science
Modeling and Analysis
of Information Systems
Prof. Dr.-Ing. Heiko Mantel

TECHNISCHE
UNIVERSITÄT
DARMSTADT

Master's Thesis

Controlled Declassification under Semantics with Schedulers

Contact: Alexander Lux <lux@mais.informatik.tu-darmstadt.de>
(Modeling and Analysis of Information Systems, S2/02 | [E 321])

Protecting the confidentiality of information is an important problem in modern networked information systems. A program might need confidential (*high*) data to perform its task, while it communicates seemingly uncritical (*low*) data (e.g. a registration process). The question is how to ensure that the program does not "leak" the confidential data, neither accidentally (bugs in the program) nor on purpose (a Trojan Horse).

An *Information Flow* analysis is a possible answer to the threat of leaking secrets. Its purpose is to check that there is no information leaking from high input to low output. Possible leaks can be explicit such as in statements like $l := h$ or, more subtly, implicit like in **if** $h = 1$ **then** $l := 1$ **else** $l := 0$, where one can draw conclusions on the (confidential) value of the high variable h just by observing the (non-secret) value of the low variable l.

Today, *noninterference*-like properties are a common approach to model the condition that there is no malicious information flow. These properties state that changing high input to a program does not lead to changes in the low output.

Many applications require some information about secrets to be released. For instance, the result of a password check has to be communicated, and this result necessarily contains some information about the secret password. Research on declassification addresses the question of how to relax noninterference-like properties as far as necessary for functionality, without giving up too much. Many approaches to control declassification have been developed [SS09], however, the problem is not yet satisfactorily solved. A layout to structure the research on declassification is given in [MS04] by the three *W-aspects*: *where* declassification can occur, *what* can be declassified, and *who* can initiate declassification.

In a multi-threaded setting, control of information flow is more complicated than for sequential programs, for instance because of implicit interactions between threads through the scheduler. *Strong security* [SS00] is one property that captures secure information flow in a multi-threaded setting. It implies a scheduler-dependent security property with any scheduler from a broad set of schedulers, i.e. it is *scheduler independent*. Scheduler independence is important, because the scheduler usually is not known before runtime.

A control of the aspects *where* and *what* in multi-threaded programs is offered in [MR07]. The security properties from [MR07] are formulated for a semantics that does

not explicitly model scheduling and are based on strong security with the intention that they are scheduler independent in a similar way. However, the scheduler independence of these properties has not been proved yet.

The purpose of this work is to provide a tightly integrated control of the two W-aspects of declassification *where* and *what* that is adequate in a multi-threaded setting with probabilistic schedulers.

Project Objectives

Core:

A: Scheduler-Independence with Probabilistic Schedulers The first objective is to strengthen the confidence in the security properties from [MR07] for applications in a setting with probabilistic schedulers by providing scheduler independence results. Such results are based on novel variants of the security properties for semantics with probabilistic schedulers and are established by a proof that the scheduler-dependent properties are implied by the properties from [MR07].

B: Integrating *what* and *where* Tightly The second objective is to advance the scope of controlled declassification by providing a tighter integration of the control of the aspects *where* and *what*. It shall be possible to specify where in the program what information can be released safely, and this shall be adequately reflected in a security property. Suitable application scenarios and programs shall be determined that guide the development. Scheduler-independence shall be established similar to objective A. A security type system shall provide a sound possibility to check programs against the novel security property.

C: Example Applications for Declassification The third objective is to demonstrate and evaluate the results from objectives A and B on suitable example applications. The applications shall cover an information flow policy that determines what may be declassified where in a multi-threaded program. The security property shall be proved by the novel security type system.

Extensions:

Additionally to the core objectives, the following objectives could be pursued.

- In [LM09], a more fine-grained approach for specifying *what* information may be declassified has been developed. It could be explored how this approach can be carried over to multi-threaded programs.
- Many multi-threaded programs use synchronization mechanisms. It could be explored how synchronization interacts with declassification.

- A common utilization of multi-threading is the separation of communication from calculation. It could be explored how to adequately support explicit input and output operations for communication with the program environment.
- In distributed programs communication partners of programs can be other programs., i.e. the output of one program can be the input of another one. Such a setting has been explored in [MS03, SM02], with a security property based on strong security. It could be explored how declassification can be treated in such a setting.

Main Activities

A includes

- defining variants of the security properties WHERE and WHAT_1 (optionally WHAT_2) [MR07] for probabilistic schedulers
- justifying the adequacy of the novel security properties
- proving scheduler-independence of WHERE, WHAT_1 (optionally WHAT_2)

B includes

- defining a possibility to specify *what can be declassified where*
- defining a scheduler-dependent security property
- defining a security property for that scheduler-independence is proved
- developing motivating example programs
- justifying the adequacy of the security properties, especially with respect to how they reflect the intention of specified declassification
- developing a suitable security type system and proving its soundness

C includes

- determining suitable applications
- developing programs that realize these applications
- determining suitable information flow policy
- demonstrating and evaluating results from objectives A and B against the example programs

Deliverables

The master's thesis shall include

- detailed presentation of security properties, type systems, and proofs as described in the prior sections
- detailed presentation of example programs and applications

- detailed explanation of decisions made (description of alternatives, discussion of their advantages and disadvantages, arguments for chosen solution, discussion in retrospective)
- detailed elaboration on insights gained, on problems identified, and on possible extensions in the future

After completing the thesis, a talk shall present the main results of the thesis.

Prerequisites

- basic knowledge of formal methods
- interest in information security

Supervision

Prof. Dr.-Ing. Heiko Mantel
Dipl.-Inform. Alexander Lux
(Modeling and Analysis of Information Systems)

References

[LM09] A. Lux and H. Mantel. Who Can Declassify? In *Proceedings of the Workshop on Formal Aspects in Security and Trust (FAST 2008)*, LNCS 5491, pages 35–49. Springer, 2009.

[MR07] H. Mantel and A. Reinhard. Controlling the What and Where of Declassification in Language-Based Security. In Rocco De Nicola, editor, *Proceedings of the 16th European Symposium on Programming (ESOP 2007)*, LNCS 4421, pages 141–156. Springer, 2007.

[MS03] H. Mantel and A. Sabelfeld. A Unifying Approach to the Security of Distributed and Multi-threaded Programs. *Journal of Computer Security*, 11(4):615–676, 2003.

[MS04] H. Mantel and D. Sands. Controlled Declassification based on Intransitive Noninterference. In *Proceedings of the 2nd ASIAN Symposium on Programming Languages and Systems (APLAS 2004)*, LNCS 3302, pages 129–145. Springer, 2004.

[SM02] A. Sabelfeld and H. Mantel. Static Confidentiality Enforcement for Distributed Programs. In *Proceedings of the 9th International Static Analysis Symposium (SAS 2002)*, LNCS 2477, pages 376–394, Madrid, Spain, 2002.

[SS00] A. Sabelfeld and D. Sands. Probabilistic Noninterference for Multi-threaded Programs. In *Proceedings of the 13th IEEE Computer Security Foundations Workshop (CSFW 2000)*, pages 200–215, 2000.

[SS09] A. Sabelfeld and D. Sands. Declassification: Dimensions and Principles. *Journal of Computer Security*, 17(5):517–548, 2009.