Eliminating Implicit Information Leaks by Transformational Typing and Unification

Boris Köpf[1] and Heiko Mantel[2]*

[1] Information Security, ETH Zürich, Switzerland, boris.koepf@inf.ethz.ch
[2] Department of Computer Science, RWTH Aachen University, Germany, mantel@cs.rwth-aachen.de

Abstract. Before starting the security analysis of an existing system, the most likely outcome is often already clear, namely that the system is not entirely secure. Modifying a program such that it passes the analysis is a difficult problem and usually left entirely to the programmer. In this article, we show that and how unification can be used to compute such program transformations. This opens a new perspective on the problem of correcting insecure programs. We demonstrate that integrating our approach into an existing transforming type system can also improve the precision of the analysis and the quality of the resulting programs.

1 Introduction

Security requirements like confidentiality or integrity can often be adequately expressed by restrictions on the permitted flow of information. This approach goes beyond access control models in that it controls not only the access to data, but also how data is propagated within a program after a legitimate access.

Security type systems provide a basis for automating the information flow analysis of concrete programs [SM03]. If type checking succeeds then a program has secure information flow. If type checking fails then the program might be insecure and should not be run. After a failed type check, the task of correcting the program is often left to the programmer. Given the significance of the problem, it would be very desirable to have automated tools that better support the programmer in this task. For the future, we envision a framework for the information flow analysis that, firstly, gives more constructive advice on how a given program could be improved and, secondly, in some cases automatically corrects the program, or parts thereof, without any need for interaction by the programmer. The current article focuses on the second of these two aspects.

Obviously, one cannot allow an automatic transformation to modify programs in completely arbitrary ways as the transformed program should resemble the original program in some well-defined way. Such constraints can be captured by defining an equivalence relation on programs and demanding that the transformed program is equivalent to the original program under this relation. A second equivalence relation can be used to capture the objective of a transformation. The problem of removing implicit information leaks from a program can

* The author gratefully acknowledges support by the DFG and the ETH Zürich.

In T. Dimitrakos, F. Martinelli, P. Y. Ryan, S. Schneider (Eds.): Revised Selected Papers, Formal Aspects in Security and Trust (FAST 2005), 3rd International Workshop, LNCS 3866, 47–62, 2006.

be viewed as the problem of making alternative execution paths observationally equivalent. For instance, if the guard of a conditional depends on a secret then the two branches must be observationally equivalent because, otherwise, an untrusted observer might be able to deduce the value of the guard and, thereby, the secret. The PER model [SS99] even reduces the problem of making an entire program secure to the problem of making the program equivalent to itself.

In our approach, meta-variables are inserted into a program and are instantiated with programs during the transformation. The problem of making two program fragments equivalent is cast as a unification problem, which allows us to automatically compute suitable substitutions using existing unification algorithms. The approach is parametric in two equivalence relations. The first relation captures the semantic equivalence to be preserved by the transformation while the second relation captures the observational equivalence to be achieved.

We define two concrete equivalence relations to instantiate our approach and integrate this instance into an existing transforming type system [SS00]. This results in a security type system that is capable of recognizing some secure programs and of correcting some insecure programs that are rejected by the original type system. Moreover, the resulting programs are faster and often substantially smaller in size. Another advantage over the cross-copying technique [Aga00], which constitutes the current state of the art in this area, is that security policies with more than two levels can be considered. Besides these technical advantages, the use of unification yields a very natural perspective on the problem of making two programs observationally equivalent. However, we do not claim that using unification will solve all problems with repairing insecure programs or that unification would be the only way to achieve the above technical advantages.

The contributions of this article are a novel approach to making the information flow in a given program secure and the demonstration that transforming security type systems can benefit from the integration of this approach.

2 The Approach

The observational capabilities of an attacker can be captured by an equivalence relation on configurations, i.e. pairs consisting of a program and a state. Namely, (C_1, s_1) is observationally equivalent to (C_2, s_2) for an attacker a if and only if the observations that a makes when C_1 is run in state s_1 equal a's observations when C_2 is run in s_2. The programs C_1 and C_2 are observationally equivalent for a if, for all states s_1 and s_2 that are indistinguishable for a, the configurations (C_1, s_1) and (C_2, s_2) are observationally equivalent for a. The resulting relation on programs is only a partial equivalence relation (PER), i.e. a transitive and symmetric relation that need not be reflexive. If a program C is not observationally equivalent to itself for a then running C in two indistinguishable states may lead to different observations and, thereby, reveal the differences between the states or, in other words, let a learn secret information. This observation is the key to capturing secure information flow in the PER model [SS99] in which a program is secure if and only if it is observationally equivalent to itself.

In this article, we focus on the removal of implicit information leaks from a program. There is a danger of implicit information leakage if the flow of control depends on a secret and the alternative execution paths are not observationally equivalent for an attacker. The program $\mathsf{if}\ h\ \mathsf{then}\ l{:=}1\ \mathsf{else}\ l{:=}0$, for instance, causes information to flow from the boolean guard h into the variable l, and this constitutes an illegitimate information leak if h stores a secret and the value of l is observable for the attacker. Information can also be leaked in a similar way, e.g., when the guard of a loop depends on a secret, when it depends on a secret whether an exception is raised, or when the target location of a jump depends on a secret. For brevity of the presentation, we focus on the case of conditionals.

We view the problem of making the branches of a conditional equivalent as a unification problem under a theory that captures observational equivalence. To this end, we insert meta-variables into the program under consideration that can be substituted during the transformation. For a given non-transforming security type system, the rule for conditionals is modified such that, instead of checking whether the branches are equivalent, the rule calculates a unifier of the branches and applies it to the conditional. Typing rules for other language constructs are lifted such that they propagate the transformations that have occurred in the analysis of the subprograms. In summary, our approach proceeds as follows:

1. Lift the given program by inserting meta-variables at suitable locations.
2. Repair the lifted program by applying lifted typing rules.
3. Eliminate all remaining meta-variables.

The approach is not only parametric in the given security type system and in the theory under which branches are unified, but also in where meta-variables are placed and how they may be substituted. The latter two parameters determine how similar a transformed program is to the original program. They also limit the extent to which insecure programs can be corrected. For instance, one might decide to insert meta-variables between every two sub-commands and to permit the substitution of meta-variables with arbitrary programs. For these choices, lifting $P_1 = \mathsf{if}\ h\ \mathsf{then}\ l{:=}1\ \mathsf{else}\ l{:=}0$ results in $\mathsf{if}\ h\ \mathsf{then}\ (\alpha_1; l{:=}1; \alpha_2)\ \mathsf{else}\ (\alpha_3; l{:=}0; \alpha_4)$ and the substitution $\{\alpha_1 \backslash l{:=}0,\ \ \alpha_2 \backslash \epsilon, \alpha_3 \backslash \epsilon,\ \ \alpha_4 \backslash l{:=}1\}$ (where ϵ is denotes the empty program) is a unifier of the branches under any equational theory as the substituted program is $\mathsf{if}\ h\ \mathsf{then}\ (l{:=}0; l{:=}1)\ \mathsf{else}\ (l{:=}0; l{:=}1)$. Alternatively, one might decide to restrict the range of substitutions to sequences of skip statements. This ensures that the transformed program more closely resembles the original program, essentially any transformed program is a slowed-down version of the original program, but makes it impossible to correct programs like P_1. However, the program $P_2 = \mathsf{if}\ h\ \mathsf{then}\ (\mathsf{skip}; l{:=}1)\ \mathsf{else}\ l{:=}1$, which is insecure in a multi-threaded setting (as we will explain later in this section), can be corrected under these choices to $\mathsf{if}\ h\ \mathsf{then}\ (\mathsf{skip}; l{:=}1)\ \mathsf{else}\ (\mathsf{skip}; l{:=}1)$. Alternatively, one could even decide to insert higher-order meta-variables such that lifting P_1 leads to $\mathsf{if}\ h\ \mathsf{then}\ \alpha_1(l{:=}1)\ \mathsf{else}\ \alpha_2(l{:=}0)$ and applying, e.g., the unifier $\{\alpha_1 \backslash (\lambda x.\mathsf{skip}),\ \ \alpha_2 \backslash (\lambda x.\mathsf{skip})\}$ results in $\mathsf{if}\ h\ \mathsf{then\ skip\ else\ skip}$ while applying the unifier $\{\alpha_1 \backslash (\lambda x.x),\ \ \alpha_2 \backslash (\lambda x.l{:=}1)\}$ results in $\mathsf{if}\ h\ \mathsf{then}\ l{:=}1\ \mathsf{else}\ l{:=}1$. These examples just illustrate the wide spectrum of possible choices for defining in which sense a transformed program must be equivalent to the original program. Ultimately it depends on the application, how flexible one is in dealing with the trade-off between being able to correct more insecure programs and having transformed programs that more closely resemble the original programs.

There also is a wide spectrum of possible choices for defining the (partial) observational equivalence relation. For simplicity, assume that variables are classified as either low or high depending on whether their values are observable by the attacker (low variables) or secret (high variables). As a convention, we denote low variables by l and high variables by h, possibly with indexes and primes. Given that the values of low variables are only observable at the end of a program run, the programs $P_3 = (\mathsf{skip}; l := 0)$ and $P_4 = (l := h; l := 0)$ are observationally equivalent and each is equivalent to itself (which means secure information flow in the PER model). However, if the attacker can observe also the intermediate values of low variables then they are not equivalent and, moreover, only P_3 is secure while P_4 is insecure. If the attacker can observe the timing of assignments or the duration of a program run then $P_2 = \mathsf{if}\ h\ \mathsf{then}\ (\mathsf{skip}; l{:=}1)\ \mathsf{else}\ l{:=}1$ is insecure and, hence, not observationally equivalent to itself. In a multi-threaded setting, P_2 should be considered insecure even if the attacker cannot observe the timing of assignments or the duration of a program run. If $P_3 = (\mathsf{skip}; l := 0)$ is run in parallel with P_2 under a shared memory and a round-robin scheduler that re-schedules after every sub-command then the final value of l is 0 and 1 if the initial value of h is 0 and 1, respectively. That is, a program that is observationally equivalent to itself in a sequential setting might not be observationally equivalent to itself in a multi-threaded setting – for the same attacker.

3 Instantiating the Approach

We are now ready to illustrate how our approach can be instantiated. We introduce a simple programming language, a security policy, an observational equivalence, and a program equivalence to be preserved under the transformation.

Programming Language. We adopt the multi-threaded while language (short: MWL) from [SS00], which includes assignments, conditionals, loops, and a command for dynamic thread creation. The set *Com* of commands is defined by

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

where V is a command vector in $\boldsymbol{Com} = \bigcup_{n\in\mathbb{N}} \mathit{Com}^n$. *Expressions* are variables, constants, or terms resulting from applying binary operators to expressions. A *state* is a mapping from variables in a given set *Var* to values in a given set *Val*. We use the judgment $\langle \mathit{Exp}, s\rangle \downarrow n$ for specifying that expression *Exp* evaluates to value n in state s. Expression evaluation is assumed to be total and to occur atomically. We say that expressions *Exp* and *Exp*$'$ are *equivalent* to each other (denoted by $\mathit{Exp}{\equiv}\mathit{Exp}'$) if and only if they evaluate to identical values in each state, i.e. $\forall s \in S : \forall v \in \mathit{Val} : \langle \mathit{Exp}, s\rangle \downarrow v \Leftrightarrow \langle \mathit{Exp}', s\rangle \downarrow v$.

The operational semantics for MWL is formalized in Figures 5 and 6 in the appendix. *Deterministic judgments* have the form $\langle C, s\rangle \twoheadrightarrow \langle W, t\rangle$ expressing that command C performs a computation step in state s, yielding a state t

and a vector of commands W, which has length zero if C terminated, length one if it has neither terminated nor spawned any threads, and length > 1 if threads were spawned. That is, a command vector of length n can be viewed as a *pool of n threads* that run concurrently. *Nondeterministic judgments* have the form $\langle\!| V, s |\!\rangle \rightarrowtriangle \langle\!| V', t |\!\rangle$ expressing that some thread C_i in the thread pool V performs a step in state s resulting in the state t and some thread pool W. The global thread pool V' results then by replacing C_i with W. For simplicity, we do not distinguish between commands and command vectors of length one in the notation and use the term *program* for referring to commands as well as to command vectors. A *configuration* is then a pair $\langle\!| V, s |\!\rangle$ where V specifies the threads that are currently active and s defines the current state of the memory.

In the following, we adopt the naming conventions used above. That is, s, t denote states, *Exp* denotes an expression, B denotes a boolean expression, C denotes a command, and V, W denote command vectors.

Security Policy and Labellings. We assume a two-domain security policy, where the requirement is that there is no flow of information from the *high domain* to the *low domain.* This is the simplest policy under which the problem of secure information flow can be studied. Each program variable is associated with a security domain by means of a *labeling* $lab : Var \to \{low, high\}$. The intuition is that values of *low* variables can be observed by the attacker and, hence, should only be used to store public data. *High variables* are used for storing secret data and, hence, their values must not be observable for the attacker. As mentioned before, we use l and h to denote high and low variables, respectively. An expression *Exp* has the security domain *low* (denoted by $Exp : low$) if all variables in *Exp* have domain *low* and, otherwise, has security domain *high* (denoted by $Exp : high$). The intuition is that values of expressions with domain *high* possibly depend on secrets while values of *low* expressions can only depend on public data.

Observational Equivalence. The rules in Figure 1 inductively define a relation $\bumpeq_L \subseteq \boldsymbol{Com} \times \boldsymbol{Com}$ that will serve us as an observational equivalence relation.

The relation $\bumpeq_L$ captures observational equivalence for an attacker who can see the values of low variables at any point during a program run and cannot distinguish states s_1 and s_2 if they are low equal (denoted by $s_1 =_L s_2$), i.e. if $\forall var \in Var : lab(var) = low \implies s_1(var) = s_2(var)$. He cannot distinguish two program runs that have equal length and in which every two corresponding states are low equal. For capturing this intuition, Sabelfeld and Sands introduce the notion of a strong low bisimulation. The relation $\bumpeq_L$ also captures this intuition and, moreover, programs that are related by $\bumpeq_L$ are also strongly bisimilar. That is, $\bumpeq_L$ is a decidable approximation of the strong bisimulation relation.

Definition 1 ([SS00]). *The* strong low-bisimulation $\cong_L$ *is the union of all symmetric relations R on command vectors $V, V' \in \boldsymbol{Com}$ of equal size, i.e. $V = \langle C_1, \ldots, C_n \rangle$ and $V' = \langle C'_1, \ldots, C'_n \rangle$, such that*

$$\begin{array}{l} \forall s, s', t \in S : \forall i \in \{1 \ldots n\} : \forall W \in \boldsymbol{Com}: \\ \;[(V \, R \, V' \wedge s =_L s' \wedge \langle\!| C_i, s |\!\rangle \rightarrowtriangle \langle\!| W, t |\!\rangle) \\ \;\;\Rightarrow \exists W' \in \boldsymbol{Com}: \exists t' \in S: (\langle\!| C'_i, s' |\!\rangle \rightarrowtriangle \langle\!| W', t' |\!\rangle \wedge W \, R \, W' \wedge t =_L t')] \end{array}$$

$$\frac{}{\mathsf{skip} \bumpeq_L \mathsf{skip}}\;[\mathit{Skip}] \qquad \frac{\mathit{Id} : \mathit{high}}{\mathsf{skip} \bumpeq_L \mathit{Id}{:=}\mathit{Exp}}\;[\mathit{SHA_1}] \qquad \frac{\mathit{Id} : \mathit{high}}{\mathit{Id}{:=}\mathit{Exp} \bumpeq_L \mathsf{skip}}\;[\mathit{SHA_2}]$$

$$\frac{\mathit{Id} : \mathit{high} \quad \mathit{Id}' : \mathit{high}}{\mathit{Id}{:=}\mathit{Exp} \bumpeq_L \mathit{Id}'{:=}\mathit{Exp}'}\;[\mathit{HA}] \qquad \frac{\mathit{Id} : \mathit{low} \quad \mathit{Exp} : \mathit{low} \quad \mathit{Exp}' : \mathit{low} \quad \mathit{Exp} \equiv \mathit{Exp}'}{\mathit{Id}{:=}\mathit{Exp} \bumpeq_L \mathit{Id}{:=}\mathit{Exp}'}\;[\mathit{LA}]$$

$$\frac{C_1 \bumpeq_L C_1', \ldots, C_n \bumpeq_L C_n'}{\langle C_1, \ldots, C_n \rangle \bumpeq_L \langle C_1', \ldots, C_n' \rangle}\;[\mathit{PComp}] \qquad \frac{C \bumpeq_L C' \quad V \bumpeq_L V'}{\mathsf{fork}(CV) \bumpeq_L \mathsf{fork}(C'V')}\;[\mathit{Fork}]$$

$$\frac{B, B' : \mathit{low} \quad B \equiv B' \quad C_1 \bumpeq_L C_1' \quad C_2 \bumpeq_L C_2'}{\mathsf{if}\ B\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2 \bumpeq_L \mathsf{if}\ B'\ \mathsf{then}\ C_1'\ \mathsf{else}\ C_2'}\;[\mathit{LIte}] \qquad \frac{B, B' : \mathit{low} \quad B \equiv B' \quad C \bumpeq_L C'}{\mathsf{while}\ B\ \mathsf{do}\ C \bumpeq_L \mathsf{while}\ B'\ \mathsf{do}\ C'}\;[\mathit{WL}]$$

$$\frac{B, B' : \mathit{high} \quad C_1 \bumpeq_L C_1' \quad C_1 \bumpeq_L C_2' \quad C_1 \bumpeq_L C_2}{\mathsf{if}\ B\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2 \bumpeq_L \mathsf{if}\ B'\ \mathsf{then}\ C_1'\ \mathsf{else}\ C_2'}\;[\mathit{HIte}] \qquad \frac{C_1 \bumpeq_L C_1' \quad C_2 \bumpeq_L C_2'}{C_1; C_2 \bumpeq_L C_1'; C_2'}\;[\mathit{SComp}]$$

$$\frac{B' : \mathit{high} \quad C_1 \bumpeq_L C_1' \quad C_1 \bumpeq_L C_2'}{\mathsf{skip}; C_1 \bumpeq_L \mathsf{if}\ B'\ \mathsf{then}\ C_1'\ \mathsf{else}\ C_2'}\;[\mathit{SHIte_1}] \qquad \frac{\mathit{Id}, B' : \mathit{high} \quad C_1 \bumpeq_L C_1' \quad C_1 \bumpeq_L C_2'}{\mathit{Id}{:=}\mathit{Exp}; C_1 \bumpeq_L \mathsf{if}\, B'\ \mathsf{then}\ C_1'\ \mathsf{else}\ C_2'}\;[\mathit{HAHIte_1}]$$

$$\frac{B : \mathit{high} \quad C_1 \bumpeq_L C_1' \quad C_2 \bumpeq_L C_1'}{\mathsf{if}\ B\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2 \bumpeq_L \mathsf{skip}; C_1'}\;[\mathit{SHIte_2}] \qquad \frac{\mathit{Id}', B : \mathit{high} \quad C_1 \bumpeq_L C_1' \quad C_2 \bumpeq_L C_1'}{\mathsf{if}\, B\,\mathsf{then}\ C_1\ \mathsf{else}\ C_2 \bumpeq_L \mathit{Id}'{:=}\mathit{Exp}'; C_1'}\;[\mathit{HAHIte_2}]$$

Fig. 1. A notion of observational equivalence

Theorem 1 (Adequacy of $\bumpeq_L$). *If $V \bumpeq_L V'$ is derivable then $V \approxeq_L V'$ holds.*

The proofs of this and all subsequent results are provided in an accompanying Technical Report [KM05].

Remark 1. Note that $\bumpeq_L$ and $\approxeq_L$ are only partial equivalence relations, i.e. they are transitive and symmetric, but not reflexive. For instance, the program $l{:=}h$ is not $\bumpeq_L$-related to itself because the precondition of $[LA]$, the only rule in Figure 1 applicable to assignments to low variables, rules out that high variables occur on the right hand side of the assignment. Moreover, the program $l{:=}h$ is not strongly low bisimilar to itself because the states s and t (defined by $s(l) = 0$, $s(h) = 0$, $t(l) = 0$, $t(h) = 1$) are low equal, but the states s' and t' resulting after $l{:=}h$ is run in s and t, respectively, are not low equal ($s'(l) = 0 \neq 1 = t'(l)$).

However, $\bumpeq_L$ is an equivalence relation if one restricts programs to the language *Slice* that we define as the largest sub-language of *Com* without assignments of high expressions to low variables, assignments to high variables, and loops or conditionals having high guards. On *Slice*, $\bumpeq_L$ even constitutes a congruence relation. This sub-language is the context in which we will apply unification and, hence, using the term *unification under an equational theory* is justified. $\Diamond$

Program Equivalence. We introduce an equivalence relation $\simeq$ to constrain the modifications caused by the transformation. Intuitively, this relation requires a transformed program to be a slowed down version of the original program. This is stronger than the constraint in [SS00].

Definition 2. *The* weak possibilistic bisimulation $\simeq$ *is the union of all symmetric relations R on command vectors such that whenever $V \; R \; V'$ then for all states s, t and all vectors W there is a vector W' such that*

$$\begin{aligned} &\langle\!| V, s |\!\rangle \twoheadrightarrow \langle\!| W, t |\!\rangle \implies (\langle\!| V', s |\!\rangle \twoheadrightarrow^* \langle\!| W', t |\!\rangle \wedge W R W') \\ &\textit{and } V = \langle\rangle \implies \langle\!| V', s |\!\rangle \twoheadrightarrow^* \langle\!| \langle\rangle, s |\!\rangle \; . \end{aligned}$$

4 Lifting a Security Type System

In this section we introduce a formal framework for transforming programs by inserting and instantiating meta-variables. Rather than developing an entirely new formalism from scratch, we adapt an existing security type system from [SS00]. We show that any transformation within our framework is sound in the sense that the output is secure and the behavior of the original program is preserved in the sense of Definition 2.

Substitutions and Liftings. We insert meta-variables from a set $\mathcal{V} = \{\alpha_1, \alpha_2, \dots\}$ into a program by sequential composition with its sub-terms. The extension of MWL with meta-variables is denoted by $\text{MWL}_{\mathcal{V}}$. The set $\mathit{Com}_{\mathcal{V}}$ of commands in $\text{MWL}_{\mathcal{V}}$ is defined by[3]

$$\begin{aligned} C ::= {} & \mathsf{skip} \mid \mathit{Id}{:=}\mathit{Exp} \mid C_1; C_2 \mid C; X \mid X; C \\ & \mathsf{if}\ B\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2 \mid \mathsf{while}\ B\ \mathsf{do}\ C \mid \mathsf{fork}(CV) \; , \end{aligned}$$

where placeholders X, Y range over $\mathcal{V}$. Analogously to MWL, the set of all command vectors in $\text{MWL}_{\mathcal{V}}$ is defined by $\boldsymbol{Com_{\mathcal{V}}} = \bigcup_{n \in \mathbb{N}} (\mathit{Com}_{\mathcal{V}})^n$. Note that the *ground programs* in $\text{MWL}_{\mathcal{V}}$ are exactly the programs in MWL. The operational semantics for such programs remain unchanged, whereas programs with meta-variables are not meant to be executed.

Meta-variables may be substituted with programs, meta-variables or the special symbol ϵ that acts as the neutral element of the sequential composition operator (";"), i.e. $\epsilon; C = C$ and $C; \epsilon = C$[4]. When talking about programs in $\mathit{Com}_{\mathcal{V}}$ under a given substitution, we implicitly assume that these equations have been applied (from left to right) to eliminate the symbol ϵ from the program. Moreover, we view sequential composition as an associative operator and implicitly identify programs that differ only in the use of parentheses for sequential composition. That is, $C_1; (C_2; C_3)$ and $(C_1; C_2); C_3$ denote the same program.

A mapping $\sigma : \mathcal{V} \to (\{\epsilon\} \cup \mathcal{V} \cup \mathit{Com}_{\mathcal{V}})$ is a *substitution* if the set $\{\alpha \in \mathcal{V} \mid \sigma(\alpha) \neq \alpha\}$ is finite. A substitution mapping each meta-variable in a program V to $\{\epsilon\} \cup \mathit{Com}$ is a *ground substitution of* V. A substitution π mapping all meta-variables in V to ϵ is a *projection of* V. Given a program V in $\boldsymbol{Com}$, we call every program V' in $\boldsymbol{Com_{\mathcal{V}}}$ with $\pi V' = V$ a *lifting of* V.

For example, the program $\mathsf{if}\ h\ \mathsf{then}\ (\alpha_1; \mathsf{skip}; \alpha_2; l{:=}1)\ \mathsf{else}\ (\alpha_3; l{:=}1)$ is in fact a lifting of $\mathsf{if}\ h\ \mathsf{then}\ (\mathsf{skip}; l{:=}1)\ \mathsf{else}\ l{:=}1$. In the remainder of this article, we will focus on substitutions with a restricted range.

[3] Here and in the following, we overload notation by using C and V to denote commands and command vectors in $\mathit{Com}_{\mathcal{V}}$, respectively.

[4] Note that skip is not a neutral element of (";") as skip requires a computation step.

Definition 3. *A substitution with range* $\{\epsilon\} \cup \mathit{Stut}_{\mathcal{V}}$ *is called* preserving*, where* $\mathit{Stut}_{\mathcal{V}}$ *is defined by* $C ::= X \mid \mathsf{skip} \mid C_1; C_2$ *(the* C_i *range over* $\mathit{Stut}_{\mathcal{V}}$*).*

The term *preserving* substitution is justified by the fact that such substitutions preserve a given program's semantics as specified in Definition 2.

Theorem 2 (Preservation of Behavior).

1. *Let* $V \in \boldsymbol{Com}_{\mathcal{V}}$*. For all preserving substitutions* σ, ρ *that are ground for* V*, we have* $\sigma(V) \simeq \rho(V)$*.*
2. *Let* $V \in \boldsymbol{Com}$*. For each lifting* V' *of* V *and each preserving substitution* σ *with* $\sigma(V')$ *ground, we have* $\sigma(V') \simeq V$*.*

Unification of Programs. The problem of finding a substitution that makes the branches of conditionals with high guards observationally equivalent can be viewed as the problem of finding a unifier for the branches under the equational theory $\bumpeq_L$.[5] To this end, we lift the relation $\bumpeq_L \subseteq \boldsymbol{Com} \times \boldsymbol{Com}$ to a binary relation on $\boldsymbol{Com}_{\mathcal{V}}$ that we also denote by $\bumpeq_L$.

Definition 4. $V_1, V_2 \in \boldsymbol{Com}_{\mathcal{V}}$ *are* observationally equivalent *(*$V_1 \bumpeq_L V_2$*) iff* $\sigma V_1 \bumpeq_L \sigma V_2$ *for each preserving substitution* σ *that is ground for* V_1 *and* V_2*.*

Definition 5. *A* $\bumpeq_L$-unification problem Δ *is a finite set of statements of the form* $V_i \bumpeq^?_L V'_i$*, i.e.* $\Delta = \{V_0 \bumpeq^?_L V'_0, \ldots, V_n \bumpeq^?_L V'_n\}$ *with* $V_i, V'_i \in \boldsymbol{Com}_{\mathcal{V}}$ *for all* $i \in \{0, \ldots, n\}$*. A substitution* σ *is a* preserving unifier for Δ *if and only if* σ *is preserving and* $\sigma V_i \bumpeq_L \sigma V'_i$ *holds for each* $i \in \{0, \ldots, n\}$*. A* $\bumpeq_L$*-unification problem is* solvable *if the set of preserving unifiers* $\mathcal{U}(\Delta)$ *for* Δ *is not empty.*

A Transforming Type System. The transforming type system in Figure 2 has been derived from the one in [SS00]. We use the judgment $V \hookrightarrow V' : S$ for denoting that the MWL$_{\mathcal{V}}$-program V can be transformed into an MWL$_{\mathcal{V}}$-program V'. The intention is that V' has secure information flow and reflects the semantics of V as specified by Definition 2. The *slice* S is a program that is in the sub-language $\mathit{Slice}_{\mathcal{V}}$ and describes the timing behavior of V'. The novelty over [SS00] is that our type system operates on $\boldsymbol{Com}_{\mathcal{V}}$ (rather than on $\boldsymbol{Com}$) and that the rule for high conditionals has been altered. In the original type system, a high conditional is transformed by sequentially composing each branch with the slice of the respective other branch. Instead of cross-copying slices, our rule instantiates the meta-variables occurring in the branches using preserving unifiers. The advantages of this modification are discussed in Section 6. Note that the rule $[\mathit{Cond}_h]$ does not mandate the choice of a specific preserving unifier of the branches. Nevertheless, we can prove that the type system meets our previously described intuition about the judgment $V \hookrightarrow V' : S$. To this end, we employ Sabelfeld and Sands's strong security condition for defining what it means for a program to have secure information flow. Many other definitions are possible (see e.g. [SM03]).

Definition 6. *A program* $V \in \boldsymbol{Com}$ is strongly secure *if and only if* $V \approxeq_L V$ *holds. A program* $V \in \boldsymbol{Com}_{\mathcal{V}}$ is strongly secure *if and only if* σV *is strongly secure for each substitution* σ *that is preserving and ground for* V*.*

$$\frac{}{\mathsf{skip} \hookrightarrow \mathsf{skip} : \mathsf{skip}}\ [Skp] \qquad \frac{Id : high}{Id{:=}Exp \hookrightarrow Id{:=}Exp : \mathsf{skip}}\ [Ass_h] \qquad \frac{C_1 \hookrightarrow C_1' : S_1 \quad C_2 \hookrightarrow C_2' : S_2}{C_1; C_2 \hookrightarrow C_1'; C_2' : S_1; S_2}\ [Seq]$$

$$\frac{Id : low \qquad Exp : low}{Id{:=}Exp \hookrightarrow Id{:=}Exp : Id{:=}Exp}\ [Ass_l] \qquad \frac{B : low \qquad C \hookrightarrow C' : S}{\mathsf{while}\ B\ \mathsf{do}\ C \hookrightarrow \mathsf{while}\ B\ \mathsf{do}\ C' : \mathsf{while}\ B\ \mathsf{do}\ S}\ [Whl]$$

$$\frac{C_1 \hookrightarrow C_1' : S_1 \quad \dots \quad C_n \hookrightarrow C_n' : S_n}{\langle C_1, \dots, C_n \rangle \hookrightarrow \langle C_1', \dots, C_n' \rangle : \langle S_1, \dots, S_n \rangle}\ [Par] \qquad \frac{C_1 \hookrightarrow C_1' : S_1 \qquad V_2 \hookrightarrow V_2' : S_2}{\mathsf{fork}(C_1 V_2) \hookrightarrow \mathsf{fork}(C_1' V_2') : \mathsf{fork}(S_1 S_2)}\ [Frk]$$

$$\frac{B : low \qquad C_1 \hookrightarrow C_1' : S_1 \qquad C_2 \hookrightarrow C_2' : S_2}{\mathsf{if}\ B\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2 \hookrightarrow \mathsf{if}\ B\ \mathsf{then}\ C_1'\ \mathsf{else}\ C_2' : \mathsf{if}\ B\ \mathsf{then}\ S_1\ \mathsf{else}\ S_2}\ [Cond_l]$$

$$\frac{B : high \quad C_1 \hookrightarrow C_1' : S_1 \quad C_2 \hookrightarrow C_2' : S_2 \quad \sigma \in \mathcal{U}(\{S_1 \overset{?}{\hat{=}}_L S_2\})}{\mathsf{if}\ B\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2 \hookrightarrow \mathsf{if}\ B\ \mathsf{then}\ \sigma C_1'\ \mathsf{else}\ \sigma C_2' : \mathsf{skip}; \sigma S_1}\ [Cond_h] \qquad \frac{}{X \hookrightarrow X : X}\ [Var]$$

Fig. 2. A transforming security type system for programs with meta-variables

Theorem 3 (Soundness Type System). *If $V \hookrightarrow V' : S$ can be derived then (1) V' has secure information flow, (2) $V \simeq V'$ holds,*[6] *and (3) $V' \approxeq_L S$ holds.*

The following corollary is an immediate consequence of Theorems 2 and 3. It shows that lifting a program and then applying the transforming type system preserves a program's behavior in the desired way.

Corollary 1. *If $V^* \hookrightarrow V' : S$ is derivable for some lifting $V^* \in \boldsymbol{Com}_{\mathcal{V}}$ of a program $V \in \boldsymbol{Com}$ then V' has secure information flow and $V \simeq V'$.*

5 Automating the Transformation

In Section 4, we have shown our type system to be sound for any choice of liftings and preserving unifiers in the applications of rule $[Cond_h]$. For automating the transformation, we have to define more concretely where meta-variables are inserted and how unifiers are determined.

Automatic Insertion of Meta-Variables. When lifting a program, one is faced with a trade off: inserting meta-variables means to create possibilities for correcting the program, but it also increases the complexity of the unification problem. Within this spectrum our objective is to minimize the number of inserted meta-variables without losing the possibility of correcting the program.

To this end, observe that two programs C_1 and C_2 within the sub-language $Pad_{\mathcal{V}}$, the extension of $Stut_{\mathcal{V}}$ with high assignments, are related via $\hat{=}_L$ whenever they contain the same number of constants, i.e., skips and assignments to high variables (denoted as $const(C_1) = const(C_2)$), and the same number of occurrences of each meta-variable α (denoted by $|C_1|_\alpha = |C_2|_\alpha$). Note that the positioning of meta-variables is irrelevant.

[5] The term *equational theory* is justified as we apply unification only to programs in the sub-language $Slice_{\mathcal{V}}$ for which $\hat{=}_L$ constitutes a congruence relation (see Remark 1).

[6] Here and in the following, we define $\simeq$ on $Com_{\mathcal{V}}$ by $C \simeq C'$ iff $\sigma C \simeq \sigma C'$ for any substitution σ that is preserving and ground for C and for C'.

$$\frac{Id : high \quad X \text{ fresh}}{Id{:=}Exp \rightharpoonup Id{:=}Exp; X} \qquad \frac{C_1 \rightharpoonup C_1' \quad V_2 \rightharpoonup V_2' \quad X, Y \text{ fresh}}{\mathsf{fork}(C_1 V_2) \rightharpoonup X; (\mathsf{fork}(C_1' V_2')); Y} \qquad \frac{C_1 \rightharpoonup C_1'; X \quad C_2 \rightharpoonup C_2'}{C_1; C_2 \rightharpoonup C_1'; C_2'}$$

Fig. 3. A calculus for computing most general liftings

Lemma 1. *For two commands C_1 and C_2 in $Pad_{\mathcal{V}}$ we have $C_1 \bumpeq_L C_2$ if and only if $const(C_1) = const(C_2)$ and $\forall \alpha \in \mathcal{V}\colon |C_1|_\alpha = |C_2|_\alpha$.*

Moreover, observe that inserting one meta-variable next to another does not create new possibilities for correcting a program. This, together with Lemma 1, implies that inserting one meta-variable into every subprogram within $Pad_{\mathcal{V}}$ is sufficient for allowing every possible correction. We use this insight to define a mapping $\rightharpoonup\colon \boldsymbol{Com} \to \boldsymbol{Com_{\mathcal{V}}}$ that calculates a lifting of a program by inserting one fresh meta-variable at the end of every sub-program in $Pad_{\mathcal{V}}$, and between every two sub-programs outside $Pad_{\mathcal{V}}$. The mapping is defined inductively: A fresh meta-variable is sequentially composed to the right hand side of each subprogram. Another fresh meta-variable is sequentially composed to the left hand side of each assignment to a low variable, fork, while loop, or conditional. A lifting of a sequentially composed program is computed by sequentially composing the liftings of the subprograms while removing the terminal variable of the left program. The three interesting cases are illustrated in Figure 3. The liftings computed by $\rightharpoonup$ are *most general* in the sense that if two programs can be made observationally equivalent for some lifting then they can be made equivalent for the lifting computed by $\rightharpoonup$. In other words, $\rightharpoonup$ is *complete.*

Theorem 4. *Let V_1', V_2', $\overline{V_1}$, and $\overline{V_2}$ be in $\boldsymbol{Com_{\mathcal{V}}}$ and let $V_1, V_2 \in \boldsymbol{Com}$.*

1. *If $V_i \rightharpoonup \overline{V_i}$ can be derived then $\overline{V_i}$ is a lifting of V_i $(i = 1, 2)$.*
2. *Suppose $\overline{V_1}$ ($\overline{V_2}$) shares no meta-variables with V_1', V_2', and $\overline{V_2}$ (V_1', V_2', and $\overline{V_1}$). If $V_1 \rightharpoonup \overline{V_1}$ and $V_2 \rightharpoonup \overline{V_2}$ can be derived and V_1' and V_2' are liftings of V_1, V_2, respectively, then $\mathcal{U}(\{V_1' \bumpeq^?_L V_2'\}) \neq \emptyset$ implies $\mathcal{U}(\{\overline{V_1} \bumpeq^?_L \overline{V_2}\}) \neq \emptyset$. Furthermore, $\mathcal{U}(\{V_1' \bumpeq^?_L V_1'\}) \neq \emptyset$ implies $\mathcal{U}(\{\overline{V_1} \bumpeq^?_L \overline{V_1}\}) \neq \emptyset$.*

Integrating Standard Unification Algorithms. Standard algorithms for unification modulo an associative and commutative operator with neutral element and constants (see, e.g., [BS01] for background information on AC_1 unification) build on a characterization of equality that is equivalent to the one in Lemma 1. This correspondence allows one to employ existing algorithms for AC_1-unification problems with constants and free function symbols (like, e.g., the one in [HS87]) to the unification problems that arise when applying the rule for conditionals and then to filter the output such that only preserving substitutions remain.[7]

[7] For the reader familiar with AC_1 unification: In the language $Stut_{\mathcal{V}}$ one views ϵ as the neutral element, skip as the constant, and ; as the operator. For $Slice_{\mathcal{V}}$, the remaining language constructs, i.e., assignments, conditionals, loops, forks, and ; (outside the language $Stut_{\mathcal{V}}$) must be treated as free constructors.

$$\frac{C_1 \bumpeq^?_L C_2 :: \eta \quad C_1, C_2 \in \mathit{Stut}_{\mathcal{V}}}{X; C_1 \bumpeq^?_L C_2 :: \eta[X \backslash \epsilon]}\ [\mathit{Seq_1}] \qquad \frac{C_1 \bumpeq^?_L C_2 :: \eta \quad C_1, C_2 \in \mathit{Stut}_{\mathcal{V}}}{\mathsf{skip}; C_1 \bumpeq^?_L \mathsf{skip}; C_2 :: \eta}\ [\mathit{Seq_2}]$$

$$\frac{C_1 \bumpeq^?_L C_1' :: \eta_1 \quad C_2 \bumpeq^?_L C_2' :: \eta_2 \quad C_1, C_1' \in \mathit{NSeq}_{\mathcal{V}}}{C_1; C_2 \bumpeq^?_L C_1'; C_2' :: \eta_1 \cup \eta_2}\ [\mathit{Seq_3}] \qquad \frac{C \in \mathit{Stut}_{\mathcal{V}} \cup \{\epsilon\}}{X \bumpeq^?_L C :: \{X \backslash C\}}\ [\mathit{Var_1}]$$

$$\frac{C_1 \bumpeq^?_L C_1' :: \eta_1 \quad C_2 \bumpeq^?_L C_2' :: \eta_2 \quad C_1, C_1' \in \mathit{Stut}_{\mathcal{V}} \cup \{\epsilon\}, C_2, C_2' \in \mathit{NStut}_{\mathcal{V}}}{C_1; C_2 \bumpeq^?_L C_1'; C_2' :: \eta_1 \cup \eta_2}\ [\mathit{Seq_4}]$$

$$\frac{\mathit{Id} : \mathit{low} \quad \mathit{Exp}_1 \equiv \mathit{Exp}_2}{\mathit{Id}{:=}\mathit{Exp}_1 \bumpeq^?_L \mathit{Id}{:=}\mathit{Exp}_2 :: \emptyset}\ [\mathit{Asg}] \qquad \frac{C \bumpeq^?_L C' :: \eta \quad V \bumpeq^?_L V' :: \eta_2}{\mathsf{fork}(CV) \bumpeq^?_L \mathsf{fork}(C'V') :: \eta_1 \cup \eta_2}\ [\mathit{Frk}]$$

Fig. 4. Unification calculus

Automating Unification. In the following, we go beyond simply applying an existing unification algorithm by exploiting the specific shape of our unification problems and the limited range of substitutions in the computation of unifiers. Recall that we operate on programs in $\mathit{Slice}_{\mathcal{V}}$, i.e., on programs without assignments to high variables, without assignments of high expressions to low variables, and without loops or conditionals having high guards.

The operative intuition behind our problem-tailored unification algorithm is to scan two program terms from left to right and distinguish two cases: if both leftmost subcommands are free constructors, (low assignments, loops, conditionals and forks) they are compared and, if they agree, unification is recursively applied to pairs of corresponding subprograms and the residual programs. If one leftmost subcommand is skip, both programs are decomposed into their maximal initial subprograms in $\mathit{Stut}_{\mathcal{V}}$ and the remaining program. Unification is recursively applied to the corresponding subprograms. Formally, we define the language $\mathit{NSeq}_{\mathcal{V}}$ of commands in $\mathit{Slice}_{\mathcal{V}} \setminus \{\mathsf{skip}\}$ without sequential composition as a top-level operator, and the language $\mathit{NStut}_{\mathcal{V}}$ of commands in which the leftmost subcommand is not an element of $\mathit{Stut}_{\mathcal{V}}$. $\mathit{NStut}_{\mathcal{V}}$ is given by $C ::= C_1; C_2$, where $C_1 \in \mathit{NSeq}_{\mathcal{V}}$ and $C_2 \in \mathit{Slice}_{\mathcal{V}}$.

The unification algorithm in Figure 4 is given in form of a calculus for deriving judgments of the form $C_1 \bumpeq^?_L C_2 :: \eta$, meaning that η is a preserving unifier of the commands C_1 and C_2. The symmetric counterparts of rules [$\mathit{Seq_1}$],[$\mathit{Var_1}$] are omitted, as are the rules for loops, conditionals and command vectors, because they are analogous to [Frk]. Note that the unifiers obtained from recursive application of the algorithm to sub-programs are combined by set union. This is admissible if the meta-variables in all subprograms are disjoint, as the following lemma shows:

Lemma 2. *Let $V_1, V_2 \in$ **$\mathit{Slice}_{\mathcal{V}}$** and let every variable occur at most once in (V_1, V_2). Then $V_1 \bumpeq^?_L V_2 :: \eta$ implies $\eta \in \mathcal{U}(\{V_1 \bumpeq^?_L V_2\})$*

Observe that the stand-alone unification algorithm is not complete, as it relies on the positions of meta-variables inserted by $\rightharpoonup$. However, we can prove a completeness result for the combination of both calculi.

Completeness. If conditionals with high guards are nested then the process of transformational typing possibly involves repeated applications of substitutions to a given subprogram. Hence, care must be taken in choosing a substitution in each application of rule $[Cond_h]$ because, otherwise, unification problems in later applications of $[Cond_h]$ might become unsolvable.[8] Fortunately, the instantiation of our framework presented in this section does not suffer from such problems.

Theorem 5 (Completeness). *Let $V \in \boldsymbol{Com}$, $\overline{V}, W \in \boldsymbol{Com}_{\mathcal{V}}$, W be a lifting of V, and $V \rightharpoonup \overline{V}$.*

1. *If there is a preserving substitution σ with $\sigma W \doteq_L \sigma W$, then $\overline{V} \hookrightarrow' V' : S$ for some $V', S \in \boldsymbol{Com}_{\mathcal{V}}$.*
2. *If $W \hookrightarrow W' : S$ for some $W', S \in \boldsymbol{Com}_{\mathcal{V}}$ then $\overline{V} \hookrightarrow' V' : S'$ for some $V', S' \in \boldsymbol{Com}_{\mathcal{V}}$.*

Here, the judgment $V \hookrightarrow' V' : S$ denotes a successful transformation of V to V' by the transformational type system, where the precondition $\sigma \in \mathcal{U}(\{S_1 \doteq_L^? S_2\})$ is replaced by $S_1 \doteq_L^? S_2 :: \sigma$ in rule $[Cond_h]$.

6 Related Work and Discussion

Type-based approaches to analyzing the security of the information flow in concrete programs have received much attention in recent years [SM03]. This resulted in security type systems for a broad range of languages (see, e.g., [VS97,SV98,HR98,Mye99,Sab01,SM02,BN02,HY02,BC02,ZM03,MS04]).

Regarding the analysis of conditionals with high guards, Volpano and Smith [VS98] proposed the atomic execution of entire conditionals for enforcing observational equivalence of alternative execution paths. This somewhat restrictive constraint is relaxed in the work of Agat [Aga00] and Sabelfeld and Sands [SS00] who achieve observational equivalence by cross-copying the slices of branches. The current article introduces unification modulo an equivalence relation as another alternative for making the branches of a conditional observationally equivalent to each other. Let us compare the latter two approaches more concretely for the relation $\doteq_L$ that we have introduced to instantiate our approach.

The type system introduced in Section 4 is capable of analyzing programs where assignments to low variables appear in the branches of conditionals with high guards, which is not possible with the type system in [SS00].

Example 1. If one lifts $C = \mathsf{if}\ h_1\ \mathsf{then}\ (h_2{:=}Exp_1; l{:=}Exp_2)\ \mathsf{else}\ (l{:=}Exp_2)$ where $Exp_2 : low$ using our lifting calculus, applies our transforming type system, and finally removes all remaining meta-variables by applying a projection then this results in $\mathsf{if}\ h_1\ \mathsf{then}\ (h_2{:=}Exp_1; l{:=}Exp_2)\ \mathsf{else}\ (\mathsf{skip}; l{:=}Exp_2)$, a program that is strongly secure and also weakly bisimilar to C. Note that the program C cannot be repaired by applying the type system from [SS00]. ◊

[8] A standard solution would be to apply *most general unifiers*. Unfortunately, they do not exist in our setting

Another advantage of our unification-based approach over the cross-copying technique is that the resulting programs are faster and smaller in size.

Example 2. The program $\mathsf{if}\ h\ \mathsf{then}\ (h_1{:=}\mathit{Exp}_1)\ \mathsf{else}\ (h_2{:=}\mathit{Exp}_2)$ is returned unmodified by our type system, while the type system from [SS00] transforms it into the bigger program $\mathsf{if}\ h\ \mathsf{then}\ (h_1{:=}\mathit{Exp}_1; \mathsf{skip})\ \mathsf{else}\ (\mathsf{skip}; h_2{:=}\mathit{Exp}_2)$. If one applies this type system a second time, one obtains an even bigger program, namely $\mathsf{if}\ h\ \mathsf{then}\ (h_1{:=}\mathit{Exp}_1; \mathsf{skip}; \mathsf{skip}; \mathsf{skip})\ \mathsf{else}\ (\mathsf{skip}; \mathsf{skip}; \mathsf{skip}; h_2{:=}\mathit{Exp}_2)$. In contrast, our type system realizes a transformation that is idempotent, i.e. the program resulting from the transformation remains unmodified under a second application of the transformation. $\Diamond$

Non-transforming security type systems for the two-level security policy can be used to also analyze programs under a policy with more domains. To this end, one performs multiple type checks where each type check ensures that no illegitimate information flow can occur into a designated domain. For instance, consider a three-domain policy with domains $\mathcal{D} = \{\mathit{top}, \mathit{left}, \mathit{right}\}$ where information may only flow from *left* and from *right* to *top*. To analyze a program under this policy, one considers all variables with label *top* and *left* as if labeled *high* in a first type check (ensuring that there is no illegitimate information flow to *right*) and, in a second type check, considers all variables with label *top* and *right* as if labeled *high*. There is no need for a third type check as all information may flow to *top*. When adopting this approach for transforming type systems, one must take into account that the guarantees established by the type check for one domain might not be preserved under the modifications caused by the transformation for another domain. Therefore, one needs to iterate the process until a fixpoint is reached for all security domains.

Example 3. For the three-level policy from above, the program $C = \mathsf{if}\ t\ \mathsf{then}\ (t{:=}t'; r{:=}r'; l{:=}l')\ \mathsf{else}\ (r{:=}r'; l{:=}l')$ (assuming $t, t' : \mathit{top}$, $r, r' : \mathit{right}$ and $l, l' : \mathit{left}$) is lifted to $\overline{C} = \mathsf{if}\ t\ \mathsf{then}\ (t{:=}t'; r{:=}r'; \alpha_1; l{:=}l'; \alpha_2)\ \mathsf{else}\ (r{:=}r'; \alpha_3; l{:=}l'; \alpha_4)$ and transformed into $\mathsf{if}\ t\ \mathsf{then}\ (t{:=}t'; r{:=}r'; l{:=}l')\ \mathsf{else}\ (r{:=}r'; \mathsf{skip}; l{:=}l')$ when analyzing security w.r.t. an observer with domain *left*. Lifting for *right* then results in $\mathsf{if}\ t\ \mathsf{then}\ (t{:=}t'; \alpha_1; r{:=}r'; l{:=}l'; \alpha_2)\ \mathsf{else}\ (\alpha_3; r{:=}r'; \mathsf{skip}; l{:=}l'; \alpha_4)$. Unification and projection gives $\mathsf{if}\ t\ \mathsf{then}\ (t{:=}t'; r{:=}r'; l{:=}l'; \mathsf{skip})\ \mathsf{else}\ (\mathsf{skip}; r{:=}r'; \mathsf{skip}; l{:=}l')$. Observe that this program is not secure any more from the viewpoint of a *left*–observer. Applying the transformation again for domain *left* results in the secure program $\mathsf{if}\ t\ \mathsf{then}\ (t{:=}t'; r{:=}r'; \mathsf{skip}; l{:=}l'; \mathsf{skip})\ \mathsf{else}\ (\mathsf{skip}; r{:=}r'; \mathsf{skip}; l{:=}l'; \mathsf{skip})$, which is a fixpoint of both transformations. $\Diamond$

Note that the idempotence of the transformation is a crucial prerequisite (but not a sufficient one) for the existence of a fixpoint and, hence, for the termination of such an iterative approach. As is illustrated in Example 2, the transformation realized by our type system is idempotent, whereas the transformation from [SS00] is not.

Another possibility to tackle multi-level security policies in our setting is to unify the branches of a conditional with guard of security level D' under the theory $\bigcap_{D \not\geq D'} \hat{=}_D$. An investigation of this possibility remains to be done.

The chosen instantiation of our approach preserves the program behavior in the sense of a weak bisimulation. Naturally, one can correct more programs if one is willing to relax this relationship between input and output of the transformation. For this reason, there are also some programs that cannot be corrected with our type system although they can be corrected with the type system in [SS00] (which assumes a weaker relationship between input and output).

Example 4. if h then (while l do $(h_1{:=}Exp)$) else $(h_2{:=}1)$ is rejected by our type system. The type system in [SS00] transforms it into the strongly secure program if h then (while l do $(h_1{:=}Exp)$; skip) else (while l do (skip); $h_2{:=}1$). Note that this program is not weakly bisimilar to the original program as the cross-copying of the while loop introduces possible non-termination. $\Diamond$

If one wishes to permit such transformations, one could, for instance, choose a simulation instead of the weak bisimulation when instantiating our approach. This would result in an extended range of substitutions beyond $Stut_{\mathcal{V}}$. For instance, to correct the program in Example 4, one needs to instantiate a meta-variable with a while loop. We are confident that, in such a setting, using our approach would even further broaden the scope of corrections while retaining the advantage of transformed programs that are comparably small and fast.

7 Conclusions

We proposed a novel approach to analyzing the security of information flow in concrete programs with the help of transforming security type systems where the key idea has been to integrate unification with typing rules. This yielded a very natural perspective on the problem of eliminating implicit information flow.

We instantiated our approach by defining a program equivalence capturing the behavioral equivalence to be preserved during the transformation and an observational equivalence capturing the perspective of a low-level attacker. This led to a novel transforming security type system and calculi for automatically inserting meta-variables into programs and for computing substitutions. We proved that the resulting analysis technique is sound and also provided a relative completeness result. The main advantages of our approach include that the precision of type checking is improved, that additional insecure programs can be corrected, and that the resulting programs are faster and smaller in size.

It will be interesting to see how our approach performs for other choices of the parameters like, e.g., observational equivalences that admit intentional declassification [MS04]). Another interesting possibility is to perform the entire information flow analysis and program transformation using unification without any typing rules, which would mean to further explore the possibilities of the PER model. Finally, it would be desirable to integrate our fully automatic transformation into an interactive framework for supporting the programmer in correcting insecure programs.

References

[Aga00] J. Agat. Transforming out Timing Leaks. In *Proceedings of the 27th ACM Symposium on Principles of Programming Languages*, pages 40–53, 2000.

[BC02] G. Boudol and I. Castellani. Noninterference for Concurrent Programs and Thread Systems. *Theoretical Computer Science*, 281:109–130, 2002.

[BN02] A. Banerjee and D. A. Naumann. Secure Information Flow and Pointer Confinement in a Java-like Language. In *Proceedings of the 15th IEEE Computer Security Foundations Workshop*, pages 253–270, Cape Breton, Nova Scotia, Canada, 2002.

[BS01] F. Baader and W. Snyder. Unification theory. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, volume I, chapter 8, pages 445–532. Elsevier Science, 2001.

[HR98] N. Heintze and J. G. Riecke. The SLam Calculus: Programming with Secrecy and Integrity. In *Proceedings of the 25th ACM Symposium on Principles of Programming Languages*, pages 365–377, 1998.

[HS87] A. Herold and J. Siekmann. Unification in Abelian Semigroups. *Journal of Automated Reasoning*, 3:247–283, 1987.

[HY02] K. Honda and N. Yoshida. A uniform type structure for secure information flow. In *Proceedings of the 29th ACM Symposium on Principles of Programming Languages*, pages 81–92. ACM Press, 2002.

[KM05] Boris Köpf and Heiko Mantel. Eliminating Implicit Information Leaks by Transformational Typing and Unification. Technical Report 498, ETH Zürich, 2005.

[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 3303, pages 129–145, Taipei, Taiwan, November 4–6 2004. Springer-Verlag.

[Mye99] A. Myers. JFlow: Practical mostly-static information flow control. In *Symposium on Principles of Programming Languages*, pages 228–241, 1999.

[Sab01] A. Sabelfeld. The Impact of Synchronisation on Secure Information Flow in Concurrent Programs. In *Proceedings of Andrei Ershov 4th International Conference on Perspectives of System Informatics*, volume 2244 of *LNCS*, pages 225–239, 2001.

[SM02] A. Sabelfeld and H. Mantel. Static Confidentiality Enforcement for Distributed Programs. In *Proceedings of the 9th International Static Analysis Symposium, SAS'02*, volume 2477 of *LNCS*, pages 376–394, Madrid, Spain, 2002.

[SM03] A. Sabelfeld and A. C. Myers. Language-based Information-Flow Security. *IEEE Journal on Selected Areas in Communication*, 21(1):5–19, 2003.

[SS99] A. Sabelfeld and D. Sands. A Per Model of Secure Information Flow in Sequential Programs. In *Proceedings of the 8th European Symposium on Programming*, LNCS, pages 50–59, 1999.

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

[SV98] G. Smith and D. Volpano. Secure Information Flow in a Multi-Threaded Imperative Language. In *25th ACM Symposium on Principles of Programming Languages, San Diego, California*, pages 355–364, 1998.

[VS97] D. Volpano and G. Smith. A Type-Based Approach to Program Security. In *TAPSOFT 97*, volume 1214 of *LNCS*, pages 607–621, 1997.

[VS98] D. Volpano and G. Smith. Probabilistic Noninterference in a Concurrent Language. In *Proceedings of the 11th IEEE Computer Security Foundations Workshop*, pages 34–43, Rockport, Massachusetts, 1998.

[ZM03] S. Zdancewic and A. Myers. Observational determinism for concurrent program security. In *Proceedings of the 16th IEEE Computer Security Foundations Workshop, 2003*, pages 29–47. IEEE Computer Society, 2003.

A Semantics of MWL

The operational semantics for MWL are given in Figures 5 and 6.

$$\frac{\lAngle C_i, s \rAngle \rightarrowtriangle \lAngle W', t \rAngle}{\lAngle \langle C_0 \ldots C_{n-1} \rangle, s \rAngle \rightarrowtriangle \lAngle \langle C_0 \ldots C_{i-1} \rangle W' \langle C_{i+1} \ldots C_{n-1} \rangle, t \rAngle}$$

Fig. 5. Small-step nondeterministic semantics

$$\lAngle \mathsf{skip}, s \rAngle \rightarrowtriangle \lAngle \langle\rangle, s \rAngle \qquad \frac{\lAngle \mathit{Exp}, s \rAngle \downarrow n}{\lAngle \mathit{Id}{:=}\mathit{Exp}, s \rAngle \rightarrowtriangle \lAngle \langle\rangle, [\mathit{Id} = n]s \rAngle}$$

$$\frac{\lAngle C_1, s \rAngle \rightarrowtriangle \lAngle \langle\rangle, t \rAngle}{\lAngle C_1; C_2, s \rAngle \rightarrowtriangle \lAngle C_2, t \rAngle} \quad \frac{\lAngle C_1, s \rAngle \rightarrowtriangle \lAngle \langle C_1' \rangle V, t \rAngle}{\lAngle C_1; C_2, s \rAngle \rightarrowtriangle \lAngle \langle C_1'; C_2 \rangle V, t \rAngle} \quad \lAngle \mathsf{fork}(CV), s \rAngle \rightarrowtriangle \lAngle \langle C \rangle V, s \rAngle$$

$$\frac{\lAngle B, s \rAngle \downarrow \mathsf{True}}{\lAngle \mathsf{if}\ B\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2, s \rAngle \rightarrowtriangle \lAngle C_1, s \rAngle} \qquad \frac{\lAngle B, s \rAngle \downarrow \mathsf{False}}{\lAngle \mathsf{if}\ B\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2, s \rAngle \rightarrowtriangle \lAngle C_2, s \rAngle}$$

$$\frac{\lAngle B, s \rAngle \downarrow \mathsf{True}}{\lAngle \mathsf{while}\ B\ \mathsf{do}\ C, s \rAngle \rightarrowtriangle \lAngle C; \mathsf{while}\ B\ \mathsf{do}\ C, s \rAngle} \qquad \frac{\lAngle B, s \rAngle \downarrow \mathsf{False}}{\lAngle \mathsf{while}\ B\ \mathsf{do}\ C, s \rAngle \rightarrowtriangle \lAngle \langle\rangle, s \rAngle}$$

Fig. 6. Small-step deterministic semantics