Master Thesis

A security analyzer for multi-threaded programs

Author(s):
Pöpper, Christina

Publication Date:
2005

Permanent Link:
https://doi.org/10.3929/ethz-a-004922814

Rights / License:
In Copyright - Non-Commercial Use Permitted
Diploma Thesis

A Security Analyzer for Multi-Threaded Programs

Christina Pöpper

ETH Zürich
Department of Computer Science
Information Security Group

Prof. Dr. David Basin

Supervisors: Dr. Heiko Mantel
Boris Köpf

November 08, 2004 – March 07, 2005
Abstract

Information flow analysis can be used to explore and protect the confidentiality and integrity of information, which is one aspect of information security. Confidentiality can be analyzed by exploring how information is propagated during the execution of a program, which may be investigated on the level of a concrete programming language. Security type systems are a mechanism for reasoning about the security of programs with respect to an information flow policy. In this thesis, we have designed and implemented a tool for mechanically analyzing the confidentiality of programs in a multi-threaded language by means of two existing security type systems. We extend the security type systems for arrays, including a proof of soundness, in order to widen the analysis to a broader class of programs. We present example programs that are suitable for studying the flow of information and demonstrate the applicability of our extended type system and its implementation in the security analyzer.

Acknowledgment

I warmly thank Heiko and Boris for introducing me into the subject and guiding me through this thesis. Both supported me whenever needed and were open for all kind of questions, theoretical ones as well as practical ones.

Regina and Mike receive my thanks for proof-reading and helping me with English corrections.

I am deeply grateful to Philippe for the fruitful discussions during this thesis and his support during my studies.
Contents

1 Introduction 5
  1.1 Motivation ................................. 5
  1.2 Background ............................... 6
  1.3 Structure of this Document ............... 7

2 Preliminaries 9
  2.1 The Multi-Threaded While Language ........ 9
  2.2 Security Model and Terminology ............. 10
    2.2.1 Setting .................................. 10
    2.2.2 Definition of Security .................. 11
  2.3 Security Type Systems ..................... 13
    2.3.1 Security Type System without Transformation .... 14
    2.3.2 Security Type System with Transformation .......... 17
    2.3.3 Soundness ............................... 20

3 Design of the Security Analyzer 21
  3.1 System Architecture .......................... 21
  3.2 Overall Class Model .......................... 22
  3.3 Representation of Programs ................... 23
  3.4 Parser ..................................... 25
  3.5 Security Type Systems Used ................... 25
  3.6 Design Pattern .............................. 25

4 Implementation 29
  4.1 Adaptation of MWL ............................ 29
  4.2 Parser ..................................... 30
  4.3 Analyzer without Transformation ............. 31
  4.4 Analyzer with Transformation ................. 33
  4.5 Output ..................................... 34
  4.6 Open Issues ................................. 35

5 Type System Extension: Variable Declarations and Arrays 37
  5.1 Variable Declarations ........................ 37
  5.2 Expressions .................................. 41
  5.3 Arrays ..................................... 43
    5.3.1 Design Questions and Literature .......... 43
    5.3.2 Syntax, Semantics, and Typing Rules ....... 44
### CONTENTS

5.3.3 Soundness of Typing ........................................... 45  
5.3.4 Implementation ............................................ 52  

6 Tests and Examples ............................................. 53  
6.1 Search Algorithms ........................................... 53  
6.1.1 Simple Search ............................................ 53  
6.1.2 Binary Search .............................................. 54  
6.2 Sort Algorithms ............................................. 56  
6.2.1 Bubble Sort ................................................ 56  
6.2.2 Insertion Sort .............................................. 57  
6.2.3 Selection Sort .............................................. 58  
6.3 Transformation by Hand ...................................... 59  
6.4 Summary ...................................................... 60  

7 Conclusion ....................................................... 61  
7.1 Discussion of Results ........................................ 61  
7.2 Related Work .................................................. 62  
7.3 Future Work .................................................... 62  

References ......................................................... 65  

A Formulation of Thesis Objectives ............................... 69  

B User Documentation ........................................... 73  
B.1 Installation Instruction ....................................... 73  
B.2 Running the Analyzer ........................................ 73  
B.3 Makefile ........................................................ 74  
B.4 Source Code Organization .................................... 74  
B.5 Example Programs ............................................ 74  
B.6 Remarks ........................................................ 75  

C Source Code ...................................................... 77  
C.1 Parser Generation ............................................. 77  
C.2 Package AST ................................................... 80  
C.3 Package Checker ............................................... 84  
C.4 Package Visitor ............................................... 89
1 Introduction

1.1 Motivation

In the increasingly connected world we live in, the protection of information is an important and challenging task. Various approaches exist for addressing different aspects of information security, such as e.g. digital signatures for guaranteeing authenticity, message authentication codes for achieving data integrity, and anti-virus programs for identifying malicious software.

This thesis investigates another aspect of information security: Protecting the confidentiality of information. Confidentiality means that no one obtains information but those who are authorized to see it. Access control is a standard mechanism for achieving confidentiality, but it only controls which data a program can access. Information flow control additionally controls how information is propagated within a program. Confidentiality is violated if information is leaked during its propagation, i.e. if executing the program reveals details about the confidential data that the program operates on, such as credit card numbers, salaries, or passwords.

As an example, imagine you want to invest money but you are still undecided with which bank. To help you decide, you have downloaded a program that computes the yield of a hypothetical investment for your assets. This program needs to access your balance in order to compute the yield. It has to communicate with various banks to get the current interest rates. How can you be sure that it does not send your balance to these banks without your consent?

Confidentiality of programs is defined with respect to security requirements expressed by a security policy. The security policy defines how information is permitted to be propagated within a program and to the outside world. This can be expressed by a noninterference policy, which allows programs to manipulate and modify private data if the visible outputs of the program do not reveal information about confidential data.

To ensure that a program processes information only in accordance with a given confidentiality policy, it is necessary to analyze how information flows within the program. Such an analysis can be performed on the level of source code, on the level of machine code or on the level of specifications [SM03].

In this thesis, we analyze the confidentiality of programs in a simple imperative language on the level of source code. We explore methods for analyzing the information flow automatically. Then, we use these methods to design and implement a security analyzer that mechanically checks programs written in this language. Additionally, we give a number of example programs that demonstrate the applicability of the developed security analyzer. Since meaningful example programs were not easy to find or come up with, we extend the method of analyzing the information flow. This includes a proof of soundness of the resulting method. The extension enables us to operate on a broader class of programs.

The formulation of the thesis task can be found in Appendix A.
1.2 Background

Information flow analysis aims at finding out whether a program is secure with respect to confidentiality or integrity. For enabling concrete reasoning, information flow security must be formalized. One way of formalizing it is to define security with respect to the way that an untrusted person observes the behavior of a program. The program should only be considered as secure if a rigorous analysis shows that it enforces the given confidentiality policy. Information must not flow to a location where this policy is violated.

Static program analysis can then be used to analyze the information flow security of a program. Security type systems approximate this analysis and provide a basis for automatic reasoning. They consist of a set of rules, typing rules, that allow reasoning about the security of a program in a modular fashion. Security type systems must not be confused with type systems in the conventional way, which check that programs are well-typed, i.e. that operations are performed on instances of proper types only.

The existing security type systems can be classified into non-transforming and transforming systems. Non-transforming type systems determine whether a given program is secure or not. If the type check succeeds the program is secure (in some well-defined sense), if it fails the program might be insecure, which is an approximation to the analysis. Transforming type systems transform a given program into a new program, trying to close information leaks. If a program can be transformed the resulting program is secure (in some well-defined sense), if it cannot be transformed the original program might be insecure and cannot be repaired. The constructed program should be similar enough so that it can replace the original program.

Non-transforming type systems are often less complex and consequently easier to apply, but they provide only little information about where the problem lies if the check fails. In particular they do not solve the problem on how to eliminate the information leak. Rather than only checking whether a given program has secure information flow, non-transforming type systems potentially construct a program with secure flow. However, with current type systems, this often happens on the cost of quality of code. Since the two kinds of security type systems have different advantages and drawbacks it makes sense to consider and apply both of them. However, if a program is accepted already by a non-transforming type system it is unnecessary to check it again with a transforming type system for the same confidentiality policy.

What is crucial to non-transforming and transforming security type systems is the concept of soundness. Soundness is defined with respect to a given security policy. It is a property of security type systems that expresses that no insecure program is accepted as secure. For being able to rely on the result of the analysis it is desirable that security type systems are sound. This is proved by so-called soundness proofs. The more secure programs a security type system actually accepts, the better it is, which is captured by the term preciseness.
1.3 Structure of this Document

This document is organized as follows. Section 2 introduces the terminology and preconditions the analysis is based on. It presents the imperative language MWL and existing security type systems as methods to analyze programs in this language. Section 3 gives insight into the design of the developed security analyzer. Implementation details are provided in Section 4. Section 5 describes theoretical extensions of the used security type systems. Section 6 states example programs to which the security analyzer has been applied and analyzes them. Section 7 concludes the thesis and gives an outlook on future work.
2 Preliminaries

We will analyze programs written in the simple imperative language MWL, which is taken from [SS00]. The language itself is introduced in Section 2.1. Section 2.2 presents the security model that underlies the security analysis, specifying the setting and terminology. Security type systems as analyzing methods are introduced in Section 2.3, including the presentation of two existing type systems.

2.1 The Multi-Threaded While Language

In [SS00], Sabelfeld and Sands suggested a minimalistic concurrent language, the Multi-threaded While Language (short: MWL), to study the security of information flow in programs with multiple threads. We adopt this language in this thesis. MWL is comprised of blank commands, assignments, conditional branching, loops, sequential composition, and commands for dynamic thread creation. The following grammar describes its set of commands:

\[
C ::= \text{skip} \mid \text{Id}:\text{Exp} \mid \text{if } B \text{ then } C_1 \text{ else } C_2 \mid \text{while } B \text{ do } C \mid \text{fork}(C,D)
\]

First of all, since MWL is a multi-threaded language, multi-threadedness has to be clarified. Threads are sequential programs. A multi-threaded program may consist of multiple threads. Shared memory and a single processor are assumed for the execution of a program. An arbitrary scheduler decides on the order in which the threads are executed.

The operational semantics of the commands is formalized in Figure 1 [SS00]. We will clarify the notation. In the following, a state \( s \) is a mapping from variables to values. A configuration is a pair \( \langle D, s \rangle \), where \( D \) is a vector of commands and \( s \) is a state. The judgment \( \langle C, s \rangle \rightarrow \langle D, s' \rangle \) stands for a transition: A thread with program \( C \) performs a computation step in state \( s \), yielding a state \( s' \) and \( \langle D, s \rangle \).
a vector of commands $\vec{D}$. $\vec{D}$ is empty (denoted by $\langle \rangle$) if $C$ has terminated, it has length one if the first command was executed without spawning new threads, and length $>1$ if threads were spawned. The judgment $\{Exp, s\} \vdash n$ expresses that expression $Exp$ evaluates to the value $n$ in state $s$.

The semantics in Figure 1 is given in the form \begin{align*}
\text{hypotheses} \quad \text{conclusion}
\end{align*}
, i.e. the conclusion can be inferred if the hypotheses are fulfilled. For instance, the first two rules can be read as follows: A skip command is processed in one step and does not modify the state. It does not depend on any hypotheses. An assignment $Id:=Exp$ is executed in one step and, if the right-hand side evaluates to $n$, the state is modified by setting the value of variable $Id$ to $n$, i.e. $Id$ is mapped to $n$.

All but the last rule for thread pools are deterministic. The last rule represents the concurrent part of the semantics. Among the currently active threads, the scheduler chooses thread $i$ for execution.

Throughout the thesis, we use a set notation for specifying the mapping of variables to values: $s = \{v_1=val_1, v_2=val_2, \ldots\}$ denotes that in state $s$ the variable $v_1$ is mapped to value $val_1$, variable $v_2$ to $val_2$, etc. States are complete in the variables: Each variable occurring in a program is mapped to a value and hence specified in the set notation of $s$.

2.2 Security Model and Terminology

This section defines the security model that is assumed in the following. It introduces the setting and terminology.

2.2.1 Setting

Assume a security policy with two security domains, a high level and a low level. The high level represents confidential information and the low level stands for public data. As depicted in Figure 2, information may flow within one level and from the low to the high level, but the security requirement forbids flow from high to low. As an example, recall the bank scenario from the introduction. Most persons would probably agree that the balance of their own bank account is high data. They would not want it to flow into the low data which is exchanged during the communication with the other banks.

In a security-typed language, every program variable is associated – or labeled – with one of the two security domains. The labeling with a security domain is static and does not change during program execution. Assume that $h$ denotes a variable of security domain high and $l$ is a low variable.

Now that we have defined the syntax and semantics of the multi-threaded language as well as the security policy we can specify what it means for a program to be secure.

The intuition is that values of low variables can be observed by an attacker and hence, they should not be used to store confidential data. Whether a program is
secure or not depends on the concrete observational capabilities of the attacker. Here, the attacker is modeled as a \textit{low-level observer}. He can monitor the timing and termination behavior of a program and may observe the values and modifications of low variables, even during the execution of the program and not only at the end of the program run. For a program to be secure, a variation of the confidential input must not cause a variation of the public output.

Before we make the definition of security concrete in Section 2.2.2, let us consider how information can be leaked. Confidential information is potentially leaked if the value of a low variable depends on the initial value of a high variable. The ways how this may happen can be categorized as follows. \textit{Direct} information leakage passes confidential data directly to the attacker, e.g. by outputting high variables or assigning a high variable to a low variable which is publicly observable (e.g. \(l := h\)). \textit{Indirect} (or \textit{implicit}) leakage refers to information flow through the control structure of a program (e.g. \(\text{if } h == 1 \text{ then } l := 1 \text{ else } l := 0\)). \textit{Timing} leakage occurs when the confidential data is encoded in the timing behavior. For instance, the runtime may differ depending on whether a loop with a high guard is executed or not (e.g. \(\text{while } h > 0 \text{ do } h := h - 1\)). Finally, \textit{termination} behavior may leak information through the fact whether the computation terminates or not (e.g. \(\text{while } h > 0 \text{ do } \text{skip}\)).

If confidential information is leaked in any way the program in question should be considered insecure. This must be reflected in a definition of security.

\subsection{Definition of Security}

Security is defined in terms of a \textit{noninterference} property: Varying the initial values of high variables does not change the low-level observations. This concept of security depends on the observational capabilities of the observer. We assume that the attacker is capable of distinguishing states if they differ in the values of low variables, but he cannot distinguish states that are low-equal.

\begin{definition}[Low-equality] Two states \(s_1\) and \(s_2\) are low-equal, denoted by \(s_1 =_L s_2\), if they map the same values to each low variable.\end{definition}

Low-equal states agree on the low variables. For instance, the states \(s_1 = \{l = 1, h_1 = 2, h_2 = 3\}\) and \(s_2 = \{l = 1, h_1 = 4, h_2 = 0\}\) are low-equal.

Since we assume that the attacker can observe the states at any time during a program run, the attacker is capable to distinguish two runs if they contain two corresponding states that are not low-equal. The attacker cannot distinguish the runs if all two corresponding states are low-equal. Sabelfeld and Sands formalize this intuition in the notion of a strong low-bisimulation \cite{SS00}.

\begin{definition}[Strong low-bisimulation] The strong low-bisimulation \(\cong_L\) is the union of all symmetric relations \(R\) on thread pools of equal size, such that if \(\langle C_1 \ldots C_n \rangle \ R \langle D_1 \ldots D_n \rangle\) then

\[\forall s_1 =_L s_2 \ \forall i \langle C_i, s_i \rangle \rightarrow \langle C_i', s_i' \rangle \implies \exists D_i, s_2' \langle D_i, s_2 \rangle \rightarrow \langle D_i', s_2' \rangle, C_i R D_i', s_1' =_L s_2'.\]

That means, that two strongly low-bisimilar thread pools must have the same number of threads, and each thread must execute in lock-step and affect the low memory in the same way as the respective thread in the related thread pool.

It can be shown that the relation $\equiv_L$ is symmetric and transitive. However, it is not reflexive.

Given that, security is specified as a partial equivalence on programs:

**Definition 3 (Strong Security).** $\bar{C}$ is (strongly) secure if and only if $\bar{C} \equiv_L \bar{C}$.

The intuition is that a program $C$ has secure information flow if and only if $C$ is strongly low-bisimilar to itself for all low-equal states. Consider the following examples:

- $C_1 = \text{h:=h+1; l:=l+1}$
- $C_2 = \text{h:=h-1; l:=l+1}$
- $C_3 = \text{l:=0; if h==1 then l:=1}$

For a low observer, the programs $C_1$ and $C_2$ are indistinguishable. They are strongly low-bisimilar: $\text{h:=h+1; l:=l+1} \equiv_L \text{h:=h-1; l:=l+1}$. The first command does not affect low variables. The second command modifies $l$ in exactly the same way in all low-equal states.

With that, we can even show that both $C_1$ and $C_2$ are strongly secure. Apply the symmetry of $\equiv_L$ to get $\text{h:=h-1; l:=l+1} \equiv_L \text{h:=h+1; l:=l+1}$ and $\text{h:=h-1; l:=l+1} \equiv_L \text{h:=h+1; l:=l+1}$. By Definition 3, $C_1$ and $C_2$ are strongly secure.

Program $C_3$ is insecure because it reflects different program behavior for the low attacker depending on the initial value of $h$. Assume the initial states $s_1 = \{h=1, l=1\}$ and $s_2 = \{h=0, l=1\}$. $s_1$ and $s_2$ are low-equal. After execution of $C_3$, the resulting states are $s'_1 = \{h=1, l=1\}$ and $s'_2 = \{h=0, l=0\}$, which are not low-equal.

The strong low-bisimulation captures timing flows, as argued in [SS00]. A program with different timing behavior depending on high data is not strongly low-bisimilar to itself. Consider the following example:

```
if h>0 then skip; skip else skip
```

This example is not strongly secure because the execution time of the program depends on the high value $h$. In a concurrent setting, this program, which might seem secure intuitively, can result in a storage leak. This is demonstrated in Figure 3. It shows that timing leaks are an even more serious problem in multi-threaded programming languages than in purely sequential languages.

```
fork(C_4, C_5)
C_4: if h>0 then skip; skip else skip end; l:=1
C_5: skip; skip; l:=0
```

Figure 3: Example program where a timing leak induces a storage leak. Assume a round-robin-scheduler for dynamic thread execution. $h$ equal to 1 will result in $l:=1$ as command that is executed as the last one and $h$ equal to 0 results in $l:=0$ as last command. That is, the final value of $l$ reveals $h$, which constitutes an information leak.
Strong security features a number of compositionality properties. In particular, if a collection of commands is strongly secure then both the sequential and parallel composition of the commands are also strongly secure [Sab03]. This is essential for modular program analysis and simplifies checks of programs significantly. Another important property of strong security is its scheduler-independence — it is robust for any choice of a particular scheduler.

[SM03] surveys the research on information-flow security, particularly focused on static program analysis, and could be interesting for further reading.

Before presenting the concrete type systems as methods to check strong security, we point out some assumptions made in the operational semantics on page 9. They are important for the given definition of strong security and the transforming type system presented in the next subsection.

- Expressions evaluate atomically, no matter how complex they are. Evaluation takes a single unit of time.

- The execution times of basic commands are equal. In particular, skipping, assigning, and branching after the evaluation of the guard in a conditional are assumed to have the same execution time. The effect of this will become clear after having seen the transforming type system.

- Influence of caching on the timing behavior is disregarded. Caching affects how quickly a variable can be addressed. If caching influences the execution time, the program \( x := 2; \text{if secret then } z := x \text{ else } z := y \text{ end} \) executes slightly faster if secret is true instead of false (\( y \) does not reside in the cache and has to be fetched from memory first). Due to [AS01], the capacity of insecure timing flow through cache-leaks is not likely to be very high and there are methods for closing them.

### 2.3 Security Type Systems

It is rather tedious to prove the strong security condition for a given program [MS04]. Security type systems provide a method for checking the security. They can be used to automate the information flow analysis. Security type systems are a collection of typing rules that describe what security type is assigned to a program (or expression), depending on the security types of the subprograms (subexpressions). If the program has a type, i.e. if the defined rules can be applied so that all commands and the overall program can be given a type, the program is accepted. The program is called typable then. If it cannot be typed there are two possibilities: simple type systems reject it right away, transforming type systems specify transformation rules which might be applied and transform the insecure program into a secure one. If no such transformation rules can be applied, the program is rejected.

In the next subsections, we present two concrete security type systems that approximate the check of strong security. Section 2.3.1 states a type system without transformations and Section 2.3.2 a transforming type system. The transforming type system is taken from [SS00] in its original version. The reader familiar with it may want to skip Section 2.3.2. We have modified this transforming type system into the non-transforming type system of Section 2.3.1 in
order to work on systems of both kinds. The non-transforming type system is simpler but it rejects some insecure programs that can be transformed into secure ones by the transforming type system. Section 2.3.3 shows that all programs that are typable by at least one of the two security type systems are sound.

2.3.1 Security Type System without Transformation

This section presents a security type system without transformations. For accepting or rejecting a given program, we apply the security typing rules given in Figure 4. If these rules cover all commands of a program the program is typable and accepted, otherwise rejected.

The typing rules have the form hypotheses \( \vdash \) conclusion, just as the rules for the semantics. Multiple hypotheses separated by blanks are to be read as connected by ‘and’. \( \text{Exp} : \text{high} \) denotes that expression \( \text{Exp} \) has security domain \( \text{high} \). \( \vdash C \) denotes that command \( C \) is typable according to the rules. These assertions are known as typing judgments.

The general idea for applying the typing rules is the following: Security domains, either low or high, are initially assigned to all identifiers (rule Var). Expressions get a security domain through the application of the rules. Any expression can be typed high. It can only be typed low if all identifiers in it are of type low. The same applies for boolean expressions \( B \). The security domains of expressions are then used for applying the typing rules for commands.

Let us consider the particular rules for commands in more detail. A skip preserves information flow security. The Assign rules prevent direct insecure information flow because the assignment \( l := h \) is not typable. The guard in If \( \text{low} \) is low for ruling out indirect information flow in the branches. The same applies for While \( \text{low} \). Additionally, the low guard prevents timing and nontermination flow. The rules Seq, Par, and Fork determine the typability of commands compositionally.

Figure 5 shows how the typing rules are applied to an example program. The program from [SS00] is a modular exponentiation algorithm. RSA-encryption is based on computing \( a^b \mod k \), where \( a \) is the plain-text and \( b \) the encryption key. A careless implementation may leak \( b \) through timing. Figure 5 shows the inference tree for typing this program. It must be read from the bottom up. The symbol ‘?’ indicates that a command can be typed. ‘?’ is used for commands which could not be typed yet because the subcommands have to be checked first. By applying these typing rules the program cannot be typed. The problem is a high conditional for which no rule exists, which is denoted by ‘\( \not\vdash \)’.

There is no rule for conditionals with a high guard in this security type system. All programs with high variables in the guard of a conditional are rejected even if they are secure. At first sight it may seem that high conditionals are secure unless a branch contains an assignment to a low variable. However, recall the example in Figure 3 on page 12: If the timing behavior of a program depends on high data the scheduler may reflect this dependence on the values of low variables. Hence, another condition is necessary, which is more demanding to check: the execution times of the branches of a high conditional must be equal, which is beyond this simple security type system.
Var \[ h : \text{high} \quad l : \text{low} \]

Exp

\[ \begin{array}{c}
\text{number : low} \\
\text{Exp}_1 : \text{low} \quad \text{Exp}_2 : \text{low}
\end{array} \frac{\text{Exp}_1 \ 	ext{op} \ 	ext{Exp}_2 : \text{low}}{}
\]

\[ \begin{array}{c}
\text{Exp}_1 : \text{high} \\
\text{Exp}_2 : \text{high}
\end{array} \frac{\text{Exp}_1 \ 	ext{op} \ 	ext{Exp}_2 : \text{high}}{}
\]

Skip \[ \vdash \text{skip} \]

Assign_{\text{low}} \[ \vdash l := \text{Exp} \]

Assign_{\text{high}} \[ \vdash h := \text{Exp} \]

While_{\text{low}} \[ \vdash \text{while} \ B \ 	ext{do} \ C \]

\[ \vdash B : \text{low} \quad \vdash C \]

If_{\text{low}} \[ \vdash \text{if} \ B \ \text{then} \ C_1 \ \text{else} \ C_2 \]

\[ \vdash C_1 \quad \vdash C_2 \]

Seq \[ \vdash C_1 ; C_2 \]

\[ \vdash C_3 \quad \vdash C_2 \]

Fork \[ \vdash \text{fork}(C_1;C_2) \]

\[ \vdash C_0 \ldots \vdash C_{n-1} \]

Par \[ \vdash \langle C_0 \ldots C_{n-1} \rangle \]

Figure 4: Typing rules without transformations.
Program
\[
\begin{align*}
\text{c} & := 0; \\
\text{d} & := 1; \\
\text{i} & := \text{k}; \\
\text{while } \text{i} \geq 0 \text{ do} & \\
& \quad \text{i} := \text{i} - 1; \\
& \quad \text{c} := 2 \times \text{c}; \\
& \quad \text{d} := (\text{d} \times \text{d}) \text{ mod n}; \\
& \quad \text{if } \text{b}[\text{i}] = 1 \text{ then} \\
& \quad & \quad \text{c} := \text{c} + 1; \\
& \quad & \quad \text{d} := (\text{d} \times \text{a}) \text{ mod n}
\end{align*}
\]

Shortcuts
\[
\begin{align*}
\text{\text{A}} & = \text{Assign} \\
\text{\text{S}} & = \text{Seq} \\
\text{\text{W}} & = \text{While}
\end{align*}
\]

Figure 5: Sample inference tree without transformations. Modular exponentiation algorithm to compute $a^b \text{ mod } k$. $b$ is given as an array of secret booleans of length $k+1$, $b[k]$ is the most significant bit. For space reasons, the names of the applied rules are abbreviated: $A_l=\text{Assign}_{\text{low}}$, $A_h=\text{Assign}_{\text{high}}$, $S=\text{Seq}$, $W=\text{While}_{\text{low}}$. Assume the security domains of the variables to be as follows: $a: \text{high}$, $b: \text{high}$, $c: \text{high}$, $d: \text{high}$, $i: \text{low}$, $k: \text{low}$. 
2 Preliminaries

2.3.2 Security Type System with Transformation

This section presents another security type system, which originates from [SS00]. It goes beyond the one of the previous section in that it eliminates a certain kind of timing leak. It does not accept or reject a given program, but it constructs a program with secure information flow, whose behavior is very similar to the original insecure program. If such a transformation can be found then the resulting program is accepted as strongly secure.

The typing rules of the transforming system are presented in Figure 6. All typing rules for commands are transformation rules. The judgments have the form $\overline{C} \rightsquigarrow \overline{C}' : \overline{S}_l$, where $\overline{C}$ is a program, $\overline{C}'$ is the result of the transformation, and $\overline{S}_l$ is the type of $\overline{C}'$. Here, the type of a program, also denoted as its low slice, is basically a copy of a secure program in which assignments to high variables have been replaced by skip’s. $\overline{S}_l$ does not contain any occurrences of high variables, but it has the same structure as $\overline{C}'$ and models the same timing behavior.

For all rules but If\_high, the transformed program is constructed by composing the constructs the original program consists of. The typing records the information about the low slice of the new program. skip is its own transformation and its own type. The same holds for Assign\_low. The rule Assign\_high types an assignment to a high variable with the low slice skip. The rules Seq, If\_low, Par, and Fork propagate types compositionally.

The most interesting rule is If\_high. It prevents indirect insecure flow as well as timing flows. $al(C)$ is a boolean function indicating whether an assignment to a low variable occurs in program $C$. The condition $al(Sl_1) = al(Sl_2) = False$ prevents indirect leaks. The rule If\_high states that both branches of a high conditional must be typable and they must be low-bisimilar. If they are typable, the branches can be made low-bisimilar by cross-copying the low slice of one branch into the other. The low slice of the overall command is the sequential composition of the slices of the branches prefixed with a skip corresponding to the time to inspect the guard of the conditional.

Figure 7 demonstrates the application of these typing rules to an example program. The program is the same as the one rejected by the non-transforming type system: the modular exponentiation algorithm. Now, a rule for high conditionals exists and can be applied, transforming the original conditional with one branch into a conditional with two branches of the same low-observable behavior. The transformed program is also given in Figure 7.

Hence, compared to the non-transforming type system of the previous section, this transforming type system additionally allows high conditionals, as long as there is no indirect information flow and the observer cannot distinguish by the timing behavior which branch has been taken. A drawback is that, in some cases, secure programs are transformed although they are secure already [KM04]. Consider for instance the program if $h>0$ then $h:=h+1$ else $h:=h-1$ which is transformed into if $h>0$ then $h:=h+1$;skip else skip; $h:=h-1$. Every transformable program containing a high conditional results in a longer program, the code is blown up by skips. The program grows if the transformations are applied several times.
Figure 6: Typing rules with transformation to eliminate timing leaks.
Figure 7: Sample inference tree with transformations. Modular exponentiation algorithm to compute $a^b \mod k$. For space reasons, the first three (secure) assignments are skipped in the tree and the names of the applied rules are abbreviated: $A_l=\text{Assign}_{\text{low}}, A_h=\text{Assign}_{\text{high}}, I_h=\text{If}_{\text{high}}, S=\text{Seq}, W=\text{While}_{\text{low}}$. Assume the security domains of the variables to be as follows: $a$: high, $b$: high, $c$: high, $d$: high, $i$: low, $k$: low.
2.3.3 Soundness

Both presented security type systems are sound. In this section, we define what soundness means for each type system and we give proofs to show that the presented type systems are actually sound.

The non-transforming type system is sound if any type-correct program is strongly secure. No program must be accepted that may leak information. The transforming type system is sound if every program that results from the transformation of an input program is strongly secure.

Now, we state the soundness of the type systems as theorems.

**Theorem 1 (Soundness of the transforming type system).** All transformed programs resulting from the transforming type system are strongly secure.

The proof of Theorem 1 is given in [SS00].

We can state a proposition concerning the relation between the original and the transformed program in the transforming type system. It will be used below for proving that the non-transforming type system is sound.

**Proposition 1.** If \( C \leftrightarrow C' : S \) and if \( C \) does not contain any high conditional then \( C' = C \).

If we disregard rule \( \text{If}_{high} \), then the transformed program is identical to the original program.

**Proof (Sketch).** Induction over the derivation tree of the judgment \( C \leftrightarrow C' : S \). For the basic rules Skip, Assign_{low}, and Assign_{high}, the transformed command is identical. With the induction hypothesis, the rules for commands composed of subcommands (If_{low}, While_{low}, Seq, Fork, and Par) construct identical commands as well.

Now we have all ingredients to state and prove the soundness theorem for the non-transforming type system.

**Theorem 2 (Soundness of the non-transforming type system).** All programs accepted by the non-transforming type system are strongly secure.

**Proof.** All programs \( C \) that are typable by the non-transforming type system (i.e. \( \vdash C \)) do not contain any high conditional because there is no rule \( \text{If}_{high} \). \( C \) can be transformed into a secure program \( C' \) by the transforming type system: \( C \leftrightarrow C' : S \). By Theorem 1, \( C' \) is strongly secure. Then, by Proposition 1, \( C' = C \), and thus, \( C \) is strongly secure.

**Remark.** Soundness of the transforming security type system states that the transformed program is strongly secure. Besides that, transforming type systems have a second main objective: The semantics of the original program shall be preserved. The transformation changes the timing behavior, but it can be argued why the transformed program is equal enough to the original program under certain assumptions, which is done in [SS00].
3 Design of the Security Analyzer

We have developed a security analyzer for automatically checking the security of a program in terms of information flow. The first step in the development is the design phase. The system architecture was basically predefined by the thesis task and is presented in Section 3.1. Section 3.2 gives an overview over the class model of the security analyzer. The remaining Sections 3.3–3.6 go more into the conceptional details of its components.

3.1 System Architecture

Analyzing the security of programs requires certain conceptional steps. Consequently, the analyzing process can be split into several phases.

First of all, a data structure is required for storing the program. The contents in this data structure are the input for the security analyzer.

Second, in order to fill the data structure, a parser for MWL programs is needed. It directly operates on the source code, i.e. on the MWL text file.

Third, the actual analyzer, which outputs the result of the security check, is required. A transformer, which modifies an insecure program into a secure one, can be part of the analyzer.

Figure 8 depicts the process of checking the security using these elements.

To summarize, the security analyzer consists of the following components:

1. Data structure
2. Parser
3. Analyzer
   • without transformation
   • with transformation

Figure 8: The analyzing process.
3.2 Overall Class Model

As addressed before, the system consists of a parser, a data structure to store the program, and the actual analyzer. We chose an object-oriented approach for the realization. Consequently, each system component is composed of a collection of classes and packages. An overview over the most important classes is given in Figure 9.

The representation of the class model complies with the UML standard, in the following simplified version: Class names in italic (Checker) represent abstract classes or interfaces and class names in upright font (Program) represent concrete classes. Solid lines with a triangle stand for an inheritance relationship and dashed lines symbolize usage (e.g. Checker initializes and calls MWLParser, but MWLParser does not know anything about Checker).

The three components of the security analyzer are in more detail:

- The data structure consists of classes representing a program, such as commands, expressions, operators, etc. It is presented in Section 3.3.
- The parser consists of two classes, the scanner and the parser itself. More details are provided in Section 3.4.
- The analyzer is composed of the classes containing the logic for the analysis. The visitor design pattern is applied for the implementation of the logic, see Sections 3.5 and 3.6.

The classes for the data structure and the analyzer component are not listed completely in Figure 9. The classes for the analyzing process constitute the largest component, not in the number of classes (more classes are needed for the data structure) but in terms of logic and lines of code.

Figure 9: Class model of the classes of the security analyzer. Boxes with dots indicate that only the most important classes are depicted.
3.3 Representation of Programs

We need a data structure that represents and stores a program. An abstract syntax tree (AST) is suitable for this purpose, because it captures the essential structure of the input program while omitting unnecessary syntactic details.

The hierarchy and the relationships of the classes implementing the abstract syntax tree can be found in Figure 11.

The main top elements are *MWLElement* and *Exp*:

- *MWLElement* stands for either a command, a program or a vector of programs.
- *Exp* represents an expression, which may be an identifier, an integer, an arithmetic expression, a truth value or a boolean expression.

The class *Program* represents an MWL program: It consists of the first command and the rest of the program (see Figure 10). This implicitly implements the sequential composition of commands, which states that a command may consist of two commands separated by a semicolon (`C ::= C1; C2 | ...`).

The granularity of the representation of the classes for the abstract syntax tree has been chosen in such a way that all constructs of the source language are represented as different classes instead of representing some constructs by a common AST class node and differentiate them using a value. This results in a class *True* for boolean value `true` and another class *False* for `false` (which have a common abstract super class `TruthValue`), instead of representing both of them by a class `TruthValue` and differentiate using an internal variable. The reason for this decision is the possible integration of behavior associated with tree nodes and the abstraction from the representation in the node. Otherwise, one would have to commit oneself to a specific representation: Should the security type `high` be represented by storing an integer, a string or something else in a security domain node?

All AST classes implement a `clone` method, which facilitates working on copies of a program.

```
Program
| comm: Command |
| prog: Program |
| +getCommand(): Command |
| +getProgram(): Program |
| ...() |
```

Figure 10: The class *Program*: UML representation (left) and tree representation (right).
Figure 11: Class diagram for representing abstract syntax trees.
3 Design of the Security Analyzer

3.4 Parser

A parser is suitable for the process of filling the AST data structure described in the previous section. The parser can be hand-written or produced by a parser generating tool. We decided to generate the parser because it was faster.

The task of parsing an input file, which means discovering the source structure, can be decomposed into the following subtasks:

1. Split the source file into tokens
2. Find the hierarchical structure of the program

The first subtask is processed by a lexical analyzer or scanner. The second one requires a syntax analyzer or parser, see Figure 12.

More details of the parser are provided in the implementation section (4.2).

![Figure 12: From the source code to the abstract syntax tree.](image)

3.5 Security Type Systems Used

The logic of the analyzer for checking input programs comes from the implemented security typing rules. As argued in Section 1.2, it makes sense to realize both a non-transforming and a transforming type system.

As non-transforming type system, we have implemented the security typing rules described in Section 2.3.1. They either accept or reject a given program, i.e. the result of the check is boolean.

As transforming type system, we have implemented the security type system from Section 2.3.2. Its more sophisticated typing rules do not accept or reject a given program, but transform it into a secure program, if this is possible.

The analyzer must be designed in such a way that it can handle both kinds of security type systems. This is addressed in the following subsection.

3.6 Design Pattern

The MWL program is read into the data structure by the parser. Then the analyzer must operate on it. We use a design pattern for realizing the analyzer.
The **visitor design pattern** is suited well for this task, in that it provides a way of separating the logic from the data structure and supports clarity of code.

In the following, we first introduce the visitor design pattern on an abstract level. Then we illustrate the application of this pattern for the security analyzer.

The visitor design pattern is used to separate the functionality from an object structure. This pattern allows to decouple the classes of the structure and the algorithms used upon them. The operation which is to be performed on the elements of an object structure is encapsulated as one object. Hence, the visitor design pattern enables defining new operations without modifying the elements of the object structure [GHJV95].

As shown in Figure 13, the basic idea of the visitor pattern is that each element class from an object structure has an accept method that takes a visitor object as argument. The visitor is an interface that has a visit method for each element class. The accept method of an element class calls back the visit method for its own class, passing itself to the visitor \( v.\text{visit}(\text{this}) \). Concrete visitor classes can then be written to perform particular operations on the elements.

![Class diagram of the classes involved in the visitor pattern.](image)

The visitor design pattern can be used for implementing the analyzer in an elegant way. The abstract syntax tree classes correspond to the object structure. All AST classes implement a method `accept` which calls the `visit` method of the passed visitor.

Concrete visitor implementations must realize several functionalities for the purpose of analyzing the security of programs. As stated in Section 3.5, the analyzer requires an encapsulated logic for analyzing a program without transformations and another logic for transforming it. Two concrete visitors realize these tasks. Both of them need to determine the security domains of expressions. Hence,
this functionality has to be implemented as well. We do this in a third visitor. Moreover, it is necessary to transform the representation of the (transformed) program in the data structure into its string representation in order to store the transformed program in a text file. This is is implemented in a forth visitor.

These four implementations of visitors have a common abstract super class. Figure 14 shows their inheritance relationship. Method overloading is used for determining the correct visit method at runtime.

The following listing explains the functionalities of the visitor classes.

- Abstract class AbstractVisitor provides default visit methods for all visitor implementations by simply returning null. This frees the concrete subclasses from implementing visit methods for all AST constructs, even those they are not concerned with (e.g. ExpVisitor is only concerned with expressions and does not implement methods for commands).

- NoTransformVisitor implements the logic for checking the information flow without transformations. (See Section 4.3)

- TransformVisitor implements the logic for the security check with transformations. It potentially creates a transformed program. (See Section 4.4)

- ExpVisitor implements the logic for typing all kinds of expressions. It determines their security domains. This information is needed by both the TransformVisitor and the NoTransformVisitor.

- ToStringVisitor produces a string representation of the program, which can be used for printing a transformed program into a file. The string is indented and contains line breaks.

![Figure 14: Class diagram of the visitor classes.](image-url)
Design of the Security Analyzer
# 4 Implementation

This section focuses on the implementation of the security analyzer. Java was prescribed as implementation language by the thesis proposal. Due to its practicability for mappings and support by other tools Java is an appropriate choice. First of all, Section 4.1 modifies and refines the core MWL language for our purposes.

As described in Section 3, the checking process can be split into several phases. The following sections give further details about the implementation of the parser (4.2) and the analyzers (without transformation: 4.3, with transformation: 4.4). Section 4.5 presents the output of the security analyzer. The final section 4.6 describes open implementation issues.

All components of the system have been built from scratch unless stated otherwise. Parts of the source code can be found in Appendix C. Instructions for installing and running the security analyzer can be found in the user documentation in Appendix B.

## 4.1 Adaptation of MWL

Writing programs in MWL and analyzing their security based on information flow requires some modifications and refinements of the core MWL (Section 2.1):

- Variable declarations are introduced, specifying the type and the security domain of a variable. Variables are declared globally in a declaration section at the beginning of the program. Each variable is declared only once.
- The dangling-else-problem is solved by finishing conditionals with `end`. Likewise, `end` is also appended to while-loops to eliminate ambiguity.
- For conditionals, the else term is optional.

This results in the following refined grammar, which is the basis for all further considerations. $P$ stands for the overall program, $Decls$ for the global variable declarations, and $C$ for the commands, which may be sequentially composed to make up the actual program.

\[
\begin{align*}
P & ::= Decls; C \mid C \\
Decls & ::= Id:\text{Type};\text{SecDom} \mid Id:\text{Type};\text{SecDom}; Decls \\
C & ::= \text{skip} \mid Id:="Exp" \mid \text{if } B \text{ then } C_1 \text{ else } C_2 \text{ end} \mid \text{if } B \text{ then } C \text{ end} \\
& \quad \mid \text{while } B \text{ do } C \text{ end} \mid C_1;C_2 \mid \text{fork}(C <D_1,\ldots,D_n>)
\end{align*}
\]

$Id$ is any alphanumeric string beginning with a letter.

Types and security domains are defined as

\[
\begin{align*}
\text{Type} & ::= \text{int} \mid \text{bool} \\
\text{SecDom} & ::= \text{high} \mid \text{low}
\end{align*}
\]

Arithmetic expressions may contain arithmetic operators, boolean expressions may contain boolean operators. The operators are defined as follows:

<table>
<thead>
<tr>
<th>Arithmetic operators:</th>
<th>Boolean operators:</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>op</code> ::= <code>+</code></td>
<td><code>-</code></td>
</tr>
</tbody>
</table>
| `boolOp` ::= `<` | `<=` | `>` | `==` | `!=` | `&` | `|`

i: int: low;
remainder: int: high;
check: bool: low;
result: bool: high;

i:=1;
check:=true;
while check==true do
  i:=i+1;
  remainder:=n mod i;
  if remainder==0 then check:=false end;
  if i>=n then check:=false end
end;
if i==n then result:=true else result:=false end

Figure 15: Sample MWL program, determining whether a secret number $n$ is prime. Assume $n$ to be given with the declaration $n:int:high$. The idea of this simple algorithm is to check every number less than $n$ if it is a divisor of $n$.

The sequential composition token ';;' is considered as separator of commands. Hence, the last command of a program does not end with ';'.

An MWL example program according to the refined grammar can be found in Figure 15.

Any valid MWL program can be stored in the data structure presented in Section 3.3. But not all programs which can be stored in this data structure are actually meaningful and valid MWL programs. As an example consider the program $res:=true+false$: it can be stored, but it makes no sense because it is not well-typed. Rejecting programs that are not well-typed and providing suggestions for improvements lies in the scope of duty of a compiler, which is not part of this thesis.

All programs that are analyzed with the security analyzer must fulfill the following precondition:

Input programs are assumed to be well-typed.

This includes e.g. that the guards of conditionals and loops are of type boolean, that arithmetic operators only operate on integers and that all variables are declared (in the declaration section, every variable exactly once).

4.2 Parser

Lex and Yacc are typical scanner and parser generating tools. As the security analyzer is implemented in Java, using the Java versions of these tools, JLex as lexical analyzer and Java CUP as syntax analyzer,1 suggested itself (recall Figure 12 for the parsing process).

1Used versions: JLex 1.2.6 and CUP v0.10k. See [Ber00] and [Hud99].
4 Implementation

";" { return (new Symbol(sym.SEMI)); }
"+" { return (new Symbol(sym.PLUS)); }
"mod" { return (new Symbol(sym.MOD)); }
":=" { return (new Symbol(sym.ASSIGN)); }
"skip" { return (new Symbol(sym.SKIP)); }
"if" { return (new Symbol(sym.IF)); }
...

Figure 16: Grammar file excerpt for generating the scanner with JLex.

terminal SEMI, PLUS, MOD, ASSIGN, SKIP;
non terminal command, identifier, expression;
command ::= SKIP {: RESULT = new Skip(); :} |
     identifier:i ASSIGN expression:e
     {: RESULT = new Assign((Identifier)i, (Exp)e); :} |
     ...
...

Figure 17: Grammar file excerpt for generating the parser with Java CUP.

Grammar files, which define the structure of the text or file to be processed, must be created as input for both generating tools.

The grammar file for JLex simply lists the set of character sequences that may occur and defines their mappings to tokens. Figure 16 shows an excerpt of the grammar file.

The grammar file for Java CUP determines the structure of an MWL program. While parsing the source file the generated parser fills the data structure as defined in this grammar file by using the tokens. It calls the scanner directly to get the tokens from it. An excerpt of the grammar file for the parser is shown in Figure 17. When a certain combination of tokens is encountered, the Java code in braces is executed, creating instances of the classes of the abstract syntax tree.

The complete grammar files can be found in Appendix C.1.

4.3 Analyzer without Transformation

As described in the design section (3) the analyzer consists of two parts. The first one does not transform the input program. The class NoTransformVisitor realizes the logic for checking the information flow without transformations based on the rules from Section 2.3.1. For the implementation, we apply the visitor design pattern.

First, we describe the process flow. The general procedure for applying the NoTransformVisitor is as follows: An instance of CheckerNoTransform calls the parser to parse the program, which returns the MWL program in the AST data structure as a Program p. Then, CheckerNoTransform creates a new visitor v, in this case a NoTransformVisitor, and invokes its execution by calling p.accept(v) (which calls v.visit(this) on its part). The result of the check can be retrieved
by calling `getResult()` of the visitor. See Figure 18 for the class interaction to understand the following source code excerpt:

In `CheckerNoTransform`:

```java
Scanner scanner = new MWLScanner(filename);
Program p = (Program) MWLParser(scanner).parse().value;
Visitor v = new NoTransformVisitor();
p.accept(v);
boolean result = ((Boolean) v.getResult()).booleanValue();
```

Figure 18: Class interaction for checking a program without transformations

Now we will clarify the basic idea of the logic that is implemented in the class `NoTransformVisitor`. In the beginning, the program is assumed to be secure. The internal result variable, which is boolean, is set to true. For each encountered command, `NoTransformVisitor` tries to apply the typing rules. The result is set to false if the command is insecure according to the rules, i.e. if no rule can be applied. If there is a rule that can be applied the internal result remains unmodified. Thus, once the result is false, it remains false until the end of the execution and can be retrieved by the `getResult` method.

In the case that the input program is insecure, the analyzer determines the first untypable command and outputs it on the command line.

Figure 19 shows the implementation of the rules Skip and While. More implementation details can be found in the source code in Appendix C.

```java
public Object visit (Skip s) { return null; }
public Object visit (While wh) {
    ExpVisitor boolV = new ExpVisitor(decls);
    wh.getBoolExp().accept(boolV);
    int sdBool = ((Integer) boolV.getResult()).intValue();
    if (sdBool == HIGH) {
        setFound("The first insecure command is a high while loop:\n"+wh);
        checkResult = false;
    }
    wh.getProgram().accept(this);
    return null;
}
```

Figure 19: Source code excerpt of the class `NoTransformVisitor`. 
4 Implementation

4.4 Analyzer with Transformation

The class TransformVisitor implements the logic for checking the security with transformations based on the rules from Section 2.3.2. The flow of process for calling it corresponds to the one of the NoTransformVisitor. Its getResult method can be called to get a boolean indicating the information flow security as for the NoTransformVisitor. Furthermore, the invocation p.accept(v) from the client method returns the transformed program\(^2\): \(\text{Program transfProg} = p.\text{accept}(v)\).

In the typing rules of Section 2.3.2, commands are decomposed into their subcommands in the hypotheses (e.g. a skip has no subcommand, but a conditional with two branches has two subcommands). For each subcommand the transformation needs three elements as hypotheses to yield the forth element, the transformed command, as a consequence. The three required elements are (for each subcommand \(C\)):

1. the transformed subcommand \(C'\) (of class MWLElement);
2. the type \(Sl\) of the transformed subcommand (of class MWLElement);
3. the information if there is an assignment to a low variable in \(Sl\) (as a boolean), which is needed by the If\(_{high}\) rule.

As an example, recall rule While\(_{low}\):

\[
\begin{align*}
B: \text{low} & \quad C \rightarrow C' : Sl \\
\text{While}_{low} & \quad \text{while } B \text{ do } C \text{ end} \rightarrow \text{while } B \text{ do } C' \text{ end} : \text{while } B \text{ do } Sl \text{ end}
\end{align*}
\]

The command is \(\text{while } B \text{ do } C \text{ end}\). It has one subcommand: \(C\). The first element of the judgment for the subcommand is the transformed subcommand: \(C'\). The second element is the type of the transformed subcommand: \(Sl\). Finally, the third element is the information whether there is an assignment to a low variable in \(Sl\), which is is determined recursively (this is only needed for rule If\(_{high}\)).

Hence, a class, called ElementPair, is implemented to represent this information. It consists of two MWLElements (the transformed subcommand and its low slice) and a boolean indicating whether there is an assignment to a low variable in the low slice. Instances of this class are returned by the visit methods and then used by the caller to construct the transformed program. As an example, see Figure 20 for the source code of the transformation of a skip command.

```java
public Object visit (Skip s) {
    Skip sNew = (Skip) s.clone();
    Skip sNewType = (Skip) s.clone();
    return new ElementPair(sNew, sNewType, false);
} ...
```

Figure 20: Source code excerpt of the class TransformVisitor.

Imagine, that the transformation analyzer tests an insecure program that can be transformed into a secure one. The transformed program can then be accessed by the first member variable MWLElement in the returned instance of

\(^2\)For this reason all accept and visit methods return an Object, which is ignored when the TransformVisitor, StringVisitor or ExpVisitor are called.
ElementPair. Calling ToStringVisitor on this transformed program results in a string. This string is then written into a second text file stored in the same directory as the input program. It has the same name, only \_secure is appended to the file name. See Figure 21 for a code fragment from CheckerTransform.

More implementation details can be found in the source code in Appendix C.

```java
Visitor v = new TransformVisitor();
ElementPair ep = (ElementPair) p.accept(visitor);
Program transfProg = (Program) ep.getElement();
boolean result = ((Boolean) v.getResult()).booleanValue();
if (result == true) {
  Visitor stringVis = new ToStringVisitor();
  transfProg.accept(stringVis);
  String s = (String) stringVis.getResult();
  FileOutputStream out = new FileOutputStream(filename+_secure);
  (new PrintStream(out)).println(s);
}
```

Figure 21: Source code excerpt of CheckerTransform.

**Remark.** A first version of the TransformVisitor did not use the return value of the visit methods to create the new program, but modified the original program using two further visitors:

- **AlVisitor** returns whether a command, program, or program vector contains an assignment to a low variable;
- **TypeDeterminer** determines the type of a program, program vector, or command.

This demonstrates a different way of solving the task. However, this first solution has several drawbacks: the program in the data structure is modified directly and the tree is traversed three times because of three visitors. The new implementation of the TransformVisitor only traverses the tree once to transform the program and hence decreases running time and improves performance. (To be precise, it is then traversed again by ToStringVisitor for writing it into a file.)

### 4.5 Output

If the input program can be typed, the security analyzer outputs that the program is secure: $$\Rightarrow$$ The program secure.mwl is secure.

If the input program cannot be typed in its original version, the analyzer provides information about where the first problem lies. Then, it tries to transform the program. If the transformation is successful a sample output could be:

The first insecure command is a high conditional:

```java
if h==value then result:=i
$$\Rightarrow$$ The program almostSecure.mwl is NOT secure.
$$\Rightarrow$$ The program almostSecure.mwl can be transformed into a secure program.
The transformed program can be found in almostSecure.mwl_secure.
```

If the program is not secure in its original version and cannot be repaired by the transforming type system, the analyzer rejects the program:
The first insecure command is a high conditional:
if h==value then l:=i

===> The program insecure.mwl is NOT secure.
===> The program insecure.mwl can NOT be transformed into a secure program.

4.6 Open Issues

This is a list of open issues that could be addressed by further implementation work.

On the one hand, the language MWL could be extended in order to widen its expressiveness to a broader class of programs. This involves extending the AST data structure, the parser, and the analyzers. It could be desirable to have the following extensions:

- further data types (like strings, float numbers, records);
- more language constructs such as procedures and input/output;
- the language could be enriched by local bindings limiting the scope of a variable, protect statements guaranteeing the atomic execution of commands, and synchronization constructs.

On the other hand, it the logic of the security check could be extended in order to accept more secure programs:

- More sophisticated typing rules and their implementations could improve the preciseness of the analyzer. There are a number of programs which are obviously secure, but are rejected by the current implementation, such as (h:=0;l:=h), (l:=h-h), or (h:=0; if h!=0 then l:=1 end). Extending the analyzer in order to accept these strongly secure but untypable programs requires theoretical explorations first.

- In order to find more transformations the security analyzer might interact with the user. It is conceivable that the analyzer provides more information about the core of the problem and the user can reply to that with suggestions to improve the program.

Furthermore, the assumption behind the patches with dummy computations is that assignments of arbitrary complex expressions as well as inspecting and branching in a conditional take as long as the execution of a skip command, as pointed out on page 13. To approach reality, expressions could be unrolled (only one operation at a time) and two skip commands with different timing behavior could be introduced as replacement for assignments and for branches, as suggested in [Aga00].
5 Type System Extension: Variable Declarations and Arrays

There is a shortage of suitable examples that can be checked with the implemented type systems. [AS01] provides examples, but they are designed for a higher programming language because they involve bit-operations, pointers, arrays, functions, etc. When we were looking for further examples for our simple language MWL, it became evident that the language had to be extended. Integrating arrays into MWL enabled us to find examples. Arrays store values with equal data types in one data structure, providing the possibility of traversing, searching, and sorting.

Before analyzing arrays, this section starts with the analysis of variable declarations. Commands for variable declarations are lacking in [SS00] because it does not distinguish different data types and retrieves the security domains for variables implicitly from the context. For declaring variables with different types and labeling them explicitly with security domains, variable declarations have been introduced as commands in the adaptation of MWL. Semantics and typing rules for variable declarations are elaborated in Section 5.1.

Extending the system for arrays involves defining the syntax, semantics, and typing rules for array expressions and commands. For analyzing array expressions, syntax and semantics for general expressions are defined, which are not stated explicitly in [SS00]. They are elaborated in Section 5.2, as preliminary for the extension by arrays.

Design decisions, syntax, semantics, and typing rules for arrays are given in Section 5.3. Moreover, the resulting security type system is proven to be sound.

5.1 Variable Declarations

This section defines the semantics and typing rules of variable declarations. Their syntax has been defined in Section 4.1. No arrays are involved so far.

**Syntax.** As defined in the adaptation of MWL, the command for the variable declaration of identifier \( \text{Id} \) declares its data type \( \text{Type} \in \{\text{int}, \text{bool}\} \) and the security domain \( \text{SecDom} \in \{\text{low, high}\} \). It has the following syntax:

\[
\text{Id:Type:SecDom}
\]

**Semantics.** We define that the declaration of a variable leads to the initialization of this variable in memory:

\[
\{v:\text{int:SecDom},s\} \rightarrow \{\},[v = 0]s \quad \text{and} \quad \{b:\text{bool:SecDom},s\} \rightarrow \{\},[b = \text{false}]s
\]

**Typing Rules.** A typing rule has to be defined for this additional command. The data type is of no interest for the typing rules, but the security domain is needed: Variable declarations change the context in which a program is executed. For our purposes, the security context is a set that contains the security domains of all declared variables. It is denoted by \( \Gamma \) and constructed during
the type check, i.e., while the typing rules are applied. $\Gamma$ is a set of (identifier/security domain)-pairs. Consequently, set operations can be performed to operate on $\Gamma$.

For the non-transforming type system, we define the rule for variable declarations as follows (where $\_\_\_\_$ is a place holder for the concrete security domain and the judgment $\Gamma \vdash C$ indicates that command $C$ is typable under security context $\Gamma$):

$$
\text{Id} : \_\_\_\_ \notin \Gamma
\quad \Rightarrow \quad
\Gamma \cup \{\text{Id} : \text{SecDom}\} \vdash \text{Id} : \text{SecDom}
$$

This command can be read as: Unless identifier $\text{Id}$ is already defined, the security domain of $\text{Id}$ is inserted into the security context and the overall command is typable.

The security context is integrated into the typing rules. This results in the extended security type system of Figure 22. The security context is required for determining the security domain of expressions: $\Gamma(\text{Exp})$ stands for the security domain of $\text{Exp}$ under security context $\Gamma$. The typing rules for commands use the security context and pass it on.

Before we consider the particular rules in more detail, it is important to stress that programs are assumed to adhere to the program structure presented in Section 4.1: Variable declarations only occur in the declaration section before the actual program starts. When the logic of the program begins every variable has been inserted into the security context.

The rules Skip and Assign do not modify the security context because no variable declarations occur in these commands (see Figure 22). Since variables are declared before the actual program, the rules If, While, Fork, and Par do not modify the security context either, they just pass it on. The interesting rule is Seq. Variable declarations may occur in the subcommands of the sequential composition, namely in the declaration section. The declarations in the security context of the first subcommand apply also for the second subcommand. The unification of the security contexts of both subcommands, which equals the security context of the overall command.

So far, we have explained the typing rules of the non-transforming type system. The type system with transformations is extended as well by variable declarations and the security context. Figure 23 shows the extended typing rules. Expressions are treated equivalently to the non-transforming system. After the considerations for the non-transforming type system, the modification of most of the rules is straightforward. An important observation is, however, that the transformed commands have the same security context as the original commands.

The fact that variables are not declared within the program is of particular relevance for the If*$_{high}$* rule. If it contained a variable declaration, the type of the transformed declaration would be padded into the other branch. Then, several questions would arise: Should the type be a skip (giving variable declarations one step to execute), the empty command (without execution time), or should it be identical to the declaration? What if a variable declaration is padded into
Figure 22: Extended non-transforming rules with variable declarations and security context $\Gamma$. $\Gamma(Exp)$ denotes the security domain of expression $Exp$ under $\Gamma$. 

\[
\begin{align*}
\text{Exp} & \quad \Gamma(\text{Const}) : l: low \quad \Gamma(l) : low \quad h : high \in \Gamma \\
& \quad \Gamma(\text{Exp}_1) : low \quad \Gamma(\text{Exp}_2) : low \\
& \quad \Gamma(\text{Exp}_1 \text{ op } \Gamma(\text{Exp}_2)) : low \\
& \quad \Gamma(\text{Exp}_1) : high \quad \Gamma(\text{Exp}_2) : high \\
& \quad \Gamma(\text{Exp}_1 \text{ op } \text{Exp}_2) : high \quad \Gamma(\text{Exp}_1 \text{ op } \text{Exp}_2) : high \\
& \quad \text{where \ } op \in \{+, -, *, \text{mod}, \text{div}\} \cup \{<, <=, >, >=, ==, !=\} \cup \{\&\& , ||\}
\end{align*}
\[
\begin{align*}
\text{B} & \quad \Gamma(\text{true}) : low \quad \Gamma(\text{false}) : low \\
& \quad Id : _\not\in \Gamma \\
\text{Var} & \quad \Gamma \cup \{Id : low\} \vdash Id : \text{Type}:low \quad \Gamma \cup \{Id : high\} \vdash Id : \text{Type}:high \\
& \quad \text{where Type} \in \{\text{int, bool}\}
\end{align*}
\[
\begin{align*}
\text{Skip} & \quad \Gamma \vdash \text{skip} \\
\text{Assign}_{\text{low}} & \quad \Gamma(l) : low \quad \Gamma(\text{Exp}) : low \\
& \quad \Gamma \vdash l := \text{Exp} \\
\text{Assign}_{\text{high}} & \quad \Gamma(h) : high \\
\text{If}_{\text{low}} & \quad \Gamma(B) : low \quad \Gamma \vdash C_1 \quad \Gamma \vdash C_2 \\
& \quad \Gamma \vdash \text{if } B \text{ then } C_1 \text{ else } C_2 \text{ end} \\
\text{While}_{\text{low}} & \quad \Gamma \vdash \text{while } B \text{ do } C \text{ end} \\
\text{Seq} & \quad \Gamma_1 \vdash C_1 \quad \Gamma_1 \cup \Gamma_2 \vdash C_2 \\
& \quad \Gamma_1 \cup \Gamma_2 \vdash C_1 ; C_2 \\
\text{Fork} & \quad \Gamma \vdash \text{fork}(C_1 \cdots C_2) \\
\text{Par} & \quad \Gamma \vdash (C_0 \cdots C_{n-1})
\end{align*}
\]
5 Type System Extension: Variable Declarations and Arrays

Assign

\[ \Gamma(l) : \text{low} \quad \Gamma(l) : \text{low} \quad \Gamma(h) : \text{high} \]

\[ \Gamma(Exp_1) : \text{low} \quad \Gamma(Exp_2) : \text{low} \]

\[ \Gamma(Exp_1) \text{ op } \Gamma(Exp_2) : \text{low} \]

\[ \Gamma(Exp_1) : \text{high} \quad \Gamma(Exp_2) : \text{high} \]

\[ \Gamma(Exp_1 \text{ op } Exp_2) : \text{high} \]

where \( \text{op} \in \{+, -, *, \text{mod}, \text{div}\} \cup \{<, >, >=, !=\} \cup \{&&, ||\} \)

Assign

\[ \Gamma(\text{true}) : \text{low} \quad \Gamma(\text{false}) : \text{low} \]

\[ \Gamma(l) : \text{low} \quad \Gamma(Exp) : \text{low} \]

Assign

\[ \Gamma(\text{true}) : \text{low} \quad \Gamma(\text{false}) : \text{low} \]

\[ \Gamma(l) : \text{low} \quad \Gamma(Exp) : \text{low} \]

While

\[ \Gamma(\text{true}) : \text{low} \quad \Gamma(\text{false}) : \text{low} \]

\[ \Gamma(\text{true}) : \text{low} \quad \Gamma(\text{false}) : \text{low} \]

\[ \Gamma(l) : \text{low} \quad \Gamma(Exp) : \text{low} \]

Assign

\[ \Gamma(l) : \text{low} \quad \Gamma(Exp) : \text{low} \]

Assign

\[ \Gamma(h) : \text{high} \]

\[ \Gamma(Exp) : \text{low} \]

While

\[ \Gamma(\text{true}) : \text{low} \quad \Gamma(\text{false}) : \text{low} \]

\[ \Gamma(l) : \text{low} \quad \Gamma(Exp) : \text{low} \]

Assign

\[ \Gamma(l) : \text{low} \quad \Gamma(Exp) : \text{low} \]

Assign

\[ \Gamma(h) : \text{high} \]

\[ \Gamma(Exp) : \text{low} \]

While

\[ \Gamma(\text{true}) : \text{low} \quad \Gamma(\text{false}) : \text{low} \]

\[ \Gamma(l) : \text{low} \quad \Gamma(Exp) : \text{low} \]

Assign

\[ \Gamma(l) : \text{low} \quad \Gamma(Exp) : \text{low} \]

Assign

\[ \Gamma(h) : \text{high} \]

\[ \Gamma(Exp) : \text{low} \]

While

\[ \Gamma(\text{true}) : \text{low} \quad \Gamma(\text{false}) : \text{low} \]

\[ \Gamma(l) : \text{low} \quad \Gamma(Exp) : \text{low} \]

Assign

\[ \Gamma(l) : \text{low} \quad \Gamma(Exp) : \text{low} \]

Assign

\[ \Gamma(h) : \text{high} \]

\[ \Gamma(Exp) : \text{low} \]

While

\[ \Gamma(\text{true}) : \text{low} \quad \Gamma(\text{false}) : \text{low} \]

\[ \Gamma(l) : \text{low} \quad \Gamma(Exp) : \text{low} \]

Assign

\[ \Gamma(l) : \text{low} \quad \Gamma(Exp) : \text{low} \]

Assign

\[ \Gamma(h) : \text{high} \]

\[ \Gamma(Exp) : \text{low} \]

While

\[ \Gamma(\text{true}) : \text{low} \quad \Gamma(\text{false}) : \text{low} \]

\[ \Gamma(l) : \text{low} \quad \Gamma(Exp) : \text{low} \]

Assign

\[ \Gamma(l) : \text{low} \quad \Gamma(Exp) : \text{low} \]

Assign

\[ \Gamma(h) : \text{high} \]

\[ \Gamma(Exp) : \text{low} \]

Figure 23: Extended transformation rules with variable declarations and the security context \( \Gamma \). \( \Gamma(Exp) \) denotes the security domain of expression \( Exp \) under \( \Gamma \).
a branch of a high conditional where a variable with this name already exists? These questions do not arise if variables are global and declared before the start of the program. Then, the type of the declaration command, i.e. its low slice, is not important because it is never used for transforming a program.

After these considerations, we decide that the type of a (low) variable declaration is identical to the declaration itself:

\[ \text{Id} : \_ \notin \Gamma \]
\[ \Gamma \cup \{\text{Id}:\text{low}\} \vdash \text{Id}:\text{Type}:\text{low} \leftrightarrow \text{Id}:\text{Type}:\text{low} \]

To summarize, the security type system from [SS00] has been extended by explicit variable declarations. A security context has been introduced to deal with the additional information from variable declarations and to make this information accessible for the typing rules of the other commands. The soundness of this extended type system will be proven in Section 5.3.3, after having extended it further for arrays.

### 5.2 Expressions

As preliminary to the analysis of array expressions, we formalize expressions in general.

**Syntax.** The syntax for expressions in the adapted version of MWL can be found in Figure 24.

\[
\text{Exp} ::= \text{Const} | \text{Id} | \text{Exp} + \text{Exp} | \text{Exp} - \text{Exp} | \text{Exp} \cdot \text{Exp} \\
| \text{Exp} \text{mod} \text{Exp} | \text{Exp} \text{div} \text{Exp} | B \\
\]

\[
B ::= \text{true} | \text{false} | \text{Id} | B \text{ && } B | B \text{ || } B | \text{Exp} < \text{Exp} | \text{Exp} \leq \text{Exp} \\
| \text{Exp} > \text{Exp} | \text{Exp} \geq \text{Exp} | \text{Exp} = \text{Exp} | \text{Exp} 
eq \text{Exp} \\
\]

Figure 24: Syntax for expressions.

**Semantics.** Figure 25 shows the semantics, i.e. how the expressions are evaluated.

**Typing Rules.** The typing rules for these expressions have already been integrated into the extended type systems of Figures 22 and 23. Integer constants Const are defined to be low; so are the boolean constants true and false. An identifier Id is of its declared security domain. Composed expressions are low if all variables in it are low, they are false otherwise.

The following theorem formalizes the intuition, that the values of low expressions can be observed by the attacker and consequently must evaluate to the same value in all low-equal states. We will get back to that when considering arrays.

**Theorem 3 (Low Expressions).** If an expression Exp is low then

\[ \forall s_1, s_2, n. s_1 =_L s_2 \Rightarrow \{\{\text{Exp}, s_1\} \downarrow n \Leftrightarrow \{\text{Exp}, s_2\} \downarrow n\}. \]

**Proof.** By induction over the structure of expressions. The theorem is proven by showing its correctness for all cases of expressions. Expression semantics is deterministic.
1. \textit{Id}: \( s_1 = L. s_2 \) from the assumption, i.e. the two states agree in all variables and in particular in \textit{Id}.

2. \textit{Const}: Constants evaluate to the same value in all states, they are the value actually. So they also evaluate to the same \( n \) in all low-equal \( s_1, s_2 \).

3. \textit{TruthValue} \( \in \{\text{true, false}\} \): Same reasoning as for case \textit{Const}.

4. \textit{Exp op Exp}: Rename the first expression to \textit{Exp}1, the second to \textit{Exp}2 by way of distinction. From the induction hypothesis it follows that if \textit{Exp}1 is low then \( \forall s_1, s_2, n_1, s_1 = L. s_2 \Rightarrow (\{\textit{Exp}_1, s_1\} \downarrow n_1 \Leftrightarrow \{\textit{Exp}_1, s_2\} \downarrow n_1) \), if \textit{Exp}2 is low then \( \forall s_1, s_2, n_2, s_1 = L. s_2 \Rightarrow (\{\textit{Exp}_2, s_1\} \downarrow n_2 \Leftrightarrow \{\textit{Exp}_2, s_2\} \downarrow n_2) \). The operator \textit{op} does not change the security domains. So, \textit{Exp op Exp} evaluates to the same value \( n_1 \) op \( n_2 \) in all low-equal states \( s_1, s_2 \).

\( (\textit{op} \in \{+,-,*,\text{mod},\text{div},<,\leq,\geq,==,!=\}) \)

5. \textit{B bop B}: Same reasoning as for case \textit{Exp op Exp}. \( (\textit{bop} \in \{\&\&,||\}) \)

The typing rules for expressions (in Figures 22 and 23) are in accordance with theorem 3.

\[
\begin{array}{c}
s(\textit{Id}) = v \\
\{\textit{Id}, s\} \downarrow v \\
\{\textit{Const}, s\} \downarrow \text{Const} \\
\{\text{true, s}\} \downarrow \text{true} \quad \{\text{false, s}\} \downarrow \text{false} \\
\{\text{Exp}_1, s\} \downarrow v_1 \quad \{\text{Exp}_2, s\} \downarrow v_2 \quad \{\text{Exp}_1, s\} \downarrow v_1 \quad \{\text{Exp}_2, s\} \downarrow v_2 \\
\{\text{Exp}_1 * \text{Exp}_2, s\} \downarrow v_1 + v_2 \quad \{\text{Exp}_1 - \text{Exp}_2, s\} \downarrow v_1 - v_2 \\
\{\text{Exp}_1, s\} \downarrow v_1 \quad \{\text{Exp}_2, s\} \downarrow v_2 \\
\{\text{Exp}_1 * \text{Exp}_2, s\} \downarrow v_1 \cdot v_2 \\
\{\text{Exp}_1, s\} \downarrow v_1 \quad \{\text{Exp}_2, s\} \downarrow v_2 \quad v_2 \neq 0 \quad \{\text{Exp}_1, s\} \downarrow v_1 \quad \{\text{Exp}_2, s\} \downarrow 0 \\
\{\text{Exp}_1 \text{mod} \text{Exp}_2, s\} \downarrow v_1 \text{mod} v_2 \quad \{\text{Exp}_1 \text{mod} \text{Exp}_2, s\} \downarrow 0 \\
\{\text{Exp}_1, s\} \downarrow v_1 \quad \{\text{Exp}_2, s\} \downarrow v_2 \quad v_2 \neq 0 \quad \{\text{Exp}_1, s\} \downarrow 0 \\
\{\text{Exp}_1 \text{div} \text{Exp}_2, s\} \downarrow v_1 \text{div} v_2 \quad \{\text{Exp}_1 \text{div} \text{Exp}_2, s\} \downarrow 0 \\
\{B_1, s\} \downarrow b_1 \quad \{B_2, s\} \downarrow b_2 \quad \{B_1, s\} \downarrow b_1 \quad \{B_2, s\} \downarrow b_2 \\
\{B_1 \&\& B_2, s\} \downarrow b_1 \&\& b_2 \quad \{B_1 || B_2, s\} \downarrow b_1 || b_2 \\
\{\text{Exp}_1 < \text{Exp}_2, s\} \downarrow v_1 < v_2 \quad \{\text{Exp}_1 <= \text{Exp}_2, s\} \downarrow v_1 < v_2 \\
\{\text{Exp}_1 \geq \text{Exp}_2, s\} \downarrow v_1 \leq v_2 \quad \{\text{Exp}_1 >= \text{Exp}_2, s\} \downarrow v_1 \leq v_2 \\
\{\text{Exp}_1 > \text{Exp}_2, s\} \downarrow v_1 > v_2 \quad \{\text{Exp}_1 >= \text{Exp}_2, s\} \downarrow v_1 > v_2 \\
\{\text{Exp}_1 \geq \text{Exp}_2, s\} \downarrow v_1 \geq v_2 \quad \{\text{Exp}_1 >= \text{Exp}_2, s\} \downarrow v_1 \geq v_2 \\
\{\text{Exp}_1 == \text{Exp}_2, s\} \downarrow v_1 = v_2 \quad \{\text{Exp}_1 != \text{Exp}_2, s\} \downarrow v_1 \neq v_2 \\
\end{array}
\]

Figure 25: Semantics for expressions. \( \text{Const} \in \mathbb{N}, v_i \in \mathbb{N}, b_i \in \{\text{true, false}\} \).
5.3 Arrays

After the considerations of variable declarations and expressions, we are now prepared to extend the type systems for arrays.

5.3.1 Design Questions and Literature

From the outset, it is not obvious how to deal with arrays in the context of information flow analysis. Questions to be reflected over are:

- **Contents**: Do all elements in an array share the same security domain? Are there applications demanding that array elements should have different security domains?

- **Length**: Should the array length be high or is there a requirement for low array lengths?

- **Indexes**: Is there a problem with high array indexes? How to deal with out-of-bound array access? Which possibilities for insecure information flow does it imply?

To emphasize the relevance of these questions, consider the example program in Figure 26. It demonstrates how information can be leaked indirectly through a high array index. Consequently, high array indexes should be reconsidered.

```plaintext
arr[secret]:=1;
i:=0;
while i<arr.length do
  if arr[i]==1 then
    leak:=i
  end;
i:=i+1
end
```

Figure 26: Leakage through high array indexes, from [DS04]. Assume array `arr` to be initialized with zeros. `secret` is high; `arr`, `i`, and `leak` are low.

A number of approaches for dealing with arrays has been proposed. We consider some of them in more detail to find suitable answers to the questions above.

[DD77] demands that the elements of an array have the same security class because they share the same data type. [Aga00] and [DS04] agree with that. Although one may conceive scenarios which use array elements of different security types, such as a bank storing account numbers with different privacy requirements, it seems reasonable to demand that all array elements share the same security domain. For those scenarios, different data structures, such as records, could be chosen.

[Aga00] allows only low values to specify the length of an array. In the case of a length specification with a high variable, already basic operations such as initialization and the traversal of all elements could reveal the secret array...
length by the timing behavior. [Aga00] also does not allow high expressions to index and update an array.

Concerning array indexing, [DS04] is not so restrictive. It allows array lengths to be high if the array contents are also high (this condition makes the insecure program in Figure 26 be rejected). The condition for array operations involving a high index is that programs do not abort, that out-of-bound array reads simply yield 0, and that out-of-bound array writes are simply skipped.

As claimed in [DD77], problems arise if the indexes select elements outside the declared range of an array. a[i]:=b might cause invalid flow from b to c where c is an object in memory which is addressed when i is out-of-bound. Then, invalid information may flow from a high b to a low c. Array semantics for out-of-bound reads and writes must be defined in a way that prevents this. Agat’s approach in [Aga00] for the array length is adopted here, due to the aforementioned reason. Array indexes are treated as in [DS04] because this allows a broader applicability than Agat’s approach.

The decisions are in detail:

- **Contents**: All array elements share the same security domain.
- **Length**: An array’s length is public and therefore of security domain low.
- **Indexes**: Array indexes may be of security domain high, if the array itself is high; otherwise they must be low.

For the last specification the following prerequisites must be fulfilled, so that the programs can be proven to be secure:

- programs do not abort (in particular not on an out-of-bound array indexing);
- out-of-bound array reads evaluate to a dummy element;
- out-of-bound array writes are skipped.

5.3.2 Syntax, Semantics, and Typing Rules

In this section, we define the syntax, semantic, and typing rules for array commands and expressions according to the decisions derived above. The proof of soundness of the typing rules follows in section 5.3.3.

**Syntax.** The syntax for expressions from page 41 (given below in small font) can now be extended by the access to the length and to the field of an array:

\[
\begin{align*}
&\text{Exp ::= Const | Id | Exp + Exp | Exp - Exp | Exp \times Exp | Exp \mod Exp} \\
&\quad | Exp \div Exp | Arr.length | Arr[Exp]
\end{align*}
\]

The syntax for commands C from page 29 (again in small font) is extended by commands for declaring an array and assigning expressions to array fields:

\[
\begin{align*}
&C ::= \text{skip | Id:Type:SecDom | Arr:Type[Exp]:SecDom} \\
&\quad | Id:=Exp | Arr[Exp]:=Exp \\
&\quad | \text{if } B \text{ then } C \text{ end | if } B \text{ then } C_1 \text{ else } C_2 \text{ end | while } B \text{ do } C \text{ end | } C_1;C_2 \\
&\quad | \text{fork}(C <D_1,...,D_n>)
\end{align*}
\]
Exp in the array declaration Arr:Type[Exp]:SecDom denotes the length of the array, i.e. the number of elements in it.

Semantics. The next step is to define the semantics for array expressions and commands. It is specified in Figures 27 and 28. The interesting part is the treatment of out-of-bound array indexes: out-of-bound writes are simply skipped and out-of-bound reads yield a dummy value, 0 for type integer and false for type boolean. In particular, programs do not abort on an out-of-bound array indexing.

Typing Rules. The security typing rules following from the decisions in the previous section are illustrated in Figure 29. The rules for array expressions apply for both type systems. The rules for commands in the non-transforming type system express that array declarations with low length specifications are secure. Assignments to arrays are secure, if the expression and the index are both low or the array is high. The rules of the transformation type system state that array declarations are unmodified under transformation and their type is the declaration itself as for variable declarations. Assignments to arrays are likewise unmodified under transformation. The low slice of a low array assignment is the same assignment and the low slice of a high array assignment is skip, corresponding to the assignment to variables.

5.3.3 Soundness of Typing

In the following, a formal verification for extending the existing typing rules for arrays is provided. The proof is based on the proof of soundness of the analysis in [SS00], which is extended for arrays.

The first step is to verify the typing rules for array expressions. The length of arrays is low by default and can be observed. Intuitively, an array expression arr[Exp] is low, if both the array contents and the index expression are low. If the contents or the index expression were high for a low array expression, the resulting states might not be low-equal:

1. Assume arr to be a low array initialized to [1,2,3] in all low-equal states s₁ and s₂. Moreover, let Exp be high and evaluating to 1 in s₁ and to 2 in s₂. The expression arr[Exp] cannot be low because evaluating it in s₁ and s₂ produces different low behavior, the result being 1 for starting in s₁ and 2 for starting in s₂. Hence, the resulting values are not low-equal and the expression is high.

2. Assume Exp to be a low variable, say 2. Array arr is high and initialized to [1,2,3] in state s₁ and to [2,3,4] in s₂, with s₁ and s₂ being low-equal. The expression arr[Exp] evaluates to 2 in s₁ and to 3 in s₂. Thus, the resulting values are not low-equal and the expression is high.

Low array expressions must obey Theorem 3. We show that the theorem holds for accessing an array’s length and that accessing a low array field with a low index expression actually yields a low expression. Recall Theorem 3 from page 41:

**Theorem 3.** If an expression Exp is low then

\[ \forall s_1, s_2, n. s_1=L_s_2 \Rightarrow (\{Exp, s_1\} \downarrow n \iff \{Exp, s_2\} \downarrow n). \]
\[
\{\text{Arr}, s\} \downarrow [\text{Arr}_0, \ldots, \text{Arr}_{l-1}]
\]
\[
\{\text{Arr}, s\} \downarrow [\text{Arr}_0, \ldots, \text{Arr}_{l-1}] \quad \{\text{Exp}, s\} \downarrow k \quad k \in \mathbb{N} \quad 0 \leq k < l
\]
\[
\{\text{Exp}, s\} \downarrow k \quad (k \notin \mathbb{N} \lor k < 0 \lor l \geq k)
\]
\[
\{\text{Arr}[\text{Exp}], s\} \downarrow 0
\]
\[
\{\text{Arr}[\text{Exp}], s\} \downarrow \text{false}
\]

Figure 27: Semantics for array expressions.

\[
\{\text{Exp}, s\} \to l
\]
\[
\{\text{Arr}: \text{int}[\text{Exp}]: \text{SecDom}, s\} \to \{\}, [\text{Arr}_0 = 0, \ldots, \text{Arr}_{l-1} = 0]s
\]
\[
\{\text{Exp}, s\} \to l
\]
\[
\{\text{Arr}: \text{bool}[\text{Exp}]: \text{SecDom}, s\} \to \{\}, [\text{Arr}_0 = \text{false}, \ldots, \text{Arr}_{l-1} = \text{false}]s
\]
\[
\{\text{Arr}, s\} \downarrow [\text{Arr}_0, \ldots, \text{Arr}_{l-1}]
\]
\[
\{\text{Exp}_2, s\} \downarrow \text{true} \quad \{\text{Exp}_1, s\} \downarrow k \quad k \in \mathbb{N} \quad 0 \leq k < l
\]
\[
\{\text{Exp}_1; s\} \to \{\}, [\text{Arr}_k = \text{true}]s
\]
\[
\{\text{Arr}, s\} \downarrow [\text{Arr}_0, \ldots, \text{Arr}_{l-1}]
\]
\[
\{\text{Exp}_1, s\} \downarrow k \quad (k \notin \mathbb{N} \lor k < 0 \lor l \geq k)
\]
\[
\{\text{Exp}_1; s\} \to \{\}, [\text{Arr}_k = \text{false}]s
\]
\[
\{\text{Exp}_1; s\} \to \{\}, [\text{Arr}_k = \text{false}]s
\]

Figure 28: Semantics for array commands.
ArrLen  \[ \Gamma(Arr.length) : low \]
\[ \Gamma(Exp) : low \quad \Gamma(Arr) : low \]
\[ \Gamma(Exp[Exp]) : low \quad \Gamma(Exp[Exp]) : high \]

Non-Transforming:
\[ \Gamma(Exp) : low \]
ArrDecl  \[ \Gamma \downarrow \{ Arr : low \} + Arr : \text{Type}[Exp] : low \]
\[ \Gamma(Exp) : low \]
\[ \Gamma \downarrow \{ Arr : high \} + Arr : \text{Type}[Exp] : high \]
\[ \Gamma(Arr) : low \quad \Gamma(Exp_1) : low \quad \Gamma(Exp_2) : low \]
\[ \Gamma \vdash Arr[Exp_1] := Exp_2 \]
\[ \Gamma(Arr) : high \]
\[ \Gamma \vdash Arr[Exp_1] := Exp_2 \]

Transforming:
\[ \Gamma(Exp) : low \]
ArrDecl  \[ \Gamma \downarrow \{ Arr : low \} + Arr : \text{Type}[Exp] : low \quad Arr : \text{Type}[Exp] : low \]
\[ \Gamma(Exp) : low \]
\[ \Gamma \downarrow \{ Arr : high \} + Arr : \text{Type}[Exp] : high \quad Arr : \text{Type}[Exp] : high \]
\[ \Gamma(Arr) : low \quad \Gamma(Exp_1) : low \quad \Gamma(Exp_2) : low \]
\[ \Gamma \vdash Arr[Exp_1] := Exp_2 \quad Arr[Exp_1] := Exp_2 \quad Arr[Exp_1] := Exp_2 \] 
\[ \Gamma(Arr) : high \]
\[ \Gamma \vdash Arr[Exp_1] := Exp_2 \quad Arr[Exp_1] := Exp_2 \quad \text{skip} \]

Figure 29: Typing rules for array expressions (ArrLen and ArrExp) and array commands (ArrDecl and ArrAssign) for the non-transforming and the transforming type system.
**Proof (extended).** The proof is extended for the two cases of array expressions.

6. \( \text{Arr}.\text{length} \): Accessing the length of an array yields the same value in all states because array lengths are public. So they also evaluate to the same \( n \) in all low-equal \( s_1, s_2 \).

7. \( \text{Arr}[\text{Exp}] \): \( \text{Arr} \) and \( \text{Exp} \) are low. In all states \( s_1 \) and \( s_2 \) with \( s_1 \leq_L s_2 \): \( \{ \text{Exp}, s_1 \} \parallel i \Leftrightarrow \{ \text{Exp}, s_2 \} \parallel i \) from the induction hypothesis. Consequently, \( \text{Arr}[\text{Exp}] \) accesses the same array field, namely the \( i \)-th. If \( 0 \leq i < l \), where \( l \) is the length of the array then the value of this low field must be the same for \( s_1 \) and \( s_2 \): \( \{ \text{arr}[i], s_1 \} \parallel v_i \Leftrightarrow \{ \text{arr}[i], s_2 \} \parallel v_i \). If \( i < 0 \) or \( i \geq l \), then \( \{ \text{arr}[i], s_1 \} \parallel 0 \Leftrightarrow \{ \text{arr}[i], s_2 \} \parallel 0 \) (or false for a boolean array), which still evaluates to the same value for all low-equal states. Thus, the expression \( \text{Arr}[\text{Exp}] \) evaluates to the same value in all low-equal states.

\[ \square \]

Having verified the soundness of the typing rules for array expressions, in a second step we check the typing rules for array commands. We want to verify their soundness. We can reuse the proof in [SS00] and extend its components.

The key idea for such soundness proofs is a compositionality property, called the hook-up property in the standard security terminology [Sab01]. Compositionality means that secure subprograms can be composed to form a secure program. This hook-up property facilitates the soundness proof significantly.

The hook-up property will now be derived in three steps:

1. Identify secure contexts, including array commands.
2. Show that the secure congruence theorem holds.
3. Show that the hook-up property holds.

For applying the hook-up property, we need to identify secure contexts, i.e. language constructs that are the building blocks of secure programs. We must prove that these contexts actually preserve security. Let \( [\_] \) be a hole for a command. The definition of secure contexts \( C[\_1, \_2] \) in [SS00] (below given in small font) is extended for declarations and (secure) assignments to array fields:

\[
C_{\text{Arr}[\_1, \_2]} := \skip | h := \text{Exp} | l := \text{Exp} \quad (\text{Exp} \text{ is low}) | [\_1]; [\_2] \\
| \text{if} \ B \ \text{then} \ [\_1] \ \text{else} \ [\_2] \ \text{end} \quad (B \text{ is low}) \\
| \text{while} \ B \ \text{do} \ [\_1] \quad (B \text{ is low}) | \text{fork}(\_1);[\_2] | ([\_1],[\_2]) \\
| \text{Id} : \text{Type} : \text{SecDom} \\
| \text{Arr} : \text{Type}[\text{Exp}] : \text{SecDom} \quad (\text{Exp} \text{ is low}) \\
| \text{Arr}_L[\text{Exp}_1] := \text{Exp}_2 \quad (\text{Arr}_L, \text{Exp}_1, \text{Exp}_2 \text{ are low}) \\
| \text{Arr}_H[\text{Exp}_1] := \text{Exp}_2 \quad (\text{Arr}_H \text{ is high})
\]

**Lemma 4 (Secure Congruence).** If \( C_1 \equiv_L C_1', C_2 \equiv_L C_2', \) and \( C[\_1, \_2] \) is a secure context, then \( C[C_1, C_2] \equiv_L C[C_1', C_2'] \). If \( C[\_1, \_2] \) is a high conditional, this holds provided that \( C_1 \equiv_L C_2 \).
Recall that $\equiv_L$ is the strong low-bisimulation relation, i.e. if $\bar{C}_1 \equiv_L \bar{C}_1'$ then a low observer cannot distinguish the commands $\bar{C}_1$ and $\bar{C}_1'$. Intuitively, the theorem of secure congruence expresses that strongly low-bisimilar programs in a secure context form a new strongly low-bisimilar program. The theorem states that secure contexts must preserve the low-bisimulation, which we prove in the following by induction over the structure of commands:

**Proof.** First, start with the commands in which arrays do not occur directly.

$\text{skip}, [\bullet]_1; [\bullet]_2, \text{fork}([\bullet]_1; [\bullet]_2)$, and $\langle [\bullet]_1; [\bullet]_2 \rangle$

The cases $\text{skip}$, sequential composition, fork, and multiple-threads are proved in [SS00] to be secure. Neither array expressions nor array commands are used directly in these commands, if ever then in subcommands. Hence, the introduction of arrays with the given conditions of out-of-bound indexing behavior does not violate any assumption in the proofs, so they still hold.

Second, consider the commands in which array expressions may occur and prove that they are still secure. Start with $s_1$ and $s_2$ where $s_1 \equiv_L s_2$. Theorem 3 will be used for the evaluation of low expressions.

$l := \text{Exp}$ (Exp is low)

The execution terminates in one time-step. Since Exp is low, $\{\text{Exp}, s_1\} \Downarrow v \Leftrightarrow \{\text{Exp}, s_2\} \Downarrow v$. The same value $v$ is assigned to $l$ in $s_1$ and $s_2$. Hence, the resulting states are low-equal.

$h := \text{Exp}$

In an assignment to a high variable $\{\text{Exp}, s_1\} \Downarrow v_1$ and $\{\text{Exp}, s_2\} \Downarrow v_2$. $v_1$ and $v_2$ may differ in $s_1$ and $s_2$, in particular for out-of-bound array indexing. For any Exp, only $h$ gets a new value, which does not change the low components of the states.

**if** $B$ **then** $[\bullet]_1$ **else** $[\bullet]_2$ **end**

First, consider a low $B$. We have $\{B, s_1\} \Downarrow b \Leftrightarrow \{B, s_2\} \Downarrow b$ according to Theorem 3. One step of computation leads to the execution of either $C_1$ and $C_1'$, if $B$ is true, or $C_2$ and $C_2'$, if $B$ is false. From the assumption we have $C_i \equiv_L C_i'$, which by definition of $\equiv_L$ shows that **if** $B$ **then** $C_1$ **else** $C_2$ **end** $\equiv_L$ if $B$ then $C_1'$ **else** $C_2'$ **end**.

Second, consider a high $B$: $\{B, s_1\} \Downarrow b_1$ and $\{B, s_2\} \Downarrow b_2$. It may happen, that the if-branch is executed in $s_1$ and the else-branch in $s_2$, especially as an array index may be out-of-bound in the expression. Since $C_i \equiv_L C_i'$ for $i, j \in \{1, 2\}$ from the assumption for high conditionals, the computation ends up in low-bisimilar commands.

**while** $B$ **do** $[\bullet]_1$ (B is low)

$\{B, s_1\} \Downarrow b \Leftrightarrow \{B, s_2\} \Downarrow b$: as $B$ is low, it has the same value $b$ in both states. Thus, one step of computation leads either to the termination of the loop (if $B$ is false) or to the new commands $C_1$ **while** $B$ **do** $C_1$ and $C_1'$ **while** $B$ **do** $C_1'$ (if $B$ is true), which are low-bisimilar even if $C_1$ and $C_1'$ spawn new processes. The resulting states are low-equal.
Finally, there are three cases for the new commands: declarations and assignments to arrays. Again, start with \( s_1 \) and \( s_2, s_1 =_L s_2 \) and use Theorem 3.

**Id:** Type \( \text{SecDom} \)

The default values assigned on initialization are low, they are publicly known. Thus, the pair identifier/value which is created on initialization is identical for all low-equal states \( s_1 \) and \( s_2 \): \( \{\text{Id}, s_j\} \downarrow v \) for \( j \in \{1, 2\} \). Other variables are not modified by the declaration command. Thus, the two resulting states are low-equal.

**Arr:** Type \( \text{[Exp]} : \text{SecDom} \) (Exp is low)

The default values assigned on initialization are again low. Thus, all array-field/value pairs which are created on initialization are identical for all low-equal states \( s_1 \) and \( s_2 \):

\[
\{\text{Arr}, s'_j\} \downarrow \text{Arr}[0] = v_1, \ldots, \text{Arr}[l-1] = v_1,
\]

where \( j \in \{1, 2\} \) and \( s'_j \) is the state resulting from \( s_j \) after this declaration. Other variables are not modified. The resulting states are again low-equal.

**ArrL:** Type \( \text{[Exp]} : \text{SecDom} \) (ArrL, Exp1, Exp2 are low)

Because \( \text{Exp1} \) and \( \text{Exp2} \) are low, \( \{\text{Exp2}, s_1\} \downarrow v \Leftrightarrow \{\text{Exp2}, s_2\} \downarrow v \) and \( \{\text{Exp1}, s_1\} \downarrow i \Leftrightarrow \{\text{Exp1}, s_2\} \downarrow i \). Thus, \( \text{Exp1} \) indexes the same array field for both states and the command assigns the same value \( v \) to this array field. In the case of an out-of-bound array index, no value is modified at all in any state. The resulting states are again low-equal.

**ArrH:** Type \( \text{[Exp]} : \text{SecDom} \) (ArrH is high)

\[
\{\text{Exp2}, s_1\} \downarrow v_1, \{\text{Exp2}, s_2\} \downarrow v_2 \text{ and } \{\text{Exp1}, s_1\} \downarrow i_1, \{\text{Exp1}, s_2\} \downarrow i_2. \]

If \( i_j \) is out-of-bound, this command does not assign any value at all; the states do not change. If \( i_j \) is in-bound, then \( \text{Arr}[i_j] \) gets a new value \( v_j \) \((j \in \{1, 2\})\). This may assign different values to different array fields, but as the array is high, the low observation of the array is equal in \( s_1 \) and \( s_2 \).

Now, at last, we can derive the hook-up property.

**Lemma 5 (Hook-up).** If \( \tilde{C}_1 \) and \( \tilde{C}_2 \) are secure and \( \mathbb{C}[ullet_i, \bullet_j] \) is a secure context, then \( \mathbb{C}[	ilde{C}_1, \tilde{C}_2] \) is secure. If \( \mathbb{C}[ullet_1, \bullet_2] \) is a high conditional, then \( \mathbb{C}[C_1, C_2] \) is secure provided \( C_1 \equiv_L C_2 \).

The hook-up property states that composing two strongly secure programs in a secure context results in a strongly secure program. If a high conditional is executed this holds as well provided that the two branches are low-bisimilar additionally.

**Proof.** The proof is identical to the one in [SS00]: If \( \tilde{C}_1 \) and \( \tilde{C}_2 \) are secure then \( \tilde{C}_i \equiv_L \tilde{C}_i \) for \( i = 1, 2 \). Consequently, by the secure congruence lemma \( \mathbb{C}[C_1, C_2] \equiv_L \mathbb{C}[C_1, C_2] \), i.e. \( \mathbb{C}[C_1, C_2] \) is secure. □

After having proved that the hook-up property can also be applied to programs containing array expressions and commands, the extended type system must be
proven to be sound. The extension of the proof will be done for the transformation type system only because the rules of the non-transforming type system are an area restriction actually. The programs accepted by the non-transforming type system are secure if the transforming analysis is sound.

To prove soundness of the extended type system, we need the following theorem:

**Theorem 6 (Soundness of the Analysis).** For the extended security type system we have $C \rightsquigarrow \tilde{C} : \tilde{Sl} \Rightarrow \tilde{C} \simeq_L \tilde{Sl}$.

Intuitively, Theorem 6 states that the attacker cannot distinguish the result of the transformation and its low slice.

**Proof.** Like in [SS00], we give the proof by induction over the transformation thread pool $\tilde{C}$. For basic commands they are:

- $\text{skip}: \text{skip}, \text{let}=\text{Exp} : \text{let}=\text{Exp}$ ($\text{Exp}$ is low), and $\text{h}=\text{Exp} : \text{skip}$

$\tilde{C}$ is low-bisimilar to its low slice, which follows directly from the operational semantics and the definition of $\equiv_L$.

$C'_1;C'_2 : \text{Sl}_1;\text{Sl}_2$, while $B$ do $C'$ end: while $B$ do $\text{Sl}$ end,

$\text{fork}(C'_1;C'_2) : \text{fork}(\text{Sl}_1;\text{Sl}_2)$, $(C'_0 \ldots C'_{n-1}) : (\text{Sl}_0 \ldots \text{Sl}_{n-1})$, and

- if $B$ then $C'_1$ else $C'_2$ end: if $B$ then $\text{Sl}_1$ else $\text{Sl}_2$ end ($B$ is low)

These cases provide secure contexts $\llbracket \cdot \rrbracket_1;\llbracket \cdot \rrbracket_2$, while $B$ do $\llbracket \cdot \rrbracket$, $\text{fork}(\llbracket \cdot \rrbracket_1;\llbracket \cdot \rrbracket_2)$, $(\llbracket \cdot_0 \ldots \llbracket \cdot_{n-1})$, and if $B$ then $\llbracket \cdot \rrbracket_1$ else $\llbracket \cdot \rrbracket_2$ end respectively. By the induction hypothesis the subcommands have low-bisimilar low slices, so the conclusion follows by the hook-up property.

- if $B$ then $C'_1;\text{Sl}_2$ else $\text{Sl}_1;C'_2$ end : skip; $\text{Sl}_1;\text{Sl}_2$

By the induction hypothesis $C'_1 \equiv_L \text{Sl}_i$ ($i=1,2$). Since $\llbracket \cdot \rrbracket_1;\llbracket \cdot \rrbracket_2$ is a secure context, we have $C'_1;\text{Sl}_2 \equiv_L \text{Sl}_1;C'_2$. Now, we can apply the special case of the hook-up property for high conditionals. With the assumption that $\text{skip}$ has the same timing behavior as branching on $B$ the strong low-bisimulation holds. Recall that variable declarations do not occur in conditionals and thus are not padded into the other branch.

Now we add the cases for array assignments and declarations:

$\text{ArrL}[\text{Exp}_1] := \text{Exp}_2 : \text{ArrL}[\text{Exp}_1] := \text{Exp}_2$ ($\text{ArrL}$, $\text{Exp}_1$, $\text{Exp}_2$ are low)

$C'$ is low-bisimilar to its low slice, which follows directly from the operational semantics and the definition of $\equiv_L$.

$\text{ArrH}[\text{Exp}_1] := \text{Exp}_2 : \text{skip}$ ($\text{ArrH}$ is high)

$C'$ is low-bisimilar to its low slice correspondingly to the high variable assignment for the same reason as in the previous case.

$\text{Id}:\text{Type}:\text{SecDom}$ : $\text{Id}:\text{Type}:\text{SecDom}$ and

$\text{Arr}:\text{Type}[\text{Exp}]:\text{SecDom} : \text{Arr}:\text{Type}[\text{Exp}]:\text{SecDom}$ ($\text{Exp}$ is low)

$C'$ is low-bisimilar to its low slice, again for the same reason. $\equiv_L$.

All cases of $\tilde{C} : \tilde{Sl}$ have been checked, which proves Theorem 6 correct.

Soundness of the analysis implies security of the analysis:
Theorem 7 (Security of the Analysis). $\overline{C} \leftarrow \overline{C'} : \overline{S} \Rightarrow \overline{C'}$ is secure.

Proof. With Theorem 6, $\overline{C'} \equiv_L \overline{S} \overline{C}$. The symmetry and transitivity of $\equiv_L$ entail $\overline{C'} \equiv_L \overline{C}$. Hence, $\overline{C'}$ is (strongly) secure.

To summarize, the typing rules of Figure 29 with the semantics of Figure 28 have been proven to fit into the system of [SS00]. The complete type system including array expressions and commands is sound. Programs typable under these typing rules are strongly secure.

5.3.4 Implementation

As described, MWL is extended by the following constructs: array declarations, access to array fields, and access to the length of an array. Consequently, three new classes are added to the AST data structure (see Figure 11 on page 24): ArrDecl, ArrField, and ArrLength.

Parallel to VarDecl, class ArrDecl represents an array declaration. Both declarations have a common superclass Decl, extending PrimitiveCommand (see Figure 30, left).

Two new expressions extend Exp: ArrField represents an array element and ArrLength represents the length of an array (see Figure 30, center).

Both Identifier and ArrField implement the new interface Assignable which represents the left side of an assignment: Values can be assigned to an Assignable (see Figure 30, right).

No new type had to be implemented for arrays because arrays use the basic types integer and boolean for their elements.

Both the analyzers with and without transformation have been extended by the typing rules. The following source code excerpt shows the extension for array declarations in the TransformVisitor:

```java
public Object visit (ArrDecl ad) {
    ArrDecl aNew = (ArrDecl) ad.clone();
    ArrDecl aNewType = (ArrDecl) ad.clone();
    return new ElementPair(aNew, aNewType, false);
}
```

The complete source code with all extensions can be found in Appendix C.

![Diagram](image)

Figure 30: AST extensions for arrays.
6 Tests and Examples

This section gives insight into the applicability of the security analyzer by checking selected search and sort algorithms. Some of the programs are rejected by both implemented type systems. Others are rejected by the non-transforming type system, but can be transformed automatically into strongly secure programs by the transforming type system. None of the programs that we have inspected was secure in its original version. This indicates that it could be interesting and important to analyze such common algorithms.

Whereas the algorithms themselves offer only few implementation options, the security domains of the variables have to be chosen reasonably. They must be defined by the programmer. For some variables this choice is obvious, e.g. for variables that come from the outside and are passed to the algorithms. They are usually high. The security domains of these variables, which would be passed in a function header, are assumed to be given in the example programs. Consequently, they are lacking in the variable declaration sections. The security domains of the other, basically local, variables are defined in the declaration sections. The choices made for the security domains are explained directly when presenting the programs.

6.1 Search Algorithms

Assume, you want to determine the index of a certain element value in an array of sorted integers. The array contains confidential information, such as staff salaries or bank account numbers. Hence, its security domain is high. The goal of the search is to get the index of element value without leaking information of the remaining contents of the array.

6.1.1 Simple Search

Description: Simple search traverses the whole array, comparing each element with value. It does not exploit the fact, that the array is sorted. See Figure 31.a for the program.

Security domains: The security policy is defined as follows: $i$ is low and result is high. $i$ is a local variable traversing the array in a predefined way. It does not represent any confidential knowledge and is thus low. After program execution, result is the index of element value. This is confidential and therefore, result is chosen to be high. The array itself is high, just as in all following examples.

Check result: The security analyzer determines that the original program is insecure. It is rejected by the non-transforming type system because of a high conditional. However, the simple search program is transformed into a secure program by the transforming type system. The conditional is padded with an else-branch that has the same timing behavior as the if-branch. Figure 31.b shows the resulting strongly secure program.

---

3Assume for simplicity that the elements in the array are pairwise distinct.
i : int : low;  
result : int : high;  
i := 0;  
result := arr.length;  
while i<arr.length do  
  if arr[i]==value then  
    result := i  
  end  
end  

31.a: Original insecure version.  
31.b: Transformed secure version.

Figure 31: Simple Search. Array arr is given as a secret integer array; imagine its declaration as arr:int[length]:high. It is searched for the secret element value; imagine it as value:int:high. After program execution, result contains the index of value if it is contained in the array. Otherwise, result equals the length of the array.

Analysis: Using the simple search algorithm in its transformed version determines the index of a element value without leaking information, neither directly, nor indirectly, nor through timing behavior. The running time is always linear.

For understanding the analysis better, one may think about setting result to low. Then, the program would have a conditional with a low assignment. This modified program is rejected by the security analyzer, it is not strongly secure. However, more intuitively, the observer does not learn any confidential information. He only learns the number of elements less and greater than value, which is publicly known after the execution of the program anyway because result is low.

6.1.2 Binary Search

Description: Binary search exploits the fact that the array is sorted. The algorithm splits up the array into two arrays of about equal length and then determines by a comparison with the median element which half of the array contains the element. It goes on like that until the element is found. See Figure 32 for the MWL program.

Security domains: The security policy is defined as follows: result is high for the same reasons as above in the simple search; left, right, and mid are low. left and right occur in the guard of a loop. The guard of loops must be low in typable programs. So, left and right are labeled with low. Because of the assignments from mid to left and right, mid is labeled with low as well.

Assume \(\{arr, s_1\} \downarrow (1, 2), \{arr, s_2\} \downarrow (2, 3), \{result, s_1\} \downarrow 2, \{result, s_2\} \downarrow 2, s_1=s_2\) before the first execution of the conditional. After its execution, \(\{result, s'_1\} \downarrow 1\) and \(\{result, s'_2\} \downarrow 2\). Hence, \(s'_1 \neq s'_2\).
left : int : low;
right : int : low;
mid : int : low;
result : int : high;

left := 0;
right := arr.length-1;
result := arr.length;
while left<=right do
  mid := (left+right) div 2;
  if arr[mid]==value then
    result := mid;
    left := right+1
  else
    if value<arr[mid] then
      right := mid-1
    else
      left := mid+1
  end
end

Figure 32: Binary Search. Assume array arr to be given as a secret integer array; again imagine its declaration as arr:int[length]:high. The array is sorted. It is searched for the secret element value with value:int:high. After program execution, result contains the index of value in the array, if it is contained. Otherwise, result equals the length of the array.

Check result: The security analyzer rejects the program. It cannot be transformed into a secure program by the implemented rules. The problem is the second high conditional with assignments to the low variables right and left. Even if left and right were high the program would be rejected.

Analysis:
Let us consider the information that an attacker can gain more precisely. He can observe the development of the values of right, left, and mid during the execution of the program because they are low. He cannot see the values of array arr, of value, and result.

Since result is high, the index of element value should be secret after program execution. This condition is violated as shown by the following observation. By observing the development of the values of left and right, the attacker can draw conclusions about how many elements in the array are smaller or greater than value. More precisely, when no new index within the array borders is assigned to neither left nor right, which the attacker can observe, then he knows the value of result because it equals mid. This represents an insecure information flow, which is detected by the security analyzer.
### 6.2 Sort Algorithms

Assume that we have again an array of integers as in the previous section. It still contains confidential data, but it is unsorted now. We analyze algorithms to sort the array.\(^5\) This may e.g. be necessary for improving the efficiency of a later computation. The sorting shall not leak any information of the contents of the array.

**Remark.** In the transformed programs presented in the following, the security analyzer inserts additional parentheses around expressions. Their goal is to eliminate ambiguity when the expressions are more complicated. They do not modify the semantics and thus can be read over.

#### 6.2.1 Bubble Sort

**Description:** Bubble sort sorts the array by comparing each element with the element next to it and swapping the two if required. In the simplest version, the algorithm repeats this process \(n\) times if \(n\) is the array length. See Figure 33.a for the MWL program.

```ml
i : int : low;
j : int : low;
temp : int : high;
i := arr.length-1;
while i>=0 do
  j := 1;
  while j<=i do
    if arr[j-1]>arr[j] then
      temp := arr[j-1];
      arr[j-1] := arr[j];
      arr[j] := temp
    else
      skip;
    end;
    j := j+1
  end;
i := i-1
end
```

33.a: Original insecure version. 33.b: Transformed secure version.

Figure 33: Bubble Sort. It sorts array \(arr\). \(arr\) is a high integer array; imagine it to be declared as \(arr:int[length]:high\).

---

\(^5\)The sort algorithms are taken from [http://linux.wku.edu/~lamonml/algor/sort/sort.html](http://linux.wku.edu/~lamonml/algor/sort/sort.html) and converted to MWL.
Security domains: $i$ and $j$ are high because they are local variables and traverse the array in a predefined manner. $\text{temp}$ is high because it is used for swapping elements of the array. Since the array contents are confidential, $\text{temp}$ must be high. Otherwise a low observer would get to know the values.

Check result: The original bubble sort program is rejected by the non-transforming security type system. The problem is a high conditional. However, it can be transformed into a secure program. The high conditional only contains assignments to high variables and hence, it can be padded. The transformed program is strongly secure and can be found in Figure 33.b.

Analysis: In the original program, information about the initial order of elements may be leaked by the timing behavior. If the array is sorted already, the program will execute noticeably faster as for a worst case input, i.e. for an array sorted from the greatest to the smallest element. After the transformation, the running time of the program is equal for any input array of equal length.

6.2.2 Insertion Sort

Description: Insertion sort inserts each element into its proper place in the final sorted array, shifting all elements in between by one position.

Security domains: $\text{temp}$ is again used for swapping the elements and therefore high. $i$, $j$, and $k$ are low. They are indexes traversing the array. They could also be high – the result of the analysis would be the same.

Check result: The security analyzer cannot type the program. It cannot find any transformation to repair the program. The problem is a high loop.

Analysis: Due to the high $\text{temp}$ variable, the guard of the second loop is high. This leads to the rejection of the program because there is no rule for high loops.

```plaintext
i : int : low;
j : int : low;
k : int : low;
\text{temp} : \text{int} : \text{high};
i := 1;
while i<arr.length do
  \text{temp} := \text{arr}[i];
  j := i;
k := j-1;
  while (j>0)&&arr[k]>\text{temp}) do
    \text{arr}[j] := \text{arr}[k];
    j := j-1;
k := j-1
  end;
\text{arr}[j] := \text{temp};
i := i+1
end
```

Figure 34: Insertion Sort. Sorts array $\text{arr}$, declared as $\text{arr} : \text{int}[\text{length}] : \text{high}$. 
The timing behavior of this programs depends on the original order of the elements in the array. Dummy instructions cannot be inserted automatically as in the previous examples because the timing leak occurs in a loop and not in a conditional. There exists no transformation rule for high loops in the implemented type system.

6.2.3 Selection Sort

Description: Selection sort works by selecting the smallest unsorted element remaining in the array and then swapping it with the element in the next position to be filled. See Figure 35.a.

Security domains: \( i \) and \( j \) are low because they are indexes that traverse the array in a predefined manner. \( n \) is low as well. It is only used for storing the length of the array, which is low by default. \( \text{temp} \) is again used for swapping the elements. Therefore \( \text{temp} \) is high. The variable \( \text{min} \) is high. \( \text{min} \) is the position of the smallest not yet sorted element that remains in the array. If it was low the program would be rejected due to a high conditional with a low assignment.

```plaintext
i : int : low;
j : int : low;
n : int : low;
temp : int : high;
min : int : high;

i := 0;
n := arr.length-1;
while i<n do
  min := i;
j := i+1;
  while j<arr.length do
    if arr[j]<arr[min] then
      min := j
    end;
j := j+1
  end;
temp := arr[i];
arr[i] := arr[min];
arr[min] := temp;
i := i+1
end
```

```plaintext
i : int : low;
j : int : low;
n : int : low;
temp : int : high;
min : int : high;

i := 0;
n := (arr.length-1);
while i<n do
  min := i;
j := (i+1);
  while j<arr.length do
    if arr[j]<arr[min] then
      min := j
    else
      skip
    end;
j := (j+1)
  end;
temp := arr[i];
arr[i] := arr[min];
arr[min] := temp;
i := (i+1)
end
```

Figure 35: Selection Sort. It sorts array \( \text{arr} \). Again imagine it to be declared as \( \text{arr} : \text{int}[\text{length}] : \text{high} \).
Check result: The original selection sort program is rejected by the non-transforming type system. The problem is a high conditional. However, it can be transformed into a strongly secure program, see Figure 35.b.

Analysis: The original program has insecure information flow because of a timing leak. By padding the empty else-branch, the program can be transformed into a strongly secure program. After the transformation, the running time of the program is equal for any input array of equal length.

However, if we label the variable $\text{min}$ with low, the security analyzer rejects the program and cannot repair it. This modified program is not typable because of an indirect information leak in the conditional. This demonstrates that it is not always obvious of how to label the variables with security domains.

6.3 Transformation by Hand

Recall the binary search algorithm from page 55. It is rejected because it cannot be typed by the security typing rules. There is no automatic transformation into a secure program with these rules. Nevertheless, the program can be transformed by hand, which is demonstrated in Figure 36. Its semantics is equal enough to the original program so that it can act as a replacement.

```plaintext
left : int : high;
right : int : high;
mid : int : high;
result : int : high;

left := 0;
right := arr.length-1;
result := arr.length;
while left<=right do
  mid := (left+right) div 2;
  if arr[mid]==value then
    result := mid;
    left := mid+1
  else
    if value<arr[mid] then
      right := mid-1
    else
      left := mid+1
  end
end

Figure 36: Transformed Binary Search. Array $\text{arr}$ is again given as a secret integer array, which is sorted. It is searched for the secret element $\text{value}$. Now, the program is strongly secure.
Let us consider the differences between the original and its transformation:

1. Variables \texttt{left}, \texttt{right}, and \texttt{mid} are \textit{high} in the transformed program whereas \textit{low} in the original.

2. The assignment \texttt{left:=right+1} in the if-branch of the original is replaced by \texttt{left:=mid+1} in the transformed program.

The original program is rejected because of a low assignment in a high conditional. The high conditional in the transformed program only contains high assignments due to the modified security policy. Both branches are observationally equivalent (they correspond to \texttt{skip;skip} for a low observer).

The security analyzer still rejects the program. The reason for this is the loop whose guard is high due to the modified security policy. But as mentioned before, the implemented type systems are not precise.

We will argue why the high guard in the loop does \textit{not} cause an information leak: Both high variables of the guard are initialized with low values right before the loop. Whether the loop is executed or not does consequently not depend on confidential data. Moreover, there are only high assignments in the loop. The attacker cannot observe these modifications of the state. By setting \texttt{left} to \texttt{mid+1} instead of \texttt{right+1} in the first branch of the conditional (which led to the premature termination of the loop in the original), the same timing behavior is achieved for all input sequences of equal length. It is logarithmic in the number of elements in the array (which is much better than the running time of the transformed simple search algorithm).

By this intuitive argumentation, we claim that the program presented in Figure 36 is secure. This cannot be verified by applying the typing rules. We do not prove the strong security property of the program formally, but want to point out, that further transformations and maybe even further transformation rules in the type systems are not impossible.

### 6.4 Summary

The presented five examples demonstrate that common implementations of basic search and sort algorithms may leak confidential information. None of the considered programs is strongly secure in its original version. The security analyzer detects the information leaks. Two of the algorithms, binary search and insertion sort, cannot be transformed automatically into secure programs by the security analyzer. Simple search, bubble sort, and selection sort can be repaired by the transforming type system. They are transformed on the cost of efficiency and length of code.

Furthermore, considering the binary search algorithm, we have shown that programs rejected by the security analyzer might still be transformed by hand. This example may be source of inspiration for developing a further typing rule for loops with high guards.
7 Conclusion

7.1 Discussion of Results

We have succeeded in developing a static security analyzer for the simple multi-threaded language MWL. The main components of the analyzer are the parser for reading MWL programs, the data structure for storing the programs, and the implementation of logic to analyze their information flow.

The parser is generated from grammar files. Hence, we were able to develop it fast. Additional MWL language constructs and modifications must only be mapped to the grammar file and a new parser can be generated at once.

We have implemented the logic of two existing security type systems, a non-transforming and a transforming one. The security analyzer first checks the program with the non-transforming type system. If the check fails it tries to transform it into a secure program with the transforming type system. We have used the visitor design pattern for the implementation of the type systems. This enabled us to separate the logic for analyzing the information flow from the program representation as well as to reuse components (the visitor for typing expressions is used by the visitors for both security type systems). This design is working for the two type systems as well as for their extensions (Section 5). We are confident that the implementation is extensible for further security type systems and maybe also for different languages without great effort.

A makefile is provided for installing and running the security analyzer and the tool itself can be explored with the help of an html documentation. For these reasons we hope that the security analyzer is user-friendly.

When we wanted to test our developed tool we were surprised how few test programs we could find, both existing ones in research papers as well as new ones. Although we came up with a program that tests whether a number is prime, this program is actually not so interesting for an information flow analysis. While searching for interesting examples we discovered that introducing arrays as language construct into MWL could remedy the lack of examples.

Hence, we decided to extend the MWL language for arrays instead of elaborating extensions listed in the thesis formulation. First, we have extended the semantics for MWL. Second, we have extended the typing rules of both security type systems accordingly. Moreover, we have proved that the resulting security type system is sound for the extended language. Finally, we have implemented the extensions for both type systems.

With these extensions, we were able to suggest programs whose information flow is interesting to analyze. We have presented and analyzed five sort and search algorithms. They have been checked with the security analyzer. None of these programs is strongly secure in its original version. Two of them cannot be transformed by our implemented type system, but the remaining three programs can be transformed into strongly secure ones. This demonstrates the

\(^6\text{when introducing our adapted MWL language on page 30}\)

\(^7\text{Even the modular exponentiation algorithm from [SS00], which we have presented when introducing the security type systems in Section 2.3, makes use of arrays. The original transforming type system does not handle them explicitly.}\)
applicability of the developed tool. However, the security analyzer is not precise because the implemented type systems only approximate strong security. We have transformed the binary search algorithm, which is rejected by both type systems, by hand.

7.2 Related Work

As described, the programs that can be repaired by the security analyzer are transformed automatically. In [AS01], Agat and Sands already discuss how searching and sorting algorithms can be adapted to work on collections of confidential data without leaking confidential information. They patch insecure Java programs meaningfully by hand. But they only argue, even though in detail, why the resulting programs are secure. The programs resulting from our automatic transformation are strongly secure, which is proven by the soundness theorem of the transformed type system.

There is at least one major fact that distinguishes the setting of this thesis from a couple of other papers on analyzing information flow on the level of a concrete programming language: Our language is not purely sequential, but we are in the setting of a multi-threaded language. The type systems we apply may be more restrictive than others. As an example, in [VSI96] Volpano et al. state a non-transforming type system in a sequential language. Their typing rules are more precise than our non-transforming typing rules in that they permit particular high conditionals: e.g. the program if h=1 then h:=h+1 else h:=h-1 is accepted. However, non-termination and timing leaks, which can become a serious problem in concurrent programs as demonstrated on page 12, are not considered. Hence, programs such as if h=1 then h:=h+1 else h:=h-1;h:=h*2 are accepted by the type system in [VSI96] but rejected by our non-transforming security type system.

7.3 Future Work

We have mentioned open issues in the program code in Section 4.6. Apart from this, it would have been interesting and promising to pursue one of the following approaches. Unfortunately, this was not possible in the four-month period prescribed for a diploma thesis at ETH Zürich.

- It would be interesting to analyze real-world programs with the security analyzer. In order to check programs in use or under development, an analyzer for a real-world language must be developed. Achieving this is a challenging task: It requires further theoretical explorations, even if only a restricted subset of the language is analyzed at first (e.g. dealing with functions and methods, local variable declarations, etc.). We suppose that a security analyzer could detect information leaks in implemented algorithms and used tools. Some of these leaks might be repaired automatically or by hand. Even if the leaks cannot be repaired, awareness about their existence might lead to using such programs or algorithms more carefully.

- In our adaptation of MWL, we have decided to label variables with their security domains in the source code. The security policy is defined in-line
to encapsulate all the required information within the program. However, real-world languages usually do not provide such declarations in the language core and programs in use do not contain these declarations. Hence, it could be desirable to define the security policy externally. The security domains could be passed to the security analyzer from the outside when specifying the input program. This has no effect on the actual analysis, but would support the real use of the security analyzer.

- In the current security analyzer, the user has to decide on the security domains of all variables. For some variables – the ones containing confidential data – this choice is obvious. For others, in particular for local variables, the decision requires second thoughts. Whether a program is accepted or not depends on the defined security policy. In the test phase of our security analyzer, we even had to test different labelings for the variables to find one so that the program is accepted. We experienced that as rather cumbersome. It would simplify life for the user if he only had to decide on the security domains of some variables, in particular of confidential data. The security analyzer must be modified in such a way that it determines the labeling for the remaining variables itself with the goal of accepting the program. This could be done by testing all possibilities or by a more sophisticated approach.

- Further work would be desirable on improving the transformations. The applied approach for transforming insecure programs into secure ones is purely local. This has the effect that e.g. the size of branches of high conditionals are increased by the padding even if they are already secure and observationally equivalent. To arrive at better transformations it may be helpful to reason more globally. One existing approach is based on unification [KM05]. It would be interesting to investigate if and how such an approach can be integrated into the security analyzer.
References


[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, New York, 1995.


List of Figures

1. Semantics of commands. ........................................... 9
2. Two-level information flow policy ............................... 10
3. Example program for a high conditional inducing a storage leak . 12
4. Typing rules without transformations. ........................... 15
5. Sample inference tree without transformations ................... 16
6. Typing rules with transformation to eliminate timing leaks. ... 18
7. Sample inference tree with transformations ...................... 19
8. The analyzing process. ............................................. 21
9. Class model of the classes of the security analyzer .......... 22
10. The class Program: UML representation and tree representation . 23
11. Class diagram for representing abstract syntax trees. ......... 24
12. From the source code to the abstract syntax tree. ............. 25
13. Class diagram of the classes involved in the visitor pattern. ... 26
14. Class diagram of the visitor classes. ............................ 27
15. Sample MWL program ............................................. 30
16. Grammar file excerpt for generating the scanner with JLex. ... 31
17. Grammar file excerpt for generating the parser with Java CUP. 31
18. Class interaction for checking a program without transformations 32
19. Source code excerpt of the class NoTransformVisitor ........ 32
20. Source code excerpt of the class TransformVisitor ............ 33
21. Source code excerpt of CheckerTransform ........................ 34
22. Extended non-transforming rules with variable declarations ... 39
23. Extended transformation rules with variable declarations ..... 40
24. Syntax for expressions. ........................................... 41
25. Semantics for expressions ........................................ 42
26. Leakage through high array indexes ............................ 43
27. Semantics for array expressions. ................................. 46
28. Semantics for array commands. .................................. 46
29. Typing rules for array expressions and array command ....... 47
30. AST extensions for arrays. ...................................... 52
31. MWL Program: Simple Search ................................. 54
32. MWL Program: Binary Search .................................. 55
33. MWL Program: Bubble Sort ..................................... 56
34. MWL Program: Insertion Sort .................................. 57
35. MWL Program: Selection Sort ................................... 58
36. MWL Program: Transformed Binary Search, strongly secure ... 59
A Formulation of Thesis Objectives

Introduction

Protecting the confidentiality of information is an important problem in modern networked information systems. In particular, the use of mobile code creates additional threats.

Imagine the following scenario: You want to run an application from a software supplier that you do not fully trust. The application (e.g. a spreadsheet) might need confidential (high) data to perform its task. On the other hand, there might be communication of seemingly uncrritical (low) data (e.g. a registration process) with the supplier of the software. The question is how to ensure that the program does not "leak" the secret data, neither accidentally (bugs in the program) nor on purpose (a Trojan Horse).

An Information Flow analysis is a possible answer to such threats. Its purpose is to check that there is no secret information leaking from high input to low output. Possible leaks include explicit assignments such as in statements like \( l ::= h \) or, more subtly, in \( \text{if } h = 1 \text{ then } l ::= 1 \text{ else } l ::= 0 \), where one can draw conclusions on the (secret) value of the high variable \( h \) only by observing the (non-secret) value of the low variable \( l \). Recently, so called Security Typed Systems have been developed for mechanizing information flow analyzes. For instance, the rules for typing assignments and conditionals read as follows:

\[
\frac{pc = \text{low} \vdash \text{exp} : \text{low}}{\frac{\text{[pc]} \vdash l ::= \text{exp}}{\text{[pc]} \vdash \text{if exp then } C_1 \text{ else } C_2}}
\]

Here, \( pc \) denotes the security context that can be either "high" or "low". Informally, this can be read: The conditional can only be typed if both the guard and the branches can be typed in the same context. Since an assignment to a low variable cannot be typed in a high context, this rules out the unwanted example from above.

Apart from toy examples as above, today's security typed languages help avoiding many subtle leaks and cover a broad range of interesting language constructs, including distributedness, concurrency, synchronization, and downgrading.

Project Objectives

Core: The objective of this Diplomarbeit is to design and implement a static analyzer for a simple imperative programming language, namely the multi-threaded while language (short: MWL) [8]. The tool should be able to parse MWL programs and to check their security based on the security type system described in [8] (with and without the program transformation).
Extension 1: The tool could be extended to support type systems for extensions of MWL (like the ones proposed in [5, 4, 6]). Based on the experiences made, the (architecture of the) tool could be improved to better support such modifications. The limits of such extensions could be explored by analyzing type systems for other programming languages.

Extension 2: The tool could be extended to facilitate the correction of programs after an unsuccessful type check. This might include an automatic correction by more sophisticated transformations (like the one in [3]) or an improved support for the manual correction by the user (e.g. localizing problems in the program, explaining the nature of the problem to the user, and giving concrete suggestions on how these problems could be resolved).

Main Activities

Defining a schedule for the entire project

Developing the basic static analyzer includes
- designing an architecture for the security analyzer,
- implementing the data structures,
- implementing/generating the parser,
- implementing the security type checker without transformation,
- implementing the security type checker with transformation,
- implementing a suitable user interface, and
- testing of and experimenting with the tool using suitable example programs.

Extension 1: Investigating the extensions of the language and the type system could include
- analyzing the extensions of MWL and the corresponding security type systems with primitives for synchronization [5], declassification [4], and message passing communication in distributed programs [6] with respect to the following questions:
  - Which interesting programs can be encoded in the extended language?
  - How difficult is the integration of the extended language and type system into the security analyzer and how could this be done?
- proposing a system architecture that is flexible in the sense that it facilitates exchanging the security type system to a fairly similar one (like the ones mentioned before)
- discussing the difficulties that arise when one moves to type systems for languages of a different flavour like, e.g., the one described in [1]

Extension 2: Supporting the correction of insecure programs could include
- extending the tool to support more sophisticated transformations and
- providing the user with information about where the problem is in the program, what its nature is, and how it could be resolved.
Deliverables

The diploma thesis should include:

- brief user documentation for the security analyzer
- detailed development documentation for the security analyzer
- detailed explanation of design choices made (description of alternatives, discussion of their advantages and disadvantages, arguments for the chosen solution, discussion of decision in retrospective)
- description elaborating on insights gained, of open problems identified, and possible extensions of the system or the type system that could be pursued in the future

The tool shall be supplied in electronic form (well-documented Java source code and executable).
The talk shall present the main results of the diploma thesis and could include a tool demonstration.

Benefits for the Student

The student will get in touch with an exciting and promising topic in information security. The design and implementation of the program analyzer deepens practical skills in software development. It is also a good opportunity for gaining insight into secure programming and security type systems.

Prerequisites

- background in programming language semantics (operational semantics)
- interest in security issues
- software development skills

Supervision

Prof. David Basin, Dr. Heiko Mantel, Boris Köpf ([heiko.mantel,boris.koepf]@inf.ethz.ch)

References


B User Documentation

This appendix documents the installation and use of the analyzer for checking the security of MWL programs. The software is provided on CD-ROM. It includes the security analyzer itself with MWL example programs, this development documentation, and a README file.

B.1 Installation Instruction

The security analyzer is designed for UNIX/Linux-like operating systems and bundled into one tar-gzipped file: SecurityAnalyzer.tar.gz. It utilizes the Java Development Kit (JDK 1.4 or later). For installing and running the tool, using make is advantageous.

Follow these steps to setup the security analyzer:

- Create a directory for the security analyzer, say secAnalyzer.
- Copy the file SecurityAnalyzer.tar.gz from the CD-ROM to this directory.
- Change the current directory to secAnalyzer.
- Extract the gzipped file using tar -xzvf SecurityAnalyzer.tar.gz. This extracts the Makefile and the folders doc, mwl, and src in secAnalyzer (see B.4 for more details to the folders).
- Compile the security analyzer by running make all. It compiles the scanner generator, generates the parser, and compiles all source files for the security analyzer. The binaries can now be found in folder class.

More details of the Makefile are provided in Appendix B.3.

B.2 Running the Analyzer

There are two possibilities for running the analyzer on an MWL program: either by using the Makefile or by calling the analyzer manually:

- **Makefile**: In the target 'run' of the Makefile, modify the name of the default MWL file to the program you want to check. Then call make run in the folder secAnalyzer. The default file mwl/selectionSort.mwl is provided, so you can test the tool right away.

- **Manually**: Call the analyzer directly by java checker.Analyzer <mwl-file>. This assumes that the classpath is set correctly to the two directories secAnalyzer/class and secAnalyzer/src. Alternatively, you can specify the classpath in the -classpath option of the java command: java -classpath :class:src checker.Analyzer mwl/selectionSort.mwl, when you are in folder secAnalyzer.

The output will be as specified in Section 4.5. If the program can be transformed, you will find the secure program in the same directory as the input program, having the suffix _secure.

---

8Ignore the note to recompile.
B.3 Makefile

The Makefile has the following targets:

- `make jlex` - Compiles the scanner generator JLex.
- `make parser` - Builds the parser for MWL programs with JLex and Cup
- `make checker` - Compiles all classes needed for the security analyzer
- `make all` - Does all of the above
- `make run` - Runs the security analyzer on an MWL file (specified in the Makefile)
- `make javadoc` - Creates HTML documentation of all java files with Javadoc
- `make clean` - Deletes the created class files
- `make cleanall` - Executes clean and deletes all javadoc files additionally
- `make help` - Prints the help for the Makefile

B.4 Source Code Organization

After the compilation, the `secAnalyzer` folder contains the following subfolders:

- `class` - the compiled binary classes
- `doc` - HTML documentation for all java files in `src`, created with Javadoc
- `mwl` - examples of MWL program
- `src` - the source code for the security analyzer

The source code in folder `src` is organized in six subfolders:

- `JLex` - scanner generator from JLex
- `ast` - abstract syntax tree for representing MWL programs
- `checker` - main folder, contains the top-level java file for calling the analyzer (`Analyzer.java`)
- `java_cup` - parser generator from Java CUP
- `jlexCup` - scanner and parser generation
- `visitor` - logic for the analyzers, implementing the visitor design pattern

Parts of the source code can be found in Appendix C.

B.5 Example Programs

A number of MWL example programs are provided in folder `mwl`.

- `binarySearch.mwl` - binary search algorithm
- `bubbleSort.mwl` - bubble sort algorithm
- `insertionSort.mwl` - insertion sort algorithm
- `largest.mwl` - simple program to test the core functionality
- `modExp.mwl` - modular exponentiation algorithm
- `prime.mwl` - simple algorithm to determine if a number is prime
- `selectionSort.mwl` - selection sort algorithm
- `simpleSearch.mwl` - simple search algorithm
B.6 Remarks

For understanding the security analyzer and for further development it may be helpful to make use of the Javadoc HTML documentation. Start with the file doc/index.html.

One further remark concerns the MWL programming language. Since the security analyzer provides no compiler, finding syntactic errors is quite cumbersome. A frequent error is to end the last command of a program with a semicolon ‘;’. This holds also for programs within commands: Write \texttt{if } h==0 \texttt{ then } h:=h+1 \texttt{ end} instead of \texttt{if } h==0 \texttt{ then } h:=h+1; \texttt{ end}.
C Source Code

C.1 Parser Generation

Input file for the lexical analyzer generator JLex:

```java
package mwlParser;
import java.lang.System;
import java_cup.runtime.Symbol;

%%
%cup
%class MWLScanner
%public

ALPHA=[A-Za-z]
DIGIT=[0-9]
NONNEWLINE_WHITE_SPACE_CHAR=[\t\b\012]
WHITE_SPACE_CHAR=[\n \t\b\012]
STRING_TEXT= {ALPHA}({ALPHA}|{DIGIT})*

%%

"," { return (new Symbol(sym.SEMI)); }
"," { return (new Symbol(sym.COMMA)); }
"(" { return (new Symbol(sym.LPAREN)); }
")" { return (new Symbol(sym.RPAREN)); }
"[" { return (new Symbol(sym.LSQBRACK)); }
"]" { return (new Symbol(sym.RSQBRACK)); }
"*" { return (new Symbol(sym.PLUS)); }
"-" { return (new Symbol(sym.MINUS)); }
"*" { return (new Symbol(sym.TIMES)); }
"mod" { return (new Symbol(sym.MOD)); }
"div" { return (new Symbol(sym.DIV)); }
"\"\" { return (new Symbol(sym.EQUAL_EQUAL)); }
"\!\!" { return (new Symbol(sym.NOT_EQUAL)); }
"<" { return (new Symbol(sym.LESS)); }
"\<=" { return (new Symbol(sym.LESS_EQUAL)); }
">" { return (new Symbol(sym.GREATER)); }
">\=" { return (new Symbol(sym.GREATER_EQUAL)); }
"&&" { return (new Symbol(sym.AND)); }
"||" { return (new Symbol(sym.OR)); }
":=" { return (new Symbol(sym.ASSIGN)); }
":" { return (new Symbol(sym.COLON)); }
"." { return (new Symbol(sym.PERIOD)); }
"skip" { return (new Symbol(sym.SKIP)); }
"if" { return (new Symbol(sym.IF)); }
"then" { return (new Symbol(sym.THEN)); }
"else" { return (new Symbol(sym.ELSE)); }
"end" { return (new Symbol(sym.END)); }
"while" { return (new Symbol(sym.WHILE)); }
"do" { return (new Symbol(sym.DO)); }
"fork" { return (new Symbol(sym.FORK)); }
"true" { return (new Symbol(sym.TRUE)); }
"false" { return (new Symbol(sym.FALSE)); }
"int" { return (new Symbol(sym.INT)); }
"bool" { return (new Symbol(sym.BOOL)); }
"high" { return (new Symbol(sym.HIGH)); }
"low" { return (new Symbol(sym.LOW)); }
"length" { return (new Symbol(sym.LENGTH)); }
```

{NONNEWLINE_WHITE_SPACE_CHAR}+ { }
{DIGIT}+ { return (new Symbol(sym.NUMBER, new Integer(yytext())));} 
{STRING_TEXT} { return (new Symbol(sym.IDENTIFIER, yytext()));}

Grammar file as input for the parser generator Java CUP:

```java
class mwlParser {
    import java_cup.runtime.*;
    import *;
    package mwlParser;
    import java_cup.runtime.*;
    import ast.*;
    /* Terminals (tokens returned by the scanner). */
    terminal LPAREN, RPAREN, LSQBRACK, RSQBRACK;
    terminal SEMI, COMMA, ASSIGN, COLON, PERIOD;
    terminal PLUS, MINUS, TIMES, MOD, DIV;
    terminal LESS, LESS_EQUAL, GREATER, GREATER_EQUAL, EQUAL_EQUAL;
    terminal NOT_EQUAL, AND, OR;
    terminal SKIP, IF, THEN, ELSE, END, WHILE, DO, FORK, LENGTH;
    terminal Boolean TRUE, FALSE;
    terminal String INT, BOOL; // types of variables
    terminal String HIGH, LOW; // types of variables
    terminal String IDENTIFIER;
    terminal Integer NUMBER;
    /* Non terminals */
    non terminal program, progVector, command, decl, varDecl, arrDecl;
    non terminal identifier, expression, boolExp, operator, compOp;
    non terminal atom, arrField, type, secDomain;
    /* Precedences */
    precedence left OR;
    precedence left AND;
    precedence left PLUS, MINUS;
    precedence left TIMES;
    precedence left MOD, DIV;
    /* The grammar */
    program ::= command:c SEMI program:p
        { RESULT = new Program((Command) c, (Program) p); }
    | command:c
        { RESULT = new Program((Command) c); }
    command ::= SKIP { RESULT = new Skip(); }
    | identifier:i ASSIGN expression:e
        { RESULT = new Assign((Identifier) i, (Exp) e); }
    | arrField:af ASSIGN expression:e
        { RESULT = new Assign((ArrField) af, (Exp) e); }
    | decl:d
        { RESULT = d; }
    | IF boolExp:b THEN program:p1 ELSE program:p2 END
        { RESULT = new If((BoolExp) b, (Program) p1, (Program) p2); }
    | IF boolExp:b THEN program:p END
        { RESULT = new If((BoolExp) b, (Program) p); }
    | WHILE boolExp:e DO program:p END
        { RESULT = new While((BoolExp) e, (Program) p); }
    | FORK LPAREN program:p LESS progVector:pv GREATER RPAREN
        { RESULT = new Fork((Program) p, (ProgramVector) pv); }
    identifier ::= IDENTIFIER:i
        { RESULT = new Identifier(i); }
    expression ::= atom:a
        { RESULT = a; }
    | expression:e1 operator:o expression:e2
        { RESULT = new ArithExp((Operator) o, (Exp) e1, (Exp) e2); }
    | LPAREN expression:e RPAREN
        { RESULT = e; }
```
atom ::= NUMBER:n {: RESULT = new Int(n.intValue()); :}
| TRUE {: RESULT = new True(); :}
| FALSE {: RESULT = new False(); :}
| IDENTIFIER:arr PERIOD LENGTH {: RESULT = new ArrLength(arr); :}
| arrField:af {: RESULT = af; :}
| identifier:i {: RESULT = i; :};

arrField ::= IDENTIFIER:i LSQBRACK expression:e RSQBRACK
{: RESULT = new ArrField(i, (Exp) e); :};

operator ::= PLUS {: RESULT = new Plus(); :}
| MINUS {: RESULT = new Minus(); :}
| TIMES {: RESULT = new Times(); :}
| MOD {: RESULT = new Modulus(); :}
| DIV {: RESULT = new Div(); :};
decl ::= varDecl:v {: RESULT = v; :}
| arrDecl:a {: RESULT = a; :};

varDecl ::= identifier:i COLON type:t COLON secDomain:s
{: RESULT = new VarDecl((Identifier) i, (Type) t, (SecDomain) s); :};

arrDecl ::= identifier:i COLON type:t LSQBRACK NUMBER:l RSQBRACK COLON secDomain:s
{: RESULT = new ArrDecl((Identifier) i, l.intValue(), (Type) t, (SecDomain) s); :}
| identifier:i COLON type:t LSQBRACK identifier:l RSQBRACK COLON secDomain:s
{: RESULT = new ArrDecl((Identifier) i, (Identifier) l, (Type) t, (SecDomain) s); :};
type ::= INT {: RESULT = new IntegerType(); :}
| BOOL {: RESULT = new BooleanType(); :};

secDomain ::= HIGH {: RESULT = new HighSec(); :}
| LOW {: RESULT = new LowSec(); :};

boolExp ::= TRUE {: RESULT = new True(); :}
| FALSE {: RESULT = new False(); :}
| atom:a1 compOp:o atom:a2
{: RESULT = new CompExp((BoolOp) o, (Exp) a1, (Exp) a2); :}
| LPAREN boolExp:b RPAREN {: RESULT = b; :}
| boolExp:b1 AND:o boolExp:b2
{: RESULT = new CompExp((BoolOp) new And(), (Exp) b1, (Exp) b2); :}
| boolExp:b1 OR:o boolExp:b2
{: RESULT = new CompExp((BoolOp) new Or(), (Exp) b1, (Exp) b2); :};

compOp ::= LESS {: RESULT = new Less(); :}
| LESS_EQUAL {: RESULT = new LessEqual(); :}
| GREATER {: RESULT = new Greater(); :}
| GREATER_EQUAL {: RESULT = new GreaterEqual(); :}
| EQUAL_EQUAL {: RESULT = new EqualEqual(); :}
| NOT_EQUAL {: RESULT = new NotEqual(); :};

progVector ::= program:p COMMA progVector:pv
{: RESULT = new ProgramVector((Program) p, (ProgramVector) pv); :}
| program:p
{: RESULT = new ProgramVector((Program) p); :};
C.2 Package AST

Example classes from the ast package: Assign and Program

```java
package ast;

import visitor.Visitor;

/**
 * Assign represents an assignment of an expression to an
 * identifier or to an array field.
 * @author Christina Pöpper
 * @version 1.0
 */
public class Assign extends PrimitiveCommand {
    private Assignable ass;
    private Exp exp;

    // Constructors
    // ---------------------------------------------------------------------------
    /**
     * Assignment to an Assignable, which may be an
     * <code>Identifier</code> or an <code>ArrField</code>.
     * @param ass the assignable
     * @param exp the expression
     */
    public Assign (Assignable ass, Exp exp) {
        this.ass = ass;
        this.exp = exp;
    }

    private Assign () {}

    // Public Methods
    // ---------------------------------------------------------------------------

    /**
     * Getter method for the variable.
     * @return the variable
     */
    public Assignable getVariable () {
        return ass;
    }

    /**
     * Getter method for the expression.
     * @return the expression
     */
    public Exp getExp () {
        return exp;
    }

    /**
     * Clones this instance.
     * @return the clone, which is a new <code>Assign</code>
     */
    public Assign clone () {
        return new Assign (this.ass, this.exp);
    }
}
```
package ast;

import visitor.Visitor;

/**
 * Program represents an MWL program which is composed of single
 * commands. A <code>Program</code> contains the first command of the
 * sequential composition and the rest of the program. If there is
 * only one command, then the rest of the program is null.
 * 
 * @author Christina Pöpper
 * @version 1.0
 */
public class Program implements Cloneable, MwlElement {
    private Command comm;
    private Program prog;

    // Constructors
    //-------------------------------------------------------------------------
    /**
     * @param comm the command
     * @param prog the program
     */
    public Program (Command comm, Program prog) {
        this.comm = comm;
        this.prog = prog;
    }

    /**
     * If the program consists of a single command.
     *
     * @param comm the command
     */

public Program (Command comm) {
    this.comm = comm;
    this.prog = null;
}

private Program () {}
public Object clone () {
    if (prog != null)
        return new Program((Command) comm.clone(),
                            (Program) prog.clone());
    else
        return new Program((Command) comm.clone());
}

/**
 * Represents this class as a string.
 * @return the string representation of this arithmetic expression
 */
public String toString () {
    StringBuffer s = new StringBuffer();
    if (comm != null) s.append(comm.toString());
    if (prog != null) s.append(prog.toString());
    return s.toString();
}

/**
 * Accept method for the visitor design pattern.
 * @param v a visitor
 * @return the result of the visitors visit method
 */
public Object accept (Visitor v) {
    return v.visit(this);
}
C.3 Package Checker

Classes from the checker package:Analyzer, Checker, CheckerException, CheckerNoTransform, and CheckerTransform

package checker;

/**
 * The analyzer for the security checks.
 *
 * @author Christina Pöpper
 * @version 1.0
 */
public class Analyzer {

    /**
     * Main method for calling concrete <code>Checker</code> instances.
     *
     * @param argv Array of String. Should contain the name of
     * the MWL file to be checked as first element.
     */
    public static void main (String[] argv) throws Exception {
        if (argv.length < 1)
            throw new Exception("Wrong number of arguments: "+
                    "java Test <mwl-filename>");
        String filename = argv[0];
        Checker checker = new CheckerNoTransform(filename);
        boolean res = checker.check();
        System.out.println("==> The program "+filename+" is "+
                boolToString(res)+"secure.");
        if (! res) {
            Checker checkerTr = new CheckerTransform(argv[0]);
            boolean res2 = checkerTr.check();
            System.out.println("==> The program "+argv[0]+" can "+
                    boolToString(res2)+"be transformed into a secure program.");
            if (res2)
                System.out.println("The transformed program can be "+
                        "found in "+filename+"_secure");
        }
    }

    /**
     * Helper method returning the String "NOT " in case of a false
     * boolean value and the empty String otherwise.
     *
     * @param b a truth value
     * @return the String
     */
    private static String boolToString (boolean b) {
        if (! b) return "NOT ";
        return " ";
    }

    import ast.*;
    import java_cup.runtime.Scanner;
    import java_cup.runtime.Symbol;
    import java.util.HashMap;
    import java.util.Map;
    import mwlParser.MWLParser;
import mwlParser.MWLScanner;
import visitor.Visitor;

/**
 * Abstract super class of all security checkers.
 * @author Christina Pöpper
 * @version 1.0
 */
public abstract class Checker {
    //-------------------------------------------------------------------------
    // Constants and Declarations
    //-------------------------------------------------------------------------
    Map hMap; // contains the variable declarations
    // key: variable name, value: Decl
    String filename;
    Program root;

    //-------------------------------------------------------------------------
    // Abstract Methods
    //-------------------------------------------------------------------------
    /**
     * Returns if a program has secure information flow or can potentially
     * have it after some transformations by applying transformation rules.
     * The file containing the program was to be set in the constructor.
     *
     * @throws CheckerException if the program cannot be checked
     * @return <code>true</code> if the program has secure
     * information flow, <code>false</code> otherwise
     */
    public abstract boolean check () throws CheckerException;

    //-------------------------------------------------------------------------
    // Public Methods
    //-------------------------------------------------------------------------
    /**
     * @param file the filename containing the MWL program
     * @throws Exception if the MWL program cannot be parsed
     */
    public Checker (String file) throws Exception {
        this.filename = file;
        Scanner scanner = new MWLScanner(new java.io.FileInputStream(file));
        Symbol sym = new MWLParser(scanner).parse();
        root = (Program) sym.value;
        readDecls(root);
    }

    private Checker () {};

    //-------------------------------------------------------------------------
    // Private Methods
    //-------------------------------------------------------------------------
    /**
     * Inserts all declarations into the local <code>Map</code>.
     * It traverses the AST tree program to find them. The variable
     * name is the key of the HashMap, <code>Decl</code> the value.
     *
     * @param p a <code>Program</code> for which the variable
     * declarations are to be found
     */

private void readDecls (Program p) {
    Command c;
    hMap = new HashMap();
    while (p != null) {
        c = p.getCommand();
        if ((c != null) && (c instanceof Decl)) {
            // insert this Decl into the Map hMap
            hMap.put(((Decl) c).getIdentifier().getName(), c);
        }
        p = p.getProgram();
    }
}

package checker;
import java.lang.Exception;

/**
 * A simple <code>Exception</code> that is thrown during execution of the
 * security check.
 * @author Christina Pöpper
 * @version 1
 */
class CheckerException extends Exception {
    public CheckerException () {
    }
    public CheckerException (String msg) {
        super(msg);
    }
}

package checker;
import ast.*;
import visitor.NoTransformVisitor;
import visitor.Visitor;

/**
 * A security checker for MWL programs that checks programs for
 * direct and indirect information flow without transforming
 * them. The result is returned by the <code>check</code> method.
 * @author Christina Pöpper
 * @version 1.0
 */
public class CheckerNoTransform extends Checker {
    // Constructor
    //------------------------------
    /**
     * @param file the filename containing the MWL program
     * @throws Exception if the MWL program cannot be parsed
     */
    public CheckerNoTransform (String file) throws Exception {
        super(file);
    }
}
// Public Methods

public boolean check() throws CheckerException {
    if (super.root == null) {
        throw new CheckerException("Abstract syntax tree is null");
    }
    Programp = super.root;
    Visitor visitor = new NoTransformVisitor(super.hMap);
    visitor.accept(visitor);
    return ((Boolean) visitor.getResult()).booleanValue();
}

package checker;

import ast.Program;
import java.io.FileOutputStream;
import java.io.PrintStream;
import visitor.ElementPair;
import visitor.ToStringVisitor;
import visitor.TransformVisitor;
import visitor.Visitor;

/**
 * A security checker for MWL programs that checks programs for direct and
 * indirect information flow and timing leaks, potentially transforming the
 * program. If the result returned by the <code>check</code> method is
 * <code>true</code>, the program was already secure or a transformation
 * was found which makes the program secure.
 * @author Christina Pöpper
 * @version 1.0
 */
public class CheckerTransform extends Checker {
    // Constructor
    public CheckerTransform (String file) throws Exception {
        super(file);
    }

    // Public Methods
    public boolean check() throws CheckerException {
        if (super.root == null) {
            throw new CheckerException("Abstract syntax tree is null");
        }
        Programp = super.root;
        Visitor visitor = new NoTransformVisitor(super.hMap);
        visitor.accept(visitor);
        return ((Boolean) visitor.getResult()).booleanValue();
    }
}
public boolean check () throws CheckerException {
    if (super.root == null) {
        throw new CheckerException("Abstract syntax tree is null");
    }

    Visitor visitor = new TransformVisitor(super.hMap);
    ElementPair ep = (ElementPair) super.root.accept(visitor);
    Program transfProg = (Program) ep.getElement();
    boolean result = ((Boolean) visitor.getResult()).booleanValue();
    if (result == true)
        writeTransformed(transfProg, "secure");

    return result;
}

private void writeTransformed (Program p, String appendix) {
    FileOutputStream out; // a file output object
    PrintStream pStream; // a print stream object
    ToStringVisitor stringVis = new ToStringVisitor();
    p.accept(stringVis);
    String s = (String) stringVis.getResult();
    try {
        // Create a new file output stream
        out = new FileOutputStream(super.filename + "_" + appendix);
        // Connect print stream to the output stream
        pStream = new PrintStream(out);
        pStream.println(s);
        pStream.close();
    } catch (Exception e) {
        System.err.println("Error writing to file");
    }
}
### C Source Code

**C.4 Package Visitor**

Classes from the visitor package: `ElementPair`, `ExpVisitor`, `NoTransformVisitor`, `ToStringVisitor`, `TransformVisitor`, and `Visitor`

```java
package visitor;
import ast.MwlElement;

/**
 * ElementPair instances are the 'return values' of the transform function:
 * They consist of an `MwlElement` (the transformed
 * @link ast.MwlElement), its type (low slice) which is also an
 * `MwlElement` and a flag indicating whether there is an
 * assignment to a low variable in the transformed `MwlElement`.
 * @author Christina Pöpper
 * @version 1.0
 */
public class ElementPair {

    // Constants and Declarations
    private MwlElement el; // el represents a transformed MwlElement
    private MwlElement type; // type represents the type (low
        // slice) of a transformed MwlElement
    private boolean assToLow; // is there a low assignment to in Element el

    // Constructors
    public ElementPair (MwlElement e1, MwlElement e2, boolean assToL) {
        el = e1;
        type = e2;
        assToLow = assToL;
    }

    // Public Methods
    public MwlElement getElement () {
        return el;
    }

    public MwlElement getType () {
    }

    // Internal Methods
    // ...
}
```

---

**Package Visitor**

Classes from the visitor package: `ElementPair`, `ExpVisitor`, `NoTransformVisitor`, `ToStringVisitor`, `TransformVisitor`, and `Visitor`
return type;
}

/**
 * Getter method for the assignment-to-low flag. It indicates if there is
 * an assignment to a low variable in the MwElement.
 *
 * @return the flag indicating an assignment to a low variable
 */
public boolean getAlFlag () {
    return assToLow;
}

/**
 * Represents this class as a string.
 *
 * @return the String representation of this ElementPair</code>
 */
public String toString() {
    return "MWLElement: "+getElement()+" of type: "+getType()+" AlFlag:"+getAlFlag();
}

package visitor;
import ast.*;
import java.util.Map;

/**
 * ExpVisitor implements the check logic for expressions, determining the
 * security domain of an expression. It realizes implicitly the following
 * rules:
 * <ul>
 * <li>[Arith_low]</li>
 * <li>[Arith_high1]</li>
 * <li>[Arith_high2]</li>
 * <li>[Exp]</li>
 * </ul>
 * The basic idea is: In the beginning, we assume each expression to be of
 * security domain LOW, i.e. the result is set to LOW. For each expression
 * and expression part (such as <code>Identifier</code>, <code>Int</code> etc.) we encounter we try to apply the security check rules. Once the
 * result is HIGH, it remains HIGH till the end of the execution.
 * Each <code>visit</code>() method returns <code>null</code> because the
 * return value is of no interest, only the result variable
 * containing the security domain counts.
 * @author Christina Pöpper
 * @version 1.0
 */
public class ExpVisitor extends AbstractVisitor {
    private static final int LOW = Visitor.LOW;
    private static final int HIGH = Visitor.HIGH;
    private int secDomResult;
    private Map decls;
    //--Constants and Declarations
    //---
// Constructors
//-------------------------------------------------------------------------
/**
 * @param decls a Map containing the declarations for the
 * program to be visited
 */
public ExpVisitor (Map decls) {
    secDomResult = LOW;
    this.decls = decls;
}

// Public Methods
//-------------------------------------------------------------------------
/**
 * Returns an Object containing the result of the analysis.
 *
 *@return an Integer instance containing int HIGH if the expression is
 * of security domain high and int LOW otherwise.
 */
public Object getResult() {
    return new Integer(secDomResult);
}

/**
 * Modifies the result according to the security domain of this
 * <code>True</code> truth value.
 *
 *@param t a <code>True</code> truth value
 * @return null
 */
public Object visit (True t) {
    setSecDom(LOW);
    return null;
}

/**
 * Modifies the result according to the security domain of this
 * <code>False</code> truth value.
 *
 *@param f a <code>False</code> truth value
 * @return null
 */
public Object visit (False f) {
    setSecDom(LOW);
    return null;
}

/**
 * Modifies the result according to the security domain of this
 * <code>Int</code>.
 *
 *@param i an <code>Int</code>
 * @return null
 */
public Object visit (Int i) {
    setSecDom(LOW);
    return null;
}
* Modifies the result according to the security domain of this
  * call.
  *
  * @param al an \texttt{ArrLength}
  * @return null
  */

public Object visit (ArrLength al) {
  // In the current used type systems, the length of an array is
  // always public and thus has security domain low.
  setSecDom(LOW);
  return null;
}

/**
 * Modifies the result according to the security domain of this
 * \texttt{Identifier}.
 *
 * @param id an \texttt{Identifier}
 * @return null
 */

public Object visit (Identifier id) {
  String name = id.getName();
  Decl decl = (Decl) decls.get(name);
  // Check whether the identifier has actually been declared as
  // a variable
  if (! (decl instanceof VarDecl)) {
    System.err.println("ERROR: Identifier "+name+" is not declared as identifier! Aborting check.");
    System.exit(1);
  }
  // set the security domain according to the security domain of
  // this variable
  if (decl != null)
    setSecDom(decl.getSecDomain());
  else {
    System.err.println("WARNING: Variable "+name+" is not declared! Security Domain is set to HIGH for "+name);
    setSecDom(HIGH);
  }
  return null;
}

/**
 * Modifies the result according to the security domain of this
 * \texttt{ArrField}.
 *
 * @param af an \texttt{ArrField}
 * @return null
 */

public Object visit (ArrField af) {
  String name = af.getName();
  Decl decl = (Decl) decls.get(name);
  // Check if the identifier has actually been declared as array,
  // respectively has been declared at all. This catches also a null
  // decl.
  if (! (decl instanceof ArrDecl)) {
    System.err.println("ERROR: Identifier "+name+" is not declared"+
                        " as array! Aborting check.");
    System.exit(1);
  }
}
Set the security domain according to the security domain of the index and of the contents of the array.

Array: H | H | L | L
Index: H | L | H | L

result: H | H | XXX | L

ExpVisitor indexV = new ExpVisitor(decls);
Exp index = af.getIndex();
index.accept(indexV);
int sdIndex = ((Integer) indexV.getResult()).intValue();

SecDomain sdArray = decl.getSecDomain();
if (sdArray instanceof HighSec)
    setSecDom(HIGH);
else {
    if (sdIndex == LOW)
        setSecDom(LOW);
    else {
        System.out.println(af + " has a high index but the array " +
                           "is low, which is forbidden by the " +
                           "security policy.");
        System.out.println("The program is NOT secure! It +
                           "cannot be transformed into a secure one.");
        System.exit(0);
    }
}

return null;

/**
 * Splits up an <code>ArithExp</code> into the two parts it consists of
 * and enqueues the analysis for both of them with itself.
 *
 * @param ae an <code>ArithExp</code>
 * @return null
 */
public Object visit (ArithExp ae) {
    ae.getExp1().accept(this);
    ae.getExp2().accept(this);
    return null;
}

/**
 * Splits up a <code>CompExp</code> into the two parts it consists of
 * and enqueues the analysis for both of them with itself.
 *
 * @param ce an <code>CompExp</code>
 * @return null
 */
public Object visit (CompExp ce) {
    ce.getExp1().accept(this);
    ce.getExp2().accept(this);
    return null;
}

/**
 * This method will be called if none of the more specialized visit methods could be called. This should never happen actually.
 * <p>
* This has no effect on the result. Simply nothing will be done and
* the check will executed further.
* @param o an <code>Object</code>
* @return null
*/
public Object visit (Object o) {
    System.out.println("ExpVisitor: This Method should never be called!");
    return null;
}

/**
* Returns the Map with the variable declarations.
* @return a Map containing the variable declarations.
*/
public Map getDecls () {
    return decls;
}

// Private Methods
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
/**
* Sets the secDomResult variable depending on its current
* value and a security domain of a new Exp.
* @param int the security domain found in the new Exp
*/
private void setSecDom(int sd) {
    if ((secDomResult == HIGH) || (sd == HIGH))
        secDomResult = HIGH;
    else
        secDomResult = LOW;
}

/**
* Calls the setSecDom(int) method with the appropriate
* parameter HIGH or LOW depending on the class of sd.
* @param sd the security domain of an identifier
*/
private void setSecDom(SecDomain sd) {
    if (sd instanceof HighSec)
        setSecDom(HIGH);
    else
        setSecDom(LOW);
}

/**
* Returns the security domain of an expression that is composed
* of two parts, whose security domains are given.
* @param int sd1 the security domain of the first expression part
* @param int sd2 the security domain of the second expression part
* @return the security domain of an expression composed of these two parts.
*/
private int getCombSecDomain(int sd1, int sd2) {
    if ((sd1 == HIGH) || (sd2 == HIGH))
        return HIGH;
    return LOW;
}
package visitor;
import ast.*;
import java.util.Map;

/**
 * NoTransformVisitor implements the check logic for security checks of
 * programs without transformation. It realizes the following rules:
 * <ul>
 * <li>[Var]</li>
 * <li>[Skip]</li>
 * <li>[Assign_high]</li>
 * <li>[Assign_low]</li>
 * <li>[If_low]</li>
 * <li>[While_low]</li>
 * <li>[Fork]</li>
 * </ul>
 * The basic idea is: In the beginning, we assume the program to be secure,
 * i.e. the result is set to <code>true</code>. For each command we encounter
 * we try to apply the security check rules. The result is set to
 * <code>false</code> if the command is insecure or it remains unmodified
 * if the command is secure. Thus, once the result is false, it remains false
 * till the end of the execution.
 * <p>
 * Each <code>visit</code> method returns <code>null</code> because
 * the return value is of no interest, only the result variable counts.
 * <p>
 * @author Christina Pöpper
 * @version 1.0
 */

public class NoTransformVisitor extends AbstractVisitor {

    // Constants and Declarations
    private static final int LOW = Visitor.LOW;
    private static final int HIGH = Visitor.HIGH;
    private boolean checkResult; // the check result for this program
    private boolean notFound; // no insecure command found yet
    private Map decls;

    // Constructors
    public NoTransformVisitor (Map decls) {
        checkResult = true;
        notFound = true;
        this.decls = decls;
    }

    // Public Methods
    //------------------------------------------------------------------------------
    /**
     * Returns an Object containing the result of the check.
     */
public Object getResult () {
    return new Boolean(checkResult);
}

/**
 * Implements the [Skip] rule.
 *
 * @param s a <code>Skip</code> command
 * @return null
 */
public Object visit (Skip s) {
    return null;
}

/**
 * Implements the [Var] rule.
 *
 * @param vd a <code>VarDecl</code> command
 * @return null
 */
public Object visit (VarDecl vd) {
    return null;
}

/**
 * Implements the [ArrDecl] rule.
 *
 * @param ad a <code>VarDecl</code> command
 * @return null
 */
public Object visit (ArrDecl ad) {
    return null;
}

/**
 * Implements the [Assign_high] and [Assign_low] rules.
 *
 * @param as an <code>Assign</code> command
 * @return null
 */
public Object visit (Assign as) {
    ExpVisitor idV = new ExpVisitor(decls);
    as.getVariable().accept(idV);
    int sdId = ((Integer) idV.getResult()).intValue();
    ExpVisitor expV = new ExpVisitor(decls);
    as.getExp().accept(expV);
    int sdExp = ((Integer) expV.getResult()).intValue();
    if (sdId == HIGH)
        checkResult = checkResult & true; // [Assign_high]
    else if ((sdId == LOW) && (sdExp == LOW))
        checkResult = checkResult & true; // [Assign_low]
    else {
        setFound("The first insecure command is a high assignment " +
                "to a low variable: \n"+as+"\n");
        checkResult = false;
    }
/**
 * Implements the \[If_low\] rule.
 *
 * @param i an \<code>If</code> command
 * @return null
 */
public Object visit (If i) {
    ExpVisitor boolV = new ExpVisitor(decls);
    i.getBoolExp().accept(boolV);
    int sdBool = ((Integer) boolV.getResult()).intValue();
    if (sdBool == LOW) { // \[If_low\]
        i.getIfProgram().accept(this);
        Program pElse = i.getElseProgram();
        if (pElse != null)
            pElse.accept(this);
        checkResult = checkResult & true;
    } else {
        setFound("The first insecure command is a high if conditional:
                  \n\n" + i);
        checkResult = false;
    }
    return null;
}

/**
 * Implements the \[While_low\] rule.
 *
 * @param wh a \<code>While</code> command
 * @return null
 */
public Object visit (While wh) {
    ExpVisitor boolV = new ExpVisitor(decls);
    wh.getBoolExp().accept(boolV);
    int sdBool = ((Integer) boolV.getResult()).intValue();
    if (sdBool == LOW) // \[While_low\]
        checkResult = checkResult & true;
    else {
        setFound("The first insecure command is a high while loop:
                  \n\n" + wh);
        checkResult = false;
    }
    wh.getProgram().accept(this);
    return null;
}

/**
 * Implements the \[Fork\] rule.
 *
 * @param fk a \<code>Fork</code> command
 * @return null
 */
public Object visit (Fork fk) { // \[Fork\]
    Program p = fk.getProgram();
    ProgramVector pv = fk.getProgramVector();
    p.accept(this);
    if (pv != null)
        pv.accept(this);
return null;
}
/**
 * Splits up the check for a Program into the check of its Command
 * and the check for the next Program. Implements the [Seq] rule
 * implicitly.
 *
 * @param p a <code>Program</code>
 * @return null
 */
public Object visit (Program p) {
    Command c = p.getCommand();
    Program nextP = p.getProgram();
    c.accept(this);
    if (nextP != null)
        nextP.accept(this);
    return null;
}
/**
 * Splits up the check for a ProgramVector into the check of its
 * Program and the check for the next ProgramVector.
 *
 * @param pv a <code>ProgramVector</code>
 * @return null
 */
public Object visit (ProgramVector pv) {
    Program p = pv.getProgram();
    ProgramVector nextPv = pv.getProgramVector();
    p.accept(this);
    if (nextPv != null)
        nextPv.accept(this);
    return null;
}
/**
 * This method will be called if none of the more specialized
 * visit methods could be called. This should never happen
 * actually.
 * @param o an <code>Object</code>
 * @return null
 */
public Object visit (Object o) {
    System.err.println("NoTransformVisitor: This method should never be called!");
    return null;
}

// Private Methods
private void setFound (String s) {
    if (notFound) {
        System.out.println(s);
        notFound = false;
    }
}
package visitor;
import ast.*;
import java.lang.StringBuffer;
import java.util.Map;

/**
 * ToStringVisitor produces a string representation of a <code>Program</code>. 
 * The main application for this visitor is to transform a program into a 
 * String that can be printed into a file. 
 * <p>
 * Each <code>visit</code> method returns <code>null</code> because the return 
 * value is of no interest, only the result variable containing the String 
 * representation of the program counts. 
 * <p>
 * @author Christina Pöpper 
 * @version 1.0 
 */
public class ToStringVisitor extends AbstractVisitor {
    //-------------------------------------------------------------------------
    // Constants and Declarations
    //-------------------------------------------------------------------------
    private StringBuffer buf;
    private int indents;

    //-------------------------------------------------------------------------
    // Constructors
    //-------------------------------------------------------------------------
    public ToStringVisitor () {
        buf = new StringBuffer();
        indents = 0;
    }

    //-------------------------------------------------------------------------
    // Public Methods
    //-------------------------------------------------------------------------
    /**
     * Returns an Object containing the result of the visits for printing. 
     * @return a String representation of this program. 
     */
    public Object getResult () {
        return buf.toString();
    }

    /**
     * Prints a <code>True</code> truth value. 
     * @param t a <code>True</code> truth value 
     * @return null 
     */
    public Object visit (True t) {
        buf.append("true");
        return null;
    }

    /**
     * Prints a <code>False</code> truth value. 
     * @param f a <code>False</code> truth value 
     */
    public void visit (false f) {
        buf.append("false");
        return null;
    }
}
public Object visit (False f) {
    buf.append("false");
    return null;
}

/**
 * Prints an <code>Int</code>.
 * @param i the <code>Int</code>
 * @return null
 */
public Object visit (Int i) {
    buf.append(i.getInt());
    return null;
}

/**
 * Prints an <code>Identifier</code>.
 * @param id the <code>Identifier</code>
 * @return null
 */
public Object visit (Identifier id) {
    buf.append(id.getName());
    return null;
}

/**
 * Prints an <code>ArrLength</code>, the call to the length of an array.
 * @param al the <code>ArrLength</code>
 * @return null
 */
public Object visit (ArrLength al) {
    buf.append(al.toString());
    return null;
}

/**
 * Prints an <code>ArrField</code>, an array field.
 * @param af the <code>ArrField</code>
 * @return null
 */
public Object visit (ArrField af) {
    buf.append(af.toString());
    return null;
}

/**
 * Splits up an <code>ArithExp</code> into the two parts it consists of
 * and envoques the analysis for both of them with itself.
 * @param ae an <code>ArithExp</code>
 * @return null
 */
public Object visit (ArithExp ae) {
    buf.append("(");
    ae.getExp1().accept(this);
    buf.append(ae.getOperator().toString());
    ae.getExp2().accept(this);
    return null;
}
ae.getExp2().accept(this);
    buf.append("=");
    return null;
}

/**
   * Splits up a <code>CompExp</code> into the two parts it consists of
   * and enqueues the analysis for both of them with itself.
   *
   * @param ce an <code>CompExp</code>
   * @return null
   */
public Object visit (CompExp ce) {
    ce.getExp1().accept(this);
    buf.append(ce.getBoolOp().toString());
    ce.getExp2().accept(this);
    return null;
}

/**
   * Prints a <code>Skip</code>.
   *
   * @param s a <code>Skip</code> command
   * @return null
   */
public Object visit (Skip s) {
    buf.append(s.toString());
    return null;
}

/**
   * Prints a variable declaration <code>VarDecl</code>.
   *
   * @param vd a <code>VarDecl</code> command
   * @return null
   */
public Object visit (VarDecl vd) {
    vd.getIdentifier().accept(this);
    buf.append(":"+vd.getType().toString());
    buf.append(":"+vd.getSecDomain().toString());
    return null;
}

/**
   * Prints an array declaration <code>ArrDecl</code>.
   *
   * @param ad a <code>ArrDecl</code> command
   * @return null
   */
public Object visit (ArrDecl ad) {
    ad.getIdentifier().accept(this);
    buf.append(":"+ad.getType().toString()+"["+ad.getLength()+"]");
    buf.append(":"+ad.getSecDomain().toString());
    return null;
}

/**
   * Prints an <code>Assign</code> command.
   *
   * @param as an <code>Assign</code> command
   * @return null
   */
public Object visit (Assign as) {
    as.getVariable().accept(this);
    buf.append("=");
    as.getExp().accept(this);
    return null;
}

/**
 * Prints an <code>If</code> command.
 * @param i an <code>If</code> command
 * @return null
 */
public Object visit (If i) {
    buf.append("if ");
    i.getBoolExp().accept(this);
    buf.append(" then\n");
    indents++;
    i.getIfProgram().accept(this);
    indents--;
    Program pElse = i.getElseProgram();
    if (pElse != null) {
        buf.append("\n");
        printIndent();
        buf.append("else\n");
        indents++;
        pElse.accept(this);
        indents--;
    }
    buf.append("\n");
    printIndent();
    buf.append("end");
    return null;
}

/**
 * Prints a <code>While</code> command.
 * @param wh a <code>While</code> command
 * @return null
 */
public Object visit (While wh) {
    buf.append("while ");
    wh.getBoolExp().accept(this);
    buf.append(" do\n");
    indents++;
    wh.getProgram().accept(this);
    indents--;
    buf.append("\n");
    printIndent();
    buf.append("end");
    return null;
}

/**
 * Prints a <code>Fork</code> command.
 * @param fk a <code>Fork</code> command
 * @return null
 */
public Object visit (Fork fk) {
    buf.append("fork\n");
indents++;  
fk.getProgram().accept(this);  
buf.append("\n");  
printIndent();  
buf.append("\\n");  
fk.getProgramVector().accept(this);  
buf.append("\n");  
printIndent();  
buf.append("\"");  
indents--;
return null;
}

/**
 * Prints a <code>Program</code>.
 */
public Object visit (Program p) {
printIndent();
Command c = p.getCommand();
c.accept(this);
Program nextP = p.getProgram();
if (nextP != null) {
    buf.append(";\n");
    nextP.accept(this);
}
return null;
}

/**
 * Splits up the check for a ProgramVector into the check of its
 * Program and the check for the next ProgramVector.
 */
public Object visit (ProgramVector pv) {
    Program p = pv.getProgram();
p.accept(this);
ProgramVector nextPv = pv.getProgramVector();
if (nextPv != null) {
    buf.append(",\n");
    nextPv.accept(this);
}
return null;
}

/**
 * This method will be called if none of the more specialized visit
 * methods could be called. This should never happen actually.
 *<p>
 * A method call has no effect on the result. Nothing will be done and
 * the check will executed further.<p>
 */
public Object visit (Object o) {
    System.err.println("ToStringVisitor: This method should never "+
    "be called!");
}
```java
public class TransformVisitor extends AbstractVisitor {

    private static final int LOW = Visitor.LOW;
    private static final int HIGH = Visitor.HIGH;

    private boolean checkResult;  // if a program is secure or can be
                                 // transformed into a secure one

    private Map decls;          // a Map containing the variable declarations
    private Map decls;          // for the program to be visited

    // Constant and Declerations
    // Constructor
    public TransformVisitor (Map decls) {
```
checkResult = true;
this.decls = decls;

public Object getResult () {
    return new Boolean(checkResult);
}

/**
 * Implements the [Skip] rule.
 * @param s a <code>Skip</code> command
 * @return the resulting <code>ElementPair</code> from the transformation
 */
public Object visit (Skip s) {
    Skip sNew = (Skip) s.clone();
    Skip sNewType = (Skip) s.clone();
    return new ElementPair(sNew, sNewType, false);
}

/**
 * Implements the [Var] rule.
 * @param vd a <code>VarDecl</code> command
 * @return the resulting <code>ElementPair</code> from the transformation
 */
public Object visit (VarDecl vd) {
    VarDecl vNew = (VarDecl) vd.clone();
    VarDecl vNewType = (VarDecl) vd.clone();
    return new ElementPair(vNew, vNewType, false);
}

/**
 * Implements the [ArrDecl] rule.
 * @param ad an <code>ArrDecl</code> command
 * @return the resulting <code>ElementPair</code> from the transformation
 */
public Object visit (ArrDecl ad) {
    ArrDecl aNew = (ArrDecl) ad.clone();
    ArrDecl aNewType = (ArrDecl) ad.clone();
    return new ElementPair(aNew, aNewType, false);
}

/**
 * Implements the [Assign_high] and [Assign_low] rules.
 * @param as a <code>Assign</code> command
 * @return the resulting <code>ElementPair</code> from the transformation
 */
/*
 * Implements the [If_low] and [If_high] rules.
 * 
 * @param i an <code>if</code> command
 * @return the resulting <code>ElementPair</code> from the
 *         transformation, <code>null</code> in case of error
 */

public Object visit (If i) {
    // First we get the two programs of the if-command.
    Program pIf = i.getIfProgram();
    Program pElse = i.getElseProgram();

    // Then we invoke the transformation of the programs which has
    // to be done anyway regardless of the security type of the
    // boolean expression.
    ElementPair pIfPair = (ElementPair) pIf.accept(this);
    ElementPair pElsePair = null;
    if (pElse != null)
        pElsePair = (ElementPair) pElse.accept(this);

    // We get the transformed if-program, its type and the flag
    // if there is an assignment to low...
    Program pIfProg = (Program) pIfPair.getElement();
    Program pIfType = (Program) pIfPair.getType();
    boolean pIfAl = pIfPair.getAlFlag();
// ... and do the same for the else-program.
Program pElseProg = null;
Program pElseType = null;
boolean pElseAl = false;
if (pElsePair != null) {
    pElseProg = (Program) pElsePair.getElement();
    pElseType = (Program) pElsePair.getType();
    pElseAl = pElsePair.getAlFlag();
}

ElementPair elPair = null; // will contain the final result

// Then we get the security domain of the boolean expression.
ExpVisitor boolV = new ExpVisitor(decls);
BoolExp be = i.getBoolExp();
be.accept(boolV);
int sdBool = ((Integer) boolV.getResult()).intValue();
if (sdBool == LOW) { // [If_low]
    // In the case of a low conditional we finally construct
    // the new if-command, its type and the resulting
    // ElementPair to be returned.
    Command ifNew = new If((BoolExp) be.clone(), pIfProg, pElseProg);
    Command ifNewType = new If((BoolExp) be.clone(), pIfType,
                                pElseType);
    elPair = new ElementPair(ifNew, ifNewType, pIfAl || pElseAl);
} else if (sdBool == HIGH) { // [If_high]
    // We need to check if there are assignments to low
    // variables in the two programs.
    boolean al = pIfAl || pElseAl;

    // We create a new (transformed) if command and its
    // type. First we clone Sl_1 and Sl_2, because we need it
    // for the transformed command as well as for its type.
    Program pIfTypeClone = (Program) pIfType.clone();
    Program pElseTypeClone = null;
    if (pElseType != null)
        pElseTypeClone = (Program) pElseType.clone();
    // Compose the "if B then C_1';Sl_2 else Sl_1;C_2'"
    pIfProg.append(pElseType);
    pIfType.append(pElseProg);
    Command ifNew = new If((BoolExp) be.clone(), pIfProg, pIfType);

    // Compose the "skip;Sl_1;Sl_2"
    pIfTypeClone.append(pElseTypeClone);
    Program ifNewType = new Program(new Skip(), pIfTypeClone);
    if (al!=false)
        // There is an assignment to a low variable in the
        // programs -> the program cannot be typed
        checkResult = false;
    elPair = new ElementPair(ifNew, ifNewType, false);
} else {
    System.err.println("Undefined security domain "+sdBool);
    checkResult = false;
    elPair = null;
}
return elPair;
/**
 * Implements the [While_low] rule.
 * @param wh a <code>While</code> command
 * @return the resulting <code>ElementPair</code> from the transformation
 */
public Object visit (While wh) {
    // First we get the program of the while-command.
    Program p = wh.getProgram();

    // Now we invoke the transformation of the program
    ElementPair pPair = (ElementPair) p.accept(this);

    // We get the transformed program, its security type
    // and the flag whether there is an assignment to low.
    Program pWhileProg = (Program) pPair.getElement();
    Program pWhileType = (Program) pPair.getType();
    boolean pWhileAl = pPair.getAlFlag();

    // Then we get the security domain of the boolean expression.
    ExpVisitor boolV = new ExpVisitor(decls);
    BoolExp be = wh.getBoolExp();
    be.accept(boolV);
    int sdBool = ((Integer) boolV.getResult()).intValue();

    // Reject this while command if its boolean expression is of
    // type HIGH
    if (sdBool == HIGH) // [While_high]
        checkResult = false;

    // Now we create a new (transformed) while command and its type.
    Command whileNew = new While((BoolExp) be.clone(), pWhileProg);
    Command whileNewType = new While((BoolExp) be.clone(), pWhileType);

    // Regardless of the check result we return the new ElementPair
    return new ElementPair(whileNew, whileNewType, pWhileAl);
}

/**
 * Implements the [Fork] rule.
 * @param fk a <code>Fork</code> command
 * @return the resulting <code>ElementPair</code> from the transformation
 */
public Object visit (Fork fk) { // [Fork]
    // First we get the program and the program vector of the fork-command.
    Program p = fk.getProgram();
    ProgramVector pv = fk.getProgramVector();

    // Now we invoke the transformation of the program as well as
    // of the program vector
    ElementPair pPair = (ElementPair) p.accept(this);
    ElementPair pvPair = null;
    if (pv != null)
        pvPair = (ElementPair) pv.accept(this);

    // We get the transformed program and program vector as well as their
    // types and the flags whether there is an assignment to low...
    Program pProg = (Program) pPair.getElement();
    Program pType = (Program) pPair.getType();
}
C Source Code

```java
boolean pAl = pPair.getAlFlag();
ProgramVector pvProg = null;
ProgramVector pvType = null;
boolean pvAl = false;
if (pvPair != null) {
    pvProg = (ProgramVector) pvPair.getElement();
    pvType = (ProgramVector) pvPair.getType();
    pvAl = pvPair.getAlFlag();
}

// Now we create a new (transformed) fork command and its type.
Command forkNew = new Fork(pProg, pvProg);
Command forkNewType = new Fork(pType, pvType);

// Regardless of the check result we return the new ElementPair
return new ElementPair(forkNew, forkNewType, pAl || pvAl);
}

/**
 * Splits up the check for a Program into the check of its Command and
 * the check for the next Program. Implements the [Seq] rule implicitly.
 * @param p a <code>Program</code>
 * @return the resulting <code>ElementPair</code> from the transformation
 */
public Object visit (Program p) {
    // First we get the command and the next program of p
    Command c = p.getCommand();
    Program pNext = p.getProgram();

    // Now we invoke the transformation of the command as well as
    // of the next program
    ElementPair cPair = (ElementPair) c.accept(this);
    ElementPair pPair = null;
    if (pNext != null)
        pPair = (ElementPair) pNext.accept(this);

    // We get the transformed command and program as well as their types
    // and the flags whether there is an assignment to low...
    Command cComm = (Command) cPair.getElement();
    MwlElement cType = (MwlElement) cPair.getType();
    boolean cAl = cPair.getAlFlag();
    Program pProg = null;
    Program pType = null;
    boolean pAl = false;
    if (pPair != null) {
        pProg = (Program) pPair.getElement();
        pType = (Program) pPair.getType();
        pAl = pPair.getAlFlag();
    }

    // Now we create a new (transformed) program and its type.
    Program progNew = new Program(cComm, pProg);
    // To create its type, which is a program, we need to differentiate on
    // the type of the new command - is it a command or a program itself
    // (-> If_high)?
    Program progNewType = null;
    if (cType instanceof Command)
        progNewType = new Program((Command) cType, pType);
    else if (cType instanceof Program) {
    }
```
Program cTypeProg = (Program) cType;
if (pType != null)
    pType.append(cTypeProg.getProgram());
else
    pType = cTypeProg.getProgram();
    progNewType = new Program(cTypeProg.getCommand(), pType);
} else
    System.err.println("Case that should never happen: The type of " +
    "the transformed command \"cs\" is a \"" +
    "ProgramVector."HF);

// We return the new ElementPair
return new ElementPair(progNew, progNewType, cAl || pAl);
}

/**
 * Splits up the check for a ProgramVector into the check of its
 * Program and the check for the next ProgramVector.
 *
 * @param pv a <code>ProgramVector</code>
 * @return the resulting <code>ElementPair</code> from the
 * transformation
 */
public Object visit (ProgramVector pv) {
    // First we get the program and the next program vector
    Program p = pv.getProgram();
    ProgramVector pvNext = pv.getProgramVector();

    // Now we invoke the transformation of the program as well as
    // of the program vector
    ElementPair pPair = (ElementPair) p.accept(this);
    ElementPair pvPair = null;
    if (pvNext != null)
        pvPair = (ElementPair) pvNext.accept(this);

    // We get the transformed program and program vector as well as their
    // types and the flags whether there is an assignment to low...
    Program pProg = (Program) pPair.getElement();
    Program pType = (Program) pPair.getType();
    boolean pAl = pPair.getAlFlag();
    ProgramVector pvProg = null;
    ProgramVector pvType = null;
    boolean pvAl = false;
    if (pvPair != null) {
        pvProg = (ProgramVector) pvPair.getElement();
        pvType = (ProgramVector) pvPair.getType();
        pvAl = pvPair.getAlFlag();
    }

    // Now we create a new (transformed) program vector and its type.
    ProgramVector pvNew = new ProgramVector(pProg, pvProg);
    ProgramVector pvNewType = new ProgramVector(pType, pvType);

    // We return the new ElementPair
    return new ElementPair(pvNew, pvNewType, pAl || pvAl);
}

/**
 * This method will be called if none of the more specialized visit
 * methods could be called. This should never happen actually.
A method call has no effect on the result. Nothing will be done and the check will executed further.

* @param o an <code>Object</code>
* @return null
*
public Object visit (Object o) {
    System.out.println("TransformVisitor: This method should never be called!");
    return null;
}

package visitor;
import ast.*;

/**
 * Visitor is an interface for executing the check logic on MWL abstract syntax trees. It is part of the 'Visitor' design pattern and defines a method <code>visit</code> for concrete AST classes.
 * The required method <code>getResult</code> returns the result of running this visitor on the AST classes. The result is returned as an Object. All required methods <code>visit</code> return an <code>Object</code>, which may be null if there is no value of interest to be returned.
 *
 * @author Christina Pöpper
 * @version 1.0
 */

public interface Visitor {
    // Constant declarations for a binary security domain
    /** Representing the low security domain. */
    public static final int LOW = 0;
    /** Representing the high security domain. */
    public static final int HIGH = 1;
    
    public Object visit (ArithExp ae);
    public Object visit (ArrDec1 ad);
    public Object visit (ArrField af);
    public Object visit (ArrLength al);
    public Object visit (Assign a);
    public Object visit (CompExp ce);
    public Object visit (False f);
    public Object visit (Fork fk);
    public Object visit (Identifier id);
    public Object visit (If i);
    public Object visit (Int in);
    public Object visit (Program p);
    public Object visit (ProgramVector pv);
    public Object visit (Skip s);
    public Object visit (True t);
    public Object visit (VarDec1 vd);
    public Object visit (While w);
    public Object getResult();
}