Information Flow Analysis for CIL

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

Darmstadt, 15. Oktober 2008

Matthias Perner
Abstract

During program execution information flows exists between resources. This flow can be restricted by flow policies to disallow public readable resources to store data that is confidential.

In an information flow analysis, a program is checked for information flows that violate a given flow policy.

Security type systems are mechanisms to reduce the analysis to a typability problem.

In the theoretical part of this thesis, we develop a security type system for a subset of the Common Intermediate Language (CIL) that handles local information resources and managed pointers to local information resources.

In the implementation part of this thesis, we develop a prototypical checking tool that uses a security type system for a subset of the CIL that handles only local information resources to analyse programs.
## Contents

1 Introduction 5
   1.1 Motivation and Goals ........................................... 5
   1.2 Structure of this work .......................................... 5
   1.3 Conventions ..................................................... 6
      1.3.1 Replacements in Functions ............................ 6
      1.3.2 Stacks and Stack operations ........................ 6

2 Basics Principles 7
   2.1 Flow Policies .................................................. 7
   2.2 Security Conditions ........................................... 8
   2.3 Security Type Systems ........................................ 9
   2.4 Soundness Proof ............................................... 9

3 CIL\textsubscript{int} - Primitives and Local Resources 10
   3.1 Memory Model and Program States ........................... 10
   3.2 Instruction Set and Semantics ............................... 10
   3.3 Non-Interference .............................................. 11
   3.4 Typesystem .................................................... 14
      3.4.1 Control Dependency Regions .......................... 14
      3.4.2 Abstract Transformation ............................... 16
      3.4.3 Proof ................................................... 18

4 Prototypical Checking Tool 19
   4.1 Requirements .................................................. 19
      4.1.1 Functionality ........................................... 19
      4.1.2 Extendability ........................................... 19
   4.2 Work Flow and Program Behaviour ............................ 19
      4.2.1 Setting up the analysis ............................... 20
      4.2.2 Course of actions during analysis ................... 21
      4.2.3 User Interface Behaviour ............................... 22
   4.3 Design .......................................................... 22
      4.3.1 Abstract Syntax Tree and Parser ...................... 22
      4.3.2 Analysis Model ........................................ 24
      4.3.3 Analysis Controller .................................. 27
      4.3.4 Print Controller ...................................... 28
      4.3.5 Console User Interface ............................... 28
      4.3.6 Overview of the Architecture ........................ 28

5 Example Analyses 29
   5.1 Direct Flow .................................................... 29
   5.2 Indirect Flow .................................................. 30
   5.3 Example of a non-interfering program ....................... 31
   5.4 Safe Interfering Programs ................................... 32
   5.5 Example of a non-typable, non-interfering program ....... 33
<table>
<thead>
<tr>
<th>Section</th>
<th>File Name</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>D.1.8</td>
<td>File AST/CIL/UnaryInstruction.cs</td>
<td>98</td>
</tr>
<tr>
<td>D.1.9</td>
<td>File AST/CIL/BinaryInstruction.cs</td>
<td>99</td>
</tr>
<tr>
<td>D.1.10</td>
<td>File AST/CIL/Pop.cs</td>
<td>100</td>
</tr>
<tr>
<td>D.1.11</td>
<td>File AST/CIL/Push.cs</td>
<td>101</td>
</tr>
<tr>
<td>D.1.12</td>
<td>File AST/CIL/Load.cs</td>
<td>102</td>
</tr>
<tr>
<td>D.1.13</td>
<td>File AST/CIL/Store.cs</td>
<td>104</td>
</tr>
<tr>
<td>D.1.14</td>
<td>File AST/CIL/UnconditionalJump.cs</td>
<td>105</td>
</tr>
<tr>
<td>D.1.15</td>
<td>File AST/CIL/ConditionalJump.cs</td>
<td>106</td>
</tr>
<tr>
<td>D.1.16</td>
<td>File AST/CIL/Return.cs</td>
<td>110</td>
</tr>
<tr>
<td>D.1.17</td>
<td>File AST/InstructionParser.cs</td>
<td>110</td>
</tr>
<tr>
<td>D.1.18</td>
<td>File AST/Method.cs</td>
<td>113</td>
</tr>
<tr>
<td>D.1.19</td>
<td>File AST/Type.cs</td>
<td>116</td>
</tr>
<tr>
<td>D.1.20</td>
<td>File AST/Variable.cs</td>
<td>118</td>
</tr>
<tr>
<td>D.2</td>
<td>Package Controller</td>
<td>119</td>
</tr>
<tr>
<td>D.2.1</td>
<td>File Visitor.cs</td>
<td>119</td>
</tr>
<tr>
<td>D.2.2</td>
<td>File Analysis/Analysis.cs</td>
<td>119</td>
</tr>
<tr>
<td>D.2.3</td>
<td>File Analysis/CILIntTransformer.cs</td>
<td>122</td>
</tr>
<tr>
<td>D.2.4</td>
<td>File Analysis/MethodTransformer.cs</td>
<td>126</td>
</tr>
<tr>
<td>D.2.5</td>
<td>File Print/CompletePrint.cs</td>
<td>130</td>
</tr>
<tr>
<td>D.2.6</td>
<td>File Print/PrintInstruction.cs</td>
<td>131</td>
</tr>
<tr>
<td>D.3</td>
<td>Package View</td>
<td>133</td>
</tr>
<tr>
<td>D.3.1</td>
<td>File ConsoleUI/ConsoleUI.cs</td>
<td>133</td>
</tr>
</tbody>
</table>

**E Source Code of Example Applications**

139
1 Introduction

1.1 Motivation and Goals

In our modern, heavily connected world many people and companies rely on software from third parties that may not be trustworthy. In other cases the original source of the software is unknown. This software often processes valuable or confidential information.

Furthermore, the software may communicate with the supplier or even another party for different reasons, e.g. registration, licensing or automated updates. As a result, it could be possible that these valuable and confidential information are leaked to a third party, either accidentally by bugs in the software, or even intentionally as a trojan horse. For these reasons, it is vital to check that the software in use is free of those leaks.

One way to ensure the absence of information leaks is a static information flow analysis. The intention of such an analysis is to look for those leaks in the program without running the software. Leaks can be distinguished in two types.

The first type are direct leaks, e.g. a secret information is assigned directly to a public visible resource.

The second type are indirect leaks, e.g. a branching instruction has different execution paths depending on a secret constraint and assigns different values to a public resource on each path.

It is possible to use a special kind of type system, a so called security type system, to find this kind of leaks. Such a type system is specific to one programming language. Therefore those type systems must be developed for every single language in use.

Furthermore, in many cases the user does not get the sourcecode, but only an executable in a low-level language. Nowadays, the executable is often written in some bytecode for a virtual machine, e.g. the Common Intermediate Language for the Virtual Execution System of the .NET-framework [ECM06]. Therefore it is essential to develop security type systems that can cope with those unstructured bytecode languages.

The goal of this work is to take a step on this road and develop such a type system for a subset of the CIL. Comparison of the bytecode language of the JVM [LY99] with the CIL suggests to use an existing type system for the JVM as base. Thus we use the type system in [BPR07] as fundament of the type system developed in this work. Furthermore, this work contains a proof that the type system really enforces the conditions described. Finally, a prototypical tool to run the analysis is developed to show that it is possible to use the type system in a semi-automated way.

1.2 Structure of this work

This work consists of two major areas. A theoretical area and a practical area. The theoretical area starts with a short introduction and overview over the basic principles of security type systems in section 2.
1.3 Conventions

Section 3 introduces a type system for a sublanguage of CIL, called $CIL_{int}$, which consists of the most basic operations in the CIL, like loading and storing of values to local variables and operations on primitive data types.

In section 4 we design and implement the type system for the $CIL_{int}$ in form of a prototypical non-interference checking tool, to show that the theoretical analysis can be used for practical information flow checks.

In section 5 we analyse some example applications and examine the results of the analyses to evaluate the benefit of those type systems for practical software development and analysis.

In section 6 we introduce a type system for $CIL_{pointer}$. This sublanguage of the CIL is a superset of $CIL_{int}$, which expands the language with basic managed pointer operations, like address loading of local variables and indirect loading and storing of values to local variables. This part of CIL is most interesting, because many other bytecode languages do not support the direct use of pointers.

Finally we finish the work with a conclusion and an outlook to similar and future work in section 7.

1.3 Conventions

During this work a lot of operations and notations are used that can not be assumed to be known by all readers, therefore it is essential to describe and define those notations to prevent misunderstandings.

1.3.1 Replacements in Functions

Let $f : A \rightarrow B$ be a function from set $A$ to set $B$. Let $a \in A$ and $b \in B$.

$f' : f \oplus \{a \mapsto b\}$ describes the function that fulfills $\forall x \in A : (x = a \Rightarrow f'(x) = b) \land (x \neq a \Rightarrow f'(x) = f(x))$.

That means $f'(x) = f(x)$ for all values $x \in A$ but $a$ and $f'(a) = b$.

1.3.2 Stacks and Stack Operations

Let $A$ be a set. A stack of elements of the set $A$ is of the type $A^*$. Let $a \in A$ and $as \in A^*$.

There are two possible operations on a stack:

- **push** means that an element is added to the top of the stack. $a :: as$ denotes the stack that results of a push $a$ operation on the stack $as$.

- **pop** means that an element is removed from the top of the stack. $as$ denotes the stack that results of a pop operation on the stack $a :: as$.

Furthermore let $size : A^* \rightarrow \mathbb{N} \cup \{0\}$ be the function that returns the amount of elements on the stack and let $as[i]$ denote the $i$th element on the stack.
2 Basics Principles

In this section we introduce security type systems with their most basic components. One foundation of security type systems are the flow policies, that describe which information flows are legal and which are illegal. The other basic foundation are the security conditions, which describe what the type system is intended to enforce.

2.1 Flow Policies

A flow policy consists of several distinct sets of information resources, called security levels.

The flow policy itself is a partially ordered set of those security levels. A flow policy describes between which security levels a information flow is allowed. Information may flow from a security level to itself or to higher security levels. From now on let $S$ be the set of all security levels in a given flow policy.

We introduce a shortcut to be able to retrieve the security level an information resource is element of. This shortcut makes the definitions and proofs easier to read.

**Definition 1: Ressource Level**

Let $SL(r) = sl$, where $r$ is an information resource and $sl$ is a security level, be the shortcut for $r \in sl \in S$.

Furthermore, we need to define the legal information flows using the partial order of the flow policy:

**Definition 2: Legal Information Flow**

If $sl_1 \leq sl_2$, then information may flow from $sl_1$ to $sl_2$.

If $sl_1 \geq sl_2$, then information may flow from $sl_2$ to $sl_1$.

To formalize the illegal information flow, we change the definition of $>$ and $\leq$. Instead of describing that a security level is higher, respectively lower than another, we use this operator to describe, that a security level is higer or not comparable, respectively lower or not comparable, than another.

**Definition 3: $>$, $<$**

Let $<$ and $>$ be relations of the type $S \times S$.

If $(sl_1, sl_2) \in <$, then $(sl_2, sl_1) \in >$ and if $(sl_2, sl_1) \in >$, then $(sl_1, sl_2) \in <$.

$\leq \{ s \in S \times S | s \not\geq \}$.

In the sequel $sl_1 > sl_2$ will be used instead of $(sl_1, sl_2) \in >$ and $sl_1 < sl_2$ instead of $(sl_1, sl_2) \in <$.

With this information it is now possible to define illegal information flow on the base of $<$ and $>$. 
2.2 Security Conditions

Definition 4: Illegal Information Flow

If $s_l_1 > s_l_2$, then no information may flow from $s_l_1$ to $s_l_2$.
If $s_l_1 < s_l_2$, then no information may flow from $s_l_2$ to $s_l_1$.

During the analysis it will be necessary to determine the least upper bound of two security levels. The least upper bound can be defined as follows:

Definition 5: ⊔

Let $\sqcup$ be a partial function of the type $S \times S \rightarrow S$.

\[ s_l_1 \sqcup s_l_2 = s_l_3, \quad \text{where } s_l_1, s_l_2, s_l_3 \in S, \quad \text{if } s_l_3 \geq s_l_1 \land s_l_3 \geq s_l_2 \land \forall s_l_4 \in S : s_l_4 \geq s_l_1 \land s_l_4 \geq s_l_2, \quad \text{then } s_l_4 \geq s_l_3. \]

One of the most simple flow policies, which is used through all this work, if not mentioned otherwise, is the policy which consists of the two security levels “low” and “high”, where “high” is a higher security level as “low”, denoted $\text{high} > \text{low}$ and $\text{low} \leq \text{high}$. That means data from information resources of security level “low” may flow to any other information resource, whereas data from information resources of security level “high” may only flow to information resources of security level “high”.

This simple flow policy is adequate to model a great many of security requirements. For example lets have a look at an simplified email client: The user may have stored confidential information, like an adress book. For obvious reasons an email client needs a network interface. It is now essential to guarantee that any confidential information does not leave the email client via the network interface. With the simple flow policy it is possible to say that the network interface has the security level “low”, whereas the confidential information has the security level “high”. Therefore any confidential information may not leave the email client via the network interface, if the model fulfils the flow policies rules.

2.2 Security Conditions

If we want to develop an analysis it is good practice to formalize the properties of the application we want to check. This is done by the security condition.

Furthermore, an adequate formalisation of properties enables us to prove that an analysis enforces the properties we strive to fulfil. Therefore it is quintessential to find a security condition which formalizes the information flow rules we want to enforce.

In this work the security conditions used are called “non-interference” conditions. This term describes the fact that the visible output of a program depends only on the visible input. In other words, a program that is run several times with the same visible input produces the same visible output on every run, independent from any invisible input.

In combination with the flow policies it is now possible to define “visibility”. Visibility depends on the possibilities of an observer and on the information resources in use. Therefore it is possible to define the visibility of a ressource as follows:
2.3 Security Type Systems

The goal in this work is to develop an analysis to find illegal information leaks. Illegal information leaks are assignments of information, that depend on secret information, to public information resources.

A security type system reduces the problem of finding those leaks to a typability problem. In this case, a type is a security level. To look if a program is typable, it is necessary to formalize the conditions on which a program is typable. This definition depends on the system developed and can be found in the section containing the type system.

In this work we will model the state of the runtime to observe the changes made by instructions as an abstract state. Using this abstract state, the rules for typability can be formalized as abstract transfer rules.

A program is typable using such a type system, if for every state of the runtime environment that is reachable by running the program an corresponding abstract state exists and an abstract transfer rule can be applied. Typability for the type systems developed is defined in the sections about the type systems, because the definition obviously depends on the definition of the abstract states and transitions.

2.4 Soundness Proof

The analysis of the program with the type system does only check, if a program is typable with this system. To ensure that a program that is typable does not violate the security conditions formalized, it is necessary to prove that typability implies the security conditions. This proof is called the soundness proof. This important part of the work will only be done for the more complex type system in section 6.
3 \(CIL_{int}\) - Primitives and Local Resources

In this section we introduce a subset of the Common Intermediate Language. It is the most basic subset containing only jumps and local integer variables. After introducing and explaining the language, we formalize the security conditions and develop a security type system for this subset.

The subset of the language is very close to the JVM\(\chi\) from [BR05], therefore we can adapt almost the complete security type system introduced in their work.

3.1 Memory Model and Program States

A \(CIL_{int}\) program consists of an instruction list and local variables and parameters. All instructions in the instruction list must be part of the \(CIL_{int}\) sublanguage.

The local variables and parameters can be combined to a set \(\chi\), because in the security type system local variables and parameters can be treated equally.

Furthermore we assign a number to every instruction in the list, called the program point. The program points can be used to identify instructions in a program. The first instruction in the list will be identified by 0, the second by 1 and so on. That means the set of program points consists of natural numbers including 0, denoted as \(PP = \{0, \ldots, n\}\), where \(n\) is the amount of the instructions minus 1. In addition, let \(P[i]\) denote the instruction at program point \(i\).

The state of the runtime environment can now be described by a triple \(\langle i, \rho, os \rangle\), where \(i \in \mathbb{N} \cup \{0\}\) is the next instruction to be executed, \(\rho \in \chi \rightarrow V\) is a mapping of local variables and parameters to values, where \(V\) is the set of primitive, native data types in the runtime environment, and \(os \in V^*\) is an operand stack, consisting of values from \(V\). The combination of all states builds the set \(State\).

Now it is possible to express the operational small step semantics, which describes the execution of each single instruction, as a relation that describes the state transitions of the runtime environment. The relation has the form \(\sim \subseteq State \times (State + V)\). \(\sim^*\) denotes the transitive closure. Furthermore, we will use \(s_1 \sim^n s_2\) in the sequel, to denote that state \(s_1\) is transformed to state \(s_2\) in \(n\) steps.

Additionally, it is useful to introduce a shortcut function to note that a program evaluates to a specific value. From now on let \(P, \rho \downarrow v\) have the same meaning as \(\langle 0, \rho, \epsilon \rangle \sim^* v\).

3.2 Instruction Set and Semantics

As mentioned further above, \(CIL_{int}\) is the most basic instruction set. It consists of operation stack manipulating instructions, unconditional jumps and conditional jumps.

To reduce the amount of state transitions and at the same time reduce the amount of transfer rules needed for the type system, it is possible to merge some instructions in equivalence classes with respect to their stack transitions.

We use the following equivalence classes:

**unary op** Every instruction that pops one value from the operand stack, calculates a result by applying an unary function \(op\) to the value and pushes the result on the operand stack.
3.3 Non-Interference

3.3 Non-Interference

As mentioned before, this work uses non-interference as security condition and it is vital to formalize the condition adequately. The goal of the analysis is to check if an illicit information flow exists in a program. The “non-interference” condition says: If a program is run several times with the same visible input, the visible output of the program is equal on every run. We want to develop a termination insensitive analysis, that means that the former conditions must only be fulfilled, if the program terminates.

With the definition of visibility in section 2.2 and the informal specification of non-interference, the necessity arises to define the indistinguishability of information, to be able to formalize runs with a combination of visible and invisible information.

It is obvious that the indistinguishability must depend on the maximum security level the observer is able to see, because this level determines which information resources are visible. Furthermore, the condition must depend on the security levels of the information resources, because those security levels determine the least security level an observer must have to be able to see the information resource.

A special information resource, when dealing with operand stack based languages, is the operand stack. It would not be accurate enough for the operand stack to be of
3.3 Non-Interference

### CIL₁₅ - Primitives and Local Resources

<table>
<thead>
<tr>
<th>P[i] = unary op</th>
<th>op ∈ U</th>
<th>op(v) = r</th>
</tr>
</thead>
<tbody>
<tr>
<td>(i, ρ, v::os) ~ (i+1, ρ, r::os)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>P[i] = binary op</th>
<th>op ∈ O</th>
<th>op(v1, v2) = r</th>
</tr>
</thead>
<tbody>
<tr>
<td>(i, v1::v2::os) ~ (i+1, ρ, r::os)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>P[i] = pop</th>
</tr>
</thead>
<tbody>
<tr>
<td>(i, ρ, v::os) ~ (i+1, ρ, os)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>P[i] = push v</th>
</tr>
</thead>
<tbody>
<tr>
<td>(i, ρ, v::os) ~ (i+1, ρ, v::os)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>P[i] = loadlocal x</th>
</tr>
</thead>
<tbody>
<tr>
<td>(i, ρ, v::os) ~ (i+1, ρ, ρ(x)::os)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>P[i] = storelocal x</th>
</tr>
</thead>
<tbody>
<tr>
<td>(i, ρ, v::os) ~ (i+1, ρ, v::os)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>P[i] = jumpWith1Argument t cond1(v) = true</th>
</tr>
</thead>
<tbody>
<tr>
<td>(i, ρ, v::os) ~ (i+1, ρ, os)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>P[i] = jumpWith1Argument t cond1(v) = false</th>
</tr>
</thead>
<tbody>
<tr>
<td>(i, ρ, v::os) ~ (i+1, ρ, os)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>P[i] = jumpWith2Arguments t cond2(v1, v2) = true</th>
</tr>
</thead>
<tbody>
<tr>
<td>(i, v1::v2::os) ~ (i+1, ρ, os)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>P[i] = jumpWith2Arguments t cond2(v1, v2) = false</th>
</tr>
</thead>
<tbody>
<tr>
<td>(i, v1::v2::os) ~ (i+1, ρ, os)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>P[i] = unconditionalJump t</th>
</tr>
</thead>
<tbody>
<tr>
<td>(i, ρ, os) ~ (i+1, ρ, os)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>P[i] = return</th>
</tr>
</thead>
<tbody>
<tr>
<td>(i, ρ, v::os) ~ v</td>
</tr>
</tbody>
</table>

where $U$ denotes the set of unary functions, $O$ denotes the set of binary functions, $cond1$ represents a function $V \to \{true, false\}$, $cond2$ represents a function $V \times V \to \{true, false\}$, $v \in V$, $t \in PP$ and $x \in \chi$.

Table 1: Semantic of CIL₁₅

A specific security level, but it is possible to introduce a so-called stack type $st \in S^*$, which is basically a stack of security levels. With the stack type it is possible to track the security levels of information on the operand stack according to their source.

Now it is possible to define indistinguishability relative to a security level $sl_{obs} \in S$ and two stack types $st, st' \in S^*$ and use this definition to define the non-interference condition for the program. In the following definitions we will write $\sim$ instead of $\sim_{sl_{obs}}$, because there is no risk of confusion.

**Definition 7: Indistinguishability of Values**

Two values are indistinguishable, if they are equal.

\[
\frac{v = v'}{v \sim v'}
\]
Definition 8: Indistinguishability of local Mappings

Two variable mappings \( \rho, \rho' \in \chi \rightarrow V \) are indistinguishable with respect to a security level \( sl_{obs} \in S \), if for every information resource \( x \in \chi \) either \( x \) is mapped to equal values in both mappings or the security level of \( x \) is higher than \( sl_{obs} \).

\[
\forall x \in \chi : \rho(x) = \rho'(x) \lor SL(x) > sl_{obs}
\]

\( \rho \sim \rho' \)

Definition 9: Indistinguishability of Operand Stacks

Two operand stacks are indistinguishable with respect to a security level \( sl_{obs} \in S \) and two security level stacks \( st, st' \in S^* \) with the same size as \( os, os' \in V^* \), if they have equal size and every value on the stack is either equal or the corresponding security level on the security level stack is higher than \( sl_{obs} \). Therefore it can be defined inductively:

\[
\begin{align*}
size(os) &= size(st) = size(st') = size(os') = 0 \\
&= \text{size}(st) = 0 \land os \sim_{st, st'} os' \\
&= \text{size}(os) \land v = v' \land sl_v \leq sl_{obs} \\
&= os \sim_{st, st'} os' \land v' \leq sl_{obs} \\
&= os \sim_{st, st'} os' \land sl_v > sl_{obs} \\
&= os' \sim_{st, st'} os' \land sl_v > sl_{obs}
\end{align*}
\]

Definition 10: Indistinguishability of States

Two runtime states are indistinguishable with respect to \( sl_{obs} \in S \) and \( st, st' \in S^* \), if the variable mappings \( \rho, \rho' \in \chi \rightarrow V \) and operand stacks \( os, os' \in V^* \) are indistinguishable.

\[
\rho \sim_{sl_{obs}} \rho' \land os \sim_{st, st'} os'
\]

\( \langle i, \rho, os \rangle \sim \langle i', \rho', os' \rangle \)

Using these definitions it is now possible to formalize the non-interference condition for a program.

Definition 11: Non-Interfering CIL\(_{int}\) Program

A program \( P \) is non-interfering, if for every two local maps \( \rho \) and \( \rho' \), where \( \rho \sim \rho' \), the result \( v \) of a run of the program is equal.

\[
\forall \rho, \rho' \in \chi \rightarrow V : \rho \sim \rho' \land P, \rho \downarrow v \land P, \rho' \downarrow v' \Rightarrow v = v'
\]

\( P \) is non-interfering
3.4 Typesystem

3.4.1 Control Dependency Regions

One problem when dealing with low-level languages is handling branching instructions. At a branching instruction the control may be passed to different instructions, depending on loaded values. In consequence, those instructions are a possible reason for information leaks, when the condition of such an instruction depends on values of a high security level and in the two branches different assignments to a variable of a low security level are made.

In the following examples let $lv$ be a variable of the security level "low" and $hv$ be a variable of the security level "high". The flow policy in use is the simple high-low policy.

This program is obviously interfering, because looking at the return value is enough to learn something about the high level variable $hv$.

This problem can be addressed with control dependency regions. A control dependency region is an overapproximation of the influence area of a branching instruction.

Let $BP$ be the set of all branching points of a program. For $CIL_{int}$ the set of branching points can be written as $BP = \{i \in PP | (P[i] = \text{jumpWith1Argument}) \lor (P[i] = \text{jumpWith2Argument}) \land t \neq i + 1\}$.

Let $CDR$ be the set of control dependency regions. $CDR$ is modeled in form of two functions:

$$junctio n : BP \rightarrow PP$$

$$region : BP \rightarrow P(PP)$$

The set $region(i)$ models the program points of instructions that may be visited on either branch of the control flow after the branching instruction $P[i]$.

The partial function $junctio n(i)$ models the program point at which both branches of the control flow converge. It is a partial function, because it is possible that the control flow paths do not converge after a branching, as shown in the following simple example:

To model the control dependency regions as an safe overapproximation of the influence areas, it is important to formalize conditions for a safe over-approximation that allows us to guarantee that the secret guard in the condition does not have influence on information resources outside its control dependency region.
Let \( \mapsto \in PP \times PP \cup \{-1\} \) be the successor relation. It is defined by:

1. if \( P[i] = \text{unconditionalBranch} \ t \), then \((i, t) \in \mapsto\)
2. if \( P[i] = \text{jumpWith1Argument} \ t \), then \((i, i+1) \in \mapsto \) and \((i, t) \in \mapsto\)
3. if \( P[i] = \text{jumpWith2Arguments} \ t \), then \((i, i+1) \in \mapsto \) and \((i, t) \in \mapsto\)
4. if \( P[i] = \text{return} \), then \((i, -1) \in \mapsto \), expressing that return has no successor in the program.
5. else \((i, i+1) \in \mapsto\)

**Definition 12: Safe Over Approximation**

A control dependency region must fulfill the following three conditions:

1. for all program points \( i \) and all their successors \( j, k \) that fulfill \( j \neq k \) (that means \( i \) is a branching instruction and therefore \( i \in BP \)), either \( k \in \text{region}(i) \) or \( k = \text{junction}(i) \)

\[
\forall i, j, k \in PP : i \mapsto k \land i \mapsto j \land j \neq k \Rightarrow k \in \text{region}(i) \lor k = \text{junction}(i)
\]

2. for all program points \( i, j, k \), that fulfill \( j \in \text{region}(i) \) and \((j, k) \in \mapsto \), either \( k \in \text{region}(i) \) or \( k = \text{junction}(i) \)

\[
\forall i, j, k \in PP : j \in \text{region}(i) \land j \mapsto k \Rightarrow k \in \text{region}(i) \lor k = \text{junction}(i)
\]

3. for all program points \( i, j \), that fulfill \( j \in \text{region}(i) \) and \( P[j] = \text{return} \), \( \text{junction}(i) \) must be undefined

\[
\forall i, j \in PP : j \in \text{region}(i) \land P[j] = \text{return} \Rightarrow \text{junction}(i) \text{ is undefined}
\]

The first condition ensures that every instruction, that is a direct successor of a branching instruction is either in the region or the junction of the branching instruction and thus is in the control dependency region of the branching instruction.

The second condition ensures that every instruction that is a direct successor of an instruction in the region of a branching instruction is either in the region or is the junction of the branching instruction and thus is in the control dependency region of the branching instruction.

The third condition ensures that in presence of a return in the region of a branching instruction the junction of the branching instruction is undefined. That is important, because if a return exists in the region of a branching instruction the control flow of different branches can not converge after the return.
3.4 Typesystem

3.4.2 Abstract Transformation

The analysis consists of a set of abstract transfer rules that manipulate an abstract state. Thus it can be seen as an abstract interpretation. This means that the analysis is an approximate run of the program. It ignores all information that is not of interest for the analysis. In the security type system the only interesting information are the security levels of information resources. Therefore these are the only information we need to model in the abstract state.

Let $ST$ be the set of stacks of security levels $S^*$ and $SE$ be the set of functions $PP \rightarrow S$. The analysis can now be described as transfer rules for tuples of the type $(st, se) \in ST \times SE$. $st$ is called the security level stack and represents the security levels of values on the operand stack. $se$ is called the security environment and represents the least upper bound of security levels of all regions the instruction is part of and therefore the least upper bound of all guards the instruction is executed under.

Furthermore we need a pointwise extensions of $\sqcup$ to be able to use it on security environments and stack types. This functions can be defined as follows:

**Definition 13:** $lift_{sl}$

Let $lift_{sl}$, where $sl \in S$, be a function of the type $ST \rightarrow ST$. If $st' = lift_{sl}(st)$, then:

$$size(st) = size(st') \land \forall i \in \{n \in \mathbb{N} | n \leq size(st)\} : st'[i] = st[i] \sqcup sl$$

Let $lift_{sl}$, where $sl \in S$, be a function of the type $SE \times region \rightarrow SE$. If $se' = lift_{sl}(se, region(i))$, then

$$\forall j \in PP : (j \in region(i) \Rightarrow se'(j) = se(j) \sqcup sl) \land (j \notin region(i) \Rightarrow se'(j) = se(j))$$

Now it is possible to define the abstract transfer rules that manipulate the abstract state. These rules can be found in table 2.

It is obvious that the store is only allowed, if the value on the stack may flow to the security level of the variable or parameter. Furthermore, the loading rules push values on the stack and push the least upper bound of security levels of all information resources that influence the new value on the stack type. Finally, the rule for program termination with return can only be applied, if the security environment of the instruction is low and the value on top of the stack is low.

After introducing the abstract transfer rules it is now necessary to define the typability of a program. In the introduction we said that a program is typable, if for every state of the runtime environment, that is reachable by running the program, a corresponding abstract state exists and an abstract transfer rule can be applied.
3.4 Typesystem

Table 2: Abstract transferrules for CIL\textsubscript{INT}

It is obvious that this informal definition requires a definition for “corresponding states”. This can be defined as follows:

**Definition 14: Corresponding States**

An abstract state $st_i, se_i \in ST \times PP \rightarrow S$ corresponds to a runtime state $\langle i, \rho_i, os_i \rangle$, if both are either the initial states $\langle 0, \rho, \epsilon \rangle$ and $\epsilon, se$ of a program or if the states result from two corresponding states by applying a semantic rule and an abstract transfer rule.

$$
st_i, se_i = \epsilon, se \quad \langle i, \rho_i, os_i \rangle = \langle 0, \rho_0, \epsilon \rangle$$

$$
\langle i, \rho_i, os_i \rangle \parallel st_i, se_i
$$

$$
st_j, se_j \Rightarrow st_i, se_i \quad \langle j, \rho_j, os_j \rangle \sim \langle i, \rho_i, os_i \rangle \quad \langle j, \rho_j, os_j \rangle \parallel st_j, se_j
$$

$$
\langle i, \rho_i, os_i \rangle \parallel st_i, se_i
$$
Now that corresponding states are defined, it is possible to define the typability using the corresponding states. For the transition of abstract states we write $\rightarrow$ instead of $\Rightarrow$ in this definition to prevent confusion.

**Definition 15: CIL\textsubscript{int} Typability**

A program $P$ is typable, if for the last runtime state before terminating a corresponding abstract state exists and an abstract transfer rule can be applied.

$$\forall v \in V : P, \rho \Downarrow v \Rightarrow \langle 0, \rho, \epsilon \rangle \sim^* \langle i, \rho_i, v :: os_i \rangle \sim v \land st_i, se_i \rightarrow \land \langle i, \rho_i, v :: os_i \rangle \parallel st_i, se_i$$

$p$ is typable

It is now possible to proof that all typable programs fulfill the non-interference condition in the soundness proof.

### 3.4.3 Proof

The soundness proof for this type system can be done as an induction on the length of executions. The basic idea is to proof that a program that is run with indistinguishable input preserves the indistinguishability during the whole length of the execution. That means that invisible instructions do not change visible information resources and visible instructions preserve the indistinguishability of states. The first claim is necessary, because changes in low level information resources in a high level region may leak information about the guard. The second claim is necessary, because if a visible instruction violates this claim, it is possible, that low level information resources get different values assigned and therefore the indistinguishability of the output could be violated.

The proof is left to the interested reader, because later in this work is a proof for the CIL\textsubscript{pointer} type system, which is more complex and contains the proof for the CIL\textsubscript{int} system, because CIL\textsubscript{pointer} is a superset of CIL\textsubscript{int} and the abstract transfer rules of the type system for CIL\textsubscript{pointer} contain the typability conditions of the CIL\textsubscript{int} type system.
4 Prototypical Checking Tool

After developing the theoretical type system for $CIL_{int}$, it is interesting to show a possibility how to implement such a type system to show that the theoretical foundations make it possible to automatically check binaries for illicit information flow. This section describes the decisions made during the development of a prototypical checking tool for $CIL_{int}$. The source code of this tool can be found in appendix D.

4.1 Requirements

A checking tool should fulfill some requirements, which can be differentiated into functionality and extendability.

4.1.1 Functionality

One reason for implementing the tool is to show that it is possible to implement the theoretical analysis in a way that correctness is guaranteed, if the tool accepts the program as valid. Correctness can only be guaranteed, if every unexpected result is interpreted as failure of the analysis. Furthermore, every condition must be checked exactly as the definitions and claims of the theoretical model and analysis demand.

Another reason to implement the prototypical checking tool is to show that it is possible to implement a tool that does those checks semi-automatically, even in the case that the source code is not available for analysis. Therefore the prototypical checking tool shall not operate on a textual representation of programs, but on assembly files instead. In this case, assembly files are portable executables that contain managed CIL code.

4.1.2 Extendability

$CIL_{int}$ is only a small part of the complete CIL, therefore it is important to design the tool with a focus on extendability. If the program is easy to extend, then it will be possible in to use the implementation in further works that analyze greater sublanguages of CIL. Therefore it is necessary to design the parser, the program and memory model and the analysis extendable. It must be easy to add more instructions to the parser. The program and memory model must support adding information that are needed for more complex CIL sublanguages to them easily. The analysis needs to be adaptable to those more complex models.

4.2 Work Flow and Program Behaviour

It is useful to define the program behaviour in a way that the work flow for using the program is comprehensible for the user. This can be done by looking at the setup process for the analysis and the course of actions during the analysis.
4.2 Work Flow and Program Behaviour

4.2.1 Setting up the analysis

1. Defining security levels and a flow policy that is to be used during the next analysis runs.

2. Choosing the methods to check. That includes choosing the assemblies and types that include the methods.

3. Assigning security levels to the information resources, that means variables and parameters.

4. Running an analysis.

From this setup process it is possible to draw several conclusions. The first important conclusion is that it would be useful to use files to save the setup settings of an analysis, because it is a complex process and the user should be spared of doing it multiple times, if a further run of the analysis is necessary.

Furthermore, it is obvious that the flow policies can be used through several different analyses. As a result it is reasonable to save the flow policy in a different file than the settings of an analysis.

The assignment of security levels, however, is not independent of the methods that are the targets of the analysis. Therefore it is useful to store those settings in one common file.

This results in the setup process for an analysis in the program shown in figure 1.

![Figure 1: Setup process of an analysis](image)

One could argue, that the security levels are not independent from the flow policy, too, but that problem can be addressed, by checking if the flow policy in use and the
settings file in use are compatible before running the analysis. By doing so it is possible to handle the settings needed for an analysis and the flow policy that is used in the analysis independent and thus improve the reusability of the settings and flow policies.

### 4.2.2 Course of actions during analysis

1. For every assembly to analyse
   
   (a) Open assembly
   
   (b) For every type and method to check

   i. Analyse the recent method

The most important conclusion to draw from this course of actions is the insight that it is helpful to present the result of an analysis, before running the next analysis and, after all analysis runs are done, presenting a summary of all runs. This results in the sequence of actions during a single method analysis shown in figure 2.

![Figure 2: Running an analysis on a single method](image)
4.3 Design

4.2.3 User Interface Behaviour

When starting the application the main menu shows up. From this menu it is possible to change into the submenus for the flow policy, the analysis settings and assemblies. Furthermore, it is possible to run an analysis from this menu.

In the flow policy menu, it is possible to load a flow policy. If the flow policy file given is not valid, the program requests a new filename. Furthermore it is possible to dump the flow policy to the console.

In the analysis settings menu, it is possible to load settings from a file. Additionally it is possible to dump the settings to the console. It is possible, to initialize an analysis settings file. When selecting this action, the user is asked for a name for the new file. If the file already exists, the program asks for another file name. After a valid file name is given, the user is prompted for assemblies to analyze, until an empty line is entered. The program creates a new file with the given filename, adding all the types and methods, including their variables and parameters to the newly created file.

In the assembly menu, it is possible, to load an assembly and to dump it to the console. If the file is not a valid managed code assembly, the program does not load the file.

When running an analysis, the first action of the program is to check if the flow policy and the analysis settings are compatible. If this check is successful, the analysis is started. After analysing a method the result is printed to the console, if the method is interfering the instruction list is shown to make it easier to find the problematic instructions. After all methods are checked, a summary is presented, that states if the methods are non-interfering and in case a method is not non-interfering the reason is given.

4.3 Design

During the design of the prototypical tool it is important to keep the goals of this tool in the mind. The first goal is to show that the implemented tool is accurate to the definitions of the type system and showing the ability to guarantee the modeled legal information flows are not violated, while only requiring a minimum of knowledge about the program itself.

The second goal is to build a fundament for further checking tools on more complex sublanguages of the CIL. This requires the tool to be easily extendable with more instructions and their typing rules. This way it is easier in the future to implement type systems that use the $CIL_{int}$ type system as base.

4.3.1 Abstract Syntax Tree and Parser

**Abstract Syntax Tree** When designing the abstract syntax tree it is important to think about which information of the program structure are interesting. There are two possibilities to consider:

- For the $CIL_{int}$ analysis, the methods and instructions are the only interesting information. The types or assemblies do not hold any information important for this analysis. In consequence, it would be possible to consider the methods as the top level of the abstract syntax tree.
4.3 Design

- For every analysis that involves objects, fields, method calls or exceptions the information about types and even assemblies are important, thus the top level of the abstract syntax tree must be the assemblies.

One of the requirements of this tool is extendability and thus it is obvious that the second solution should be favoured. Furthermore, it is easy to design the syntax tree in a way that it supports the assemblies and types from the beginning, but it could be hard to implement all the changes necessary to support those, when extending the application. As a result the second path is chosen. However, only the information needed for our analysis will be extracted from the assemblies and types. In particular, that means only the types needed are extracted from the assemblies and only the methods needed are extracted from the types. The main focus is on the instructions, therefore it is reasonable to introduce one abstract class for all instructions and for every type of instruction one class that implements the abstract class. This results in the architecture shown by the class diagram in figure 3.

Figure 3: Overview of the abstract syntax tree
Parser  When designing the parser it is important to think about the requirements the parser must fulfil. Furthermore, it must be considered that the implementation of the parser should not be too complex, because it is not the main goal to implement a feature rich parser, but a functional parser that fulfills the goal to parse the instructions needed for the analysis. There are two possible solutions for the implementation to compare during design:

- A simple solution for a parser could be one class that takes the bytecode array and parses it according to a huge switch table with one case for every single bytecode instruction.

- A more complex solution could be a parser where the instructions can register for some bytecodes. The parser just preparses the instruction to the type that is for this bytecode and the instruction parses the relevant part of the bytecode array to gain all information that it needs.

When looking at those two possibilities it is easy to realise that the second approach is much easier to extend. Extending the parser with a new equivalence class of instructions in the first approach involves changing a huge, possibly confusing, switch table in the parser. Furthermore, it is necessary to write a class that represents the new equivalence class of instruction. After that, the switch table in the parser must be corrected to parse all necessary parts of the bytecode array for every single instruction in the new equivalence class. This leads to confusing parser code.

In the second approach, introducing a new equivalence class of instructions involves only the implementation of a concrete instruction that represents the equivalence class and write a method in this new class that parses the relevant parts of the bytecode array. After doing so, it is only necessary to register the new class.

Even though the second approach is more complex to implement, the advantages outbalance the disadvantages and therefore this is the solution chosen. This results in the architecture shown by the class diagram in figure 4 and the course of actions in figure 5.

For the registration of the implementations of the $CIL_{int}$ instructions at the parser, we use the static method "registerInstructions" of the class "CILIntInstructions". The only use for this class is the registration of the $CIL_{int}$ instruction set at the parser.

Location The combination of the abstract syntax tree and the parser can be seen as the model that describes the program and therefore are contained in the package "Model", in the subpackage "AST".

4.3.2 Analysis Model

For the design of the analysis it is important to look at the requirements for an analysis and the course of actions during the analysis of a method.

To be able to run an analysis, the program requires a flow policy, the knowledge which methods to analyse and in which assemblies and types they are contained. Furthermore, the security levels for each information ressource and the control dependency regions for each branching point of the methods must be available for the program.
4.3 Design

Flow Policy and Security Levels  The flow policy can be seen independent from the other information, therefore it is reasonable to implement the flow policy independent in an independent class. The flow policy consists of security levels. As a result, the flow policy is a class “FlowPolicy” that contains a collection of security levels, which are implemented as a class “SecurityLevel”. Furthermore, it is reasonable to implement the functions on security levels, that are needed for the analysis, in the flow policy, because these operations depend on the given flow policy. Therefore “FlowPolicy” must implement methods to retrieve the security levels information may flow to from a specific security level. Furthermore a method to calculate the least upper bound for two security levels must be implemented.

This results in the design shown by the class diagram in figure 6.

Abstract State  When running the analysis it is important to keep track of the abstract state of the virtual machine during the execution of a method. The formal definition of this state is a tuple of a stack of security levels, the so called stack type, and a security environment. Two different approaches have been considered during the design of the tool:

- The abstract state of the runtime environment is represented by a security level stack and the security environment is stored in the analysis settings.
- The abstract state of the runtime environment is represented by a history of calculated security level stacks and the security environment is stored in the analysis settings.
4.3 Design

While the first approach seems to be easier to implement on the first thought, it includes some implementation problems and requires the recalculation of security level stacks at branching targets. Therefore the second approach was chosen to be implemented. This implementation makes it possible to restore any already calculated state of the security level stack. It is implemented in the class “AbstractState”.

The decision to store the security environment in the analysis settings may seem confusing at first sight, but in fact the security environment can be seen as a mapping from control dependency regions to security levels and the analysis settings keep track of the control dependency regions, as we see later. As a result it is not necessary to introduce redundancy in information by adding an extra security environment to the abstract runtime state.

This results in the design shown by the class diagram in figure 6.

**Analysis Settings and Control Dependency Regions** The information about the security levels of information resources and the control dependency regions are directly dependent to the methods to analyse. Thus it is reasonable to collect those information in a class that describes all those properties and call it analysis settings. However, these setting do not reference the assemblies, types, methods and security levels, but only identifiers that enable the loading of the methods and security levels from the abstract syntax tree and the flow policy.

In the formal definition of the analysis, the control dependency regions are a structure of two functions that assign a range and a single instruction to the branching points. In consequence, it is reasonable to introduce a class for the control dependency regions that resembles this structure. These structures are direct part of the analysis settings and therefore referenced in the class containing the analysis settings.

This results in the design shown by the class diagram in figure 7.

**Location** All the components of the analysis mentioned in the recent paragraphs can be seen as a model for the analysis and are therefore contained in the package “Model” in the subpackage “Analysis”.

---

**Figure 6: Design of Security Level, Flow Policy and Abstract State**
4.3 Design

4.3.3 Analysis Controller

During the design of the analysis two different approaches has been considered. The first approach is to let the instructions take care of the transformation of the abstract state. This approach has been discarded, because the analysis would depend on the implementation of the instructions and if the analysis model changes, every existing instruction implementation has to be changed. That violates the goal to implement an extendable tool. Therefore the second approach has been chosen. This approach involves the use of the so-called visitor pattern. The abstract syntax tree consists of the elements to be visited and the analysis is the visitor.

The visitor has been split in two different visitors. The first visitor, the “InstructionContainerVisitor”, is for the structure of the abstract syntax tree, that means for visiting the assemblies, types and methods.

The second visitor, the “CILInstrInstructionVisitor”, is for each single instruction. In consequence it is necessary to extend this visitor, if a new instruction is introduced. But the implementations of the instructions can remain unchanged.

This design results in the class diagram 8.

Both abstract visitor classes can be found in the package “Controller”, the analysis transformations can be found in the subpackage “Analysis”. This package contains a wrapper class “Analysis” to call, when running an analysis, that handles the calls for the
4.3 Design

This wrapper implements an algorithm, that visits all instructions in order of the control flow graph. At branching points it chooses the direct successor as next instruction to visit and adds all instructions of the region to a worklist, if the branching instruction is visited the first time. If the algorithm reaches a return instruction, it looks for the first instruction in the worklist, according to the control flow graph. The upper bound for the runtime approximation is $O(n^2)$. This upper bound may be reached when a program consists of branching instructions with regions that consist of the whole method. There may be algorithms that handle the analysis with a better complexity, but that is left for future work.

4.3.4 Print Controller

It is possible to use the visitor pattern to print the program to the console. This print controller is implemented in the subpackage “Print”. “CompletePrint” implements the “InstructionContainerVisitor” and “PrintInstruction” implements the “CILintInstructionVisitor”. This way it is possible to use this visitor for printing and parameterizing the print. It is possible to show the exact instruction or only the equivalence class by setting the boolean “equivalenceClass” of the print controller to either true or false.

4.3.5 Console User Interface

The console user interface is a simple design that supports the steps described in section 4.2.3 about the behaviour of the user interface. It is a simple implementation without any special design patterns applied. All menus and actions are implemented in the class “Main” in the subpackage “ConsoleUI” in the package “ProgramView”.

4.3.6 Overview of the Architecture

The whole program is loosely designed after the mvc-pattern, where the model, the controller and the view are independent subsystems. The model can be seen as the data used by the program, the controller can be seen as the manipulations that are made on the data and the view can be seen as the interface to the user.

Therefore the program consists of the three packages “Model”, that contains the information needed for the analysis, the “Controller”, that manipulates the model according to our transfer rules of the type system, and the “View”, that contains the console user interface.

This design simplifies the replacement of single parts of the application, for example it is easy to write a new user interface, that has the same functionality by using the classes from the “Model” and “Controller” to implement the functionality.
5 Example Analyses

Now that the type system for CIL\textsubscript{int} is implemented, we shall analyse some example applications to present how the tool could be used to check methods for illicit information flow and to show how the tool reports different analysis failures. Furthermore, the direct comparison of structures in high level languages and their CIL representation gives an insight on the equivalences of the different errors in high level programs and low level programs.

5.1 Direct Flow

This simple example illustrates the analysis of programs that contain a direct flow. The simplicity of the program makes it easy to recognize the error class that it contains. In a high level language, e.g. C\#, the most common reason for direct information flow is an assignment from a high information resource to a low information resource. The following C\# example illustrates this kind of leak:

```csharp
int exampleDirectFlow(int l1, int h1)
{
    l1 = h1;
    return l1;
}
```

It is obvious that this program violates the non-interference condition, because the program could be run with \(l1 = 0, h1 = 0\) resulting in 0 and with \(l1 = 0, h1 = 1\) resulting in 1. The input is indistinguishable for the observer, but the output is not indistinguishable. Hence, the tool should report that this program is interfering.

Using a C\# compiler this method results in an instruction list like in the following listing:

0: ldarg.2
1: starg.s 1
2: ldarg.1
3: return

Using the tool to analyse this method with the simple high-low policy loaded and arg\_1 being of the security level “high”, while arg\_2 is of the security level “low”, reveals an information leak at instruction 1. This can be concluded from the output:


-1 illegal flow from high to low

Furthermore, this output tells us that in instruction 1 an assignment from a value that depends on a high level information resource is assigned to a low level information resource. The tool detected a direct information flow and reported that the program is not non-interfering.
5.2 Indirect Flow

A little bit more complex are indirect information flows. The most simple case is that a conditional that depends on a high level information resource does result in a branching and in both branches a return exists. The following C# example illustrates this problem:

```csharp
int exampleIndirectFlow(int l1, int h1)
{
    if (l1 == h1)
        return 1;
    else
        return 0;
}
```

This example obviously violates the non-interference condition, because the program could be run with \( l1 = 0, h1 = 0 \) resulting in 1 and with \( l1 = 0, h1 = 1 \) resulting in 0. The input is indistinguishable for the observer, but the output is not indistinguishable. Hence, the tool should report that this program is interfering.

Using a C# compiler the source code is translated into following instruction list:

0: ldarg.1
1: ldarg.2
2: bne.un.s 5
3: ldc.i4.1
4: return
5: ldc.i4.0
6: return

The similarities in the C# and the CIL code are easy to find. In both listings, we find two different returns that depend on a conditional.

Using the tool to analyse this method with the simple high-low policy loaded and arg_1 being of the security level “low”, while arg_2 is of the security level “high”, reveals an information leak at instruction 4. This can be concluded from the output:

Analysing method: exampleIndirectFlow(System.Int32, System.Int32)
->4 return reveals higher level value.

Furthermore, the output tells us that the leak is caused by a return instruction that is part of a region, which is not of the lowest security level in the flow policy. In consequence, we could try to fix this information leak by moving the return statement out of the control dependency region of the conditional. The resulting C# source code would look like this:

```csharp
int exampleIndirectFlow2(int l1, int h1)
{
    if (l1 == h1)
        l1 = 1;
    else
        l1 = 0;
    return l1;
}
```
Using a C# compiler this code results in the following instruction list:

0: ldarg.1
1: ldarg.2
2: bne.un.s 6
3: ldc.i4.1
4: starg.s 1
5: br.int 8
6: ldc.i4.0
7: starg.s 1
8: ldarg.1
9: return

The similarities of the C# source code and the CIL code are still easy to recognize.

Using the tool to analyse this method with the simple high-low policy loaded and arg_1 being of the security level ‘low’, while arg_2 is of the security level “high”, reveals that the information leak at instruction 4 still exists. This can be concluded from the output:

Analysering method: exampleDirectFlow(System.Int32, System.Int32)

-> 4 illegal flow from high to low

However, the reason for the error changed. Instruction 4 is a store instruction. It seems that a value that depends on a high level information resource is assigned to arg_1, which is a low level information resource. The explanation can be found in the security environment of instruction 3. Instruction 3 is in the control dependency region of the conditional instruction 2 which has a high level guard. In consequence the security environment of instruction 3 is high and the value that is loaded on to the stack depends on a high level information resource.

5.3 Example of a non-interfering program

After looking at this examples one could wonder, if there exist non-interfering programs with secret input in real world applications. In fact, the examples that are possible with the limited instruction set of CIL look very constructed. Nonetheless it is possible to illustrate real world applications with some abstractions. For example, look at a program that creates a new user with a given password and adds this new association to the user storage of the system:

```csharp
public int addUserToSystem(int password, int salt)
{
    // storage location, like pwd file
    int pwdentry;
    // generate userid
    int uid = 5;
    // calculate salted hash of password
    int saltedhash = salt * password;
    // associate uid with salted hash in storage
    pwdentry = uid + saltedhash;
    // report the new uid
    return uid;
}
```
It is obvious that a user knows his chosen password and the program must tell him his new generated user id or he is not able to use the new account, but he does not need to know the salt value and the salted hash for the password. In consequence, it would be reasonable to chose the *uid* and *password* of security level “low”.

The information ressource *pwdentry* is just a representation for a method call or a store operation in a safe location, like the pwd file in a system. For this reason, we want it to be not readable for the observer and say that it is of security level “high”. Furthermore, the salt value and the salted hash shall be secret and in consequence they are of security level *high*, too.

The return value of this program is the value of the low variable *uid*. Therefore, the tool should report that this program is non-interfering.

Using a C# compile, the source code results in the CIL instruction list:

0: ldc.i4.5
1: stloc.1
2: ldarg.2
3: ldarg.1
4: mul
5: stloc.2
6: ldloc.1
7: ldloc.2
8: add
9: stloc.0
10: ldloc.1
11: return

Running the analysis with the parameters mentioned, the tool reports


as expected.

### 5.4 Safe Interfering Programs

If the user wants to use his *uid* and password with a simple login program, a common problem arises. The following program shall represent the simple login program:

```csharp
public int login(int uid, int password, int salt)
{
    int pwdentry = 5 + 6 * 3;
    // if the uid is associated with the password, login succeeds
    if (pwdentry == uid + password * salt)
    {
        return 1;
    }
    return 0;
}
```

If we assume the same security level assignments like in the “addUserToSystem” example, this program is obviously interfering, because the return value depends on the two secret ressources *salt* and *storageolocation*. The tool reports:

Analysing method: login(System.Int32, System.Int32, System.Int32) -> 10 return reveals higher level value
It is not possible to construct a login program that is non-interfering with this type system and security level assignments.

In consequence, it is necessary to develop rules that allow the typing of a program, even if some dependencies of the output with high level resources exist, but are assumed to be safe. In fact, those rules exist for type systems for other languages. The technique of using such rules is called declassification. The development of declassification rules is left for future work.

5.5 Example of a non-typable, non-interfering program

After the analysis of typable, non-interfering programs and non-typable, interfering programs, it is important to look at a special case.

Some non-interfering programs are not typable, because the conditions that are enforced by the type system are stronger than the non-interference condition.

This problem basically appears, if a program has either no return value, or, due to the fact that the language only uses local values, if values from high level resources are assigned to a low level resource that is not used for calculating the return value of the program.

In the following example, a user can store data in a system. Depending on the user account, the user may have a special storage location, e.g. an encrypted drive, associated with the account, but the system does not reveal the location chosen for storing the information and always returns 1 instead:

```csharp
public int storeData(int uid, int password, int data, int salt)
{
    int storageLocation = 23;
    // if account is associated with a special storage location
    if (storageLocation == uid + password * salt)
    {
        // store data in special location
        int specialStorage = data;
        return 1;
    }
    // data not stored in special location, store it in common location
    int commonStorage = data;
    return 1;
}
```

A C# compiler would create a CIL instruction list like:

```
0: ldc.i4.s 23
1: stloc.0
2: ldloc.0
3: ldarg.1
4: ldarg.2
5: ldarg.s 3
6: mul
7: add
8: bne.un.s 11
9: ldc.i4.1
10: return
11: ldc.i4.1
12: return
```

It is obvious, that the program is non-interfering, according to our condition, because it always returns 1. Let `uid`, `password` and `data` be low level ressources, because they are
user input, and salt, storagelocation, specialstorage and commonstorage be high level resources, because they may not be revealed to the user. The analysis returns:


The program is not accepted as non-interfering, but it does not violate the non-interference condition. That shows us that our analysis will reject some non-interfering programs, because of their special attributes.

5.6 Conclusions From PNIC and Example Applications

As shown in the "Prototypical Non-Interference Checker", it is possible to implement a security type system for a low level language to analyse it for information leaks. The examples showed, that the tool can be used to detect those information leaks semi-automatically with very little knowledge of the program.

We have seen that the tool can be used to prove that a program is non-interfering. However, the rejection of a program by the tool does not necessarily mean that the program is not non-interfering.

In "Prototypical Non-Interference Checker" the user must parameterize the analysis with the control dependency regions. To reduce the amount of knowledge an user needs to have of a program he wants to analyse, future works could develop and evaluate methods to automatically generate the control dependency regions.

In consequence, the use of security type systems in software development helps to prevent programming errors that lead to information leaks and the use in security analysis of software could reveal intentional leaks that are used to spy on the users confidential data.

As a result, security type systems have great relevance in modern program analysis and the development of those tools should advance.
After adapting the basic analysis of [BR05] for JVM\(_\chi\) to an equivalent subset of CIL, it is now interesting to extend the analysis to support a concept that is not known in the java bytecode. \textit{CILpointer} extends CIL\(_{int}\) to support basic managed pointers. Therefore we introduce instructions for loading addresses of local information resources as well as indirect loading and storing of values using these addresses. In consequence, some new possibilities for security leaks emerge. Furthermore, the concept of managed pointers requires that the soundness proof is shown at the end of this section, because there is no work known with a similar concept in a security type system.

### 6.1 Extension of the Memory Model and Program States

\textit{CILpointer} makes use of the same information resources as CIL\(_{int}\), therefore the changes to the memory model are minimal. To understand the changes that are necessary, it is useful to have a look at some properties of managed pointers first. These properties can be found in [ECM06], but for the analysis the most important properties of managed pointers are the following:

- Managed pointers can be passed as arguments and stored in local variables.
- Managed pointers may not point to another managed pointer.
- Managed pointers are not interchangeable with object references.

From these properties it is possible to draw several conclusions about the use of managed pointers and the handling of the managed pointers in the runtime environment and therefore in the analysis later on.

The first property states that managed pointers can be stored in local variables. Therefore it is necessary that the security type system tracks if a variable is used to store a pointer and all the additional information a pointer may have.

Furthermore, it is necessary to handle pointers that are passed as arguments. However, the type system does not have much information available about those pointees, because these are dynamic values that are not available in a static analysis. But the only information interesting for the analysis is if high level information is written to a low level information resource. Therefore it is possible to handle addresses that are passed as an argument the same way as addresses of local variables. As a result, we extend \(\chi\) to include the local variables and parameters as well as the variables that are referenced by pointers passed as arguments. This is safe, because the pointers used in this subset of the language reference local variables only and can be handled as local variables. However, the arguments need to be handled adequately, if the type system is extended to support message calls.

The second property states that managed pointers may not point to other managed pointers. As a result, managed pointers do only have one indirection level, because it is safe to assume that the pointee is not a pointer. This makes the analysis less complex.

The third property states that pointers are not interchangeable with object references. This has the effect that the concept of managed pointers is independent from objects and
6.2 Instructionset and Semantics

Object references and, in consequence, objects and object references are not needed for the analysis.

In consequence of these properties, it is possible to keep the complete memory model, but the part that says that a program evaluates to a specific value. Due to the changes in $\chi$ and the fact that it may now contain resources that can be read from outside the program, it is not safe to say that a program evaluates to a specific value.

This problem can be addressed by saying that a program evaluates to a local mapping and a value. By doing so, we over approximate the amount of information resources that may be visible from outside the program. In consequence, a complete execution of a program can now be described by $\langle 0, \rho_0, \epsilon \rangle \leadsto^* \rho_r, v$. In the sequel we will use the shortcut $P, \rho_0 \Downarrow \rho_r, v$ to denote the fact that a program that is run with the local mapping $\rho_0$ evaluates to the local mapping $\rho_r$ and the value $v$.

Additionally, our analysis assumes that no changes are done to the information resources in $\chi$ from outside the program. This includes the subset of non-local resources that are referenced by local pointers.

6.2 Instructionset and Semantics

After introducing managed pointers, we introduce instructions that use managed pointers to local variables. The extension of the language to support managed pointers needs to include instructions that load local addresses on the operand stack, instructions that load values indirectly from variables referenced by addresses and store values indirectly in variables referenced by addresses. Therefore we introduce the following equivalence classes of instructions:

- **loadlocaladress** $x$ Every instruction that pushes the address of a local variable or parameter on the operand stack.
- **loadindirect** Every instruction that pops one value from the stack, interprets it as an address and loads the value stored in the variable or parameter referenced by this address on the operand stack.
- **storeindirect** Every instruction that pops two values from the operand stack, interprets one as address and stores the other in the variable or parameter referenced by the address.

We can describe the small step semantic of these new instructions using their resulting transfers of the states in the runtime environment. Furthermore, it is necessary to change the semantic of “return” to describe the new possibility of returning values using pointers. The semantic can be found in Table 3.
6.3 Additional Pointer Information

To be able to analyse programs containing managed pointers it is necessary to track the security levels of the pointer targets, the so called pointees. It is important to know the exact security level of a pointee to be able to decide if a value may be loaded or stored to the information resource it references.

Furthermore, the security system must be able to determine the security level of a referenced resource, even if the pointer is loaded from a local information resource.

Additionally, not every variable may be used as a pointer and therefore it must be possible to distinguish pointer variables from value variables.

For these reasons we introduce a second function, called the pointee level. It is possible to distinguish the two types of variables using the pointee level, by assigning a special value to the pointee level, if a variable may not be used as a pointer, and tracking the security level of pointees, by assigning a security level the variable may point to, if the variable may be used as a pointer.

As a result, it is possible to define the pointee level as follows:

**Definition 16: Pointee Level**

Let $PL: \chi \rightarrow S \cup \{\bot\}$ be a function that describes the security level $s \in S \cup \{\bot\}$ the variable $x \in \chi$ may point to.

If $PL(x) = \bot$ the variable may not be used as a pointer.

6.4 Non-Interference

The pointee levels are additional information that may influence the visibility of information. In consequence, it is necessary to adapt the definitions of indistinguishability and the non-interference condition.

The indistinguishability of values can remain untouched, because the additional information do not influence the equivalence of values that is used in the definition.
The definition of indistinguishable local mappings says, that two variable mappings \( \rho, \rho' \in \chi \rightarrow V \) are indistinguishable with respect to a security level \( sl_{obs} \in S \), if for every information resource \( x \in \chi \) either \( x \) is mapped to equal values in both mappings or the security level of \( x \) is higher than \( sl_{obs} \).

The pointee levels do not influence the security levels of local variables and parameters. In consequence, it is obvious that it is safe to keep the definition of indistinguishable local mappings, too.

The definition of indistinguishable operand stacks is relative to the security level of the observer and the stack type. However, the stack type does not account information of pointee levels. Therefore the stack type needs to be extended to support this information to keep track of the pointee levels of addresses. As a result the stack type is extended to be of the type \( PT = (S \times (S \cup \{\bot\}))^\ast \).

After changing the stack type it is necessary to adapt the definition for operand stack indistinguishability to be consistent with the new stack type.

**Definition 17: Indistinguishability of Operand Stacks**

Two operand stacks are indistinguishable with respect to a security level \( sl_{obs} \in S \) and two security level stacks \( st, st' \in (S \times (S \cup \{\bot\}))^\ast \) with the same size as \( os, os' \in V^\ast \), if they have equal size and every value on the stack is either equal or the corresponding security level on the security level stack is higher than \( sl_{obs} \).

\[
\begin{align*}
\text{size}(os) &= \text{size}(st) = \text{size}(st') = \text{size}(os') = 0 \\
\text{size}(os) &= \text{size}(st) = 0 \land os \sim_{st, st'} os' \land \forall v : os \land \forall v' : os' \\
\text{size}(os) &= \text{size}(st) = 0 \land \forall v : os \land \forall v' : os' \\
\end{align*}
\]

The definition of indistinguishable states can remain as it is, because it is defined inductively on the indistinguishability of local mappings and operand stacks.

The definition for non-interfering programs said that a program \( P \) is non-interfering, if for every two local mappings \( \rho \) and \( \rho' \), where \( \rho \sim \rho' \), the results \( v \) and \( v' \) of the program executions are equal.

Now it is necessary for the non-interference condition to include the local mappings, because non-local resources may be referenced and are therefore assumed to be part of \( \chi \).

In conclusion, the non-interference condition of the program can be defined as follows:
6.5 Typesystem

Now it is possible to adapt the typesystem for CIL\textsubscript{int} to support managed pointers. The concept of control dependency regions can remain unchanged, because pointers do not change the influence branching instructions may have on the visible output in another way than value variables do.

6.5.1 Abstract transformation

The basic principle of the analysis is identical to the analysis for CIL\textsubscript{int}. Managed pointers introduce a level of indirection, therefore that must be addressed in the transfer rules.

The analysis for CIL\textsubscript{int} has been defined as a set of rules that manipulate tuples of the type $ST \times SE$. To enable tracking of pointee level of a value, it is necessary to change the tuples that represent the abstract state to be of the type $PT \times SE$.

Let $(pt, se)$ be a tuple of this type. $pt$ is a stack of tuples $(sl, pl) \in PT$. It represents a combination of the security level and the pointee level of the sources of the operands on the stack. $se$ is called the security environment and represents the least upper bound of security levels of all regions the instruction is part of and therefore an upper bound of all guards the instruction may be executed under.

Furthermore it is necessary to extend the lift operation to support the new pointee level to be able to use the lift operations in the new abstract transfer rules. The definition can be changed as follows:

**Definition 19: Pointee Level \textit{lift}_{sl}**

Let $\textit{lift}_{sl}$, where $sl \in S$, be a function of the type $PT \rightarrow PT$. If $pt' = \textit{lift}_{sl}(pt)$, then

$$\text{size}(pt) = \text{size}(pt') \land \forall i \in \{n \in \mathbb{N} | n \leq \text{size}(pt)\} : (sl_{pt}, pl_{pt}) = pt[i] \land pt'[i] = (sl_{pt} \cup sl, pl_{pt})$$

$$\text{Let lift}_{sl}$, where $sl \in S$, be a function of the type $SE \times \text{region} \rightarrow SE$. If $se' = \textit{lift}_{sl}(se, \text{region}(i))$, then

$$\forall j \in PP : (j \in \text{region}(i) \Rightarrow se'(j) = se(j) \cup sl) \land (j \notin \text{region}(i) \Rightarrow se'(j) = se(j))$$

Now it is possible to define the analysis as a set of rules manipulating those tuples that represent the abstract state of the runtime environment. The transfer rules can be found in table 4.
6.5 Typesystem

Table 4: Abstract TransferRules for CIL\_pointer

<table>
<thead>
<tr>
<th>Rule</th>
<th>Statement</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P[i]=\text{unary op}$</td>
<td>$\triangledown(s_1,p_1):\text{pt,se} \Rightarrow (s_1 \sqcup se(i), \bot):\text{pt,se}$</td>
</tr>
<tr>
<td>$P[i]=\text{binary op}$</td>
<td>$\triangledown(s_1,p_1):(s_2,p_2):\text{pt,se} \Rightarrow (s_1 \sqcup s_2 \sqcup se(i), \bot):\text{pt,se}$</td>
</tr>
<tr>
<td>$P[i]=\text{pop}$</td>
<td>$\triangledown(s_1,p_1):\text{pt,se} \Rightarrow \text{pt,se}$</td>
</tr>
<tr>
<td>$P[i]=\text{push v}$</td>
<td>$\triangledown\text{pt,se} \Rightarrow (se(i), \bot):\text{pt,se}$</td>
</tr>
<tr>
<td>$P[i]=\text{loadlocal x}$</td>
<td>$\triangledown\text{pt,se} \Rightarrow (SL(x) \sqcup se(i), PL(x))::\text{pt,se}$</td>
</tr>
<tr>
<td>$P[i]=\text{storelocal x}$</td>
<td>$s_1 \sqcup se(i) \leq SL(x) \quad p_1 = PL(x)$</td>
</tr>
<tr>
<td>$P[i]=\text{unconditional Jump t}$</td>
<td>$\triangledown\text{pt,se} \Rightarrow \text{pt,se}$</td>
</tr>
<tr>
<td>$P[i]=\text{jumpWith1Argument t}$</td>
<td>$\triangledown(s_1,p_1):\text{pt,se} \Rightarrow \text{if } f_{s_1} { \text{pt}, f_{s_1,se}(\text{region}) }$</td>
</tr>
<tr>
<td>$P[i]=\text{jumpWith2Arguments t}$</td>
<td>$\triangledown(s_1,p_1):(s_2,p_2):\text{pt,se} \Rightarrow \text{if } f_{s_1} { \text{pt}, f_{s_1,se}(\text{region}) }$</td>
</tr>
<tr>
<td>$P[i]=\text{loadlocaladdress x}$</td>
<td>$PL(x) \neq \bot$</td>
</tr>
<tr>
<td>$\triangledown\text{pt,se} \Rightarrow (SL(x) \sqcup se(i), SL(x))::\text{pt,se}$</td>
<td></td>
</tr>
<tr>
<td>$P[i]=\text{loadindirect}$</td>
<td>$p_1 \neq \bot$</td>
</tr>
<tr>
<td>$\triangledown(s_1,p_1):\text{pt,se} \Rightarrow (s_1 \sqcup p_1 \sqcup se(i), \bot):\text{pt,se}$</td>
<td></td>
</tr>
<tr>
<td>$P[i]=\text{storeindirect}$</td>
<td>$p_2 \geq s_1 \sqcup s_2 \sqcup se(i)$</td>
</tr>
<tr>
<td>$\triangledown(s_1,p_1):(s_2,p_2):\text{pt,se} \Rightarrow \text{pt,se}$</td>
<td></td>
</tr>
<tr>
<td>$P[i]=\text{return}$</td>
<td>$s_1 \sqcup se(i) = lsl$</td>
</tr>
<tr>
<td>$\triangledown(s_1,p_1):\text{pt,se} \Rightarrow$</td>
<td></td>
</tr>
</tbody>
</table>

where $lsl \in S$ is the least security level of the flow policy, $v \in V$, $t \in PP$ and $x \in \chi$.

The first interesting change is the new transfer rule for $\text{loadlocaladdress x}$. When executing this instruction an address gets pushed on the stack, therefore this is the new top value $v$ on the stack. The top security level of the stack type has to be equal to the source of $v$. The source is the variable $x$ in combination with the instruction at position $i$, therefore the security level of the value must be $SL(x) \sqcup se(i)$. The address references a resource of the security level $SL(x)$, therefore the pointee level must be $SL(x)$.

The second interesting rule is the rule for $\text{storelocal x}$. When executing this transfer-rule a value gets popped from the stack and stored in a local variable or parameter. The first condition for applying the rule has not changed from the $CIL_{int}$ transferrule, but the second condition has been added. The security level of an address stored in this location
must be equal to the pointee level of \( x \), resulting in the condition \( \text{pl}_1 = \text{PL}(x) \). This way it is possible to store an address in a variable and keep track of the security level of the pointee.

The third interesting rule is for the instruction \textit{loadindirect}. When executing this instruction an address gets popped from the stack and the value stored in the resource referenced by this address gets popped on the stack. First thing to check is that the value on the stack is an address. The pointee level has been introduced to track the security level of the pointee. When loading an address onto the stack with \textit{loadlocaladdress} \( x \), the security level of the variable referenced by the address is pushed as pointee level on the stack type. With these prerequisites it is possible to assume that the top value of the stack has a pointee level that is not \( \bot \), if it is an address. Therefore it is possible to use this to check, if an address is on the operandstack, resulting in the condition \( \text{pl}_1 \neq \bot \). After executing the instruction a new value is on top of the stack. The security level of this new value must be equal to its source. The source of the value can be seen as a combination of the instruction, the address and the source of the address. Hence, the security level of the new value must be a combination of the security environment of the instruction \( \text{se}(i) \), the pointee level of the address \( \text{pl}_1 \) and the security level of the source of the address \( \text{sl}_1 \). As result, the security level of the new value must be \( \text{se}(i) \sqcup \text{ml}_1 \sqcup \text{sl}_1 \). Managed pointers may not point to other managed pointers. As a result it is safe to assume that the new value is not an address and therefore pushing \( \bot \) as pointee level on the stack, signaling that the value is not an address.

The last interesting rule is the rule for \textit{storeindirect}. When executing this instruction an address and a value get popped from the stack and the value is stored in the variable referenced by the address. The first value on the stack is the value to be stored, the second is the address. First of all, the pointee level of the address must be higher or equal to the security level of the source of the value to be stored or an illegal direct flow of information exists. It must be of higher or equal security level than the security environment, too, or an illegal indirect flow of information exists. Furthermore, we must check if the address depends on a high level resource and we must include \( \text{sl}_2 \). This results in the condition \( \text{pl}_2 \geq \text{sl}_1 \sqcup \text{sl}_2 \sqcup \text{se}(i) \).

After introducing the abstract transfer rules, it is obvious that it is necessary to change the definitions for corresponding states and typability, because of the changes in the abstract state.

**Definition 20: Corresponding States With Pointers**

An abstract state \( pt_i, \text{se}_i \in PT \times PP \rightarrow S \) corresponds to a runtime state \( \langle i, \rho_i, os_i \rangle \) if both are either the initial states \( \langle 0, \rho, \epsilon \rangle \) and \( \epsilon, \text{se} \) of a program or if the states result from two corresponding states by applying a semantic rule and an abstract transfer rule.

\[
\begin{align*}
pt_i, \text{se}_i &= \epsilon, \text{se} \quad \langle i, \rho_i, os_i \rangle = \langle 0, \rho_0, \epsilon \rangle \\
&\quad \langle i, \rho_i, os_i \rangle \parallel pt_i, \text{se}_i \\
pt_j, \text{se}_j \Rightarrow pt_i, \text{se}_i \quad \langle j, \rho_j, os_j \rangle \sim \langle i, \rho_i, os_i \rangle \quad \langle j, \rho_j, os_j \rangle \parallel pt_j, \text{se}_j \\
&\quad \langle i, \rho_i, os_i \rangle \parallel pt_i, \text{se}_i
\end{align*}
\]
Now that corresponding states are defined, it is possible to define the typability using the new definition of corresponding states. We use \( \rightarrow \) instead of \( \Rightarrow \) to denote the transfer of the abstract state in the definition to prevent confusion with the implication.

**Definition 21: CILpointer Typability**

A program \( P \) is typable, if for the last runtime state before terminating a corresponding abstract state exists and an abstract transfer rule can be applied and if the stack type at every program point is constant.

\[
\forall \rho_r \in \chi \rightarrow V, r \in V : P, \rho \downarrow \rho_r, r \Rightarrow (0, \rho, \epsilon) \sim^* \langle i, \rho_i, r :: os_i \rangle \sim \rho_r, r \land pt_i, se_i \rightarrow \land \langle i, \rho_i, r :: os_i \rangle \parallel pt_i, se_i
\]

Now that the type system is introduced and typability is adapted to the new transfer rules, it is necessary to prove that this type system enforces the conditions for a non-interfering program to show that the type system fulfills its purpose.

### 6.5.2 Proof

**Theorem: Soundness**

If a program \( P \) is typable, then \( P \) is non-interfering.

According to the definition, a program is typable, if it fulfills the condition:

\[
\forall \rho, \rho' \in \chi \rightarrow V : P, \rho \downarrow \rho', v \Rightarrow (0, \rho, \epsilon) \sim^* \langle i, \rho_i, v :: os_i \rangle \sim >^* \rho', r \land pt_i, se_i \rightarrow \land \langle i, \rho_i, v :: os_i \rangle \parallel pt_i, se_i
\]

The soundness theorem says that, if this condition is fulfilled, then the program is non-interfering and, in consequence, must fulfill:

\[
\forall \rho, \rho' \in \chi \rightarrow V : \rho \sim \rho' \land P, \rho \downarrow \rho_r, r \land P, \rho' \downarrow \rho'_r, r' \Rightarrow \rho_r \sim \rho'_r \land r = r'
\]

If \( \rho \sim \rho' \) or \( P, \rho \downarrow \rho_r, r \) or \( P, \rho' \downarrow \rho'_r, r' \) is not fulfilled, the non-interference condition is fulfilled. Hence, it is only necessary to proof that \( r = r' \) and \( \rho_r \sim \rho'_r \) is fulfilled, in the case that the first three subterms are fulfilled.

Furthermore, we know that \( P, \rho \downarrow \rho_r, v \) is a special case of \( \langle i, \rho_i, v :: os_i \rangle \sim^* \rho_r, v \), where \( i = 0, \rho_i = \rho \) and \( os_i = \epsilon \).

In consequence, it is possible to write the non-interference condition as:

\[
\forall \rho, \rho' \in \chi \rightarrow V : \rho \sim \rho' \land (0, \rho, \epsilon) \sim^* \rho_r, r \land (0, \rho', \epsilon) \sim^* \rho'_r, r' \Rightarrow \rho_r \sim \rho'_r \land r = r'
\]

This proof can be done by induction on the length of execution paths. We will use some lemmas during the proof. Furthermore, we use virtual steps instead of the real steps in the type system. The virtual steps allow us to assume equal lengths of execution paths even if the real length of the execution paths may differ in high level regions.
6.5 Typesystem

Definition 22: Virtual Step

A virtual step is a transition from one runtime state to another runtime state, that results from either a single instruction in a low level region or a complete high level region.

\[ \sim_v = \{ (i, \rho_i, os_i), (j, \rho_j, os_j) \} \leftrightarrow se(i) \leq sl_{obs} \}
\]

\[ \cup \{ (i, \rho_i, os_i), (j, \rho_j, os_j) \} \in \text{State} \times \text{State} \}
\]

Lemma 1: High Level Regions Converge

We can use this definition, because of the lemma “High Level Regions Converge”.

Lemma 2: Virtual Steps Preserve Indistinguishability of States

This lemma can be proved by showing that every single abstract transfer rule of the typesystem enforces that the indistinguishability of states is preserved. Furthermore, we must prove that high level regions preserve the indistinguishability of states.

We want to show only an excerpt of the proof in this section, but the complete proof can be found in appendix C.2.

Proof: unary op

Using the semantic of the instruction, we know that

\[ \rho_i \sim \rho'_i \land \rho_i = \rho_j \land \rho'_j = \rho'_j \land \rho_i \sim (sl_{v.plv})::pt(sl_{v.plv})::pt' os_i' \]

\[ \land os_i = v :: os \land os'_i = v' :: os' \land os_j = f(v) :: os \land os'_j = f(v') :: os' \land os_i = v' :: os' \]

And we can conclude that

\[ \rho_j \sim \rho'_j \land \rho_i \sim (sl_{v.plv})::pt(sl_{v.plv})::pt' os'_i \land os_i = v :: os \land os'_i = v' :: os' \]

\[ \land os_j = f(v) :: os \land os'_j = f(v') :: os' \]

In consequence, we know that
\( \rho_j \sim \rho'_j \land v \vdash os \sim_{\text{pt,v'}} os' \land os_j = f(v) :: os \land os'_j = f(v') :: os' \)

From the definition of indistinguishable operand stacks we know that we have to distinguish two cases. The first \( sl_v \leq sl_{obs} \land sl'_v \leq sl_{obs} \) that describes that the top value of the stack is visible and the second \( sl_v > sl_{obs} \land sl'_v > sl_{obs} \) that describes that the top value of the stack is invisible.

1. case: \( sl_v \leq sl_{obs} \land sl'_v \leq sl_{obs} \)

Using the definition for indistinguishable operand stacks, we know that

\( \rho_j \sim \rho'_j \land v \vdash os \sim_{\text{pt,v'}} os' \land os_j = f(v) :: os \land os'_j = f(v') :: os' \)

And we can conclude that

\( \rho_j \sim \rho'_j \land v \vdash f(v) = f(v') \land os \sim_{\text{pt,v'}} os' \land os_j = f(v) :: os \land os'_j = f(v') :: os' \)

With the definition of indistinguishable operand stacks, we know that

\( \rho_j \sim \rho'_j \land f(v) :: os \sim_{\text{pt,v'}} os' \land \land os_j = f(v) :: os \land os'_j = f(v') :: os' \)

In consequence, we know that

\( \rho_j \sim \rho'_j \land os_j \sim_{\text{pt,v'}} os'_j \)

Using the definition for indistinguishable states, we can conclude that

\( \langle j, \rho_j, os_j \rangle \sim \langle j, \rho'_j, os'_j \rangle \)

and finally we know that

\( \langle i, \rho_i, os_i \rangle \sim_v \langle j, \rho_j, os_j \rangle \land \langle i, \rho'_i, os'_i \rangle \sim_v \langle j, \rho'_j, os'_j \rangle \land \langle i, \rho_i, os_i \rangle \sim \langle j, \rho'_j, os'_j \rangle \)

\( \Rightarrow \langle j, \rho_j, os_j \rangle \sim \langle j, \rho'_j, os'_j \rangle \) is fulfilled.

2. case: \( sl_v > sl_{obs} \land sl'_v > sl_{obs} \)

Using the definition for indistinguishable operand stacks, we know that

\( \rho_j \sim \rho'_j \land sl_v > sl_{obs} \land sl'_v > sl_{obs} \land os \sim_{\text{pt,v'}} os' \land os_j = f(v) :: os \land os'_j = f(v') :: os' \)

Using the abstract transfer rule, we know that

\( \rho_j \sim \rho'_j \land sl_v > sl_{obs} \land sl'_v > sl_{obs} \land os \sim_{\text{pt,v'}} os' \land os_j = f(v) :: os \land os'_j = f(v') :: os' \land sl_f(v) = sl_v \sqcup se(i) \land sl_f(v') = sl'_v \sqcup se(i) \)

We can conclude that

\( \rho_j \sim \rho'_j \land sl_f(v) > sl_{obs} \land sl_f(v') > sl_{obs} \land os \sim_{\text{pt,v'}} os' \land os_j = f(v) :: os \land os'_j = f(v') :: os' \land sl_f(v) > sl_{obs} \land sl_f(v') > sl_{obs} \land os_j = f(v) :: os \land os'_j = f(v') :: os' \)

In consequence, we know that

\( \rho_j \sim \rho'_j \land os_j \sim_{\text{pt,v'}} os'_j \)

Using the definition for indistinguishable states, we can conclude that

\( \langle j, \rho_j, os_j \rangle \sim \langle j, \rho'_j, os'_j \rangle \)

and finally we know that

\( \langle i, \rho_i, os_i \rangle \sim_v \langle j, \rho_j, os_j \rangle \land \langle i, \rho'_i, os'_i \rangle \sim_v \langle j, \rho'_j, os'_j \rangle \land \langle i, \rho_i, os_i \rangle \sim \langle j, \rho'_j, os'_j \rangle \)

\( \Rightarrow \langle j, \rho_j, os_j \rangle \sim \langle j, \rho'_j, os'_j \rangle \) is fulfilled.
Proof: storelocal x

Using the semantic of the instruction, we know that
\[ p_i \sim \rho_i \land p_i \oplus \{x \mapsto v\} = \rho_j \land \rho'_i \oplus \{x \mapsto v'\} = \rho'_j \land \text{os}_i \sim (sl_v, pt_v); (sl'_v, pt'_v); pt' \text{ os'}_i \]
\[ \land \text{os}_i = v :: \text{os} \land \text{os'}_i = v' :: \text{os'} \land \text{os}_j = \text{os} \land \text{os'}_j = \text{os'} \]

And we can conclude that
\[ p_i \sim \rho_i' \land p_i \oplus \{x \mapsto v\} = \rho_j \land \rho'_i \oplus \{x \mapsto v'\} = \rho'_j \]
\[ \land v :: \text{os} \sim (sl_v, pt_v); (sl'_v, pt'_v); pt' \text{ os'} = \text{os} \land \text{os'}_j = \text{os'} \]

We have to distinguish two cases \( sl_v \leq sl'_v \land sl'_v \leq sl_v \) and \( sl_v > sl'_v \land sl'_v > sl_v \)

1. case: \( sl_v \leq sl'_v \land sl'_v \leq sl_v \)

Using the definition for indistinguishable operand stacks, we know that
\[ p_i \sim \rho_i' \land p_i \oplus \{x \mapsto v\} = \rho_j \land \rho'_i \oplus \{x \mapsto v'\} = \rho'_j \land \text{os}_i \sim pt, pt' \text{ os'}_i \]
\[ \land \text{os}_j = \text{os} \land \text{os'}_j = \text{os'} \land v = v' \]

We can conclude that
\[ p_i \sim \rho_i' \land p_i \oplus \{x \mapsto v\} = \rho_j \land \rho'_i \oplus \{x \mapsto v'\} = \rho'_j \land \text{os}_j \sim pt, pt' \text{ os'}_j \]

And we can conclude that
\[ p_j \sim \rho'_j \land \text{os}_j \sim pt, pt' \text{ os'}_j \]

Using the definition for indistinguishable states, we can conclude that
\[ \langle j, p_j, \text{os}_j \rangle \sim \langle j, \rho'_j, \text{os'}_j \rangle \]

and finally we know that
\[ \langle i, \rho_i, \text{os}_i \rangle \sim_v \langle j, p_j, \text{os}_j \rangle \sim_v \langle i, \rho'_i, \text{os'}_i \rangle \]
\[ \Rightarrow \langle j, \rho_j, \text{os}_j \rangle \sim \langle j, \rho'_j, \text{os'}_j \rangle \] is fulfilled.

2. case: \( sl_v > sl'_v \land sl'_v > sl_v \)

Using the abstract transfer rule, we know that
\[ p_i \sim \rho_i' \land p_i \oplus \{x \mapsto v\} = \rho_j \land \rho'_i \oplus \{x \mapsto v'\} = \rho'_j \land v :: \text{os} \sim (sl_v, pt_v); (sl'_v, pt'_v); pt' \text{ v'} :: \text{os'} \land \text{os}_j = \text{os} \land \text{os'}_j = \text{os'} \land SL(x) \supseteq sl_v \cup se(i) > sl'_v \land SL(x) \supseteq sl'_v \cup se(i) > sl_v \]

Using the definition of indistinguishable local mappings, we can conclude that
\[ p_i \oplus \{x \mapsto v\} \sim \rho_i' \oplus \{x \mapsto v'\} \land p_i \oplus \{x \mapsto v\} = \rho_j \land \rho'_i \oplus \{x \mapsto v'\} = \rho'_j \land v :: \text{os} \sim (sl_v, pt_v); (sl'_v, pt'_v); pt' \text{ v'} :: \text{os'} \land \text{os}_j = \text{os} \land \text{os'}_j = \text{os'} \]

We can conclude that
\[ p_j \sim \rho'_j \land v :: \text{os} \sim (sl_v, pt_v); (sl'_v, pt'_v); pt' \text{ v'} :: \text{os'} \land \text{os}_j = \text{os} \land \text{os'}_j = \text{os'} \]

Using the definition for indistinguishable operand stacks, we know that
\[ p_j \sim \rho'_j \land \text{os} \sim pt, pt' \text{ os'} \land \text{os}_j = \text{os} \land \text{os'}_j = \text{os'} \]

And now we can conclude that
\[ p_j \sim \rho'_j \land \text{os}_j \sim pt, pt' \text{ os'}_j \]

Using the definition for indistinguishable states, we can conclude that
\[ \langle j, p_j, \text{os}_j \rangle \sim \langle j, \rho'_j, \text{os'}_j \rangle \]

Finally, we know that
\[ \langle i, \rho_i, \text{os}_i \rangle \sim_v \langle j, p_j, \text{os}_j \rangle \sim_v \langle i, \rho'_i, \text{os'}_i \rangle \sim_v \langle j, \rho'_j, \text{os'}_j \rangle \]
\[ \Rightarrow \langle j, \rho_j, \text{os}_j \rangle \sim \langle j, \rho'_j, \text{os'}_j \rangle \] is fulfilled.
Proof: loadlocaladdress \( x \)

Using the semantic of the instruction, we know that
\[
\rho_i \sim \rho'_i \land \rho_i = \rho_j \land \rho'_j = \rho_j \land \text{o}s_i \sim_{\ast, pt, pt'}\text{o}s'_i \land \text{o}s_j = \ast(x) :: \text{o}s_i \land \text{o}s'_j = \ast(x) :: \text{o}s_i' \text{ is fulfilled.}
\]

And we can conclude that
\[
\rho_j \sim \rho'_j \land \text{o}s_i \sim_{\ast, pt, pt'}\text{o}s'_i \land \text{o}s_j = \ast(x) :: \text{o}s_i \land \text{o}s'_j = \ast(x) :: \text{o}s'_i
\]

Using the definition for indistinguishable operand stacks, we know that
\[
\rho_j \sim \rho'_j \land \ast(x) :: \text{o}s_i \sim_{(\ast(x), \Gamma_1, pt,(\text{pt}, \text{pt}'))} \ast(x) :: \text{o}s'_i \land \text{o}s_j = \ast(x) :: \text{o}s_i \land \text{o}s'_j = \ast(x) :: \text{o}s'_i
\]

And we can conclude that
\[
\rho_j \sim \rho'_j \land \text{o}s_j \sim_{\ast, pt, pt'}\text{o}s'_j
\]

Using the definition for indistinguishable states, we can conclude that
\[
\langle i, \rho_i, \text{o}s_i \rangle \sim \langle j, \rho_j, \text{o}s_j \rangle
\]

and finally we know that
\[
\langle i, \rho_i, \text{o}s_i \rangle \sim_v \langle j, \rho_j, \text{o}s_j \rangle \land \langle i, \rho'_i, \text{o}s'_i \rangle \sim_v \langle j, \rho'_j, \text{o}s'_j \rangle \land \langle i, \rho_i, \text{o}s_i \rangle \sim \langle i, \rho'_i, \text{o}s'_i \rangle
\]

\[\Rightarrow \langle j, \rho_j, \text{o}s_j \rangle \sim \langle j, \rho'_j, \text{o}s'_j \rangle \text{ is fulfilled.}\]

\[\Box\]

Proof: loadindirect

Using the semantic of the instruction, we know that
\[
\rho_i \sim \rho'_i \land \rho_i = \rho_j \land \rho'_j = \rho_j \land \text{o}s_i \sim_{\ast, \text{pt}, \text{pt}'}, \text{o}s'_i \land \text{o}s_j = \ast(x) :: \text{o}s_i \land \text{o}s'_j = \ast(x) :: \text{o}s_i' \text{ is fulfilled.}
\]

And we can conclude that
\[
\rho_j \sim \rho'_j \land \text{o}s_i \sim_{\ast, \text{pt}, \text{pt}'}, \text{o}s'_i \land \text{o}s_j = \ast(x) :: \text{o}s_i \land \text{o}s'_j = \ast(x) :: \text{o}s_i'
\]

We have to distinguish two cases. Either all information is visible \( \text{s}_k(x) \leq \text{s}_k(x) \leq \text{s}_k(x) \) or either the address is invisible or the resource that is referenced by the address \( \text{s}_k(x) > \text{s}_k(x) \) or \( \text{s}_k(x) > \text{s}_k(x) > \text{s}_k(x) \).

1. case: \( \text{s}_k(x) \leq \text{s}_k(x) \leq \text{s}_k(x) \leq \text{s}_k(x) \)

Using the definition of indistinguishable local mappings, we know that
\[
\rho_j \sim \rho'_j \land \text{o}s_i \sim_{\ast, \text{pt}, \text{pt}'}, \text{o}s'_i \land \text{o}s_j = \ast(x) :: \text{o}s_i
\]

And we can conclude that
\[
\rho_j \sim \rho'_j \land \text{o}s_i \sim_{\ast, \text{pt}, \text{pt}'}, \text{o}s'_i \land \text{o}s_j = \ast(x) :: \text{o}s_i \land \text{o}s'_j = \ast(x) :: \text{o}s'_i
\]

Using the definition of indistinguishable operand stacks, we can conclude
\[
\rho_j \sim \rho'_j \land \text{o}s_i \sim_{\ast, \text{pt}, \text{pt}'}, \text{o}s'_i \land \text{o}s_j = \ast(x) :: \text{o}s_i \land \text{o}s'_j = \ast(x) :: \text{o}s'_i
\]

Using the definition for indistinguishable operand stacks, we know that
\[
\rho_j \sim \rho'_j \land \rho_i(x) :: \ast(x) \sim_{\ast, \text{pt}, \text{pt}'}, \text{o}s_i \land \text{o}s_j = \ast(x) :: \text{o}s_i \land \text{o}s'_j = \ast(x) :: \text{o}s'_i
\]

And we can conclude that
\[
\rho_j \sim \rho'_j \land \text{o}s_i \sim_{\ast, \text{pt}, \text{pt}'}, \text{o}s'_i
\]

Using the definition for indistinguishable states, we can conclude that
\[
\langle j, \rho_j, \text{o}s_j \rangle \sim \langle j, \rho'_j, \text{o}s'_j \rangle
\]

and finally we know that
\[
\langle i, \rho_i, \text{o}s_i \rangle \sim_v \langle j, \rho_j, \text{o}s_j \rangle \land \langle i, \rho'_i, \text{o}s'_i \rangle \sim_v \langle j, \rho'_j, \text{o}s'_j \rangle \land \langle i, \rho_i, \text{o}s_i \rangle \sim \langle i, \rho'_i, \text{o}s'_i \rangle
\]

46
6.5 Types

\[ \Rightarrow (j, \rho_j, os_j) \sim (j, \rho'_j, os'_j) \] is fulfilled.

2. case: \( sl_{k(z)} > sl_{obs} \land sl'_{k(z)} > sl_{obs} \lor pl_{k(z)} > sl_{obs} \land pl'_{k(z)} > sl_{obs} \)

Using the definition of indistinguishable operand stacks, we can conclude

\[ \rho_j \sim \rho'_j \land os \sim pt \ruit os' \land os_j = \rho_1(x) :: os \land os'_j = \rho'_1(x) :: os' \]

\[ (sl_{k(z)} > sl_{obs} \land sl'_{k(z)} > sl_{obs} \lor pl_{k(z)} > sl_{obs} \land pl'_{k(z)} > sl_{obs}) \]

We can conclude that

\[ \rho_j \sim \rho'_j \land os \sim pt \ruit os' \land os_j = \rho_1(x) :: os \land os'_j = \rho'_1(x) :: os' \]

\[ (sl_{k(z)} \lor pl_{k(z)} \lor se(i) > sl_{obs} \land sl'_{k(z)} \lor pl'_{k(z)} \lor se'(i) > sl_{obs}) \]

We can conclude that

\[ \rho_j \sim \rho'_j \land os \sim pt \ruit os' \land os_j = \rho_1(x) :: os \land os'_j = \rho'_1(x) :: os' \]

Using the definition for indistinguishable operand stacks, we know that

\[ os_j = \rho_1(x) :: os \land os'_j = \rho'_1(x) :: os' \]

And we can conclude that

\[ \rho_j \sim \rho'_j \land os_j \sim pt \ruit os'_j \]

Using the definition for indistinguishable states, we can conclude that

\[ \left( j, \rho_j, os_j \right) \sim \left( j, \rho'_j, os'_j \right) \]

and finally we know that

\[ \left( i, \rho_i, os_i \right) \sim_v \left( j, \rho_j, os_j \right) \land \left( i, \rho'_i, os'_i \right) \sim_v \left( j, \rho'_j, os'_j \right) \land \left( i, \rho_j, os_j \right) \sim \left( i, \rho'_j, os'_j \right) \]

\[ \Rightarrow \left( j, \rho_j, os_j \right) \sim \left( j, \rho'_j, os'_j \right) \] is fulfilled.

\[ \square \]

Proof: storeindirect

Using the semantic of the instruction, we know that

\[ \rho_i \sim \rho'_i \land os_i \sim (sl_{v,pv}):(sl_{k(z)},pl_{k(z)}):pt:(sl_{v',pv}):(sl_{k(z')},pl_{k(z')}):pt' \] \( os'_i \]

\[ \land \rho_i \uplus \{ x \mapsto v \} = \rho_j \land \rho'_i \uplus \{ x' \mapsto v' \} = \rho'_j \]

\[ \land os_j = v :: \& (x) :: os \land os'_j = v' :: \& (x') :: os' \land os_j = os \land os'_j = os' \] is fulfilled.

And we can conclude that

\[ \rho_i \sim \rho'_i \land \& (x) :: os \sim (sl_{v,pv}):(sl_{k(z)},pl_{k(z)}):pt:(sl_{v',pv}):(sl_{k(z')},pl_{k(z')}):pt' \] \( v' :: \& (x') :: os' \]

\[ \land \rho_i \uplus \{ x \mapsto v \} = \rho_j \land \rho'_i \uplus \{ x' \mapsto v' \} = \rho'_j \land os_j = os \land os'_j = os' \land v = v' \land \& (x) = \& (x') \]

We have to distinguish two cases. Either all information on the stack is from visible ressources \( sl_v \leq sl_{obs} \land sl'_v \leq sl_{obs} \land sl_{k(z)} \leq sl_{obs} \land sl_{k(z')} \leq sl_{obs} \) or one of the information ressources is invisible \( sl_v > sl_{obs} \land sl'_v > sl_{obs} \lor sl_{k(z)} > sl_{obs} \land sl_{k(z')} > sl_{obs} \)

1. case: \( sl_v \leq sl_{obs} \land sl'_v \leq sl_{obs} \land sl_{k(z)} \leq sl_{obs} \land sl_{k(z')} > sl_{obs} \)

Using the definition for indistinguishable operand stacks, we know that

\[ \rho_i \sim \rho'_i \land \& (x) :: os \sim (sl_{v,pv}):(sl_{k(z)},pl_{k(z)}):pt:(sl_{v',pv}):(sl_{k(z')},pl_{k(z')}):pt' \] \( v' :: \& (x') :: os' \]

\[ \land \rho_i \uplus \{ x \mapsto v \} = \rho_j \land \rho'_i \uplus \{ x' \mapsto v' \} = \rho'_j \land os_j = os \land os'_j = os' \land v = v' \land \& (x) = \& (x') \]

And we can conclude that

\[ \rho_i \sim \rho'_i \uplus \{ x \mapsto v \} = \rho_j \land \rho'_i \uplus \{ x' \mapsto v' \} = \rho'_j \land os_j \sim pt \ruit os'_j \land v = v' \land \& (x) = \& (x') \]

Using the definition for indistinguishable local mappings, we know that

\[ \rho_i \uplus \{ x \mapsto v \} \sim \rho'_i \uplus \{ x' \mapsto v' \} \land \rho_i \uplus \{ x \mapsto v \} = \rho_j \land \rho'_i \uplus \{ x' \mapsto v' \} = \rho'_j \land os_j \sim pt \ruit os'_j \land v = v' \land \& (x) = \& (x') \]

And we can conclude that

\[ \rho_j \sim \rho'_j \land os_j \sim pt \ruit os'_j \]

Using the definition for indistinguishable states, we can conclude that
6.5 Typesystem

\[ \langle j, p_j, os_j \rangle \sim \langle j, p'_j, os'_j \rangle \]

and finally we know that

\[ \langle i, p_i, os_i \rangle \sim_v \langle j, p_j, os_j \rangle \land \langle i, p'_i, os'_i \rangle \sim_v \langle j, p'_j, os'_j \rangle \land \langle i, p_i, os_i \rangle \sim \langle i, p'_i, os'_i \rangle \]

\[ \Rightarrow \langle j, p_j, os_j \rangle \sim \langle j, p'_j, os'_j \rangle \]

is fulfilled.

2. case: \( sl_u > sl_{obs} \land sl'_u > sl_{obs} \lor sl_{k(x)} > sl_{obs} \land sl_{k(x')} > sl_{obs} \)

Using the abstract transfer rule, we know that

\[ \rho_i \sim \rho'_i \land v :: \& (x) :: os \sim (sl_v, pl_v) :: (sl_{k(x)} : pl_{k(x)}) :: pt \cdot (sl_v', pl_v') :: pt' \cdot v' :: \& (x') :: os' \]

\[ \land \rho_i \oplus \{ x \mapsto v \} = \rho_j \land \rho'_i \oplus \{ x' \mapsto v' \} = \rho'_j \land os_j = os \land os'_j = os' \]

\[ \land SL(x) = pl_{k(x)} \geq sl_u \sqcup sl_{k(x)} \sqcup se(i) > sl_{obs} \]

\[ \land SL(x) = pl_{k(x')} \geq sl'_u \sqcup sl_{k(x')} \sqcup se'(i) > sl_{obs} \]

Using the definition of indistinguishable local mappings, we can conclude that

\[ \rho_i \sim \rho'_j \land os_j = os \land os'_j = os' \]

\[ \land v :: \& (x) :: os \sim (sl_v, pl_v) :: (sl_{k(x)} : pl_{k(x)}) :: pt \cdot (sl_v', pl_v') :: pt' \cdot v' :: \& (x') :: os' \]

Using the definition for indistinguishable operand stacks, we know that

\[ \rho_j \sim \rho'_j \land os \sim pt \cdot pt' \land os_j = os \land os'_j = os' \]

And now we can conclude that

\[ \rho_j \sim \rho'_j \land os_j \sim pt \cdot pt' \land os'_j \]

Using the definition for indistinguishable states, we can conclude that

\[ \langle j, p_j, os_j \rangle \sim \langle j, p'_j, os'_j \rangle \]

and finally we know that

\[ \langle i, p_i, os_i \rangle \sim_v \langle j, p_j, os_j \rangle \land \langle i, p'_i, os'_i \rangle \sim_v \langle j, p'_j, os'_j \rangle \land \langle i, p_i, os_i \rangle \sim \langle i, p'_i, os'_i \rangle \]

\[ \Rightarrow \langle j, p_j, os_j \rangle \sim \langle j, p'_j, os'_j \rangle \]

is fulfilled.

\[ \square \]

From these proofs we can conclude that these virtual steps preserve indistinguishability. The proofs for the other virtual steps can be found in the appendix C.2

Proof: Soundness

Let \( P \) be a typable program. Let \( \langle i, p_i, os_i \rangle \sim \langle i, p'_i, os'_i \rangle \) be two indistinguishable states and \( \langle i, p_i, os_i \rangle \sim_v \rho_r, v, \langle i, p'_i, os'_i \rangle \sim_v \rho'_r, v' \) be the execution paths.

Now we need to show that \( \rho_r \sim \rho'_r \land r = r' \) is fulfilled.

Induction base:

Let \( n = 1 \) be the amount of virtual steps before termination of the program. As a result, the execution paths must be:

\[ \langle i, p_i, os_i \rangle \sim_v \rho_r, v \land \langle i, p'_i, os'_i \rangle \sim_v \rho'_r, v' \]

We know that “return” is the only instruction that can terminate a program. Using the semantic of “return”, we know \( \langle r, \rho_r, v :: os_r \rangle \sim \rho_r, v \). In consequence, \( \rho_i = \rho_r, \rho'_i = \rho'_r, os_i = v :: os_r \) and \( os'_i = v' :: os'_r \) and the resulting execution paths:

\[ \langle i, p_r, v :: os_r \rangle \sim \rho_r, v \land \langle i, p'_r, v' :: os'_r \rangle \sim \rho'_r, v' \]

Using the definition of indistinguishable states, we can conclude that:

\[ 48 \]
6.5 Typesystem

\[ \rho_r \sim \rho'_r \land v :: os_r \sim_{(sl, pt); pt'} (sl', pt'); pt' v' :: os' \]

From the condition \( sI \sqcup se(i) = lsl \) of the abstract transfer rule for “return”, we know that \( sI \) must be the least security level in the flow policy and, in consequence, any observer may see it, hence \( sI \leq sI_{obs} \). Using the definition of indistinguishable operand stacks, we can conclude:

\[ \rho_r \sim \rho'_r \land os_r \sim_{pt, pt'} os' \land v = v' \]

Finally we know that \( \rho_r \sim \rho'_r \land v = v' \) is fulfilled.

**Induction requirement:**

Any execution with \( n \) states before termination fulfills:

\[ \langle i, \rho_i, os_i \rangle \sim \langle i, \rho'_i, os'_i \rangle \land \langle i, \rho_i, os_i \rangle \sim^n \rho_r, v \land \langle i, \rho'_i, os'_i \rangle \sim^n \rho'_r, v' \Rightarrow \rho_r \sim \rho'_r \land v = v' \]

**Induction step:**

Let the execution path have \( n + 1 \) states. The execution paths can be written as:

\[ \langle i, \rho_i, os_i \rangle \sim \langle j, \rho_j, os_j \rangle \sim^n \rho_r, v \land \langle i, \rho'_i, os'_i \rangle \sim \langle j, \rho'_j, os'_j \rangle \sim^n \rho'_r, v' \]

Using the lemma “Virtual Steps Preserve Indistinguishability of States”, we can conclude that

\[ \langle i, \rho_i, os_i \rangle \sim \langle i, \rho'_i, os'_i \rangle \land \langle i, \rho_i, os_i \rangle \sim \langle j, \rho_j, os_j \rangle \sim^n \rho_r, v \land \langle j, \rho'_j, os'_j \rangle \sim^n \rho'_r, v' \Rightarrow \langle j, \rho_j, os_j \rangle \sim \langle j, \rho'_j, os'_j \rangle \Rightarrow \rho_r \sim \rho'_r \land v = v' \]

is fulfilled. Using the induction requirement, we can conclude that

\[ \langle i, \rho_i, os_i \rangle \sim \langle i, \rho'_i, os'_i \rangle \land \langle i, \rho_i, os_i \rangle \sim \langle j, \rho_j, os_j \rangle \sim^n \rho_r, v \land \langle j, \rho'_j, os'_j \rangle \sim^n \rho'_r, v' \Rightarrow \langle j, \rho_j, os_j \rangle \sim \langle j, \rho'_j, os'_j \rangle \Rightarrow \rho_r \sim \rho'_r \land v = v' \]

is fulfilled. And in consequence

\[ \langle i, \rho_i, os_i \rangle \sim \langle i, \rho'_i, os'_i \rangle \land \langle i, \rho_i, os_i \rangle \sim \langle j, \rho_j, os_j \rangle \sim^n \rho_r, v \land \langle i, \rho'_i, os'_i \rangle \sim \langle j, \rho'_j, os'_j \rangle \sim^n \rho'_r, v' \Rightarrow \langle j, \rho_j, os_j \rangle \sim \langle j, \rho'_j, os'_j \rangle \Rightarrow \rho_r \sim \rho'_r \land v = v' \]

is fulfilled.

As a result, a typable program is non-interfering.

\[ \square \]
7 Conclusion

7.1 Summary
The goal of this paper has been to show a way to analyse CIL programs for illicit information flow. For this reason the security conditions and the principles used have been introduced. Furthermore, the basic principles of security type systems have been shown. After that, a subset of the CIL has been defined to show at a simple example the work that must be done when developing a type system. This type system has been implemented. Some example applications have been tested and the results illustrated the usefulness of security type systems, but, on the other hand, revealed the restrictions those simple systems have and which areas need to get improved. It has been shown that rules for declassification of information and for handling method calls must be developed to improve the relevance for testing and developing of real software systems.

After that, the sublanguage of CIL has been extended to support managed pointers on local resources. The type system has been extended to support the new instructions and the new concepts that were introduced by the managed pointers. Furthermore, it has been shown that the type system does guarantee that a typable program is non-interfering according to our non-interference condition. As a result, it is obvious that code using managed pointers can be checked by a security type system and that further development of security type systems for CIL is reasonable. However, this type system can not be used to support unmanaged pointers, because it is restricted to one level of indirection.

7.2 Related and Future Work
The most important related works are the security type systems developed for the java bytecode in [BR05] and [BPR07]. This work has been the base for the type system of the small subset in our work. In those works they already support method calls, exception handling, objects and arrays. Great parts of their type system should be adaptable for use with the CIL. However, some small differences exist in the object handling between the JVM and CIL. In consequence, it is not possible to use their type system, but it can be used as base.

When extending the type system to support objects, it is necessary to keep the managed pointers in mind, because objects may be referenced by managed pointers, too.

Another extension for the future would be to develop efficient algorithms that can calculate the control dependency regions to minimize the amount of parameterization for the user. The conditions for the safe over approximation can be used to develop such algorithms.

On the practical side, the prototypical checking tool must be extended to support those more complex type systems to show that they can be used for automatic information flow analysis.

All in all, this work is just the first step towards a security type system for the complete CIL, but it is still a lot of work to do to use the type checks for real world applications.
References


A Task

Bachelor Thesis
Information Flow Analysis for CIL
Contact: lux@mais.informatik.tu-darmstadt.de

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 uncritical (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 \( t := h \) or, more subtly, in if \( h = 1 \) then \( t := 1 \) else \( t := 0 \), 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 \( t \). 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:

\[
\begin{align*}
\text{exp : low} & \quad \text{exp : low} \\
\vdash t := \text{exp} & \quad \vdash \text{if exp then } C_1 \text{ else } C_2
\end{align*}
\]

Informally, this can be read: The conditional can only be typed if both the branches can be typed and the guard is of "low" security type.

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 declassification.

In the scenario laid out above, the user of some software usually does not receive the source code of a program, but only the output of some compiler in a low-level language. Today, this is often done in a language for a virtual machine, like the Java bytecode language for the Java Virtual Machine (JVM) [LY99] or the Common Intermediate Language (CIL) for the Virtual Execution System (VES) [ECM06] of the .NET framework. Security Type Systems for these languages have to cope with unstructured code, stack-based computation and language features like objects or exceptions. In [BR05, BPR07] for a Java-like bytecode language a security type system is defined and proved sound according to an information flow condition.

Project Objectives

Core:

- Firstly, you shall develop a prototypical tool to analyze CIL-programs for information flow security according to security type system for a bytecode language without objects and methods [BR05]. This language subset can be considered an abstraction of suitable language subsets from both Java Bytecode and CIL. It includes instructions to load and store values from and to variables, instructions to apply arithmetic operations on values of the evaluation stack, and instructions to jump conditionally or unconditionally.

- Secondly, you shall develop example programs that serve as input for the information flow analysis. You shall use them to assess the type system and your prototypical implementation.

- Thirdly, you shall extend the security type system to include the concept of managed pointers to variables. Managed pointers are a special type of variable which contain the address of another variable. Managed pointers allow indirect access to variables via referencing. This represents a great difference to Java bytecode, where only objects on the heap are accessible through references. This concept could lead to further information leaks and need special checking.

53
Extensions: Additionally to the core objectives, the following objectives could be pursued:

- You could extend the security type system to CIL-like object-oriented concepts, also based on [BR05]. Instructions to create objects, fetch values of fields and put values to fields have to be considered.
- Provide an overview on the differences between Java bytecode and CIL, with focus on differences affecting information flow security and its analysis.
- You could extend your solution to include further concepts you found to be different from Java bytecode.

Main Activities

1. Defining a schedule for the entire project

2. Developing tool to analyze CIL programs includes
   - determining a suitable design and libraries to implement the tool,
   - implementing the tool,
   - and evaluating the choices of design and library based on the experiences made.

3. Developing example programs to assess implementation includes
   - determining programs for all three classes, secure and typable, secure and not
typable (false positives), and insecure and not typable,
   - writing the programs as CIL programs, and
   - testing and evaluating the developed tool using the programs.

4. Type system for the language with pointers includes
   - defining the extended syntax and operational semantics of a suitable CIL-like
language with pointers,
   - defining an adapted semantic security condition,
   - defining an adapted security type systems, and
   - proving soundness of the adapted type system.

Deliverables

The bachelor’s thesis shall include:
- the well commented code of the tool implemented by you
- the well commented code of the example programs implemented by you
- definition and explanation of syntax and operational semantics
- definition and explanation of security conditions
- definition and explanation of security type systems
- soundness proof and explanation for the extended security type system
- detailed explanation of decisions may (description of alternatives, discussion of their
advantages and disadvantages, arguments for the chosen solution, discussion of de-
cision in retrospective)
- detailed elaboration on insights gained, on open problems identified, and on possible
extensions in the future

A talk at the end shall present the main results of the bachelor’s thesis.

Prerequisites

- basic knowledge of low-level languages (assembler, Java bytecode or CIL)
- basic knowledge of program analysis
- programming skills
- Knowledge of the English language is necessary to understand the documentation
the thesis is based on. The thesis itself may be written in German or English.
Benefits
The student can acquire insights in the technology of virtual machines, especially for the .NET-framework. Further the student learns a formal method for automatic and efficient program analysis. The task of the thesis also offers possibilities to approach more fundamental questions.

Supervision
Prof. Dr. Heiko Mantel (mantel@mais.informatik.tu-darmstadt.de)
Alexander Lux (lux@mais.informatik.tu-darmstadt.de)

References
B PNIC - Manual

B.1 Requirements
On windows machines “PNIC” requires the .NET 2.0 or higher framework to be installed. On linux machines “PNIC” requires mono 1.2.2 or higher to be installed.

B.2 Using PNIC
To start “PNIC” in a terminal, change into the directory containing the executable. On a windows machine, just enter “PNIC.exe”. On a linux machine, enter “mono PNIC.exe”. The main menu opens, when starting the program.

B.2.1 Quick Start
1. Start the program
2. Press “R” to run an analysis
3. Enter the file name for the flow policy
4. Enter the file name for the analysis settings
5. Wait for the results of the analysis

B.2.2 Main Menu
In the main menu it is possible to change into the submenus “Flow Policy”, “Analysis Settings” and “Assembly” by pressing the associated keys that are given at the beginning of the line.
Furthermore it is possible to run an analysis by pressing “r” or to exit the program by pressing “x”.

B.2.3 Submenu Flow Policy
In the flow policy menu it is possible to load a flow policy from a file by pressing “L”. When loading a flow policy, “PNIC” prompts for a file name until the name of a valid policy file is given or no file name is entered.
The flow policy can be written to the console by pressing “D”. If no flow policy is loaded, “PNIC” asks for a policy file to load.
Pressing “M” returns to the main menu.

B.2.4 Submenu Analysis Settings
In the analysis settings menu, it is possible to load analysis settings from a file by pressing “L”. When loading analysis settings, “PNIC” prompts for a file name until the name of a valid settings file is given or no file name is entered.
B.3 Creating Flow Policies

The settings can be written to the console by pressing “D”. If no settings are loaded, “PNIC” asks for a settings file to load.

It is possible to initialize a new settings file by pressing “I”. “PNIC” prompts for a file name for the new settings and for a list of assemblies to include. The assembly names are entered one per line. An empty line ends the list. The initialized settings file is a skeleton that includes all types and methods of the included assemblies and their local variables and parameters.

Pressing “M” returns to the main menu.

B.2.5 Submenu Assemblies

In the assemblies menu, it is possible to load assemblies by pressing “L”. When loading assemblies, “PNIC” prompts for an assembly file to load.

It is possible to unload assemblies by pressing “U”. When unloading assemblies, “PNIC” prompts for the assembly to unload.

The loaded assemblies can be written to the console by pressing “D”.

Pressing “M” returns to the main menu.

B.3 Creating Flow Policies

Flow policies are stored in XML files, hence you need a text editor to edit flow policies. In the flow policy file, the legal flows are defined by combinations of source levels and destination levels.

The flow policy is defined between “<flow>” and “</flow>” tags.

The legal flow for a security level is defined between “<source level="id">” and “</source>” tags. The legal destination levels are given as a tag “<destination level="id" />”.

An example flow policy could look like this:

```xml
<?xml version="1.0" encoding="ISO-8859-1" standalone="yes"?>
<flow>
    <source level="low">
        <destination level="high" />
    </source>
    <source level="high">
        <destination level="high" />
    </source>
</flow>
```

B.4 Creating Analysis Settings

Analysis Settings are stored in XML files, hence you need a text editor to edit settings files. The best way to create a new settings file is to initialize the settings file using “PNIC” and configure the settings using the text editor.

An example settings file could look like this:
B.5 CD Content

In the folder /bin on the cd, the “PNIC” executable can be found. Furthermore, an executable containing some example methods, two example settings files and a simple high low policy definition can be found in this folder.
C Proofs of Lemmas

In this section, the missing and incomplete proofs for the lemmas used in the soundness proof can be found.

C.1 High Region Lemmas

C.1.1 High Level Regions Converge

The first lemma we need to prove is the lemma “High Level Regions Converge”. The lemma says that in a typable and terminating program execution, the execution paths that pass a high level region converge at the junction.

∀i, j, k, l ∈ PP : l /∈ region(i) ∧ k ∈ region(i) ∧ j = junction(i)
∧ P, ρ ↓ ρ_r, v ∧ ⟨l, p_l, os_l⟩ ⊳ ⟨k, p_k, os_k⟩
⇒ ⟨l, p_l, os_l⟩ ⊳ ⟨k, p_k, os_k⟩ ⊳∗ ⟨j, p_j, os_j⟩

Proof: High Level Regions Converge

Let i ∈ BP be a branching instruction with a high level guard of the security level sl-guard ∈ S. Let sl-obs ∈ S be the security level of an observer and sl-obs < sl-guard. Let j = junction(i), k ∈ region(i) and l /∈ region(i). Let P, ρ ↓ ρ_r, v be the complete execution path and let ⟨l, p_l, os_l⟩ ⊳ ⟨k, p_k, os_k⟩.

Looking at the semantic for CIL pointer, we see that only “return” can terminate a program and we can conclude that the last step of the complete execution path P, ρ ↓ ρ_r, v must be ⟨r, ρ_r, v :: os_r⟩ ⊳ ρ_r, v.

From the condition sl ∪ se(i) = lsl of the abstract transfer rule for “return” we can conclude that a typeable program P can only terminate in low regions. In consequence, r /∈ region(i).

From requirement 2 of the definition 3.4.1 of “Safe Over Approximation”, we know that ∀m, n, o ∈ PP : n ∈ region(m) ∧ n ⊳ o ⇒ o ∈ region(m) ∨ o = junction(m) is fulfilled.

In consequence, if an execution path leaves a region, it must pass the junction of the region and we know that that a program may not terminate in a high level region, because of the abstract transfer rule for “return”.

That means ∀i, n, j ∈ PP : n ∈ region(i) ∧ j = junction(i) ∧ ⟨n, p_n, os_n⟩ ⊳∗ ρ_r, v ⇒ ⟨n, p_n, os_n⟩ ⊳∗ ⟨j, p_j, os_j⟩ is fulfilled.

Especially, it is fulfilled for every entry point of the region. The entry of a region in an execution path can be written as l /∈ region(i) ∧ k ∈ region(i) ∧ ⟨l, p_l, os_l⟩ ⊳ ⟨k, p_k, os_k⟩. Furthermore, ⟨l, p_l, os_l⟩ must be reachable from the initial state ⟨0, ρ, e⟩. This can be written as ⟨0, ρ, e⟩ ⊳∗ ⟨l, p_l, os_l⟩. The resulting execution path is ⟨0, ρ, e⟩ ⊳∗ ⟨l, p_l, os_l⟩ ⊳ ⟨k, p_k, os_k⟩ ⊳∗ ⟨j, p_j, os_j⟩ ⊳∗ ρ_r, v.

Therefore we can conclude that ∀i, j, k, l ∈ PP : l /∈ region(i) ∧ k ∈ region(i) ∧ j = junction(i) ∧ P, ρ ↓ ρ_r, v ∧ ⟨l, p_l, os_l⟩ ⊳ ⟨k, p_k, os_k⟩ ⇒ ⟨l, p_l, os_l⟩ ⊳ ⟨k, p_k, os_k⟩ ⊳∗ ⟨j, p_j, os_j⟩ is fulfilled.

□
C.1.2 No Visible Changes in High Regions

Lemma 3: No Visible Changes in High Regions

In a typeable program, changes in high regions are invisible.

\[ se(i) > sl_{obs} \land (i, \rho_i, os_i) \leadsto (j, \rho_j, os_j) \Rightarrow (i, \rho_i, os_i) \leadsto (j, \rho_j, os_j) \]

Proof: No Visible Changes in High Regions

We need to show that

\[ se(i) > sl_{obs} \land (i, \rho_i, os_i) \leadsto (j, \rho_j, os_j) \Rightarrow (i, \rho_i, os_i) \leadsto (j, \rho_j, os_j) \]

is fulfilled for every instruction in \( \text{CIL}_{\text{pointer}} \), except “return”, because in a typeable program no “return” can be executed in a high region, due to the requirement \( sl_1 \sqcup sl = sl \) of the abstract transfer rule.

Let \( (i, \rho_i, os_i) \leadsto (j, \rho_j, os_j) \) be the transition of the states, \( se(i) > sl_{obs} \), where \( sl_{obs} \) is the security level of the observer.

From the lift operation we know that for every security level on the security level stack corresponding to \( os_i \), \( sl \geq se(i) \) is fulfilled and we know that \( se(i) > sl_{obs} \), because the instruction is in a high region. That means \( sl > sl_{obs} \) for every \( sl \) on the security level stack corresponding to \( os_i \).

1. case: unary op

Using the semantic of the instruction, we know that \( \rho_i = \rho_j \land os_i = v :: os \land os_j = f(v) :: os \).

Using the definition for indistinguishable local mappings, we can conclude that \( \rho_i \sim rho_j \land os_i = v :: os \land os_j = f(v) :: os \).

We can conclude from the abstract transfer rule that the security level of the new security level \( sl_{f(v)} \) on the security level stack is \( sl_v \sqcup se(i) \). In consequence \( sl_{f(v)} \geq se(i) \) and \( sl_{f(v)} > sl_{obs} \). In consequence, we know that every value on \( os_i \) and on \( os_j \) is invisible.

Using the definition of indistinguishable operand stacks, we can conclude that \( \rho_i \sim rho_j \land os_i \sim pt, pt' os_j \).

Using the definition of indistinguishable states, we can conclude that \( (i, \rho_i, os_i) \sim (j, \rho_j, os_j) \) is fulfilled and finally

\[ se(i) > sl_{obs} \land (i, \rho_i, os_i) \leadsto (j, \rho_j, os_j) \Rightarrow (i, \rho_i, os_i) \leadsto (j, \rho_j, os_j) \] is fulfilled.

2. case: binary op

Using the semantic of the instruction, we know that \( \rho_i = \rho_j \land os_i = v_1 :: v_2 :: os \land os_j = f(v_1, v_2) :: os \).

Using the definition for indistinguishable local mappings, we can conclude that \( \rho_i \sim rho_j \land os_i = v_1 :: v_2 :: os \land os_j = f(v_1, v_2) :: os \).

We can conclude from the abstract transfer rule that the security level of the new security level \( sl_{f(v_1, v_2)} \) on the security level stack is \( sl_{v_1, v_2} \sqcup sl = se(i) \). In consequence \( sl_{f(v_1, v_2)} \geq se(i) \) and \( sl_{f(v_1, v_2)} > sl_{obs} \). In consequence, we know that every value on \( os_i \) and on \( os_j \) is invisible.
Using the definition of indistinguishable operand stacks, we can conclude that \( \rho_i \sim \rho_{j} \land \text{os}_i \sim_{\text{pt}, \text{pt}'} \text{os}_j \).

Using the definition of indistinguishable states, we can conclude that \( \langle i, \rho_i, \text{os}_i \rangle \sim \langle j, \rho_j, \text{os}_j \rangle \) is fulfilled and finally

\[
\text{se}(i) > \text{sl}_{\text{obs}} \land \langle i, \rho_i, \text{os}_i \rangle \sim \langle j, \rho_j, \text{os}_j \rangle \Rightarrow \langle i, \rho_i, \text{os}_i \rangle \sim \langle j, \rho_j, \text{os}_j \rangle \text{ is fulfilled.}
\]

3. \text{case: pop}

Using the semantic of the instruction, we know that \( \rho_i = \rho_j \land \text{os}_i = \text{os} \land \text{os}_j = \text{os} \).

Using the definition for indistinguishable local mappings, we can conclude that \( \rho_i \sim \rho_{j} \land \text{os}_i = \text{os} \land \text{os}_j = \text{os} \).

Furthermore, we know that every value on \text{os}_i and on \text{os}_j is invisible.

Using the definition of indistinguishable operand stacks, we can conclude that \( \rho_i \sim \rho_{j} \land \text{os}_i = \text{os} \land \text{os}_j = \text{os} \).

Using the definition of indistinguishable states, we can conclude that \( \langle i, \rho_i, \text{os}_i \rangle \sim \langle j, \rho_j, \text{os}_j \rangle \) is fulfilled and finally

\[
\text{se}(i) > \text{sl}_{\text{obs}} \land \langle i, \rho_i, \text{os}_i \rangle \sim \langle j, \rho_j, \text{os}_j \rangle \Rightarrow \langle i, \rho_i, \text{os}_i \rangle \sim \langle j, \rho_j, \text{os}_j \rangle \text{ is fulfilled.}
\]

4. \text{case: push} \ v

Using the semantic of the instruction, we know that \( \rho_i = \rho_j \land \text{os}_i = \text{os} \land \text{os}_j = \text{os} \).

Using the definition for indistinguishable local mappings, we can conclude that \( \rho_i \sim \rho_{j} \land \text{os}_i = \text{os} \land \text{os}_j = \text{os} \).

As a result, we know that every value on \text{os}_i and on \text{os}_j is invisible.

Using the definition of indistinguishable operand stacks, we can conclude that \( \rho_i \sim \rho_{j} \land \text{os}_i = \text{os} \land \text{os}_j = \text{os} \).

Using the definition of indistinguishable states, we can conclude that \( \langle i, \rho_i, \text{os}_i \rangle \sim \langle j, \rho_j, \text{os}_j \rangle \) is fulfilled and finally

\[
\text{se}(i) > \text{sl}_{\text{obs}} \land \langle i, \rho_i, \text{os}_i \rangle \sim \langle j, \rho_j, \text{os}_j \rangle \Rightarrow \langle i, \rho_i, \text{os}_i \rangle \sim \langle j, \rho_j, \text{os}_j \rangle \text{ is fulfilled.}
\]

5. \text{case: loadlocal} \ x

Using the semantic of the instruction, we know that \( \rho_i = \rho_j \land \text{os}_i = \text{os} \land \text{os}_j = \text{os} \).

Using the definition for indistinguishable local mappings, we can conclude that \( \rho_i \sim \rho_{j} \land \text{os}_i = \text{os} \land \text{os}_j = \text{os} \).

We can conclude from the abstract transfer rule that the security level of the new security level \text{sl}_v on the security level stack is equal to \text{se}(i). In consequence \text{sl}_v > \text{sl}_{\text{obs}}.

Using the definition of indistinguishable operand stacks, we can conclude that \( \rho_i \sim \rho_{j} \land \text{os}_i = \text{os} \land \text{os}_j = \text{os} \).

Using the definition of indistinguishable states, we can conclude that \( \langle i, \rho_i, \text{os}_i \rangle \sim \langle j, \rho_j, \text{os}_j \rangle \) is fulfilled and finally

\[
\text{se}(i) > \text{sl}_{\text{obs}} \land \langle i, \rho_i, \text{os}_i \rangle \sim \langle j, \rho_j, \text{os}_j \rangle \Rightarrow \langle i, \rho_i, \text{os}_i \rangle \sim \langle j, \rho_j, \text{os}_j \rangle \text{ is fulfilled.}
\]

6. \text{case: storelocal} \ x

Using the semantic of the instruction, we know that \( \rho_i \oplus \{ x \mapsto v \} = \rho_j \land \text{os}_i = v :: \text{os} \land \text{os}_j = \text{os} \).
From the abstract transfer rule we know that $SL(x) \geq se(i) \sqcup sl_v$ and we know that $se(i) > sl_{obs}$. In consequence, $SL(x) > sl_{obs}$.

Using the definition for indistinguishable local mappings, we can conclude that $\rho_i \sim \rho_{o_j} \land os_i = v :: os \land os_j = os$.

Furthermore, we know that every value on $os_i$ and on $os_j$ is invisible.

Using the definition of indistinguishable operand stacks, we can conclude that $\rho_i \sim \rho_{o_j} \land os_i \sim_{pt,pt'} os_j$.

Using the definition of indistinguishable states, we can conclude that $\langle i, \rho_i, os_i \rangle \sim \langle j, \rho_j, os_j \rangle$ is fulfilled and finally $se(i) > sl_{obs} \land \langle i, \rho_i, os_i \rangle \sim \langle j, \rho_j, os_j \rangle \Rightarrow \langle i, \rho_i, os_i \rangle \sim \langle j, \rho_j, os_j \rangle$ is fulfilled.

7. case: unconditional jump

Using the semantic of the instruction, we know that $\rho_i = \rho_j \land os_i = v :: os_j$.

Using the definition for indistinguishable local mappings, we can conclude that $\rho_i \sim \rho_{o_j} \land os_i = v :: os_j$.

Using the definition of indistinguishable operand stacks, we can conclude that $\rho_i \sim \rho_{o_j} \land os_i \sim_{pt,pt'} os_j$.

Using the definition of indistinguishable states, we can conclude that $\langle i, \rho_i, os_i \rangle \sim \langle j, \rho_j, os_j \rangle$ is fulfilled and finally $se(i) > sl_{obs} \land \langle i, \rho_i, os_i \rangle \sim \langle j, \rho_j, os_j \rangle \Rightarrow \langle i, \rho_i, os_i \rangle \sim \langle j, \rho_j, os_j \rangle$ is fulfilled.

8. case: jumpWith1Argument t

Using the semantic of the instruction, we know that $\rho_i = \rho_j \land os_i = v :: os_j$.

Using the definition for indistinguishable local mappings, we can conclude that $\rho_i \sim \rho_{o_j} \land os_i = v :: os_j$.

Furthermore, we know that every value on $os_i$ and on $os_j$ is invisible.

Using the definition of indistinguishable operand stacks, we can conclude that $\rho_i \sim \rho_{o_j} \land os_i \sim_{pt,pt'} os_j$.

Using the definition of indistinguishable states, we can conclude that $\langle i, \rho_i, os_i \rangle \sim \langle j, \rho_j, os_j \rangle$ is fulfilled and finally $se(i) > sl_{obs} \land \langle i, \rho_i, os_i \rangle \sim \langle j, \rho_j, os_j \rangle \Rightarrow \langle i, \rho_i, os_i \rangle \sim \langle j, \rho_j, os_j \rangle$ is fulfilled.

9. case: jumpWith2Arguments t

Using the semantic of the instruction, we know that $\rho_i = \rho_j \land os_i = v_1 :: v_2 :: os_j$.

Using the definition for indistinguishable local mappings, we can conclude that $\rho_i \sim \rho_{o_j} \land os_i = v_1 :: v_2 :: os_j$.

Furthermore, we know that every value on $os_i$ and on $os_j$ is invisible.

Using the definition of indistinguishable operand stacks, we can conclude that $\rho_i \sim \rho_{o_j} \land os_i \sim_{pt,pt'} os_j$.

Using the definition of indistinguishable states, we can conclude that $\langle i, \rho_i, os_i \rangle \sim \langle j, \rho_j, os_j \rangle$ is fulfilled and finally $se(i) > sl_{obs} \land \langle i, \rho_i, os_i \rangle \sim \langle j, \rho_j, os_j \rangle \Rightarrow \langle i, \rho_i, os_i \rangle \sim \langle j, \rho_j, os_j \rangle$ is fulfilled.

10. case: loadlocaladdress x

Using the semantic of the instruction, we know that $\rho_i = \rho_j \land os_i = os \land os_j = \& (x) :: os$.  

62
Using the definition for indistinguishable local mappings, we can conclude that $\rho_i \sim \rho_{ho_j} \land os_i = os \land os_j = \& (x) :: os$.

We can conclude from the abstract transfer rule that the security level of the new security level $sl_{k(x)}$ on the security level stack is equal to $se(i)$. In consequence $sl_{k(x)} > sl_{obs}$. As a result, we know that every value on $os_i$ and on $os_j$ is invisible.

Using the definition of indistinguishable operand stacks, we can conclude that $\rho_i \sim \rho_{ho_j} \land os_i \sim_{pt, pt'} os_j$.

Using the definition of indistinguishable states, we can conclude that $\langle i, \rho_i, os_i \rangle \sim \langle j, \rho_j, os_j \rangle$ is fulfilled and finally $se(i) > sl_{obs} \land \langle i, \rho_i, os_i \rangle \sim \langle j, \rho_j, os_j \rangle \Rightarrow \langle i, \rho_i, os_i \rangle \sim \langle j, \rho_j, os_j \rangle$ is fulfilled.

11. case: loadindirect

Using the semantic of the instruction, we know that $\rho_i = \rho_j \land os_i = \& (x) :: os \land os_j = v :: os$.

Using the definition for indistinguishable local mappings, we can conclude that $\rho_i \sim \rho_{ho_j} \land os_i = \& (x) :: os \land os_j = v :: os$.

We can conclude from the abstract transfer rule that the security level of the new security level $sl_v$ on the security level stack is equal to $sl_{k(x)} \cup pl_{k(x)} \cup se(i)$. In consequence $sl_v \leq sl_{obs}$. As a result, we know that every value on $os_i$ and on $os_j$ is invisible.

Using the definition of indistinguishable operand stacks, we can conclude that $\rho_i \sim \rho_{ho_j} \land os_i \sim_{pt, pt'} os_j$.

Using the definition of indistinguishable states, we can conclude that $\langle i, \rho_i, os_i \rangle \sim \langle j, \rho_j, os_j \rangle$ is fulfilled and finally $se(i) > sl_{obs} \land \langle i, \rho_i, os_i \rangle \sim \langle j, \rho_j, os_j \rangle \Rightarrow \langle i, \rho_i, os_i \rangle \sim \langle j, \rho_j, os_j \rangle$ is fulfilled.

12. case: storeindirect

Using the semantic of the instruction, we know that $\rho_i \oplus \{x \mapsto v\} = \rho_j \land os_i = v :: \& (x) :: os \land os_j = os$.

From the abstract transfer rule we know that $pl_{k(x)} \geq se(i) \cup sl_v$. That means $SL(x) \geq se(i) \cup sl_v$. We know that $se(i) > sl_{obs}$. In consequence, $SL(x) > sl_{obs}$.

Using the definition for indistinguishable local mappings, we can conclude that $\rho_i \sim \rho_{ho_j} \land os_i = v :: \& (x) :: os \land os_j = os$.

Furthermore, we know that every value on $os_i$ and on $os_j$ is invisible.

Using the definition of indistinguishable operand stacks, we can conclude that $\rho_i \sim \rho_{ho_j} \land os_i \sim_{pt, pt'} os_j$.

Using the definition of indistinguishable states, we can conclude that $\langle i, \rho_i, os_i \rangle \sim \langle j, \rho_j, os_j \rangle$ is fulfilled and finally $se(i) > sl_{obs} \land \langle i, \rho_i, os_i \rangle \sim \langle j, \rho_j, os_j \rangle \Rightarrow \langle i, \rho_i, os_i \rangle \sim \langle j, \rho_j, os_j \rangle$ is fulfilled.

From those 12 cases we can conclude that $se(i) > sl_{obs} \land \langle i, \rho_i, os_i \rangle \sim \langle j, \rho_j, os_j \rangle \Rightarrow \langle i, \rho_i, os_i \rangle \sim \langle j, \rho_j, os_j \rangle$ is fulfilled for a step in a high region.
C.1.3 High Regions Preserve Indistinguishability

**Lemma 4: High Regions Preserve the Indistinguishability of States**

In a typeable program, high regions preserve the indistinguishability of states.

\[ \text{Let } i \in \text{region}(m) \land j = \text{junction}(m) \land i' \in \text{region}(m) \land (i, \rho_i, o_{si}) \sim_v (i', \rho'_i, o_{si}') \land \langle i, \rho_i, o_{si} \rangle \sim (j, \rho_j, o_{sj}) \Rightarrow (j, \rho'_j, o_{sj}') \]

**Proof: High Regions Preserve Indistinguishability**

Let \( \langle i, \rho_i, o_{si} \rangle \sim_v (j, \rho_j, o_{sj}) \), \( \langle i, \rho'_i, o_{si}' \rangle \sim_v (j, \rho'_j, o_{sj}') \) and \( \langle i, \rho_i, o_{si} \rangle \sim \langle i, \rho'_i, o_{si}' \rangle \). Furthermore let \( j = \text{junction}(m) \).

We must prove, that

\[ \text{Let } i \in \text{region}(m) \land j = \text{junction}(m) \land i' \in \text{region}(m) \land (i, \rho_i, o_{si}) \sim_v \langle i', \rho'_i, o_{si}' \rangle \land \langle i, \rho_i, o_{si} \rangle \sim \langle j, \rho_j, o_{sj} \rangle \land \langle i', \rho'_i, o_{si}' \rangle \sim \langle j', \rho'_j, o_{sj}' \rangle \]

be the transitions.

Using the lemma “No Visible Changes in High Regions”, we know that

\[ \text{Let } \langle i, \rho_i, o_{si} \rangle \sim_v \langle j, \rho_j, o_{sj} \rangle \land \langle i, \rho'_i, o_{si}' \rangle \sim_v \langle j, \rho'_j, o_{sj}' \rangle \land \langle i', \rho'_i, o_{si}' \rangle \sim \langle j', \rho'_j, o_{sj}' \rangle \]

is fulfilled.

Using the transitivity of indistinguishability, we can conclude that

\[ \text{Let } \langle i, \rho_i, o_{si} \rangle \sim_v \langle i', \rho'_i, o_{si}' \rangle \land \langle i', \rho'_i, o_{si}' \rangle \sim \langle j, \rho_j, o_{sj} \rangle \land \langle j, \rho_j, o_{sj} \rangle \sim \langle j', \rho'_j, o_{sj}' \rangle \]

is fulfilled and finally we know that

\[ \text{Let } \langle j, \rho_j, o_{sj} \rangle \land \langle i, \rho'_i, o_{si}' \rangle \sim_v \langle j, \rho'_j, o_{sj}' \rangle \land \langle i, \rho'_i, o_{si}' \rangle \sim \langle j, \rho_j, o_{sj} \rangle \Rightarrow (j, \rho_j, o_{sj}) \sim (j', \rho'_j, o_{sj}') \]

holds for any two execution lengths \( n \) and \( o \).

**Step:**

Let \( n + 1 \) and \( o \) be the length of two execution paths in the high region.

The transitions are

\[ i \in \text{region}(m) \land k \in \text{region}(m) \land j = \text{junction}(m) \land i' \in \text{region}(m) \land (i, \rho_i, o_{si}) \sim \langle i', \rho'_i, o_{si}' \rangle \land \langle i, \rho_i, o_{si} \rangle \sim \langle k, \rho_k, o_{sk} \rangle \sim (j, \rho_j, o_{sj}) \land \langle i', \rho'_i, o_{si}' \rangle \sim \langle j', \rho'_j, o_{sj}' \rangle \]

From the induction requirement we can conclude that

\[ i \in \text{region}(m) \land k \in \text{region}(m) \land j = \text{junction}(m) \land i' \in \text{region}(m) \land (i, \rho_i, o_{si}) \sim \langle i', \rho'_i, o_{si}' \rangle \land \langle k, \rho_k, o_{sk} \rangle \sim (j, \rho_j, o_{sj}) \land \langle i', \rho'_i, o_{si}' \rangle \sim (j', \rho'_j, o_{sj}') \Rightarrow \]

64
Using the lemma “No Visible Changes in High Regions”, we know that \( \langle k, \rho_k, os_k \rangle \sim \langle j, \rho_j, os_j \rangle \) and can conclude that
\[
i \in \text{region}(m) \land k \in \text{region}(m) \land j = \text{junction}(m) \land i' \in \text{region}(m) \land \langle i, \rho_i, os_i \rangle \sim \langle i', \rho_{i}', os_{i}' \rangle \sim \langle j, \rho_j, os_j \rangle \land \langle i', \rho_{i}', os_{i}' \rangle \sim \langle j', \rho_{j}', os_{j}' \rangle \Rightarrow \langle k, \rho_k, os_k \rangle \sim \langle j', \rho_{j}', os_{j}' \rangle \land \langle k, \rho_k, os_k \rangle \sim \langle j, \rho_j, os_j \rangle \text{ is fulfilled.}
\]

Using the transitivity of indistinguishable states, we can conclude that
\[
i \in \text{region}(m) \land k \in \text{region}(m) \land j = \text{junction}(m) \land i' \in \text{region}(m) \land j' = \text{junction}(m) \land \langle i, \rho_i, os_i \rangle \sim \langle i', \rho_{i}', os_{i}' \rangle \sim \langle j, \rho_j, os_j \rangle \land \langle i', \rho_{i}', os_{i}' \rangle \sim \langle j', \rho_{j}', os_{j}' \rangle \Rightarrow \langle j, \rho_j, os_j \rangle \sim \langle j', \rho_{j}', os_{j}' \rangle \text{ is fulfilled.}
\]
### C.2 Virtual Steps

#### C.2.1 Virtual Steps Preserve Indistinguishability

The lemma “Virtual Steps Preserve Indistinguishability of States” says that in a typeable program P, the execution of a single instruction preserves the indistinguishability of states:

\[
\langle i, \rho_i, os_i \rangle \sim_v \langle j, \rho_j, os_j \rangle \wedge \langle i, \rho'_i, os'_i \rangle \sim_v \langle j, \rho'_j, os'_j \rangle \Rightarrow \langle i, \rho_i, os_i \rangle \sim \langle j, \rho'_j, os'_j \rangle
\]

This lemma can be proved by showing that every single abstract transfer rule of the type system enforces that the indistinguishability of the states is preserved. Furthermore, we must prove that high level regions preserve the indistinguishability of states.

**Proof: Virtual Steps Preserve Indistinguishability**

We must show that

\[
\langle i, \rho_i, os_i \rangle \sim_v \langle j, \rho_j, os_j \rangle \wedge \langle i, \rho'_i, os'_i \rangle \sim_v \langle j, \rho'_j, os'_j \rangle \Rightarrow \langle i, \rho_i, os_i \rangle \sim \langle j, \rho'_j, os'_j \rangle
\]

is fulfilled for every typeable instruction in a low level region and for complete high regions.

Let \(\langle i, \rho_i, os_i \rangle \sim_v \langle j, \rho_j, os_j \rangle\), \(\langle i, \rho'_i, os'_i \rangle \sim_v \langle j, \rho'_j, os'_j \rangle\) and \(\langle i, \rho_i, os_i \rangle \sim \langle i, \rho'_i, os'_i \rangle\).

Using the definition of indistinguishable states, we know that \(\rho_i \sim \rho'_i\) and \(os_i \sim_{pt, pt'} os'_i\).

**Proof: unary op**

Using the semantic of the instruction, we know that

\[
\rho_i \sim \rho'_i \land \rho_i = \rho_j \land \rho'_i = \rho'_j \land os_i \sim_{(sl_v, pl_v)} pt_{(sl'_v, pl'_v)} pt' os'_i \land os_i = v :: os \land os'_i = v' :: os' \text{ is fulfilled.}
\]

And we can conclude that

\[
\rho_j \sim \rho'_j \land v :: os \sim_{(sl_v, pl_v)} pt_{(sl'_v, pl'_v)} pt' v' :: os' \land os_j = f(v) :: os \land os'_j = f(v') :: os'
\]

In consequence, we know that

\[
\rho_j \sim \rho'_j \land v :: os \sim_{(sl_v, pl_v)} pt_{(sl'_v, pl'_v)} pt' v' :: os' \land os_j = f(v) :: os \land os'_j = f(v') :: os'
\]

From the definition of indistinguishable operand stacks we know that we have to distinguish two cases. The first \(sl_v \leq sl_{obs} \land sl'_v \leq sl_{obs}\) that describes that the top value of the stack is visible and the second \(sl_v > sl_{obs} \land sl'_v > sl_{obs}\) that describes that the top value of the stack is invisible.

1. **case:** \(sl_v \leq sl_{obs} \land sl'_v \leq sl_{obs}\)

Using the definition for indistinguishable operand stacks, we know that

\[
\rho_j \sim \rho'_j \land v = v' \land os \sim_{pt, pt'} os' \land os_j = f(v) :: os \land os'_j = f(v') :: os'
\]

And we can conclude that

\[
\rho_j \sim \rho'_j \land v = v' \land f(v) = f(v') \land os \sim_{pt, pt'} os' \land os_j = f(v) :: os \land os'_j = f(v') :: os'
\]

With the definition of indistinguishable operand stacks, we know that

\[
\rho_j \sim \rho'_j \land f(v) :: os \sim_{(sl_f(v), \perp)} pt_{(sl'_f(v'), \perp)} pt' f(v') :: os'
\]

And \(os_j = f(v) :: os \land os'_j = f(v') :: os'\)

In consequence, we know that
C.2 Virtual Steps

Using the definition for indistinguishable states, we can conclude that
\[ \langle j, \rho_j, o_s \rangle \sim \langle j, \rho'_j, o'_s \rangle \]
and finally we know that
\[ \langle i, \rho_i, o_s \rangle \sim_v (j, \rho_j, o_s) \land \langle i, \rho'_i, o'_s \rangle \sim_v (j, \rho'_j, o'_s) \]
\[ \Rightarrow \langle j, \rho_j, o_s \rangle \sim \langle j, \rho'_j, o'_s \rangle \]
is fulfilled.

2. case: \( sl_v > sl_{obs} \land sl'_v > sl_{obs} \)

Using the definition for indistinguishable operand stacks, we know that
\[ \rho_j \sim \rho'_j \land \rho_v > sl_{obs} \land \rho'_v > sl_{obs} \land \rho = f(v) \equiv o_s \land \rho'_v = f(v') \equiv o'_s \]
Using the abstract transfer rule, we know that
\[ \rho_j \sim \rho'_j \land \rho_v > sl_{obs} \land \rho'_v > sl_{obs} \land \rho = f(v) \equiv o_s \land \rho'_v = f(v') \equiv o'_s \]
\[ \land sl_j > sl_{obs} \land sl_j' > sl_{obs} \]
Using the definition for indistinguishable operand stacks, we can conclude that
\[ \rho_j \sim \rho'_j \land f(v) \equiv o_s \land f(v') \equiv o'_s \]
In consequence, we know that
\[ \rho_j \sim \rho'_j \land o_s \equiv o'_s \]
Using the definition for indistinguishable states, we can conclude that
\[ \langle j, \rho_j, o_s \rangle \sim \langle j, \rho'_j, o'_s \rangle \]
and finally we know that
\[ \langle i, \rho_i, o_s \rangle \sim_v (j, \rho_j, o_s) \land \langle i, \rho'_i, o'_s \rangle \sim_v (j, \rho'_j, o'_s) \]
\[ \Rightarrow \langle j, \rho_j, o_s \rangle \sim \langle j, \rho'_j, o'_s \rangle \]
is fulfilled.

\[ \square \]

**Proof: binary op**

Using the semantic of the instruction, we know that
\[ \rho_i \sim \rho'_i \land \rho_i = \rho'_i \land \rho_i = \rho'_i \land o_s \equiv (sl_{v1} \oplus : pt, sl_{v1'}, \oplus : pt, : pe') o'_s \land o_s = v_1 \equiv o_v \equiv v'_1 \land v_2 \equiv o'_s \land o_s = f(v_1, v_2) \equiv o_s \land o'_s = f(v'_1, v'_2) \equiv o'_s \]
is fulfilled.

And we can conclude that
\[ \rho_j \sim \rho'_j \land o_s \equiv (sl_{v1} \oplus : pt, sl_{v1'}, \oplus : pt, : pe') o'_s \land o_s = v_1 \equiv o_v \equiv v'_1 \land v_2 \equiv o'_s \land o_s = f(v_1, v_2) \equiv o_s \land o'_s = f(v'_1, v'_2) \equiv o'_s \]
In consequence,
\[ \rho_j \sim \rho'_j \land o_v \equiv : pe \equiv (sl_{v1} \oplus : pt, sl_{v1'}, \oplus : pt, : pe') o'_s \land o_s = f(v_1, v_2) \equiv o_s \land o'_s = f(v'_1, v'_2) \equiv o'_s \]
From the definition of indistinguishable operand stacks, we have to distinguish two cases. The first \( sl_{v1} \leq sl_{obs} \land sl_{v1'} \leq sl_{obs} \land sl_{v2} \leq sl_{obs} \land sl_{v2'} \leq sl_{obs} \) that describes that the top values of each stack are visible and the second \( sl_{v1} > sl_{obs} \land sl_{v1'} > sl_{obs} \land sl_{v2} > sl_{obs} \land sl_{v2'} > sl_{obs} \) that describes that one of the two top values of each stack is invisible.

1. case: \( sl_{v1} \leq sl_{obs} \land sl_{v1'} \leq sl_{obs} \land sl_{v2} \leq sl_{obs} \land sl_{v2'} \leq sl_{obs} \)

67
Using the definition for indistinguishable operand stacks, we know that
\[ \rho_j \sim \rho'_j \land v_1 = v'_2 \land v_2 = v'_2 \land os \sim_{pt,pt'} os' \land os_j = f(v_1, v_2) \land os \land os_j' = f(v'_1, v'_2) \land os' \]
And we can conclude that
\[ \rho_j \sim \rho'_j \land f(v_1, v_2) = f(v'_1, v'_2) \land os \sim_{pt,pt'} os' \land os_j = f(v_1, v_2) \land os \land os_j' = f(v'_1, v'_2) \land os' \]
With the definition of indistinguishable operand stacks, we know that
\[ \rho_j \sim \rho'_j \land f(v_1, v_2) \land os \sim_{(sl_f(v_1, v_2) \land pt), (sl_f(v'_1, v'_2) \land pt')} f(v'_1, v'_2) \land os' \land os_j = f(v_1, v_2) \land os \land os_j' = f(v'_1, v'_2) \land os' \]
In consequence, we know that
\[ \rho_j \sim \rho'_j \land os_j \sim_{(sl_f(v_1, v_2) \land pt), (sl_f(v'_1, v'_2) \land pt')} os_j' \]
Using the definition for indistinguishable states, we can conclude that
\[ \langle j, \rho_j, os_j \rangle \sim \langle j, \rho'_j, os'_j \rangle \]
and finally we know that
\[ \langle i, \rho_i, os_i \rangle \sim_v \langle j, \rho_j, os_j \rangle \land \langle i, \rho'_i, os'_i \rangle \sim_v \langle j, \rho'_j, os'_j \rangle \land \langle i, \rho_i, os_i \rangle \sim \langle i, \rho'_i, os'_i \rangle \Rightarrow \langle j, \rho_j, os_j \rangle \sim \langle j, \rho'_j, os'_j \rangle \] is fulfilled.

2. case: \[ sl_{v_1} > sl_{obs} \land sl_{v'_1} > sl_{obs} \lor sl_{v_2} > sl_{obs} \land sl_{v'_2} > sl_{obs} \]
Using the definition for indistinguishable operand stacks, we know that
\[ \rho_j \sim \rho'_j \land (sl_{v_1} \land sl_{v'_1} > sl_{obs} \lor sl_{v_2} > sl_{obs}) \land sl_{v'_2} > sl_{obs}) \sim_{pt,pt'} os' \land os_j = f(v_1, v_2) \land os \land os_j' = f(v'_1, v'_2) \land os' \]
Using the abstract transfer rule, we know that
\[ \rho_j \sim \rho'_j \land (sl_{v_1} \land sl_{v'_1} > sl_{obs} \lor sl_{v_2} > sl_{obs}) \land sl_{v'_2} > sl_{obs}) \sim_{pt,pt'} os' \land os_j = f(v_1, v_2) \land os \land os_j' = f(v'_1, v'_2) \land os' \land sl_f(v_1, v_2) = sl_{v_1} \land sl_{v_2} \land se(i) \land sl_f(v'_1, v'_2) = sl_{v'_1} \land sl_{v'_2} \land se(i) \]
We can conclude that
\[ \rho_j \sim \rho'_j \land (sl_{v_1} \land sl_{v'_1} > sl_{obs} \lor sl_{v_2} > sl_{obs}) \land sl_f(v'_1, v'_2) > sl_{obs} \land sl_f(v'_1, v'_2) > sl_{obs} \]
Using the definition for indistinguishable operand stacks, we can conclude that
\[ \rho_j \sim \rho'_j \land f(v_1, v_2) \land os \sim_{(sl_f(v_1, v_2) \land pt), (sl_f(v'_1, v'_2) \land pt')} f(v'_1, v'_2) \land os' \land os_j = f(v_1, v_2) \land os \land os_j' = f(v'_1, v'_2) \land os' \]
In consequence, we know that
\[ \rho_j \sim \rho'_j \land os_j \sim_{(sl_f(v_1, v_2) \land pt), (sl_f(v'_1, v'_2) \land pt')} os_j' \]
Using the definition for indistinguishable states, we can conclude that
\[ \langle j, \rho_j, os_j \rangle \sim \langle j, \rho'_j, os'_j \rangle \]
and finally we know that
\[ \langle i, \rho_i, os_i \rangle \sim_v \langle j, \rho_j, os_j \rangle \land \langle i, \rho'_i, os'_i \rangle \sim_v \langle j, \rho'_j, os'_j \rangle \land \langle i, \rho_i, os_i \rangle \sim \langle i, \rho'_i, os'_i \rangle \Rightarrow \langle j, \rho_j, os_j \rangle \sim \langle j, \rho'_j, os'_j \rangle \]
is fulfilled.

\[ \square \]

**Proof: pop**

Using the semantic of the instruction, we know that
\[ \rho_i \sim \rho'_i \land \rho_i = \rho_j \land \rho'_i \sim (sl_{v_i} \land pt, pt') \land os_i \sim (sl_f(v'_i, pt')) \land os_i = v \land os_i = v' \land os' \land os_j = os \land os_j' = os' \]
And we can conclude that
\[\rho_j \sim \rho'_j \wedge os_i \sim (sl_{i,p} = pt_{sl_{i,p}},sl_{i,p}';pt') \quad os'_i \wedge os_i = v \quad os_i \wedge os_j = v' \quad os' \wedge os_j = os \wedge os'_j = os'\]

In consequence,
\[\rho_j \sim \rho'_j \wedge v \sim os \sim (sl_{i,p} = pt_{sl_{i,p}},sl_{i,p}';pt') \quad v' \sim os' \wedge os_j = f(v) \sim os \wedge os'_j = f(v') \sim os'\]

Using the definition for indistinguishable operand stacks, we know that
\[\rho_j \sim \rho'_j \wedge os \sim_{pt,pt'} os'_j \wedge os_j = os \wedge os'_j = os'\]

And we can conclude that
\[\rho_j \sim \rho'_j \wedge os_j \sim_{pt,pt'} os'_j\]

Using the definition for indistinguishable states, we can conclude that
\[\langle j, \rho_j, os_j \rangle \sim \langle j, \rho'_j, os'_j \rangle\]

and finally we know that
\[\langle i, \rho_i, os_i \rangle \sim_v \langle j, \rho_j, os_j \rangle \wedge \langle i, \rho'_i, os'_i \rangle \sim_v \langle j, \rho'_j, os'_j \rangle \wedge \langle i, \rho_i, os_i \rangle \sim \langle i, \rho'_i, os'_i \rangle \Rightarrow \langle j, \rho_j, os_j \rangle \sim \langle j, \rho'_j, os'_j \rangle\]
is fulfilled.

\[\square\]

**Proof:** push \(v\)

Using the semantic of the instruction, we know that
\[\rho_i \sim \rho'_i \wedge \rho_i = \rho_j \wedge \rho'_i = \rho'_j \wedge os_i \sim_{pt,pt'} os'_j \wedge os_j = v \sim os_i \wedge os'_j = v \sim os'_i\]
is fulfilled.

And we can conclude that
\[\rho_j \sim \rho'_j \wedge os_i \sim_{pt,pt'} os'_j \wedge os_j = v \sim os_i \wedge os'_j = v \sim os'_i\]

Using the definition for indistinguishable operand stacks, we know that
\[\rho_j \sim \rho'_j \wedge v \sim os_i \sim (sl_{i,p} = pt_{sl_{i,p}},sl_{i,p}';pt') \quad v \sim os'_i \wedge os_j = v \sim os'_i \wedge os'_j = v \sim os'_i\]

And we can conclude that
\[\rho_j \sim \rho'_j \wedge os_j \sim_{pt,pt'} os'_j\]

Using the definition for indistinguishable states, we can conclude that
\[\langle j, \rho_j, os_j \rangle \sim \langle j, \rho'_j, os'_j \rangle\]

and finally we know that
\[\langle i, \rho_i, os_i \rangle \sim_v \langle j, \rho_j, os_j \rangle \wedge \langle i, \rho'_i, os'_i \rangle \sim_v \langle j, \rho'_j, os'_j \rangle \wedge \langle i, \rho_i, os_i \rangle \sim \langle i, \rho'_i, os'_i \rangle \Rightarrow \langle j, \rho_j, os_j \rangle \sim \langle j, \rho'_j, os'_j \rangle\]
is fulfilled.

\[\square\]

**Proof:** load local \(x\)

Using the semantic of the instruction, we know that
\[\rho_i \sim \rho'_i \wedge \rho_i = \rho_j \wedge \rho'_i = \rho'_j \wedge os_i \sim_{pt,pt'} os'_j \wedge os_j = \rho_i(x) \wedge os'_i \wedge os_j = \rho'_i(x) \wedge os'_i \wedge \rho_i(x) = \rho'_i(x)\]
is fulfilled.

And we can conclude that
\[\rho_j \sim \rho'_j \wedge os_i \sim_{pt,pt'} os'_j \wedge os_j = \rho_i(x) \wedge os'_i \wedge os'_j = \rho'_i(x) \wedge os'_i \wedge \rho_i(x) = \rho'_i(x)\]

We have to distinguish between two cases. Either the information resource is visible, \(SL(x) \leq sl_{obs}\) or the information resource is invisible \(SL(x) > sl_{obs}\).

1. case: \(SL(x) \leq sl_{obs}\)

Using the definition of indistinguishable local mappings, we know that
\[\rho_j \sim \rho'_j \wedge os_i \sim_{pt,pt'} os'_i \wedge os_j = \rho_i(x) \wedge os'_i \wedge os'_j = \rho'_i(x) \wedge os'_i \wedge \rho_i(x) = \rho'_i(x)\]
C.2 Virtual Steps

Using the definition for indistinguishable operand stacks, we know that
\[ \rho_j \sim \rho'_j \land \rho_i(x) :: os_i \sim (SL(x) \downarrow se(i), PL(x)) :: pt(SL(x) \downarrow se'(i), PL(x)) :: pt' \rho'_i :: os'_i \land os_j = \rho_i(x) :: os_i \land os'_j = \rho'_i(x) :: os'_i \land \rho_i(x) = \rho'_i(x) \]

And we can conclude that
\[ \rho_j \sim \rho'_j \land os_j \sim (SL(x) \downarrow se(i), PL(x)) :: pt(SL(x) \downarrow se'(i), PL(x)) :: pt' \rho'_j \land os'_j \]

Using the definition for indistinguishable states, we can conclude that
\[ \langle j, j, os_j \rangle \sim \langle j, j', os'_j \rangle \]

and finally we know that
\[ \langle i, \rho_i, os_i \rangle \sim \langle j, \rho_j, os_j \rangle \land \langle i, \rho_i', os'_i \rangle \sim \langle j, \rho_j', os'_j \rangle \Rightarrow \langle j, j, os_j \rangle \sim \langle j, j', os'_j \rangle \]

2. case: \( SL(x) > sl_{obs} \)

Using the definition of indistinguishable local mappings, we know that
\[ \rho_j \sim \rho'_j \land \rho_i(x) :: os_i \sim pt, pt' os'_i \land os_j = \rho_i(x) :: os_i \land os'_j = \rho'_i(x) :: os'_i \land SL(x) > sl_{obs} \]

Using the definition for indistinguishable operand stacks, we know that
\[ \rho_j \sim \rho'_j \land \rho_i(x) :: os_i \sim (SL(x) \downarrow se(i), PL(x)) :: pt(SL(x) \downarrow se'(i), PL(x)) :: pt' \rho'_i :: os'_i \land os_j = \rho_i(x) :: os_i \land os'_j = \rho'_i(x) :: os'_i \land \rho_i(x) = \rho'_i(x) \]

And we can conclude that
\[ \rho_j \sim \rho'_j \land os_j \sim (SL(x) \downarrow se(i), PL(x)) :: pt(SL(x) \downarrow se'(i), PL(x)) :: pt' os'_j \]

Using the definition for indistinguishable states, we can conclude that
\[ \langle j, j, os_j \rangle \sim \langle j, j', os'_j \rangle \]

and finally we know that
\[ \langle i, \rho_i, os_i \rangle \sim \langle j, \rho_j, os_j \rangle \land \langle i, \rho'_i, os'_i \rangle \sim \langle j, \rho'_j, os'_j \rangle \Rightarrow \langle j, j, os_j \rangle \sim \langle j, j', os'_j \rangle \]

is fulfilled.

\[ \square \]

**Proof: storelocal x**

Using the semantic of the instruction, we know that
\[ \rho_i \sim \rho'_i \land \rho_i \downarrow \{ x \mapsto v \} = \rho_j \land \rho'_j \downarrow \{ x \mapsto v' \} = \rho'_j \land \rho_i \downarrow sl_v = \rho'_i \land \rho_i \downarrow os_i = v : os \land os'_i = v' : os' \land os_j = os \land os'_j = os' \]

is fulfilled.

And we can conclude that
\[ \rho_i \sim \rho'_i \land \rho_i \downarrow \{ x \mapsto v \} = \rho_j \land \rho'_j \downarrow \{ x \mapsto v' \} = \rho'_j \land \rho_i \downarrow v = v' \land os_j = os \land os'_j = os' \land v = v' \]

We have to distinguish two cases \( sl_v \leq sl_{obs} \land sl'_v \leq sl_{obs} \) and \( sl_v > sl_{obs} \land sl'_v > sl_{obs} \)

1. case: \( sl_v \leq sl_{obs} \land sl'_v \leq sl_{obs} \)

Using the definition for indistinguishable operand stacks, we know that
\[ \rho_i \sim \rho'_i \land \rho_i \downarrow \{ x \mapsto v \} = \rho_j \land \rho'_j \downarrow \{ x \mapsto v' \} = \rho'_j \land os_j \sim pt, pt' os'_j \land v = v' \]

We can conclude that
\[ \rho_i \sim \rho'_i \land \rho_i \downarrow \{ x \mapsto v \} = \rho_j \land \rho'_j \downarrow \{ x \mapsto v' \} = \rho'_j \land os_j \sim pt, pt' os'_j \land v = v' \]

Using the definition for indistinguishable local mappings, we know that
\[ \rho_i \downarrow \{ x \mapsto v \} \sim \rho'_i \downarrow \{ x \mapsto v' \} \land \rho_i \downarrow \{ x \mapsto v \} = \rho_j \land \rho'_j \downarrow \{ x \mapsto v' \} = \rho'_j \land os_j \sim pt, pt' os'_j \]

And we can conclude that
\[ \rho_j \sim \rho_j' \land os_j \sim_{pt, pt'} os_j' \]

Using the definition for indistinguishable states, we can conclude that
\[ \langle i, \rho_j, os_i \rangle \sim_{v} \langle j, \rho_j, os_j \rangle \land \langle i, \rho_j', os_i' \rangle \sim_{v} \langle j, \rho_j', os_j' \rangle \land \langle i, \rho_i, os_i \rangle \sim \langle i, \rho_i', os_i' \rangle \]
\[ \Rightarrow \langle j, \rho_j, os_j \rangle \sim \langle j, \rho_j', os_j' \rangle \text{ is fulfilled.} \]

**2. Case:** \( sl_v > sl_{obs} \land sl'_v > sl_{obs} \)

Using the abstract transfer rule, we know that
\[ \rho_i \sim \rho'_i \land \rho_i \oplus \{ x \mapsto v \} = \rho_j \land \rho_j' \oplus \{ x \mapsto v' \} = \rho_j' \land v : : os \sim_{(sl_v, pt_v) : : pt'_v, pt'_v} os' \land SL(x) \geq sl_v \cup se(i) > sl_{obs} \land SL(x) \geq sl'_v \cup se(i) > sl_{obs} \]

Using the definition of indistinguishable local mappings, we can conclude that
\[ \rho_i \sim \rho'_i \land \rho_i \oplus \{ x \mapsto v \} = \rho_j \land \rho_j' \oplus \{ x \mapsto v' \} = \rho_j' \land v : : os \sim_{(sl_v, pt_v) : : pt'_v, pt'_v} os' \land os_j = os \land os_j' = os' \land os_j \]

We can conclude that
\[ \rho_j \sim \rho'_j \land v : : os \sim_{(sl_v, pt_v) : : pt'_v, pt'_v} os' \land os_j = os \land os_j' = os' \]

Using the definition for indistinguishable operand stacks, we know that
\[ \rho_j \sim \rho'_j \land os \sim_{pt, pt'_v} os' \land os_j = os \land os_j' = os' \land os_j \]

And now we can conclude that
\[ \rho_j \sim \rho'_j \land os_j \sim_{pt, pt'_v} os_j' \land os_j \]

Using the definition for indistinguishable states, we can conclude that
\[ \langle j, \rho_j, os_j \rangle \sim \langle j, \rho_j', os_j' \rangle \]

Finally, we know that
\[ \langle i, \rho_i, os_i \rangle \sim_{v} \langle j, \rho_j, os_j \rangle \land \langle i, \rho_i', os_i' \rangle \sim_{v} \langle j, \rho_j', os_j' \rangle \land \langle i, \rho_i, os_i \rangle \sim \langle i, \rho_i', os_i' \rangle \]
\[ \Rightarrow \langle j, \rho_j, os_j \rangle \sim \langle j, \rho_j', os_j' \rangle \text{ is fulfilled.} \]

\[ \square \]

**Proof: unconditionalJump t**

Using the semantic of the instruction, we know that
\[ \rho_i \sim \rho'_i \land \rho_i = \rho_j \land \rho_i' = \rho_j' \land os_i \sim_{pt, pt'} os'_i \land os_i = os_j \land os'_i = os_j' \text{ is fulfilled.} \]

And we can conclude that
\[ \rho_j \sim \rho'_j \land os_j \sim_{pt, pt'} os'_j \land os_j \]

Using the definition for indistinguishable states, we can conclude that
\[ \langle j, \rho_j, os_j \rangle \sim \langle j, \rho_j', os_j' \rangle \]

And finally we know that
\[ \langle i, \rho_i, os_i \rangle \sim_{v} \langle j, \rho_j, os_j \rangle \land \langle i, \rho_i, os_i \rangle \sim_{v} \langle j, \rho_j', os_j' \rangle \land \langle i, \rho_i, os_i \rangle \sim \langle i, \rho_i', os_i' \rangle \Rightarrow \langle j, \rho_j, os_j \rangle \sim \langle j, \rho_j', os_j' \rangle \text{ is fulfilled.} \]

\[ \square \]

**Proof: jumpWith1 Argument t**

Using the semantic of the instruction, we know that
C.2 Virtual Steps

We can conclude that $\rho_i \sim \rho'_i \land \rho_i = \rho_j \land \rho'_j = \rho'_j \land \os_i \sim (s_{e, pl_{e}}) :: pt, \os'_i) :: pt' \os'_i \land \os_i = v :: \os \land \os'_i = v' :: \os' \land \os_j = \os \land \os'_j = \os'$ is fulfilled.

We can conclude that $\rho_j \sim \rho'_j \land v :: \os \sim (s_{e, pl_{e}}) :: pt, \os'_i) :: pt' \os'_i \land \os_j = \os \land \os'_j = \os'$

Using the definition for indistinguishable stacks, we know that $\rho_j \sim \rho'_j \land \os \sim pt, pt' \os' \land \os_j = \os \land \os'_j = \os'$

And we can conclude that $\rho_j \sim \rho'_j \land \os_j \sim pt, pt' \os'_j$.

Using the definition for indistinguishable states, we can conclude that $\langle i, \rho_j, \os_j \rangle \sim \langle i, \rho'_j, \os'_j \rangle$

and finally we know that $\langle i, \rho_i, \os_i \rangle \sim_v \langle i, \rho_j, \os_j \rangle \land \langle i, \rho'_i, \os'_i \rangle \sim_v \langle i, \rho'_j, \os'_j \rangle \land \langle i, \rho_i, \os_i \rangle \sim \langle i, \rho_i, \os_i \rangle \Rightarrow \langle j, \rho_j, \os_j \rangle \sim \langle j, \rho'_j, \os'_j \rangle$ is fulfilled.

Proof: jumpWith2 Arguments t

Using the semantic of the instruction, we know that $\rho_i \sim \rho'_i \land \rho_i = \rho_j \land \rho'_i = \rho'_j \land \os_i \sim (s_{e, pl_{e}}) :: pt, \os'_i) :: pt' \os'_i \land \os_i = v_1 :: v_2 :: \os \land \os'_i = v'_1 :: v'_2 :: \os' \land \os_j = \os \land \os'_j = \os'$ is fulfilled.

We can conclude that $\rho_j \sim \rho'_j \land v_1 :: v_2 :: \os \sim (s_{e, pl_{e}}) :: pt, \os'_i) :: pt' \os'_i \land \os_j = \os \land \os'_j = \os'$

Using the definition for indistinguishable stacks, we know that $\rho_j \sim \rho'_j \land \os \sim pt, pt' \os' \land \os_j = \os \land \os'_j = \os'$

And we can conclude that $\rho_j \sim \rho'_j \land \os_j \sim pt, pt' \os'_j$.

Using the definition for indistinguishable states, we can conclude that $\langle j, \rho_j, \os_j \rangle \sim \langle j, \rho'_j, \os'_j \rangle$

and finally we know that $\langle i, \rho_i, \os_i \rangle \sim_v \langle j, \rho_j, \os_j \rangle \land \langle i, \rho'_i, \os'_i \rangle \sim_v \langle j, \rho'_j, \os'_j \rangle \land \langle i, \rho_i, \os_i \rangle \sim \langle i, \rho_i, \os_i \rangle \Rightarrow \langle j, \rho_j, \os_j \rangle \sim \langle j, \rho'_j, \os'_j \rangle$ is fulfilled.

Proof: load localaddress x

Using the semantic of the instruction, we know that $\rho_i \sim \rho'_i \land \rho_i = \rho_j \land \rho'_i = \rho'_j \land \os_i \sim pt, pt' \os'_i \land \os_j = \&(x) :: \os_i \land \os'_j = \&(x) :: \os'_j$ is fulfilled.

And we can conclude that $\rho_j \sim \rho'_j \land \os_i \sim pt, pt' \os'_i \land \os_j = \&(x) :: \os_i \land \os'_j = \&(x) :: \os'_j$

Using the definition for indistinguishable stacks, we know that $\rho_j \sim \rho'_j \land \&(x) :: \os_i \sim (s_{e, l}) :: pt, \os'_i) :: pt' \&(x) :: \os'_i \land \os_j = \&(x) :: \os_i \land \os'_j = \&(x) :: \os'_j$.
And we can conclude that
\[ \rho_j \sim \rho'_j \land \os_j \sim_{pt,pt'} \os'_j \]
Using the definition for indistinguishable states, we can conclude that
\[ \langle j, \rho_j, \os_j \rangle \sim \langle j, \rho'_j, \os'_j \rangle \]
and finally we know that
\[ \langle i, \rho_i, \os_i \rangle \sim_o \langle j, \rho_j, \os_j \rangle \land \langle i, \rho_i, \os_i \rangle \sim_o \langle j, \rho_j, \os_j \rangle \land \langle i, \rho_i, \os_i \rangle \sim_o \langle i, \rho'_i, \os'_i \rangle \]
\[ \Rightarrow \langle j, \rho_j, \os_j \rangle \sim \langle j, \rho'_j, \os'_j \rangle \]

**Proof: load indirect**

Using the semantic of the instruction, we know that
\[ \rho_i \sim \rho'_i \land \rho_1 = \rho_j \land \rho'_1 = \rho'_j \land \os_i \sim_{(sl_{k(x)}:pl_{k(x)}):pt,(sl'_{k(x)}:pl'_{k(x)}):pt'} \os'_i \]
\[ \land \os_i = \& (x) \land \os'_i = \& (x) \land \os_j = \rho_i (x) \land \os'_j = \rho'_i (x) \]
\[ \land \os_i \sim \os'_i \land \os_i \sim \os'_i \land \os_j \sim \os_j \]

We have to distinguish two cases. Either all information is visible \( sl_{k(x)} \leq sl_{obs} \land sl'_{k(x)} \leq sl_{obs} \) or either the address is invisible or the resource that is referenced by the address \( sl_{k(x)} \) is not visible.

1. case: \( sl_{k(x)} \leq sl_{obs} \land sl'_{k(x)} \leq sl_{obs} \land pl_{k(x)} \leq sl_{obs} \land pl'_{k(x)} \leq sl_{obs} \)

Using the definition of indistinguishable local mappings, we know that
\[ \rho_j \sim \rho'_j \land \os_i \sim_{(sl_{k(x)}:pl_{k(x)}):pt,(sl'_{k(x)}:pl'_{k(x)}):pt'} \os'_i \land \os_i = \& (x) \land \os'_i \]
\[ \land \os'_i \sim \os'_i \land \os_j = \rho_i (x) \land \os'_j = \rho'_i (x) \land \rho_i (x) = \rho'_i (x) \]
Using the definition of indistinguishable operand stacks, we can conclude
\[ \rho_j \sim \rho'_j \land \os_j \sim_{pt,pt'} \os'_j \land \rho_j = \rho'_j \land \os'_i = \os'_i \land \rho_i (x) = \rho'_i (x) \]
Using the definition of indistinguishable operand stacks, we know that
\[ \rho_j \sim \rho'_j \land \os_j \sim_{pt,pt'} \os'_j \land \os_j = \rho_i (x) \land \os'_j = \rho'_i (x) \land \os'_i = \os'_i \land \rho_i (x) = \rho'_i (x) \]
And we can conclude that
\[ \rho_j \sim \rho'_j \land \os_j \sim_{pt,pt'} \os'_j \]
Using the definition for indistinguishable states, we can conclude that
\[ \langle j, \rho_j, \os_j \rangle \sim \langle j, \rho'_j, \os'_j \rangle \]
and finally we know that
\[ \langle i, \rho_i, \os_i \rangle \sim_o \langle j, \rho_j, \os_j \rangle \land \langle i, \rho_i, \os'_i \rangle \sim_o \langle j, \rho'_j, \os'_j \rangle \land \langle i, \rho_i, \os_i \rangle \sim_o \langle i, \rho'_i, \os'_i \rangle \]
\[ \Rightarrow \langle j, \rho_j, \os_j \rangle \sim \langle j, \rho'_j, \os'_j \rangle \]

2. case: \( sl_{k(x)} > sl_{obs} \land sl'_{k(x)} \leq sl_{obs} \land pl_{k(x)} \leq sl_{obs} \land pl'_{k(x)} > sl_{obs} \)

Using the definition of indistinguishable operand stacks, we can conclude
\[ \rho_j \sim \rho'_j \land \os_j \sim_{pt,pt'} \os'_j \land \os_j = \rho_i (x) \land \os'_j = \rho'_i (x) \land \os'_i \]
\[ \land (sl_{k(x)} > sl_{obs} \land sl'_{k(x)} \leq sl_{obs} \land pl_{k(x)} \leq sl_{obs} \land pl'_{k(x)} > sl_{obs}) \]
We can conclude that
\[ \rho_j \sim \rho'_j \land \os_j \sim_{pt,pt'} \os'_j \land \os_j = \rho_i (x) \land \os'_j = \rho'_i (x) \land \os'_i \]
\[ \land \text{ref}_{k(x)} \cup p_l_{k(x)} \cup sce(i) > sl_{obs} \land s'_{ref}\cup p'u_{k(x)} \cup sce'(i) > sl_{obs} \]

Using the definition for indistinguishable operand stacks, we know that

\[ \rho_j \sim \rho'_j \land \rho_i(x) :: os \sim (\text{ref}_{k(x)} \cup p_l_{k(x)} \cup sce(i))::pt.(s_{ref},p_{ref})::(sl_{ref},p_{ref})::(pt',p'_{ref})::os' \]

\[ \land os_j = \rho_i(x) :: os \land os'_j = \rho'_i(x) :: os' \]

\[ \text{and we can conclude that} \]

\[ \rho_j \sim \rho'_j \land os_j \sim pt.p't \land os'_j \]

Using the definition for indistinguishable states, we can conclude that

\[ \langle j, \rho_j, os_j \rangle \sim \langle j, \rho'_j, os'_j \rangle \]

and finally we know that

\[ \langle i, \rho_i, os_i \rangle \sim_v \langle j, \rho_j, os_j \rangle \land \langle i, \rho'_i, os'_i \rangle \sim_v \langle j, \rho'_j, os'_j \rangle \land \langle i, \rho_i, os_i \rangle \sim \langle i, \rho'_i, os'_i \rangle \]

\[ \Rightarrow \langle j, \rho_j, os_j \rangle \sim \langle j, \rho'_j, os'_j \rangle \] is fulfilled.

\[ \square \]

**Proof:** \text{storeindirect}

Using the semantic of the instruction, we know that

\[ \rho_i \sim \rho'_i \land os_i \sim (s_{ref},p_{ref})::(sl_{ref},p_{ref})::(p_l_{ref},p_{ref})::(sl_{ref},p_{ref})::(pt',p'_{ref})::os' \]

\[ \land \rho_i \oplus \{ x \mapsto v \} = \rho_j \land \rho'_i \oplus \{ x' \mapsto v' \} = \rho'_j \]

\[ \land os_i = v::&(x) :: os \land os'_i = v'::&(x') :: os' \land os_j = os \land os'_j = os' \]

\[ \text{is fulfilled.} \]

And we can conclude that

\[ \rho_i \sim \rho'_i \land v :: &(x) :: os \sim (s_{ref},p_{ref})::(sl_{ref},p_{ref})::(p_l_{ref},p_{ref})::(sl_{ref},p_{ref})::(pt',p'_{ref})::os' \]

\[ \land \rho_i \oplus \{ x \mapsto v \} = \rho_j \land \rho'_i \oplus \{ x' \mapsto v' \} = \rho'_j \land os_j = os \land os'_j = os' \land v = v' \land & (x) = & (x') \]

We have to distinguish two cases. Either all information on the stack is from visible resources \( s_l \leq sl_{obs} \land s'_l \leq sl_{obs} \land sl_{k(x)} \leq sl_{obs} \) or one of the information resources is invisible \( s_l > sl_{obs} \land s'_l > sl_{obs} \lor sl_{k(x)} > sl_{obs} \land sl_{k(x')} > sl_{obs} \)

1. **case:** \( s_l \leq sl_{obs} \land s'_l \leq sl_{obs} \land sl_{k(x)} \leq sl_{obs} \land sl_{k(x')} \leq sl_{obs} \)

Using the definition for indistinguishable operand stacks, we know that

\[ \rho_i \sim \rho'_i \land v :: &(x) :: os \sim (s_{ref},p_{ref})::(sl_{ref},p_{ref})::(p_l_{ref},p_{ref})::(sl_{ref},p_{ref})::(pt',p'_{ref})::os' \]

\[ \land \rho_i \oplus \{ x \mapsto v \} = \rho_j \land \rho'_i \oplus \{ x' \mapsto v' \} = \rho'_j \land os_j = os \land os'_j = os' \land v = v' \land & (x) = & (x') \]

And we can conclude that

\[ \rho_i \sim \rho'_i \land \rho_i \oplus \{ x \mapsto v \} = \rho_j \land \rho'_i \oplus \{ x' \mapsto v' \} = \rho'_j \land os_j \sim_{pt,p't} os'_j \land v = v' \land & (x) = & (x') \]

Using the definition for indistinguishable local mappings, we know that

\[ \rho_i \oplus \{ x \mapsto v \} \sim \rho'_i \oplus \{ x' \mapsto v' \} \land \rho_i \oplus \{ x \mapsto v \} = \rho_j \land \rho'_i \oplus \{ x' \mapsto v' \} = \rho'_j \land os_j \sim_{pt,p't} os'_j \]

And we can conclude that

\[ \rho_j \sim \rho'_j \land os_j \sim_{pt,p't} os'_j \]

Using the definition for indistinguishable states, we can conclude that

\[ \langle j, \rho_j, os_j \rangle \sim \langle j, \rho'_j, os'_j \rangle \]

and finally we know that

\[ \langle i, \rho_i, os_i \rangle \sim_v \langle j, \rho_j, os_j \rangle \land \langle i, \rho'_i, os'_i \rangle \sim_v \langle j, \rho'_j, os'_j \rangle \land \langle i, \rho_i, os_i \rangle \sim \langle i, \rho'_i, os'_i \rangle \]

\[ \Rightarrow \langle j, \rho_j, os_j \rangle \sim \langle j, \rho'_j, os'_j \rangle \] is fulfilled.

2. **case:** \( s_l > sl_{obs} \land s'_l > sl_{obs} \lor sl_{k(x)} > sl_{obs} \land sl_{k(x')} > sl_{obs} \)

Using the abstract transfer rule, we know that

\[ \rho_i \sim \rho'_i \land v :: &(x) :: os \sim (s_{ref},p_{ref})::(sl_{ref},p_{ref})::(p_l_{ref},p_{ref})::(sl_{ref},p_{ref})::(pt',p'_{ref})::os' \]

\[ \land v = v' \land & (x) = & (x') \]
And we can conclude that
\[ \rho_j \sim \rho_j' \wedge \rho_i \sim \rho_i' \] is fulfilled.

\[ \square \]

**Proof: return**

This case is a special case, because we do not have a transfer of the type \( \langle i, \rho_i, os_i \rangle \sim_\nu \langle j, \rho_j, os_j \rangle \), but \( \langle i, \rho_i, os_i \rangle \sim \rho_j, v \)

Using the semantic of the instruction, we know that
\[ \rho_j \sim \rho_j' \wedge \rho_i \sim (\rho_i', os_i) \sim \nu^o \langle i, \rho_i', os_i' \rangle \sim (\rho_j, \rho_j', os_j') \wedge \langle i, \rho_i, os_i \rangle \sim (i, \rho_i', os_i') \]

and finally we know that
\[ (i, \rho_i, os_i) \sim_v (j, \rho_j, os_j) \wedge \langle i, \rho_i', os_i' \rangle \sim \nu^o \langle i, \rho_i', os_i' \rangle \wedge \langle i, \rho_i, os_i \rangle \sim (i, \rho_i', os_i') \]
\[ \Rightarrow (j, \rho_j, os_j) \sim (j, \rho_j', os_j') \]

is fulfilled.

\[ \square \]

**Proof: High Region**

Using the lemma “High Regions Preserve Indistinguishability”, we know that
\[ \text{se}(i) > sl_{obs} \wedge (i, \rho_i, os_i) \sim_v (j, \rho_j, os_j) \wedge \langle i, \rho_i', os_i' \rangle \sim_v (j, \rho_j', os_j') \wedge \langle i, \rho_i, os_i \rangle \sim (i, \rho_i', os_i') \]
\[ \Rightarrow (j, \rho_j, os_j) \sim (j, \rho_j', os_j') \]

In consequence, we can conclude that
\[ \langle i, \rho_i, os_i \rangle \sim_v (j, \rho_j, os_j) \wedge \langle i, \rho_i', os_i' \rangle \sim_v (j, \rho_j', os_j') \wedge \langle i, \rho_i, os_i \rangle \sim (i, \rho_i', os_i') \]
\[ \Rightarrow (j, \rho_j, os_j) \sim (j, \rho_j', os_j') \]

is fulfilled.
From these proofs we can conclude that every virtual step preserves the indistinguishability of states.
D Source Code of PNIC

In this section, the complete source code of the “PNIC” can be found. The source code is released under the GPL version 2. The complete source code can be found on the cd-rom in the folder /src/PNIC/.

D.1 Package Model

D.1.1 File Analysis/AbstractState.cs

```csharp
using System;
using System.Collections.Generic;
using System.Collections;
using System.Text;
using PNIC.Model.AST;

namespace PNIC.Model.Analysis
{
    /// <summary>
    /// the class representing the abstract state during an analysis
    /// </summary>
    public class AbstractState
    {
        // state
        private HashTable state;
        private Stack<SecurityLevel> recentSecurityLevelStack;

        // recentSecurityLevelStack
        public Stack<SecurityLevel> RecentSecurityLevelStack
        {
            get { return recentSecurityLevelStack; } 
        }

        // getSecurityLevelStack
        public Stack<SecurityLevel> getSecurityLevelStack(Instruction instruction)
        {
            if (instruction == null)
                return new Stack<SecurityLevel>();
            Stack<SecurityLevel> buffer = new Stack<SecurityLevel>();
            Stack<SecurityLevel> result = new Stack<SecurityLevel>();
            Stack<SecurityLevel> stored = (Stack<SecurityLevel>)state[instruction];

            if (stored != null)
            {
                int stacksize = stored.Count;
                for (int i = 0; i < stacksize; i++)
                {
                    buffer.Push(stored.Pop());
                }
            }

            for (int i = 0; i < stacksize; i++)
            {
                SecurityLevel securityLevel = buffer.Pop();
            }
        }
    }
}```
D.1 Package Model

```csharp
D.1.2 File Analysis/AnalysisSettings.cs

using System.Collections.Generic;
using System.Text;
using PNIC.Model.AST;
using System.Collections;
using System.Xml;
using System;
using PNIC.Model.AST.CIL;

namespace PNIC.Model.Analysis
{
    class AnalysisSettings
    {
        public AnalysisSettings()
    }
}
```
```csharp
{    methodsToCheck = new Hashtable();
  securityMappings = new Hashtable();
  regions = new Hashtable();
  name = "";
}

private string name;
public string Name
{
  get { return name; }
}

/// <summary>
/// unused; may later be used to determine the analysis/language to analyse
/// </summary>
private string analysisType;

private Hashtable methodsToCheck;

/// <summary>
/// Hashtable containing security level associations for information resources
/// </summary>
private Hashtable securityMappings;

/// <summary>
/// a human readable representation of this settings
/// </summary>
public string ReadableConfiguration
{
  get
  {
    StringBuilder readable = new StringBuilder();
    foreach (string asmname in getAssembliesToCheck())
    {
      foreach (string typename in getTypesToCheck(asmname))
      {
        foreach (string methodname in getMethodsToCheck(asmname, typename))
        {
          readable.AppendLine(asmname + "/" + typename + "/" + methodname + "\n");
          foreach (object varname in securityMappings.Keys)
          {
            if (((string) varname).StartsWith(asmname + "/" + typename + "/" + methodname))
            {
              string shortname = ((string) varname).Split('/')[1];
              readable.AppendLine("\t" + shortname + ": " + securityMappings["" + varname]);
            }
          }
        }
      }
    }
  }
```

79
D.1 Package Model

```csharp
private Hashtable regions;

public void addMethodToCheck(string assemblyName, string typeName, string method)
{
    if ( !(methodsToCheck[assemblyName] is Hashtable) )
        methodsToCheck[assemblyName] = new Hashtable();
    if ( !((((Hashtable)methodsToCheck[assemblyName])[typeName] is List<string>))
        (((Hashtable)methodsToCheck[assemblyName])[typeName] = new List<string>());
    (((List<string>)(((Hashtable)methodsToCheck[assemblyName])[typeName])).Add(method);
}

public List<string> getAssembliesToCheck()
{
    List<string> assemblyNames = new List<string>();
    foreach (Object obj in methodsToCheck.Keys)
    {
        if (obj is string)
            assemblyNames.Add((string)obj);
    }
    return assemblyNames;
}

public List<string> getTypesToCheck(string assemblyName)
{
    if (methodsToCheck[assemblyName] is Hashtable)
    {
        List<string> typeNames = new List<string>();
        foreach (Object obj in ((Hashtable)methodsToCheck[assemblyName]).Keys)
        {
            if (obj is string)
                typeNames.Add((string)obj);
        }
        return typeNames;
    }
```
D.1 Package Model

```csharp
return new List<string>();
```

```csharp
/// <summary>
/// retrieves all methods in a type that are marked for analysis
/// </summary>
/// <param name="assemblyname">containing assembly</param>
/// <param name="typename">containing type</param>
/// <returns> list of all methods to analyze </returns>
public List<string> getMethodsToCheck(string assemblyname, string typename)
{
    if (methodsToCheck[assemblyname] is HashTable)
    {
        if (((HashTable)methodsToCheck[assemblyname])[typename] is List<string>)
            return (List<string>)((HashTable)methodsToCheck[assemblyname])[typename];
    }
    return new List<string>();
}
```

```csharp
/// <summary>
/// loads analysis settings from a file
/// </summary>
/// <param name="configurationfilename">file name of the settings file</param>
public void loadConfigurationFromFile(string configurationfilename)
{
    if (configurationfilename != null && System.IO.File.Exists(configurationfilename))
    {
        try
        {
            XmlDocument configurationFile = new XmlDocument();
            configurationFile.Load(configurationfilename);

            XmlNodeReader reader = new XmlNodeReader(configurationFile);
            reader.ReadToFollowing("analysis");
            analysisType = reader.GetAttribute("type");

            methodsToCheck = new HashTable();

            while (reader.ReadToFollowing("assembly"))
            {
                string assemblyName = reader.GetAttribute("file");

                XmlReader typeReader = reader.ReadSubtree();
                while (typeReader.ReadToFollowing("type"))
                {
                    string typeName = typeReader.GetAttribute("id");

                    XmlReader mtdReader = typeReader.ReadSubtree();
                    while (mtdReader.ReadToFollowing("method"))
                    {
                        string methodName = mtdReader.GetAttribute("id");
                        addMethodToCheck[assemblyName, typeName, methodName];
                    }
                    XmlReader varReader = mtdReader.ReadSubtree();
                    while (varReader.Read())
                    {
                        string key = assemblyName + "/" + typeName + "/" + methodName;
                        switch (varReader.LocalName.ToLower())
                        {
                            case "argument":
                                case "variable":
                                    key = key + "/" + varReader.GetAttribute("id");
                                    string level = varReader.GetAttribute("level");
                                    addVariableMapping(key, level);
                                    break;
                            case "region":
                                key = key + varReader.GetAttribute("index");
                                break;
                        }
                    }
                }
            }
        }
    }
```
```csharp
string regionString = varReader.ReadString();
addRegion(key, regionString);
break;
case "junction":
    key = key + varReader.GetAttribute("index");
    int junctionPoint = int.Parse(varReader.GetAttribute("instruction"));
    addJunctionMapping(key, junctionPoint);
    break;
}
}

this.name = configurationfilename;
}
catch (XmlException ex)
{
    Console.Out.WriteLine("Error in configuration file: " + ex.Message);
}
}

/// <summary>
/// retrieves the control dependency region associated with a branching instruction
/// </summary>
/// <param name="instruction">associated branching instruction</param>
/// <returns>control dependency region for a branching instruction</returns>
public ControlDependencyRegion getControlDependencyRegion(IInstruction instruction)
{
    StringBuilder key = new StringBuilder();
    key.Append(instruction.ContainingMethod.ContainingType.ContainingAssembly.Name);
    key.Append("/");
    key.Append(instruction.ContainingMethod.ContainingType.Name);
    key.Append("/");
    key.Append(instruction.ContainingMethod.Name);
    key.Append(instruction.Position);
    return (ControlDependencyRegion)regions[key.ToString()];
}

/// <summary>
/// adds a junction to a control dependency region
/// </summary>
/// <param name="key">the fully qualified position of the branching instruction</param>
/// <param name="junction">the junction instruction</param>
private void addJunctionMapping(string key, int junctionPoint)
{
    ControlDependencyRegion cdr = (ControlDependencyRegion)regions[key];
    if (cdr == null)
    {
        cdr = new ControlDependencyRegion();
        regions[key] = cdr;
    }
    cdr.Junction = junctionPoint;
}

/// <summary>
/// adds a region to a control dependency region
/// </summary>
/// <param name="key">the fully qualified position of the branching instruction</param>
/// <param name="region">string describing the range of the region</param>
private void addRegion(string key, string regionString)
{
    ControlDependencyRegion cdr = (ControlDependencyRegion)regions[key];
    if (cdr == null)
    {
```
{  
cdr = new ControlDependencyRegion();
regions[key] = cdr;
}

foreach (string range in regionString.Split(','))
{
  try
  {
    if (range.Contains("-"))
    {
      int begin, end;
      begin = int.Parse(range.Split('-')[0]);
      end = int.Parse(range.Split('-')[1]);
      for (int i = begin; i <= end; i++)
      {
        cdr.addToRegion(i);
      }
    }
    else
    cdr.addToRegion(int.Parse(range));
  }
  catch (FormatException)
  {
    Console.Out.WriteLine("Region has an invalid format");
  }
}

/// <summary>
/// adds a variable-security level association
/// </summary>
/// <param name="key">the fully qualified variable name</param>
/// <param name="level">the security level to associate</param>
private void addVariableMapping(string key, string level)
{
  securityMappings[key] = level;
}

/// <summary>
/// sets the security environment of an instruction to a given level
/// </summary>
/// <param name="fullyQualifiedNameOfInstruction">the fully qualified position of the instruction</param>
/// <param name="securityEnvironment">the security level to set</param>
public void setSecurityEnvironment(string fullyQualifiedNameOfInstruction, SecurityLevel securityEnvironment)
{
  securityMappings[fullyQualifiedNameOfInstruction] = securityEnvironment.Name;
}

/// <summary>
/// retrieves the security environment of an instruction
/// </summary>
/// <param name="fullyQualifiedNameOfInstruction">the fully qualified instruction position</param>
/// <returns>the level of the security environment</returns>
public string getSecurityEnvironment(string fullyQualifiedNameOfInstruction)
{
  string environment = (string)securityMappings[fullyQualifiedNameOfInstruction];
  if (environment != null)
    return environment;
  return "";
}

public string getSecurityEnvironment(Instruction instruction)
{
  StringBuilder key = new StringBuilder();
  key.Append(instruction.ContainingMethod.ContainingType.ContainingAssembly.Name);
}
public string getSecurityLevel(Variable variable) {
    StringBuilder key = new StringBuilder();
    key.Append(variable.ContainingMethod.ContainingType.ContainingAssembly.Name);
    key.Append("/");
    key.Append(variable.ContainingMethod.Name);
    key.Append("/");
    key.Append(variable.ContainingMethod.Name);
    key.Append("/");
    key.Append(variable.Name);
    string level = (string)securityMappings[key.ToString()];
    if (level != null)
        return level;
    return "";
}

public void initializeNewConfiguration(string filename, List<string> assemblies) {
    if (!filename == null || filename.Equals(""))
    {
        XmlDocument xmlConfigFile = new XmlDocument();
        try
        {
            xmlConfigFile.Load(filename);
        }
        catch (System.IO.FileNotFoundException)
        {
            XmlTextWriter xmlWriter = new XmlTextWriter(filename, System.Text.Encoding.UTF8);
            xmlWriter.Formatting = Formatting.Indented;
            xmlWriter.WriteProcessingInstruction("xml", "version='1.0' encoding='UTF-8'");
            xmlWriter.WriteStartElement("analysis type='""");
            xmlWriter.Close();
            xmlConfigFile.Load(filename);
        }
    }
    XmlNode xmlanalysis = xmlConfigFile.DocumentElement;
    foreach (string asm in assemblies)
    {
        XmlElement xmlasm = xmlConfigFile.CreateElement("assembly");
        try
        {
            Assembly assembly = new Assembly(asm);
            xmlasm.SetAttribute("file", asm);
            xmlanalysis.AppendChild(xmlasm);
            foreach (PNIC.Model.AST.Type type in assembly.Typen)
D.1 Package Model

```csharp
using System;
using System.Collections.Generic;
using System.Text;
using PNIC.Model.AST;

namespace PNIC.Model.Analysis
{

    class PackageModel
    {
        XElement xmlclass = xmlConfigFile.CreateElement("type");
        xmlclass.SetAttribute("id", type.Name);
        xmlasm.AppendChild(xmlclass);
        foreach (Method method in type.MemberMethods)
        {
            if (method.Readable.EndsWith(";"))
            {
                XElement xmlmd = xmlConfigFile.CreateElement("method");
                xmlmd.SetAttribute("id", method.Name);
                xmlclass.AppendChild(xmlmd);
                if (method.LocalVariables != null)
                {
                    foreach (Variable parameter in method.Parameters)
                    {
                        XElement xmlarg = xmlConfigFile.CreateElement("argument");
                        xmlarg.SetAttribute("id", parameter.Name);
                        xmlarg.SetAttribute("level", "");
                        xmlmd.AppendChild(xmlarg);
                    }
                    foreach (Variable variable in method.LocalVariables)
                    {
                        XElement xmlvar = xmlConfigFile.CreateElement("variable");
                        xmlvar.SetAttribute("id", variable.Name);
                        xmlvar.SetAttribute("level", "");
                        xmlmd.AppendChild(xmlvar);
                    }
                    foreach (Instruction instruction in method.Instructions)
                    {
                        if (instruction is ConditionalJump)
                        {
                            XElement xmlregion = xmlConfigFile.CreateElement("region");
                            xmlregion.SetAttribute("index", instruction.Position.ToString());
                            XElement xmljunction = xmlConfigFile.CreateElement("junction");
                            xmljunction.SetAttribute("index", instruction.Position.ToString());
                            xmljunction.SetAttribute("instruction", "");
                            xmlmd.AppendChild(xmlregion);
                            xmlmd.AppendChild(xmljunction);
                        }
                    }
                }
            }
        }
    }
}
```

D.1.3 File Analysis/ControlDependencyRegion.cs

```csharp
namespace PNIC.Model.Analysis
{

    class ControlDependencyRegion
    {
        public ControlDependencyRegion
        {
            foreach (Method method in type.MemberMethods)
            {
                if (method.Readable.EndsWith(";"))
                {
                    XElement xmlmd = xmlConfigFile.CreateElement("method");
                    xmlmd.SetAttribute("id", method.Name);
                    xmlclass.AppendChild(xmlmd);
                    if (method.LocalVariables != null)
                    {
                        foreach (Variable parameter in method.Parameters)
                        {
                            XElement xmlarg = xmlConfigFile.CreateElement("argument");
                            xmlarg.SetAttribute("id", parameter.Name);
                            xmlarg.SetAttribute("level", "");
                            xmlmd.AppendChild(xmlarg);
                        }
                        foreach (Variable variable in method.LocalVariables)
                        {
                            XElement xmlvar = xmlConfigFile.CreateElement("variable");
                            xmlvar.SetAttribute("id", variable.Name);
                            xmlvar.SetAttribute("level", "");
                            xmlmd.AppendChild(xmlvar);
                        }
                        foreach (Instruction instruction in method.Instructions)
                        {
                            if (instruction is ConditionalJump)
                            {
                                XElement xmlregion = xmlConfigFile.CreateElement("region");
                                xmlregion.SetAttribute("index", instruction.Position.ToString());
                                XElement xmljunction = xmlConfigFile.CreateElement("junction");
                                xmljunction.SetAttribute("index", instruction.Position.ToString());
                                xmljunction.SetAttribute("instruction", "");
                                xmlmd.AppendChild(xmlregion);
                                xmlmd.AppendChild(xmljunction);
                            }
                        }
                    }
                }
            }
        }
    }
}
```
class ControlDependencyRegion
{
    public ControlDependencyRegion()
    {
        region = new List<int>();
        junction = -1;
    }

    private List<int> region;
    public List<int> Region
    {
        get { return region; }
    }

    private int junction;
    public int Junction
    {
        get { return junction; }
        set { this.junction = value; }
    }

    public void addToRegion(int instructionPosition)
    {
        if (region.Contains(instructionPosition))
            return;
        region.Add(instructionPosition);
    }
}

D.1.4 File Analysis/Flowpolicy.cs

namespace PNIC.Model.Analysis
{
    class FlowPolicy
    {
        public FlowPolicy()
        {
            securityLevels = new Hashtable();
            name = "";
        }

        private string name;
        public string Name
        {
            get { return name; }
            set { name = value; }
        }

        public string Readable
D.1 Package Model

```java
{
    get {
        StringBuilder readable = new StringBuilder();
        readable.AppendLine("Flow policy: " + Name);
        readable.AppendLine();
        readable.AppendLine("Security levels:");
        for (Object securityLevel in securityLevels.Values)
        {
            readable.AppendLine((securityLevel).Name + " allows flow to ");
            for (SecurityLevel legalFlow in this.getLegalFlow((SecurityLevel) securityLevel))
            {
                readable.AppendLine(legalFlow.Name + ", ");
            }
            readable.AppendLine();
        }
        return readable.ToString();
    }
}
```

```
/// <summary>
/// collection of all security levels in the flow policy
/// </summary>
private Hashtable securityLevels;

/// <summary>
/// the lowest security level in the flow policy
/// </summary>
private SecurityLevel lowestSecurityLevel;

public SecurityLevel LowestSecurityLevel
{
    get { return lowestSecurityLevel; }
}

/// <summary>
/// adds a security level to the flow policy
/// </summary>
/// <param name="securityLevel">the security level to add</param>
public void addSecurityLevel(SecurityLevel securityLevel)
{
    securityLevels[securityLevel.Name.ToLower()] = securityLevel;
    calculateLowestSecurityLevel();
}

/// <summary>
/// retrieves the security level identified by its name
/// </summary>
/// <param name="name">the name of the security level</param>
/// <returns>security level associated with the name</returns>
public SecurityLevel getSecurityLevel(string name)
{
    SecurityLevel securityLevel = (SecurityLevel)securityLevels[name.ToLower()];
    if (securityLevel == null)
    {
        securityLevel = lowestSecurityLevel;
        return securityLevel;
    }
}

/// <summary>
/// retrieves all security levels information may flow to from a given security level
/// </summary>
/// <param name="sourceLevel">the level of the information source</param>
/// <returns>a list of levels the information may flow to</returns>
public List<SecurityLevel> getLegalFlow(SecurityLevel sourceLevel)
{
    if (sourceLevel == null)
    {
        return new List<SecurityLevel>();
    }
    List<SecurityLevel> legalFlow = new List<SecurityLevel>();
```
if (!legalFlow.Contains(sourceLevel))
    legalFlow.Add(sourceLevel);
foreach (SecurityLevel higherSecurityLevel in sourceLevel.HigherSecurityLevels)
{
    foreach (SecurityLevel securityLevel in getLegalFlow(higherSecurityLevel))
    {
        if (!legalFlow.Contains(securityLevel))
            legalFlow.Add(securityLevel);
    }
} return legalFlow;

private SecurityLevel getLeastUpperBound(SecurityLevel securityLevel1, SecurityLevel securityLevel2)
{
    if (securityLevel1 == securityLevel2)
        return securityLevel1;
    if (getLegalFlow(securityLevel1).Contains(securityLevel2))
        return securityLevel2;
    if (getLegalFlow(securityLevel2).Contains(securityLevel1))
        return securityLevel1;
    SecurityLevel leastUpperBound = null;
    foreach (SecurityLevel securityLevel in securityLevel1.HigherSecurityLevels)
    {
        SecurityLevel buffer = getLeastUpperBound(securityLevel, securityLevel2);
        if (compare(leastUpperBound, buffer) != -1)
            leastUpperBound = buffer;
    }
    return leastUpperBound;
}

public int compare(SecurityLevel securityLevel1, SecurityLevel securityLevel2)
{
    if (securityLevel1 == null || securityLevel2 == null || securityLevel1 == securityLevel2)
        return 0;
    if (getLegalFlow(securityLevel1).Contains(securityLevel2))
        return -1;
    if (getLegalFlow(securityLevel2).Contains(securityLevel1))
        return 1;
    return 0;
}

public bool loadFlowPolicyFromFile(string policyFilename)
{
    if (policyFilename != null && System.IO.File.Exists(policyFilename))
    {
        try
        {
            XmlDocument flowPolicyFile = new XmlDocument();
            flowPolicyFile.Load(policyFilename);
        }
        catch (Exception ex)
        {
            // Handle exception
        }
    }
    return true;
}
XmlNodeReader reader = new XmlNodeReader(flowPolicyFile);
reader.Read();
reader.Read();
if (reader.LocalName.ToLower().Equals("flow"))
{
    securityLevels = new Hashtable();
    while (reader.ReadToFollowing("source"))
    {
        string levelName = reader.GetAttribute("level").ToLower();
        SecurityLevel sourceLevel = (SecurityLevel)securityLevels[levelName];
        // if source level does not exist yet
        if (!sourceLevel is SecurityLevel)
        {
            sourceLevel = new SecurityLevel(levelName);
            securityLevels[levelName] = sourceLevel;
        }
    }
    XmlReader destreader = reader.ReadSubtree();
    // for every destination entry in this source entry
    while (destreader.ReadToFollowing("destination"))
    {
        string destLevelName = destreader.GetAttribute("level").ToLower();
        // only if the dest level != source level we need to link them
        if (!destLevelName.Equals(sourceLevel.Name))
        {
            SecurityLevel destinationLevel = (SecurityLevel)securityLevels[destLevelName];
            // if the destination level does not exist yet, create it
            if (!destinationLevel is SecurityLevel)
            {
                destinationLevel = new SecurityLevel(destLevelName);
                securityLevels[destLevelName] = destinationLevel;
            }
            // add the destination level as higher level than source and vice versa
            if (!sourceLevel.HigherSecurityLevels.Contains(destinationLevel))
                sourceLevel.HigherSecurityLevels.Add(destinationLevel);
            if (!destinationLevel.LowerSecurityLevels.Contains(sourceLevel))
                destinationLevel.LowerSecurityLevels.Add(sourceLevel);
        }
    }
    return true;
}
else
{
    return false;
}
}
catch (XmlException)
{
    return false;
}
else
{
    return false;
}
calculateLowestSecurityLevel();
this.name = policyFilename;
return true;

/// <summary>
/// calculates the lowest security level in the flow policy
/// </summary>
private void calculateLowestSecurityLevel()
D.1.5 File AST/Assembly.cs

```
using System;
using System.Collections.Generic;
using System.IO;

namespace PNIC.Model.AST
{
    /// <summary>
    /// This class represents an assembly containing CIL code;
    /// It is the toplevel of the abstract syntax tree
    /// </summary>
    class Assembly
    {
        #region ctor

        public Assembly()
        {
        }
    }
}
```
public Assembly(string filename)
{
    this.loadAssembly(filename);
}
@endregion

#region fields and getters/setters

/// <summary>
/// The filename of this assembly
/// </summary>
private string name;
public string Name
{
    get { return this.name; }
}

/// <summary>
/// A list of all types in this assembly
/// </summary>
private List<Type> types;
public List<Type> Types
{
    get { return this.types; }
}
@endregion

#region methods

/// <summary>
/// loads a file; generating the list of types
/// </param>
public void loadAssembly(string filename)
{
    this.types = new List<Type>();
    try
    {
        System.Reflection.Assembly assembly = System.Reflection.Assembly.LoadFrom(filename);
        foreach (System.Type type in assembly.GetTypes())
        {
            Type newType = new Type(type);
            newType.ContainingAssembly = this;
            this.types.Add(newType);
        }
        this.name = filename;
    }
    catch (FileNotFoundException)
    {
        Console.WriteLine("Could not open file " + filename);
    }
    catch (BadImageFormatException)
    {
        Console.WriteLine(filename + " is not an assembly file");
    }
    catch (ArgumentException)
    {
        Console.Out.WriteLine("Invalid assemblyname: " + filename);
    }
}
/// <summary>
/// retrieves the type identified by its name;
/// returns null if type can not be found in this assembly
public Type getType(string typeName) {
    foreach (Type type in types) {
        // System.Console.WriteLine(type.Name);
        if (type.Name.Equals(typeName))
            return type;
    }
    return null;
}

public bool isValid() {
    if (name == null || this.Name == "")
        return false;
    return true;
}

class AssemblyLoadingException : Exception {
    public AssemblyLoadingException(string reason) {
        this.reason = reason;
    }

    private string reason;
    private string Reason {
        get { return reason; }
    }
}

D.1.6 File AST/Instruction.cs

using System;
using System.Collections.Generic;
using System.Text;
using PNIC.Controller;

namespace PNIC.Model.AST {
    /// <summary>
    /// abstract class representing an instruction of this language;
    /// lowest level of abstract syntax tree
    /// </summary>
    abstract class Instruction {
        #region ctors

        public Instruction() :
            this("default unnamed instruction")
        {
        }

        public Instruction(string name) {
            this.name = name;
            this.nextInstructions = new List<Instruction>();
        }

    }
}
```java
    this.previousInstructions = new List<Instruction>();
}

@endregion

@region fields and getters/setters

    /// <summary>
    /// Human readable name of the instruction
    /// </summary>
    protected string name;
    public string Name
    {
        get { return this.name; }
        set { this.name = value; }
    }

    /// <summary>
    /// position of the instruction in the instructionlist of containing method
    /// </summary>
    protected int position;
    public int Position
    {
        get { return position; }
        set { this.position = value; }
    }

    /// <summary>
    /// A human readable representation of this instruction and its properties
    /// </summary>
    public string ReadableInstruction
    {
        get {
            StringBuilder readable = new StringBuilder();
            readable.Append(position + " ");
            while (readable.Length < 5)
                readable.Append(" ");
            return readable.Append(name).ToString();
        }
    }

    /// <summary>
    /// A representation of the equivalence class used during analysis
    /// </summary>
    protected string equivalenceClass;
    public string ReadableEquivalenceClass
    {
        get { return equivalenceClass; }
    }

    /// <summary>
    /// a string identifying this instruction
    /// by combining names of assembly, type, method and instruction's position
    /// </summary>
    public string FullyQualifiedPosition
    {
        get {
            return containingMethod.ContainingType.ContainingAssembly.Name + "/" +
                containingMethod.ContainingType.Name + "/" +
                containingMethod.Name + position.ToString();
        }
    }

    /// <summary>
    /// Instructions that control can pass to after this instruction
    /// </summary>
    private List<Instruction> nextInstructions;
```
```java
public List<Instruction> NextInstructions
{
    get { return this.nextInstructions; }
    set { this.nextInstructions = value; }
}

/// <summary>
/// The instruction directly following in control flow graph
/// </summary>
private Instruction directSuccessor;

public Instruction DirectSuccessor
{
    set {
        directSuccessor = value;
        nextInstructions.Add(value);
    }
    get {
        return directSuccessor;
    }
}

/// <summary>
/// The instructions the control flow can come from to this instruction
/// </summary>
private List<Instruction> previousInstructions;

public List<Instruction> PreviousInstructions
{
    get { return this.previousInstructions; }
    set { this.previousInstructions = value; }
}

/// <summary>
/// The instruction directly preceeding this in control flow graph
/// </summary>
private Instruction directPredecessor;

public Instruction DirectPredecessor
{
    get {
        if (directPredecessor is Instruction)
            return this.directPredecessor;
        else
            return null;
    }
    set {
        this.previousInstructions.Add(value);
        this.directPredecessor = value;
    }
}

/// <summary>
/// The method this instruction is part of
/// </summary>
private Method containingMethod;

public Method ContainingMethod
{
    get { return this.containingMethod; }
    set { this.containingMethod = value; }
}

#endregion

#region methods

/// <summary>
/// Self parsing function for this instruction with help of the parser
/// </summary>
/// <param name="parser">The parser that preparsed this instruction</param>
```
D.1 Package Model

abstract public int parse (InstructionParser parser, int position);

abstract public void accept (CILInstructionVisitor visitor);

public virtual void link (InstructionParser parser)
{
    Instruction nextInstruction = null;
    int positionOfNextInstruction = this. position + 1;
    while (! (nextInstruction is Instruction))
    {
        nextInstruction = parser.ParsedInstructions[positionOfNextInstruction];
        positionOfNextInstruction++;
    }
    this. DirectSuccessor = nextInstruction;
    nextInstruction. DirectPredecessor = this;
}

D.1.7 File AST/CIL/CILInstructions.cs

using System;
using System.Collections;
using System.Text;
using System.Reflection;

namespace PNIC.Model.AST.CIL
{
    class CILInstructions
    {
        public static void registerInstructions (InstructionParser parser)
        {
            System. Type binaryType = (new BinaryInstruction()).GetType();
            System. Type cond1Type = (new ConditionalJump1Parameter()).GetType();
            System. Type cond2Type = (new ConditionalJump2Parameter()).GetType();
            System. Type loadType = (new Load()).GetType();
            System. Type popType = (new Pop()).GetType();
            System. Type pushType = (new Push()).GetType();
            System. Type returnType = (new Return()).GetType();
            System. Type storeType = (new Store()).GetType();
            System. Type unaryType = (new UnaryInstruction()).GetType();
            System. Type uncondType = (new UnconditionalJump()).GetType();
            System. Type nopType = (new Nop()).GetType();

            parser.InstructionTable .Add (0x00 , nopType);
            parser.InstructionTable .Add (0x02 , loadType);
            parser.InstructionTable .Add (0x03 , loadType);
            parser.InstructionTable .Add (0x04 , loadType);
            parser.InstructionTable .Add (0x05 , loadType);
            parser.InstructionTable .Add (0x06 , loadType);
        }
    }
}
D.1 Package Model

D SOURCE CODE OF PNIC

```c
parser/InstructionTable/Add(0x07, loadType);
parsers/InstructionTable/Add(0x08, loadType);
parsers/InstructionTable/Add(0x09, loadType);
parsers/InstructionTable/Add(0x0A, storeType);
parsers/InstructionTable/Add(0x0B, storeType);
parsers/InstructionTable/Add(0x0C, storeType);
parsers/InstructionTable/Add(0x0D, storeType);
parsers/InstructionTable/Add(0x10, storeType);
parsers/InstructionTable/Add(0x0E, loadType);
parsers/InstructionTable/Add(0x11, loadType);
parsers/InstructionTable/Add(0x13, storeType);
parsers/InstructionTable/Add(0x15, pushType);
parsers/InstructionTable/Add(0x16, pushType);
parsers/InstructionTable/Add(0x17, pushType);
parsers/InstructionTable/Add(0x18, pushType);
parsers/InstructionTable/Add(0x19, pushType);
parsers/InstructionTable/Add(0x1A, pushType);
parsers/InstructionTable/Add(0x1B, pushType);
parsers/InstructionTable/Add(0x1C, pushType);
parsers/InstructionTable/Add(0x1D, pushType);
parsers/InstructionTable/Add(0x1E, pushType);
parsers/InstructionTable/Add(0x1F, pushType);
parsers/InstructionTable/Add(0x20, pushType);
parsers/InstructionTable/Add(0x21, pushType);
parsers/InstructionTable/Add(0x22, pushType);
parsers/InstructionTable/Add(0x23, pushType);
parsers/InstructionTable/Add(0x26, popType);
parsers/InstructionTable/Add(0x2A, returnType);
parsers/InstructionTable/Add(0x2B, uncondType);
parsers/InstructionTable/Add(0x2C, cond1Type);
parsers/InstructionTable/Add(0x2D, cond1Type);
parsers/InstructionTable/Add(0x2E, cond2Type);
parsers/InstructionTable/Add(0x2F, cond2Type);
parsers/InstructionTable/Add(0x30, cond2Type);
parsers/InstructionTable/Add(0x31, cond2Type);
parsers/InstructionTable/Add(0x32, cond2Type);
parsers/InstructionTable/Add(0x33, cond2Type);
parsers/InstructionTable/Add(0x34, cond2Type);
parsers/InstructionTable/Add(0x35, cond2Type);
parsers/InstructionTable/Add(0x36, cond2Type);
parsers/InstructionTable/Add(0x37, cond2Type);
parsers/InstructionTable/Add(0x38, uncondType);
parsers/InstructionTable/Add(0x39, cond1Type);
parsers/InstructionTable/Add(0x3A, cond1Type);
parsers/InstructionTable/Add(0x3B, cond2Type);
parsers/InstructionTable/Add(0x3C, cond2Type);
parsers/InstructionTable/Add(0x3D, cond2Type);
parsers/InstructionTable/Add(0x3E, cond2Type);
parsers/InstructionTable/Add(0x3F, cond2Type);
parsers/InstructionTable/Add(0x40, cond2Type);
parsers/InstructionTable/Add(0x41, cond2Type);
parsers/InstructionTable/Add(0x42, cond2Type);
parsers/InstructionTable/Add(0x43, cond2Type);
parsers/InstructionTable/Add(0x44, cond2Type);
parsers/InstructionTable/Add(0x58, binaryType);
parsers/InstructionTable/Add(0x59, binaryType);
parsers/InstructionTable/Add(0x5A, binaryType);
parsers/InstructionTable/Add(0x5F, binaryType);
parsers/InstructionTable/Add(0x60, binaryType);
```

96
D.1.8 File AST/CIL/UnaryInstruction.cs

using System;
using System.Collections.Generic;
using System.Text;
using PNIC.Controller;

namespace PNIC.Model.AST.CIL
{
    public class UnaryInstruction : Instruction
    {
        public UnaryInstruction()
        {
            this.NextInstructions = new List<Instruction>();
        }

        public override int parse(InstructionParser parser, int position)
        {
            this.position = position;
            this.equivalenceClass = "unary_op";
            switch (parser.ILByteArray[position])
            {
                case 0x65:
                    this.Name = "Neg";
                    return 1;
                case 0x66:
                    this.Name = "Not";
                    return 1;
                default:
                    throw new InstructionParserException("Parsing error at " + parser.Method.Name + " position " + position + "; Not a valid unary instruction");
            }
        }

        public override void accept(CILintInstructionVisitor visitor)
        {
            visitor.visit(this);
        }
    }
}

D.1.9 File AST/CIL/BinaryInstruction.cs

using System;
using System.Collections.Generic;
using System.Text;
using PNIC.Controller;

namespace PNIC.Model.AST.CIL
{
  class BinaryInstruction : Instruction
  {
    public BinaryInstruction()
    {
      this.NextInstructions = new List<Instruction>();
    }

    public override int parse(InstructionParser parser, int position)
    {
      this.position = position;
      this.equivalenceClass = "binary op";
      switch (parser.I1ByteArray[position])
      {
        case 0x58:
          this.Name = "add";
          return 1;
        case 0x59:
          this.Name = "sub";
          return 1;
        case 0x5A:
          this.Name = "mul";
          return 1;
        case 0x5F:
          this.Name = "and";
          return 1;
        case 0x60:
          this.Name = "or";
          return 1;
        case 0x61:
          this.Name = "xor";
          return 1;
        case 0x62:
          this.Name = "shl";
          return 1;
        case 0x63:
          this.Name = "shr";
          return 1;
        case 0x64:
          this.Name = "shr.un";
          return 1;
        case 0xFE:
          switch (parser.I1ByteArray[position + 1])
          {
            case 0x01:
              this.Name = "Ceq";
              return 2;
            case 0x02:
              this.Name = "Cgt";
              return 2;
            case 0x03:
              this.Name = "Cgt.un";
              return 2;
            case 0x04:
              this.Name = "Clt";
              return 2;
            case 0x05:
              this.Name = "Clt.un";
              return 2;
          }
        throw new InstructionParserException("Parsing error at " + parser.Method.Name + " position " + position + "; Not a valid Load instruction");
      }
    }
  }
}
throw new InstructionParserException("Parsing error at " + parser.Method.Name + " position " + position + "; Not a valid Load instruction");
}

public override void accept(CILintInstructionVisitor visitor)
{
    visitor.visit(this);
}

D.1.10 File AST/CIL/Pop.cs

using System;
using System.Collections.Generic;
using System.Text;
using PNIC.Controller;

namespace PNIC.Model.AST.CIL
{
    class Pop : Instruction
    {
        public Pop()
        {
            this.NextInstructions = new List<Instruction>();
        }

        public override int parse(InstructionParser parser, int position)
        {
            this.position = position;
            this.Name = "pop";
            this.equivalenceClass = this.Name;
            return 1;
        }

        public override void accept(CILintInstructionVisitor visitor)
        {
            visitor.visit(this);
        }
    }
}

D.1.11 File AST/CIL/Push.cs

using System;
using System.Collections.Generic;
using System.Text;
using PNIC.Controller;

namespace PNIC.Model.AST.CIL
{
    class Push : Instruction
    {
        public Push()
        {
            this.NextInstructions = new List<Instruction>();
        }

        public override int parse(InstructionParser parser, int position)
        {
            this.position = position;
            this.equivalenceClass = "push v";
            switch (parser.IIByteArray[position])
            {
                case 0x15:
                    this.Name = "ldc.i4.m1";
                    return 1;
            }
        }
    }
}
D.1 Package Model

```csharp
D.1.12 File AST/CIL/Load.cs

```
#region ctor
public Load()
{
    this.NextInstructions = new List<Instruction>();
}
#endregion

#region fields and getters/setters
public new string ReadableEquivalenceClass
{
    get
    {
        StringBuilder readable = new StringBuilder();
        readable.Append(position + " : ");
        while (readable.Length < 5)
        {
            readable.Append(" ");
        }
        readable.Append(equivalenceClass);
        readable.Append(" "+target.Name);
        return readable.ToString();
    }
}
private Variable target;
public Variable Target
{
    get { return this.target; }
}
#endregion

#region methods
public override int parse(InstructionParser parser, int position)
{
    this.position = position;
    this.equivalenceClass = "load local";
    switch (parser.IILByteArray[position])
    {
        //region ldarg
        case 0x02:
            this.Name = "ldarg.0";
            this.target = parser.Method.Parameters[0];
            return 1;
        case 0x03:
            this.Name = "ldarg.1";
            this.target = parser.Method.Parameters[1];
            return 1;
        case 0x04:
            this.Name = "ldarg.2";
            this.target = parser.Method.Parameters[2];
            return 1;
        case 0x05:
            this.Name = "ldarg.3";
            this.target = parser.Method.Parameters[3];
            return 1;
        case 0x0E:
            this.target = parser.Method.Parameters[(byte)parser.IILByteArray[position + 1]];
            this.Name = "ldarg.8";
            return 2;
        //endregion

        //region ldloc
        case 0x06:
            this.Name = "ldloc.0";
            this.target = parser.Method.LocalVariables[0];
            this.target = parser.Method.Parameters[0];
            return 1;
        //endregion
    }
}
return 1;
case 0x07:
    this.Name = "ldloc.1";
    this.target = parser.Method.LocalVariables[1];
    return 1;
case 0x08:
    this.Name = "ldloc.2";
    this.target = parser.Method.LocalVariables[2];
    return 1;
case 0x09:
    this.Name = "ldloc.3";
    this.target = parser.Method.LocalVariables[3];
    return 1;
case 0x11:
    this.target = parser.Method.LocalVariables[(byte)parser.IlByteArray[position + 1]];
    this.Name = "ldloc.s";
    return 2;

#endregion

case 0xFE:
    if ((parser.IlByteArray[position + 1] == 0x0C)
    { 
        this.target = parser.Method.LocalVariables[(Int16)parser.IlByteArray[position + 2]];
        this.Name = "ldloc";
        return 4;
    } 
    if ((parser.IlByteArray[position + 1] == 0x09)
    { 
        this.target = parser.Method.Parameters[(Int16)parser(IlByteArray[position + 2]];
        this.Name = "ldarg";
        return 4;
        throw new InstructionParserException("Parsing error at " + parser.Method.Name + " position " + position + "; Not a valid Load instruction");
    } 
    throw new InstructionParserException("Parsing error at " + parser.Method.Name + " position " + position + "; Not a valid Load instruction");
}

public override void accept(CILInstructionVisitor visitor)
{
    visitor.visit(this);
}

#endregion

D.1.13 File AST/CIL/Store.cs

using System;
using System.Collections.Generic;
using System.Text;
using PNIC.Controller;

namespace PNIC.Model.AST.CIL {

class Store : Instruction 
{
    public Store()
    {
        this.NextInstructions = new List<Instruction>();
    }

    private Variable target;
}
public Variable Target
{
    get { return this.target; }
}

public new string ReadableEquivalenceClass
{
    get {
        StringBuilder readable = new StringBuilder();
        readable.Append(position + ":");
        while (readable.Length < 5)
            readable.Append(" ");
        readable.Append(equivalenceClass);
        readable.Append(" + target.Name");
        return readable.ToString();
    }
}

public override int parse(InstructionParser parser, int position)
{
    this.position = position;
    this.equivalenceClass = "storelocal";
    switch (parser.IlByteArray[position])
    {
        case 0x0A:
            this.Name="stloc.0";
            this.target = parser.Method.LocalVariables[0];
            return 1;
        case 0x0B:
            this.Name="stloc.1";
            this.target = parser.Method.LocalVariables[1];
            return 1;
        case 0x0C:
            this.Name="stloc.2";
            this.target = parser.Method.LocalVariables[2];
            return 1;
        case 0x0D:
            this.Name="stloc.3";
            this.target = parser.Method.LocalVariables[3];
            return 1;
        case 0x13:
            this.target = parser.Method.LocalVariables[(byte)parser.IlByteArray[position + 1]];
            this.Name = "stloc.s";
            return 2;
        case 0x10:
            this.target = parser.Method.Parameters[(byte)parser.IlByteArray[position + 1]];
            this.Name = "starg.s";
            return 2;
        case 0xFE:
            if (parser.IlByteArray[position + 1] == 0x0B)
            {
                this.target = parser.Method.Parameters[(Int16)parser.IlByteArray[position + 2]];
                this.Name = "starg";
                return 4;
            }
            if (parser.IlByteArray[position + 1] == 0x0E)
            {
                this.target = parser.Method.LocalVariables[(Int16)parser.IlByteArray[position + 2]];
                this.Name = "stloc";
                return 4;
            }
            throw new InstructionParserException("Parsing error at " + parser.Method.Name + " position " + position + "; Not a valid Store instruction;");
    }
}
throw new InstructionParserException("Parsing error at "+ parser.Method.Name + " 
position "+ position + "; Not a valid Store instruction;");
}

public override void accept(CILInstructionVisitor visitor)
{
    visitor.visit(this);
}

D.1.14 File AST/CIL/UnconditionalJump.cs

using System;
using System.Collections.Generic;
using System.Text;
using PNIC.Controller;

namespace PNIC.Model.AST.CIL
{
class UnconditionalJump : Instruction
{
    public UnconditionalJump()
    {
        this.NextInstructions = new List<Instruction>();
    }

    public new string ReadableInstruction
    {
        get
        {
            StringBuilder readable = new StringBuilder();
            readable.Append(position + " ");
            while (readable.Length < 5)
                readable.Append(" ");
            readable.Append(Name);
            if (DirectSuccessor != null)
                readable.Append(" "+ DirectSuccessor.Position);
            return readable.ToString();
        }
    }

    public new string ReadableEquivalenceClass
    {
        get
        {
            StringBuilder readable = new StringBuilder();
            readable.Append(position + " ");
            while (readable.Length < 5)
                readable.Append(" ");
            readable.Append(equivalenceClass);
            if (DirectSuccessor != null)
                readable.Append(" "+ DirectSuccessor.Position);
            return readable.ToString();
        }
    }

    private int positionOfNextInstruction;

    public override int parse(InstructionParser parser, int position)
    {
        this.position = position;
        this.equivalenceClass = "unconditionalJump";
        if (parser.I1ByteArray[position] == 0x2B)
        {
            positionOfNextInstruction = position + 2 + (sbyte)parser.I1ByteArray[position + 1];
            this.Name = "br.int8";
        }
    }
}
D.1 Package Model

D SOURCE CODE OF PNIC

```csharp
return 2;
}
if (parser.ilByteArray[position] == 0x38)
{
    positionOfNextInstruction = position + 5 + (int)parser.ilByteArray[position + 1];
    this.Name = "br.int32";
    return 5;
}
throw new InstructionParserException("Parsing error at " + parser.Method.Name + " position " + position + "; Not a valid unconditional Jump instruction");
}
public override void accept(CILInstructionVisitor visitor)
{
    visitor.visit(this);
}
public override void link(InstructionParser parser)
{
    Instruction nextInstruction = parser.ParsedInstructions[positionOfNextInstruction];
    NextInstructions.Add(nextInstruction);
    DirectSuccessor = nextInstruction;
    DirectSuccessor.PreviousInstructions.Add(this);
}
}
```

D.1.15 File AST/CIL/ConditionalJump.cs

```csharp
using System;
using System.Collections.Generic;
using System.Text;
using PNIC.Controller;

namespace PNIC.Model.AST.CIL
{
    abstract class ConditionalJump : Instruction
    {
        public new string ReadableInstruction
        {
            get
            {
                StringBuilder readable = new StringBuilder();
                readable.Append(position + ": ");
                while (readable.Length < 5)
                {
                    readable.Append(" ");
                    readable.Append(Name).ToString();
                    readable.Append(" "+jumpInstruction.Position);
                    return readable.ToString();
                }
            }
        }
        public new string ReadableEquivalenceClass
        {
            get
            {
                StringBuilder readable = new StringBuilder();
                readable.Append(position + ": ");
                while (readable.Length < 5)
                {
                    readable.Append(" ");
                    readable.Append(equivalenceClass);
                    readable.Append(" "+jumpInstruction.Position);
                    return readable.ToString();
                }
            }
        }
        protected Instruction nextInstruction;
        public Instruction NextInstruction
    }
```
protected int branchingTarget;
protected Instruction jumpInstruction;
public Instruction JumpInstruction
{
    set
    {
        jumpInstruction = value;
        NextInstructions.Add(value);
    }
    get
    {
        return this.jmpInstruction;
    }
}

public override void link(InstructionParser parser)
{
    int positionOfNextInstruction = this.position + 1;
    while ( !(this.nextInstruction is Instruction) )
    {
        this.nextInstruction = parser.ParsedInstructions[positionOfNextInstruction];
        positionOfNextInstruction++;
    }
    this.DirectSuccessor = nextInstruction;
    nextInstruction.DirectPredecessor = this;
    JumpInstruction = parser.ParsedInstructions[branchingTarget];
    jumpInstruction.PreviousInstructions.Add(this);
}

class ConditionalJump2Parameters : ConditionalJump
{
    public ConditionalJump2Parameters()
    {
        this.NextInstructions = new List<Instruction>();
    }
    public override int parse(InstructionParser parser, int position)
    {
        this.position = position;
        this.equivalenceClass = "jumpWith2Arguments";
        switch ( parser.IlByteArray[position] )
        {
            case 0x3B:
                branchingTarget = position + 5 + (int)parser.IlByteArray[position + 1];
                this.Name = "beq";
                return 5;
            case 0x2E:
                branchingTarget = position + 2 + (sbyte)parser.IlByteArray[position + 1];
                this.Name = "beq.s";
                return 2;
            case 0x3C:
                branchingTarget = position + 5 + (int)parser.IlByteArray[position + 1];
                this.Name = "bge";
                return 5;
            case 0x3F:
                branchingTarget = position + 2 + (sbyte)parser.IlByteArray[position + 1];
                this.Name = "bge.s";
                return 2;
            case 0x41:
                branchingTarget = position + 5 + (int)parser.IlByteArray[position];
                this.Name = "bge.un";
                return 5;
            case 0x44:
                branchingTarget = position + 2 + (sbyte)parser.IlByteArray[position + 1];
                this.Name = "bge.un.s";
        }
    }
}
D.1 Package Model

```java
return 2;
case 0x3D:
    branchingTarget = position + 5 + (int)parser.ILByteArray[position + 1];
    this.Name = "bgt";
    return 5;
case 0x30:
    branchingTarget = position + 2 + (sbyte)parser.ILByteArray[position + 1];
    this.Name = "bgt.s";
    return 2;
case 0x32:
    branchingTarget = position + 2 + (sbyte)parser.ILByteArray[position + 1];
    this.Name = "bbl.s";
    return 2;
case 0x42:
    branchingTarget = position + 5 + (int)parser.ILByteArray[position + 1];
    this.Name = "bgt.un";
    return 5;
case 0x35:
    branchingTarget = position + 2 + (sbyte)parser.ILByteArray[position + 1];
    this.Name = "bgt.un.s";
    return 2;
case 0x3E:
    branchingTarget = position + 5 + (int)parser.ILByteArray[position + 1];
    this.Name = "ble";
    return 5;
case 0x31:
    branchingTarget = position + 2 + (sbyte)parser.ILByteArray[position + 1];
    this.Name = "ble.s";
    return 2;
case 0x3E:
    branchingTarget = position + 5 + (int)parser.ILByteArray[position + 1];
    this.Name = "ble.un";
    return 5;
case 0x36:
    branchingTarget = position + 2 + (sbyte)parser.ILByteArray[position + 1];
    this.Name = "ble.un.s";
    return 2;
case 0x43:
    branchingTarget = position + 5 + (int)parser.ILByteArray[position + 1];
    this.Name = "ble.un";
    return 5;
case 0x37:
    branchingTarget = position + 2 + (sbyte)parser.ILByteArray[position + 1];
    this.Name = "ble.un.s";
    return 2;
case 0x44:
    branchingTarget = position + 5 + (int)parser.ILByteArray[position + 1];
    this.Name = "blt";
    return 5;
case 0x3F:
    branchingTarget = position + 2 + (sbyte)parser.ILByteArray[position + 1];
    this.Name = "blt.s";
    return 2;
case 0x32:
    branchingTarget = position + 2 + (sbyte)parser.ILByteArray[position + 1];
    this.Name = "blt.un";
    return 5;
case 0x40:
    branchingTarget = position + 5 + (int)parser.ILByteArray[position + 1];
    this.Name = "blt.un.s";
    return 5;
case 0x33:
    branchingTarget = position + 2 + (sbyte)parser.ILByteArray[position + 1];
    this.Name = "blt.un.s";
    return 2;
default:
    throw new InstructionParserException("Parsing error at " + parser.Method.Name + " position " + position + "; Not a valid ConditionalJump instruction");
}
}

public override void accept(CILIntInstructionVisitor visitor)
{
    visitor.visit(this);
}
```
class ConditionalJump1Parameter : ConditionalJump
{
    public ConditionalJump1Parameter()
    {
        this.NextInstructions = new List<Instruction>();
    }

    public override int parse(InstructionParser parser, int position)
    {
        this.position = position;
        this.equivalenceClass = "jumpWith1Argument";
        switch (parser.IILByteArray[position])
        {
            case 0x39:
                branchingTarget = position + 5 + (int)parser.IILByteArray[position + 1];
                this.Name = "brfalse";
                return 5;
            case 0x2C:
                branchingTarget = position + 2 + (sbyte)parser.IILByteArray[position + 1];
                this.Name = "brfalse.s";
                return 2;
            case 0x3A:
                branchingTarget = position + 5 + (int)parser.IILByteArray[position + 1];
                this.Name = "brtrue";
                return 5;
            case 0x2D:
                branchingTarget = position + 2 + (sbyte)parser.IILByteArray[position + 1];
                this.Name = "brtrue.s";
                return 2;
            default:
                throw new InstructionParserException("Parsing error at " + parser.Method.Name + " position " + position + "; Not a valid ConditionalJump instruction");
        }
    }

    public override void accept(CILInstructionVisitor visitor)
    {
        visitor.visit(this);
    }
}

D.1.16 File AST/CIL/Return.cs

using System;
using System.Collections.Generic;
using System.Text;
using PNIC.Controller;

namespace PNIC.Model.AST.CIL
{
    class Return : Instruction
    {
        public Return()
        {
            this.NextInstructions = new List<Instruction>();
        }

        #region methods

        public override int parse(InstructionParser parser, int position)
        {
            this.position = position;
            this.Name = "return";
            this.equivalenceClass = this.Name;
            return 1;
        }
    }
}
public override void accept(CILintInstructionVisitor visitor) {
    visitor.visit(this);
}

public override void link(InstructionParser parser) {
    this.NextInstructions.Add(null);
}

namespace PNIC.Model.AST
{
    /// <summary>
    /// a simple bytecode array parser;
    /// instructions must be registered to be preparseable
    /// </summary>
    class InstructionParser
    {
        #region ctor

        public InstructionParser()
        {
            this.instructionTable = new Hashtable();
            this.extendedInstructionTable = new Hashtable();
        }

        #endregion

        #region fields and getters/setters

        /// <summary>
        /// Mappings from instructions/bytecodes to constructors for the corresponding AST representation
        /// </summary>
        private Hashtable instructionTable;
        public Hashtable InstructionTable
        {
            get { return instructionTable; }
            set { this.instructionTable = value; }
        }

        private Hashtable extendedInstructionTable;
        public Hashtable ExtendedInstructionTable
        {
            get { return extendedInstructionTable; }
            set { this.extendedInstructionTable = value; }
        }

        /// <summary>
        /// the bytecode array to be parsed
        /// </summary>
        private byte[] ilByteArray;
        public byte[] ilByteArray
        {
            get { return ilByteArray; }
        }
    }
}

D.1.17 File AST/InstructionParser.cs

using System;
using System.Collections.Generic;
using System.Text;
using System.Collections;
using System.Reflection;

namespace PNIC.Model.AST
{
    public InstructionParser()
    {
        this.instructionTable = new Hashtable();
        this.extendedInstructionTable = new Hashtable();
    }

    public InstructionTable
    {
        get { return instructionTable; }
        set { this.instructionTable = value; }
    }

    public ExtendedInstructionTable
    {
        get { return extendedInstructionTable; }
        set { this.extendedInstructionTable = value; }
    }

    public byte[] ilByteArray
    {
        get { return ilByteArray; }
    }
}
private Instruction[] parsedInstructions;

public Instruction[] ParsedInstructions
{
    get { return this.parsedInstructions; }
}

private Method method;

public Method Method
{
    get { return method; }
}

private int position;

public int Position
{
    get { return this.position; }
}

public Method parseInstructions(Method method, byte[] ilByteArray)
{
    // set the method in progress, the corresponding byte array and the initial position
    this.method = method;
    this.ilByteArray = ilByteArray;
    this.position = 0;
    // create an array for already parsed instructions
    parsedInstructions = new Instruction[ilByteArray.Length];

    // go through the byte array and parse each instruction
    while (this.position < ilByteArray.Length)
    {
        // parse the instruction at current position and store it in method instructions
        method.Instructions.Add(parseInstruction());
    }

    // link instructions to create the control flow graph
    // and relabel positions with program points
    int count = 0;
    foreach (Instruction instruction in method.Instructions)
    {
        instruction.link(this);
        instruction.ContainingMethod = method;
        instruction.Position = count;
        count++;
    }
    method.FirstInstruction = method.Instructions[0];
return method;
}

/// <summary>
/// parses the instruction at the current position of this parser
/// </summary>
/// <returns>returns an Instruction object representing the parsed instruction</returns>
private Instruction parseInstruction()
{
    // check if the position is inside the array to parse
    if (this.position > ilByteArray.Length)
        throw new InstructionParserException("Position to parse out of array range");

    // if the instruction is parsed already, just return it
    if (parsedInstructions[position] is Instruction)
    {
        return parsedInstructions[position];
    }

    // else parse the instruction
    else
    {
        // determine the type of the instruction via the instruction table
        System.Type instructionType = (System.Type)extendsInstructionTable[(int)((byte)ilByteArray[position + 1])];
        // Console.WriteLine("2-Byte-Bytecode parsed: " + (int)((byte)ilByteArray[position]) + 256 + (byte)ilByteArray[position + 1]);

        if (instructionType == null)
            throw new InstructionParserException("Instruction not part of the language of this parser in method "+ method.Name + " at position "+ position + " bytecode: "+ ilByteArray[position]);

        // create preparse instruction
        Instruction instruction = (Instruction)Activator.CreateInstance(instructionType);

        // if instruction got preparsed successfully
        if (instruction != null)
        {
            // store instruction as parsed
            parsedInstructions[position] = instruction;

            // finally let the instruction parse itself and update position to next instruction
            this.position = this.position + instruction.parse(this, position);
        }
        else
            throw new InstructionParserException("Failed to parse instruction at position "+ position + " in method "+ method.Name);

        return instruction;
    }

@endregion

class InstructionParserException : Exception
```csharp
private string reason;
public string Reason
{
    get { return this.reason; }
}

public InstructionParserException(string reason)
{
    this.reason = reason;
}

namespace PNIC.Model.AST
{
    /// <summary>
    /// represents an il-method;
    /// one level above instructions in the abstract syntax tree
    /// </summary>
    class Method
    {
        #region ctor
        public Method(MethodInfo methodInfo)
        {
            StringBuilder nameBuilder = new StringBuilder();
            nameBuilder.Append(methodInfo.Name + "(");
            foreach (ParameterInfo parameterInfo in methodInfo.GetParameters())
            {
                nameBuilder.Append(parameterInfo.ParameterType.ToString() +", ");
            }
            if (nameBuilder.ToString().EndsWith(" "))
            {
                nameBuilder.Remove(nameBuilder.Length - 2, 2);
                this.name = nameBuilder.ToString();
            }
            loadingException = null;
        }
        #endregion

        #region fields getters/setters

        /// <summary>
        /// name identifying this method
        /// </summary>
        private string name;
        public string Name
        {
            get { return name; }
        }
        public string Readable
        {
            get { return name + ": could not parse instructions"; }
        }
    }

    using System;
    using System.Collections.Generic;
    using System.Text;
    using System.Reflection;
    using PNIC.Model.AST.CIL;

    File AST/Method.cs
```
private Type containingType;
public Type ContainingType
{
    get { return containingType; }
    set { this.containingType = value; }
}

/// <summary>
/// The parameters of this method
/// </summary>
private Variable[] parameters;
public Variable[] Parameters
{
    get { return parameters; }
    set { this.parameters = value; }
}

/// <summary>
/// The local variables of this method
/// </summary>
private Variable[] localVariables;
public Variable[] LocalVariables
{
    get { return localVariables; }
    set { this.localVariables = value; }
}

/// <summary>
/// The first instruction of the method body
/// Can be used to traverse the whole method body via control flow graph
/// </summary>
private Instruction firstInstruction;
public Instruction FirstInstruction
{
    get { return firstInstruction; }
    set { this.firstInstruction = value; }
}

private List<Instruction> instructions;
public List<Instruction> Instructions
{
    get { return instructions; }
    set { this.instructions = value; }
}

public List<ConditionalJump> ConditionalJumps
{
    get
    {
        List<ConditionalJump> conditionalJumps = new List<ConditionalJump>();
        foreach (Instruction instruction in instructions)
        {
            if (instruction is ConditionalJump)
            {
                conditionalJumps.Add((ConditionalJump)instruction);
            }
        }
        return conditionalJumps;
    }
}

private InstructionParserException loadingException;
public InstructionParserException LoadingException
{
    get { return this.loadingException; }
}

@endregion
package model

private void loadMethod(MethodInfo methodInfo)
{
    // update its parameters
    this.Parameters = new Variable[methodInfo.GetParameters().Length + 1];
    //this.Parameters = new Variable[100];
    for (int i = 0; i < Parameters.Length; i++)
    {
        this.Parameters[i] = new Variable();
        this.Parameters[i].ContainingMethod = this;
        this.Parameters[i].Name = "arg_" + i.ToString();
    }

    // update its local variables
    int variableCount = methodInfo.GetMethodBody().LocalVariables.Count;
    this.LocalVariables = new Variable[variableCount];
    for (int i = 0; i < LocalVariables.Length; i++)
    {
        this.LocalVariables[i] = new Variable();
        this.LocalVariables[i].Name = "var_" + i.ToString();
        this.LocalVariables[i].ContainingMethod = this;
    }

    this.instructions = new List<Instruction>();
    // initialize parser with the sublanguage to use
    InstructionParser parser = new InstructionParser();
    CIL.CILintInstructions.registerInstructions(parser);
    try
    {
        parser.parseInstructions(this, methodInfo.GetMethodBody().GetILAsByteArray());
    }
    catch (InstructionParserException ex)
    {
        this.loadingException = ex;
        Console.Out.WriteLine(ex.Reason);
        Console.Out.WriteLine("Press enter to continue");
        Console.In.ReadLine();
    }

    // retrieve the instruction at given position
}

public Instruction getInstructionAtPosition(int position)
{
    if (position < instructions.Count)
        return instructions[position];
    else
        return null;
}

class MethodLoadingException : Exception
{
public MethodLoadingException(string reason)
{
    this.reason = reason;
}

private string reason;
public string Reason
{
    get { return reason; }
}

D.1.19 File AST/Type.cs

using System;
using System.Collections.Generic;
using System.Text;
using System.Reflection;

namespace PNIC.Model.AST
{
    /// <summary>
    /// represents a type in an assembly;
    /// one level beneath assembly in the abstract syntax tree
    /// </summary>
    class Type
    {
        #region ctor

        public Type(System.Type type)
        {
            this.name = type.Name;
            loadType(type);
        }
        #endregion

        #region fields and getters/setters

        /// <summary>
        /// a string identifying this type
        /// </summary>
        private string name;
        public string Name
        {
            get { return this.name; }
        }

        /// <summary>
        /// the assembly that inherits this type
        /// </summary>
        private Assembly containingAssembly;
        public Assembly ContainingAssembly
        {
            get { return containingAssembly; }
            set { this.containingAssembly = value; }
        }

        /// <summary>
        /// the flags used to load this type
        /// </summary>

        /// <summary>
        /// a list of all membermethods in this type
        /// </summary>
        private List<Method> memberMethods;
```csharp
public List<Method> MemberMethods {
    get { return memberMethods; }
}

#region methods

/// <summary>
/// loads the type represented by the System.Type
/// <param name="type">the System.Type described by this type</param>
/// </summary>
public void loadType(System.Type type) {
    this.memberMethods = new List<Method>();
    //try to load each method of this type and add it to the memberMethods list
    foreach (MethodInfo methodInfo in type.GetMethods(this.customFlags)) {
        try {
            Method newMethod = new Method(methodInfo);
            newMethod.ContainingType = this;
            this.memberMethods.Add(newMethod);
        } catch (MethodLoadingException ex) {
            Console.Error.WriteLine(ex.Reason);
        }
    }
}

/// <summary>
/// returns the member identified by its name;
/// null if member not found
/// <param name="methodName">the name of the instruction as string</param>
/// <returns>Method object in this type identified by name</returns>
public Method getMemberMethod(string methodName) {
    foreach (Method method in memberMethods) {
        if (method.Name.Equals(methodName))
            return method;
    }
    return null;
}
#endregion

D.1.20 File AST/Variable.cs

using System;
using System.Collections.Generic;

namespace PNIC.Model.AST {
    /// <summary>
    /// represents a variable or parameter in an il method
    /// </summary>
    class Variable {
        #region fields and getter/setter
```
/// <summary>
/// a human readable identifier for this variable
/// </summary>
private string name;
public string Name
{
    get { return name; }
    set { this.name = value; }
}

/// <summary>
/// the method that inherits this variable
/// </summary>
private Method containingMethod;
public Method ContainingMethod
{
    get { return containingMethod; }
    set { this.containingMethod = value; }
}

@endregion

D.2 Package Controller

D.2.1 File Visitor.cs

using PNIC.Model.AST.CIL;
using PNIC.Model.AST;

namespace PNIC.Controller
{
    /// <summary>
    /// abstract class for visiting instructions of the CIL instruction set
    /// </summary>
    abstract class CILInstructionVisitor
    {
        abstract public void visit(BinaryInstruction instruction);
        abstract public void visit(ConditionalJump1Parameter instruction);
        abstract public void visit(ConditionalJump2Parameters instruction);
        abstract public void visit(Pop instruction);
        abstract public void visit(Push instruction);
        abstract public void visit(Return instruction);
        abstract public void visit(Store instruction);
        abstract public void visit(UnaryInstruction instruction);
        abstract public void visit(UnconditionalJump instruction);
        abstract public void visit(Nop instruction);
    }

    /// <summary>
    /// abstract class for visiting containers
    /// </summary>
    abstract class InstructionContainerVisitor
    {
        abstract public void visit(Method method);
        abstract public void visit(Type type);
        abstract public void visit(Assembly assembly);
    }
}

D.2.2 File Analysis/Analysis.cs

using System.Collections.Generic;
using System.Text;
namespace PNIC.Controller.Analysis
{
    /// <summary>
    /// A wrapper class that is intended for use in programs
    /// that directly present the result of the analysis.
    /// </summary>
    class InteractiveAnalysis
    {
        /// <summary>
        /// Settings to use in this analysis
        /// </summary>
        private AnalysisSettings configuration;
        public AnalysisSettings AnalysisSettings
        {
            get { return configuration; }
            set { this.configuration = value; }
        }

        /// <summary>
        /// Flow policy to use in this analysis
        /// </summary>
        private FlowPolicy flowPolicy;
        public FlowPolicy FlowPolicy
        {
            get { return flowPolicy; }
            set { this.flowPolicy = value; }
        }

        /// run an analysis
        /// <returns>string representing the result of the analysis </returns>
        public string runAnalysis()
        {
            StringBuilder summary = new StringBuilder();
            if (configuration is AnalysisSettings && flowPolicy is FlowPolicy)
            {
                foreach (string assemblyname in configuration.getAssembliesToCheck())
                {
                    System.Console.Out.WriteLine("Analysing assembly: "+ assemblyname);
                    summary.AppendLine("Analysing assembly: "+ assemblyname);
                    Assembly assembly = new Assembly(assemblyname);
                    foreach (string typename in configuration.getTypesToCheck(assemblyname))
                    {
                        System.Console.Out.WriteLine(" Analysing type: "+ typename);
                        summary.AppendLine(" Analysing type: "+ typename);
                        Type type = assembly.GetType(typename);
                        if (type != null)
                        {
                            foreach (string methodname in configuration.getMethodsToCheck(assemblyname, typename))
                            {
                                string result = null;
                                Method method = type.GetMemberMethod(methodname);
                                if (method != null & & checkFlowPolicyCompatibility(method))
                                {
                                    if (method.LoadException == null)
                                    {
                                        MethodTransformer transformer = new MethodTransformer();
                                        transformer.Configuration = configuration;
                                        transformer.FlowPolicy = flowPolicy;
                                    }
                                }
                            }
                        }
                    }
                }
            }
        }
    }
}
try {
    transform.visit(method);
    result = method.Name + " is non-interfering";
    summary.AppendLine("−" + result);
    System.Console.Out.WriteLine("−" + result);
}
catch (AnalysisException ex) {
    result = ex.Reason;
    summary.AppendLine("−" + result);
    System.Console.Out.WriteLine("−" + result);
    CompletePrint printer = new CompletePrint();
    printer.EquivalenceClass = true;
    printer.Configuration = configuration;
    printer.visit(method);
}
else {
    System.Console.Out.WriteLine("−" + method>LoadingException.Reason);
    summary.AppendLine("−" + method>LoadingException.Reason);
}
else {
    System.Console.Out.WriteLine("→" + method not found or analysed");
    summary.AppendLine("→" + method not found or analysed");
}
System.Console.Out.WriteLine();
}
}
else {
    System.Console.Out.WriteLine("→" + type not found");
    summary.AppendLine("→" + type not found");
}
}
return summary.ToString();
}
return "Flow policy or analysis settings not valid!";

private bool checkFlowPolicyCompatibility(Method method) {
    // check if every security level assigned to a variable exists in the flow policy
    foreach (Variable parameter in method.Parameters) {
        if (flowPolicy.getSecurityLevel(configuration.getSecurityLevel(parameter)) == null)
        {
            System.Console.WriteLine("−" + configuration.getSecurityLevel(parameter) + " is not part of the flow policy");
            return false;
        }
    }
    foreach (Variable variable in method.LocalVariables) {
        
    }
}
if (flowPolicy.getSecurityLevel(configuration.getSecurityLevel(variable)) == null)
{
    System.Console.WriteLine("->" + configuration.getSecurityLevel(variable) + " is not part of the flow policy");
    return false;
}
return true;
}

class AnalysisException : System.Exception
{
    public AnalysisException(string reason)
    {
        this.reason = reason;
    }

    private string reason;
    public string Reason
    {
        get { return reason; }
    }
}

namespace PNIC.Controller.Analysis
{
    /// <summary>
    /// class representing abstract transformations for CILint
    /// </summary>
    class CILintTransformer : CILintInstructionVisitor
    {
        #region ctor
        public CILintTransformer()
        {
            state = new AbstractState();
            position = 0;
        }
        #endregion

        #region fields and getters/setters

        private int position;
        public int Position
        {
            get { return this.position; }
        }

        private AnalysisSettings configuration;
        public AnalysisSettings Configuration
        {
            get { return configuration; }
            set { this.configuration = value; }
        }

        private FlowPolicy flowPolicy;
        public FlowPolicy FlowPolicy
        {get; set;}

        #endregion
    }

D.2.3 File Analysis/CILintTransformer.cs

using System;
using System.Collections.Generic;
using System.Text;
using PNIC.Model.Analysis;
namespace PNIC.Controller.Analysis
{
D.2 Package Controller

```csharp
[72x753]D.2 Package Controller D SOURC CODE OF PNIC

private AbstractState state;
public AbstractState State
{
    get { return state; }
    set { this.state = value; }
}

public override void visit (PNIC.Model.AST.CIL.BinaryInstruction instruction)
{
    checkSecurityLevelStackUniformity (instruction);
    SecurityLevel securityEnvironment = flowPolicy.getSecurityLevel (configuration, getSecurityEnvironment (instruction));
    SecurityLevel topOfStackLevel;
    SecurityLevel secondOfStackLevel;
    try
    {
        topOfStackLevel = state.RecentSecurityLevelStack.Pop();
        secondOfStackLevel = state.RecentSecurityLevelStack.Pop();
    }
    catch (InvalidOperationException)
    {
        throw new AnalysisException (instruction.Position + " too few values left on stack ");
    }
    SecurityLevel leastUpperBound = flowPolicy.getLeastUpperBound (topOfStackLevel, secondOfStackLevel);
    if (leastUpperBound == null)
        throw new AnalysisException (instruction.Position + " no least upper bound found for " + topOfStackLevel.Name + " and " + secondOfStackLevel.Name);
    leastUpperBound = flowPolicy.getLeastUpperBound (leastUpperBound, securityEnvironment);
    if (leastUpperBound == null)
        throw new AnalysisException (instruction.Position + " no least upper bound found for " + securityEnvironment.Name + " and " + leastUpperBound.Name);
    state.RecentSecurityLevelStack.Push (leastUpperBound);
    state.setSecurityLevelStack (instruction, state.RecentSecurityLevelStack);
}

public override void visit (PNIC.Model.AST.CIL.ConditionalJump1Parameter instruction)
{
    checkSecurityLevelStackUniformity (instruction);
    throw new NotImplementedException();
}

public override void visit (PNIC.Model.AST.CIL.ConditionalJump2Parameters instruction)
{
    checkSecurityLevelStackUniformity (instruction);
    SecurityLevel topOfStackLevel;
    SecurityLevel secondOfStackLevel;
}
```

121
try
{
    topOfStackLevel = state.RecentSecurityLevelStack.Pop();
    secondOfStackLevel = state.RecentSecurityLevelStack.Pop();
}
catch (InvalidOperationException)
{
    throw new AnalysisException(instruction.Position + " too few values left on stack ");
}

SecurityLevel leastUpperBound = flowPolicy.getLeastUpperBound(topOfStackLevel,
    secondOfStackLevel);

if (leastUpperBound == null)
    throw new AnalysisException(instruction.Position + " no least upper bound found
    for "+topOfStackLevel.Name + " and " + secondOfStackLevel.Name);

// lift stack
int stacksize = state.RecentSecurityLevelStack.Count;
Stack<SecurityLevel> buffer = new Stack<SecurityLevel>();

for (int i = 0; i < stacksize; i++)
{
    SecurityLevel stackLevel = state.RecentSecurityLevelStack.Pop();
    SecurityLevel liftedLevel = flowPolicy.getLeastUpperBound(leastUpperBound,
        stackLevel);
    if (liftedLevel == null)
        throw new AnalysisException(instruction.Position + " no least upper bound found
        for "+leastUpperBound.Name + " and " + stackLevel.Name);
    buffer.Push(liftedLevel);
}

for (int i = 0; i < stacksize; i++)
{
    state.RecentSecurityLevelStack.Push(buffer.Pop());
}

// lift region
ControlDependencyRegion region = configuration.getControlDependencyRegion(
    instruction);
if (region == null)
    throw new AnalysisException(instruction.Position + " no controldependencyregion
    found for conditional jump");

string key = instruction.ContainingMethod.ContainingType.ContainingAssembly.Name +
    "/" +
    instruction.ContainingMethod.ContainingType.Name + "/" +
    instruction.ContainingMethod.Name;

foreach (int position in region.Region)
{
    SecurityLevel environmentLevel = flowPolicy.getSecurityLevel(
        configuration.getSecurityEnvironment(key + position.ToString()));
    SecurityLevel liftedLevel = flowPolicy.getLeastUpperBound(leastUpperBound,
        environmentLevel);
    if (liftedLevel == null)
        throw new AnalysisException(instruction.Position + " no least upper bound found
        for "+environmentLevel.Name + " and " + leastUpperBound.Name);
    configuration.setSecurityEnvironment(key + position.ToString(), liftedLevel);
}

state.setSecurityLevelStack(instruction, state.RecentSecurityLevelStack);
public override void visit(PNIC.Model.AST.CIL.Load instruction) {
    checkSecurityLevelStackUniformity(instruction);

    SecurityLevel securityEnvironment = flowPolicy.getSecurityLevel(
        configuration.getSecurityEnvironment(instruction));

    SecurityLevel securityLevelOfVariable = flowPolicy.getSecurityLevel(
        configuration.getSecurityLevel(instruction.Target));

    SecurityLevel leastUpperBound = flowPolicy.getLeastUpperBound(securityEnvironment,
        securityLevelOfVariable);

    if (leastUpperBound == null)
        throw new AnalysisException(instruction.Position + " no least upper bound found for " + securityEnvironment.Name + " and " + securityLevelOfVariable.Name);

    state.RecentSecurityLevelStack.Push(leastUpperBound);

    state.setSecurityLevelStack(instruction, state.RecentSecurityLevelStack);
}

public override void visit(PNIC.Model.AST.CIL.Pop instruction) {
    checkSecurityLevelStackUniformity(instruction);

    SecurityLevel securityEnvironment = flowPolicy.getSecurityLevel(
        configuration.getSecurityEnvironment(instruction));

    state.RecentSecurityLevelStack.Pop();

    state.setSecurityLevelStack(instruction, state.RecentSecurityLevelStack);
}

public override void visit(PNIC.Model.AST.CIL.Push instruction) {
    checkSecurityLevelStackUniformity(instruction);

    SecurityLevel securityEnvironment = flowPolicy.getSecurityLevel(
        configuration.getSecurityEnvironment(instruction));

    state.RecentSecurityLevelStack.Push(securityEnvironment);

    state.setSecurityLevelStack(instruction, state.RecentSecurityLevelStack);
}

public override void visit(PNIC.Model.AST.CIL.Return instruction) {
    checkSecurityLevelStackUniformity(instruction);

    SecurityLevel leastUpperBound;

    if (state.RecentSecurityLevelStack.Count > 0)
        
    SecurityLevel topOfStackLevel = state.RecentSecurityLevelStack.Pop();

    SecurityLevel securityEnvironment = flowPolicy.getSecurityLevel(
        configuration.getSecurityEnvironment(instruction));

    leastUpperBound = flowPolicy.getLeastUpperBound(topOfStackLevel,
        securityEnvironment);

    if (leastUpperBound == null)
        throw new AnalysisException(instruction.Position + " no least upper bound found for " + securityEnvironment.Name + " and " + topOfStackLevel.Name);

    else
        leastUpperBound = flowPolicy.getSecurityLevel(
            configuration.getSecurityEnvironment(instruction));

    if (leastUpperBound != flowPolicy.LowestSecurityLevel)
        throw new AnalysisException(instruction.Position + " return reveals higher level value");

public override void visit(PNIC.Model.AST.CIL.Store instruction) {

D.2 Package Controller

public override void visit(PNIC.Model.AST.CIL.UnaryInstruction instruction)
{
    checkSecurityLevelStackUniformity(instruction);
    throw new NotImplementedException();
}

public override void visit(PNIC.Model.AST.CIL.UnconditionalJump instruction)
{
    checkSecurityLevelStackUniformity(instruction);
    state.setSecurityLevelStack(instruction, state.RecentSecurityLevelStack);
}

public override void visit(PNIC.Model.AST.CIL.Nop instruction)
{
    checkSecurityLevelStackUniformity(instruction);
    state.setSecurityLevelStack(instruction, state.RecentSecurityLevelStack);
}

@endregion

#region helpermethods

private void checkSecurityLevelStackUniformity(PNIC.Model.AST.Instruction instruction)
{
    //Console.WriteLine(instruction.ReadableInstruction + ", Stacksize: "+ state.RecentSecurityLevelStack.Count);
    if (instruction.PreviousInstructions.Count > 1)
    {
        foreach (PNIC.Model.AST.Instruction previousInstruction in instruction.PreviousInstructions)
        {
            Stack<SecurityLevel> stackAtPreviousInstruction = state.getSecurityLevelStack(previousInstruction);
            if (stackAtPreviousInstruction is Stack<SecurityLevel>)
            {
                if (stackAtPreviousInstruction.Count != state.RecentSecurityLevelStack.Count)
                    throw new AnalysisException(instruction.Position + " security level is not constant at this programpoint");
                SecurityLevel[] stackAtPreviousAsArray = stackAtPreviousInstruction.ToArray();
                SecurityLevel[] recentStackAsArray = state.RecentSecurityLevelStack.ToArray();
            }
        }
    }
}

public override void visit(PNIC.Model.AST.CIL.UnaryInstruction instruction)
{
    checkSecurityLevelStackUniformity(instruction);
    throw new NotImplementedException();
}

public override void visit(PNIC.Model.AST.CIL.UnconditionalJump instruction)
{
    checkSecurityLevelStackUniformity(instruction);
    state.setSecurityLevelStack(instruction, state.RecentSecurityLevelStack);
}

public override void visit(PNIC.Model.AST.CIL.Nop instruction)
{
    checkSecurityLevelStackUniformity(instruction);
    state.setSecurityLevelStack(instruction, state.RecentSecurityLevelStack);
}
@endregion

//Console.WriteLine(instruction.ReadableInstruction + ", Stacksize: "+ state.RecentSecurityLevelStack.Count);
if (instruction.PreviousInstructions.Count > 1)
{
    foreach (PNIC.Model.AST.Instruction previousInstruction in instruction.PreviousInstructions)
    {
        Stack<SecurityLevel> stackAtPreviousInstruction = state.getSecurityLevelStack(previousInstruction);
        if (stackAtPreviousInstruction is Stack<SecurityLevel>)
        {
            if (stackAtPreviousInstruction.Count != state.RecentSecurityLevelStack.Count)
                throw new AnalysisException(instruction.Position + " security level is not constant at this programpoint");
            SecurityLevel[] stackAtPreviousAsArray = stackAtPreviousInstruction.ToArray();
            SecurityLevel[] recentStackAsArray = state.RecentSecurityLevelStack.ToArray();
        }
    }
}
for (int i = 0; i < recentStackAsArray.Length; i++)
{
    if (recentStackAsArray[i] != stackAtPreviousAsArray[i])
        throw new AnalysisException(Instruction.Position + " security level is not constant at this program point");
}

#endif

D.2.4 File Analysis/MethodTransformer.cs

using System.Collections.Generic;
using System.Text;
using PNIC.Model.AST;
using PNIC.Model.AST.CIL;
using PNIC.Model.Analysis;

namespace PNIC.Controller.Analysis
{
    /// <summary>
    /// class representing an algorithm that analysis a method
    /// </summary>
    class MethodTransformer : InstructionContainerVisitor
    {
        private AnalysisSettings configuration;
        public AnalysisSettings Configuration
        {
            get { return configuration; }
            set { this.configuration = value; }
        }
        private FlowPolicy flowPolicy;
        public FlowPolicy FlowPolicy
        {
            get { return flowPolicy; }
            set { this.flowPolicy = value; }
        }

        private CILInstructionTransformer instructionTransformer;
        private List<Instruction> workList;
        private List<ConditionalJump> checkedBranchingInstructions;
        private Method recentMethod;

        /// <summary>
        /// method visitor for cil int analysis
        /// </summary>
        public override void visit(PNIC.Model.AST.Method method)
        {
            //mark the method as the method in progress
            recentMethod = method;
            //check configuration regions
            checkRegions();

            //initialize an instructionTransformer for this method
            instructionTransformer = new CILInstructionTransformer();
            instructionTransformer.Configuration = configuration;
            instructionTransformer.FlowPolicy = flowPolicy;

            //initialize the worklist and checkedBranches
            workList = new List<Instruction>();
            workList.Add(method.FirstInstruction);
        }
    }
checkedBranchingInstructions = new List<ConditionalJump> ();

Instruction recentInstruction;

//while the worklist is not empty, we are not done
while ( workList . Count > 0 ) {

recentInstruction = getFirstInstructionFromWorkList ();
if ( recentInstruction == null )
    throw new AnalysisException ( "Method contains instructions which are not part of the language" );
instructionTransformer . State . restoreState ( recentInstruction . DirectPredecessor );

//for all instructions that are transitively linked as direct successors to recent instruction
while ( recentInstruction != null ) {
    //decide how to handle this instruction
    if ( recentInstruction is ConditionalJump )
        visit (( ConditionalJump ) recentInstruction );
    else if ( recentInstruction is Instruction )
        visit ( recentInstruction );
    recentInstruction = recentInstruction . DirectSuccessor;
}
}

/// <summary>
/// checks if the regions confirm to the safe over approximation definition
/// if not so: throws an exception
/// </summary>
private void checkRegions () {
foreach ( ConditionalJump conditionalJump in recentMethod . ConditionalJumps ) {
    ControlDependencyRegion controlDependencyRegion = configuration .
        getControlDependencyRegion ( conditionalJump );
    //if the cdr of a conditional jump is empty, then the region is not correct
    if ( controlDependencyRegion == null )
        throw new AnalysisException ( "Control dependency region not set for conditional jump " + conditionalJump . Position );

    //for every programpoint i, j, k: if i is a branching instruction, then both following instructions j and k must be either element of region(i) or junction(i);
if ( conditionalJump . DirectSuccessor != conditionalJump . JumpInstruction ) {
    //check all successors
    foreach ( Instruction successor in conditionalJump . NextInstructions ) {
        if ( ! controlDependencyRegion . Region . Contains ( successor . Position )
            && controlDependencyRegion . Junction != successor . Position )
            throw new AnalysisException ( "Successor of conditional jump " + conditionalJump . Position + " is not junction and not in region" );
    }
}

    //#1: for every programpoint i, j, k: if j el region(i) and j→k, then k el region(i) or k == junction(i)
    //#2: for every i, j: if j el region(i) and j is return, then junction(i) == null
foreach ( int position in controlDependencyRegion . Region ) {
Instruction instructionInRegion = recentMethod/InstructionAtPosition(position);

// #1
if (instructionInRegion is ConditionalJump)
{
    ConditionalJump jumpInRegion = (ConditionalJump)instructionInRegion;
    if (controlDependencyRegion.Junction != jumpInRegion.JumpInstruction.Position
        & controlDependencyRegion.Region.Contains(jumpInRegion.JumpInstruction.Position))
        throw new AnalysisException("Target of jump " + jumpInRegion.Position + " in region of " + conditionalJump.Position + " is not junction and not in region");
    if (! (instructionInRegion is Return))
    {
        if (controlDependencyRegion.Junction != instructionInRegion.DirectSuccessor.Position
            & controlDependencyRegion.Region.Contains(instructionInRegion.DirectSuccessor.Position))
            throw new AnalysisException("Direct successor of instruction " + instructionInRegion.Position + " in region of " + conditionalJump.Position + " is not junction and not in region");
    }
}

// #2
if (instructionInRegion is Return)
{
    if (controlDependencyRegion.Junction != -1)
        throw new AnalysisException("Return " + instructionInRegion.Position + " in region of " + conditionalJump.Position + " with junction defined");
}

private void visit/Instruction(instruction)
{
    // analyse this instruction and remove it from the worklist
    instruction.accept(instructionTransformer);
    workList.Remove(instruction);
}

private void visit/ConditionalJump(conditionalJump)
{
    // analyse this instruction
    conditionalJump.accept(instructionTransformer);
    workList.Remove(conditionalJump);

    ControlDependencyRegion region = configuration/getControlDependencyRegion(conditionalJump);

    // if branching not checked
    if (!checkedBranchingInstructions.Contains(conditionalJump))
    {
        // add region to worklist
        foreach (int position in region.Region)
        {
            Instruction instructionAtPosition = recentMethod/InstructionAtPosition(position);
            // if instruction is not in worklist
            if (! workList.Contains(instructionAtPosition))
                workList.Add(instructionAtPosition);
        }

        // if jump target not in worklist, add jump target to worklist
        if (! workList.Contains(conditionalJump.JumpInstruction))
            workList.Add(conditionalJump.JumpInstruction);
D.2 Package Controller

D.2.5 File Print/CompletePrint.cs

```csharp
using PNIC.Model.AST;
using PNIC.Model.Analysis;

namespace PNIC.Controller.Print {

class CompletePrint : InstructionContainerVisitor {

    // If true, the output contains only the equivalence classes of instructions
    // else the concrete instruction is printed
}
D.2 Package Controller

D.2.6 File Print/Instruction.cs

```csharp
using System;
using System.Collections.Generic;
using System.Text;
using PNIC.Model.Analysis;

namespace PNIC.Controller.Print
{
    class PrintInstruction : CILintInstructionVisitor
    {
        public PrintInstruction()
        {
            //this.prefix = null;
        }

        /// <summary>
        /// if true, the output contains only the equivalence classes of instructions
        /// else the concrete instruction is printed
        /// </summary>
        private bool equivalenceClass;
    }
}
```
```java
public bool EquivalenceClass
{
    get { return this.equivalenceClass; }
    set { this.equivalenceClass = value; }
}

/// <summary>
/// If set additional information, e.g. security environments of instructions
/// are printed
/// </summary>
private AnalysisSettings configuration;
public AnalysisSettings Configuration
{
    get { return this.configuration; }
    set { this.configuration = value; }
}

private void printAdditionalInfos(PNIC.Model.AST.Instruction instruction)
{
    if (this.configuration is AnalysisSettings)
    {
        StringBuilder infoBuilder = new StringBuilder();
        infoBuilder.Append(configuration.getSecurityEnvironment(instruction));
        infoBuilder.Length = 5;
        infoBuilder.Append(" - ");
        Console.Out.WriteLine(infoBuilder.ToString());
    }
}

public override void visit(PNIC.Model.AST.CIL.BinaryInstruction instruction)
{
    printAdditionalInfos(instruction);
    if (this.equivalenceClass)
    {
        Console.Out.WriteLine(instruction.ReadableEquivalenceClass);
    }
    else
    {
        Console.Out.WriteLine(instruction.ReadableInstruction);
    }
}

public override void visit(PNIC.Model.AST.CIL.ConditionalJump1Parameter instruction)
{
    printAdditionalInfos(instruction);
    if (this.equivalenceClass)
    {
        Console.Out.WriteLine(instruction.ReadableEquivalenceClass);
    }
    else
    {
        Console.Out.WriteLine(instruction.ReadableInstruction);
    }
}

public override void visit(PNIC.Model.AST.CIL.ConditionalJump2Parameters instruction)
{
    printAdditionalInfos(instruction);
    if (this.equivalenceClass)
    {
        Console.Out.WriteLine(instruction.ReadableEquivalenceClass);
    }
    else
    {
        Console.Out.WriteLine(instruction.ReadableInstruction);
    }
}

public override void visit(PNIC.Model.AST.CIL.Load instruction)
{
    printAdditionalInfos(instruction);
    if (this.equivalenceClass)
    {
        Console.Out.WriteLine(instruction.ReadableEquivalenceClass);
    }
    else
    {
        Console.Out.WriteLine(instruction.ReadableInstruction);
    }
}

public override void visit(PNIC.Model.AST.CIL.Pop instruction)
{
    printAdditionalInfos(instruction);
```

if (this.equivalenceClass)
    Console.Out.WriteLine(instruction.ReadableEquivalentClass);
else
    Console.Out.WriteLine(instruction.ReadableInstruction);
}

public override void visit (PNIC.Model.AST.CIL.Push instruction) {
    printAdditionalInfos(instruction);
    if (this.equivalenceClass)
        Console.Out.WriteLine(instruction.ReadableEquivalentClass);
    else
        Console.Out.WriteLine(instruction.ReadableInstruction);
}

public override void visit (PNIC.Model.AST.CIL.Return instruction) {
    printAdditionalInfos(instruction);
    if (this.equivalenceClass)
        Console.Out.WriteLine(instruction.ReadableEquivalentClass);
    else
        Console.Out.WriteLine(instruction.ReadableInstruction);
}

public override void visit (PNIC.Model.AST.CIL.Store instruction) {
    printAdditionalInfos(instruction);
    if (this.equivalenceClass)
        Console.Out.WriteLine(instruction.ReadableEquivalentClass);
    else
        Console.Out.WriteLine(instruction.ReadableInstruction);
}

public override void visit (PNIC.Model.AST.CIL.UnaryInstruction instruction) {
    printAdditionalInfos(instruction);
    if (this.equivalenceClass)
        Console.Out.WriteLine(instruction.ReadableEquivalentClass);
    else
        Console.Out.WriteLine(instruction.ReadableInstruction);
}

public override void visit (PNIC.Model.AST.CIL.UnconditionalJump instruction) {
    printAdditionalInfos(instruction);
    if (this.equivalenceClass)
        Console.Out.WriteLine(instruction.ReadableEquivalentClass);
    else
        Console.Out.WriteLine(instruction.ReadableInstruction);
}

public override void visit (PNIC.Model.AST.CIL.Nop instruction) {
    printAdditionalInfos(instruction);
    if (this.equivalenceClass)
        Console.Out.WriteLine(instruction.ReadableEquivalentClass);
    else
        Console.Out.WriteLine(instruction.ReadableInstruction);
}

D.3 Package View

D.3.1 File ConsoleUI/ConsoleUI.cs

using System;
D.3 Package View

```csharp
using System.Collections.Generic;
using System.Text;
using PNIC.Model.Analysis;
using PNIC.Model.AST;
using PNIC.Controller.Print;
using PNIC.Controller.Analysis;

namespace PNIC.View.ConsoleUI
{
    /// <summary>
    /// console user interface
    /// </summary>
    class ConsoleUI
    {
        public ConsoleUI()
        {
            assemblies = new List<Assembly>();
            invokeMainMenu();
        }

        private List<Assembly> assemblies;
        private FlowPolicy flowPolicy;
        private AnalysisSettings configuration;

        private void printHeader()
        {
            Console.Clear();
            Console.WriteLine("Prototypical Non-Interference Checker");
            Console.WriteLine("-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-");
            Console.WriteLine();
            if (flowPolicy is FlowPolicy)
            {
                Console.WriteLine("Flow policy loaded: " + flowPolicy.Name);
            }
            if (configuration is AnalysisSettings)
            {
                Console.WriteLine("Analysis settings loaded: " + configuration.Name);
            }
            if (assemblies.Count != 0)
            {
                Console.WriteLine("Assemblies loaded: " + assemblies[0].Name);
                for (int i = 1; i < assemblies.Count; i++)
                {
                    Console.WriteLine(" " + assemblies[i].Name);
                }
                Console.WriteLine();
            }
        }

        private void invokeMainMenu()
        {
            while (true)
            {
                printHeader();
                Console.WriteLine("F - Flow Policy Menu");
                Console.WriteLine("S - Analysis Settings Menu");
                Console.WriteLine("A - Assembly Menu");
                Console.WriteLine("R - Run Analysis");
                Console.WriteLine("X - Exit");
                Console.WriteLine();
                char action = Console.ReadKey().KeyChar;
                Console.WriteLine();
                switch (action)
                {
                    case 's':
```
case 'S':
    invokeConfigurationMenu();
    break;
case 'F':
case 'f':
    invokeFlowPolicyMenu();
    break;
case 'A':
case 'a':
    invokeAssemblyMenu();
    break;
case 'r':
case 'R':
    runAnalysis();
    break;
case 'x':
case 'X':
    return;
}
}

#region flowpolicy

private void invokeFlowPolicyMenu()
{
    while (true)
    {
        printHeader();
        Console.Out.WriteLine("L - Load Flow Policy");
        Console.Out.WriteLine("D - Dump loaded Flow Policy");
        Console.Out.WriteLine("M - Back to Main Menu");
        Console.Out.WriteLine();
        Console.Out.Write("Action: ");
        char action = Console.ReadKey().KeyChar;
        switch (action)
        {
        case 'l':
        case 'L':
            LoadFlowPolicy();
            break;
        case 'd':
        case 'D':
            DumpFlowPolicy();
            break;
        case 'm':
        case 'M':
            return;
        }
    }
}

private void LoadFlowPolicy()
{
    Console.Out.WriteLine("Please enter the file name of the flow policy: ");
    string filename = Console.In.ReadLine();
    FlowPolicy oldpolicy = flowpolicy;
    flowpolicy = new FlowPolicy();
    while (!(flowpolicy is FlowPolicy) || !flowpolicy.loadFlowPolicyFromFile(filename))
    {
        Console.Out.WriteLine(filename + " is not a valid flow policy!");
        Console.Out.Write("Please enter the file name of the flow policy (empty line to abort): ");
        filename = Console.In.ReadLine();
        if (filename.Equals(""))
        {
            return;
        }
    }
}
D.3 Package View

```java
flowpolicy = oldpolicy;
break;
}
flowpolicy = new FlowPolicy();
}

private void DumpFlowPolicy()
{
    printHeader();
    if ( !( flowpolicy is FlowPolicy ))
        LoadFlowPolicy();
    if ( flowpolicy is FlowPolicy)
        Console.Out.WriteLine(flowpolicy.Readable);
    else
        Console.Out.WriteLine("No flow policy loaded");
    Wait();
}

#endregion

#region assembly menu

private void invokeAssemblyMenu()
{
    while (true)
    {
        printHeader();
        Console.Out.WriteLine("Loaded Assemblies:");
        foreach (Assembly assembly in assemblies)
        {
            Console.Out.WriteLine(assembly.Name);
        }
        Console.Out.WriteLine();
        Console.Out.WriteLine("L - Load Assembly");
        Console.Out.WriteLine("U - Unload Assembly");
        Console.Out.WriteLine("D - Dump loaded Assemblies");
        Console.Out.WriteLine("M - Back to Main Menu");
        Console.Out.WriteLine();
        Console.Out.WriteLine("Action: ");
        char action = Console.ReadKey().KeyChar;
        switch (action)
        {
            case 'l':
            case 'L':
                LoadAssembly();
                break;
            case 'u':
            case 'U':
                UnloadAssembly();
                break;
            case 'd':
            case 'D':
                DumpLoadedAssemblies();
                break;
            case 'm':
            case 'M':
                return;
        }
    }
}

private void LoadAssembly()
{
    Console.Out.WriteLine("Please enter the filename of the assembly: ");
    string filename = Console.In.ReadLine();
    foreach (Assembly existingAssembly in assemblies)
```
private void UnloadAssembly()
{
    Console.Out.WriteLine("Please enter the filename of the assembly: ");
    string filename = Console.In.ReadLine();
    Assembly asmToRemove = null;
    foreach (Assembly assembly in assemblies)
    {
        if (assembly.Name.ToLower().Equals(filename.ToLower()))
        {
            asmToRemove = assembly;
            break;
        }
    }
    if (asmToRemove != null)
        assemblies.Remove(asmToRemove);
}

private void DumpLoadedAssemblies()
{
    foreach (Assembly assembly in assemblies)
    {
        CompletePrint printer = new CompletePrint();
        printer.visit(assembly);
    }
    Wait();
}

private void InvokeConfigurationMenu()
{
    while (true)
    {
        printHeader();
        Console.Out.WriteLine("L - Load Analysis Settings");
        Console.Out.WriteLine("D - Dump loaded Analysis Settings");
        Console.Out.WriteLine("M - Back to Main Menu");
        Console.Out.WriteLine();
        char action = Console.In.ReadKey().KeyChar;
        Console.Out.WriteLine();
        switch (action)
        {
            case 'l':
                LoadConfiguration();
                break;
            case 'd':
                DumpConfiguration();
                break;
        }
    }
}
D.3 Package View

```csharp
private void InitializeNewConfiguration()
{
    Console.WriteLine("Please enter the filename of the analysis settings: ");
    string filename = Console.In.ReadLine();

    while (System.IO.File.Exists(filename))
    {
        Console.WriteLine("File already exists, please choose another name: ");
        filename = Console.In.ReadLine();
    }

    List<string> assemblies = new List<string>();
    Console.WriteLine("Please enter filenames for assemblies to include, one per line, empty line closes the list: ");
    string buffer = Console.In.ReadLine();
    while (!buffer.Equals(""))
    {
        assemblies.Add(buffer);
        buffer = Console.In.ReadLine();
    }

    AnalysisSettings configuration = new AnalysisSettings();
    configuration.InitializeNewConfiguration(filename, assemblies);
}

private void DumpConfiguration()
{
    Console.WriteLine("Settings for this analysis: ");
    Console.WriteLine("Settings File: " + configuration.Name);
    Console.WriteLine();
    Console.WriteLine(configuration.ReadableConfiguration);
    Wait();
}

private void LoadConfiguration()
{
    Console.WriteLine("Loading analysis settings");
    Console.WriteLine("Please enter the filename for the settings: ");
    string filename = Console.In.ReadLine();
    configuration = new AnalysisSettings();
    configuration.LoadConfigurationFromFile(filename);
}

#endregion

#region Analysis

private void RunAnalysis()
{
    if (!(flowpolicy is FlowPolicy))
    {
        Console.WriteLine("No flow policy loaded");
        LoadFlowPolicy();
    }
    if (!(configuration is AnalysisSettings))
    {
        
```
```csharp
Console.Out.WriteLine("No analysis settings loaded");
LoadConfiguration();
}
if (flowpolicy is FlowPolicy && configuration is AnalysisSettings)
{
    InteractiveAnalysis analysis = new InteractiveAnalysis();
    analysis.AnalysisSettings = configuration;
    analysis.FlowPolicy = flowpolicy;

    string result = analysis.runAnalysis();
    Console.Out.WriteLine();
    Console.Out.WriteLine(result);
    Wait();
}
#endregion

private void Wait()
{
    Console.Out.WriteLine("Press Enter to Continue");
    Console.ReadKey();
}
```
In this section, the source code for the example applications that were analysed can be found. Furthermore, the source code can be found in the folder /src/ExampleProgram on the cd-rom.

```java
class Testclass {
    public static void Main() {}  
    public int exampleDirectFlow(int l1, int h1) {
        l1 = h1;
        return l1;
    }

    public int exampleIndirectFlow1(int l1, int h1) {
        if (l1 == h1)
            return 1;
        else
            return 0;
    }

    public int exampleIndirectFlow2(int l1, int h1) {
        if (l1 == h1)
            l1 = 1;
        else
            l1 = 0;
        return l1;
    }

    public int addUserToSystem(int password, int salt) {
        //storage location, like pwd file
        int storagelocation;
        //generate userid
        int uid = 5;
        //calculate salted hash of password
        int saltedhash = salt * password;  
        //associate uid with salted hash in storage
        storagelocation = uid + saltedhash;
        storagelocation = storagelocation;
        //report the new uid
        return uid;
    }

    public int login(int uid, int password, int salt) {
        int storagelocation = 5 + 6 * 3;
        //if the uid is associated with the password, login succeeds
        if (storagelocation == uid + password + salt) {
            return 1;
        }
        return 0;
    }

    public int storeData(int uid, int password, int data, int salt) {
        int storagelocation = 23;
        //if account is associated with a special storage location
        if (storagelocation == uid + password + salt) {
            //store data in special location
            int specialstorage = data;
        }
```
public int licensedUpdate(int publicUserData, int privateUserData) {
    // the version number of this software
    int softwareVersion = 1;
    // store data in a user dataset
    int userdata = publicUserData + privateUserData;
    // send necessary user information to update server
    int unencryptedChannel = userdata;
    // receive answer from update server
    // if the user has a valid license
    if (unencryptedChannel < 5) {
        // receive and apply update
        int patchVersion = unencryptedChannel;
        softwareVersion = softwareVersion + patchVersion;
    } else {
        // disable the software
        softwareVersion = 0;
    }
    // return the new version number
    return softwareVersion;
}

public Testclass unhandledInstructionTest() {
    Testclass testclass = new Testclass();
    return this;
}

public int findGreatestDivisor(int number) {
    int i = 1;
    int j = number - 1;
    while ((i * j) != number && i <= j)
    {
        i += 1;
        while (i * j > number)
        {
            j -= 1;
        }
        if (i * j == number)
            return j;
    } else
        return number;
}

public int getKey(int uid, int pw) {
    int key;
    int secret = 23;
    if (uid == 5)
        if (pw == secret)
{ key = 1;
} else {
    key = 0;
}
else {
    key = 0;
} return key;