Bachelor Thesis
Bachelor of Science Informatik

Information Flow Analysis for CIL

Matthias Perner

TU Darmstadt
Fachbereich Informatik

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

Abgabetermin: 15. Oktober 2008

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 ressources. This flow can be restricted by flow policies to disallow public readable ressources 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 ressources and managed pointers to local information ressources.

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 ressources to analyse programs.

Contents

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 orginal 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 valueable 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 ressource.

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 ressource 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.

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 missunderstandings.

1.3.1 Replacements in Functions

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

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

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

1.3.2 Stacks and Stackoperations

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 $size : A^* \to \mathbb{N} \cup \{0\}$ be the function that returns the amount of elements on the stack and let $as[i]$ denote the ith 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 ressources, 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 ressource 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 ressource 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 $<$. 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 $(sl_1, sl_2) \in <$.
$<= \{s \in S \times S | s \notin \geq\}$.
In the sequel $sl_1 > sl_2$ will be used insted 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 $>$.

Definition 4: Illegal Information Flow

If $sl_1 > sl_2$, then no information may flow from sl_1 to sl_2.
If $sl_1 < sl_2$, then no information may flow from sl_2 to sl_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: $\sqcup$

Let $\sqcup$ be a partial function of the type $S \times S \rightharpoonup S$.
$sl_1 \sqcup sl_2 = sl_3$, where $sl_1, sl_2, sl_3 \in S$, if $sl_3 \geq sl_1 \wedge sl_3 \geq sl_2$ and $\forall sl_4 \in S$: if $sl_4 \geq sl_1 \wedge sl_4 \geq sl_2$, then $sl_4 \geq sl_3$.

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 $high > low$ and $low \leq high$. That means data from information ressources of security level "low" may flow to any other information ressource, whereas data from information ressources of security level "high" may only flow to information ressources 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 emailclient 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 ressources in use. Therefore it is possible to define the visibility of a ressorce as follows:

Definition 6: Visibility

Let sl_{obs} be the highest security level the observer is able to see.
An information ressource res is visibile to the observer, if $sl_{observer} \geq SL(res)$.
An information ressource res is invisible to the observer, if $sl_{observer} < SL(res)$.

2.3 Security Type Systems

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

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 Ressources

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_χ 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 χ, 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, $\rho \in \chi \rightharpoonup V$ 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 $\rightsquigarrow \subseteq State \times (State + V)$. $\rightsquigarrow^*$ denotes the transitive closure. Furthermore, we will use $s_1 \rightsquigarrow^n s_2$ 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 $\langle 0, \rho, \epsilon \rangle \rightsquigarrow^* v$.

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.

binary *op* Every instruction that pops two values from the operand stack, calculates a result by applying a binary function *op* to those values and pushes the result on the operand stack.

pop Every instruction that pops one instruction from the operand stack.

push v Every instruction that pushes one constant value v on the operand stack

loadlocal x Every instruction that loads the value of the local information ressource x on the operand stack.

storelocal x Every instruction that pops a value from the stack and stores it in the local information ressource x.

jumpWith1Argument t Every instruction that pops one value from the operand stack and passes control to either next instruction or target instruction $P[t]$ based on the value.

jumpWith2Arguments t Every instruction that pops two values from the operand stack and passes control to either next instruction or target instruction $P[t]$ based on these values.

unconditionalJump Every instruction that passes control to the target instruction $P[t]$ instead of the next instruction.

return Every instruction that exits the program and returns the top value of the operand stack.

The semantic of the different euqivalence classes is described in table 1.

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 fulfiled, 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 ressources are visible. Furthermore, the condition must depend on the security levels of the information ressources, because those security levels determine the least security level an observer must have to be able to see the information ressource.

A special information ressource, when dealing with operand stack based languages, is the operand stack. It would not be accurate enough for the operand stack to be of

$$\frac{P[i]{=}unary\ op \quad op{\in}\mathbb{U} \quad op(v){=}r}{\langle i,\rho,v{::}os\rangle{\rightsquigarrow}\langle i{+}1,\rho,r{::}os\rangle}$$

$$\frac{P[i]{=}binary\ op \quad op{\in}\mathbb{O} \quad op(v1,v2){=}r}{\langle i,\rho,v1{::}v2{::}os\rangle{\rightsquigarrow}\langle i{+}1,\rho,r{::}os\rangle}$$

$$\frac{P[i]{=}pop}{\langle i,\rho,v{::}os\rangle{\rightsquigarrow}\langle i{+}1,\rho,os\rangle}$$

$$\frac{P[i]{=}push\ v}{\langle i,\rho,os\rangle{\rightsquigarrow}\langle i{+}1,\rho,v{::}os\rangle}$$

$$\frac{P[i]{=}loadlocal\ x}{\langle i,\rho,os\rangle{\rightsquigarrow}\langle i{+}1,\rho,\rho(x){::}os\rangle}$$

$$\frac{P[i]{=}storelocal\ x}{\langle i,\rho,v{::}os\rangle{\rightsquigarrow}\langle i{+}1,\rho{\oplus}\{x{\mapsto}v\},os\rangle}$$

$$\frac{P[i]{=}jumpWith1Argument\ t \quad cond1(v){=}true}{\langle i,\rho,v{::}os\rangle{\rightsquigarrow}\langle t,\rho,os\rangle} \qquad \frac{P[i]{=}jumpWith1Argument\ t \quad cond1(v){=}false}{\langle i,\rho,v{::}os\rangle{\rightsquigarrow}\langle i{+}1,\rho,os\rangle}$$

$$\frac{P[i]{=}jumpWith2Arguments\ t \quad cond2(v1,v2){=}true}{\langle i,\rho,v1{::}v2{::}os\rangle{\rightsquigarrow}\langle t,\rho,os\rangle} \qquad \frac{P[i]{=}jumpWith2Arguments\ t \quad cond2(v1,v2){=}false}{\langle i,\rho,v1{::}v2{::}os\rangle{\rightsquigarrow}\langle i{+}1,\rho,os\rangle}$$

$$\frac{P[i]{=}unconditionalJump\ t}{\langle i,\rho,os\rangle{\rightsquigarrow}\langle t,\rho,os\rangle}$$

$$\frac{P[i]{=}return}{\langle i,\rho,v{::}os\rangle{\rightsquigarrow}v}$$

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

Table 1: Semantic of CIL_{int}

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 defintions to define the non-interference condition for the program. In the following definitions we will write $\sim$ instead $\sim_{sl_{obs}}$, because there is no risk of confusion.

Definition 7: Indistinguishability of Values

Two values are indistinguishable, if they are equal.

$$\frac{v = v\prime}{v \sim v\prime}$$

Definition 8: Indistinguishability of local Mappings

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

$$\frac{\forall x \in \chi : \rho(x) = \rho'(x) \vee SL(x) > sl_{obs}}{\rho \sim \rho\prime}$$

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:

$$\frac{size(os) = size(st) = size(st') = size(os') = 0}{os \sim_{st,st'} os'}$$

$$\frac{size(os) = size(st) = 0 \wedge os \sim_{st,st'} os' \wedge sl_v > sl_{obs}}{os \sim_{st,sl_v::st\prime} v :: os'}$$

$$\frac{os \sim_{st,st'} os' \wedge v = v' \wedge sl_v \leq sl_{obs}}{v :: os \sim_{sl_v::st,sl_v::st'} v' :: os'}$$

$$\frac{os \sim_{st,st'} os' \wedge sl_v > sl_{obs} \wedge sl'_v > sl_{obs}}{v :: os \sim_{sl_v::st,sl'_v::st'} v' :: os'}$$

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 $\rho, \rho' \in \chi \rightharpoonup V$ and operandstacks $os, os' \in V^*$ are indistinguishable.

$$\frac{\rho \sim_{sl_{obs}} \rho' \wedge os \sim_{st,st'} os'}{\langle i, \rho, os\rangle \sim \langle i', \rho', os'\rangle}$$

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 ρ and ρ', where $\rho \sim \rho\prime$, the result v of a run of the program is equal.

$$\frac{\forall \rho, \rho' \in \chi \rightharpoonup V : \rho \sim \rho' \wedge P, \rho \Downarrow v \wedge P, \rho' \Downarrow v' \Rightarrow v = v'}{P\ is\ non - interfering}$$

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.

```
00 loadlocal hv
01 jumpWith1Argument 05
02 push 0
03 storelocal lv
04 unconditionalJump 07
05 push 1
06 storelocal lv
07 loadlocal lv
08 return
```

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 over approximation 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 $BP = \{i \in PP | (P[i] = jumpWith1Argumentt \vee P[i] = jumpWith2Argumentst) \wedge t \neq i + 1\}$.

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

$$junction : BP \rightharpoonup PP$$

$$region : BP \rightarrow \mathcal{P}(PP)$$

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 $junction(i)$ 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:

```
00 loadlocal lv
01 jumpWith1Argument 03
02 return
03 return
```

To model the control dependency regions as an safe over approximation 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 ressources outside its control dependency region.

Let $\mapsto \in PP \times PP \cup \{-1\}$ be the successor relation. It is defined by:

1. if $P[i] = unconditionalBranch\ t$, then $(i,t) \in \mapsto$
2. if $P[i] = jumpWith1Argument\ t$, then $(i, i+1) \in \mapsto$ and $(i,t) \in \mapsto$
3. if $P[i] = jumpWith2Arguments\ t$, then $(i, i+1) \in \mapsto$ and $(i,t) \in \mapsto$
4. if $P[i] = return$, 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 dependecy region must fulfil the following three conditions:

1. for all program points i and all their succesors j, k that fulfil $j \neq k$ (that means i is a branching instruction and therefore $i \in BP$), either $k \in region(i)$ or $k = junction(i)$

$$\forall i,j,k \in PP : i \mapsto k \wedge i \mapsto j \wedge j \neq k \Rightarrow k \in region(i) \vee k = junction(i)$$

2. for all program points i, j, k, that fulfil $j \in region(i)$ and $(j,k) \in \mapsto$, either $k \in region(i)$ or $k = junction(i)$

$$\forall i,j,k \in PP : j \in region(i) \wedge j \mapsto k \Rightarrow k \in region(i) \vee k = junction(i)$$

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

$$\forall i,j \in PP : j \in region(i) \wedge P[j] = return \Rightarrow junction(i)\ is\ undefined$$

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.

3.4.2 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 ressources. 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:

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

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

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

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 paramter. Furthermore, the loading rules push values on the stack and push the least upper bound of security levels of all information ressources 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.

$$\frac{P[i]=unary\ op}{i\vdash sl::st,se\Rightarrow sl\sqcup se(i)::st,se}$$

$$\frac{P[i]=binary\ op}{i\vdash sl_1::sl_2::st,se\Rightarrow sl_1\sqcup sl_2\sqcup se(i)::st,se}$$

$$\frac{P[i]=pop}{i\vdash sl_1::st,se\Rightarrow st,se}$$

$$\frac{P[i]=push\ v}{i\vdash st,se\Rightarrow se(i)::st,se}$$

$$\frac{P[i]=loadlocal\ x}{i\vdash st,se\Rightarrow SL(x)\sqcup se(i)::st,se}$$

$$\frac{P[i]=stloc\ x \quad SL(X)\geq sl_1\sqcup se(i)}{i\vdash sl_1::st,se\Rightarrow st,se}$$

$$\frac{P[i]=unconditionalJump\ t}{i\vdash st,se\Rightarrow st,se}$$

$$\frac{P[i]=jumpWith1Argument\ t}{i\vdash sl_1::st,se\Rightarrow lift_{sl_1}(st),lift_{sl_1}(se,region(i))}$$

$$\frac{P[i]=jumpWith2Arguments\ t}{i\vdash sl_1::sl_2::st,se\Rightarrow lift_{sl_1\sqcup sl_2}(st),lift_{sl_1\sqcup sl_2}(se,region(i))}$$

$$\frac{P[i]=return \quad sl_1\sqcup se(i)=lsl}{i\vdash sl_1::st,se\Rightarrow}$$

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

Table 2: Abstract transferrules for CIL_{int}

It is obvious that this informal definition requires a defintion 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 initital states $\langle 0, \rho, \epsilon\rangle$ and ϵ, se of a program or if the states result from two corresponding states by applying a semantic rule and an abstract transfer rule.

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

$$\frac{st_j, se_j \Rightarrow st_i, se_i \quad \langle j, \rho_j, os_j\rangle \rightsquigarrow \langle i, \rho_i, os_i\rangle \quad \langle j, \rho_j, os_j\rangle \parallel st_j, se_j}{\langle i, \rho_i, os_i\rangle \parallel st_i, se_i}$$

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_{int} **Typeablility**

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.

$$\frac{\forall v \in V : P, \rho \Downarrow v \Rightarrow \langle 0, \rho, \epsilon \rangle \rightsquigarrow^* \langle i, \rho_i, v :: os_i \rangle \rightsquigarrow v \wedge st_i, se_i \rightarrow \wedge \langle i, \rho_i, v :: os_i \rangle \parallel st_i, se_i}{p \text{ is typable}}$$

It is now possible to proof that all typable programs fulfil 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 ressources and visible instructions preserve the indistinguishability of states. The first claim is necessary, because changes in low level information ressources 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 ressources 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 $CIL_{pointer}$ type system, which is more complex and contains the proof for the CIL_{int} system, because $CIL_{pointer}$ is a superset of CIL_{int} and the abstract transfer rules of the type system for $CIL_{pointer}$ contain the typability conditions of the CIL_{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 fulfil 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 analyse 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.

4.2.1 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 ressources, 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 indepedent 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.

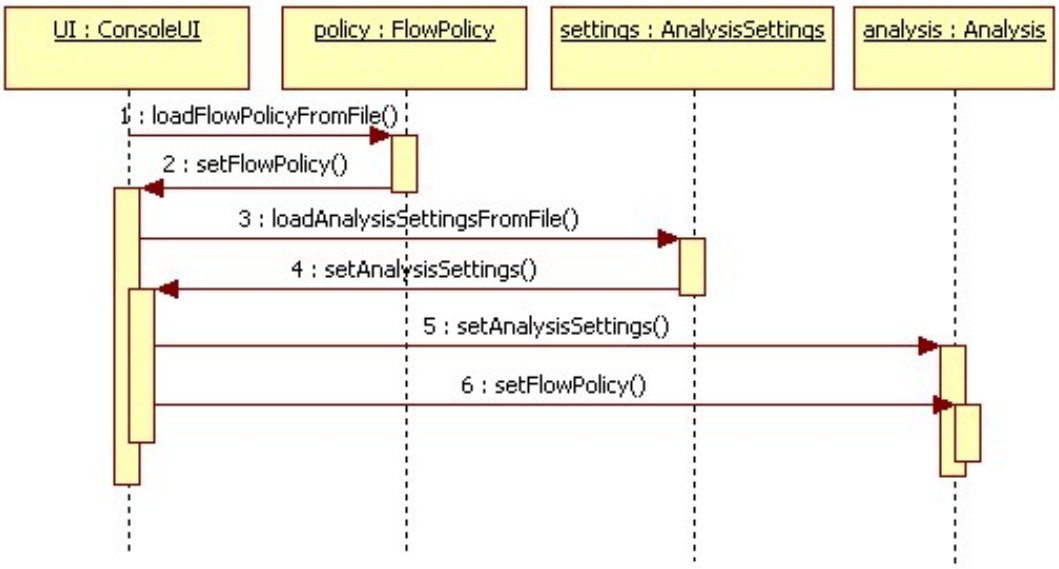

Figure 1: Setup process of an analysis

One could argument, 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 indepedent and thus improve the reuseability 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.

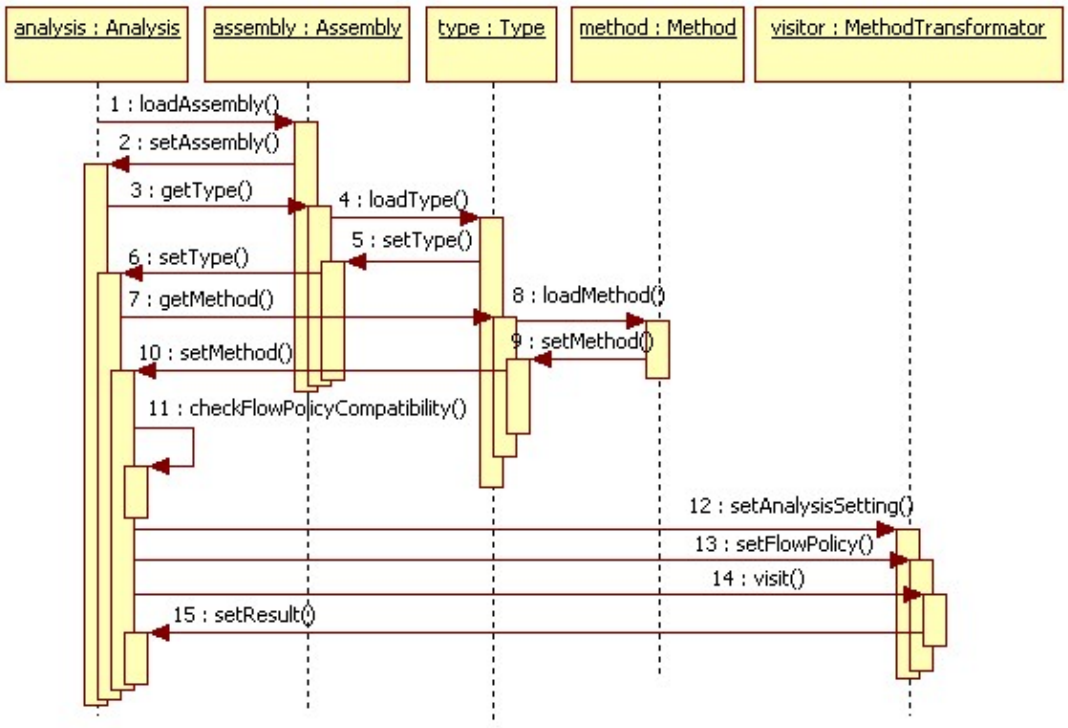

Figure 2: Running an analysis on a single method

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. Furhtermore, 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.

- 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.

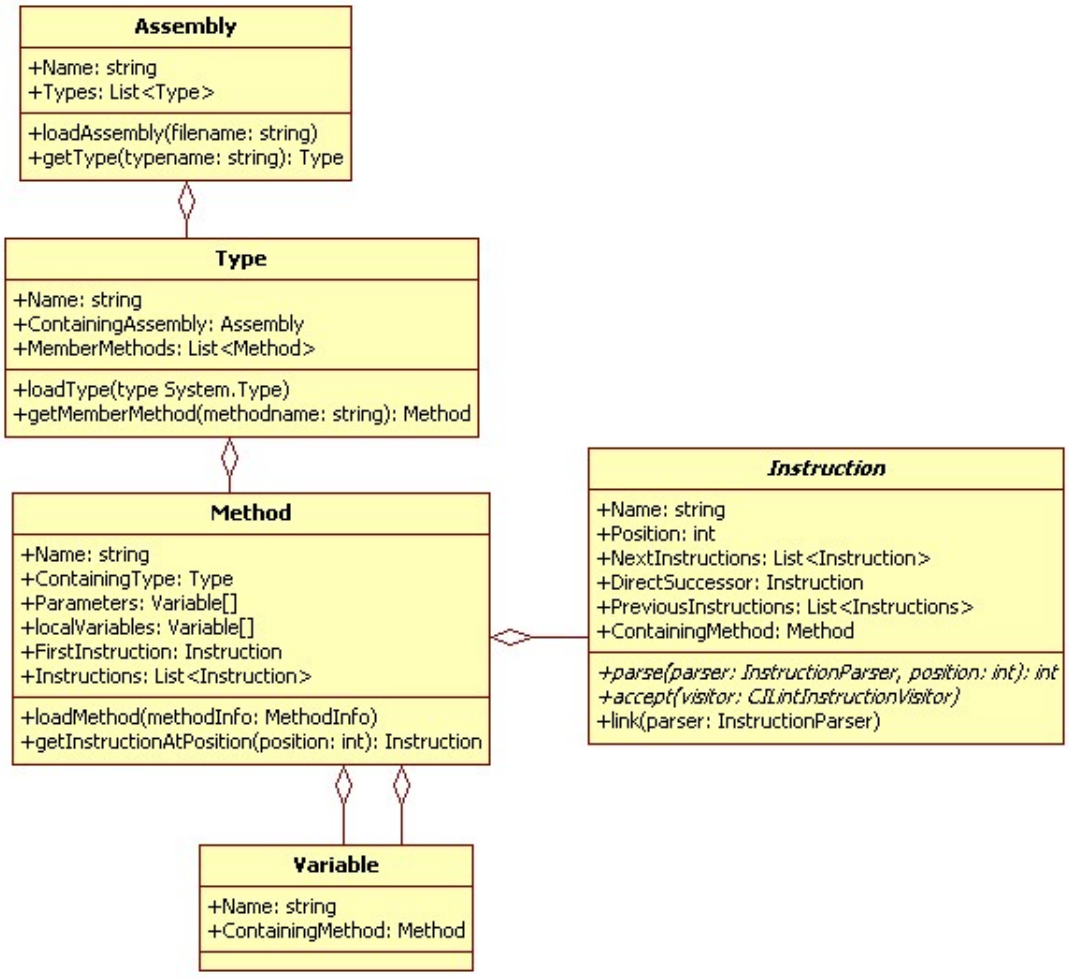

Figure 3: Overview of the abstract syntax tree

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 to complex, because it is not the main goal to implement a feature rich parser, but a functional parser that fulfils 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 instuctions 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.

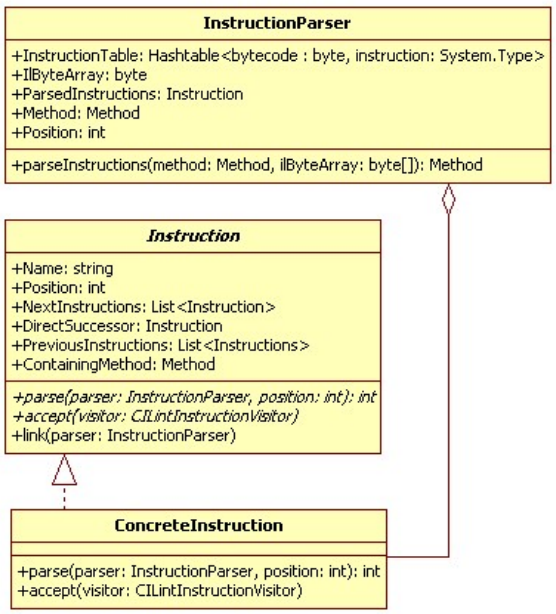

Figure 4: Instruction Parser

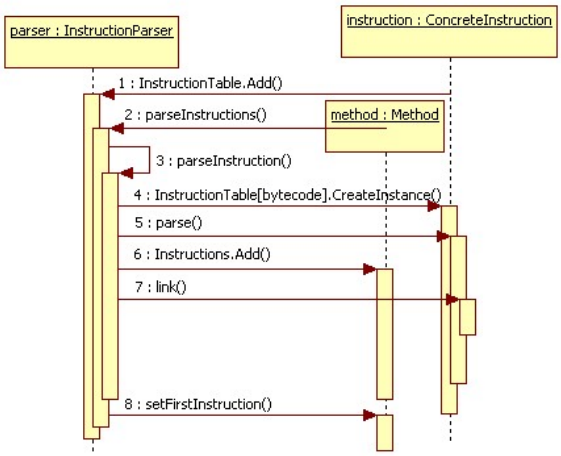

Figure 5: Parsing of Instructions

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.

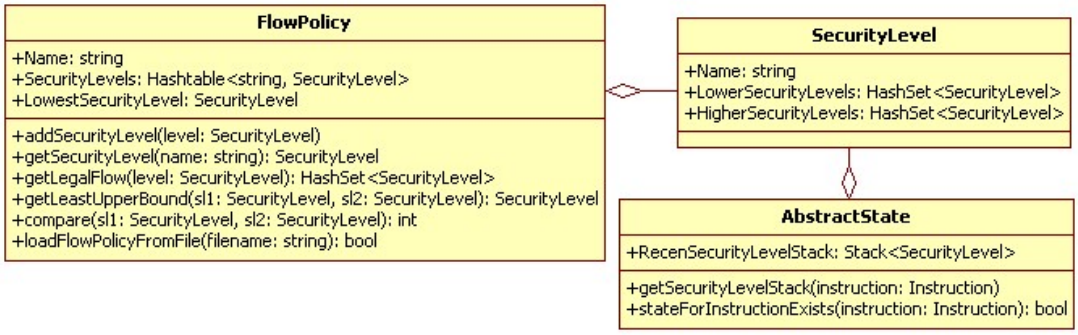

Figure 6: Design of Security Level, Flow Policy and Abstract State

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 ressources 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 pragraphs can be seen as a model for the analysis and are therefore contained in the package "Model" in the subpackage "Analysis".

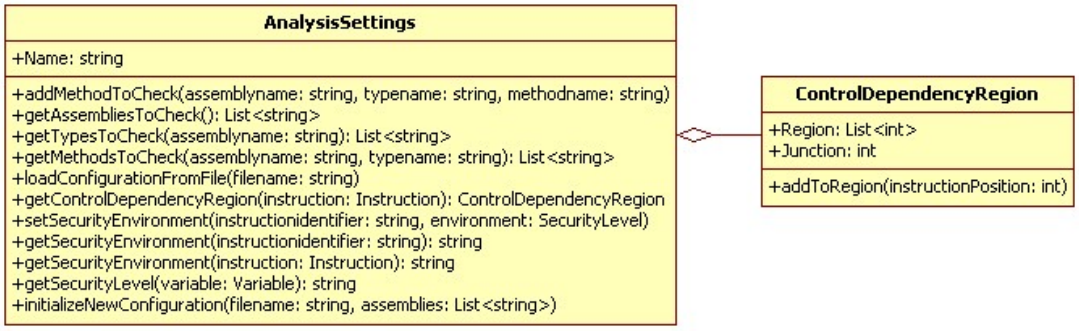

Figure 7: Analysis Settings

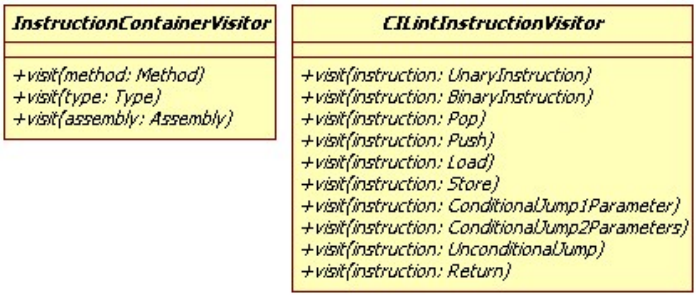

Figure 8: Abstract Visitors

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 implemantation 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 "CILintInstructionVisitor", is for each single instruction. In consequence it is necessary to extend this visitor, if a new instruction is introduced. But the implemenations 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

visitor during the analysis.

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 "CILintInstruction-Visitor". 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_{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 ressource to an low information ressource. The following C# example illustrates this kind of leak:

```
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:

```
Analysing method: exampleDirectFlow(System.Int32, System.Int32)
->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 ressource is assigned to a low level information ressource. 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 ressource does result in a branching and in both branches a return exists. The following C# example illustrates this problem:

```
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 an 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:

```
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.int8 8
6: ldc.i4.0
7: starg.s 1
8: ldarg.1
9: return
```

The similarities of the C# sourcecode 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:

```
Analysing 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 ressource is assigned to arg_1, which is a low level information ressource. 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 ressource.

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_{int} look very constructed. Nontheless 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:

```
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

```
Analysing method: addUserToSystem(System.Int32, System.Int32)
->addUserToSystem(System.Int32, System.Int32) is non-interfering
```

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:

```
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 *storagelocation*. 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 ressources 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 ressources are assigned to a low level ressource 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:

```
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 ressources, because they may not be revealed to the user. The analysis returns:

```
Analysing method: storeData(System.Int32, System.Int32, System.Int32, System.Int32)
->10 return reveals higher level value
```

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-Intereference Checker" the user must parameterize the analysis with the control dependecy 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.

6 $CIL_{pointer}$ - Basic Managed Pointers

After adadapting the basic analysis of [BR05] for JVM_χ to an equivalent subset of CIL. It is now interesting to extend the analysis to suppot a concept that is not known in the java bytecode. $CIL_{pointer}$ extends CIL_{int} to support basic managed pointers. Therefore we introduce instructions for loading addresses of local information ressources 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

$CIL_{pointer}$ makes use of the same information ressources 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 ressource. 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 χ 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

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 χ and the fact that it may now contain ressources 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 ressources that may be visible from outside the program. In consequence, a complete execution of a program can now be described by $\langle 0, \rho_0, \epsilon \rangle \rightsquigarrow^* \rho_r, v$. 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 ρ_0 evaluates to the local mapping ρ_r and the value v.

Additionaly, our analysis assumes that no changes are done to the information ressources in χ from outside the program. This includes the subset of non-local ressources 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, interpretes 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, interpretes 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.

$$\frac{P[i]{=}loadlocaladdress\ x}{\langle i,\rho,os\rangle \rightsquigarrow \langle i{+}1,\rho,\&(x){::}os\rangle}$$

$$\frac{P[i]{=}loadindirect}{\langle i,\rho,\&(x){::}os\rangle \rightsquigarrow \langle i{+}1,\rho,\rho(x){::}os\rangle}$$

$$\frac{P[i]{=}storeindirect}{\langle i,\rho,v{::}\&(x){::}os\rangle \rightsquigarrow \langle i{+}1,\rho{\oplus}\{x{\mapsto}v\},os\rangle}$$

$$\frac{P[i]{=}return}{\langle i,\rho,v{::}os\rangle \rightsquigarrow \rho,v}$$

where $v \in V$, $x \in \chi$,
$\&(x)$ denotes the address of the information ressource $x \in \chi$

Table 3: Semantic of additional or changed $CIL_{pointer}$ Instructions

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 ressource it references.

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

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 defintition of indistinguishable local mappings says, that two variable mappings $\rho, \rho' \in \chi \rightharpoonup V$ are indistiguishable with respect to a security level $sl_{obs} \in S$, if for every information ressource $x \in \chi$ 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 $PT = (S \times (S \cup \{\bot\}))^*$.

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 operandstacks are indistinguishable with respect to a security level $sl_{obs} \in S$ and two security level stacks $st, st' \in (S \times (S \cup \{\bot\}))^*$ 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}.

$$\frac{size(os) = size(st) = size(st') = size(os') = 0}{os \sim_{st,st'} os'}$$

$$\frac{size(os) = size(st) = 0 \wedge os \sim_{st,st'} os' \wedge sl_v > sl_{obs}}{os \sim_{st,(sl_v,pl_v)::st'} v :: os'}$$

$$\frac{os \sim_{st,st'} os' \wedge v = v' \wedge sl_v \leq sl_{obs}}{v :: os \sim_{(sl_v,pl_v)::st,(sl_v,pl_v)::st'} v' :: os'}$$

$$\frac{os \sim_{st,st'} os' \wedge sl_v > sl_{obs} \wedge sl'_v > sl_{obs}}{v :: os \sim_{(sl_v,pl_v)::st,(sl'_v,pl'_v)::st'} v' :: os'}$$

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

The defintion for non-interfering programs said that a program P is non-interfering, if for every two local mappings ρ and ρ', where $\rho \sim \rho\prime$, 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 χ.

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

Definition 18: Non-Interfering $CIL_{pointer}$ Program

A program P is non-interfering, if for every pair of indistinguishable local mappings $\rho \sim \rho'$ and the results of the executions, in case they terminate, are ρ_r, r and ρ'_r, r', then $\rho_r \sim \rho'_r$ and $r = r'$.

$$\frac{\forall \rho, \rho' \in \chi \rightharpoonup V : \rho \sim \rho' \wedge P, \rho \Downarrow \rho_r, r \wedge P, \rho' \Downarrow \rho'_r, r' \Rightarrow \rho_r \sim \rho'_r \wedge r = r'}{P \; is \; non - interfering}$$

6.5 Typesystem

Now it is possible to adapt the typesystem for CIL_{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_{int}. Managed pointers introduce a level of indirection, therefore that must be addressed in the transfer rules.

The analysis for CIL_{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 instrution 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 $lift_{sl}$

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

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

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

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

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.

$$\frac{P[i]{=}unary\ op}{i{\vdash}(sl_1,pl_1){::}pt,se{\Rightarrow}(sl_1{\sqcup}se(i),{\bot}){::}pt,se}$$

$$\frac{P[i]{=}binary\ op}{i{\vdash}(sl_1,pl_1){::}(sl_2,pl_2){::}pt,se{\Rightarrow}(sl_1{\sqcup}sl_2{\sqcup}se(i),{\bot}){::}pt,se}$$

$$\frac{P[i]{=}pop}{i{\vdash}(sl_1,pl_1){::}pt,se{\Rightarrow}pt,se}$$

$$\frac{P[i]{=}push\ v}{i{\vdash}pt,se{\Rightarrow}(se(i),{\bot}){::}pt,se}$$

$$\frac{P[i]{=}loadlocal\ x}{i{\vdash}pt,se{\Rightarrow}(SL(x){\sqcup}se(i),PL(x)){::}pt,se}$$

$$\frac{P[i]{=}storelocal\ x \quad sl_1{\sqcup}se(i){\leq}SL(x) \quad pl_1{=}PL(x)}{i{\vdash}(sl_1,pl_1){::}pt,se{\Rightarrow}pt,se}$$

$$\frac{P[i]{=}unconditionalJump\ t}{i{\vdash}pt,se{\Rightarrow}pt,se}$$

$$\frac{P[i]{=}jumpWith1Argument\ t}{i{\vdash}(sl_1,pl_1){::}pt,se{\Rightarrow}lift_{sl_1}(pt),lift_{sl_1}(se,region(i))}$$

$$\frac{P[i]{=}jumpWith2Arguments\ t}{i{\vdash}(sl_1,pl_1){::}(sl_2,pl_2){::}pt,se{\Rightarrow}lift_{sl_1{\sqcup}sl_2}(pt),lift_{sl_1{\sqcup}sl_2}(se,region(i))}$$

$$\frac{P[i]{=}loadlocaladdress\ x \quad PL(x){\neq}{\bot}}{i{\vdash}pt,se{\Rightarrow}(SL(x){\sqcup}se(i),SL(x)){::}pt,se}$$

$$\frac{P[i]{=}loadindirect \quad pl_1{\neq}{\bot}}{i{\vdash}(sl_1,pl_1){::}pt,se(i){\Rightarrow}(sl_1{\sqcup}pl_1{\sqcup}se(i),{\bot}){::}pt,se}$$

$$\frac{P[i]{=}storeindirect \quad pl_2{\geq}sl_1{\sqcup}sl_2{\sqcup}se(i)}{i{\vdash}(sl_1,pl_1){::}(sl_2,pl_2){::}pt,se{\Rightarrow}pt,se}$$

$$\frac{P[i]{=}return \quad sl_1{\sqcup}se(i){=}lsl}{i{\vdash}(sl_1,pl_1){::}pt,se{\Rightarrow}}$$

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

Table 4: Abstract Transferrules for $CIL_{pointer}$

The first interesting change is the new transfer rule for *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 ressource of the security level $SL(x)$, therefore the pointee level must be $SL(x)$.

The second interesting rule is the rule for *storelocal* x. When executing this transferrule a value gets poped 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 $pl_1 = PL(x)$. 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 *loadindirect*. When executing this instruction an address gets poped from the stack and the value tored in the ressource referenced by this address gets poped 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 *loadlocaladdress* x, the security level of the variable referenced by the address is pushed as pointee level on the stack type. With these prerequisits 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 $pl_1 \neq \bot$. 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 $se(i)$, the pointee level of the address pl_1 and the security level of the source of the address sl_1. As result, the security level of the new value must be $se(i) \sqcup ml_1 \sqcup sl_1$. 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 *storeindirect*. When executing this instruction an address and a value get poped 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 environtment, too, or an illegal indirect flow of information exists. Furthermore, we must check if the address depends on a high level ressource and we must include sl_2. This results in the condition $pl_2 \geq sl_1 \sqcup sl_2 \sqcup se(i)$.

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

$$\frac{pt_i, se_i = \epsilon, se \quad \langle i, \rho_i, os_i \rangle = \langle 0, \rho_0, \epsilon \rangle}{\langle i, \rho_i, os_i \rangle \parallel pt_i, se_i}$$

$$\frac{pt_j, se_j \Rightarrow pt_i, se_i \quad \langle j, \rho_j, os_j \rangle \rightsquigarrow \langle i, \rho_i, os_i \rangle \quad \langle j, \rho_j, os_j \rangle \parallel pt_j, se_j}{\langle i, \rho_i, os_i \rangle \parallel pt_i, se_i}$$

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: $CIL_{pointer}$ **Typablility**

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.

$$\frac{\forall \rho_r \in \chi \rightharpoonup V, r \in V : P, \rho \Downarrow \rho_r, r \Rightarrow \langle 0, \rho, \epsilon \rangle \rightsquigarrow^* \langle i, \rho_i, r :: os_i \rangle \rightsquigarrow \rho_r, r \wedge pt_i, se_i \rightarrow \wedge \langle i, \rho_i, r :: os_i \rangle \parallel pt_i, se_i}{p \ is \ typable}$$

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 fulfils 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 fulfils the condition:
$\forall \rho_r \in \chi \rightharpoonup V, r \in V : P, \rho \Downarrow \rho_r, r \Rightarrow \langle 0, \rho, \epsilon \rangle \rightsquigarrow^* \langle i, \rho_i, r :: os_i \rangle \rightsquigarrow \rho_r, r \wedge pt_i, se_i \rightarrow \wedge \langle i, \rho_i, r :: os_i \rangle \parallel pt_i, se_i$
The soundness theorem says that, if this condition is fulfiled, then the program is non-interfering and, in consequence, must fulfil:

$$\forall \rho, \rho' \in \chi \rightharpoonup V : \rho \sim \rho' \wedge P, \rho \Downarrow \rho_r, r \wedge P, \rho' \Downarrow \rho'_r, r' \Rightarrow \rho_r \sim \rho'_r \wedge r = r'$$

If $\rho \sim \rho'$ or $P, \rho \Downarrow \rho_r, r$ or $P, \rho' \Downarrow \rho'_r, r'$ is not fulfiled, the non-interference condition is fulfiled. Hence, it is only necessary to proof that $r = r'$ and $\rho_r \sim \rho'_r$ is fulfiled, in the case that the first three subterms are fulfiled.
Furthermore, we know that $P, \rho \Downarrow \rho_r, v$ is a special case of $\langle i, \rho_i, os_i \rangle \rightsquigarrow^* \rho_r, v$, where $i = 0$, $\rho_i = \rho$ and $os_i = \epsilon$.
In consequence, it is possible to write the non-interference condition as:

$$\forall \rho, \rho' \in \chi \rightharpoonup V : \rho \sim \rho' \wedge \langle 0, \rho, \epsilon \rangle \rightsquigarrow^* \rho_r, r \wedge \langle 0, \rho', \epsilon \rangle \rightsquigarrow^* \rho'_r, r' \Rightarrow \rho_r \sim \rho'_r \wedge r = r'$$

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.

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.

$\rightsquigarrow_v = \{(\langle i, \rho_i, os_i\rangle, \langle j, \rho_j, os_j\rangle) \in \rightsquigarrow | se(i) \leq sl_{obs}\}$
$\cup\{\langle i, \rho_i, os_i\rangle, \langle j, \rho_j, os_j\rangle) \in State \times State | \langle k, \rho_k, os_k\rangle \rightsquigarrow \langle i, \rho_i, os_i\rangle \rightsquigarrow^* \langle j, \rho_j, os_j\rangle$
$\wedge k \notin region(l) \wedge i \in region(l) \wedge j = junction(l)\}$

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

Lemma 1: High Level Regions Converge

In a typable and terminating program execution, the execution paths that pass a high level region converge at the junction.

$\forall i, j, k, l \in PP : l \notin region(i) \wedge k \in region(i) \wedge j = junction(i)$
$\wedge P, \rho \Downarrow \rho_r, v \wedge \langle l, \rho_l, os_l\rangle \rightsquigarrow \langle k, \rho_k, os_k\rangle$
$\Rightarrow \langle l, \rho_l, os_l\rangle \rightsquigarrow \langle k, \rho_k, os_k\rangle \rightsquigarrow^* \langle j, \rho_j, os_j\rangle$

The proof for this lemma can be found in the appendix C.1.1.

Lemma 2: Virtual Steps Preserve Indistinguishability of States

In a typable program P, the execution of a single instruction preserves the indistinguishability of states:

$$\langle i, \rho_i, os_i\rangle \rightsquigarrow_v \langle j, \rho_j, os_j\rangle \wedge \langle i, \rho'_i, os'_i\rangle \rightsquigarrow_v \langle j, \rho'_j, os'_j\rangle \wedge \langle i, \rho_i, os_i\rangle \sim \langle i, \rho'_i, os'_i\rangle \Rightarrow \langle j, \rho_j, os_j\rangle \sim \langle j, \rho'_j, os'_j\rangle$$

This lemma can be proved by showing that every single abstract transfer rule of the typesystem enforces that the indistinguishability of the 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.

Let $\langle i, \rho_i, os_i\rangle \rightsquigarrow_v \langle j, \rho_j, os_j\rangle$, $\langle i, \rho'_i, os'_i\rangle \rightsquigarrow_v \langle j, \rho'_j, os'_j\rangle$ and $\langle i, \rho_i, os_i\rangle \sim \langle i, \rho'_i, os'_i\rangle$ for the following proofs.

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

Now we will show that the instructions "unary op", "storelocal x", "loadlocaladdress x", "loadindirect" and "storeindirect" preserve the indistinguishability of the values.

Proof: unary op

Using the semantic of the instruction, we know that
$\rho_i \sim \rho'_i \wedge \rho_i = \rho_j \wedge \rho'_i = \rho'_j \wedge os_i \sim_{(sl_v, pl_v)::pt, (sl'_v, pl'_v)::pt'} os'_i$
$\wedge os_i = v :: os \wedge os'_i = v' :: os' \wedge os_j = f(v) :: os \wedge os'_j = f(v') :: os'$ is fulfiled.
And we can conclude that
$\rho_j \sim \rho'_j \wedge os_i \sim_{(sl_v, pl_v)::pt, (sl'_v, pl'_v)::pt'} os'_i \wedge os_i = v :: os \wedge os_i = v' :: os'$
$\wedge os_j = f(v) :: os \wedge os'_j = f(v') :: os'$
In consequence, we know that

$\rho_j \sim \rho'_j \wedge v :: os \sim_{(sl_v,pl_v)::pt,(sl'_v,pl'_v)::pt'} v' :: os' \wedge os_j = f(v) :: os \wedge os'_j = f(v') :: os'$
From the defintion of indistinguishable operand stacks we know that we have to distinguish two cases. The first $sl_v \leq sl_{obs} \wedge sl'_v \leq sl_{obs}$ that describes that the top value of the stack is visible and the second $sl_v > sl_{obs} \wedge sl'_v > sl_{obs}$ that describes that the top value of the stack is invisible.

1. case: $sl_v \leq sl_{obs} \wedge sl'_v \leq sl_{obs}$
Using the definition for indistinguishable operand stacks, we know that
$\rho_j \sim \rho'_j \wedge v = v' \wedge os \sim_{pt,pt'} os' \wedge os_j = f(v) :: os \wedge os'_j = f(v') :: os'$
And we can conclude that
$\rho_j \sim \rho'_j \wedge v = v' \wedge f(v) = f(v') \wedge os \sim_{pt,pt'} os' \wedge os_j = f(v) :: os \wedge os'_j = f(v') :: os'$
With the defintion of inditinguishable operand stacks, we know that
$\rho_j \sim \rho'_j \wedge f(v) :: os \sim_{(sl_{f(v)},\bot)::pt,(sl_{f(v')},\bot)::pt'} f(v') :: os'$
$\wedge os_j = f(v) :: os \wedge os'_j = f(v') :: os'$
In consequence, we know that
$\rho_j \sim \rho'_j \wedge os_j \sim_{(sl_{f(v)},\bot)::pt,(sl_{f(v')},\bot)::pt'} os'_j$
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
$\langle i, \rho_i, os_i \rangle \rightsquigarrow_v \langle j, \rho_j, os_j \rangle \wedge \langle i, \rho'_i, os'_i \rangle \rightsquigarrow_v \langle j, \rho'_j, os'_j \rangle \wedge \langle i, \rho_i, os_i \rangle \sim \langle i, \rho'_i, os'_i \rangle$
$\Rightarrow \langle j, \rho_j, os_j \rangle \sim \langle j, \rho'_j, os'_j \rangle$ is fulfiled.

2. case: $sl_v > sl_{obs} \wedge sl'_v > sl_{obs}$
Using the definition for indistinguishable operand stacks, we know that
$\rho_j \sim \rho'_j \wedge sl_v > sl_{obs} \wedge sl'_v > sl_{obs} \wedge os \sim_{pt,pt'} os' \wedge os_j = f(v) :: os \wedge os'_j = f(v') :: os'$
Using the abstract transfer rule, we know that
$\rho_j \sim \rho'_j \wedge sl_v > sl_{obs} \wedge sl'_v > sl_{obs} \wedge os \sim_{pt,pt'} os'$
$\wedge os_j = f(v) :: os \wedge os'_j = f(v') :: os' \wedge sl_{f(v)} = sl_v \sqcup se(i) \wedge sl_{f(v')} = sl'_v \sqcup se(i)$
We can conclude that
$\rho_j \sim \rho'_j \wedge os \sim_{pt,pt'} os' \wedge os_j = f(v) :: os \wedge os'_j = f(v') :: os'$
$\wedge sl_{f(v)} > sl_{obs} \wedge sl_{f(v')} > sl_{obs}$
Using the definition for indistinguishable operand stacks, we can conclude that
$\rho_j \sim \rho'_j \wedge f(v) :: os \sim_{(sl_{f(v)},\bot)::pt,(sl_{f(v')},\bot)::pt'} f(v') :: os' \wedge$
$os_j = f(v) :: os \wedge os'_j = f(v') :: os'$
In consequence, we know that
$\rho_j \sim \rho'_j \wedge os_j \sim_{(sl_{f(v)},\bot)::pt,(sl_{f(v')},\bot)::pt'} os'_j$
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
$\langle i, \rho_i, os_i \rangle \rightsquigarrow_v \langle j, \rho_j, os_j \rangle \wedge \langle i, \rho'_i, os'_i \rangle \rightsquigarrow_v \langle j, \rho'_j, os'_j \rangle \wedge \langle i, \rho_i, os_i \rangle \sim \langle i, \rho'_i, os'_i \rangle$
$\Rightarrow \langle j, \rho_j, os_j \rangle \sim \langle j, \rho'_j, os'_j \rangle$ is fulfiled.

□

Proof: storelocal x

Using the semantic of the instruction, we know that
$\rho_i \sim \rho'_i \wedge \rho_i \oplus \{x \mapsto v\} = \rho_j \wedge \rho'_i \oplus \{x \mapsto v'\} = \rho'_j \wedge os_i \sim_{(sl_v,pl_v)::pt,(sl'_v,pl'_v)::pt'} os'_i$
$\wedge os_i = v :: os \wedge os'_i = v' :: os' \wedge os_j = os \wedge os'_j = os'$ is fulfiled.
And we can conclude that
$\rho_i \sim \rho'_i \wedge \rho_i \oplus \{x \mapsto v\} = \rho_j \wedge \rho'_i \oplus \{x \mapsto v'\} = \rho'_j$
$\wedge v :: os \sim_{(sl_v,pl_v)::pt,(sl'_v,pl'_v)::pt'} v' :: os' \wedge os_j = os \wedge os'_j = os'$
We have to distinguish two cases $sl_v \leq sl_{obs} \wedge sl'_v \leq sl_{obs}$ and $sl_v > sl_{obs} \wedge sl'_v > sl_{obs}$

1. case: $sl_v \leq sl_{obs} \wedge sl'_v \leq sl_{obs}$
Using the definition for indistinguishable operand stacks, we know that
$\rho_i \sim \rho'_i \wedge \rho_i \oplus \{x \mapsto v\} = \rho_j \wedge \rho'_i \oplus \{x \mapsto v'\} = \rho'_j \wedge os \sim_{pt,pt'} os'$
$\wedge os_j = os \wedge os'_j = os' \wedge v = v'$
We can conclude that
$\rho_i \sim \rho'_i \wedge \rho_i \oplus \{x \mapsto v\} = \rho_j \wedge \rho'_i \oplus \{x \mapsto v'\} = \rho'_j \wedge os_j \sim_{pt,pt'} os'_j \wedge v = v'$
Using the defintion for indistinguishable local mappings, we know that
$\rho_i \oplus \{x \mapsto v\} \sim \rho'_i \oplus \{x \mapsto v'\} \wedge \rho_i \oplus \{x \mapsto v\} = \rho_j \wedge \rho'_i \oplus \{x \mapsto v'\} = \rho'_j \wedge os_j \sim_{pt,pt'} os'_j$
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
$\langle i, \rho_i, os_i \rangle \rightsquigarrow_v \langle j, \rho_j, os_j \rangle \wedge \langle i, \rho'_i, os'_i \rangle \rightsquigarrow_v \langle j, \rho'_j, os'_j \rangle \wedge \langle i, \rho_i, os_i \rangle \sim \langle i, \rho'_i, os'_i \rangle$
$\Rightarrow \langle j, \rho_j, os_j \rangle \sim \langle j, \rho'_j, os'_j \rangle$ is fulfiled.

2. case: $sl_v > sl_{obs} \wedge sl'_v > sl_{obs}$
Using the abstract transfer rule, we know that
$\rho_i \sim \rho'_i \wedge \rho_i \oplus \{x \mapsto v\} = \rho_j \wedge \rho'_i \oplus \{x \mapsto v'\} = \rho'_j \wedge v :: os \sim_{(sl_v,pl_v)::pt,(sl'_v,pl'_v)::pt'} v' ::$
$os' \wedge os_j = os \wedge os'_j = os' \wedge SL(x) \geq sl_v \sqcup se(i) > sl_{obs} \wedge SL(x) \geq sl'_v \sqcup se(i) > sl_{obs}$
Using the definition of indistinguishable local mappings, we can conclude that
$\rho_i \oplus \{x \mapsto v\} \sim \rho'_i \oplus \{x \mapsto v'\} \wedge \rho_i \oplus \{x \mapsto v\} = \rho_j \wedge \rho'_i \oplus \{x \mapsto v'\} = \rho'_j \wedge v ::$
$os \sim_{(sl_v,pl_v)::pt,(sl'_v,pl'_v)::pt'} v' :: os' \wedge os_j = os \wedge os'_j = os'$
We can conclude that
$\rho_j \sim \rho'_j \wedge v :: os \sim_{(sl_v,pl_v)::pt,(sl'_v,pl'_v)::pt'} v' :: os' \wedge os_j = os \wedge os'_j = os'$
Using the definition for indistinguishable operand stacks, we know that
$\rho_j \sim \rho'_j \wedge os \sim_{pt,pt'} os' \wedge os_j = os \wedge os'_j = os'$
And now 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
$\langle j, \rho_j, os_j \rangle \sim \langle j, \rho'_j, os'_j \rangle$
Finally, we know that
$\langle i, \rho_i, os_i \rangle \rightsquigarrow_v \langle j, \rho_j, os_j \rangle \wedge \langle i, \rho'_i, os'_i \rangle \rightsquigarrow_v \langle j, \rho'_j, os'_j \rangle \wedge \langle i, \rho_i, os_i \rangle \sim \langle i, \rho'_i, os'_i \rangle$
$\Rightarrow \langle j, \rho_j, os_j \rangle \sim \langle j, \rho'_j, os'_j \rangle$ is fulfiled.

□

Proof: loadlocaladdress x

Using the semantic of the instruction, we know that
$\rho_i \sim \rho'_i \wedge \rho_i = \rho_j \wedge \rho'_i = \rho'_j \wedge os_i \sim_{pt,pt'} os'_i \wedge os_j = \&(x) :: os_i \wedge os'_j = \&(x) :: os'_i$ is fulfiled.
And we can conclude that
$\rho_j \sim \rho'_j \wedge os_i \sim_{pt,pt'} os'_i \wedge os_j = \&(x) :: os_i \wedge os'_j = \&(x) :: os'_i$
Using the definition for indistinguishable operand stacks, we know that
$\rho_j \sim \rho'_j \wedge \&(x) :: os_i \sim_{(sl_v,\bot)::pt,(sl'_v,\bot)::pt'} \&(x) :: os'_i \wedge os_j = \&(x) :: os_i \wedge os'_j = \&(x) :: os'_i$
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
$\langle i, \rho_i, os_i\rangle \rightsquigarrow_v \langle j, \rho_j, os_j\rangle \wedge \langle i, \rho'_i, os'_i\rangle \rightsquigarrow_v \langle j, \rho'_j, os'_j\rangle \wedge \langle i, \rho_i, os_i\rangle \sim \langle i, \rho'_i, os'_i\rangle$
$\Rightarrow \langle j, \rho_j, os_j\rangle \sim \langle j, \rho'_j, os'_j\rangle$ is fulfiled.

□

Proof: loadindirect

Using the semantic of the instruction, we know that
$\rho_i \sim \rho'_i \wedge \rho_i = \rho_j \wedge \rho'_i = \rho'_j \wedge os_i \sim_{(sl_{\&(x)},pl_{\&(x)})::pt,(sl'_{\&(x)},pl'_{\&(x)})::pt'} os'_i$
$\wedge os_i = \&(x) :: os \wedge os'_i = \&(x) :: os' \wedge os_j = \rho_i(x) :: os \wedge os'_j = \rho'_i(x) :: os'$ is fulfiled.
And we can conclude that
$\rho_j \sim \rho'_j \wedge os_i \sim_{(sl_{\&(x)},pl_{\&(x)})::pt,(sl'_{\&(x)},pl'_{\&(x)})::pt'} os'_i$
$\wedge os_i = \&(x) :: os \wedge os'_i = \&(x) :: os' \wedge os_j = \rho_i(x) :: os \wedge os'_j = \rho'_i(x) :: os'$
We have to distinguish two cases. Either all information is visible $sl_{\&(x)} \leq sl_{obs} \wedge sl'_{\&(x)} \leq sl_{obs} \wedge pl_{\&(x)} \leq sl_{obs} \wedge pl'_{\&(x)} \leq sl_{obs}$ or either the address is invisible or the ressource that is referenced by the address $sl_{\&(x)} > sl_{obs} \wedge sl'_{\&(x)} > sl_{obs} \wedge pl_{\&(x)} > sl_{obs} \wedge pl'_{\&(x)} > sl_{obs}$.

1. case: $sl_{\&(x)} \leq sl_{obs} \wedge sl'_{\&(x)} \leq sl_{obs} \wedge pl_{\&(x)} \leq sl_{obs} \wedge pl'_{\&(x)} \leq sl_{obs}$
Using the definition of indistinguishable local mappings, we know that
$\rho_j \sim \rho'_j \wedge os_i \sim_{(sl_{\&(x)},pl_{\&(x)})::pt,(sl'_{\&(x)},pl'_{\&(x)})::pt'} os'_i \wedge os_i = \&(x) :: os$
$\wedge os'_i = \&(x) :: os' \wedge os_j = \rho_i(x) :: os \wedge os'_j = \rho'_i(x) :: os' \wedge \rho_i(x) = \rho'_i(x)$
Using the definition of indistinguishable operand stacks, we can conclude
$\rho_j \sim \rho'_j \wedge os \sim_{pt,pt'} os' \wedge os_j = \rho_i(x) :: os \wedge os'_j = \rho'_i(x) :: os' \wedge \rho_i(x) = \rho'_i(x)$
Using the definition for indistinguishable operand stacks, we know that
$\rho_j \sim \rho'_j \wedge \rho_i(x) :: os \sim_{(sl_{\&(x)} \sqcup pl_{\&(x)} \sqcup se(i),\bot)::pt,(sl'_{\&(x)} \sqcup pl'_{\&(x)} \sqcup se(i),\bot)::pt'} \rho'_i(x) :: os'_i$
$\wedge os_j = \&(x) :: os_i \wedge os'_j = \&(x) :: os'_i$
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
$\langle i, \rho_i, os_i\rangle \rightsquigarrow_v \langle j, \rho_j, os_j\rangle \wedge \langle i, \rho'_i, os'_i\rangle \rightsquigarrow_v \langle j, \rho'_j, os'_j\rangle \wedge \langle i, \rho_i, os_i\rangle \sim \langle i, \rho'_i, os'_i\rangle$

$\Rightarrow \langle j, \rho_j, os_j\rangle \sim \langle j, \rho'_j, os'_j\rangle$ is fulfiled.

2. case: $sl_{\&(x)} > sl_{obs} \wedge sl'_{\&(x)} > sl_{obs} \vee pl_{\&(x)} > sl_{obs} \wedge pl'_{\&(x)} > sl_{obs}$
Using the definition of indistinguishable operand stacks, we can conclude
$\rho_j \sim \rho'_j \wedge os \sim_{pt,pt'} os' \wedge os_j = \rho_i(x) :: os \wedge os'_j = \rho'_i(x) :: os'$
$\wedge(sl_{\&(x)} > sl_{obs} \wedge sl'_{\&(x)} > sl_{obs} \vee pl_{\&(x)} > sl_{obs} \wedge pl'_{\&(x)} > sl_{obs})$
We can conclude that
$\rho_j \sim \rho'_j \wedge os \sim_{pt,pt'} os' \wedge os_j = \rho_i(x) :: os \wedge os'_j = \rho'_i(x) :: os'$
$\wedge sl_{\&(x)} \sqcup pl_{\&(x)} \sqcup se(i) > sl_{obs} \wedge sl'_{\&(x)} \sqcup pl'_{\&(x)} \sqcup se'(i) > sl_{obs}$
Using the definition for indistinguishable operand stacks, we know that
$\rho_j \sim \rho'_j \wedge \rho_i(x) :: os \sim_{(sl_{\&(x)} \sqcup pl_{\&(x)} \sqcup se(i), \bot)::pt, (sl'_{\&(x)} \sqcup pl'_{\&(x)} \sqcup se'(i), \bot)::pt'} \rho'_i :: os'$
$\wedge os_j = \rho_i(x) :: os \wedge os'_j = \rho'_i(x) :: os'$
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
$\langle i, \rho_i, os_i\rangle \rightsquigarrow_v \langle j, \rho_j, os_j\rangle \wedge \langle i, \rho'_i, os'_i\rangle \rightsquigarrow_v \langle j, \rho'_j, os'_j\rangle \wedge \langle i, \rho_i, os_i\rangle \sim \langle i, \rho'_i, os'_i\rangle$
$\Rightarrow \langle j, \rho_j, os_j\rangle \sim \langle j, \rho'_j, os'_j\rangle$ is fulfiled.

□

Proof: storeindirect

Using the semantic of the instruction, we know that
$\rho_i \sim \rho'_i \wedge os_i \sim_{(sl_v, pl_v)::(sl_{\&(x)}, pl_{\&(x)})::pt, (sl'_v, pl'_v)::(sl_{\&(x')}, pl_{\&(x')})::pt'} os'_i$
$\wedge \rho_i \oplus \{x \mapsto v\} = \rho_j \wedge \rho'_i \oplus \{x' \mapsto v'\} = \rho'_j$
$\wedge os_i = v :: \&(x) :: os \wedge os'_i = v' :: \&(x') :: os' \wedge os_j = os \wedge os'_j = os'$ is fulfiled.
And we can conclude that
$\rho_i \sim \rho'_i \wedge v :: \&(x) :: os \sim_{(sl_v, pl_v)::(sl_{\&(x)}, pl_{\&(x)})::pt, (sl'_v, pl'_v)::(sl_{\&(x')}, pl_{\&(x')})::pt'} v' :: \&(x') :: os'$
$\wedge \rho_i \oplus \{x \mapsto v\} = \rho_j \wedge \rho'_i \oplus \{x' \mapsto v'\} = \rho'_j \wedge os_j = os \wedge os'_j = os'$
We have to distinguish two cases. Either all information on the stack is from visible ressources $sl_v \leq sl_{obs} \wedge sl'_v \leq sl_{obs} \wedge sl_{\&(x)} \leq sl_{obs} \wedge sl_{\&(x')} \leq sl_{obs}$ or one of the information ressources is invisible $sl_v > sl_{obs} \wedge sl'_v > sl_{obs} \vee sl_{\&(x)} > sl_{obs} \wedge sl_{\&(x')} > sl_{obs}$

1. case: $sl_v \leq sl_{obs} \wedge sl'_v \leq sl_{obs} \wedge sl_{\&(x)} \leq sl_{obs} \wedge sl_{\&(x')} \leq sl_{obs}$
Using the definition for indistinguishable operand stacks, we know that
$\rho_i \sim \rho'_i \wedge v :: \&(x) :: os \sim_{(sl_v, pl_v)::(sl_{\&(x)}, pl_{\&(x)})::pt, (sl'_v, pl'_v)::(sl_{\&(x')}, pl_{\&(x')})::pt'} v' :: \&(x') :: os'$
$\wedge \rho_i \oplus \{x \mapsto v\} = \rho_j \wedge \rho'_i \oplus \{x' \mapsto v'\} = \rho'_j \wedge os_j = os \wedge os'_j = os' \wedge v = v' \wedge \&(x) = \&(x')$
And we can conclude that
$\rho_i \sim \rho'_i \wedge \rho_i \oplus \{x \mapsto v\} = \rho_j \wedge \rho'_i \oplus \{x' \mapsto v'\} = \rho'_j \wedge os_j \sim_{pt,pt'} os'_j \wedge v = v' \wedge \&(x) = \&(x')$
Using the defintion for indistinguishable local mappings, we know that
$\rho_i \oplus \{x \mapsto v\} \sim \rho'_i \oplus \{x' \mapsto v'\} \wedge \rho_i \oplus \{x \mapsto v\} = \rho_j \wedge \rho'_i \oplus \{x' \mapsto v'\} = \rho'_j \wedge os_j \sim_{pt,pt'} os'_j$
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
$\langle i, \rho_i, os_i\rangle \rightsquigarrow_v \langle j, \rho_j, os_j\rangle \wedge \langle i, \rho'_i, os'_i\rangle \rightsquigarrow_v \langle j, \rho'_j, os'_j\rangle \wedge \langle i, \rho_i, os_i\rangle \sim \langle i, \rho'_i, os'_i\rangle$
$\Rightarrow \langle j, \rho_j, os_j\rangle \sim \langle j, \rho'_j, os'_j\rangle$ is fulfiled.

2. case: $sl_v > sl_{obs} \wedge sl'_v > sl_{obs} \vee sl_{\&(x)} > sl_{obs} \wedge sl_{\&(x')} > sl_{obs}$
Using the abstract transfer rule, we know that
$\rho_i \sim \rho'_i \wedge v :: \&(x) :: os \sim_{(sl_v,pl_v)::(sl_{\&(x)},pl_{\&(x)})::pt,(sl'_v,pl'_v)::(sl_{\&(x')},pl_{\&(x')})::pt'} v' :: \&(x') :: os'$
$\wedge \rho_i \oplus \{x \mapsto v\} = \rho_j \wedge \rho'_i \oplus \{x' \mapsto v'\} = \rho'_j \wedge os_j = os \wedge os'_j = os'$
$\wedge SL(x) = pl_{\&(x)} \geq sl_v \sqcup sl_{\&(x)} \sqcup se(i) > sl_{obs}$
$\wedge SL(x) = pl_{\&(x')} \geq sl'_v \sqcup sl_{\&(x')} \sqcup se'(i) > sl_{obs}$
Using the definition of indistinguishable local mappings, we can conclude that
$\rho_i \oplus \{x \mapsto v\} \sim \rho'_i \oplus \{x' \mapsto v'\} \wedge \rho_i \oplus \{x \mapsto v\} = \rho_j \wedge \rho'_i \oplus \{x' \mapsto v'\} = \rho'_j \wedge os_j = os$
$\wedge os'_j = os' \wedge v :: \&(x) :: os \sim_{(sl_v,pl_v)::(sl_{\&(x)},pl_{\&(x)})::pt,(sl'_v,pl'_v)::(sl_{\&(x')},pl_{\&(x')})::pt'} v' :: \&(x') :: os'$
We can conclude that
$\rho_j \sim \rho'_j \wedge os_j = os \wedge os'_j = os'$
$\wedge v :: \&(x) :: os \sim_{(sl_v,pl_v)::(sl_{\&(x)},pl_{\&(x)})::pt,(sl'_v,pl'_v)::(sl_{\&(x')},pl_{\&(x')})::pt'} v' :: \&(x') :: os'$
Using the definition for indistinguishable operand stacks, we know that
$\rho_j \sim \rho'_j \wedge os \sim_{pt,pt'} os' \wedge os_j = os \wedge os'_j = os'$
And now 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
$\langle i, \rho_i, os_i\rangle \rightsquigarrow_v \langle j, \rho_j, os_j\rangle \wedge \langle i, \rho'_i, os'_i\rangle \rightsquigarrow_v \langle j, \rho'_j, os'_j\rangle \wedge \langle i, \rho_i, os_i\rangle \sim \langle i, \rho'_i, os'_i\rangle$
$\Rightarrow \langle j, \rho_j, os_j\rangle \sim \langle j, \rho'_j, os'_j\rangle$ is fulfiled.

□

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 $\langle i, \rho_i, os_i\rangle \sim \langle i, \rho'_i, os'_i\rangle$ be two indistinguishable states and $\langle i, \rho_i, os_i\rangle \rightsquigarrow^*_v \rho_r, v$, $\langle i, \rho'_i, os'_i\rangle \rightsquigarrow^*_v \rho'_r, v'$ be the execution paths.
Now we need to show that $\rho_r \sim \rho'_r \wedge r = r'$ is fulfiled.

Induction base:
Let $n = 1$ be the amount of virtual steps before termination of the program. As a result, the execution paths must be:
$\langle i, \rho_i, os_i\rangle \rightsquigarrow_v \rho_r, v \wedge \langle i, \rho'_i, os'_i\rangle \rightsquigarrow_v \rho'_r, v'$
We know that "return" is the only instruction that can terminate a program. Using the semantic of "return", we know $\langle r, \rho_r, v :: os_r\rangle \rightsquigarrow \rho_r, v$. 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:
$\langle i, \rho_r, v :: os_r\rangle \rightsquigarrow \rho_r, v \wedge \langle i, \rho'_r, v' :: os'_r\rangle \rightsquigarrow \rho'_r, v'$
Using the definition of indistinguishable states, we can conclude that:

$\rho_r \sim \rho'_r \wedge v :: os_r \sim_{(sl,pl)::pt,(sl',pl')::pt'} v' :: os'$
From the condtition $sl \sqcup se(i) = lsl$ of the abstract transfer rule for "return", we know that sl must be the least security level in the flow policy and, in consequence, any observer may see it, hence $sl \leq sl_o bs$. Using the defintion of indistinguishable operand stacks, we can conclude:
$\rho_r \sim \rho'_r \wedge os_r \sim_{pt,pt'} os' \wedge v = v'$
Finally we know that
$\rho_r \sim \rho'_r \wedge v = v'$ is fulfiled.

Induction requirement:

Any execution with n states before termination fulfils:
$\langle i, \rho_i, os_i\rangle \sim \langle i, \rho'_i, os'_i\rangle \wedge \langle i, \rho_i, os_i\rangle \rightsquigarrow^n \rho_r, v \wedge \langle i, \rho'_i, os'_i\rangle \rightsquigarrow^n \rho'_r, v' \Rightarrow \rho_r \sim \rho'_r \wedge v = v'$

Induction step:

Let the execution path have $n+1$ states. The execution paths can be written as:
$\langle i, \rho_i, os_i\rangle \rightsquigarrow \langle j, \rho_j, os_j\rangle \rightsquigarrow^n \rho_r, v \wedge \langle i, \rho'_i, os'_i\rangle \rightsquigarrow \langle j, \rho'_j, os'_j\rangle \rightsquigarrow^n \rho'_r, v'$
Using the lemma "Virtual Steps Preserve Indistinguishability of States", we can conclude that
$\langle i, \rho_i, os_i\rangle \sim \langle i, \rho'_i, os'_i\rangle \wedge \langle i, \rho_i, os_i\rangle \rightsquigarrow \langle j, \rho_j, os_j\rangle \rightsquigarrow^n \rho_r, v$
$\wedge \langle i, \rho'_i, os'_i\rangle \rightsquigarrow \langle j, \rho'_j, os'_j\rangle \rightsquigarrow^n \rho'_r, v' \Rightarrow \langle j, \rho_j, os_j\rangle \sim \langle j, \rho'_j, os'_j\rangle$
is fulfiled. Using the induction requirement, we can conclude that
$\langle i, \rho_i, os_i\rangle \sim \langle i, \rho'_i, os'_i\rangle \wedge \langle i, \rho_i, os_i\rangle \rightsquigarrow \langle j, \rho_j, os_j\rangle \rightsquigarrow^n \rho_r, v$
$\wedge \langle i, \rho'_i, os'_i\rangle \rightsquigarrow \langle j, \rho'_j, os'_j\rangle \rightsquigarrow^n \rho'_r, v' \Rightarrow \langle j, \rho_j, os_j\rangle \sim \langle j, \rho'_j, os'_j\rangle \Rightarrow \rho_r \sim \rho'_r \wedge v = v'$
is fulfiled. And in consequence
$\langle i, \rho_i, os_i\rangle \sim \langle i, \rho'_i, os'_i\rangle \wedge \langle i, \rho_i, os_i\rangle \rightsquigarrow \langle j, \rho_j, os_j\rangle \rightsquigarrow^n \rho_r, v \wedge \langle i, \rho'_i, os'_i\rangle \rightsquigarrow \langle j, \rho'_j, os'_j\rangle \rightsquigarrow^n \rho'_r, v' \Rightarrow \rho_r \sim \rho'_r \wedge v = v'$
is fulfiled.
As a result, a typable program is non-interfering.

□

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 ressources. 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 dependecy 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

[BPR07] G. Barthe, D. Pichardie, and T. Rezk. A Certified Lightweight Non-interference Java Bytecode Verifier. In *Proceedings of 16th European Symposium on Programming*, volume 4421 of *LNCS*, pages 125–140. Springer, 2007.

[BR05] G. Barthe and T. Rezk. Non-interference for a JVM-like language. In *Proceedings of the ACM SIGPLAN international Workshop on Types in Languages Design and Implementation*, pages 103–112, New York, NY, USA, 2005. ACM Press.

[ECM06] ECMA International. Standard ECMA-335: Common Language Infrastructure (CLI). http://www.ecma-international.org/publications/standards/Ecma-335.htm, 2006.

[LY99] T. Lindholm and F. Yellin. *The Java™ Virtual Machine Specification. Second Edition.* Sun Microsystems, Inc., 1999. http://java.sun.com/docs/books/vmspec/.

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 $l := h$ or, more subtly, in **if** $h = 1$ **then** $l := 1$ **else** $l := 0$, where one can draw conclusions on the (secret) value of the high variable h only by observing the (non-secret) value of the low variable l. Recently, so called *Security Typed* Systems have been developed for mechanizing information flow analyzes. For instance, the rules for typing assignments and conditionals read as follows:

$$\frac{exp : \mathrm{low}}{\vdash l := exp} \qquad \frac{exp : \mathrm{low} \quad \vdash C_1 \quad \vdash C_2}{\vdash \textbf{if } exp \textbf{ then } C_1 \textbf{ else } C_2}$$

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 often is code 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 as 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. Manged 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.

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 decision 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

[BPR07] G. Barthe, D. Pichardie, and T. Rezk. A Certified Lightweight Non-interference Java Bytecode Verifier. In *Proceedings of 16th European Symposium on Programming*, volume 4421 of *LNCS*, pages 125–140. Springer, 2007.

[BR05] G. Barthe and T. Rezk. Non-interference for a JVM-like language. In *Proceedings of the ACM SIGPLAN international Workshop on Types in Languages Design and Implementation*, pages 103–112, New York, NY, USA, 2005. ACM Press.

[ECM06] ECMA International. Standard ECMA-335: Common Language Infrastructure (CLI). http://www.ecma-international.org/publications/standards/Ecma-335.htm, 2006.

[LY99] T. Lindholm and F. Yellin. *The Java™ Virtual Machine Specification. Second Edition*. Sun Microsystems, Inc., 1999. http://java.sun.com/docs/books/vmspec/.

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.

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 assemlbies, “PNIC” prompts for an assembly file to load.
It is possible to unload assemblies by pressing “U”. When unloading assemlbies, “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 destinantion 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 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 intialize the settings file using “PNIC” and configure the settings using the text editor.
An example settings file could look like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<analysis>
 <assembly file="Application.exe">
  <type id="SomeClass">
   <method id="methodToCheck(System.Int32)">
    <argument id="arg_1" level="Low" />
    <variable id="var_0" level="High" />
    <region index="20">12-20</region>
    <junction index="20" instruction="21" />
   </method>
  </type>
 </assembly>
</analysis>
```

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 proof 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.
$\forall i, j, k, l \in PP : l \notin region(i) \wedge k \in region(i) \wedge j = junction(i)$
$\wedge P, \rho \Downarrow \rho_r, v \wedge \langle l, \rho_l, os_l \rangle \rightsquigarrow \langle k, \rho_k, os_k \rangle$
$\Rightarrow \langle l, \rho_l, os_l \rangle \rightsquigarrow \langle k, \rho_k, os_k \rangle \rightsquigarrow^* \langle j, \rho_j, os_j \rangle$

Proof: High Level Regions Converge

Let $i \in BP$ be a branching instruction with a high level guard of the security level $sl_{guard} \in S$. Let $sl_{obs} \in S$ be the security level of an observer and $sl_{obs} < sl_{guard}$. Let $j = junction(i)$, $k \in region(i)$ and $l \notin region(i)$. Let $P, \rho \Downarrow \rho_r, v$ be the complete execution path and let $\langle l, \rho_l, os_l \rangle \rightsquigarrow \langle k, \rho_k, os_k \rangle$.

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 $P, \rho \Downarrow \rho_r, v$ must be $\langle r, \rho_r, v :: os_r \rangle \rightsquigarrow \rho_r, v$.

From the condition $sl \sqcup se(i) = lsl$ of the abstract transfer rule for "return" we can conclude that a typeable program P can only terminate in low regions. In consequence, $r \notin region(i)$.

From requirement 2 of the definition 3.4.1 of "Safe Over Approximation", we know that $\forall m, n, o \in PP : n \in region(m) \wedge n \mapsto o \Rightarrow o \in region(m) \vee o = junction(m)$ is fulfiled. 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 $\forall i, n, j \in PP : n \in region(i) \wedge j = junction(i) \wedge \langle n, \rho_n, os_n \rightsquigarrow^* \rho_r, v \Rightarrow \langle n, \rho_n, os_n \rightsquigarrow^* \langle j, \rho_j, os_j \rangle$ is fulfiled.

Especially, it is fulfiled for every entry point of the region. The entry of a region in an execution path can be written as $l \notin region(i) \wedge k \in region(i) \wedge \langle l, \rho_l, os_l \rangle \rightsquigarrow \langle k, \rho_k, os_k \rangle$. Furthermore, $\langle l, \rho_l, os_l \rangle$ must be reachable from the initial state $\langle 0, \rho, \epsilon \rangle$. This can be written as $\langle 0, \rho, \epsilon \rangle \rightsquigarrow^* \langle l, \rho_l, os_l \rangle$. The resulting execution path is $\langle 0, \rho, \epsilon \rangle \rightsquigarrow^* \langle l, \rho_l, os_l \rangle \rightsquigarrow \langle k, \rho_k, os_k \rangle \rightsquigarrow^* \langle j, \rho_j, os_j \rangle \rightsquigarrow^* \rho_r, v$.

Therefore we can conclude that $\forall i, j, k, l \in PP : l \notin region(i) \wedge k \in region(i) \wedge j = junction(i) \wedge P, \rho \Downarrow \rho_r, v \wedge \langle l, \rho_l, os_l \rangle \rightsquigarrow \langle k, \rho_k, os_k \rangle \Rightarrow \langle l, \rho_l, os_l \rangle \rightsquigarrow \langle k, \rho_k, os_k \rangle \rightsquigarrow^* \langle j, \rho_j, os_j \rangle$ is fulfiled.

□

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.

$$se(i) > sl_{obs} \wedge \langle i, \rho_i, os_i\rangle \rightsquigarrow \langle j, \rho_j, os_j\rangle \Rightarrow \langle i, \rho_i, os_i\rangle \sim \langle j, \rho_j, os_j\rangle$$

Proof: No Visible Changes in High Regions

We need to show that

$$se(i) > sl_{obs} \wedge \langle i, \rho_i, os_i\rangle \rightsquigarrow \langle j, \rho_j, os_j\rangle \Rightarrow \langle i, \rho_i, os_i\rangle \sim \langle j, \rho_j, os_j\rangle$$

is fulfiled for every instruction in $CIL_{pointer}$, except "return", because in a typeable program no "return" can be executed in a high region, due to the requirement $sl_1 \sqcup se(i) = lsl$ of the abstract transfer rule.
Let $\langle i, \rho_i, os_i\rangle \rightsquigarrow \langle j, \rho_j, os_j\rangle$ 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 fulfiled 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 $\rho_i = \rho_j \wedge os_i = v :: os \wedge os_j = f(v) :: os$.
Using the defintion for indistinguishable local mappings, we can conclude that $\rho_i \sim rho_j \wedge os_i = v :: os \wedge os_j = f(v) :: os$.
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 $\rho_i \sim rho_j \wedge os_i \sim_{pt,pt'} os_j$.
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 fulfiled and finally
$se(i) > sl_{obs} \wedge \langle i, \rho_i, os_i\rangle \rightsquigarrow \langle j, \rho_j, os_j\rangle \Rightarrow \langle i, \rho_i, os_i\rangle \sim \langle j, \rho_j, os_j\rangle$ is fulfiled.

2. case: binary op
Using the semantic of the instruction, we know that $\rho_i = \rho_j \wedge os_i = v_1 :: v_2 :: os \wedge os_j = f(v_1, v_2) :: os$.
Using the defintion for indistinguishable local mappings, we can conclude that $\rho_i \sim rho_j \wedge os_i = v_1 :: v_2 :: os \wedge os_j = f(v_1, v_2) :: os$.
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 $sl_{v_1} \sqcup sl_{v_2} \sqcup se(i)$. 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 $\rho_i \sim rho_j \wedge os_i \sim_{pt,pt'} os_j$.
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 fulfiled and finally
$se(i) > sl_{obs} \wedge \langle i, \rho_i, os_i\rangle \rightsquigarrow \langle j, \rho_j, os_j\rangle \Rightarrow \langle i, \rho_i, os_i\rangle \sim \langle j, \rho_j, os_j\rangle$ is fulfiled.

3. case: pop
Using the semantic of the instruction, we know that $\rho_i = \rho_j \wedge os_i = v :: os \wedge os_j = os$.
Using the defintion for indistinguishable local mappings, we can conclude that $\rho_i \sim rho_j \wedge os_i = v :: os \wedge os_j = os$.
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 $\rho_i \sim rho_j \wedge os_i \sim_{pt,pt'} os_j$.
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 fulfiled and finally
$se(i) > sl_{obs} \wedge \langle i, \rho_i, os_i\rangle \rightsquigarrow \langle j, \rho_j, os_j\rangle \Rightarrow \langle i, \rho_i, os_i\rangle \sim \langle j, \rho_j, os_j\rangle$ is fulfiled.

4. case: push v
Using the semantic of the instruction, we know that $\rho_i = \rho_j \wedge os_i = os \wedge os_j = v :: os$.
Using the defintion for indistinguishable local mappings, we can conclude that $\rho_i \sim rho_j \wedge os_i = os \wedge os_j = v :: os$.
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 $se(i)$. In consequence $sl_v > sl_{obs}$.
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 $\rho_i \sim rho_j \wedge os_i \sim_{pt,pt'} os_j$.
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 fulfiled and finally
$se(i) > sl_{obs} \wedge \langle i, \rho_i, os_i\rangle \rightsquigarrow \langle j, \rho_j, os_j\rangle \Rightarrow \langle i, \rho_i, os_i\rangle \sim \langle j, \rho_j, os_j\rangle$ is fulfiled.

5. case: loadlocal x
Using the semantic of the instruction, we know that $\rho_i = \rho_j \wedge os_i = os \wedge os_j = v :: os$.
Using the defintion for indistinguishable local mappings, we can conclude that $\rho_i \sim rho_j \wedge os_i = os \wedge os_j = v :: os$.
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 $SL(X) \sqcup se(i)$. In consequence $sl_v \geq se(i)$ and $sl_v > sl_{obs}$. 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 $\rho_i \sim rho_j \wedge os_i \sim_{pt,pt'} os_j$.
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 fulfiled and finally
$se(i) > sl_{obs} \wedge \langle i, \rho_i, os_i\rangle \rightsquigarrow \langle j, \rho_j, os_j\rangle \Rightarrow \langle i, \rho_i, os_i\rangle \sim \langle j, \rho_j, os_j\rangle$ is fulfiled.

6. case: storelocal x
Using the semantic of the instruction, we know that $\rho_i \oplus \{x \mapsto v\} = \rho_j \wedge os_i = v :: os \wedge os_j = os$.

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 defintion for indistinguishable local mappings, we can conclude that $\rho_i \sim rho_j \wedge os_i = v :: os \wedge os_j = os$.
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 $\rho_i \sim rho_j \wedge os_i \sim_{pt,pt'} os_j$.
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 fulfiled and finally
$se(i) > sl_{obs} \wedge \langle i, \rho_i, os_i\rangle \rightsquigarrow \langle j, \rho_j, os_j\rangle \Rightarrow \langle i, \rho_i, os_i\rangle \sim \langle j, \rho_j, os_j\rangle$ is fulfiled.

7. case: unconditionaljump
Using the semantic of the instruction, we know that $\rho_i = \rho_j \wedge os_i = os_j$.
Using the defintion for indistinguishable local mappings, we can conclude that $\rho_i \sim rho_j \wedge os_i = os_j$.
Using the definition of indistinguishable operand stacks, we can conclude that $\rho_i \sim rho_j \wedge os_i \sim_{pt,pt'} os_j$.
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 fulfiled and finally
$se(i) > sl_{obs} \wedge \langle i, \rho_i, os_i\rangle \rightsquigarrow \langle j, \rho_j, os_j\rangle \Rightarrow \langle i, \rho_i, os_i\rangle \sim \langle j, \rho_j, os_j\rangle$ is fulfiled.

8. case: jumpWith1Argument t
Using the semantic of the instruction, we know that $\rho_i = \rho_j \wedge os_i = v :: os_j$.
Using the defintion for indistinguishable local mappings, we can conclude that $\rho_i \sim rho_j \wedge os_i = v :: os_j$.
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 $\rho_i \sim rho_j \wedge os_i \sim_{pt,pt'} os_j$.
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 fulfiled and finally
$se(i) > sl_{obs} \wedge \langle i, \rho_i, os_i\rangle \rightsquigarrow \langle j, \rho_j, os_j\rangle \Rightarrow \langle i, \rho_i, os_i\rangle \sim \langle j, \rho_j, os_j\rangle$ is fulfiled.

9. case: jumpWith2Arguments t
Using the semantic of the instruction, we know that $\rho_i = \rho_j \wedge os_i = v_1 :: v_2 :: os_j$.
Using the defintion for indistinguishable local mappings, we can conclude that $\rho_i \sim rho_j \wedge os_i = v_1 :: v_2 :: os_j$.
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 $\rho_i \sim rho_j \wedge os_i \sim_{pt,pt'} os_j$.
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 fulfiled and finally
$se(i) > sl_{obs} \wedge \langle i, \rho_i, os_i\rangle \rightsquigarrow \langle j, \rho_j, os_j\rangle \Rightarrow \langle i, \rho_i, os_i\rangle \sim \langle j, \rho_j, os_j\rangle$ is fulfiled.

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

Using the defintion for indistinguishable local mappings, we can conclude that $\rho_i \sim rho_j \wedge os_i = os \wedge os_j = \&(x) :: os$.
We can conclude from the abstract transfer rule that the security level of the new security level $sl_{\&(x))}$ on the security level stack is equal to $se(i)$. In consequence $sl_{\&(x)} > sl_{obs}$. 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 $\rho_i \sim rho_j \wedge os_i \sim_{pt,pt'} os_j$.
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 fulfiled and finally
$se(i) > sl_{obs} \wedge \langle i, \rho_i, os_i\rangle \rightsquigarrow \langle j, \rho_j, os_j\rangle \Rightarrow \langle i, \rho_i, os_i\rangle \sim \langle j, \rho_j, os_j\rangle$ is fulfiled.

11. case: loadindirect
Using the semantic of the instruction, we know that $\rho_i = \rho_j \wedge os_i = \&(x) :: os \wedge os_j = v :: os$.
Using the defintion for indistinguishable local mappings, we can conclude that $\rho_i \sim rho_j \wedge os_i = \&(x) :: os \wedge os_j = v :: os$.
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 $sl_{\&(x)} \sqcup pl_{\&(x)} \sqcup se(i)$. In consequence $sl_v \not\leq sl_{obs}$. 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 $\rho_i \sim rho_j \wedge os_i \sim_{pt,pt'} os_j$.
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 fulfiled and finally
$se(i) > sl_{obs} \wedge \langle i, \rho_i, os_i\rangle \rightsquigarrow \langle j, \rho_j, os_j\rangle \Rightarrow \langle i, \rho_i, os_i\rangle \sim \langle j, \rho_j, os_j\rangle$ is fulfiled.

12. case: storeindirect
Using the semantic of the instruction, we know that $\rho_i \oplus \{x \mapsto v\} = \rho_j \wedge os_i = v :: \&(x) :: os \wedge os_j = os$.
From the abstract transfer rule we know that $pl_{\&(x)} \geq se(i) \sqcup sl_v$. That means $SL(x) \geq se(i) \sqcup sl_v$. We know that $se(i) > sl_{obs}$. In consequence, $SL(x) > sl_{obs}$.
Using the defintion for indistinguishable local mappings, we can conclude that $\rho_i \sim rho_j \wedge os_i = v :: \&(x) :: os \wedge os_j = os$.
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 $\rho_i \sim rho_j \wedge os_i \sim_{pt,pt'} os_j$.
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 fulfiled and finally
$se(i) > sl_{obs} \wedge \langle i, \rho_i, os_i\rangle \rightsquigarrow \langle j, \rho_j, os_j\rangle \Rightarrow \langle i, \rho_i, os_i\rangle \sim \langle j, \rho_j, os_j\rangle$ is fulfiled.

From those 12 cases we can conclude that $se(i) > sl_{obs} \wedge \langle i, \rho_i, os_i\rangle \rightsquigarrow \langle j, \rho_j, os_j\rangle \Rightarrow \langle i, \rho_i, os_i\rangle \sim \langle j, \rho_j, os_j\rangle$ is fulfiled 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.
$se(i) > sl_{obs} \wedge j = junction(m) \langle i, \rho_i, os_i \rangle \rightsquigarrow_v \langle j, \rho_j, os_j \rangle \wedge \langle i, \rho'_i, os'_i \rangle \rightsquigarrow_v \langle j, \rho'_j, os'_j \rangle \wedge \langle i, \rho_i, os_i \rangle \sim \langle i, \rho'_i, os'_i \rangle \Rightarrow \langle j, \rho_j, os_j \rangle \sim \langle j, \rho'_j, os'_j \rangle$

Proof: High Regions Preserve Indistinguishability

Let $\langle i, \rho_i, os_i \rangle \rightsquigarrow_v \langle j, \rho_j, os_j \rangle$, $\langle i, \rho'_i, os'_i \rangle \rightsquigarrow_v \langle j, \rho'_j, os'_j \rangle$ and $\langle i, \rho_i, os_i \rangle \sim \langle i, \rho'_i, os'_i \rangle$.
Furthermore let $j = junction(m)$.
We must prove, that
$se(i) > sl_{obs} \wedge junction(m) = j \wedge i \in region(m) \wedge i' \in region(m) \wedge \langle i, \rho_i, os_i \rangle \rightsquigarrow_v \langle j, \rho_j, os_j \rangle \wedge \langle i, \rho'_i, os'_i \rangle \rightsquigarrow_v \langle j, \rho'_j, os'_j \rangle \wedge \langle i, \rho_i, os_i \rangle \sim \langle i, \rho'_i, os'_i \rangle \Rightarrow \langle j, \rho_j, os_j \rangle \sim \langle j, \rho'_j, os'_j \rangle$
is fulfiled, if m is a branching instruction with high level guard.

Base:
Let $n = 1$ be the amount of steps in the high region.
Let $i \in region(m) \wedge j = junction(m) \wedge i' \in region(m) \wedge \langle i, \rho_i, os_i \rangle \sim \langle i', \rho'_i, os'_i \rangle \wedge \langle i, \rho_i, os_i \rangle \rightsquigarrow \langle j, \rho_j, os_j \rangle \wedge \langle i', \rho'_i, os'_i \rangle \rightsquigarrow \langle j', \rho'_j, os'_j \rangle$
be the transitions.
Using the lemma "No Visible Changes in High Regions", we know that
$se(i) > sl_{obs} \wedge i \in region(m) \wedge j = junction(m) \wedge i' \in region(m) \wedge \langle i, \rho_i, os_i \rangle \sim \langle i', \rho'_i, os'_i \rangle \wedge \langle i, \rho_i, os_i \rangle \rightsquigarrow \langle j, \rho_j, os_j \rangle \wedge \langle i', \rho'_i, os'_i \rangle \rightsquigarrow \langle j', \rho'_j, os'_j \Rightarrow \langle i, \rho_i, os_i \rangle \sim \langle j, \rho_j, os_j \rangle \wedge \langle i', \rho'_i, os'_i \rangle \sim \langle j', \rho'_j, os'_j \rangle$
is fulfiled.
Using the transitivity of indistinguishability, we can conclude that
$se(i) > sl_{obs} \wedge i \in region(m) \wedge j = junction(m) \wedge i' \in region(m) \wedge \langle i, \rho_i, os_i \rangle \sim \langle i', \rho'_i, os'_i \rangle \wedge \langle i, \rho_i, os_i \rangle \rightsquigarrow \langle j, \rho_j, os_j \rangle \wedge \langle i', \rho'_i, os'_i \rangle \rightsquigarrow \langle j', \rho'_j, os'_j \Rightarrow \langle j, \rho_j, os_j \rangle \sim \langle j', \rho'_j, os'_j \rangle$
is fulfiled and finally we know that
$se(i) > sl_{obs} \wedge junction(m) = j \wedge i \in region(m) \wedge i' \in region(m) \wedge \langle i, \rho_i, os_i \rangle \rightsquigarrow_v \langle j, \rho_j, os_j \rangle \wedge \langle i, \rho'_i, os'_i \rangle \rightsquigarrow_v \langle j, \rho'_j, os'_j \rangle \wedge \langle i, \rho_i, os_i \rangle \sim \langle i, \rho'_i, os'_i \rangle \Rightarrow \langle j, \rho_j, os_j \rangle \sim \langle j, \rho'_j, os'_j \rangle$
is fulfiled.

Requirement:
$se(i) > sl_{obs} \wedge junction(m) = j \wedge i \in region(m) \wedge i' \in region(m) \wedge \langle i, \rho_i, os_i \rangle \rightsquigarrow_v \langle j, \rho_j, os_j \rangle \wedge \langle i, \rho'_i, os'_i \rangle \rightsquigarrow_v \langle j, \rho'_j, os'_j \rangle \wedge \langle i, \rho_i, os_i \rangle \sim \langle i, \rho'_i, os'_i \rangle \Rightarrow \langle j, \rho_j, os_j \rangle \sim \langle j, \rho'_j, os'_j \rangle$
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
$i \in region(m) k \in region(m) \wedge j = junction(m) \wedge i' \in region(m) \wedge \langle i, \rho_i, os_i \rangle \sim \langle i', \rho'_i, os'_i \rangle \wedge \langle i, \rho_i, os_i \rangle \rightsquigarrow^n \langle k, \rho_k, os_k \rangle \rightsquigarrow \langle j, \rho_j, os_j \rangle \wedge \langle i', \rho'_i, os'_i \rangle \rightsquigarrow^o \langle j', \rho'_j, os'_j \rangle$
From the induction requirement we can conclude that
$i \in region(m) \wedge k \in region(m) \wedge j = junction(m) \wedge i' \in region(m) \wedge \langle i, \rho_i, os_i \rangle \sim \langle i', \rho'_i, os'_i \rangle \wedge \langle i, \rho_i, os_i \rangle \rightsquigarrow^n \langle k, \rho_k, os_k \rangle \rightsquigarrow \langle j, \rho_j, os_j \rangle \wedge \langle i', \rho'_i, os'_i \rangle \rightsquigarrow^o \langle j', \rho'_j, os'_j \rangle \Rightarrow$

$\langle k, \rho_k, os_k \rangle \sim \langle j', \rho'_j, os'_j \rangle$
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
$i \in region(m) \wedge k \in region(m) \wedge j = junction(m) \wedge i' \in region(m) \wedge \langle i, \rho_i, os_i \rangle \sim \langle i', \rho'_i, os'_i \rangle \wedge \langle i, \rho_i, os_i \rangle \rightsquigarrow^n \langle k, \rho_k, os_k \rangle \rightsquigarrow \langle j, \rho_j, os_j \rangle \wedge \langle i', \rho'_i, os'_i \rangle \rightsquigarrow^o \langle j', \rho'_j, os'_j \rangle \Rightarrow \langle k, \rho_k, os_k \rangle \sim \langle j', \rho'_j, os'_j \rangle \wedge \langle k, \rho_k, os_k \rangle \sim \langle j, \rho_j, os_j \rangle$ is fulfiled.
Using the transitivity of indistinguishable states, we can conclude that
$i \in region(m) \wedge k \in region(m) \wedge j = junction(m) \wedge i' \in region(m) \wedge j' = junction(m) \wedge \langle i, \rho_i, os_i \rangle \sim \langle i', \rho'_i, os'_i \rangle \wedge \langle i, \rho_i, os_i \rangle \rightsquigarrow^n \langle k, \rho_k, os_k \rangle \rightsquigarrow \langle j, \rho_j, os_j \rangle \wedge \langle i', \rho'_i, os'_i \rangle \rightsquigarrow^o \langle j', \rho'_j, os'_j \rangle \Rightarrow \langle j, \rho_j, os_j \rangle \sim \langle j', \rho'_j, os'_j$
Using the definition of "Virtual Steps", we can conclude that
$i \in region(m) \wedge j = junction(m) \wedge i' \in region(m) \wedge j' = junction(m) \wedge \langle i, \rho_i, os_i \rangle \sim \langle i', \rho'_i, os'_i \rangle \wedge \langle i, \rho_i, os_i \rangle \rightsquigarrow_v \rightsquigarrow \langle j, \rho_j, os_j \rangle \wedge \langle i', \rho'_i, os'_i \rangle \rightsquigarrow_v \langle j', \rho'_j, os'_j \rangle \Rightarrow \langle j, \rho_j, os_j \rangle \sim \langle j', \rho'_j, os'_j$ is fulfiled.

□

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:

$$\langle i, \rho_i, os_i\rangle \rightsquigarrow_v \langle j, \rho_j, os_j\rangle \wedge \langle i, \rho'_i, os'_i\rangle \rightsquigarrow_v \langle j, \rho'_j, os'_j\rangle \wedge \langle i, \rho_i, os_i\rangle \sim \langle i, \rho'_i, os'_i\rangle \Rightarrow \langle j, \rho_j, os_j\rangle \sim \langle j, \rho'_j, os'_j\rangle$$

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

$$\langle i, \rho_i, os_i\rangle \rightsquigarrow_v \langle j, \rho_j, os_j\rangle \wedge \langle i, \rho'_i, os'_i\rangle \rightsquigarrow_v \langle j, \rho'_j, os'_j\rangle \wedge \langle i, \rho_i, os_i\rangle \sim \langle i, \rho'_i, os'_i\rangle \Rightarrow \langle j, \rho_j, os_j\rangle \sim \langle j, \rho'_j, os'_j\rangle$$

is fulfiled for every typeable instruction in a low level region and for complete high regions.
Let $\langle i, \rho_i, os_i\rangle \rightsquigarrow_v \langle j, \rho_j, os_j\rangle$, $\langle i, \rho'_i, os'_i\rangle \rightsquigarrow_v \langle j, \rho'_j, os'_j\rangle$ 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
$\rho_i \sim \rho'_i \wedge \rho_i = \rho_j \wedge \rho'_i = \rho'_j \wedge os_i \sim_{(sl_v,pl_v)::pt,(sl'_v,pl'_v)::pt'} os'_i$
$\wedge os_i = v :: os \wedge os'_i = v' :: os' \wedge os_j = f(v) :: os \wedge os'_j = f(v') :: os'$ is fulfiled.
And we can conclude that
$\rho_j \sim \rho'_j \wedge os_i \sim_{(sl_v,pl_v)::pt,(sl'_v,pl'_v)::pt'} os'_i \wedge os_i = v :: os \wedge os_i = v' :: os'$
$\wedge os_j = f(v) :: os \wedge os'_j = f(v') :: os'$
In consequence, we know that
$\rho_j \sim \rho'_j \wedge v :: os \sim_{(sl_v,pl_v)::pt,(sl'_v,pl'_v)::pt'} v' :: os' \wedge os_j = f(v) :: os \wedge os'_j = f(v') :: os'$
From the defintion of indistinguishable operand stacks we know that we have to distinguish two cases. The first $sl_v \leq sl_{obs} \wedge sl'_v \leq sl_{obs}$ that describes that the top value of the stack is visible and the second $sl_v > sl_{obs} \wedge sl'_v > sl_{obs}$ that describes that the top value of the stack is invisible.

1. case: $sl_v \leq sl_{obs} \wedge sl'_v \leq sl_{obs}$
Using the definition for indistinguishable operand stacks, we know that
$\rho_j \sim \rho'_j \wedge v = v' \wedge os \sim_{pt,pt'} os' \wedge os_j = f(v) :: os \wedge os'_j = f(v') :: os'$
And we can conclude that
$\rho_j \sim \rho'_j \wedge v = v' \wedge f(v) = f(v') \wedge os \sim_{pt,pt'} os' \wedge os_j = f(v) :: os \wedge os'_j = f(v') :: os'$
With the defintion of inditinguishable operand stacks, we know that
$\rho_j \sim \rho'_j \wedge f(v) :: os \sim_{(sl_{f(v)},\perp)::pt,(sl_{f(v')},\perp)::pt'} f(v') :: os'$
$\wedge os_j = f(v) :: os \wedge os'_j = f(v') :: os'$
In consequence, we know that

$\rho_j \sim \rho'_j \wedge os_j \sim_{(sl_{f(v)},\bot)::pt,(sl_{f(v')},\bot)::pt'} os'_j$
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
$\langle i, \rho_i, os_i\rangle \rightsquigarrow_v \langle j, \rho_j, os_j\rangle \wedge \langle i, \rho'_i, os'_i\rangle \rightsquigarrow_v \langle j, \rho'_j, os'_j\rangle \wedge \langle i, \rho_i, os_i\rangle \sim \langle i, \rho'_i, os'_i\rangle$
$\Rightarrow \langle j, \rho_j, os_j\rangle \sim \langle j, \rho'_j, os'_j\rangle$ is fulfiled.

2. case: $sl_v > sl_{obs} \wedge sl'_v > sl_{obs}$
Using the definition for indistinguishable operand stacks, we know that
$\rho_j \sim \rho'_j \wedge sl_v > sl_{obs} \wedge sl'_v > sl_{obs} \wedge os \sim_{pt,pt'} os' \wedge os_j = f(v) :: os \wedge os'_j = f(v') :: os'$
Using the abstract transfer rule, we know that
$\rho_j \sim \rho'_j \wedge sl_v > sl_{obs} \wedge sl'_v > sl_{obs} \wedge os \sim_{pt,pt'} os'$
$\wedge os_j = f(v) :: os \wedge os'_j = f(v') :: os' \wedge sl_{f(v)} = sl_v \sqcup se(i) \wedge sl_{f(v')} = sl'_v \sqcup se(i)$
We can conclude that
$\rho_j \sim \rho'_j \wedge os \sim_{pt,pt'} os' \wedge os_j = f(v) :: os \wedge os'_j = f(v') :: os'$
$\wedge sl_{f(v)} > sl_{obs} \wedge sl_{f(v')} > sl_{obs}$
Using the definition for indistinguishable operand stacks, we can conclude that
$\rho_j \sim \rho'_j \wedge f(v) :: os \sim_{(sl_{f(v)},\bot)::pt,(sl_{f(v')},\bot)::pt'} f(v') :: os' \wedge$
$os_j = f(v) :: os \wedge os'_j = f(v') :: os'$
In consequence, we know that
$\rho_j \sim \rho'_j \wedge os_j \sim_{(sl_{f(v)},\bot)::pt,(sl_{f(v')},\bot)::pt'} os'_j$
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
$\langle i, \rho_i, os_i\rangle \rightsquigarrow_v \langle j, \rho_j, os_j\rangle \wedge \langle i, \rho'_i, os'_i\rangle \rightsquigarrow_v \langle j, \rho'_j, os'_j\rangle \wedge \langle i, \rho_i, os_i\rangle \sim \langle i, \rho'_i, os'_i\rangle$
$\Rightarrow \langle j, \rho_j, os_j\rangle \sim \langle j, \rho'_j, os'_j\rangle$ is fulfiled.

□

Proof: binary op
Using the semantic of the instruction, we know that
$\rho_i \sim \rho'_i \wedge \rho_i = \rho_j \wedge \rho'_i = \rho'_j \wedge os_i \sim_{(sl_{v_1},pl_{v_1})::(sl_{v_2},pl_{v_2})::pt,(sl_{v'_1},pl_{v'_1})::(sl_{v'_2},pl_{v'_2})::pt'} os'_i \wedge os_i = v_1 :: v_2 :: os \wedge os_i = v'_1 :: v'_2 :: os' \wedge os_j = f(v_1, v_2) :: os \wedge os'_j = f(v'_2, v'_2) :: os'$ is fulfiled.
And we can conclude that
$\rho_j \sim \rho'_j \wedge os_i \sim_{(sl_{v_1},pl_{v_1})::(sl_{v_2},pl_{v_2})::pt,(sl_{v'_1},pl_{v'_1})::(sl_{v'_2},pl_{v'_2})::pt'} os'_i \wedge os_i = v_1 :: v_2 :: os \wedge os_i = v'_1 :: v'_2 :: os' \wedge os_j = f(v_1, v_2) :: os \wedge os'_j = f(v'_2, v'_2) :: os'$
In consequence,
$\rho_j \sim \rho'_j \wedge v_1 :: v_2 :: os \sim_{(sl_{v_1},pl_{v_1})::(sl_{v_2},pl_{v_2})::pt,(sl_{v'_1},pl_{v'_1})::(sl_{v'_2},pl_{v'_2})::pt'} v'_1 :: v'_2 :: os' \wedge os_j = f(v_1, v_2) :: os \wedge os'_j = f(v'_2, v'_2) :: os'$
From the defintion of indistinguishable operand stacks, we have to distinguish two cases. The first $sl_{v_1} \leq sl_{obs} \wedge sl_{v'_1} \leq sl_{obs} \wedge sl_{v_2} \leq sl_{obs} \wedge sl_{v'_2} \leq sl_{obs}$ that describes that the top values of each stack are visible and the second $sl_{v_1} > sl_{obs} \wedge sl_{v'_1} > sl_{obs} \vee sl_{v_2} > sl_{obs} \wedge sl_{v'_2} > sl_{obs}$ that describes that one of the two top values of each stack is invisible.

1. case: $sl_{v_1} \leq sl_{obs} \wedge sl_{v'_1} \leq sl_{obs} \wedge sl_{v_2} \leq sl_{obs} \wedge sl_{v'_2} \leq sl_{obs}$

Using the definition for indistinguishable operand stacks, we know that
$\rho_j \sim \rho'_j \wedge v_1 = v'_2 \wedge v_2 = v'_2 \wedge os \sim_{pt,pt'} os' \wedge os_j = f(v_1, v_2) :: os \wedge os'_j = f(v'_1, v'_2) :: os'$
And we can conclude that
$\rho_j \sim \rho'_j \wedge f(v_1, v_2) = f(v'_1, v'_2) \wedge os \sim_{pt,pt'} os' \wedge os_j = f(v_1, v_2) :: os \wedge os'_j = f(v'_1, v'_2) :: os'$
With the defintion of inditinguishable operand stacks, we know that
$\rho_j \sim \rho'_j \wedge f(v_1, v_2) :: os \sim_{(sl_{f(v_1,v_2)},\perp)::pt,(sl_{f(v'_1,v'_2)},\perp)::pt'} f(v'_1, v'_2) :: os' \wedge os_j = f(v_1, v_2) :: os \wedge os'_j = f(v'_1, v'_2) :: os'$
In consequence, we know that
$\rho_j \sim \rho'_j \wedge os_j \sim_{(sl_{f(v_1,v_2)},\perp)::pt,(sl_{f(v'_1,v'_2)},\perp)::pt'} os'_j$
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
$\langle i, \rho_i, os_i\rangle \rightsquigarrow_v \langle j, \rho_j, os_j\rangle \wedge \langle i, \rho'_i, os'_i\rangle \rightsquigarrow_v \langle j, \rho'_j, os'_j\rangle \wedge \langle i, \rho_i, os_i\rangle \sim \langle i, \rho'_i, os'_i\rangle \Rightarrow \langle j, \rho_j, os_j\rangle \sim \langle j, \rho'_j, os'_j\rangle$ is fulfiled.

2. case: $sl_{v_1} > sl_{obs} \wedge sl_{v'_1} > sl_{obs} \vee sl_{v_2} > sl_{obs} \wedge sl_{v'_2} > sl_{obs}$
Using the definition for indistinguishable operand stacks, we know that
$\rho_j \sim \rho'_j \wedge (sl_{v_1} > sl_{obs} \wedge sl_{v'_1} > sl_{obs} \vee sl_{v_2} > sl_{obs} \wedge sl_{v'_2} > sl_{obs}) os \sim_{pt,pt'} os' \wedge os_j = f(v_1, v_2) :: os \wedge os'_j = f(v'_1, v'_2) :: os'$
Using the abstract transfer rule, we know that
$\rho_j \sim \rho'_j \wedge (sl_{v_1} > sl_{obs} \wedge sl_{v'_1} > sl_{obs} \vee sl_{v_2} > sl_{obs} \wedge sl_{v'_2} > sl_{obs}) os \sim_{pt,pt'} os' \wedge os_j = f(v_1, v_2) :: os \wedge os'_j = f(v'_1, v'_2) :: os' sl_{f(v_1,v_2)} = sl_{v_1} \sqcup sl_{v_2} \sqcup se(i) \wedge sl_{f(v'_1,v'_2)} = sl_{v'_1} \sqcup sl_{v'_2} \sqcup se(i)$
We can conclude that
$\rho_j \sim \rho'_j \wedge os \sim_{pt,pt'} os' \wedge os_j = f(v_1, v_2) :: os \wedge os'_j = f(v'_1, v'_2) :: os' sl_{f(v_1,v_2)} > sl_{obs} \wedge sl_{f(v'_1,v'_2)} > sl_{obs}$
Using the definition for indistinguishable operand stacks, we can conclude that
$\rho_j \sim \rho'_j \wedge f(v_1, v_2) :: os \sim_{(sl_{f(v_1,v_2)},\perp)::pt,(sl_{f(v'_1,v'_2)},\perp)::pt'} f(v'_1, v'_2) :: os' \wedge os_j = f(v_1, v_2) :: os \wedge os'_j = f(v'_1, v'_2) :: os'$
In consequence, we know that
$\rho_j \sim \rho'_j \wedge os_j \sim_{(sl_{f(v_1,v_2)},\perp)::pt,(sl_{f(v'_1,v'_2)},\perp)::pt'} os'_j$
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
$\langle i, \rho_i, os_i\rangle \rightsquigarrow_v \langle j, \rho_j, os_j\rangle \wedge \langle i, \rho'_i, os'_i\rangle \rightsquigarrow_v \langle j, \rho'_j, os'_j\rangle \wedge \langle i, \rho_i, os_i\rangle \sim \langle i, \rho'_i, os'_i\rangle \Rightarrow \langle j, \rho_j, os_j\rangle \sim \langle j, \rho'_j, os'_j\rangle$ is fulfiled.

□

Proof: pop
Using the semantic of the instruction, we know that
$\rho_i \sim \rho'_i \wedge \rho_i = \rho_j \wedge \rho'_i = \rho'_j \wedge os_i \sim_{(sl_v,pl_v)::pt,(sl'_v,pl'_v)::pt'} os'_i \wedge os_i = v :: os \wedge os_i = v' :: os' \wedge os_j = os \wedge os'_j = os'$ is fulfiled.
And we can conclude that

$\rho_j \sim \rho'_j \wedge os_i \sim_{(sl_v,pl_v)::pt,(sl'_v,pl'_v)::pt'} os'_i \wedge os_i = v :: os \wedge os_i = v' :: os' \wedge os_j = os \wedge os'_j = os'$
In consequence,
$\rho_j \sim \rho'_j \wedge v :: os \sim_{(sl_v,pl_v)::pt,(sl'_v,pl'_v)::pt'} v' :: os' \wedge os_j = f(v) :: os \wedge os'_j = f(v') :: os'$
Using the definition for indistinguishable operand stacks, we know that
$\rho_j \sim \rho'_j \wedge os \sim_{pt,pt'} os' \wedge os_j = os \wedge os'_j = os'$
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
$\langle i, \rho_i, os_i \rangle \rightsquigarrow_v \langle j, \rho_j, os_j \rangle \wedge \langle i, \rho'_i, os'_i \rangle \rightsquigarrow_v \langle j, \rho'_j, os'_j \rangle \wedge \langle i, \rho_i, os_i \rangle \sim \langle i, \rho'_i, os'_i \rangle \Rightarrow \langle j, \rho_j, os_j \rangle \sim \langle j, \rho'_j, os'_j \rangle$ is fulfiled.

□

Proof: push v
Using the semantic of the instruction, we know that
$\rho_i \sim \rho'_i \wedge \rho_i = \rho_j \wedge \rho'_i = \rho'_j \wedge os_i \sim_{pt,pt'} os'_i \wedge os_j = v :: os_i \wedge os'_j = v :: os'_i$ is fulfiled.
And we can conclude that
$\rho_j \sim \rho'_j \wedge os_i \sim_{pt,pt'} os'_i \wedge os_j = v :: os_i \wedge os'_j = v :: os'_i$
Using the definition for indistinguishable operand stacks, we know that
$\rho_j \sim \rho'_j \wedge v :: os_i \sim_{(sl_v,\bot)::pt,(sl'_v,\bot)::pt'} v :: os'_i \wedge os_j = v :: os_i \wedge os'_j = v :: os'_i$
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
$\langle i, \rho_i, os_i \rangle \rightsquigarrow_v \langle j, \rho_j, os_j \rangle \wedge \langle i, \rho'_i, os'_i \rangle \rightsquigarrow_v \langle j, \rho'_j, os'_j \rangle \wedge \langle i, \rho_i, os_i \rangle \sim \langle i, \rho'_i, os'_i \rangle \Rightarrow \langle j, \rho_j, os_j \rangle \sim \langle j, \rho'_j, os'_j \rangle$ is fulfiled.

□

Proof: loadlocal x
Using the semantic of the instruction, we know that
$\rho_i \sim \rho'_i \wedge \rho_i = \rho_j \wedge \rho'_i = \rho'_j \wedge os_i \sim_{pt,pt'} os'_i \wedge os_j = \rho_i(x) :: os_i \wedge os'_j = \rho'_i(x) :: os'_i$ is fulfiled.
And we can conclude that
$\rho_j \sim \rho'_j \wedge os_i \sim_{pt,pt'} os'_i \wedge os_j = \rho_i(x) :: os_i \wedge os'_j = \rho'_i(x) :: os'_i$
We have to distinguish between two cases. Either the information ressource is visible, $SL(x) \leq sl_{obs}$ or the information ressource is invisible $SL(x) > sl_{obs}$.

1. case: $SL(x) \leq sl_obs$
Using the definition of indistinguishable local mappings, we know that
$\rho_j \sim \rho'_j \wedge os_i \sim_{pt,pt'} os'_i \wedge os_j = \rho_i(x) :: os_i \wedge os'_j = \rho'_i(x) :: os'_i \wedge \rho_i(x) = \rho'_i(x)$

Using the definition for indistinguishable operand stacks, we know that
$\rho_j \sim \rho'_j \wedge \rho_i(x) :: os_i \sim_{(SL(x)\sqcup se(i),PL(x))::pt,(SL(x)\sqcup se'(i),PL(x))::pt'} \rho'_i :: os'_i \wedge os_j = \rho_i(x) :: os_i \wedge os'_j = \rho'_i(x) :: os'_i \wedge \rho_i(x) = \rho'_i(x)$
And we can conclude that
$\rho_j \sim \rho'_j \wedge os_j \sim_{(SL(x)\sqcup se(i),PL(x))::pt,(SL(x)\sqcup se'(i),PL(x))::pt'} os'_j$
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
$\langle i, \rho_i, os_i\rangle \rightsquigarrow_v \langle j, \rho_j, os_j\rangle \wedge \langle i, \rho'_i, os'_i\rangle \rightsquigarrow_v \langle j, \rho'_j, os'_j\rangle \wedge \langle i, \rho_i, os_i\rangle \sim \langle i, \rho'_i, os'_i\rangle \Rightarrow \langle j, \rho_j, os_j\rangle \sim \langle j, \rho'_j, os'_j\rangle$ is fulfiled.

2. case: $SL(x) > sl_{obs}$
Using the definition of indistinguishable local mappings, we know that
$\rho_j \sim \rho'_j \wedge os_i \sim_{pt,pt'} os'_i \wedge os_j = \rho_i(x) :: os_i \wedge os'_j = \rho'_i(x) :: os'_i \wedge SL(x) > sl_{obs}$
Using the definition for indistinguishable operand stacks, we know that
$\rho_j \sim \rho'_j \wedge \rho_i(x) :: os_i \sim_{(SL(x)\sqcup se(i),PL(x))::pt,(SL(x)\sqcup se'(i),PL(x))::pt'} \rho'_i :: os'_i \wedge os_j = \rho_i(x) :: os_i \wedge os'_j = \rho'_i(x) :: os'_i \wedge \rho_i(x) = \rho'_i(x)$
And we can conclude that
$\rho_j \sim \rho'_j \wedge os_j \sim_{(SL(x)\sqcup se(i),PL(x))::pt,(SL(x)\sqcup se'(i),PL(x))::pt'} os'_j$
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
$\langle i, \rho_i, os_i\rangle \rightsquigarrow_v \langle j, \rho_j, os_j\rangle \wedge \langle i, \rho'_i, os'_i\rangle \rightsquigarrow_v \langle j, \rho'_j, os'_j\rangle \wedge \langle i, \rho_i, os_i\rangle \sim \langle i, \rho'_i, os'_i\rangle \Rightarrow \langle j, \rho_j, os_j\rangle \sim \langle j, \rho'_j, os'_j\rangle$ is fulfiled.

□

Proof: storelocal x
Using the semantic of the instruction, we know that
$\rho_i \sim \rho'_i \wedge \rho_i \oplus \{x \mapsto v\} = \rho_j \wedge \rho'_i \oplus \{x \mapsto v'\} = \rho'_j \wedge os_i \sim_{(sl_v,pl_v)::pt,(sl'_v,pl'_v)::pt'} os'_i$
$\wedge os_i = v :: os \wedge os'_i = v' :: os' \wedge os_j = os \wedge os'_j = os'$ is fulfiled.
And we can conclude that
$\rho_i \sim \rho'_i \wedge \rho_i \oplus \{x \mapsto v\} = \rho_j \wedge \rho'_i \oplus \{x \mapsto v'\} = \rho'_j$
$\wedge v :: os \sim_{(sl_v,pl_v)::pt,(sl'_v,pl'_v)::pt'} v' :: os' \wedge os_j = os \wedge os'_j = os'$
We have to distinguish two cases $sl_v \leq sl_{obs} \wedge sl'_v \leq sl_{obs}$ and $sl_v > sl_{obs} \wedge sl'_v > sl_{obs}$

1. case: $sl_v \leq sl_{obs} \wedge sl'_v \leq sl_{obs}$
Using the definition for indistinguishable operand stacks, we know that
$\rho_i \sim \rho'_i \wedge \rho_i \oplus \{x \mapsto v\} = \rho_j \wedge \rho'_i \oplus \{x \mapsto v'\} = \rho'_j \wedge os \sim_{pt,pt'} os'$
$\wedge os_j = os \wedge os'_j = os' \wedge v = v'$
We can conclude that
$\rho_i \sim \rho'_i \wedge \rho_i \oplus \{x \mapsto v\} = \rho_j \wedge \rho'_i \oplus \{x \mapsto v'\} = \rho'_j \wedge os_j \sim_{pt,pt'} os'_j \wedge v = v'$
Using the defintion for indistinguishable local mappings, we know that
$\rho_i \oplus \{x \mapsto v\} \sim \rho'_i \oplus \{x \mapsto v'\} \wedge \rho_i \oplus \{x \mapsto v\} = \rho_j \wedge \rho'_i \oplus \{x \mapsto v'\} = \rho'_j \wedge os_j \sim_{pt,pt'} os'_j$
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
$\langle i, \rho_i, os_i\rangle \rightsquigarrow_v \langle j, \rho_j, os_j\rangle \wedge \langle i, \rho'_i, os'_i\rangle \rightsquigarrow_v \langle j, \rho'_j, os'_j\rangle \wedge \langle i, \rho_i, os_i\rangle \sim \langle i, \rho'_i, os'_i\rangle$
$\Rightarrow \langle j, \rho_j, os_j\rangle \sim \langle j, \rho'_j, os'_j\rangle$ is fulfiled.

2. case: $sl_v > sl_{obs} \wedge sl'_v > sl_{obs}$
Using the abstract transfer rule, we know that
$\rho_i \sim \rho'_i \wedge \rho_i \oplus \{x \mapsto v\} = \rho_j \wedge \rho'_i \oplus \{x \mapsto v'\} = \rho'_j \wedge v :: os \sim_{(sl_v,pl_v)::pt,(sl'_v,pl'_v)::pt'} v' :: os' \wedge os_j = os \wedge os'_j = os' \wedge SL(x) \geq sl_v \sqcup se(i) > sl_{obs} \wedge SL(x) \geq sl'_v \sqcup se(i) > sl_{obs}$
Using the definition of indistinguishable local mappings, we can conclude that
$\rho_i \oplus \{x \mapsto v\} \sim \rho'_i \oplus \{x \mapsto v'\} \wedge \rho_i \oplus \{x \mapsto v\} = \rho_j \wedge \rho'_i \oplus \{x \mapsto v'\} = \rho'_j \wedge v :: os \sim_{(sl_v,pl_v)::pt,(sl'_v,pl'_v)::pt'} v' :: os' \wedge os_j = os \wedge os'_j = os'$
We can conclude that
$\rho_j \sim \rho'_j \wedge v :: os \sim_{(sl_v,pl_v)::pt,(sl'_v,pl'_v)::pt'} v' :: os' \wedge os_j = os \wedge os'_j = os'$
Using the definition for indistinguishable operand stacks, we know that
$\rho_j \sim \rho'_j \wedge os \sim_{pt,pt'} os' \wedge os_j = os \wedge os'_j = os'$
And now 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
$\langle j, \rho_j, os_j\rangle \sim \langle j, \rho'_j, os'_j\rangle$
Finally, we know that
$\langle i, \rho_i, os_i\rangle \rightsquigarrow_v \langle j, \rho_j, os_j\rangle \wedge \langle i, \rho'_i, os'_i\rangle \rightsquigarrow_v \langle j, \rho'_j, os'_j\rangle \wedge \langle i, \rho_i, os_i\rangle \sim \langle i, \rho'_i, os'_i\rangle$
$\Rightarrow \langle j, \rho_j, os_j\rangle \sim \langle j, \rho'_j, os'_j\rangle$ is fulfiled.

□

Proof: unconditionalJump t
Using the semantic of the instruction, we know that
$\rho_i \sim \rho'_i \wedge \rho_i = \rho_j \wedge \rho'_i = \rho'_j \wedge os_i \sim_{pt,pt'} os'_i \wedge os_i = os_j \wedge os'_i = os'_j$ is fulfiled.
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
$\langle i, \rho_i, os_i\rangle \rightsquigarrow_v \langle j, \rho_j, os_j\rangle \wedge \langle i, \rho'_i, os'_i\rangle \rightsquigarrow_v \langle j, \rho'_j, os'_j\rangle \wedge \langle i, \rho_i, os_i\rangle \sim \langle i, \rho'_i, os'_i\rangle \Rightarrow \langle j, \rho_j, os_j\rangle \sim \langle j, \rho'_j, os'_j\rangle$ is fulfiled.

□

Proof: jumpWith1Argument t
Using the semantic of the instruction, we know that

$\rho_i \sim \rho'_i \wedge \rho_i = \rho_j \wedge \rho'_i = \rho'_j \wedge os_i \sim_{(sl_v,pl_v)::pt,(sl'_v,pl'_v)::pt'} os'_i \wedge os_i = v :: os \wedge os'_i = v' :: os' \wedge os_j = os \wedge os'_j = os'$ is fulfiled.
We can conclude that
$\rho_j \sim \rho'_j \wedge v :: os \sim_{(sl_v,pl_v)::pt,(sl'_v,pl'_v)::pt'} v' :: os' \wedge os_j = os \wedge os'_j = os'$
Using the definition for indistinguishable operandstacks, we know that
$\rho_j \sim \rho'_j \wedge os \sim_{pt,pt'} os' \wedge os_j = os \wedge os'_j = os'$
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
$\langle i, \rho_i, os_i\rangle \rightsquigarrow_v \langle j, \rho_j, os_j\rangle \wedge \langle i, \rho'_i, os'_i\rangle \rightsquigarrow_v \langle j, \rho'_j, os'_j\rangle \wedge \langle i, \rho_i, os_i\rangle \sim \langle i, \rho'_i, os'_i\rangle \Rightarrow \langle j, \rho_j, os_j\rangle \sim \langle j, \rho'_j, os'_j\rangle$ is fulfiled.

□

Proof: jumpWith2Arguments t
Using the semantic of the instruction, we know that
$\rho_i \sim \rho'_i \wedge \rho_i = \rho_j \wedge \rho'_i = \rho'_j \wedge os_i \sim_{(sl_{v_1},pl_{v_1})::(sl_{v_2},pl_{v_2})::pt,(sl_{v'_1},pl_{v'_1})::(sl_{v'_2},pl_{v'_2})::pt'} os'_i \wedge os_i = v_1 :: v_2 :: os \wedge os'_i = v'_1 :: v'_2 :: os' \wedge os_j = os \wedge os'_j = os'$ is fulfiled.
We can conclude that
$\rho_j \sim \rho'_j \wedge v_1 :: v_2 :: os \sim_{(sl_{v_1},pl_{v_1})::(sl_{v_2},pl_{v_2})::pt,(sl_{v'_1},pl_{v'_1})::(sl_{v'_2},pl_{v'_2})::pt'} v'_1 :: v'_2 :: os' \wedge os_j = os \wedge os'_j = os'$
Using the definition for indistinguishable operandstacks, we know that
$\rho_j \sim \rho'_j \wedge os \sim_{pt,pt'} os' \wedge os_j = os \wedge os'_j = os'$
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
$\langle i, \rho_i, os_i\rangle \rightsquigarrow_v \langle j, \rho_j, os_j\rangle \wedge \langle i, \rho'_i, os'_i\rangle \rightsquigarrow_v \langle j, \rho'_j, os'_j\rangle \wedge \langle i, \rho_i, os_i\rangle \sim \langle i, \rho'_i, os'_i\rangle \Rightarrow \langle j, \rho_j, os_j\rangle \sim \langle j, \rho'_j, os'_j\rangle$ is fulfiled.

□

Proof: loadlocaladdress x
Using the semantic of the instruction, we know that
$\rho_i \sim \rho'_i \wedge \rho_i = \rho_j \wedge \rho'_i = \rho'_j \wedge os_i \sim_{pt,pt'} os'_i \wedge os_j = \&(x) :: os_i \wedge os'_j = \&(x) :: os'_i$ is fulfiled.
And we can conclude that
$\rho_j \sim \rho'_j \wedge os_i \sim_{pt,pt'} os'_i \wedge os_j = \&(x) :: os_i \wedge os'_j = \&(x) :: os'_i$
Using the definition for indistinguishable operand stacks, we know that
$\rho_j \sim \rho'_j \wedge \&(x) :: os_i \sim_{(sl_v,\perp)::pt,(sl'_v,\perp)::pt'} \&(x) :: os'_i \wedge os_j = \&(x) :: os_i \wedge os'_j = \&(x) :: os'_i$

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
$\langle i, \rho_i, os_i\rangle \rightsquigarrow_v \langle j, \rho_j, os_j\rangle \wedge \langle i, \rho'_i, os'_i\rangle \rightsquigarrow_v \langle j, \rho'_j, os'_j\rangle \wedge \langle i, \rho_i, os_i\rangle \sim \langle i, \rho'_i, os'_i\rangle$
$\Rightarrow \langle j, \rho_j, os_j\rangle \sim \langle j, \rho'_j, os'_j\rangle$ is fulfiled.

□

Proof: loadindirect
Using the semantic of the instruction, we know that
$\rho_i \sim \rho'_i \wedge \rho_i = \rho_j \wedge \rho'_i = \rho'_j \wedge os_i \sim_{(sl_{\&(x)},pl_{\&(x)})::pt,(sl'_{\&(x)},pl'_{\&(x)})::pt'} os'_i$
$\wedge os_i = \&(x) :: os \wedge os'_i = \&(x) :: os' \wedge os_j = \rho_i(x) :: os \wedge os'_j = \rho'_i(x) :: os'$ is fulfiled.
And we can conclude that
$\rho_j \sim \rho'_j \wedge os_i \sim_{(sl_{\&(x)},pl_{\&(x)})::pt,(sl'_{\&(x)},pl'_{\&(x)})::pt'} os'_i$
$\wedge os_i = \&(x) :: os \wedge os'_i = \&(x) :: os' \wedge os_j = \rho_i(x) :: os \wedge os'_j = \rho'_i(x) :: os'$
We have to distinguish two cases. Either all information is visible $sl_{\&(x)} \leq sl_{obs} \wedge sl'_{\&(x)} \leq sl_{obs} \wedge pl_{\&(x)} \leq sl_{obs} \wedge pl'_{\&(x)} \leq sl_{obs}$ or either the address is invisible or the ressource that is referenced by the address $sl_{\&(x)} > sl_{obs} \wedge sl'_{\&(x)} > sl_{obs} \wedge pl_{\&(x)} > sl_{obs} \wedge pl'_{\&(x)} > sl_{obs}$.

1. case: $sl_{\&(x)} \leq sl_{obs} \wedge sl'_{\&(x)} \leq sl_{obs} \wedge pl_{\&(x)} \leq sl_{obs} \wedge pl'_{\&(x)} \leq sl_{obs}$
Using the definition of indistinguishable local mappings, we know that
$\rho_j \sim \rho'_j \wedge os_i \sim_{(sl_{\&(x)},pl_{\&(x)})::pt,(sl'_{\&(x)},pl'_{\&(x)})::pt'} os'_i \wedge os_i = \&(x) :: os$
$\wedge os'_i = \&(x) :: os' \wedge os_j = \rho_i(x) :: os \wedge os'_j = \rho'_i(x) :: os' \wedge \rho_i(x) = \rho'_i(x)$
Using the definition of indistinguishable operand stacks, we can conclude
$\rho_j \sim \rho'_j \wedge os \sim_{pt,pt'} os' \wedge os_j = \rho_i(x) :: os \wedge os'_j = \rho'_i(x) :: os' \wedge \rho_i(x) = \rho'_i(x)$
Using the definition for indistinguishable operand stacks, we know that
$\rho_j \sim \rho'_j \wedge \rho_i(x) :: os \sim_{(sl_{\&(x)} \sqcup pl_{\&(x)} \sqcup se(i),\bot)::pt,(sl'_{\&(x)} \sqcup pl'_{\&(x)} \sqcup se(i),\bot)::pt'} \rho'_i(x) :: os'_i$
$\wedge os_j = \&(x) :: os_i \wedge os'_j = \&(x) :: os'_i$
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
$\langle i, \rho_i, os_i\rangle \rightsquigarrow_v \langle j, \rho_j, os_j\rangle \wedge \langle i, \rho'_i, os'_i\rangle \rightsquigarrow_v \langle j, \rho'_j, os'_j\rangle \wedge \langle i, \rho_i, os_i\rangle \sim \langle i, \rho'_i, os'_i\rangle$
$\Rightarrow \langle j, \rho_j, os_j\rangle \sim \langle j, \rho'_j, os'_j\rangle$ is fulfiled.

2. case: $sl_{\&(x)} > sl_{obs} \wedge sl'_{\&(x)} > sl_{obs} \vee pl_{\&(x)} > sl_{obs} \wedge pl'_{\&(x)} > sl_{obs}$
Using the definition of indistinguishable operand stacks, we can conclude
$\rho_j \sim \rho'_j \wedge os \sim_{pt,pt'} os' \wedge os_j = \rho_i(x) :: os \wedge os'_j = \rho'_i(x) :: os'$
$\wedge (sl_{\&(x)} > sl_{obs} \wedge sl'_{\&(x)} > sl_{obs} \vee pl_{\&(x)} > sl_{obs} \wedge pl'_{\&(x)} > sl_{obs})$
We can conclude that
$\rho_j \sim \rho'_j \wedge os \sim_{pt,pt'} os' \wedge os_j = \rho_i(x) :: os \wedge os'_j = \rho'_i(x) :: os'$

$\wedge sl_{\&(x)} \sqcup pl_{\&(x)} \sqcup se(i) > sl_{obs} \wedge sl'_{\&(x)} \sqcup pl'_{\&(x)} \sqcup se'(i) > sl_{obs}$
Using the definition for indistinguishable operand stacks, we know that
$\rho_j \sim \rho'_j \wedge \rho_i(x) :: os \sim_{(sl_{\&(x)} \sqcup pl_{\&(x)} \sqcup se(i), \bot)::pt, (sl'_{\&(x)} \sqcup pl'_{\&(x)} \sqcup se'(i), \bot)::pt'} \rho'_i :: os'$
$\wedge os_j = \rho_i(x) :: os \wedge os'_j = \rho'_i(x) :: os'$
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
$\langle i, \rho_i, os_i \rangle \rightsquigarrow_v \langle j, \rho_j, os_j \rangle \wedge \langle i, \rho'_i, os'_i \rangle \rightsquigarrow_v \langle j, \rho'_j, os'_j \rangle \wedge \langle i, \rho_i, os_i \rangle \sim \langle i, \rho'_i, os'_i \rangle$
$\Rightarrow \langle j, \rho_j, os_j \rangle \sim \langle j, \rho'_j, os'_j \rangle$ is fulfiled.

□

Proof: storeindirect
Using the semantic of the instruction, we know that
$\rho_i \sim \rho'_i \wedge os_i \sim_{(sl_v, pl_v)::(sl_{\&(x)}, pl_{\&(x)})::pt, (sl'_v, pl'_v)::(sl_{\&(x')}, pl_{\&(x')})::pt'} os'_i$
$\wedge \rho_i \oplus \{x \mapsto v\} = \rho_j \wedge \rho'_i \oplus \{x' \mapsto v'\} = \rho'_j$
$\wedge os_i = v :: \&(x) :: os \wedge os'_i = v' :: \&(x') :: os' \wedge os_j = os \wedge os'_j = os'$ is fulfiled.
And we can conclude that
$\rho_i \sim \rho'_i \wedge v :: \&(x) :: os \sim_{(sl_v, pl_v)::(sl_{\&(x)}, pl_{\&(x)})::pt, (sl'_v, pl'_v)::(sl_{\&(x')}, pl_{\&(x')})::pt'} v' :: \&(x') :: os'$
$\wedge \rho_i \oplus \{x \mapsto v\} = \rho_j \wedge \rho'_i \oplus \{x' \mapsto v'\} = \rho'_j \wedge os_j = os \wedge os'_j = os'$
We have to distinguish two cases. Either all information on the stack is from visible ressources $sl_v \leq sl_{obs} \wedge sl'_v \leq sl_{obs} \wedge sl_{\&(x)} \leq sl_{obs} \wedge sl_{\&(x')} \leq sl_{obs}$ or one of the information ressources is invisible $sl_v > sl_{obs} \wedge sl'_v > sl_{obs} \vee sl_{\&(x)} > sl_{obs} \wedge sl_{\&(x')} > sl_{obs}$

1. case: $sl_v \leq sl_{obs} \wedge sl'_v \leq sl_{obs} \wedge sl_{\&(x)} \leq sl_{obs} \wedge sl_{\&(x')} \leq sl_{obs}$
Using the definition for indistinguishable operand stacks, we know that
$\rho_i \sim \rho'_i \wedge v :: \&(x) :: os \sim_{(sl_v, pl_v)::(sl_{\&(x)}, pl_{\&(x)})::pt, (sl'_v, pl'_v)::(sl_{\&(x')}, pl_{\&(x')})::pt'} v' :: \&(x') :: os'$
$\wedge \rho_i \oplus \{x \mapsto v\} = \rho_j \wedge \rho'_i \oplus \{x' \mapsto v'\} = \rho'_j \wedge os_j = os \wedge os'_j = os' \wedge v = v' \wedge \&(x) = \&(x')$
And we can conclude that
$\rho_i \sim \rho'_i \wedge \rho_i \oplus \{x \mapsto v\} = \rho_j \wedge \rho'_i \oplus \{x' \mapsto v'\} = \rho'_j \wedge os_j \sim_{pt,pt'} os'_j \wedge v = v' \wedge \&(x) = \&(x')$
Using the defintion for indistinguishable local mappings, we know that
$\rho_i \oplus \{x \mapsto v\} \sim \rho'_i \oplus \{x' \mapsto v'\} \wedge \rho_i \oplus \{x \mapsto v\} = \rho_j \wedge \rho'_i \oplus \{x' \mapsto v'\} = \rho'_j \wedge os_j \sim_{pt,pt'} os'_j$
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
$\langle i, \rho_i, os_i \rangle \rightsquigarrow_v \langle j, \rho_j, os_j \rangle \wedge \langle i, \rho'_i, os'_i \rangle \rightsquigarrow_v \langle j, \rho'_j, os'_j \rangle \wedge \langle i, \rho_i, os_i \rangle \sim \langle i, \rho'_i, os'_i \rangle$
$\Rightarrow \langle j, \rho_j, os_j \rangle \sim \langle j, \rho'_j, os'_j \rangle$ is fulfiled.

2. case: $sl_v > sl_{obs} \wedge sl'_v > sl_{obs} \vee sl_{\&(x)} > sl_{obs} \wedge sl_{\&(x')} > sl_{obs}$
Using the abstract transfer rule, we know that
$\rho_i \sim \rho'_i \wedge v :: \&(x) :: os \sim_{(sl_v, pl_v)::(sl_{\&(x)}, pl_{\&(x)})::pt, (sl'_v, pl'_v)::(sl_{\&(x')}, pl_{\&(x')})::pt'} v' :: \&(x') :: os'$

$\wedge \rho_i \oplus \{x \mapsto v\} = \rho_j \wedge \rho'_i \oplus \{x' \mapsto v'\} = \rho'_j \wedge os_j = os \wedge os'_j = os'$
$\wedge SL(x) = pl_{\&(x)} \geq sl_v \sqcup sl_{\&(x)} \sqcup se(i) > sl_{obs}$
$\wedge SL(x) = pl_{\&(x')} \geq sl'_v \sqcup sl_{\&(x')} \sqcup se'(i) > sl_{obs}$
Using the definition of indistinguishable local mappings, we can conclude that
$\rho_i \oplus \{x \mapsto v\} \sim \rho'_i \oplus \{x' \mapsto v'\} \wedge \rho_i \oplus \{x \mapsto v\} = \rho_j \wedge \rho'_i \oplus \{x' \mapsto v'\} = \rho'_j \wedge os_j = os$
$\wedge os'_j = os' \wedge v :: \&(x) :: os \sim_{(sl_v,pl_v)::(sl_{\&(x)},pl_{\&(x)})::pt,(sl'_v,pl'_v)::(sl_{\&(x')},pl_{\&(x')})::pt'} v' :: \&(x') :: os'$
We can conclude that
$\rho_j \sim \rho'_j \wedge os_j = os \wedge os'_j = os'$
$\wedge v :: \&(x) :: os \sim_{(sl_v,pl_v)::(sl_{\&(x)},pl_{\&(x)})::pt,(sl'_v,pl'_v)::(sl_{\&(x')},pl_{\&(x')})::pt'} v' :: \&(x') :: os'$
Using the definition for indistinguishable operand stacks, we know that
$\rho_j \sim \rho'_j \wedge os \sim_{pt,pt'} os' \wedge os_j = os \wedge os'_j = os'$
And now 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
$\langle i, \rho_i, os_i \rangle \rightsquigarrow_v \langle j, \rho_j, os_j \rangle \wedge \langle i, \rho'_i, os'_i \rangle \rightsquigarrow_v \langle j, \rho'_j, os'_j \rangle \wedge \langle i, \rho_i, os_i \rangle \sim \langle i, \rho'_i, os'_i \rangle$
$\Rightarrow \langle j, \rho_j, os_j \rangle \sim \langle j, \rho'_j, os'_j \rangle$ is fulfiled.

□

Proof: return
This case is a special case, because we do not have a transfer of the type $\langle i, \rho_i, os_i \rangle \rightsquigarrow \langle j, \rho_j, os_j \rangle$, but $\langle i, \rho_i, os_i \rangle \rightsquigarrow \rho_j, v$
Using the semantic of the instruction, we know that
$\rho_i \sim \rho'_i \wedge \rho_i = \rho_j \wedge \rho'_i = \rho'_j \wedge os_i \sim_{(sl_v,pl_v)::pt,(sl_v,pl_v)::pt'} os'_i \wedge os_i = v :: os \wedge os'_i = v' :: os'$
is fulfiled.
And we can conclude that
$\rho_j \sim \rho'_j \wedge os_i \sim_{(sl_v,pl_v)::pt,(sl_v,pl_v)::pt'} os'_i \wedge os_i = v :: os \wedge os'_i = v' :: os'$
From the abstract transfer rule we know that
$\rho_j \sim \rho'_j \wedge os_i \sim_{(sl_v,pl_v)::pt,(sl_v,pl_v)::pt'} os'_i \wedge os_i = v :: os \wedge os'_i = v' :: os' sl_v = lsl \leq sl_{obs} \wedge sl'_v = lsl \leq sl_{obs}$
Using the defintion for indistinguishable operand stacks, we can conclude
$\rho_j \sim \rho'_j \wedge v = v'$ is fulfiled.

□

Proof: High Region
Using the lemma "High Regions Preserve Indistinguishability", we know that
$se(i) > sl_{obs} \wedge \langle i, \rho_i, os_i \rangle \rightsquigarrow_v \langle j, \rho_j, os_j \rangle \wedge \langle i, \rho'_i, os'_i \rangle \rightsquigarrow_v \langle j, \rho'_j, os'_j \rangle \wedge \langle i, \rho_i, os_i \rangle \sim \langle i, \rho'_i, os'_i \rangle \Rightarrow \langle j, \rho_j, os_j \rangle \sim \langle j, \rho'_j, os'_j \rangle$
In consequence, we can conclude that
$\langle i, \rho_i, os_i \rangle \rightsquigarrow_v \langle j, \rho_j, os_j \rangle \wedge \langle i, \rho'_i, os'_i \rangle \rightsquigarrow_v \langle j, \rho'_j, os'_j \rangle \wedge \langle i, \rho_i, os_i \rangle \sim \langle i, \rho'_i, os'_i \rangle \Rightarrow \langle j, \rho_j, os_j \rangle \sim \langle j, \rho'_j, os'_j \rangle$
is fulfiled.

□

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 und 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

```
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>
  class AbstractState
  {
    public AbstractState()
    {
      state = new Hashtable();
      recentSecurityLevelStack = new Stack<SecurityLevel>();
    }

    /// <summary>
    /// associations of instructions with stack types;
    /// used as a history of already calculated stack types
    /// </summary>
    private Hashtable state;
    private Stack<SecurityLevel> recentSecurityLevelStack;
    public Stack<SecurityLevel> RecentSecurityLevelStack
    {
      get { return recentSecurityLevelStack; }
    }

    /// <summary>
    /// retrieves the stack type associated with an instruction
    /// </summary>
    /// <param name="instruction">the instruction associated</param>
    /// <returns>the stack type associated with the instruction</returns>
    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();
```

```
            stored.Push(securityLevel);
            result.Push(securityLevel);
          }
        }
        return result;
      }

      /// <summary>
      /// associates the stack type with an instruction
      /// </summary>
      /// <param name="instruction">the instruction to associate</param>
      /// <param name="newStack">the stack type to associate</param>
      public void setSecurityLevelStack(Instruction instruction, Stack<SecurityLevel>
          newStack)
      {
        state[instruction] = newStack;
        recentSecurityLevelStack = getSecurityLevelStack(instruction);
      }

      /// <summary>
      /// restores the abstract state that is associated with an instruction
      /// </summary>
      /// <param name="instruction">the instruction associated with the abstract state to
          restore</param>
      public void restoreState(Instruction instruction)
      {
        recentSecurityLevelStack = getSecurityLevelStack(instruction);
      }

      /// <summary>
      /// looks if the abstract state for this instruction has been calculated yet
      /// </summary>
      /// <param name="instruction">the instruction to look for</param>
      /// <returns>true, if abstract state already calculated</returns>
      public bool stateForInstructionExists(Instruction instruction)
      {
        Stack<SecurityLevel> stored = (Stack<SecurityLevel>)state[instruction];
        if (stored != null && stored is Stack<SecurityLevel>)
          return true;
        return false;
      }

      /// <summary>
      /// creates a memberwise clone of this abstract state
      /// </summary>
      /// <returns>AbstractState obbject memberwise clone</returns>
      public AbstractState getClone()
      {
        return (AbstractState) this.MemberwiseClone();
      }
    }
}
```

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()
```

```
{
  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 ressources
/// </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.Append(asmname + "/" + typename + "/" + methodname + "\n");
          foreach(object varname in securityMappings.Keys)
          {
            if (((string) varname).StartsWith(asmname + "/" + typename + "/" +
                methodname))
            {
              string shortname = ((string)varname).Split('/')[((string)varname).Split
                  ('/').Length-1];
              readable.AppendLine("\t" + shortname + ": " + securityMappings[(string)
                  varname]);
            }
          }
          foreach(object region in regions.Keys)
          {
            if (((string) region).StartsWith(asmname + "/" + typename + "/" +
                methodname))
            {
              string shortname = ((string)region).Split(')')[((string)region).Split
                  (')').Length - 1];
              ControlDependencyRegion cdr = (ControlDependencyRegion)regions[(string)
                  region];
              if (cdr == null)
                readable.AppendLine((string)region);
              else
              {
                readable.Append("\tregion(" + shortname + ") = ");
                foreach (int i in cdr.Region)
                {
                  readable.Append(i + " ");
```

```
                    }
                    readable.AppendLine();
                    if (cdr.Junction != -1)
                    {
                      readable.AppendLine("\tjunction(" + shortname + ")" + cdr.Junction)
                          ;
                    }
                  }
                }
              }
            }
          }
        }
        return readable.ToString();
      }
    }

    /// <summary>
    /// the control dependency regions of the programs
    /// </summary>
    private Hashtable regions;

    /// <summary>
    /// adds a method to the list of methods that get checked during an analysis
    /// </summary>
    /// <param name="assemblyname">the containing assembly</param>
    /// <param name="typename">the containing type</param>
    /// <param name="method">the method to analyse</param>
    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);
    }

    /// <summary>
    /// retrieves all assemblies that are marked for analysis
    /// </summary>
    /// <returns>list of assemblies to analyse</returns>
    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;
    }

    /// <summary>
    /// retrieves all types in an assembly that are marked for analysis
    /// </summary>
    /// <param name="assemblyname">the containing assembly</param>
    /// <returns>list of all types to analyse</returns>
    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;
```

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

  /// <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 analyse</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>();
  }

  /// <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 = reader.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");
```

```
                    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 an branching instruction
  /// </summary>
  /// <param name="instruction">associated branching instruction</param>
  /// <returns>control dependency region for a branching instruction</returns>
  public ControlDependencyRegion getControlDependencyRegion(Instruction 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="junctionPoint">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="regionString">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 variablename</param>
  /// <param name="level">the security level to associate</param>
  private void addVariableMapping(string key, string level)
  {
    securityMappings[key] = level;
  }

  /// <summary>
  /// sets the secuirty 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
      instructionposition</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);
```

```
      key.Append("/");
      key.Append(instruction.ContainingMethod.ContainingType.Name);
      key.Append("/");
      key.Append(instruction.ContainingMethod.Name);
      key.Append(instruction.Position);

      string environment = (string)securityMappings[key.ToString()];
      if ( environment != null )
        return environment;
      return "";
    }

    /// <summary>
    /// retrieves the security level of a variable
    /// </summary>
    /// <param name="variable">the variable of interest</param>
    /// <returns>the security level assoicated with the variable</returns>
    public string getSecurityLevel(Variable variable)
    {
      StringBuilder key = new StringBuilder();
      key.Append(variable.ContainingMethod.ContainingType.ContainingAssembly.Name);
      key.Append("/");
      key.Append(variable.ContainingMethod.ContainingType.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 "";
    }

    /// <summary>
    /// initializes a new analysis settings file
    /// </summary>
    /// <param name="filename">the name of the new file</param>
    /// <param name="assemblies">list of assemblies to include</param>
    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.Types)
```

```
            {
              XmlElement xmlclass = xmlConfigFile.CreateElement("type");
              xmlclass.SetAttribute("id", type.Name);
              xmlasm.AppendChild(xmlclass);
              foreach (Method method in type.MemberMethods)
              {
                if (method.Readable.EndsWith(")"))
                {
                  XmlElement xmlmtd = xmlConfigFile.CreateElement("method");
                  xmlmtd.SetAttribute("id", method.Name);
                  xmlclass.AppendChild(xmlmtd);
                  if (method.LocalVariables != null)
                  {
                    foreach (Variable parameter in method.Parameters)
                    {
                      XmlElement xmlarg = xmlConfigFile.CreateElement("argument");
                      xmlarg.SetAttribute("id", parameter.Name);
                      xmlarg.SetAttribute("level", "");
                      xmlmtd.AppendChild(xmlarg);
                    }
                    foreach (Variable variable in method.LocalVariables)
                    {
                      XmlElement xmlvar = xmlConfigFile.CreateElement("variable");
                      xmlvar.SetAttribute("id", variable.Name);
                      xmlvar.SetAttribute("level", "");
                      xmlmtd.AppendChild(xmlvar);
                    }
                    foreach (Instruction instruction in method.Instructions)
                    {
                      if (instruction is ConditionalJump)
                      {
                        XmlElement xmlregion = xmlConfigFile.CreateElement("region");
                        xmlregion.SetAttribute("index", instruction.Position.ToString());
                        XmlElement xmljunction = xmlConfigFile.CreateElement("junction");
                        xmljunction.SetAttribute("index", instruction.Position.ToString()
                            );
                        xmljunction.SetAttribute("instruction", "");
                        xmlmtd.AppendChild(xmlregion);
                        xmlmtd.AppendChild(xmljunction);
                      }
                    }
                  }
                }
              }
              catch (AssemblyLoadingException )
              {
                System.Console.Out.WriteLine("Configuration error: " + asm + " is not a valid
                    assembly");
              }

            }
            xmlConfigFile.Save(filename);
          }
        }
      }
    }
```

D.1.3 File Analysis/ControlDependencyRegion.cs

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

namespace PNIC.Model.Analysis
{
```

```
  /// <summary>
  /// class resembling the control dependency region structure
  /// </summary>
  class ControlDependencyRegion
  {
    public ControlDependencyRegion()
    {
      region = new List<int>();
      junction = -1;
    }

    /// <summary>
    /// program points that are in the region
    /// </summary>
    private List<int> region;
    public List<int> Region
    {
      get { return region; }
    }

    /// <summary>
    /// the junction point of the region
    /// </summary>
    private int junction;
    public int Junction
    {
      get { return junction; }
      set { this.junction = value; }
    }

    /// <summary>
    /// adds an programpoint to the region
    /// </summary>
    /// <param name="instructionPosition">the programpoint to add</param>
    public void addToRegion(int instructionPosition)
    {
      if (region.Contains(instructionPosition))
      return;
      region.Add(instructionPosition);
    }
  }
}
```

D.1.4 File Analysis/Flowpolicy.cs

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Collections;
using System.Xml;

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

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

    public string Readable
```

```
{
  get {
    StringBuilder readable = new StringBuilder();
    readable.AppendLine("Flowpolicy: " + Name);
    readable.AppendLine();
    readable.AppendLine("Securitylevels: ");
    foreach(Object securityLevel in securityLevels.Values)
    {
      readable.Append(((SecurityLevel)securityLevel).Name + " allows flow to ");
      foreach(SecurityLevel legalFlow in this.getLegalFlow((SecurityLevel)
          securityLevel))
      {
        readable.Append(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 securitylevel 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>securitylevel 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 securitye 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;
  }

  /// <summary>
  /// retrieves the least upper bound of two security levels
  /// </summary>
  /// <param name="securityLevel1">first securitylevel</param>
  /// <param name="securityLevel2">second secuirtylevel</param>
  /// <returns>least upper bound security level, null if lub does not exist</returns>
  public 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;
  }

  /// <summary>
  /// loads a flow policy from a file
  /// </summary>
  /// <param name="policyFilename">the name of the policy file</param>
  /// <returns>true, if loading was successful</returns>
  public bool loadFlowPolicyFromFile(string policyFilename)
  {
    if (policyFilename != null && System.IO.File.Exists(policyFilename))
    {
      try
      {
        XmlDocument flowPolicyFile = new XmlDocument();
        flowPolicyFile.Load(policyFilename);
```

```
        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 sourcelevel 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 destlevel != sourcelvl 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);
              }
            }
          }
        }
        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()
  {
```

```
      foreach (Object obj in securityLevels.Values)
      {
        if ((obj is SecurityLevel) && (((SecurityLevel)obj).LowerSecurityLevels.Count ==
            0))
          lowestSecurityLevel = (SecurityLevel)obj;
      }
    }
  }

  class SecurityLevel
  {
    public SecurityLevel(string name)
    {
      this.name = name;
      this.LowerSecurityLevels = new List<SecurityLevel>();
      this.HigherSecurityLevels = new List<SecurityLevel>();
    }

    /// <summary>
    /// identifier fo the secuirty level
    /// </summary>
    private string name;
    public string Name
    {
      get { return name; }
    }

    /// <summary>
    /// direct connected lower secuirty levels
    /// </summary>
    private List<SecurityLevel> lowerSecurityLevels;
    public List<SecurityLevel> LowerSecurityLevels
    {
      set { lowerSecurityLevels = value; }
      get { return lowerSecurityLevels; }
    }

    /// <summary>
    /// direct connected higher security levels
    /// </summary>
    private List<SecurityLevel> higherSecurityLevels;
    public List<SecurityLevel> HigherSecurityLevels
    {
      set { higherSecurityLevels = value; }
      get { return higherSecurityLevels; }
    }
  }
}
```

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
/// </summary>
/// <param name="filename">relative or absolut filename of the assembly</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
```

```
    /// </summary>
    /// <param name="typename">the name of the type</param>
    /// <returns>type object representing type of typename</returns>
    public Type getType(string typename)
    {
      foreach (Type type in types)
      {
        //System.Console.WriteLine(type.Name);
        if (type.Name.Equals(typename))
          return type;
      }
      return null;
    }

    /// <summary>
    /// checks if a valid assembly is loaded
    /// </summary>
    /// <returns>boolean representing validity of the assembly</returns>
    public bool isValid()
    {
      if (name == null || this.Name == "")
        return false;
      return true;
    }
    #endregion
  }

  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>();
```

```
    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 instructions 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;
```

```
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 DirectSuccesor
{
  set
  {
    directSuccessor = value;
    nextInstructions.Add(value);
  }
  get
  {
    return directSuccessor;
  }
}

/// <summary>
/// The instructions the controlflow 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 predecessing 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>
/// Selfparsing function for this instruction with help of the parser
/// </summary>
/// <param name="parser">The parser that preparsed this instruction</param>
```

```
    /// <param name="position">The position of this instruction in the bytecodearray</
        param>
    /// <returns>The amount of bytes this instruction is in the bytecodearray</returns>
    abstract public int parse(InstructionParser parser, int position);

    /// <summary>
    /// accept method for visitor pattern
    /// </summary>
    /// <param name="visitor">The visitor to use</param>
    abstract public void accept(CILintInstructionVisitor visitor);

    /// <summary>
    /// links the instruction with its successors
    /// </summary>
    /// <param name="parser">The parser that preparsed this instruction</param>
    virtual public void link(InstructionParser parser)
    {
      Instruction nextInstruction = null;
      int positionOfNextInstruction = this.position+1;
      while (!(nextInstruction is Instruction))
      {
        nextInstruction = parser.ParsedInstructions[positionOfNextInstruction];
        positionOfNextInstruction++;
      }
      this.DirectSuccesor = nextInstruction;
      nextInstruction.DirectPredecessor = this;
    }

    #endregion
  }
}
```

D.1.7 File AST/CIL/CILintInstructions.cs

```
using System;
using System.Collections;
using System.Text;
using System.Reflection;

namespace PNIC.Model.AST.CIL
{
  class CILintInstructions
  {
    /// <summary>
    /// Registers all instructions of the CILint sublanguage at a given parser
    /// </summary>
    /// <param name="parser">the parser to register the instructions at</param>
    public static void registerInstructions(InstructionParser parser)
    {
      System.Type binaryType = (new BinaryInstruction()).GetType();
      System.Type cond1Type = (new ConditionalJump1Parameter()).GetType();
      System.Type cond2Type = (new ConditionalJump2Parameters()).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);
```

```
parser.InstructionTable.Add(0x07, loadType);
parser.InstructionTable.Add(0x08, loadType);
parser.InstructionTable.Add(0x09, loadType);

parser.InstructionTable.Add(0x0A, storeType);
parser.InstructionTable.Add(0x0B, storeType);
parser.InstructionTable.Add(0x0C, storeType);
parser.InstructionTable.Add(0x0D, storeType);
parser.InstructionTable.Add(0x10, storeType);

parser.InstructionTable.Add(0x0E, loadType);
parser.InstructionTable.Add(0x11, loadType);

parser.InstructionTable.Add(0x13, storeType);

parser.InstructionTable.Add(0x15, pushType);
parser.InstructionTable.Add(0x16, pushType);
parser.InstructionTable.Add(0x17, pushType);
parser.InstructionTable.Add(0x18, pushType);
parser.InstructionTable.Add(0x19, pushType);
parser.InstructionTable.Add(0x1A, pushType);
parser.InstructionTable.Add(0x1B, pushType);
parser.InstructionTable.Add(0x1C, pushType);
parser.InstructionTable.Add(0x1D, pushType);
parser.InstructionTable.Add(0x1E, pushType);
parser.InstructionTable.Add(0x1F, pushType);
parser.InstructionTable.Add(0x20, pushType);
parser.InstructionTable.Add(0x21, pushType);
parser.InstructionTable.Add(0x22, pushType);
parser.InstructionTable.Add(0x23, pushType);

parser.InstructionTable.Add(0x26, popType);
parser.InstructionTable.Add(0x2A, returnType);

parser.InstructionTable.Add(0x2B, uncondType);
parser.InstructionTable.Add(0x2C, cond1Type);
parser.InstructionTable.Add(0x2D, cond1Type);
parser.InstructionTable.Add(0x2E, cond2Type);
parser.InstructionTable.Add(0x2F, cond2Type);
parser.InstructionTable.Add(0x30, cond2Type);
parser.InstructionTable.Add(0x31, cond2Type);
parser.InstructionTable.Add(0x32, cond2Type);
parser.InstructionTable.Add(0x33, cond2Type);
parser.InstructionTable.Add(0x34, cond2Type);
parser.InstructionTable.Add(0x35, cond2Type);
parser.InstructionTable.Add(0x36, cond2Type);
parser.InstructionTable.Add(0x37, cond2Type);

parser.InstructionTable.Add(0x38, uncondType);
parser.InstructionTable.Add(0x39, cond1Type);
parser.InstructionTable.Add(0x3A, cond1Type);
parser.InstructionTable.Add(0x3B, cond2Type);
parser.InstructionTable.Add(0x3C, cond2Type);
parser.InstructionTable.Add(0x3D, cond2Type);
parser.InstructionTable.Add(0x3E, cond2Type);
parser.InstructionTable.Add(0x3F, cond2Type);
parser.InstructionTable.Add(0x40, cond2Type);
parser.InstructionTable.Add(0x41, cond2Type);
parser.InstructionTable.Add(0x42, cond2Type);
parser.InstructionTable.Add(0x43, cond2Type);
parser.InstructionTable.Add(0x44, cond2Type);

parser.InstructionTable.Add(0x58, binaryType);
parser.InstructionTable.Add(0x59, binaryType);
parser.InstructionTable.Add(0x5A, binaryType);

parser.InstructionTable.Add(0x5F, binaryType);
parser.InstructionTable.Add(0x60, binaryType);
```

```
      parser.InstructionTable.Add(0x61, binaryType);

      parser.InstructionTable.Add(0x62, binaryType);
      parser.InstructionTable.Add(0x63, binaryType);
      parser.InstructionTable.Add(0x64, binaryType);

      parser.InstructionTable.Add(0x65, unaryType);
      parser.InstructionTable.Add(0x66, unaryType);

      parser.ExtendedInstructionTable.Add(0x01, binaryType);
      parser.ExtendedInstructionTable.Add(0x02, binaryType);
      parser.ExtendedInstructionTable.Add(0x03, binaryType);
      parser.ExtendedInstructionTable.Add(0x04, binaryType);
      parser.ExtendedInstructionTable.Add(0x05, binaryType);

      parser.ExtendedInstructionTable.Add(0x09, loadType);
      parser.ExtendedInstructionTable.Add(0x0B, storeType);
      parser.ExtendedInstructionTable.Add(0x0C, loadType);
      parser.ExtendedInstructionTable.Add(0x0E, storeType);
    }
  }
}
```

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
{
  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.IlByteArray[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.IlByteArray[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.IlByteArray[position])
      {
        case 0x15:
          this.Name ="ldc.i4.m1";
          return 1;
```

```
        case 0x16:
          this.Name ="ldc.i4.0";
          return 1;
        case 0x17:
          this.Name ="ldc.i4.1";
          return 1;
        case 0x18:
          this.Name ="ldc.i4.2";
          return 1;
        case 0x19:
          this.Name ="ldc.i4.3";
          return 1;
        case 0x1A:
          this.Name ="ldc.i4.4";
          return 1;
        case 0x1B:
          this.Name ="ldc.i4.5";
          return 1;
        case 0x1C:
          this.Name ="ldc.i4.6";
          return 1;
        case 0x1D:
          this.Name ="ldc.i4.7";
          return 1;
        case 0x1E:
          this.Name ="ldc.i4.8";
          return 1;
        case 0x1F:
          this.Name ="ldc.i4.s";
          return 2;

        case 0x20:
          this.Name ="ldc.i4";
          return 5;
        case 0x21:
          this.Name ="ldc.i8";
          return 9;
        case 0x22:
          this.Name ="ldc.r4";
          return 5;
        case 0x23:
          this.Name ="ldc.r8";
          return 9;
        default:
          throw new InstructionParserException("Parsing error at " + parser.Method.Name +
               " position " + position + "; Not a valid ldc instruction");
      }
    }

    public override void accept(CILintInstructionVisitor visitor)
    {
      visitor.visit(this);
    }
  }
}
```

D.1.12 File AST/CIL/Load.cs

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

namespace PNIC.Model.AST.CIL
{
  class Load:Instruction
  {
```

```
#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 = "loadlocal";
  switch (parser.IlByteArray[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.IlByteArray[position + 1]];
      this.Name = "ldarg.s";
      return 2;
    #endregion

    #region ldloc
    case 0x06:
      this.Name = "ldloc.0";
      this.target = parser.Method.LocalVariables[0];
```

```
            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(CILintInstructionVisitor 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(CILintInstructionVisitor 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 (DirectSuccesor != null)
          readable.Append(" " + DirectSuccesor.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 (DirectSuccesor != null)
          readable.Append(" " + DirectSuccesor.Position);
        return readable.ToString();
      }
    }

    private int positionOfNextInstruction;

    public override int parse(InstructionParser parser, int position)
    {
      this.position = position;
      this.equivalenceClass = "unconditionalJump";
      if (parser.IlByteArray[position] == 0x2B)
      {
        positionOfNextInstruction = position + 2 + (sbyte)parser.IlByteArray[position +
            1];
        this.Name = "br.int8";
```

```
        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(CILintInstructionVisitor visitor)
    {
      visitor.visit(this);
    }

    public override void link(InstructionParser parser)
    {
      Instruction nextInstruction = parser.ParsedInstructions[positionOfNextInstruction];
      NextInstructions.Add(nextInstruction);
      DirectSuccesor = nextInstruction;
      DirectSuccesor.PreviousInstructions.Add(this);
    }
  }
}
```

D.1.15 File AST/CIL/ConditionalJump.cs

```
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
```

```
    {
      get { return this.nextInstruction; }
    }

    protected int branchingTarget;
    protected Instruction jumpInstruction;
    public Instruction JumpInstruction
    {
      set
      {
        jumpInstruction = value;
        NextInstructions.Add(value);
      }
      get { return this.jumpInstruction; }
    }

    public override void link(InstructionParser parser)
    {
      int positionOfNextInstruction = this.position + 1;
      while (!(this.nextInstruction is Instruction))
      {
        this.nextInstruction = parser.ParsedInstructions[positionOfNextInstruction];
        positionOfNextInstruction++;
      }
      this.DirectSuccesor = 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 0x2F:
          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 0x34:
          branchingTarget = position + 2 + (sbyte)parser.IlByteArray[position + 1];
          this.Name = "bge.un.s";
```

```
            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 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 0x43:
            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 0x3F:
            branchingTarget = position + 5 + (int)parser.IlByteArray[position + 1];
            this.Name = "blt";
            return 5;
          case 0x32:
            branchingTarget = position + 2 + (sbyte)parser.IlByteArray[position + 1];
            this.Name = "blt.s";
            return 2;
          case 0x44:
            branchingTarget = position + 5 + (int)parser.IlByteArray[position + 1];
            this.Name = "blt.un";
            return 5;
          case 0x37:
            branchingTarget = position + 2 + (sbyte)parser.IlByteArray[position + 1];
            this.Name = "blt.un.s";
            return 2;
          case 0x40:
            branchingTarget = position + 5 + (int)parser.IlByteArray[position + 1];
            this.Name = "bne.un";
            return 5;
          case 0x33:
            branchingTarget = position + 2 +(sbyte)parser.IlByteArray[position+1];
            this.Name = "bne.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.IlByteArray[position])
      {
        case 0x39:
          branchingTarget = position + 5 + (int)parser.IlByteArray[position+1];
          this.Name = "brfalse";
          return 5;
        case 0x2C:
          branchingTarget = position + 2 +(sbyte)parser.IlByteArray[position+1];
          this.Name = "brfalse.s";
          return 2;
        case 0x3A:
          branchingTarget = position + 5 +(int)parser.IlByteArray[position+1];
          this.Name = "brtrue";
          return 5;
        case 0x2D:
          branchingTarget = position + 2 + (sbyte)parser.IlByteArray[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(CILintInstructionVisitor 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);
    }

    #endregion
  }
}
```

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
{
  /// <summary>
  /// a simple bytecodearray preparser;
  /// 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; }
```

```
    }

    /// <summary>
    /// the instructions already parsed
    /// </summary>
    private Instruction[] parsedInstructions;
    public Instruction[] ParsedInstructions
    {
      get { return this.parsedInstructions; }
    }

    /// <summary>
    /// the resulting method object
    /// </summary>
    private Method method;
    public Method Method
    {
      get { return method; }
    }

    /// <summary>
    /// current position of the parser
    /// </summary>
    private int position;
    public int Position
    {
      get { return this.position; }
    }

    #endregion

    #region methods

    /// <summary>
    /// Parses a given method an returns the corresponding Method object
    /// </summary>
    /// <param name="methodInfo">MethodInfo of the method to parse</param>
    /// <param name="ilByteArray">bytecodearray containing the ilinstructions</param>
    /// <returns>Method object representing the parsed method</returns>
    public Method parseInstructions(Method method, byte[] ilByteArray)
    {
      //set the method in progress, the corresponding bytearray 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 bytearray 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 controlflow graph
      //and relabel positions with programpoints
      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 arrayrange");

      //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 instructiontable
        System.Type instructionType;
        //if the instruction is a 2byte instruciton use extendedinstructiontable
        if (ilByteArray[position] == 0xFE)
        {
          instructionType = (System.Type)this.extendedInstructionTable[(int)((byte)
              ilByteArray[position+1])];
          //Console.Out.WriteLine("2-Byte-Bytecode parsed: " + (int)(((byte)ilByteArray[
              position]) * 256 + (byte)ilByteArray[position+1] ));
        }
        else
        {
          instructionType = (System.Type)this.instructionTable[(int)((byte)ilByteArray[
              position])];
        }
        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 preparsed instruction
        Instruction instruction = (Instruction)Activator.CreateInstance(instructionType);

        //if instruction got preparsed succesfully
        if (instruction != null)
        {
          //store instruction as parsed
          parsedInstructions[position] = instruction;

          //finally let the instruciton 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
```

```
  {
    private string reason;
    public string Reason
    {
      get { return this.reason; }
    }

    public InstructionParserException(string reason)
    {
      this.reason = reason;
    }
  }
}
```

D.1.18 File AST/Method.cs

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Reflection;
using PNIC.Model.AST.CIL;

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);
      nameBuilder.Append(")");
      this.name = nameBuilder.ToString();
      loadMethod(methodInfo);
      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
      {
        if (instructions == null || instructions.Count == 0)
          return name + ": could not parse instructions";
        return name;
      }
    }
```

```
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 methodbody
/// Can be used to traverse the whole methodbody via controlflowgraph
/// </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
```

```
  #region methods

  /// <summary>
  /// loads the method described by methodInfo
  /// </summary>
  /// <param name="methodInfo">MethodInfo of the method to load</param>
  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();
    }
  }

  /// <summary>
  /// retrieves the instruction at given position
  /// </summary>
  /// <param name="position">position of the instruction in the instruction list</param
    >
  /// <returns></returns>
  public Instruction getInstructionAtPosition(int position)
  {
    if (position < instructions.Count)
      return instructions[position];
    else
      return null;
  }


  #endregion
}

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>
    BindingFlags customFlags = BindingFlags.Instance | BindingFlags.Public | BindingFlags
        .NonPublic | BindingFlags.Static | BindingFlags.DeclaredOnly;

    /// <summary>
    /// a list of all membermethods in this type
    /// </summary>
    private List<Method> memberMethods;
```

```
    public List<Method> MemberMethods
    {
      get { return memberMethods; }
    }

    #endregion

    #region methods

    /// <summary>
    /// loads the type represented by the Syste.Type
    /// </summary>
    /// <param name="type">the System.Type decribed by this type</param>
    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
    /// </summary>
    /// <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;

using System.Text;

namespace PNIC.Model.AST
{
  /// <summary>
  /// represents a variable or parameter in an il method
  /// </summary>
  class Variable
  {
    #region fields and getter/setter
```

```
    /// <summary>
    /// a humanreadable 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 CILint instructionset
  /// </summary>
  abstract class CILintInstructionVisitor
  {
    abstract public void visit(BinaryInstruction instruction);
    abstract public void visit(ConditionalJump1Parameter instruction);
    abstract public void visit(ConditionalJump2Parameters instruction);
    abstract public void visit(Load 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>
  /// abstreact 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;
```

```
using PNIC.Model.AST;
using PNIC.Model.Analysis;
using PNIC.Controller.Print;

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 user in this analysis
    /// </summary>
    private FlowPolicy flowPolicy;
    public FlowPolicy FlowPolicy
    {
      get { return flowPolicy; }
      set { this.flowPolicy = value; }
    }

    /// <summary>
    /// run an analysis
    /// </summary>
    /// <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))
              {
                summary.AppendLine("  Analysing method: " + methodname);
                System.Console.WriteLine("  Analysing method: " + methodname);
                string result;
                Method method = type.getMemberMethod(methodname);

                if (method != null && checkFlowPolicyCompatibility(method))
                {
                  if (method.LoadingException == null)
                  {
                    MethodTransformator transform = new MethodTransformator();
                    transform.Configruation = configuration;
                    transform.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!";
}

/// <summary>
/// checks if the flow policy is comptaible with the analysis settings
/// </summary>
/// <param name="method">the method the settings are for</param>
/// <returns>true, if policy and settings are compatible</returns>
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; }
    }
  }
}
```

D.2.3 File Analysis/CILintTransformator.cs

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

using System.Text;
using PNIC.Model.Analysis;

namespace PNIC.Controller.Analysis
{
  /// <summary>
  /// class representing abstract transformations for CILint
  /// </summary>
  class CILintTransformator:CILintInstructionVisitor
  {
    #region ctor
    public CILintTransformator()
    {
      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 { return flowPolicy; }
      set { this.flowPolicy = value; }
    }

    private AbstractState state;
    public AbstractState State
    {
      get { return state; }
      set { this.state = value; }
    }

    #endregion

    #region visit methods for instructions

    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;
```

```
    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);
  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)
{
```

```
    checkSecurityLevelStackUniformity(instruction);

    SecurityLevel securityEnvironment = flowPolicy.getSecurityLevel(configuration.
        getSecurityEnvironment(instruction));

    SecurityLevel topOfStackLevel = state.RecentSecurityLevelStack.Pop();

    SecurityLevel leastUpperBound = flowPolicy.getLeastUpperBound(securityEnvironment,
        topOfStackLevel);

    if (leastUpperBound == null)
      throw new AnalysisException(instruction.Position + " no least upper bound found
          for " + securityEnvironment.Name + " and " + topOfStackLevel.Name);

    SecurityLevel securityLevelOfVariable = flowPolicy.getSecurityLevel(configuration.
        getSecurityLevel(instruction.Target));

    if (!flowPolicy.getLegalFlow(leastUpperBound).Contains(securityLevelOfVariable))
      throw new AnalysisException(instruction.Position + " illegal flow from " +
          leastUpperBound.Name + " to " + securityLevelOfVariable.Name);

    state.setSecurityLevelStack(instruction, state.RecentSecurityLevelStack);
  }

  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 + " securitylevel 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 + " securitylevel is not
                    constant at this programpoint");
            }
          }
        }
      }
    }

    #endregion
  }
}
```

D.2.4 File Analysis/MethodTransformator.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 MethodTransformator:InstructionContainerVisitor
  {
    private AnalysisSettings configuration;
    public AnalysisSettings Configruation
    {
      get { return configuration; }
      set { this.configuration = value; }
    }
    private FlowPolicy flowPolicy;
    public FlowPolicy FlowPolicy
    {
      get { return flowPolicy; }
      set { this.flowPolicy = value; }
    }

    private CILintTransformator instructionTransformator;
    private List<Instruction> workList;
    private List<ConditionalJump> checkedBranchingInstructions;
    private Method recentMethod;

    /// <summary>
    /// method visitor for cil int analysis
    /// </summary>
    /// <param name="method">the method to analyse</param>
    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 instructionTransformator for this method
      instructionTransformator = new CILintTransformator();
      instructionTransformator.Configuration = configuration;
      instructionTransformator.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");
      instructionTransformator.State.restoreState(recentInstruction.DirectPredecessor);

      //for all instructions that are ransitivly 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.DirectSuccesor;
      }
    }
  }

  /// <summary>
  /// checks if the regions confirm to the safe over approximation definition
  /// if not soap: 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 depency 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.DirectSuccesor != conditionalJump.JumpInstruction)
      {
        //check all successors
        foreach(Instruction successor in conditionalJump.NextInstructions)
        {
          if ( !controlDependencyRegion.Region.Contains(successor.Position)
            && controlDependencyRegion.Junction != successor.Position )
            throw new AnalysisException("Succesor 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.getInstructionAtPosition(
            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.DirectSuccesor.
              Position
            && !controlDependencyRegion.Region.Contains(instructionInRegion.
                DirectSuccesor.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(instructionTransformator);
    workList.Remove(instruction);
  }

  private void visit(ConditionalJump conditionalJump)
  {
    //analyse this instruction
    conditionalJump.accept(instructionTransformator);
    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.getInstructionAtPosition(
            position);
        //if instruction is not in worklist
        if (!workList.Contains(instructionAtPosition))
          workList.Add(instructionAtPosition);
      }

      //if jumtarget not in worklist, add jumptarget to worklist
      if (!workList.Contains(conditionalJump.JumpInstruction))
        workList.Add(conditionalJump.JumpInstruction);
```

```
      }
      //mark this conditional jump as checked
      checkedBranchingInstructions.Add(conditionalJump);
    }

    /// <summary>
    /// get the first instruction in execution order from worklist
    /// </summary>
    /// <returns>Instruction object representing the first instruction</returns>
    private Instruction getFirstInstructionFromWorkList()
    {
      Instruction firstInstruction = null;
      foreach (Instruction instruction in workList)
      {
        //if firstinstruction not defined or instruction is in front of recent
            firstinstruction, set firstinstruciton
        if ((firstInstruction == null) ||
          (firstInstruction.DirectPredecessor == instruction) ||
          (instruction is UnconditionalJump && firstInstruction.PreviousInstructions.
              Contains(instruction))
          )
          firstInstruction = instruction;
      }
      return firstInstruction;
    }


    /// <summary>
    /// not needed int cilint analysis
    /// </summary>
    public override void visit(PNIC.Model.AST.Type type)
    {
      throw new NotImplementedException();
    }

    /// <summary>
    /// not needed in cilint analysis
    /// </summary>
    public override void visit(PNIC.Model.AST.Assembly assembly)
    {
      throw new NotImplementedException();
    }
  }
}
```

D.2.5 File Print/CompletePrint.cs

```
using PNIC.Model.AST;
using PNIC.Model.Analysis;

namespace PNIC.Controller.Print
{
  class CompletePrint:InstructionContainerVisitor
  {
    /// <summary>
    /// if set additional information, e.g.  secuirty environments of instructions
    /// are printed
    /// </summary>
    private AnalysisSettings configuration;
    public AnalysisSettings Configuration
    {
      get { return this.configuration; }
      set { this.configuration = value; }
    }

    /// <summary>
    /// if true, the output contains only the equivalence classes of instructions
    /// else the concrete instruction is printed
```

```
    /// </summary>
    private bool equivalenceClass;
    public bool EquivalenceClass
    {
      get { return this.equivalenceClass; }
      set { this.equivalenceClass = value; }
    }


    public override void visit(PNIC.Model.AST.Method method)
    {
      System.Console.WriteLine("Method: " + method.Name);
      PrintInstruction printer = new PrintInstruction();
      printer.EquivalenceClass = this.equivalenceClass;
      if (configuration is AnalysisSettings)
        printer.Configuration = configuration;
      if (method.LoadingException is InstructionParserException)
        System.Console.WriteLine(method.LoadingException.Message);
      else
      {
        foreach (Instruction instruction in method.Instructions)
        {
          instruction.accept(printer);
        }
      }
    }

    public override void visit(PNIC.Model.AST.Type type)
    {
      System.Console.WriteLine("Type: " + type.Name);
      foreach (Method method in type.MemberMethods)
      {
        visit(method);
      }
    }

    public override void visit(PNIC.Model.AST.Assembly assembly)
    {
      System.Console.Out.WriteLine("Assembly: " + assembly.Name);
      foreach (Type type in assembly.Types)
      {
        visit(type);
      }
    }
  }
}
```

D.2.6 File Print/PrintInstruction.cs

```
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;
```

```
public bool EquivalenceClass
{
  get { return this.equivalenceClass; }
  set { this.equivalenceClass = value; }
}

/// <summary>
/// if set additional information, e.g. secuirty 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.Write(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.ReadableEquivalenceClass);
      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.ReadableEquivalenceClass);
      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.ReadableEquivalenceClass);
      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.ReadableEquivalenceClass);
      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.ReadableEquivalenceClass);
      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.ReadableEquivalenceClass);
      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.ReadableEquivalenceClass);
      else
        Console.Out.WriteLine(instruction.ReadableInstruction);
    }
  }
}
```

D.3 Package View

D.3.1 File ConsoleUI/ConsoleUI.cs

```
using System;
```

```
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>
  /// consolen 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.Out.WriteLine("Prototypical Non-Interference Checker");
      Console.Out.WriteLine("-+=============================================-");
      Console.Out.WriteLine();
      if (flowpolicy is FlowPolicy)
      {
        Console.Out.WriteLine("Flow policy loaded: " + flowpolicy.Name);
      }
      if (configuration is AnalysisSettings)
      {
        Console.Out.WriteLine("Analysis settings loaded: " + configuration.Name);
      }
      if (assemblies.Count != 0)
      {
        Console.Out.Write("Assemblies loaded: " + assemblies[0].Name);
        for (int i = 1; i < assemblies.Count; i++)
        {
          Console.Out.Write("            " + assemblies[i].Name);
        }
        Console.Out.WriteLine();
      }
    }

    private void invokeMainMenu()
    {
      while (true)
      {
        printHeader();
        Console.Out.WriteLine("F - Flow Policy Menu");
        Console.Out.WriteLine("S - Analysis Settings Menu");
        Console.Out.WriteLine("A - Assembly Menu");
        Console.Out.WriteLine("R - Run Analysis");
        Console.Out.WriteLine("X - Exit");
        Console.Out.WriteLine();
        Console.Out.Write("Action: ");
        char action = Console.ReadKey().KeyChar;
        Console.Out.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;
      Console.Out.WriteLine();
      switch (action)
      {
        case 'l':
        case 'L':
          LoadFlowPolicy();
          break;
        case 'd':
        case 'D':
          DumpFlowPolicy();
          break;
        case 'm':
        case 'M':
          return;
      }
    }
  }

  private void LoadFlowPolicy()
  {
    Console.Out.Write("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(""))
      {
```

```
        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.Write("Action: ");
      char action = Console.ReadKey().KeyChar;
      Console.Out.WriteLine();
      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)
```

```
    {
      if (existingAssembly.Name == filename)
        return;
    }
    Assembly assembly = new Assembly(filename);
    if (assembly is Assembly && assembly.isValid())
      assemblies.Add(assembly);
    else
    {
      Console.Out.WriteLine(filename + " is not a valid assembly");
      Wait();
    }
  }

  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();
  }

  #endregion

  #region configuration menu

  private void invokeConfigurationMenu()
  {
    while (true)
    {
      printHeader();
      Console.Out.WriteLine("L - Load Analysis Settings");
      Console.Out.WriteLine("D - Dump loaded Analysis Settings");
      Console.Out.WriteLine("I - Initialize Analysis Settings");
      Console.Out.WriteLine("M - Back to Main Menu");
      Console.Out.WriteLine();
      Console.Out.Write("Action: ");
      char action = Console.ReadKey().KeyChar;
      Console.Out.WriteLine();
      switch (action)
      {
        case 'l':
        case 'L':
          LoadConfiguration();
          break;
        case 'd':
        case 'D':
          DumpConfiguration();
          break;
```

```
        case 'i':
        case 'I':
          InitializeNewConfiguration();
          break;
        case 'm':
        case 'M':
          return;
      }
    }
  }

  private void InitializeNewConfiguration()
  {
    Console.Out.WriteLine("Please enter the filename of the analysis settings: ");
    string filename = Console.In.ReadLine();

    while (System.IO.File.Exists(filename))
    {
      Console.Out.WriteLine("File already exists, please choose another name: ");
      filename = Console.In.ReadLine();
    }

    List<string> assemblies = new List<string>();
    Console.Out.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.Out.WriteLine("Settings for this analysis: ");
    Console.Out.WriteLine("Settings File: " + configuration.Name);
    Console.Out.WriteLine();
    Console.Out.WriteLine(configuration.ReadableConfiguration);
    Wait();
  }

  private void LoadConfiguration()
  {
    Console.Out.WriteLine("Loading analysis settings");
    Console.Out.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.Out.WriteLine("No flowpolicy loaded");
      LoadFlowPolicy();
    }
    if (!(configuration is AnalysisSettings))
    {
```

```
        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();
    }
  }
}
```

E Source Code of Example Applications

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.

```
{
  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;
```

```
    return 1;
  }
  //data not stored in special location, store it in common location
  int commonstorage = data;
  return 1;
}

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 userinformation to update server
  int unencryptedChannel = userdata;

  //recieve answer from update server
  //if the user has a valid license
  if (unencryptedChannel < 5)
  {
    //recieve 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++;
    while (i * j > number)
    {
      j--;
    }
  }
  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;
    }

  }
}
```