MASTER'S THESIS 2024

SinfoJ: A simplistic Information Flow Analysis with Reference Attribute Grammars

Max Soller

ISSN 1650-2884
LU-CS-EX: 2023-79

DEPARTMENT OF COMPUTER SCIENCE
LTH | LUND UNIVERSITY

EXAMENSARBETE
Datavetenskap

LU-CS-EX: 2023-79

SinfoJ: A simplistic Information Flow Analysis with Reference Attribute Grammars

SinfoJ: En simplistisk informationsflödesanalys med Reference Attribute Grammars

Max Soller

SinfoJ: A simplistic Information Flow Analysis with Reference Attribute Grammars

Max Soller
ma8251so-s@student.lu.se

February 8, 2024

Master's thesis work carried out at
the Department of Computer Science, Lund University.

Supervisor: Görel Hedin, gorel.Hedin@cs.lth.se

Examiner: Niklas Fors, niklas.fors@cs.lth.se

Abstract

Information flow analysis is a concept that aims to analyze how information propagates in a program with the goal of detecting program points where sensitive information might leak. This thesis introduces **SINFOJ**, a simplistic static information flow analysis for Java, inspired by the **JFLOW** language [12]. A key component in JFLOW and information flow analysis, is the fact that variables and objects are labeled with something called *Security Labels*. These labels depict the sensitivity of information within variables and objects and determine the information flows that are allowed in a program. Unlike JFLOW, our approach utilizes Java Annotations instead of extending the Java syntax, aiming for easier usage and implementation. The analysis employs intraprocedural *Control Flow Graphs* and extends to basic interprocedural analysis using a *Call Graph*. With this thesis we aim to answer questions such as: How can *Reference Attribute Grammars* be employed for intraprocedural and basic interprocedural analysis? What subset of JFLOW can be implemented with the use of annotations? How could **SINFOJ** be further improved? The thesis contributes a foundation for future research in information flow analysis as well as showcasing some of the capabilities and limitations of Reference Attribute Grammars.

Keywords: Information Flow Analysis, Static Program Analysis, Control Flow Graph, Decentralized Label Model, Reference Attribute Grammars

Contents

Chapter 1
Introduction

The importance of ensuring the security and integrity of sensitive information in computer programs cannot be overstated. The rise of cyber threats necessitates strong measures to protect highly sensitive data from unintended exposure. *Information flow analysis* has emerged as a discipline in addressing this challenge by providing an approach to analyze how information propagates through a program and how this potentially could lead to security vulnerabilities.

In this thesis, we will explore how a static information flow analysis in Java can be implemented with the help of *Reference Attribute Grammars* (RAGs) and the extensible Java compiler **ExtendJ** [6][14]. With this analysis we aim to examine how sensitive data inadvertently can leak into less restrictive variables or objects, i.e. variables or objects with lower security levels, and with this, potentially compromising the security of a program. To address this issue, we take on a concept of *security-class labels*, which we in this thesis generally and simply refer to as *labels*.

These labels serve the purpose of adding security levels to variables, objects and methods in order to preserve safe information flows within a program. In practical terms, this aims to prevent high sensitive information in e.g. a variable, from flowing into a variable with lower sensitivity. A simple example of this could be that a high sensitive variable cannot be assigned to a low sensitive variable.

We aim to implement a subset of the information flow control language JFlow [12]. JFlow is an extension to Java which enables information flow analysis with the help of the *decentralized label model* [13]. To be able to add labels to variables, methods and classes, JFlow introduce new syntax to the Java language. In this thesis we want to investigate how we can implement a similar analysis as the one in JFlow without the need of introducing new Java syntax, making our analysis both easier to implement and use. Instead of extending the Java language, we will explore the use of *Java Annotations* to label variables, objects and methods with security-class labels.

To implement the mentioned information flow analysis, we need to construct a *Control Flow Graph* (CFG). This CFG serves as the foundation for conducting a *dataflow analysis*. To accomplish this, we will use two frameworks: **IntraJ**, a framework for constructing an *intraprocedural* CFG, i.e. a CFG within a process such as a method or a function [15], and **Cat** (Call Graph Analysis Tool) [1], which facilitates *interprocedural analysis*, i.e. analyzing interactions between processes such as methods and functions. Both frameworks are implemented using Reference Attribute Grammars and ExtendJ.

By working with a subset of JFlow, we aim to introduce an implementation of an information flow analysis that can be used for more in-depth exploration and research. This exploration will hopefully be an initial step toward examining the potential applications of ExtendJ and RAGs in future, more advanced information flow analysis or similar endeavors.

1.1 Scope and Limitations

To ensure that the purpose of this thesis is viable, we need to narrow our focus. We will begin with an investigation of JFlow and its implementation. This will enable us to select a subset of JFlow. The goal of this chosen subset is to produce an analysis that is minimally viable and still capable of detecting fundamental information flow violations. Furthermore, we intentionally restrain ourselves from extending the Java syntax. The reason for this decision is the desire to prevent a more complicated implementation and ensure that the analysis is useful for future research and potential users.

1.2 Aims and goals

Within the context of our defined scope, we have identified certain aims and goals to clarify the objectives of this thesis.

Our initial goal is to identify a subset of JFlow that can utilize Java annotations instead of extending the Java language. This will be explored with the help of the Background Chapter 2. Secondly, we want to investigate how Reference Attribute Grammars can be employed to implement an intraprocedural and basic interprocedural information flow analysis with the chosen subset of JFlow. This will be covered in the Design Chapter 3. We also want to assess how the implemented analysis perform in terms of speed compared to similar analyses, which will be evaluated in the Evaluation Chapter 4. Lastly, we are interested in how the implemented analysis could be further improved, which will be discussed in the Limitations and Further Work Chapter 5.

[1] https://github.com/IdrissRio/cat/

Chapter 2
Background

The upcoming chapter will outline the theory necessary to understand the content of this thesis and to help achieve the goals mentioned in Section 1.2. Firstly, we will provide an introduction to *Information Flow* and it's connection to security vulnerabilities in Section 2.1. This will be followed by theory regarding *Static Program Analysis* and *Dataflow Analysis* in Section 2.2. Lastly we will give a rather basic and general explanation of *Compilers* in Section 2.3, with a more focused discussion on the compiler components relevant to this thesis.

2.1 Information Flow

Information flow generally refers to the movement and propagation of data or information through a program, system, or network. It encompasses how data is generated, modified, transmitted, and consumed by different components or entities within the system. In a standard access control scheme, the primary goal is to ensure that only authorized individuals or entities are granted access to restricted data while also preventing unauthorized access or security breaches [16].

Information flow becomes of utmost importance when looking at how sensitive information, such as confidential data, personal details, passwords, etc., moves and spreads within a computer system or software application. The primary objective of an information flow analysis from a security perspective is therefore to ensure that sensitive information is not inadvertently leaked to unauthorized parties or used inappropriately. This analysis involves tracing the paths that sensitive data takes through a program or system and understanding how it interacts with other data, processes, and components.

Confidentiality and *Integrity* are two terms often used to in the context of data security. In order to preserve confidentiality a system needs to protect sensitive information from leaking to unauthorized destinations or unauthorized disclosure. Integrity on the other hand

refers to the accuracy and trustworthiness of information and in order to preserve integrity the system needs to protect sensitive information from unauthorized modification. A distinct difference between these two concepts is that integrity can be compromised without external interactions, such as by unregulated computations in a program in the form of for example a programming error, i.e. a bug.

Explicit and Implicit Flows

There exist two distinct different information flows, namely *explicit* and *implicit flows*. Any statement or construct that directly lets data flow into another such as in an assignment or a declaration is called an explicit flow, see the left example in figure 2.1. An implicit flow on the other hand is the result of a program structure that implicitly lets data from a variable flow into another, see the right example in figure 2.1. In these examples, the variable l refers to *public* variable, i.e. a variable with low sensitivity, and the variable h refers to a *secret* variable, i.e. a variable with high sensitivity.

```
1: int l;
2: int h;
. . .
3: l = h;
```

```
1: int l;
2: boolean h;
. . .
3: int l = 0;
4: if (h) {
5:    l = 1;
6: }
```

Figure 2.1: Example of explicit and implicit flows.

The left example in figure 2.1, is an example of an explicit flow, it occurs on line 3 where the secret variable h directly flows into the public variable l. The right example in figure 2.1, shows an implicit flow on line 5 where the assignment to the public variable l is conditioned on the secret variable h. This lets an observer deduct the value of h when observing the value of l.

Through regulation of these information flows, we can locate leakage of sensitive information to unauthorized destinations and thereby maintaining confidentiality. Additionally, this helps preserving data integrity by ensuring that unauthorized computations do not occur, which otherwise might result in security risks arising from unregulated security-critical decisions within a program.

2.1.1 Taint Analysis

Taint analysis is a concept used to track the information flow of sensitive or *tainted data* from a source to a *sink* within a program. The term tainted data refers to data that originates from untrusted or external sources, such as user inputs or data obtained from network communication. A sink refers to the final destination in the program, whether that may be a return statement in a method or some sort of an output stream.

Essentially, a taint analysis tracks how and where tainted data is being processed, manipulated, or potentially used to influence the program's behavior. For example, in an assignment, the left-hand expression becomes tainted if the right-hand expression is tainted. If, at any point in this analysis, the tainted data is used inappropriately, the taint analysis flags it as a potential security vulnerability [11] [17].

Taint analysis is closely related to information flow in the sense that both analyses tracks how data flows trough a program, essentially a taint analysis can be said to track information flows.

2.1.2 Decentralized Label Model

The decentralized label model is a security model used to enforce information flow control in computer systems and software applications. It is designed to prevent unauthorized information flow and protect sensitive data from leakage or unauthorized access. Unlike traditional access control models, which rely on a centralized authority to manage permissions, the decentralized label model distributes control over information flow to individual data objects in the system [13].

In the decentralized label model, data objects, e.g. variables, are associated with *Security Labels* that represents their sensitivity or confidentiality level. These security labels are used to enforce access controls and information flow policies throughout a system. The labels are typically represented as tags or markings attached to the data object, and contain information about the sensitivity and the security requirements for its handling.

Essentially, a label consists of *policies* that defines who can read and change the entity associated with the label. The *owners* are allowed to manipulate and read the labeled data object while the *readers* are only allowed to read the data from the entity. Owners and readers, commonly referred to as *principals*, are users or groups for a given security system, much like in a access-control scheme.

To further explain this concept, consider a label that looks like the following:

$$L_1 = \{p_1 \,;\, p_2\} \quad , \textit{where } p_1 = \{o_1 : r_1,\, r_2\},\; p_2 = \{o_2 : r_2,\, r_3\}$$

The label L_1 has two so called policies, p_1 and p_2. The policies in turn have readers and owners that are separated by a colon. p_1 has an owner o_1 and p_2 has an owner o_2. Furthermore, p_1 has the readers $\{r_1, r_2\}$ and p_2 has the readers $\{r_2, r_3\}$. The effective reader set of the label L_1, which is the intersection of the readers in the policies, is $\{r_2\}$. If a variable x is labeled with L_1, only the defined users o_1 and o_2 can manipulate the value of x. The principals that are allowed to read the value of x are the effective reader set, i.e. r_2, along with the owners o_1 and o_2, since owners implicitly are allowed readers. Consider another label:

$$L_2 = \{\,\}$$

This is an empty label and is the least restrictive label, i.e. an entity with this label can be viewed as a public entity.

An important thing to mention about the decentralized label model is that it enables an owner of a policy, to declassify it's own policy. This means that it can downgrade or essentially lower it's security level in order to be able to allow some information flows. Certain systems sometimes needs to leak sensitive information, such as a login procedure. If a user provides a wrong password, the system will tell the user this, which in turn can be used to deduce the actual password. This allows for a relaxation of the strict information flows but since a principal only can act for its own policies this still preserves the overall security. Although this is very useful in practical scenarios, this concept will not be further covered in this thesis since it would add another layer of complexity to the analysis and isn't needed for a basic information flow analysis.

2.1.3 JFlow

JFlow is an information flow analysis that employs the decentralized label model. It is an extension to the Java language and adds the support to label e.g. variables, methods and objects [12]. Look at the following example:

```
int{Alice:Bob} x;
int{Bob} y;
...
x = y;
```

This is a simple example of how JFlow has extended normal Java-syntax to add the support for labels. In this example the variable x has been labeled with the policy *{Alice : Bob}*, i.e the owner is a defined principal called *Alice* and the reader is a defined principal called *Bob*. The boolean b has been labeled with the policy *{Bob}*, i.e. *Bob* is the owner and since no readers are listed, it is also the sole reader of b. In the above example there is an information flow from y to x. Since Bob is allowed to read the variable x but not change it, this program will lead to an information flow violation.

Generally, consider a variable y with label L_1, it can be assigned to another variable x with the label L_2, if the label L_1 can be relabeled to the label L_2. This flow is denoted with $L_1 \sqsubseteq L_2$ and is allowed if and only if, for every policy within the label L_2, the label L_1 contain a policy that is at least as restrictive. Furthermore, the label of a computed value is at least as restrictive as the concerned variables in the computation, i.e. it causes a join of the labels.

An implementation of JFlow is the security-typed language **Jif** [1]. The entire specification of JFlow and Jif would take up to much space and all of it isn't necessary for this thesis, but most of it can be read in the paper *JFlow: Practical Mostly-Static Information Flow Control* by Andrew C. Myers [12].

2.2 Static Program Analysis

Static program analysis is a technique used to analyze source code or compiled code without actually executing the program. It aims to identify potential issues, errors, vulnerabilities,

and other properties of a program by examining its code structure, syntax, and semantics. By examining the static behaviour, these type of analysis can look at all possible execution paths. The alternative is dynamic analysis which only looks at one execution paths when running the program. Static analysis is performed before the program is executed and can therefor help developers catch problems early in the development process, leading to improved code quality and reduced software defects. A few examples of different static program analyses are *type analysis*, *dataflow analysis*, *control flow analysis* and *abstract interpretation* [11].

2.2.1 Lattice Theory

A *lattice* is an abstract structure commonly used in dataflow analysis to provide a formal mathematical framework for modeling and solving dataflow problems. The lattice is constructed to represent possible program states. This lattice is a partially ordered set where each element signifies a state of the program's variables and the partial ordering reflects the information flow between these states. A *partial order*, $(S, \sqsubseteq)$, consist of a set, S, and a partial order relation, $\sqsubseteq$, which signifies the relations between the set elements. Furthermore, the conditions reflexivity, transitivity and anti-symmetry needs to be satisfied for the elements in the set for a partial order [11].

The notion $x \sqsubseteq y$ can be used to signify that x is at least as precise as y, or we can say that y is a *safe approximation* of x. In the context of our analysis and security-classes, a high sensitive security-class is a safe approximation of a lower sensitive security-class.

Additionally we have a *least upper bound*, $\bigsqcup X$, and a *greatest lower bound*, $\bigsqcap X$, for a subset X in a lattice. If X is a subset of the set S, i.e. $X \subseteq S$, then the least upper bound for X is the smallest element in the set S that is greater than or equal to every other element in X. This can also be expressed in the following equation:

$$X \sqsubseteq \bigsqcup X \quad \wedge \quad \forall y \in S : X \sqsubseteq y \quad \Longrightarrow \quad \bigsqcup X \sqsubseteq y \tag{2.1}$$

And the greatest lower bound for the subset X is instead the largest element in S that is less than or equal to every other element in X. The definiton is as following:

$$\bigsqcap X \sqsubseteq X \quad \wedge \quad \forall y \in S : y \sqsubseteq X \quad \Longrightarrow \quad y \sqsubseteq \bigsqcap X \tag{2.2}$$

When these terms are used for pairs of elements in the set we use the notion $x \sqcup y$ called *join* and $x \sqcap y$ called *meet*. Both of these operations are fundamental in program analysis for merging different states in a program, which we will discuss more in Section 2.2.3 and later in our implementation in Chapter 3.

As mentioned, a lattice is a partial order $(S, \sqsubseteq)$ where $\forall x, y \in S$, $x \sqcup y$ and $x \sqcap y$ exists. A *complete lattice* is a lattice where the previous condition also holds for all $X \subseteq S$. Additionally, a complete lattice has two elements that denotes the *smallest* and *largest* elements in the set, called *bottom*, $\bot$, and *top*, $\top$, respectively.

All finite lattices can be represented with a *Hasse diagram* where nodes are the elements in the set and edges show relations. Let's look at an example, assume we have the partial order

$(\mathcal{P}(A), \subseteq)$ where $\mathcal{P}(A)$ is the powerset of the set $A = \{0, 1, 2\}$. Additionally $\top = A$ and $\bot = \emptyset$. The resulting Hasse diagram can be seen in Figure 2.2.

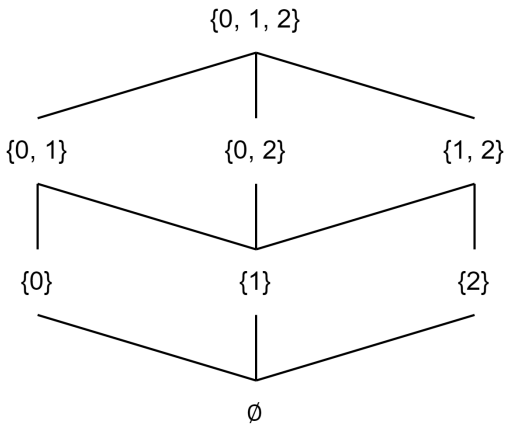


Figure 2.2: A Hasse diagram representation of the complete lattice $(\mathcal{P}(A), \subseteq)$.

A *fixed point* is a solution to an equation system over a lattice where each equation is called a *constraint equation*. Formally we say that x in a lattice L is a fixed point for the function f if $f(x) = x$. The *least* fixed point x is the most precise solution to these equations. Theorem 1 is called *Kleene's Fixed-Point Theorem* [11] and tells us that if we have a complete lattice with finite height and strictly monotone functions, we can ensure convergence and the finding of a unique most precise solution. This is important in dataflow analysis, which we will discuss more in Section 2.2.3, where we need to reach a state when further iterations does not alter the analysis result. One thing to consider is that this equation system is a conservative approximation of the actual program and will only produce the most semantically precise solution, which often is a good enough approximation.

Theorem 1. ***Kleene's Fixed-Point Theorem*** *[11]: In a complete lattice L with finite height, every monotone function $f : L \to L$ has a unique least fixed point denoted lfp(f) defined as:*

$$lfp\,(f) = \bigsqcup_{i \geq 0} f^i(\bot)$$

2.2.2 Control Flow Graph

A *control flow graph* (CFG) is a graph representation used in static program analysis to represent the possible execution paths of a program. It represents how a program's control flow, or the order of execution of instructions, moves from one instruction to another based on various conditions and decisions. Control flow graphs are particularly useful for analyzing the structure of a program where it is more important to analyze the order of execution rather than the *Abstract Syntax Tree* (AST) [11].

A CFG can be seen as a directed graph where nodes symbolizes e.g. expressions or statements and edges depict potential pathways of control. When considering a node within a

CFG, denoted as v, the term $pred(v)$ designates the set of preceding nodes, while $succ(v)$, represents the set of successor nodes. A CFG can also be assumed to posses an entry point, referred to as $entry$ and an exit point, referred to as $exit$.

2.2.3 Dataflow Analysis

Dataflow analysis is a technique in program analysis and compiler optimization that aims to understand how data values propagate through a program's variables and expressions along a CFG. Its primary goal is to gather information about how data is used and modified throughout a program's execution, which helps compilers make informed decisions about optimizing code and performing various tasks, such as for example controlling information flow [11].

In order to perform dataflow analysis, the standard approach is centered around a CFG in combination with a complete lattice of finite height. This lattice contains abstract information for different CFG nodes targeted to deduct dataflow information. Each CFG node is linked to a constraint variable, which has a value of one of the elements in the lattice. Additionally, dataflow constraints are formulated for each CFG node, controlling the interrelationships between the constraint variable of the node and those of other nodes, typically neighbors. These dataflow constraints are based on the programming construct the node embodies. As mentioned earlier, if the lattice has a finite height and consists of monotone constraint functions, there exists a unique least fixed point. A framework with a complete lattice and monotone functions is called a *monotone framework*. It suffices to construct a CFG and specifying the monotone constraint functions for each CFG node to perform a dataflow analysis.

When doing dataflow analysis on a CFG, a normal approach is to define the following sets, where $[\![v]\!]$ refers to a constraint variable which models the abstract state at a program point.

- in_v: Contains knowledge at entrance of the node v, i.e immediately before the program point at v.
- out_v: Contains knowledge at exit of the node v, i.e immediately after the program point at v.
- $trans_v$: The *transfer function* is used to transform the state at node v. In a *forward analysis* it updates out_v from in_v and vice versa for a *backward analysis*. In order to instantiate a dataflow analysis it is necessary to define the transfer function for each CFG node.
- $join_v$: Combines abstract states of the predecessors or successors of the CFG node at v. It is defined by:

$$JOIN(v) = \bigsqcup_{w \in pred(v)} [\![w]\!]$$

.

Another characteristic of a dataflow analysis is if it is a *forward* or *backward analysis*. A forward analysis computes information regarding something in the past, i.e it depends on the $pred$ values of the CFG node. A backward analysis is contrary an analysis that computes

information regarding the future, and depends on ***succ*** of the CFG node. In a forward analysis, $join_v$ is defined by using ***pred***, as seen in definition above, and starts at the CFG ***entry*** point. In a backward analysis, $join_v$ is defined by the use of ***succ*** and instead starts at the CFG ***exit*** point.

Live Variable Analysis

To provide a contextual illustration and enhance the practical understanding of dataflow analysis, the following exemplifies a prevalent analysis. *Live Variable Analysis* (LVA) identifies whether a variable's value might be read later in the program without any intervening writes. While the property is inherently undecidable, a LVA static analysis helps optimize programs by conservatively assuming that "not live" results are trustworthy, with "live" results considered safe but unnecessary for optimization purposes.

The lattice used in a LVA is a so called *powerset* lattice, such as the one in Figure **??**. Each element in the lattice corresponds to a set of variables in the program, which makes lattice is unique for the program being analyzed. The bottom element, $\bot$, is the empty set and the top element, $\top$, is the set of all the variables in the program.

As stated in section 2.2.3, the initiation of this process begins with the formulation of the defined functions. The *join* function, for instance, can be defined in the subsequent manner.

$$JOIN(v) = \bigcup_{w \in succ(v)} [\![w]\!]$$

In this context, v represents a CFG node, and $[\![v]\!]$ signifies the constraint variable, whose value is the set of program variables that may be live immediately before the node v. The most important constraint in LVA is for assignments and that rule along with a few others can be defined in the following manner:

$$\text{Assignment: } X = E: \quad [\![v]\!] = JOIN(v) \setminus X \cup vars(E) \tag{2.3}$$

$$\text{Declaration: } T\ X_1, \ldots, X_n: \quad [\![v]\!] = JOIN(v) \setminus X_1, \ldots, X_n \tag{2.4}$$

$$\text{Branch statements: } \left.\begin{array}{l} \textbf{\textit{if}}(E): \\ \textbf{\textit{while}}(E): \end{array}\right\} \quad [\![v]\!] = JOIN(v) \cup vars(E) \tag{2.5}$$

$$\text{Output statements: } \textbf{\textit{output}}\ E: \quad [\![v]\!] = JOIN(v) \cup vars(E) \tag{2.6}$$

$$\text{Exit node: } exit: \quad [\![exit]\!] = \emptyset \tag{2.7}$$

To exemplify what these equations signifies, we will cover the equation 2.3, which is for assignments. This equation tells us that the variables within the set $[\![v]\!]$ directly subsequent to the assignment operation is the set of live variables before the assignment without the variables that were written to in union with the variables in the expression on the right-hand side, denoted as $vars(E)$. For nodes not encompassed by any of the specified equations, the constraint rule simplifies to the following:

$$[\![v]\!] = JOIN(v)$$

With these constraint rules we can perform the live variable analysis which can be used by a compiler to optimize code with the knowledge of which variable that are live at a certain

point in the program [11]. In this thesis, we will implement a similar dataflow analysis to LVA but instead will be able to detect information flow violations.

2.2.4 IntraJ

IntraJ is a framework for *intraprocedural* control flow and dataflow analysis for Java. It is implemented using the ExtendJ compiler and IntraCFG, which is a language-independent framework for control flow. IntraJ adds necessary interfaces, such as *CFGRoot* and *CFGNode*, to different AST nodes so that it works with Java in order to construct a Control Flow Graph [15]. We will use the intraprocedural CFG constructed from IntraJ in our analysis to perform a dataflow analyis.

In conjunction with a constructed Control Flow Graph, IntraJ includes several dataflow analyses, among them Live Variable Analysis, Reaching Definition Analysis, and May Null Analysis. One noteworthy analysis is the Live Variable Analysis, elaborated upon in Section 2.2.3. This analysis is notably facilitated by the inherent CFG structure. It involves the determination, for each CFG node, of whether a variable is referenced after the node and prior to reaching the CFG's ***exit*** node.

A important aspect of IntraJ pertains to its implementation, which relies on the extensible ExtendJ framework, supported by the **JastAdd** framework. Consequently, IntraJ's architectural flexibility facilitates the incorporation of additional dataflow analyses via the established CFG structure. To accomplish this, one can introduce additional *aspects* employing new *RAGs*, discussed more in sections 2.3.4 and 2.3.3.

Intraprocedural and Interprocedural Analysis

Intraprocedural and *Interprocedural* analysis are two different approaches for statically analyzing programs. Intraprocedural analysis focuses on examining the behaviour of a single procedure or method, i.e. it only looks at a specific procedure without considering how it may interact with other parts of the program. Conversely, interprocedural analysis involves analysing how different procedures and methods interact with each other in a program.

Performing interprocedural analysis poses a significant challenge, as it requires precise identification of the exact method being invoked. In object-oriented languages like Java, this determination is not always statically apparent. For instance, classes can override an inherited method and the specific instance of the class may not be known at compile-time. To enable interprocedural analysis despite such challenges, an approximate representation of an interprocedural control flow graph, known as the *call graph*, is employed. This graph outlines all possible functions that could be invoked from a given method call, providing a basis for this type of analysis. [11].

Various approaches can be employed to build the call graph, including methods such as *Class Hierarchy Analysis* (CHA), *Rapid Type Analysis* (RTA), and *Variable Type Analysis* (VTA) [11].

A tool that, similarly to IntraJ, is implemented using ExtendJ, is *Call Graph Analysis Tool* (CAT). This tool utilizes Class Hierarchy Analysis to construct call graphs. Class Hierarchy Analysis essentially analysis the class hierarchy to find suitable methods that may be the one that is invoked [3]. In order to do interprocedural analysis we will employ CAT in our analysis.

The example in Figure 2.3 shows how a call graph constructed from CHA might look like. Since a compiler at compile-time do not know whether the method call **a.m()** refers to the method **m()** in class **A** or class **B**, both calls are included in the call graph. In the Figure, the dotted line signifies potential calls while the solid line signifies the actual call, but note that both are included in the call graph.

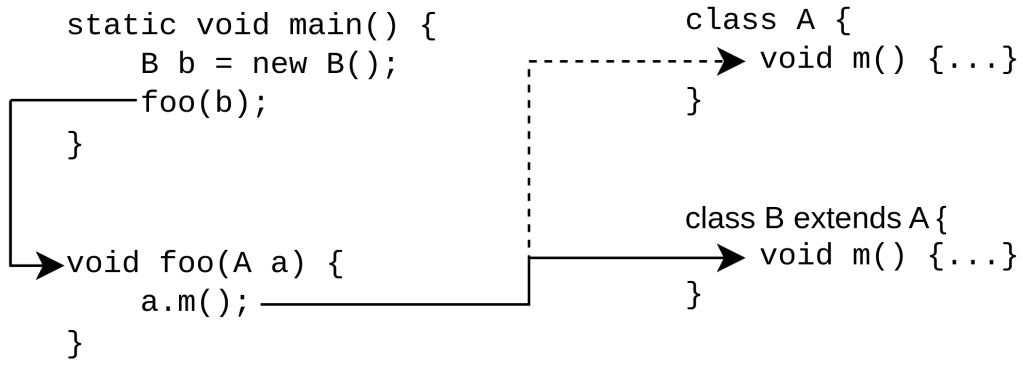


Figure 2.3: Example of a call graph constructed from Class Hierarchy Analysis.

2.3 Compilers

A compiler is a specialized software tool that translates source code written in a high-level programming language that is human-understandable into machine code or bytecode, which in turn can be executed by a computer's hardware. The compilation process involves several stages, including lexical analysis, syntax analysis, semantic analysis, code generation and code optimization. Each phase is responsible for specific tasks that leads to the final executable output [2].

2.3.1 Abstract Syntax Trees

The *Abstract Syntax Tree* (AST) is a hierarchical data structure that represents the syntactic structure of the source code. It is generated during the syntax analysis phase of the compilation process. The AST captures the essential elements of the source code, abstracting away irrelevant details like white space and comments, while preserving the relationships and hierarchy of the code's component [2] [11].

The AST is often used as an intermediate representation of the source code when proceeding to subsequent stages of compilation, such as semantic analysis and code generation. By using an AST, compilers can efficiently analyze and manipulate the structure of the code without the need to work directly with the raw source code text.

Each node in the AST corresponds to a specific language construct in the source code, such as variable declarations, expressions, control flow statements (if, while, etc.), function definitions and more. The nodes are organized in a tree-like structure, where parent nodes represent higher-level constructs, and their children nodes represent the sub-components of those constructs.

When constructing Abstract Syntax Trees (ASTs) using object-oriented programming languages, such as Java, it is customary to organize the tree structure within a class hierarchy. In this schema, an AST can effectively represent abstract nodes like "Statement" as abstract classes and concrete nodes like "Assignment" as concrete classes.

Consider the code seen in Figure 2.4. The resulting AST could for example be illustrated as in the same figure. Note that this is just a simplified example and in practicality a lot more nodes would exist.

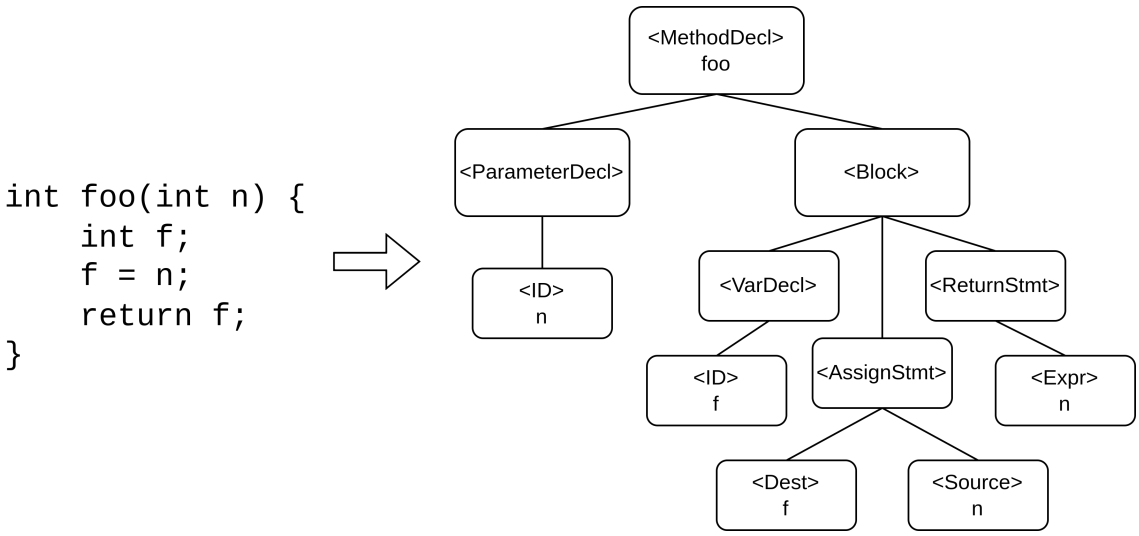


Figure 2.4: Abstract Syntax Tree of an example program

2.3.2 Semantic Analysis

The semantic analysis phase of a compiler is a vital step in the compilation process, focusing on understanding the meaning and structure of source code beyond its initial syntax. Key tasks include ensuring consistent data types, determining variable scope, constructing symbol tables, checking function overloading, simplifying constant expressions and performing other language-specific tasks. This phase bridges the gap between syntactic representation and intended program behavior, enhancing the compiler's understanding of the source code for subsequent optimization and code generation.

Many statically typed languages, such as Java or C++, utilize ASTs extensively for their semantic analysis. These languages often require a thorough analysis of types, scoping, and other semantic rules before generating machine code. By traversing the AST during the semantic analysis, the compiler can for example infer the types of operands or perform type coercion if necessary.

One of the more crucial analyses done in the semantic analysis phase is type checking. Essentially it involves examining the AST to ensure that the types of expressions and statements are used correctly according to the programming language's rules. The AST nodes are in some manner annotated with type information. A example is type checking an expression that produces a value from a computation. The compiler checks that the types of the operands are compatible with the operation being performed.

2.3.3 Reference Attribute Grammars

D.E Knuth introduced *Attribute Grammars* (AGs) in the article *"Semantics of Context-Free Languages"*, published in 1968 [8]. Attribute grammars provide a formal framework for specifying the relationships between the syntactic components of an AST and the associated attributes, enabling a structured approach to handling both the syntax and semantics of programming languages.

Attribute grammars have some key components. First, the syntax rules of a *context-free grammar* define the structure of a language through production rules that describe how symbols are derived from others. These rules establish the ASTs structure with the hierarchical arrangement of the language's constructs.

The second component, attributes, introduce an additional layer of information associated with the AST nodes. Each node possesses attributes that convey additional properties or characteristics related to the underlying syntactic construct. These attributes offer a means of encoding semantic information that goes beyond the syntactic structure.

Attribute equations constitute another essential component of attribute grammars. These equations define how attributes are computed based on the attributes of other nodes in the AST. By specifying these equations, developers can systematically determine how attributes are derived, facilitating a structured approach to semantic analysis and computation.

Central to Knuth's framework is the distinction between synthesized and inherited attributes. Synthesized attributes are computed at the given node and depend on information in the node or its children. In contrast, inherited attributes are propagated from parent nodes to their children, to provide information of e.g. the environment. This distinction allows for a comprehensive representation of information flow and dependencies within the syntax tree.

Reference Attribute Grammars (RAGs) are an extension to AGs in the way that it adds the feature that an attribute can be a reference to an object or a node in the AST. This feature allows a graph structure to be superimposed on the tree structure of an AST. It thereby en-

ables references from one node to e.g. the declaration of a variable that is far away in the tree. This is a natural extension to object-oriented ASTs and makes it easy to add declarative behaviour to the compiler [6] [7].

2.3.4 JastAdd

JastAdd represents a meta-compilation framework designed for the creation of compilers within a Java-based computational environment. Fundamentally, JastAdd consists of declarative programming, employing Reference Attribute Grammars (RAGs), along with the incorporation of imperative programming manifested as conventional Java code. This serves the purpose of implementing the modular construction of compilers [7].

JastAdd allows methods and fields for AST nodes to be defined modularly in separate files called *aspects*. During the compilation process, JastAdd interweaves these aspects alongside the regular Java code to generate an AST. This architectural design enhances the framework's capacity for the inclusion of new functionalities, e.g. new type-checking routines, by encapsulating them within dedicated modules presented in the form of aspects.

As mentioned, JastAdd is partly based on the use of RAGs, but it also has support for a few other types of attributes such as *Higher-Order attributes* [18], *Collection attributes* [9], *Node type interfaces* [5] and *Attribution aspects* [7]. An additional characteristic of the JastAdd framework pertains to its on-demand attribute evaluation methodology, wherein attribute computations are triggered solely upon their utilization, complemented by an optional incorporation of memoization to store and retrieve previously computed results.

Another important type of attribute that JastAdd supports and that are pertinent to e.g. dataflow analysis, are *Circular attributes* [10]. These are attributes that may exhibit dependencies on their own values. Circular attributes incorporate declarative fix-point computations, rendering them suitable for the computation of dataflow properties.

To illustrate how RAGs can be implemented using JastAdd, consider the AST in figure 2.5.

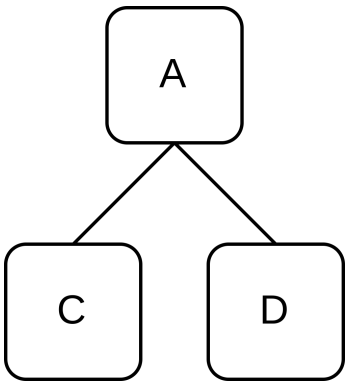


Figure 2.5: Abstract Syntax Tree Example

In JastAdd, a synthesized attribute of ***A*** can be written as follow:

```
syn int C.x();
eq C.x() = 1;
```

In AST nodes of type ***C***, the attribute ***x*** will assume the value of 1, whereas the other nodes, ***A*** and ***D***, do not posses the attribute ***x***. If we now instead assume that ***C*** and ***D*** are sub-types of an abstract node ***B***, we can define different equations for these nodes.

```
syn int B.x() = 1;
eq C.x() = 2;
```

The attribute ***x*** now has the value 1 for all nodes that are sub-types of type ***B***, but for AST nodes of type ***C*** the value of ***x*** is 2. This example illustrates that a synthesized attribute can be initially computed at the node level and subsequently this holds for all sub-types, or it can be overwritten by a sub-type node.

An inherited attribute can be expressed in JastAdd as the following:

```
inh int D.y();
eq A.getD().y() = 2;
```

In the current context, the value of the attribute ***y*** assumes the value of 2 for nodes of type ***D*** that also are children to a node of type ***A***. Now again assume that ***C*** and ***D*** are sub-types to the abstract node ***B***.

```
inh int B.y();
eq A.getC().y() = f() + 1;
eq A.getD().y() = g() + 1;
```

Here you can see how the value of ***y*** is defined by the parent node and differs depending on the context of the parent, assume ***g***() and ***f*** () are attributes in the context of the parent node ***A***.

2.3.5 ExtendJ

ExtendJ is a Java compiler developed using the JastAdd framework. It is built to be extensible, allowing developers to customize and augment the language processing capabilities to meet their specific needs. It provides a Java method API for compiler-based information [14].

One of the strengths of ExtendJ lies in, as stated earlier, its extensibility. By integrating new language constructs or experimental features, ExtendJ provides a ground for research, innovation, and exploration in language design and compiler construction. With this in mind, we have chosen to utilize ExtendJ to implement our analysis.

As this thesis is written, ExtendJ supports Java 8. Using JastAdd, ExtendJ has defined the necessary AST nodes along with RAGs that enables various semantic analysis, such as node types, bindings, types, compile-time errors and corresponding byte-code. Because of the modular aspects in JastAdd, it is simple to add new node, attributes or analyses to the compiler.

Chapter 3
Design of SinfoJ

With the goal of detecting security vulnerabilities associated with information flow, we introduce **SINFOJ**, a client analysis for Java in form of a forward dataflow analysis. The implementation of SINFOJ is inspired by JFLOW and was made possible with the help of tools and frameworks, including EXTENDJ, JASTADD, INTRAJ and CAT.

Within the framework of this analysis, a pre-defined set of *security classes* is integrated, allowing for the assignment of these security classes to different variables and methods. The central objective of this analysis is to pinpoint occurrences where information flow violates established constraint rules. The precise specifications of these constraints follows in subsequent sections.

This chapter will provide comprehensive descriptions of the various components comprising SINFOJ. Furthermore, we will clarify the underlying design decisions that have been made with theory from the Background Section 2 as a foundation.

3.1 Monotone framework

As explained in Section 2.2.3, the initiation of a dataflow analysis necessitates specifying a monotone framework. To accomplish this goal, we establish a complete lattice with finite height, along with a designated set of monotone functions. Figure 3.1 illustrates this lattice as a *Hasse Diagram* in order to visualize the increasing restrictiveness of the security classes. The directed arrows signifies that an entity with a lower security class is allowed to flow into an entity of a security class but not vice-versa, this will be discussed in more detail further on.

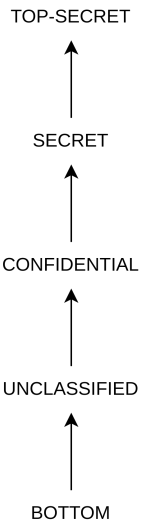


Figure 3.1: Hasse diagram over the described lattice

$$\boldsymbol{SC} = \{\mathit{BOTTOM}, \mathit{UNCLASSIFIED}, \mathit{CONFIDENTIAL}, \mathit{SECRET}, \mathit{TOPSECRET}\} \tag{3.1}$$

$$\boldsymbol{SC}_i \rightarrow \boldsymbol{SC}_j \quad \mathit{iff.} \quad i \leq j \tag{3.2}$$

$$\boldsymbol{SC}_i \sqcup \boldsymbol{SC}_j \equiv \boldsymbol{SC}_{max}(i, j) \tag{3.3}$$

$$\boldsymbol{SC}_i \sqcap \boldsymbol{SC}_j \equiv \boldsymbol{SC}_{min}(i, j) \tag{3.4}$$

$$\bot = \textsc{Bottom}, \top = \textsc{TopSecret} \tag{3.5}$$

The specified complete lattice describes a linear partial order among the security classes within the set denoted as $\boldsymbol{SC}$. In this abstract domain, the security classes are pre-defined. In contrast to the decentralized label model used in JFlow, as discussed in section 2.1.2 and 2.1.3, the inclusion of principals is essentially omitted. This simplification aims to further simplify the implementation of the analysis, focusing on studying how an information flow analysis can be realized using RAGs and ExtendJ, rather than delving into the intricacies of handling multiple principals.

This approach diverges from JFlow in several ways. Firstly, the security classes are pre-defined. Secondly, this approach does not employ owners and readers as JFlow does, which makes the analysis less powerful in terms of usage. This design makes the analysis into some sort of a taint analysis, with the distinction that it has multiple security classes beyond just *tainted data* and *non-tainted data*. It is once again worthy to acknowledge that while this design choice may limit adaptability to specific security requirements of a program, it significantly simplifies the implementation.

The monotone functions expressed in Equation 3.3 and 3.4 describes how these pre-defined security classes can be combined and compared within this framework. In SinfoJ, the *meet*

operator, denoted as $\sqcap$, yields the least restrictive security class, as defined in Equation 3.3. This can be illustrated through cases such as Unclassified $\sqcap$ Secret = Unclassified. Conversely, the *join* operator, denoted as $\sqcup$, produces the most restrictive security class when two are combined, as defined in Equation 3.4. This is exemplified in scenarios like Confidential $\sqcup$ Secret = Secret. Furthermore, within the lattice structure, the highest point, denoted as $\top$, is defined as the security class TopSecret, while the lowest point, designated as $\bot$, is defined as the security class Bottom.

The specified monotone functions, 3.2 - 3.4 play a pivotal role in maintaining the order delineated in the lattice. They are instrumental in enforcing the security constraints essential for identifying information flows that violates the established security constraints.

3.2 Dataflow analysis

In the context of the established monotone framework, we can undertake the implementation SinfoJ. The initial phase of the implementation involves an intraprocedural analysis, denoting an analytical process that considers the propagation of information within a procedure, e.g. a method.

3.2.1 Labels

In SinfoJ, variables exhibit dual attributes, namely, a conventional data type akin to those encountered in Java programs (e.g., *int*, *boolean*, *string*), as well as a distinct security-class classification that we refer to as a *label*. A variable x's label is denoted by $\underline{x}$. This label signifies what sensitivity a variable manifests. The label corresponds to a lattice element outlined in Section 3.1. With this information the analysis can maintain secure information flows that adhere to the lattice structure. To further clarify the purpose of this, consider the following examples, where X and Y signifies some variables in a program:

$$\underline{X} \rightarrow \underline{X} = \text{Secret} \rightarrow \text{Secret} \quad \Rightarrow \text{Valid flow} \quad \because \quad \text{Secret} \leq \text{Secret}$$
$$\underline{X} \rightarrow \underline{Y} = \text{Secret} \rightarrow \text{TopSecret} \quad \Rightarrow \text{Valid flow} \quad \because \quad \text{Secret} \leq \text{TopSecret}$$
$$\underline{X} \rightarrow \underline{X} = \text{Secret} \rightarrow \text{Bottom} \quad \Rightarrow \text{Not a valid flow} \quad \because \quad \text{Secret} \not\leq \text{Bottom}$$

In order to be able to perform an interprocedural analysis, methods also need to have labels. In fact, methods possess a few different labels in order to preserve the information flow constraints between methods. A method in SinfoJ has two labels, namely a *begin-label* and a *return-label*. The begin-label is an upper bound on the *caller-pc*, i.e. it is a restriction of when the method can be called upon. The return-label is a label of what information that will be returned from the method. This is the same as in JFlow, however, they also have a so called *end-label* for methods. This label specifies if information can be gained knowing whether the method terminates normally or not. As discussed in section 2.1.3, JFlow handles and checks termination paths in a manner which is rather complicated. It requires the need to check whether or not the program may terminate from an exception. It also requires the developer to explicitly list all exceptions that may be thrown. This part in JFlow has been omitted in our analysis for the same reason as omitting principals, i.e., to simplify the analysis.

Statements and expressions may also have labels, but this is only for the sake of the ease of implementation. One cannot explicitly label a whole statement or expression. In SinfoJ, a label of a statement or a expression is the label of the most restrictive label of the concerned objects in the statement. Consider the following example, $x = y + z$, where $\underline{y}$ = Secret and $\underline{z}$ = Unclassified. The expression $y + x$ then obtain the label Secret and the same applies to the whole statement if needed in the analysis.

Something that is worth mentioning regarding the lattice elements and labels, is that the Bottom security-class label is only used internally in the analysis. This means that a user cannot annotate a variable or method with this label, the lowest element they intend to annotate with is Unclassified. The reason for this is that the analysis needs to be able to differentiate between something being unlabeled and something having the label Unclassified, unlabeled variables will be discussed in the Section 3.2.2 below.

3.2.2 Default labels

In SinfoJ, developers have the option to abstain from labeling variables or methods in a program. In such cases, an unlabeled variable or method will be assigned a label via inference or given a default label. Here follows the rules for the default labels used in this analysis along with the labels of *Literals* and *Booleans*.

- **Default variable label**: The lattice $\bot$ element, i.e. Bottom, which implies that the variable can be used in any context.
- **Default parameter label**: The lattice $\top$ element, i.e. TopSecret. The reason for this being that the parameter may come from untrusted sources.
- **Default method begin-label**: The lattice $\top$ element, i.e. TopSecret. This implicitly means that the upper-bound on any method call is the top element, indicating that the method can always be called upon.
- **Default method return-label**: The join of all declared parameters. This is a common case, as a method often is the result of some computation on the parameters. If none are declared, the default label is the lattice $\bot$ element, Bottom. This means that the method's return value won't violate any information flows.
- **Default array label**: The lattice $\bot$ element, i.e. *Bottom*.
- **Literals and Booleans label**: The label of the *pc*, see Section 3.2.3.

In addition to using default labels, our analysis incorporates label inference, leveraging the properties of the lattice model detailed in Section 3.1. To elaborate, label inference assesses how security-class labels interact, utilizing the lattice structure and partial order relationships to deduce labels for a given variable. If an unlabeled variable with the label Bottom, is assigned a value, the analysis will infer the label of this variable with whatever label the right-hand side expression has. If the expression on the right-hand side has multiple operands, the label can be deduced by joining, $\sqcup$, the label's of the operands.

3.2.3 Explicit and Implicit flows

Within the scope of the intraprocedural analysis, our focus is directed towards two distinct forms of information flows, specifically denoted as *explicit* and *implicit* flows. These two flows were discussed upon in section 2.1. In our endeavor to address implicit flows, we have explored two approaches.

The first approach, used in JFlow, involves the introduction of a program-counter, denoted as *pc*, to accompany every statement and expression. This *pc* serves the purpose of monitoring information that can be gained if a given statement or expression were to be evaluated. The *pc* is inherently a set of variables which were evaluated and that also has affected that the execution has reached the program-point under consideration [12].

```
1 int x;                  // pc = {}
2 int y;                  // pc = {}
3 ...
4 int y = 0;              // pc = {}
5 if (x == 0) {           // pc = {x}
6     y = 1;              // pc = {x}
7 }                       // pc = {}
8                         // pc = {}
```

Figure 3.2: An implicit flow and the values of the *pc*.

Illustratively, let us consider the example depicted in Figure 3.2. In this example, $\underline{x}$ = Secret and $\underline{y}$ = Unclassified. Examine the value of the *pc* at each line. Up to and including line 4, the *pc* possesses an empty set as its value, denoted as $\{\}$. This signifies that no information from evaluated expressions and statements up to this point led to any gain in information. However, upon reaching line 5, we encounter an if-statement with the condition $x == 0$. Subsequent to the evaluation of this expression, the variable x is added to the set in *pc*. This indicates that any statements, such as assignments or method calls, within this if-statement is influenced by the variable x in the conditional expression. In other words, an implicit flow emanates from x to any entity residing within the if-statement, such as the expression $y = 1$ at line 6. The literal value 1 obtains has the same label as the *pc*, which is the highest label of any variables seen in the set. In this scenario that is $\underline{x}$ = Secret. Following the exit of the if-statement, the *pc* reverts to its initial value of $\{\}$, since no information regarding the condition on line 5 affects information flow here after. By retaining this information within the *pc*, the analysis can detect implicit flows arising from a conditional construct.

Another approach, which at first may seem advantageous, involves the incorporation of a backward dataflow analysis within conditional constructs. Consider an example program consisting of a conditional statement, S, with a condition c, followed by some other statements affected by the condition.

$$c \; ; \; S_1; \; \ldots; \; S_n;$$

When the analysis reaches a conditional statement, e.g. an if-, while- or for-statement, the statement S assumes the label $\underline{S} = \underline{S_1} \sqcap \cdots \sqcap S_n$. In other words, the label of the entire conditional statement S assumes the same label as the least restrictive label of the affected statements $S_1, \ldots, S_n$. Subsequently, the analysis proceeds to execute a label-check operation on the information flow $\underline{c} \rightarrow \underline{S}$. This serves the purpose of enforcing that the implicit flow adheres to the allowed information flows and that it does not violate any of the constraints established in the monotone framework.

In the example in Figure 3.2, this would mean that an the if-statement would assume the label $\underline{S} = \{ \underline{y = 1} \} =$ Unclassified. A label-check is then performed on the flow $\underline{c} \rightarrow \underline{S}$, which evaluates to Secret $\rightarrow$ Unclassified. This is an information flow violation since Secret $\not\leq$ Unclassified.

The second approach holds greater mathematical appeal; however, its implementation is rendered nontrivial due to the current configuration of IntraJ's CFG we employ. As mentioned in Section 2.2.3, a backward analysis starts at an exit-node in a CFG. The CFG we use, as currently structured, features an exit-node positioned solely at the end of a method. Implementing a backward analysis of this nature would require the incorporation of exit-nodes at the end of relevant conditional structures, a modification that is not easily integrated within the current CFG framework. Furthermore, an additional consideration pertains to the challenge of generating informative compile-time warning messages under this approach. The analysis, when following the second approach, only possesses knowledge of the potential violation of an implicit flow of the affected statements, without discerning the precise location within the source code. In the example in Figure 3.2, an error would be detected on the label-check at line 5 and not on line 6 where the information flow violation actually occur. As a result, it becomes slightly more complicated to provide meaningful warning messages.

Consequently, the approach centered around the pc is adopted for this analysis, as it offers greater ease of implementation and practicality.

3.3 Constraint rules

As discussed in Section 2.2.3, a typical forward dataflow analysis comprises both an *in* set and an *out* set that contains knowledge at the entrance respectively immediately after a CFG node. Additionally, it involves a *join* function responsible for merging abstract states from the CFG node's predecessors in the CFG, as well as a *transfer* function that transforms the state at the given node. This has been applied in this analysis and the *join* function, where v signifies a CFG-node, is defined as follows:

$$JOIN(v) = \bigcup_{w \in pred(v)} [\![w]\!] \tag{3.6}$$

The definition of the transfer function is dependent upon the CFG node under consideration. To clarify further, let's delve into the most interesting constraint rules governing information

flow, namely those associated with assignments and declarations.

$$\text{Assignment:} \quad X = E: \qquad [\![v]\!] = JOIN(v) \bigsqcup \underline{X} \sqcup \underline{E} \sqcup \underline{pc} \tag{3.7}$$

$$\text{Declaration:} \quad T\ X: \qquad [\![v]\!] = JOIN(v) \bigsqcup \underline{X} \sqcup \underline{pc} \tag{3.8}$$

In Equation 3.7, the symbol X represents the destination for the assignment, while E designates the source. The constraint also defines the transfer function for an assignment. The transfer function here can be seen if we rewrite the equation on the following form: $[\![v]\!] = t_v(\,JOIN\,(v)\,)$, where t_v signifies the transfer function. Upon closer examination of the equation, it becomes evident that the constraint conveys that the abstract state subsequent to an assignment, is the state immediately preceding the assignment, in union with $\underline{X}$, $\underline{E}$, and $\underline{pc}$ — in other words, the most restrictive label associated with all three entities.

Equation 3.8 outlines the definition of the constraint rule for a declaration. Here, T represents the data type of the declared variable X. This constraint rule is defined as the abstract state immediately preceding the declaration, in union with $\underline{X}$ and $\underline{pc}$. It is worth noting that in the context of a declaration, a right-hand side expression may exist. In such cases, it can be construed as a declaration followed by an assignment statement.

It is also noteworthy to highlight that within this analysis, the inclusion of the pc label remains a consistent component within the transfer functions for all constraint rules.

3.3.1 Label-checking rules

Enforcing and validating information flows present a comparatively straightforward endeavor. As stated earlier variables and methods have their own labels which embodies their security class. With these labels we can perform label checks where needed. Much like type checking in a regular compiler, the label-checking checks whether or not the labels under consideration are compatible.

Java constructs that are interesting here are various assign-statements, encompassing both normal assign-statements and declarations coupled with right-hand side expressions, declarations, method calls and procedural terminations, e.g. returns or throw-statements.

Assignments

In the context of assignments, SInfoJ assesses whether the information flow from the source to the destination aligns with the security constraints of the monotone framework. For instance, if we consider the following segment of Java code:

```
1 int x = y;
```

Assume $\underline{x} = \underline{y}$ = Confidential and $\underline{pc}$ = Secret. The assignment would result in an information flow violation since the label of the right-hand side expression is equal to Confidential ⊔ Secret, which yields the label Secret. A label check is then performed on the

information flow Secret → Confidential, which in turn would lead to a warning since Secret ⋢ Confidential.

Another design choice, made to enhance the analysis, was made to be able to keep track of what type of information that has flowed into various variables in the program. To illustrate this point, let's contemplate the following scenario:

```
1 String password = "password";          // TopSecret
2 String s1 = "Hello,";                  // Unclassifed
3 String s2 = " World!"                  // Unclassifed
4
5 s2 += password;
6 s1 += s2;
```

In this example, $pc = \{\}$, i.e. it has the label Unclassified. This example will result in the generation of two warnings, one on line 5 and another on line 6. The first warning is relatively straightforward, as it involves the flow of information from the variable *password*, labeled TopSecret into the variable $s2$, labeled Unclassified. However, the warning on line 6 presents a more nuanced scenario. Here, the variable $s1$, initially labeled as Unclassified, receives information from the variable $s2$, also labeled Unclassified, which instinctively may seem safe. Nevertheless, since $s2$ has received information from the variable *password*, it contains information with the label TopSecret. Consequently, this program breaches the permissible information flow constraints. This violation can be demonstrated using the following equation:

$$\begin{aligned} \underline{s2} \sqcup \underline{pc} \rightarrow \underline{s1} \quad &= \quad \underline{s2} \sqcup \underline{password} \sqcup \underline{pc} \rightarrow \underline{s1} && \equiv \\ \textsc{Unclassified} \sqcup \textsc{TopSecret} \sqcup \textsc{Unclassified} &\rightarrow \textsc{Unclassified} && \equiv \\ \textsc{TopSecret} &\rightarrow \textsc{Unclassified} \end{aligned}$$

The information flow from TopSecret → Unclassified is not allowed due to the hierarchical relationship where Unclassified is less restrictive than TopSecret.

Declarations

In the context of declarations, the requirement of label-checks arises primarily in cases where a right-hand side is present. In such instances, the label-check procedure mirrors that which is conducted during an assignment, discussed in the previous section. A second circumstance necessitating label-checks for declarations pertains to their susceptibility to implicit flows. To illustrate this point, consider the following example:

```
1 int x;                 // pc = {}
2 ...
3 if (x == 1) {          // pc = {x}
4     int y;             // pc = {x}
5 }                      // pc = {}
```

In this example, $\underline{y}$ = TopSecret, $\underline{x}$ = Secret and therefore, $\underline{pc}$ = Secret on lines 3 and 4 due to the conditional statement. Consequently, the declaration occurring on line 4 becomes subject to the influence of an implicit flow originating from the condition expressed on line 3. Specifically, an implicit flow, denoted as $\underline{x} \rightarrow \underline{y}$, necessitates a subsequent label-checking procedure since the variable y implicitly receives information with the label Secret. In this example, the label-check would not produce a warning since Secret $\leq$ TopSecret.

Procedural terminations

Additional program points where label-checks are essential encompass those which lead to termination of a procedure, including return- and throw-statements. To illustrate the significance of label-checks in these contexts, examine the example in Figure 3.3.

```
1  public int run() throws Exception {
2      int x;                          // x: Secret
3      ...
4      if (x < 0) {
5          throw new Exception();
6      }
7
8      if (x == 0) {
9          return -1;
10     }
11
12     return x;
13 }
```

Figure 3.3: Example of return and throw statements affected by implicit flows.

In the example depicted in Figure 3.3, $pc = \{x\}$ on lines 4, 5, 8, and 9. The potential presence of implicit flows within the if-statements introduces the potential for information leakage upon method termination via the throw-statement on line 5 and the return-statement on line 9. This necessitates the implementation of a label-checking mechanism at these program points. Conversely, the return-statement on line 12 diverges in that it does not entail an implicit flow; rather, it returns the variable x, which is labeled Secret. Notably, if the method `run()` is assigned a return-label which is less restrictive than Secret, each of the discussed statements leads to information flow violations.

The potential of information flow violations through exceptions exists in scenarios where a variable, with a more restrictive security label than the return-label of the enclosing method, is passed as an argument to an exception. This could potentially lead to that argument leaking to less restricted variables where the exceptions is caught. Figure 3.4 shows an example of this scenario. In this example, the method `run()` has the return-label Unclassified. We can note that a problem emerges when the variable x is passed as an argument to the throw-statement on line 6. This action gives rise to an information flow, denoted as $\underline{x} \rightarrow \underline{\texttt{run}}$, which

violates the established label hierarchy, as Secret $\not\leq$ Unclassified. Note that the same violation does not occur on line 11 since this exceptions is caught. Consequently, this example shows an information flow violation where a variable with a more restrictive label is passed as an argument to an operation or construct which leads to a termination of a procedure. This necessitates the enforcement of a label-check mechanism to address such scenarios.

```
1  public int run() throws Exception {
2      int x;                                  // x: Secret
3      int y;                                  // y: Unclassified
4      ...
5      if (y < 0) {
6          throw new Exception(x);             // -> Warning!
7      }
8
9      try {
10         if (x < 0) {
11             throw new Exception(x);         // -> No Warning!
12         }
13     } catch (Exception e) {
14         // thrown exception caught
15     }
16
17     return y;
18 }
```

Figure 3.4: Example of a throw statements with an argument that has a more restrictive label than the method along with an example of a caught exception.

On lines 9 ... 15 of the example in Figure 3.4, a try-catch is illustrated. In this situation, the throw-statement is influenced by an implicit flow from the condition on line 10. This would result in an information flow violation if there were no catch-clause in this example to handle the thrown exception. However, the thrown exception doesn't lead to a termination and therefor it does not violate the same information flow violation as the throw-statement on line 6.

Arrays

Due to the static nature of this client analysis, it presents certain limitations when it comes to examining the individual labels of array elements. Consequently, dealing with labels for arrays and their constituent elements becomes a non-trivial task. To address this challenge while upholding the integrity of the label hierarchy and ensuring secure information flows, a solution has been devised.

In this analysis, arrays are assigned a single label that is inherited by all elements contained within that array. This approach results in the creation of a label that serves as an over-approximation of the potential labels that individual array elements may possess. This method

is often referred to as a *may analysis* in static analysis since it describes information that may be true [11]. In particular, this approach acknowledges the possibility that certain elements within the array may possess the over-approximated label assigned to the entire array while other elements do not. Nonetheless, for the purposes of maintaining the integrity and reliability of a sound static analysis, all elements are treated as if they in fact do have this over-approximated label.

```
1 public int run() throws Exception {
2     int x;                    // x: Secret
3     int a, b, c;              // a, b, c: Unclassified
4     ...
5     int[] list = {x, a}       // list: Secret
6
7     b = list[0];              // -> Warning!
8     c = list[1];              // -> Warning!
9 }
```

Figure 3.5: Example of a how the label of an array is over-approximated.

Due to the aforementioned over-approximation of an array label, the array ***list*** in the example in Figure 3.5, obtains the label Secret since it contains the element x. Consequently, all array accesses within ***list*** are treated as though they may potentially possess the label Secret label, e.g. the variable a with the actual label Unclassified.

On lines 7 and 8 of the example, two distinct array accesses are illustrated. The array access on line 7 attempts to retrieve the first element, which corresponds to the variable x whose actual label is Secret. As this assignment should not be permitted based on the actual label, the over-approximated label for the entire array aligns with this restriction. Conversely, the array access on line 8 is directed towards the 1-indexed element within the array. This access depends on our ability to ascertain the static label of the array's element, which, in practice, is not achievable in a static analysis. Consequently, the over-approximated label assigned to the array serves as a safeguard, producing a warning in such accesses to ensure compliance with the underlying security constraints. In general for all accesses of this array, the following information flow is true:

$$\underline{list[i]} \rightarrow l \;=\; \textsc{Secret} \rightarrow l \quad , where \;\; i < \text{list.length}, \;\; l \text{ is an arbitrary label}$$

Methods and Method calls

As mentioned in section 3.2.1, methods possesses two labels, a begin-label and a return-label. The begin-label has the purpose of maintaining secure information flows when the method is called upon. The return-label has the purpose of maintaining secure information flows when the method terminates as well as being the approximated label of the returned value to where the method was called. In addition to the two labels for a method, the parameters of the method also come with their own labels, indicating what variables that can be used as

```
1  // Return-label: Secret
2  // Begin-label: Secret
3  public int add(int x, int y) {       // x, y: Unclassified
4      return y + x;
5  }
6
7  public void run() {
8      int m1 = add(1, 2);              // -> No Warning!
9
10     boolean x = true;                // x: TopSecret
11     int m2;                          // m2: Unclassified
12
13     if (x) {
14         m2 = m(1, 2);                // -> Warning!
15     }
16 }
```

Figure 3.6: Example of method calls.

arguments in a method call.

In Figure 3.6, we have two methods, `add(int x, int y)` and `run()`. The first of the two has a return-label Secret, a begin-label Secret and two parameters, x and y. In `run()` two method calls to `add(int x, int y)` is made, one on line 8 and one on line 14. For the first call, the *pc* at the call site, also called the caller-pc is empty and the arguments 1 and 2 are both literals with the same label as the *pc*'s. This means that the call does not violate any constraints since the flow $\underline{\text{pc}} \rightarrow$ Secret is allowed. However, the call on line 14 violates the information flow constraints since it's affected by the fact that caller-pc = $\{x\}$. This flow, TopSecret $\rightarrow$ Secret, is not allowed. Furthermore the return-value from the method call has the label Secret which leads to the flow in the assignment of Secret $\rightarrow$ Unclassified, and since Secret $\not\leq$ Unclassified this violates the established constraints.

3.4 Client Analysis

In order to enable the analysis described in Section 3.2, we have developed and implemented a client analysis called SinfoJ. This analysis has been realized with the help of ExtendJ, RAGs (facilitated by JastAdd), IntraJ and Cat, all discussed in 2.

Utilizing the JastAdd framework, our analysis was instantiated using declarative RAGs in various aspects, discussed in section 2.3.4. Additionally, several regular Java classes were implemented to handle different data structures or other tasks more suitable for Java, such as the use of custom Java annotations. This entire setup was then integrated as an extension to the ExtendJ Java compiler.

Due to space constraints, we won't delve into the entire implementation, but the most note-

worthy aspects of the analysis involve the sections pertaining to the actual information flow analysis on the Control Flow Graph. To implement these sections, we utilized IntraJ to construct and navigate trough the control flow graph. The following are three important attributes that implement the described dataflow analysis in section 3.2.

```
syn Alpha CFGNode.IF_in() circular[new Alpha()] {...}
eq Entry.IF_in() = new Alpha();

syn Alpha CFGNode.IF_out() circular[new Alpha()] {...}

syn Alpha CFGNode.IF_trFun(Alpha alpha) = alpha;
```

`Alpha` is a custom HashMap that stores variables and their labels, this corresponds to $[\![v]\!]$ for a CFG-node. Additionally, `Alpha` includes the $join_v$ function, responsible for combining information flow from the predecessors of v. The in_v and out_v functions, discussed in section 2.2.3, compute sets of information flow labels for various variables at the entrance and exit of the CFG-node v. Since this is a forward analysis, it commences at the entry-node, as indicated above. The transfer function is specified for the required CFG-nodes, detailed in section 3.2. Both the in_v and out_v attributes are circular, implying that they may depend on themselves and compute a fix-point.

Another noteworthy aspect of the implementation is the segment dealing with the pc discussed in section 3.2.3. This was implemented in a manner similar to the information flow dataflow analysis just discussed. The following are key attributes for computing pc.

```
syn Beta CFGNode.PC_in() circular[new Beta()] {...}
eq Entry.PC_in() = new Beta();

syn Beta CFGNode.PC_out() circular[new Beta()] {...}

syn Beta CFGNode.PC_trFun(Beta beta) = beta;
```

Just like `Alpha`, `Beta` is a custom HashMap, but it differs by tracking variables that have been observed and impact the information flow at the specified CFG-node. The implementation principles for in_v, out_v, and the transfer function for pc align with those used in the overall information flow dataflow analysis. This means that the transfer-function is implemented for expressions and variable accesses that are conditions in conditional statements, such as if-, for- and while-statements.

To be able to make the interprocedural analysis possible, i.e. controlling information flows regarding method calls, the class hierarchy analysis Cat discussed in Section 2.2.4 is utilized. In practice one attribute is especially important, namely the `allDecls()` below.

```
syn Set<InvocationTarget> Invocable.allDecls(){...}
```

This attribute returns a set of all declarations of the specific `Invocable`, which is something that can be invoked or called, in our case a method call. With this set we can approximate which method declaration is called upon and through that, approximate the method's begin-, return- and parameter-labels.

3.4.1 Annotations

To enable security-class labeling of a program, Java custom annotations have been utilized. These annotations serves as a practical means by which developers can annotate variables and methods as needed. Consequently, with the integration of these annotations, developers can label their Java programs as necessary without the need to use special syntax.

The custom Java annotations are located within a package called **org.extendj.infoflow.utils**. Figure 3.7 provides an illustrative demonstration of the practical application of these annotations. As depicted in the figure, the method `bar()` is assigned the return-label Secret and the begin-label Unclassified. The variable x is assigned the label TopSecret, and the uninitialized variable y is assigned the label Secret. Upon subjecting this program to analysis, two warnings would be produced. Firstly, a warning would manifest in response to the assignment operation on line 15, where $\underline{x} \rightarrow \underline{y}$ lead to an information flow violation, as TopSecret $\not\leq$ Secret. Secondly, another warning would emerge concerning the return statement on line 17, where $\underline{y} \rightarrow \underline{\texttt{bar()}}$, delineates an information flow violation. This violation is rooted in the fact that y now has TopSecret information, while `bar()` has the return-label Secret.

There exists two different types of annotations in the package **org.extendj.infoflow.utils**. A collection of annotations used for variables and parameters, namely **@Unclassified**, **@Confidential**, **@Secret** and **@TopSecret**. One for each security-class in the lattice model, except *BOTTOM*, which is not intended to be used for labeling. The other type of annotations are the ones intended to be used for method declarations, namely **@BeginLabel(LabelDomain label)** and **@ReturnLabel(LabelDomain label)**. These are used to declare a method's begin- and return-label. They incorporates a parameter, namely a Java enum called **LabelDomain**. This **LabelDomain** enum, also located within the package **org.extendj.infoflow.utils**, encompasses a set of constants, each corresponding to one of the security classes defined within the lattice model. Additionally, **LabelDomain** has functionalities facilitating the comparison of the constants.

3.4.2 Warnings

The fundamental objective of this client analysis is to provide developers with a understanding of how data flows within their program with regard to the potential existence of information flows that violates the lattice properties delineated in Section 3.1. To give informative insights, the analysis is designed to generate warnings whenever it detects flows that breach the permissible information flow.

It is important to emphasize that these warnings are intended to provide information and do not interfere with the regular compilation and execution of the program. In the presence of potential information flow errors, the program can proceed with its normal compilation and execution if there are no compilation errors. This approach is chosen to streamline the development process while also providing developers with feedback concerning potential information flow security vulnerabilities.

```
1  import org.extendj.infoflow.utils.LabelDomain;
2  import org.extendj.infoflow.utils.InfoFlowLabel;
3
4  public class Foo
5  {
6      @BeginLabel(label=LabelDomain.UNCLASSIFIED)
7      @ReturnLabel(label=LabelDomain.SECRET)
8      public void bar() {
9
10         @TopSecret int x = 1;
11         @Secret int y;
12
13         y = x;
14
15         return y;
16     }
17 }
```

Figure 3.7: Example of how to use SinfoJ in a program with the package **org.extendj.infoflow.utils**.

Given the inherent complexity and the approximations employed by this analysis, it is anticipated that a substantial number of warnings may be generated. Consequently, some of these warnings might be considered superfluous for certain developer use cases, and as such, it is necessary for the developer to exercise discernment when interpreting these warnings in the context of their specific scenarios.

Chapter 4
Evaluation

The upcoming chapter outlines how the analysis was tested and evaluated. First, we will confirm that our analysis is a functional subset of the JFlow language by comparing it to the Jif test suite. This will be followed by an performance evaluation with two different benchmarks. The purpose of this evaluation is to determine whether our analysis performs within a reasonable time compared to the standard compilation time for ExtendJ and similar analyses implemented with IntraJ, such as the *Null-Pointer-Analysis* (NPA).

4.1 Tests

As stated, we compare our analysis with the Jif test suite, which consists of 626 tests. Since Jif has a more complex security-class lattice than our framework, which we outlined in section 3.1, we made necessary adjustments to their test cases. These modifications should not affect the test validity and should still produce the same information flow results.

Among the 626 test cases in Jif, many involve syntax and concepts not implemented in SinfoJ. This is largely due to their use of the full decentralized label model, as discussed in section 2.1.2. This model introduces additional complexity and features such as *acts for*, *principals*, and *declassification* [1]. Additionally, Jif supports runtime and generic labels. However, in cases where tests exclude these more advanced concepts and focus solely on labels and basic information flows, our implemented analysis effectively addresses almost all of them. When rewriting the test cases that do not include syntax SinfoJ does not support, we can confirm that SinfoJ handles $\boldsymbol{X}$ of the test-cases.

It's important to highlight that while Jif is more complex than our analyis and supports the complete JFlow language, it is implemented with a total of **38.616** lines of code (LOC). In contrast, our implementation comprises only **650** LOC of standard Java code, and **675** LOC of JastAdd code.

4.2 Performance

Evaluation Setup and Methodology

The benchmarks were performed on a machine with a Intel Core i7-8665U running at 1.90GHz and 16GB RAM. The machine ran Ubuntu 22.04.03 LTS and the benchmarks were executed on OpenJDK Runtime Environment 8.0.275.fx-zulu. The analysis was implemented using JastAdd version 2.3.6, which was the latest version available at the time of this publication.

We first performed measurements for start-up performance, cold-starting the Java Virtual Machine (JVM) for each run. Then we performed measurements for steady-state performance, with a single measurements after 24 warm-up runs. Each benchmark iteration was then repeated 25 times, resulting in a total of 625 runs for steady-state. The report metrics include median values along with the 95% confidence intervals. The two benchmarks used can be seen in table 4.1.

Benchmark	LOC	Version
pmd	60749	4.2.5
jfreechart	95664	1.0.0

Table 4.1: The Java benchmarks used for evaluation.

Benchmark	Start-up			
	ExtendJ	CFG	NPA	SinfoJ
	Time(s)	Time(s)	Time(s)	Time(s)
pmd	$1.07_{\pm 0.03}$	$5.35_{\pm 0.13}$	$8.90_{\pm 0.43}$	$15.65_{\pm 0.53}$
jfreechart	$1.27_{\pm 0.02}$	$5.17_{\pm 0.07}$	$10.72_{\pm 0.37}$	$18.43_{\pm 0.27}$

Table 4.2: Benchmark results for Start-up measurement.

Benchmark	Steady-state			
	ExtendJ	CFG	NPA	SinfoJ
	Time(s)	Time(s)	Time(s)	Time(s)
pmd	*N/A*	$1.49_{\pm 0.11}$	$2.95_{\pm 0.14}$	$6.51_{\pm 0.16}$
jfreechart	*N/A*	$1.50_{\pm 0.07}$	$4.22_{\pm 0.20}$	$10.16_{\pm 0.19}$

Table 4.3: Benchmark results for Steady-state measurement.

Results

The benchmark evaluation results are presented in Table 4.2 and 4.3. They show the average time for running the entire respective benchmark. As stated earlier, we aim to show that our analysis performs within reasonable execution time. As shown in the table, SinfoJ exhibits slower performance than NPA, but note that the difference is not unreasonably large. Looking at the steady-state measurement for SinfoJ compared to NPA we see that it is a factor of **2.21** for the *pmd* benchmark and **2.41** for the *jfreechart* benchmark. This outcome is expected, considering that NPA is a relatively simple and strictly intraprocedural analysis. In contrast, SinfoJ incorporates the ***pc***, the information flow intraprocedural analyses and the integration of CAT, which introduces a bit of interprocedural analysis.

Even though the analysis was not designed with the primary goal of optimizing performance, the results indicate that, in comparison to the already established and tested NPA, it performs quite well.

Chapter 5
Limitations and Further Work

In this section we will discuss how our analysis could be further improved and developed. We will also address some of the problems we faced when implementing this analysis within the scope defined in the introduction and which limitations this resulted in.

The goal of this thesis was not to implement the full JFlow information flow analysis, however we hope that the work done with this thesis can contribute to further research within the area of information flow analysis and RAGs. The goal for further research would be to implement the entire JFlow language since there are a lot of feature still missing in our analysis. The most interesting feature of JFlow which would be a good starting point when further developing this analysis, would be to extend our analysis to support the decentralized label model. This more complex model enables programs to have more advanced security-models. It does not necessarily mean a more advanced implementation due to that the dataflow analysis in this analysis will remain the same. In a lattice model that supports the decentralised label model, every lattice element would consist of a set of policies, mentioned in Section 2.1.2. The lattice model would no longer be linear with increasing restrictive security-classes as the one we use. A variable with a label L_1 can flow into another variable with a label L_2 if and only if for all policies in L_1, there exist a policy in L_2 that is at least as restrictive. This means that L_2 is at least as restrictive as L_1 and therefore information can flow from the variable with label L_1. The monotone framework described in Section 3.1 would then need to be updated to conform with this. The entire decentralized label model and it's lattice model is described in the paper *"A lattice model of secure information flow"* by Denning, Dorothy E [4] and in the paper *"Complete, safe information flow with decentralized labels"* by A.C. Myers and B. Liskov [13].

Another interesting feature in JFlow worth exploring is the way JFlow examine termination paths. JFlow keeps track of where a program might terminate due to exceptions or potential run-time errors. This leads to a more precise information flow analysis, since the analysis can catch information flow violations resulting from these potential terminations.

Our analysis only identifies terminations that are explicitly declared, such as return statements or throw-statement. However, it does not identify potential violations in situations where run-time exceptions like null-pointer or division-by-zero could occur. A solution here could be to utilize e.g. the Null Pointer Analysis already existing in IntraJ in order to detect potential terminations due to null-pointers.

In terms of performance there exists a lot of room for improvement. First and foremost, the implementation of the *pc* leads to one more dataflow analysis being performed. Our analysis basically consists of two independent dataflow analysis, one for information flow states and one for the *pc*. This could potentially be incorporated into one dataflow analysis. This area and solution has not been explored but would be interesting and a good starting point for performance optimization.

The data structure used for storing states in both the *pc* and the information flow analysis are Java HashMaps. Which are effective from one point of vies, since most of the operations performed are retrieving and updating elements, which have an average time complexity of $O(1)$. Although, when we want to retrieve the expression with the most restrictive label in the pc, a linear search is performed on the HashMap, which obviously isn't ideal. A very simple solution would be to keep track of what expression that has the most restrictive label, which we update when inserting or removing values.

Another aspect that would be interesting to further investigate is how we could utilize JastAdd and it's on-demand evaluation to further improve the performance of this analysis and analysis of this kind.

5.1 Java Annotations

The use of Java annotations made it possible to implement this analysis without the need to extend Java syntax. This enabled more focus on the actual dataflow analysis. However, as already discussed, we did never implemented the entire decentralized label model as in JFlow. While it might be theoretically feasible to implement this model using Java annotations, it is likely to be less practical in reality. To elaborate, the decentralized label model permits a label to have multiple policies, which in turn allows multiple readers. When labels are implemented with annotations, this would lead to either excessively lengthy annotation declarations or the use of multiple annotations for a labeled object. This approach therefor becomes less practical, for example in scenarios such as parameter declarations within a method declaration.

Furthermore, Java annotations are intended to be used for providing a compiler with metadata that can be used during compile-time or run-time. A feature JFlow and Jif allows is the use of generic and run-time labels, which would be difficult to implement with the fact that values given to annotations should be available at compile-time.

5.2 Interprocedural Analysis

The interprocedural part of this analysis can also be further developed to make it more precise. Right now, we utilize *CAT* in order to construct a call graph, which we then use to approximately know which method is called. This is then used to retrieve the labels for that method and check that they do not violate any information flow constraints. However, as the labels serve as approximations from either annotations or inferred labels, a bit of precision is lost. One improvement would be to use an interprocedural Control Flow Graph instead of an intraprocedural, which IntraJ is. This would give our dataflow analysis the ability to preserve information between processes, i.e. between methods.

Another way to deal with this loss in precision due to approximations of the interprocedural flows, would be to inline the information flow states from the point of the call-site to the entry-node of the method. Then at the termination of the method combine the states with the information at the program point of the caller. This proves to be rather non-trivial due to the nature of RAGs. An equation in JastAdd cannot result in any visible side effects. This means that in the equations were we specify the dataflow analysis, we cannot set the attributes for other AST nodes.

Chapter 6
Conclusion

This thesis has introduced a static information flow analysis inspired by the JFlow language. It was implemented using RAGs and utilizes JastAdd, ExtendJ, IntraJ and CAT. The analysis is capable of detecting basic information flow violations in intraprocedural and interprocedural programs. The dataflow analysis that is used follows the monotone framework discussed in Section 3.1. With this we were able to implement a simplistic version of the JFlow language with labels using Java Annotations. Although this leads to some limitations and restrictions, which were discussed previously, this decision made the it easier to implement and the work could be more focused on the actual analysis. The use of Reference Attribute Grammars (RAGs) proved to be rather effective. Implementing the dataflow analysis was rather trivial when inherited and synthesized attributes could be utilized. Furthermore, the constraint rules for various constructs, e.g. assignments and declarations were easily implemented and if needed, it's easy to add more rules to other Control Flow Graph nodes. If one wants to add further label-checks or more constraint rules for the dataflow analysis one simply adds a synthesized attribute for that specific CFG node. The same implies if more label-checks is needed, then it's easy to add attributes for the specific Java constructs.

As the implemented analysis is rather simplistic and does not allow for very complex security models, it might not be practical in real-life scenarios. However, this analysis lays the groundwork for further development of both this particular analysis or for using RAGs in general. With this we hope that our work can contribute to others that wish to implement an information flow analysis or further research in the area of Reference Attribute Grammars.

References

[1] Jif reference manual. `https://www.cs.cornell.edu/jif/doc/jif-3.3.0/manual.html`. Accessed: 2023-12-05.

[2] Andrew W. Appel and Jens Palsberg. *Modern compiler implementation in Java.* Cambridge University Press, 2002.

[3] Jeffrey Dean, David Grove, and Craig Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In *European Conference on Object-Oriented Programming*, 1995.

[4] Dorothy E Denning. A lattice model of secure information flow. *Communications of the ACM*, 19(5):236–243, 1976.

[5] Niklas Fors, Emma Söderberg, and Görel Hedin. Principles and patterns of jastadd-style reference attribute grammars. In *Proceedings of the 13th ACM SIGPLAN International Conference on Software Language Engineering*, SLE 2020, page 86–100, New York, NY, USA, 2020. Association for Computing Machinery.

[6] Görel Hedin. Reference attributed grammars. *Informatica (Slovenia)*, 24(3):301–317, 2000.

[7] Görel Hedin and Eva Magnusson. Jastadd—an aspect-oriented compiler construction system. *Science of Computer Programming*, 47(1):37–58, 2003. Special Issue on Language Descriptions, Tools and Applications (L DTA'01).

[8] Donald E Knuth. Semantics of context-free languages. *Mathematical systems theory*, 2(2):127–145, 1968.

[9] Eva Magnusson, Torbjorn Ekman, and Gorel Hedin. Extending attribute grammars with collection attributes–evaluation and applications. In *Seventh IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2007)*, pages 69–80. IEEE, 2007.

[10] Eva Magnusson and Görel Hedin. Circular reference attributed grammars—their evaluation and applications. *Science of Computer Programming*, 68(1):21–37, 2007.

[11] Anders Møller and Michael I. Schwartzbach. Static program analysis, October 2018. Department of Computer Science, Aarhus University, `http://cs.au.dk/~amoeller/spa/`.

[12] Andrew C Myers. Jflow: Practical mostly-static information flow control. In *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 228–241, 1999.

[13] Andrew C Myers and Barbara Liskov. Complete, safe information flow with decentralized labels. In *Proceedings. 1998 IEEE Symposium on Security and Privacy (Cat. No. 98CB36186)*, pages 186–197. IEEE, 1998.

[14] Jesper Öqvist. Extendj: Extensible java compiler. In *Companion Proceedings of the 2nd International Conference on the Art, Science, and Engineering of Programming*, Programming '18, page 234–235, New York, NY, USA, 2018. Association for Computing Machinery.

[15] Idriss Riouak, Christoph Reichenbach, Görel Hedin, and Niklas Fors. A precise framework for source-level control-flow analysis. In *2021 IEEE 21st International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 1–11. IEEE, 2021.

[16] A. Sabelfeld and A.C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, 2003.

[17] Omer Tripp, Marco Pistoia, Stephen J. Fink, Manu Sridharan, and Omri Weisman. Taj: Effective taint analysis of web applications. *SIGPLAN Not.*, 44(6):87–97, jun 2009.

[18] H. H. Vogt, S. D. Swierstra, and M. F. Kuiper. Higher order attribute grammars. *SIGPLAN Not.*, 24(7):131–145, jun 1989.

Appendices

EXAMENSARBETE Application Specific Instruction-set Processor Using a Parametrizable multi-SIMD Synthesizeable Model Supporting Design Space Exploration
STUDENT Magnus Hultin
HANDLEDARE Flavius Gruian (LTH)
EXAMINATOR Krzysztof Kuchcinski (LTH)

Parametrisk processor modell för design utforskning

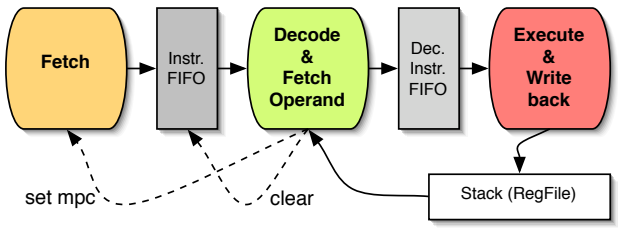
POPULÄRVETENSKAPLIG SAMMANFATTNING **Magnus Hultin**

Applikations-specifika processorer är allt mer vanligt för få ut rätt prestanda med så lite resurser som möjligt. Detta arbete har en parametrisk modell för att kunna testa hur mycket resurser som behövs för en specifik applikation.

För att öka prestandan i dagens processorer finns det vektorenheter och flera kärnor i processorer. Vektorenheten finns till för att kunna utföra en operation på en mängd data samtidigt och flera kärnor gör att man kan utföra fler instruktioner samtidigt. Ofta är processorerna designade för att kunna stödja en mängd olika datorprogram. Detta resulterar i att det blir kompromisser som kan påverka prestandan för vissa program och vara överflödigt för andra. I t.ex. videokameror, mobiltelefoner, medicinsk utrustning, digital kameror och annan inbyggd elektronik, kan man istället använda en processor som saknar vissa funktioner men som istället är mer energieffektiv. Man kan jämföra det med att frakta ett paket med en stor lastbil istället för att använda en mindre bil där samma paketet också skulle få plats.

I mitt examensarbete har jag skrivit en modell som kan användas för att snabbt designa en processor enligt vissa parametrar. Dessa parametrar väljs utifrån vilket eller vilka program man tänkta köra på den. Vissa program kan t.ex. lättare använda flera kärnor och vissa program kan använda korta eller längre vektorenheter för dess data.

Modellen testades med olika multimedia program. Den mest beräkningsintensiva och mest upprepande delen av programmen användes. Dessa kallas för kärnor av programmen. Kärnorna som användes var ifrån MPEG och JPEG, som används för bildkomprimering och videokomprimering.



Resultatet visar att det finns en prestanda vinst jämfört med generella processorer men att detta också ökar resurserna som behövs. Detta trots att den generella processorn har nästan dubbelt så hög klockfrekvens än dem applikationsspecifika processorerna. Resultatet visar också att schemaläggning av instruktionerna i programmen spelar en stor roll för att kunna utnyttja resurserna som finns tillgängliga och därmed öka prestandan. Med den schemaläggningen som utnyttjade resurserna bäst var prestandan minst 79% bättre än den generella processorn.