INFORMATION FLOW IN A JAVA INTERMEDIATE LANGUAGE

by
Ahmer Ahmedani

School of Computer Science
McGill University, Montréal

August 2006

A THESIS SUBMITTED TO THE FACULTY OF GRADUATE STUDIES AND RESEARCH
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE

Abstract

It is a common practice to retrieve code from an outside source, execute it and return the result to the user. During execution secure data can enter the program by user input or access of a data resource. It is important to track the secure data once it enters the program to identify possible information flows to unwanted regions of the code which would permit undesirable data output to a user. Most approaches to restrict information flow in programs have fallen short of providing a practical solution for mainstream programming languages.

To address this issue, this thesis presents two context-sensitive inter-procedural analyses which analyze an intermediate representation of Java Bytecode for secure information flow. The first analysis assumes that there is only one instance of all class fields where as the second analysis uses points-to information to differentiate between instance fields which belong to different instances of the same class. The analyses track secure information in the program by maintaining sets of secure data. The analyses resolve dynamic method resolution in Java statically by analyzing all possible methods which may be invoked at a call site and merging the secure data sets. We were able to define rules to analyze all the statements in the intermediate representation and also accounted for Java libraries. The analyses do not expect any security annotations in the program.

Type information is useful in debugging, guiding optimizations, and specifying and providing safety proofs for programs. A type system for a subset of the Java Bytecode intermediate representation is also formulated in this thesis. An operational semantics is specified and a type preservation proof assures the soundness of the type system.

Non-trivial benchmarks were analyzed and explicit and implicit information flows were counted for both analyses. The empirical data collected suggests secure data is used in many statements of programs and output of data to a user at several places in a program

can lead to information flow if the user does not have the right permission to observe the data.

Résumé

Il est commun d'obtenir du code d'une source externe, de l'exécuter et de retourner le résultat à l'utilisateur. Pendant l'exécution, des données sensibles peuvent êtres entrées le programme par l'utilisateur ou l'accès à une autre source de donnée. Il est important de faire un suivi des données sensibles lorsqu'elles entrent dans le programme pour garder la trace du flot de l'information et ainsi identifier la circulation possible de données sensibles vers des régions du code qui pourraient permettre à ces données d'êtres vues par des utilisateurs non concernés. Les approches communes pour restreindre le flot de l'information dans les programmes n'ont pas été a la hauteur pour ce qui est de fournir une solution pratique pour les langages de programmation les plus utilisés.

Pour adresser ce problème, cette thèse présente deux analyses inter procédurales sensibles au contexte qui analysent une représentation intermédiaire du code compilé Java pour le flot sécuritaire de l'information. La première analyse assume qu'il n'y a qu'une instance de tous les champs des classes alors que la seconde utilisent une analyse des références a fin de différencier entre les champs d'instance qui appartiennent à différentes instances de la même classe. L'analyse conserve la trace de l'information sensible dans le programme en maintenant des groupes d'information sensible. L'analyse résout les appels de méthodes dynamique en Java de faon statique en analysant toutes les méthodes qui pourraient être invoquées en un site d'appel et en combinant les groupes d'information sensible. Nous avons défini des règles pour analyser toutes les expressions dans la représentation intermédiaire et pris en compte les librairies Java. Les analyses n'ont pas besoin d'annotations de sécurité dans le programme.

L'information sur les types est utile pour le débogage, pour guider les optimisations, et pour spécifier et fournir des preuves de la sécurité de programmes. Un système de type pour

un sous-ensemble du code compilé Java est aussi formulé dans cette thèse. Une sémantique opérationnelle est spécifiée et une preuve de préservation des types assure la consistance du système de type.

Des tests non triviaux ont étés analysés et le flot implicite et explicite de l'information a été compté pour les deux analyses. Les données empiriques collectées suggèrent que les données sensibles sont utilisées dans plusieurs expressions des programmes et les sorties de données peuvent mener à des brèches de sécurité si l'utilisateur n'a pas les permissions correctes pour observer les données.

Acknowledgements

I am grateful to my thesis supervisors for their help, support, and encouragement. Clark Verbrugge was always eager to discuss my work and suggest improvements. He was very kind and patient, and I have a lot of respect for him as a teacher. Brigitte Pientka introduced me to the challenging area of logic and type systems.

A very warm thanks to Laurie Hendren, the head of the Sable Research Group. She was always ready to share her experience with me and suggest ideas on my work. I started my thesis work as a project in the Optimizing Compilers class I took with her.

I would also like to thank the teachers with whom I took classes at McGill: Bettina Kemme, Doina Precup, Godfried Toussaint, Martin Robillard and Sue Whitesides.

I had a very good time in the Sable Research Group and would like to thank all the members who were always eager to brainstorm ideas. In particular, I would like to thank Ondřej and Jennifer Lhoták, Christopher Goard, Patrick Lam, Sokhom Pheng, Dayong Gu, Michael Batchelder, Greg Prokopsky, and Feng Qian for the discussions and Christopher J F Pickett for letting me use his Backward Control Flow analysis in my thesis work. Thank you Maxime Chevalier-Boisvert for translating the thesis abstract into French.

My time at McGill was made memorable because of the very special friends I made while studying at McGill and I will always remember their help, guidance and companionship. In particular, I would like to thank Ahmad Shawky El-Behery, Ahmed Shehada, Hani Ezzadeen and Waqqas Khokhar.

I would like to specially mention Nomair Ahmed Naeem with whom I spent a lot of time working on assignments and projects ever since my undergraduate days at McGill. It was fun discussing our work for hours. Good luck with your PhD at Waterloo.

My family has always been close to me even though they were far away in Pakistan. I

would like to thank my brother, sister-in-law, fiancée and specially my parents for believing in me and encouraging me during my studies at McGill.

All praise is due to God.

Table of Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1
Introduction

1.1 Motivation

Data security is of utmost importance in todays global computing world. Programs which are executed on machines are obtained from various sources and mechanisms are employed to enforce specific security policies as per the need of the computing system. One security related issue is *information flow* of secure data to areas in the program source which give out information to users. The output given out to users has to be controlled in order to make sure that information regarding secure data is not given out to unwanted users.

Information flow leaks are present in code due to the fact that data resources (such as databases) only check for permission to access data from them. They perform access control by giving out data after making sure the user has the required permission to obtain the data. Once the data has been retrieved from the resource by the program source, the data resource does not control the propagation of data in the program which can result in data flowing to insecure areas in the program, which may be read by users without the right permissions. Secure information may leak to variables which do not have a security level equal to or more than that of the data by way of explicit or implicit flows.

Medical and military are two important areas where secrecy and privacy are very important. Confidential medical data about a patient or military plans are restricted for use by relevant people. In such cases it is important that the flow of information is tracked and the

leaks are identified and corrected. This ensures the integrity and correctness of the program using the secure data.

Data, at different security levels, flows through programs and we need mechanisms which enforce information flow security in program code. The Java programming language, in recent years, has gained in popularity and its *Bytecode* can be executed on any machine with a Java Virtual Machine. It is very common for users to obtain Bytecode from an outside source and execute it on their machine. The program may access secure data on the machine and so the code must be analyzed to ascertain if it can leak any secure data.

This thesis aims to address the problem of information flow in Java. We investigate information flow for single-threaded programs written in Java by defining *context-sensitive inter-procedural data-flow* analyses on a Java intermediate language called the `JIMPLE` Intermediate Representation (IR) of Java Bytecode in the Soot Bytecode analysis and optimization framework [VRGH+00, Soo]. All of the Java language features are considered in our analyses. The analyses examine all `JIMPLE` statements according to the defined rules which are designed to conservatively protect secure data. We present the data sets which are maintained by the each analysis and the algorithms which manipulate them.

Recently there has been a lot of interest in formalizing intermediate languages to preserve types after compilation of high-level code to low-level code. Typed intermediate languages also provide a platform to reason about the correctness of code transformations and optimizations. We formalize a small subset of the `JIMPLE` IR and formulate a type preservation proof for its type system. Thus we approach the problem from two directions: practical (data-flow) and theoretical (type systems).

In addition to the design and implementation of the data-flow analyses on `JIMPLE` and formalization of a subset of `JIMPLE`, this thesis presents the results of the experiments that were carried out for the data-flow analyses on moderate size, although non-trivial, benchmarks. The experimental data suggests that many program statements use secure data due to the conservative assumptions of the analyses. We consider several practical issues in this respect, illustrating some major sources of conservative imprecision.

1.2 Contributions

This thesis presents work that can be grouped into two categories: one in the area of information flow and the other in the area of formalizing an intermediate representation of Java. The contributions for each part are reported in the following two subsections.

1.2.1 Information Flow

Zdancewic [Zda04] pointed out that "the real challenge in information flow is to make use of all the work to apply to real applications and understand what are the real issues stopping us from achieving that goal". Keeping this goal in mind of having a practical solution for a widely used programming language we designed information flow analyses (described in detail in Chapter 5 and Chapter 6) for the Java programming language. The analyses have several noteworthy features.

- Our design provides a practical solution. Unlike other approaches we do not necessarily require user annotations for security level of variables.
- We provide a complete solution including all the statements in our program representation, as well as complex issues such as recursion and use of library code.
- This is a fine-grained approach since security is enforced at the level of the fields in objects and not the objects themselves.
- The rules for each statement are implemented in a modular fashion and allow for refinement easily.
- We consider both explicit and implicit information flows that occur due to conditional branching on secure data.
- Information leaks due to explicit exceptions are also tracked.
- As well as analyzing programs we provide a warning mechanism that alerts users to specific sites where secure data may leak.

- Our design builds on a sophisticated program analysis framework. As far as we know this is the first time points-to analysis has been used in a practical information flow analysis.
- We define and provide several metrics for measuring the quality of information flow results.

1.2.2 Formalizing an Intermediate Representation of Java

Type information is useful in debugging, guiding optimizations and specifying and proving safety proofs for programs. In order to achieve these properties in an optimizing compiler we need a typed intermediate representation. This thesis work describes a small typed intermediate representation of Java (described in detail in Chapter 4). The following is presented:

- formalization of a non-trivial subset of `JIMPLE` with if-statements, assignments statements and goto statements;
- operational semantics and type system for the subset; and
- a soundness proof for the type system.

1.3 Thesis Organization

The rest of this thesis is organized as follows. The next chapter defines the problem of information flow clearly and also presents the `JIMPLE` IR. Chapter 3 is a survey of related work. Chapter 4 presents the formalization of a subset of `JIMPLE`. The data-flow analysis design and details of implementation including data structures and algorithms used is presented in Chapter 5. Chapter 6 describes alternative approaches that improve upon type-based analysis, giving more precise information. The empirical results on the information leaks and metric calculations for several programs are given in Chapter 7. Finally, Chapter 8 concludes this work along with a description of ways to extend this work.

Chapter 2
Information Flow

It was pointed out in Chapter 1 that there are confidentiality issues with respect to information flow in programs. This chapter will explain the problem highlighting the core issue at hand in the first section. The second section presents the problem model and how security level of variables are considered in program code where as the last section defines the `JIMPLE` IR on which we carry out our investigations in this thesis.

2.1 Problem Statement

The problem occurs due to the execution of unverified code which is obtained from some outside source on a computer. Figure 2.1 captures this phenomenon in a diagram. The unverified program code is received by a user and then executed. During execution it is possible that there are inputs of secure data or accesses of secure data residing in some data resource. The program completes execution and then gives an output. It might be possible that partial information about secure data can be inferred by observing the program code and the output.

Figure 2.2 presents an example method which retrieves a student's record from a database to find out the faculty in which the student is enrolled. The student's faculty information is likely not confidential data but the method also retrieves the gpa of the student and uses the value to decide on the font of the result. Clearly the font will give away partial information about a student's gpa to the one executing the method. If we consider the gpa to be a

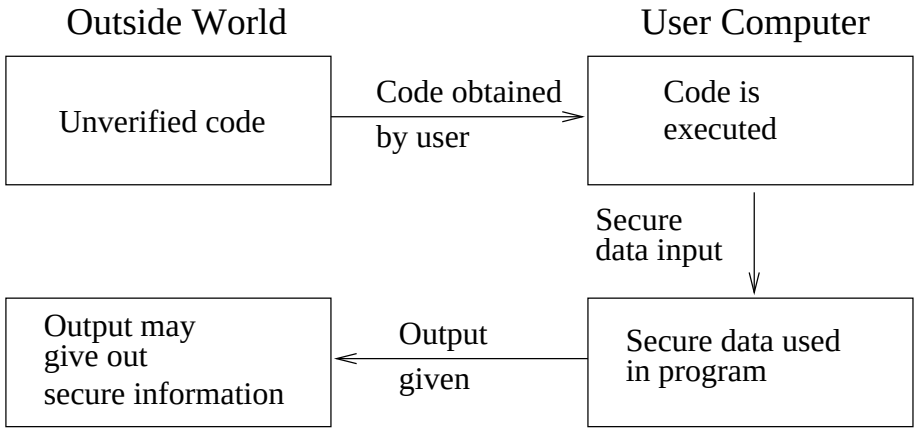

Figure 2.1: *Information Flow Problem*

confidential value which only approved people can see, there is a need to analyze code and check for possible information leaks.

```
String getStudentFaculty(int StudentID){
   StudentRecord rec := database[StudentID];
   String result := rec.faculty;
   if rec.gpa > 3.0 then result.font := "bold"
}
```

Figure 2.2: *Example Method*

In Java, Bytecode is commonly obtained from an outside source and executed on the local machine. Secure data may be used in the program and the output can give away information about the secure data.

2.2 The Model

The previous section gives an overview of the information flow problem. In this section the problem model is explained in detail along with the security levels description and the different kinds of information flows.

2.2.1 Security Levels Lattice

In order to ensure security of data in programs the variables which store data are assigned security levels and variables can only be assigned data which has a security level equal or less than their own. However, first we need to define a policy with a set of security levels and indicate what is more secure than the other. The security level policy used in most information flow work over the years was specified by Denning [Den76]. Security policies are defined by a complete lattice of security levels and information is permitted to flow from variables of a given security level to variables of the same or higher security levels only.

The set of security levels could have a linear or hierarchical structure but in both cases it should satisfy the lattice properties. In this research we use a linear security level model consisting of two levels of security: one is a high H security level and the other is a low L security level as shown in Figure 2.3. The linear model conforms to the properties required of a lattice. Our model is a trivial one but even more complex models specify a domain of security levels which is finite and the security levels are partially ordered. The models also have a lower bound, an upper bound and defintions for the least upper bound (LUB) and greatest lower bound (GLB) operators.

Lattice Properties for Our Model:

- The domain of security levels D = $\{L,H\}$.
- It is partially ordered - L is at a lower security level than H.
- It is finite with only two security levels.
- It has a lower bound L, where $L \subseteq \mathrm{A} \; \forall \; \mathrm{A} \in \mathrm{D}$.
- It has an upper bound which is H.
- The LUB and GLB operators are defined easily for a linear lattice.
  - LUB$\{$i,j$\}$ = max(i,j). In our case LUB$\{H,L\}$ = max(H,L) = H.
  - GLB$\{$i,j$\}$ = min(i,j). In our case GLB$\{H,L\}$ = min(H,L) = L.

Figure 2.3: *Lattice Model*

2.2.2 Explicit and Implicit Information Flows

An explicit flow occurs when a variable containing confidential data at a certain security is assigned to a variable whose security level is less than the security level of the data. In the example below we consider two security levels: restricted and unrestricted (corresponding to high and low respectively). The assignment of j to i is a clear leak of information since i's security level is lower then that of variable j.

```
int i,j; //i is unrestricted, j is a restricted variable
//variable j contains secure data and i is an
//unrestricted variable

i = j;
```

Implicit flow arises from the control structure of a program like in the case of an if-statement. In the example below the program branches depending upon the restricted data value stored in variable H. In the true branch variable L is assigned 1 whereas in the false branch variable L is assigned 2. Partial information about the value of variable H can be inferred by observing variable L. In this case it can be inferred whether or not H is equal to 4.

```
int L,H; //L is unrestricted, H has restricted data

if(H == 4){
    L = 1;
```

```
}else{
    L = 2;
}
```

In order to be certain that a program does not leak confidential data, the secure data must only flow to variables in programs which are not publicly read. The property which we are trying to enforce is *noninterference* and it is formally described as follows:

Definition of noninterference: One group of users, using a certain set of data, is non-interfering with another group of users if what the first group does with that data has no effect on what the second group of users can see [GM82].

2.2.3 Code and Data Visibility Model

The model we use in our work is presented in Figure 2.4 [1]. The attacker has access to the program source and is providing or can observe the insecure or publicly observable input which we call *Low in* corresponding to *L in* in Figure 2.4. The attacker can also see the publicly observable output *Low out* produced by the program after execution corresponding to *L out* in the figure. The attacker does not read the secure data *High in* corresponding to *H in* in the figure which enters the program if the program accesses some secure data resource.

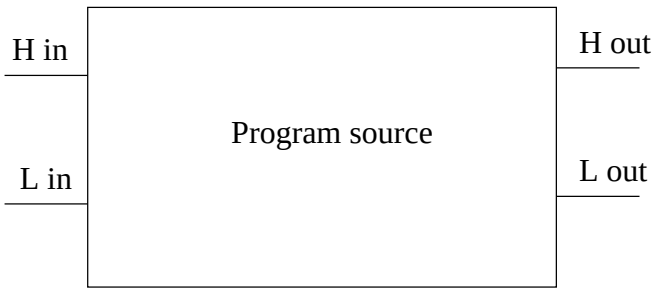

Figure 2.4: *Code and Data Visibility Model*

The value of the secure data may be used by the program source in computations. The attacker can figure out the secure value if for the same *L in* changing the *H in* causes the

[1] The model is inspired by a talk given by Anindya Banerjee

L out to change. In the following examples, in Figure 2.5 and Figure 2.6, we assume that variable h contains secure data where as variable l is publicly readable.

Figure 2.5 shows two cases in which no information leaks to the attacker. In example (a), no matter which branch of the if-statement is executed depending on the conditional expression, the same value will be observed at *L out*. Similarly in example (b) the difference of the secure value "h - h" will always result in the same value observed at *L out*.

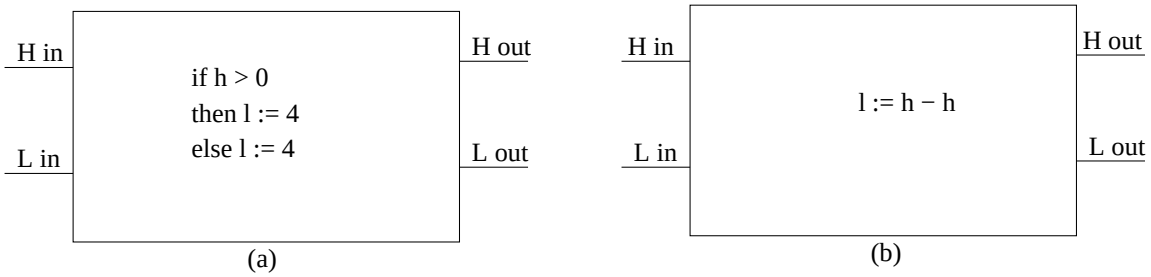

Figure 2.5: *Safe Examples*

Figure 2.6 consists of two examples which violate secure information flow. In example (a), the then and else branches of the if-statement generate different results. Since the conditional expression is dependant on the value of the secure variable h, the attacker can make out partial information about the value of h by observing the *L out*. Consider a simple case in which *L in* is 3 and *H in* is 0. Since variable h is 0, the else branch is executed and the statement l := l + 2 gives a *L out* result of 5. Suppose the *H in* changes to 1 now. In this case, the then branch statement l := l + 3 gets executed producing a *L out* result of 6. Clearly a change in the value of *H in* with *L in* remaining the same produces a different result which can be observed and partial information can be obtained about the secure value. This is a case of an implicit flow.

In example (b), the result of an addition involving a secure data value is assigned to an insecure variable which can be viewed at *L out*. In this case consider *L in* fixed at 2. Now for two different values (3 and 4) of *H in*, the addition will produce different results (5 and 6 respectively). This is a case of an explicit flow.

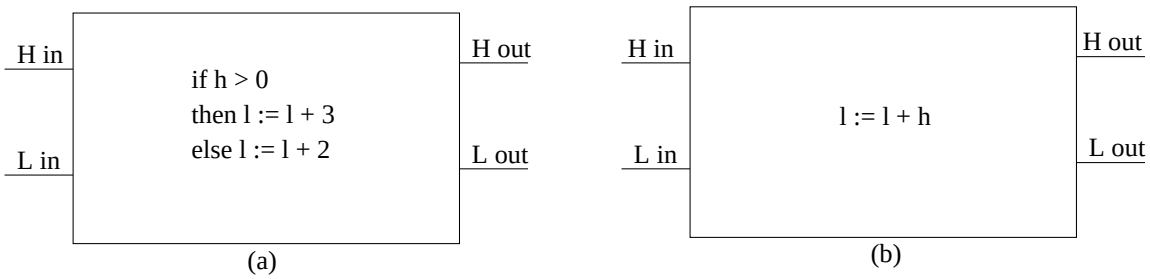

Figure 2.6: *Unsafe Examples*

2.3 Jimple Intermediate Representation

The problem model was presented in the previous section. In this section we describe the grammar of the JIMPLE IR that we base our investigations on in this research. We formalize a subset of JIMPLE (details in Chapter 4) and our information flow analyses based on the Code and Data Visibility Model given in Section 2.2.3 are defined for full JIMPLE (details in Chapter 5 and Chapter 6).

2.3.1 Jimple in the Context of Soot

Java is compiled into Bytecode which is executed by the Java Virtual Machine (JVM). The Bytecode generated can be optimized to achieve faster runtime efficiency. Sable Object-Oriented Toolkit (Soot) [VRGH+00, Soo] is a framework which processes the Java Bytecode to improve the code output by performing transformations, run optimizations and now also generates high-quality decompiled code. The code optimizations and transformations are carried out on intermediate representations of Java Bytecode. SOOT has four IRs namely: BAF, JIMPLE, SHIMPLE and GRIMP. In this research, we chose JIMPLE to formalize and write our flow analyses on.

Figure 2.7 presents an overview of the Soot framework. It accepts Java class files and translates the code into JIMPLE IR on which analyses, optimizations and code transformations are carried out. The analyses make use of the *call graph* and *Spark points-to* information available in Soot. After performing the analyses and optimizations the optimized code

is translated back into Bytecode which is executed by the JVM.

2.3.2 Advantages of Jimple

JIMPLE is a *3-address* IR of the Java Bytecode. It is a compact, stackless representation and provides considerable ease for writing compiler optimizations and analyses. Each 3-address instruction can be described as a quadruple (operator, operand1, operand2, result) and each statement has the general form of: x := y op z. The key feature of three address code is that every instruction implements exactly one fundamental operation. The main advantages of JIMPLE over Bytecode and other Soot IRs are:

- The expressions are directly available and there is no need to build an expression from the Bytecode instructions.
- An expression's code may not be available in continuous Bytecodes and so the analyses become very complex.
- In an IR with an operand stack, removing or changing the position of Bytecodes is cumbersome due to the fact that the stack height must be of a particular height across control flow boundaries.
- JIMPLE has very few instructions (19 in all) compared to the 200 different Bytecode instructions.
- All local variables (the declared as well as the operand stack ones) are typed which allows accurate and complex analyses to be written.

2.3.3 Jimple Grammar

We introduce the JIMPLE grammar [VR00] in this section and give part of it to illustrate how JIMPLE code looks like. Additionally the grammar for some JIMPLE constructs which have have no affect on the security of variables and are not part of the information flow analyses is not given here.

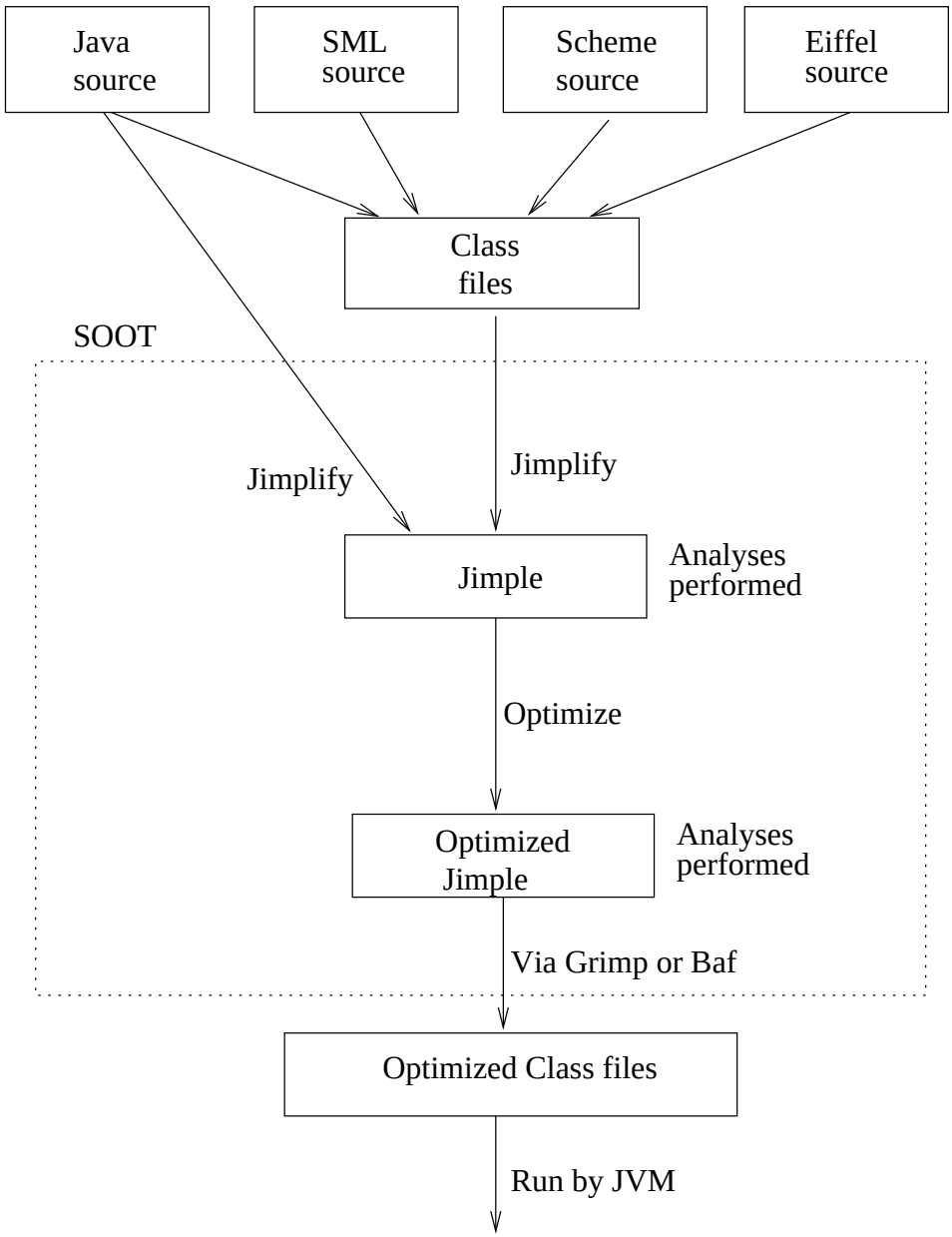

Figure 2.7: *Overview of Soot*

$$
\begin{array}{rcl}
\textit{assignStmt} & ::= & \texttt{local} = \textit{rvalue};| \\
& & \texttt{field} = \textit{imm};| \\
& & \texttt{local.field} = \textit{imm};| \\
& & \texttt{local}[\textit{imm}] = \textit{imm};
\end{array}
$$

The most commonly occurring statements in JIMPLE are the assignment statements since JIMPLE is a 3-address code and a variable is never assigned to more than once. All declarations occur at the beginning of a class and at the beginning of each method in the class. The assignment statement (assignStmt) has four variations. The first one assigns an expression value to a local, the second assigns a local or constant value to a static field, the third assigns a local or a constant value to an instance field and the last assignment statement assigns a local or a constant value to an array location. Field references or array references never occur in the same statement.

$$
\begin{array}{rcl}
\textit{identityStmt} & ::= & \texttt{local} := \texttt{@this.type};| \\
& & \texttt{local} := \texttt{@parameter}\textit{n}\texttt{:type};| \\
& & \texttt{local} := \texttt{@exception};
\end{array}
$$

The first two identity statements occur at the beginning of each method and they are necessary because there is no array of local variables in JIMPLE which has the information for the this and the parameters. The first one assigns the instance of the current class to a local variable of the class type. This statement is the first statement in all instance methods and constructors. The next one assigns all the parameters to locals of the appropriate type. Each method has a number of these statements equal to the number of parameters. The last identity statement may occur anywhere in a method and it assigns a caught exception object to a local of the exact same type.

$$
\begin{array}{rcl}
\textit{gotoStmt} & ::= & \texttt{goto label};| \\
\textit{ifStmt} & ::= & \texttt{if}\ \textit{conditionExpr}\ \texttt{goto label};
\end{array}
$$

The above two statements are control-flow statements in JIMPLE. The gotoStmt is an unconditional jump where as the ifStmt is a conditional jump on the conditionExpr. The switch statement is similar to the ifStmt but has several branches.

$$\textit{invokeStmt} \quad ::= \quad \texttt{invoke}\ \textit{invokeExpr};$$

The invoke statement invokes a method and the result of the invocation is not assigned to a local. In the case of a non-void method, the result is lost as described by the Java semantics.

$$\begin{aligned} \textit{returnStmt} \quad ::= \quad & \texttt{return}\ \textit{imm};| \\ & \texttt{return};\end{aligned}$$

The two kinds of return statements correspond to the return statements in methods with some return type and methods with void return type. In the first return statement, a local or a constant specify the return value.

$$\textit{throwStmt} \quad ::= \quad \texttt{throw}\ \texttt{local};$$

At a throw statement an exception is thrown and a local specifies the class of the exception that is thrown.

$$\begin{aligned} \textit{imm} \quad ::= \quad & \texttt{local}| \\ & \texttt{constant}\end{aligned}$$

The immediates are the local variables in a method or constant values.

$$\textit{conditionalExpr} \quad ::= \quad \textit{imm}_1\ \textit{condop}\ \textit{imm}_2$$

The conditional expression occurs in the if-statement and has two operators. There are six conditional operators in JIMPLE.

$$\begin{aligned} \textit{rvalue} \quad & ::= \quad \textit{concreteRef}|\textit{imm}|\textit{expr} \\ \textit{concreteRef} \quad & ::= \quad \texttt{field}| \\ & \qquad\quad \texttt{local.field}| \\ & \qquad\quad \texttt{local}[\textit{imm}]\end{aligned}$$

The rvalues are the right-hand side of an assignment statement. They could be any of the concrete references which are the static fields, instance fields or array accesses, the immediate values or the various expressions.

$$
\begin{aligned}
\textit{invokeExpr} \;::=\; & \texttt{specialinvoke}\ \texttt{local.m}(\textit{imm}_1,\ldots,\textit{imm}_n)\mid \\
& \texttt{interfaceinvoke}\ \texttt{local.m}(\textit{imm}_1,\ldots,\textit{imm}_n)\mid \\
& \texttt{virtualinvoke}\ \texttt{local.m}(\textit{imm}_1,\ldots,\textit{imm}_n)\mid \\
& \texttt{staticinvoke}\ \texttt{m}(\textit{imm}_1,\ldots,\textit{imm}_n)
\end{aligned}
$$

There are four kinds of invoke expressions in JIMPLE. The specialinvoke expressions are specific to init methods which are the constructor calls in JIMPLE. The interfaceinvoke and virtualinvoke expressions are the method invocations of instance objects where as the staticinvoke expressions invoke static methods. Note that m is the method signature, local points to the instance of the object and $\textit{imm}_i$ are the method arguments.

$$
\begin{aligned}
\textit{expr} \;::=\; & \textit{imm}_1\ \textit{binop}\ \textit{imm}_2\mid \\
& \texttt{(type)}\ \textit{imm}\mid \\
& \textit{imm}\ \texttt{instanceof type}\mid \\
& \textit{invokeExpr}\mid \\
& \texttt{new refType}\mid \\
& \texttt{newarray (type)}\ [\textit{imm}]\mid \\
& \texttt{newmultiarray (type)}\ [\textit{imm}_1]\ldots[\textit{imm}_n]\ []\star\mid \\
& \texttt{length}\ \textit{imm}\mid \\
& \texttt{neg}\ \textit{imm}
\end{aligned}
$$

The grammar for all the expressions in JIMPLE is as above. They occur on the right hand side of the assignment statement and the result of the expressions is assigned to a local variable.

Figure 2.8 presents the code of a Java program. The corresponding JIMPLE code is given in Figure 2.9. The static initializations (initialization of field i in the example) are in a clinit method in JIMPLE. The init method in JIMPLE corresponds to the class constructor. The important differences to observe between the main method code in Java and JIMPLE are that the structured if-statement in Java corresponds to the if-statement with goto in JIMPLE and the statements in JIMPLE are simple with no field references in conditional expressions. In the Java code we have a field access in the conditional expression where as in the JIMPLE code the field is first assigned to a local variable which is then used in

the conditional expression. In JIMPLE all declarations are at the beginning of the method, followed by the identity statements assigning parameters to local variables and then the other statements.

```
public class Example{
    private static int i = 10;
    public static void main(String[] args){
        int[] myArray = new int[12];
        int k=0;
        myArray[k] = 5;
        k = myArray.length;
        if(i < k) //field i used in conditional expression
            i = 1;
        else
            i = 2;
    }
}
```

Figure 2.8: *Example - Java*

```
public class Example extends java.lang.Object{
    private static int i;
    public void <init>(){
        Example r0;
        r0 := @this: Example;
        specialinvoke r0.<java.lang.Object: void <init>()>();
        return;
    }
    public static void main(java.lang.String[]){
        java.lang.String[] r0;
        int[] r1;
        int i0, i1, $i2;
        r0 := @parameter0: java.lang.String[];
        r1 = newarray (int)[12];
        i0 = 0;
        r1[i0] = 5;
        i1 = lengthof r1;
        $i2 = <Example: int i>; //field i assigned to a local variable
        if $i2 >= i1 goto label0;
        <Example: int i> = 1;
        goto label1;
     label0:
        <Example: int i> = 2;
     label1:
        return;
    }
    static void <clinit>(){
        <Example: int i> = 10;
        return;
    }
}
```

Figure 2.9: *Example - Jimple*

Chapter 3
Related Work

This chapter presents work previously done on information flow analysis. The first section is an overview of the early work leading to information flow analysis. The second section covers the compile-time analyses and techniques to check flow of information in a program. The third section explains several approaches that have been tried to enforce non-interference in Java related languages. The fourth section discusses prior work on typing low-level languages and formalizing the compilation process. An extensive survey of research on information flow, highlighting all the open areas of research, is given by Sabelfeld and Myers [SM03].

3.1 Early Work on Information Flow

The earliest work in the area of information flow can be attributed to Denning and Denning [DD77] where they define a simple language in which a security class is attached to all user-defined variables in a program. Denning also defined the concept of a set of security classes in the form of a lattice structure which defined the permissible and impermissible flows in a program statement [Den76]. A compile time check analyzes the statements in the program and ensures that information does not flow into objects whose security class will permit unauthorized data leaks. This was the first time that the actual program was analyzed for information flow since previous work concentrated on the input/output of data into a program. The language they consider is procedural with assignments, simple control

structures such as if-statements and while-statements, procedure calls, and explicit exceptions. In their work the exception handler has to be within the same procedure in which the exception may be generated.

One area of work concentrated on limiting access rights for a program. This means that it cannot read outside data or write the data outside the program unless the security classes of all the data read can be given out on any output in the program. The Case System [WSO+75] and the MITRE system [BL75, Mil76] are examples of such research attempts.

The second area of work analyzes programs to determine flows from the reads to the writes external to the program. The mechanism transforms the programs such that the flows are checked at run time accurately. Fenton's data mark machine [Fen74] and the surveillance protection mechanism of Jones and Lipton [JL75] employ this technique.

3.2 Analyses for Information Flow

Denning and Denning's compiler [Den76, DD77] only produced a pass or fail message for the program code and they gave no formal proofs of their approach. Volpano, Smith and Irvine's [VSI96] work centered around giving soundness proofs of Denning and Denning's language. They only took a subset of the language including assignments, if-statements, while-statements and the let-statement to better explain their proof technique of type soundness which they relate to noninterference. They leave out method calls and exceptions. They define two rules, no read up and no write down, and their type system rejects programs which do not observe the rules.

Heintze and Riecke [HR98] enriched the λ-calculus for information flow security and prove a noninterference theorem. They enhanced the type system by adding security properties to each type and they named it Secure Lambda Calculus (or SLam calculus). They cover the basic λ-calculus along with assignments and also include concurrency. While their work can be adapted for more high-level languages with complex type systems, there is a need to find a way to reduce the amount of type specifications that a user has to provide.

Miyamoto and Igarashi [MI04] proposed a typed lambda calculus $\lambda_s^{\Box}$ as a foundation for information flow analysis. Their type system corresponds to a proof system of an in-

tuitionistic modal logic of validity by the Curry-Howard isomorphism. Their type system is designed for a functional language with three new constructs in the grammar of a functional language. They are: u corresponding to the modal variables, $box_l\ M$ which envelopes a secure statement M and $let\ box_l\ u = M\ in\ M$ which only permits reading a secure statement which is boxed at the right security level. l are the security levels which are partially ordered. The type system successfully types only those programs which will not leak information. The rules enforce that a modal variable is only used when the level of the variable is lower than or equal to the level in which it is type checked. This prevents confidential information from flowing into insecure levels.

Pottier and Simonet [PS02] gave a type-based information flow analysis for a realistic sequential programming language. They call it Core ML since it consists of λ-calculus with references, exceptions and let-polymorphism. Their type system is based on type reconstruction and it is constraint-based since they allow for subtyping.

The type systems for information flow are normally conservative and sometimes even reject programs that would otherwise be safe to execute. In order to make sure that the type system correctly rejects a program Unno, Kobayashi and Yonezawa [UKY06] suggest combining model checking with type based analysis to find counterexamples that the program will indeed leak information. However, their method does not guarantee that a program does satisfy noninterference.

Amtoft, Bandhakavi and Banerjee [ABB06] presented a flow sensitive inter-procedural information flow analysis in object-oriented programs using a Hoare-like logic. They describe possible aliasing by region assertions and information flow properties by independence assertions. Method summaries are used to specify the assertions that must hold before and after a method call.

Noticing that most type systems were flow insensitive and that Denning and Denning's original analysis was also flow insensitive, Hunt and Sands [HS06] proposed flow sensitive security types; this increased the accuracy by it providing a snapshot of the security level of a variable at each program point. They presented flow sensitive typing for a simple While language which could be extended to more complex programming languages.

Most of the approaches are rigorously proven and assure that there is no leak of secure data in well-typed programs but the languages are either very restrictive and writing mean-

ingful programs of even moderate size is not easy like $\lambda_s^{\Box}$, or require a lot of programming effort like the SLam calculus.

3.3 Information Flow Work on Java Related Languages

The work on information flow presented in the previous subsections were not on Java or Java related languages. Enforcing information flow security in Java is hard due to its object-oriented model. However, a lot of attempts have been made to enforce information flow security in Java related languages.

Banerjee and Naumann [BN02] gave a noninterference proof for secure information flow in a sequential object-oriented language with pointers and mutable state, private fields and class-based visibility, dynamic binding and inheritance, casts and type tests, and mutually recursive classes and methods. They ensure pointers are safe by disallowing assignment of a high security object instance to a low security variable. They also present an access control mechanism [BN03] to enforce secure information flow for the same language which specifies that a system call to retrieve classified data can only be made with the right permission. However, they do not consider exceptions.

JFlow [Mye99] is an extension of Java and it supports the decentralized label model [ML98]. It allows security policy labels to be assigned to data values and the security policies can be changed to suit the program needs. JFlow requires static annotations in the Java code. The JFlow compiler certifies the programs and produces Java code which can be compiled by a Java compiler and then executed. Our work compares to the solution presented by Myers [Mye99]. However, JFlow focuses on Java source and it does not account for Java Bytecode that can be obtained from an outside source and then executed on the local machine by a user.

Avvenuti, Barnardeschi and Francesco [ABF03] presented an idea of performing information flow analysis on Java Bytecode. They only consider a very small subset of Java Bytecode with security levels assigned to classes, methods parameters and return values. It is a very restricted language because data from a class with a high security level cannot be assigned to fields of a class with low security level. Genaim and Spoto [GS05] gave information flow analysis for the full set of Java Bytecodes. They defined an abstract in-

terpretation model of information flow which formally defines explicit and implicit flows possible at the Bytecode level. They do not assign any security levels and their analysis only informs the user if a value can affect some other value. They give experimental results which only mention the size of the program and the time it took to analyze the program. We extend their concept in our experiments and design more meaningful metrics for measuring quality of information flow results.

3.4 Typed Low-Level Languages and Formalization of the Compilation Process

High-level languages are typed to enforce some desirable properties and they are checked by the compiler when the source is compiled. The resulting intermediate languages and low-level languages usually lose all the type information, properties which hold at the source level may no longer be valid. Researchers have tried to associate types with the low-level code in search of ways to ensure important properties hold after compilation.

Barthe, Rezk and Naumann's [BRN06] proposed a type preserving compilation technique for a subset of Java with exceptions. They show that the information flow security policy is observed by the high level source code as well as the Bytecode.

Chen and Tarditi [CT05] formulated a typed intermediate language for object-oriented languages which supports translation of Featherweight Java (FJ) [IPW01] with assignments and one dimensional arrays of objects added. They prove soundness for the intermediate language. League, Shao and Trifonov [LST02] also formalize a typed intermediate language which supports compilation of FJ and it also preserves types.

Callahan [O'C99] proposed a type system to formalize Java Bytecode and prove its safety since earlier attempts were only successful for subsets of Java Bytecode. Morrisett *et al.* [MCG+99] formulated a generic and realistic Typed Assembly Language (TAL) which can support several source languages. Our work on formalization of a subset of `JIMPLE` is similar to a subset of TAL which is called TAL-0 [Pie05]

Other work in this area includes a strongly typed intermediate language to compile richly typed source languages such as ML [SLM98] and a typed intermediate language to

represent Java classes [LST99].

Chapter 4

MINI-JIMPLE

This chapter presents the formal specification of a subset of JIMPLE IR with a soundness proof of type preservation for the type system. In the next section we highlight the importance of typed intermediate languages. Section 4.2 describes the grammar of the subset we picked to formalize. Section 4.3 and Section 4.4 give the operational semantics and type system respectively. In Section 4.5 the type preservation proof is explained along with some lemmas.

4.1 Typed Intermediate Languages

Invariably all compilers employ techniques to optimize the code written by a programmer. In order to maximize the runtime efficiency of the generated code the compilers use several mechanisms to improve the code. Analyses are performed and code is transformed at the source level, at several highly specific intermediate languages, at the Bytecode level or at the native code level in order to achieve speedups in execution time. Traditional analyses such as dead code elimination etcetera are common and can be performed on every representation of the code but other advanced analyses such as virtual method resolution in Java are easier to perform on specific representations such as the JIMPLE IR. In the case of Java, source code is compiled into Bytecode and various intermediate languages which are more suitable for program understanding and to perform optimizations and analyses

on. The SOOT framework performs several analyses on the intermediate representations including JIMPLE.

During compilation from source into Bytecode the type information is lost. A type inference algorithm [GHM00] analyzes JIMPLE and generates static types for the variables. The types are useful in debugging, guiding optimizations and specifying and proving safety proofs for programs. The following benefits are achieved as a result of having typed JIMPLE :

- we get a refined *call graph* built using the *class hierarchy analysis* [DGC95];
- it can be ascertained when an invokeinterface can be replaced by an invokevirtual call (meaning that we can tell that the receiver of a class even though the instruction is an invokeinterface);
- a decision on method inlining can be made;
- variable types are known and it simplifies the decompilation process;
- variables can be grouped by type which can be useful for run-time type analyses.

The correctness of Bytecode is verified by the Java Virtual Machine (JVM). The only correctness measure in SOOT for the optimizations and transformations is to generate Bytecode after transformations on the IRs and that should be verifiable by the JVM. This verified code executes safely but in order to be absolutely sure about the correctness of the code translations and each individual analysis and the portability of the model to different languages and platforms we need to be able to reason about the whole compilation strategy and procedure.

Prior to this work, no property of the SOOT framework was carefully proven to be correct. It would be ideal to have a formal specification of all the intermediate languages and all the translations in Soot. We formalized part of the JIMPLE IR and our work is explained in the following sections. Our type system is similar to the one for TAL-0 [Pie05] given by Morrisett.

4.2 Abstract Syntax

In this work we concentrate on the key elements of JIMPLE. The subset of JIMPLE (here after referred to as MINI-JIMPLE) we chose to formalize includes the if-statement, assignment statement and the goto statement. The subset grammar of JIMPLE is presented in a different format than in Section 2.3 which allows for the operational semantics and type system to be defined in an elegant manner. The abstract syntax for MINI-JIMPLE is given in Figure 4.1.

Program:

$$
\begin{array}{lcl}
p & ::= & sdec\ ;\ s\_seq\ ; \\
s\_dec & ::= & \tau\ x \mid s\_dec\ ;\ \tau\ y \\
s\_seq & ::= & () \mid s\ ;\ s\_seq \\
s & ::= & \texttt{goto label} \mid \texttt{if}\ (exp)\ \texttt{goto label} \mid x = exp \\
exp & ::= & exp_1 + exp_2 \mid exp_1 < exp_2 \mid x \mid n \mid \texttt{true} \mid \texttt{false}
\end{array}
$$

Declared Types:

$$\tau \ ::= \ int \mid bool$$

Figure 4.1: *MINI-JIMPLE Abstract Syntax*

A program in MINI-JIMPLE is a sequence of declarations followed by no statements or a sequence of statements. All declarations are at the beginning of the program before any other statement. A declaration defines a local variable, ranged over by x, y and z, along with its type τ which may be *int* or *bool* in MINI-JIMPLE. There are three statements: if-statement, assignment statement, and the goto statement. An if-statement branches on the value of an expression. An assignment statement stores the value of an expression in a local variable and the goto statement is an unconditional jump to a sequence of statements pointed to by the label.

Expressions occur in the if-statement and assignment statement. We keep the expres-

sions limited to binary additions, binary less than comparisons and values *v*. The values *v* in MINI-JIMPLE are local variables (represented by *x* in the syntax), integer constants (represented by *n* in the syntax), and the boolean constants `true` and `false`. More expressions may be added but they would add no special meaning to the language and its formal reasoning. The running example for MINI-JIMPLE is given in the Figure 4.2. We often abbreviate the variable declarations `int` *x*; `int` *y*; as `int` *x*,*y*;. Line (1) in the running example uses the abbreviation. We give type proof trees for the example's statements in Section 4.4.

```
          int x,y;                        (1)
label_1:  x = 2 + 2;                      (2)
          if (x < 4) goto label_2;        (3)
          y = 3 + 3;                      (4)
          goto label3;                    (5)
label_2:  y = 2 + 3;                      (6)
label_3:  <continue executing the code>   (7)
```

Figure 4.2: *Running Example of MINI-JIMPLE*

4.3 Operational Semantics

The operational semantics of MINI-JIMPLE specifies the behaviour of the language. We define an *abstract machine* and its description is given in this section.

4.3.1 The Abstract Machine

The abstract machine (M) has three components which describe a state in the machine:

1. a finite heap (H) which maps the labels to statement sequences;
2. a sequence of instructions (s); and

3. a store (represented by metavariable μ) which is a partial function which maps local variables to values.

The syntax of the abstract machine is given in Figure 4.3. The evaluation of `MINI-JIMPLE` is defined by a transition function from one state to another:

$$(H, s \mid \mu) \rightarrow (H, s' \mid \mu')$$

The transition function is defined for each statement except for () which is the final value returned at the end of evaluating a program. The machine has a small step operational semantics. The heap (H) is initialized with the labels pointing to the corresponding statement sequences before the program is executed and it does not change during execution. Explicitly carrying H during evaluation is not necessary since it doesn't change. We will however be explicit about it for clarity. The variable declarations are at the beginning of the program and so μ is initialized by the first few statements of the program. The evaluation rule for variable declarations is presented in Figure 4.4 and the six evaluation rules for statements are given in Figure 4.5.

Abstract Machine:

$$M \quad ::= \quad (H,\ s\_seq,\ \mu)$$

Heap:

$$H \quad ::= \quad \{label_1 \ = \ s\_seq_1, \ldots, label_m \ = \ s\_seq_m\}$$

Store:

$$\mu \quad ::= \quad . \mid \mu, (x, v) \mid \mu, x$$

Figure 4.3: *Abstract Machine Syntax*

$$\frac{\forall_i\, y_i \notin \mu \quad \mu' = \mu, y_i}{(H, (\tau_1 y_1, \ldots, \tau_n y_n);\ s\_seq \mid \mu) \rightarrow (H, s\_seq \mid \mu')} \text{ (E-Dec)}$$

Figure 4.4: *Operational Semantics for Declarations*

In the case of a goto statement evaluation rule (E-Goto) applies. The evaluation continues with the sequence of statements the label points to is given by the heap H.

We have three rules for the evaluation of an if-statement. If the conditional expression is not a value it is evaluated by the machine according to the rule (E-IfCond). When the conditional value is known, the evaluation jumps to the sequence of statements s_2 pointed to by the label according to rule (E-IfTrue). On the other hand if the conditional value is false the evaluation continues with the statement s_1 after the if-statement according to the rule (E-IfFalse).

$$\frac{H(\texttt{label}) = s\_seq_2}{(H, (\texttt{goto label}); s\_seq_1 \mid \mu) \rightarrow (H, s\_seq_2 \mid \mu)} \text{ (E-Goto)}$$

$$\frac{exp \rightarrow v}{(H, (\texttt{if}\ (exp)\ \texttt{goto label}; s\_seq) \mid \mu) \rightarrow (H, (\texttt{if}\ v\ \texttt{goto label}; s\_seq) \mid \mu)} \text{ (E-IfCond)}$$

$$\frac{H(\texttt{label}) = s\_seq_2}{(H, (\texttt{if true goto label}; s\_seq_1) \mid \mu) \rightarrow (H, s\_seq_2 \mid \mu)} \text{ (E-IfTrue)}$$

$$\frac{}{(H, (\texttt{if false goto label}; s\_seq_1) \mid \mu) \rightarrow (H, s\_seq_1 \mid \mu)} \text{ (E-IfFalse)}$$

$$\frac{exp \rightarrow v}{(H, (x\ =\ exp; s\_seq) \mid \mu) \rightarrow (H, (x\ =\ v; s\_seq) \mid \mu)} \text{ (E-AssignExp)}$$

$$\frac{}{(H, (x\ =\ v; s\_seq) \mid \mu) \rightarrow (H, s\_seq \mid [x \mapsto v]\mu)} \text{ (E-AssignV)}$$

Figure 4.5: *Operational Semantics for MINI-JIMPLE*

The assignment statements has two evaluation rules. If the expression is not a value it is evaluated by the machine according to the rule (E-AssignExp). When the expression

is a value the local variable x is updated in the store by the rule `(E-AssignV)`.

The evaluation for addition ($+$) and less than ($<$) expressions ($exp \rightarrow v$) is assumed to be provided by the machine and is correct. Store lookup is incorporated in the expressions since local variables are dereferenced in expressions. We do not include expressions in our soundness proof.

The store behaviour to update the value stored in a local variable is presented in Figure 4.6 by four rules. Rule (4.1) and (4.2) state that if a previously assigned local variable x is being assigned a value v it does not change the contents of variable y. According to rule (4.3) if variable x is assigned a value v, it replaces any previous value v' stored in x. The last rule (4.4) defines that a value may be assigned to a previously defined local variable which had not been assigned a value before.

$$[x \mapsto v](\mu,(y,v')) = [x \mapsto v]\mu,(y,v') \tag{4.1}$$

$$[x \mapsto v](\mu,y) = ([x \mapsto v]\mu),y \tag{4.2}$$

$$[x \mapsto v](\mu,(x,v')) = (\mu,(x,v)) \tag{4.3}$$

$$[x \mapsto v](\mu,x) = (\mu,(x,v)) \tag{4.4}$$

Figure 4.6: *Store Behaviour in the Abstract Machine*

4.4 Type System

In this section we present a type system for `MINI-JIMPLE` along with the environments, judgments, types and typing rules. The type system is a type checker and does not infer types for expressions.

4.4.1 Environments

We have variables in `MINI-JIMPLE` which need a typing environment and labels which require a general environment since the typing judgments will verify if the labels are valid.

In order to fulfil these requirements we define the following two environments:

Local Variable Environment (Γ)

Γ is a typing environment for local variables in the store μ and is defined as follows:

$$\Gamma ::= . \mid \Gamma, x : \tau$$

Labels Environment (Ψ)

Ψ is an environment for labels. It is defined as follows:

$$\Psi ::= . \mid \Psi, \texttt{label\_i}$$

4.4.2 Typing Judgments

The typing judgments in MINI-JIMPLE are defined for the store, the heap, the program, the statements in the program, the expressions and the values. An *ok* is assigned for the environments, program and statements if they are well-typed.

The store has to be well-typed and the type of the variables in the store and the value mapped to the variable should match.

$$\Gamma \vdash \mu : \text{ok}$$

The heap (H) has to be well formed by the environment Ψ for valid labels. In order to type check H in Ψ we erase all labels in the program and type check the erased program.

$$\Psi \vdash H : \text{ok}$$

The program p is well-typed under Ψ if it returns an ok.

$$\Psi \vdash p : \text{ok}$$

The statement sequences are well-typed if the type checker returns an ok under the environments Ψ and Γ.

$$\Psi;\Gamma \vdash s\_seq : \text{ok}$$

The expressions are well-typed if they have a type τ under Ψ and Γ.

$$\Psi;\Gamma \vdash exp : \tau$$

4.4.3 Types in `MINI-JIMPLE`

There are two program types INT and BOOL corresponding to the declared types *int* and *bool* in `MINI-JIMPLE`. τ is defined as follow:

$$\tau ::= \text{INT} \mid \text{BOOL}$$

4.4.4 Typing Rules

The typing rules which assign types to terms for `MINI-JIMPLE` are presented in Figure 4.7. The typing rule for declarations `(T-Program)` states that all local variables are added to the Γ environment and the program statements are type checked under the Ψ and Γ environments.

There are two typing rules for sequence of statements: the first one `(T-NoStmt)` assigns an *ok* for no statement where as `(T-Stmts)` assigns an *ok* based on the fact that both the first statement and the remaining sequence of statements type check to *ok*.

The typing rule `(T-GotoStmt)` assigns an *ok* to a goto statement if the `label` is in environment Ψ. `(T-IfStmt)` rule assigns an *ok* to the if-statement after checking that the expression type checks to BOOL and the `label` is in environment Ψ.

Typing rule (T-AssignStmt) for the assignment statement assigns an *ok* based on the types of the variable x and the expression in the premise. x and the expression must evaluate to the same type.

According to the type rules for the conditional expression (rule (T-Condexp)) and the addition expression (rule (T-AddExp)) they are assigned type BOOL and type INT respectively based on types of the operands. In both cases the operands exp_1 and exp_2 must have type INT.

The typing rule (T-Var) assigns a type τ to variable x if the variable along with its type τ is present in the local variable typing environment Γ. The constant n (representing integer values) is given type INT by the rule (T-Int) where as the constants `true` and `false` are assigned type BOOL by rules (T-True) and (T-False) respectively.

While we present typing rules for expressions, we will assume that evaluation of expressions preserves types. This lemma will be heavily used in the overall type preservation proof given later. There is no typing rule for labels since label lookup is incorporated in the rules (T-GotoStmt) and (T-IfStmt) for typing the goto statement and if-statement respectively.

The environment Γ describes the store and gives the store typing for each location according to the rules in Figure 4.8. An empty is assigned *ok* by the rule (T-StoreEmpty). Rule (T-StoreVar) states that (μ, y) is well-typed under Γ if μ is well-typed under Γ and the declared but unassigned variable y has a type τ in Γ. In the case when a store location has been assigned a value (for example $(\mu, (y, v))$) rule (T-StoreVarVal) assigns the store an *ok* if μ is well-typed under Γ, variable y has type τ and the value v assigned to y also type checks to τ under Γ.

We give the type proof trees for our running example in Figure 4.9 presented earlier in Figure 4.2 to illustrate how our type system would check MINI-JIMPLE programs. All labels are added to Ψ at the beginning of execution before a statement is actually executed and the environment is available while type checking the rest of the program. Proof trees of statements (2),(3) and (5) are given. Proof trees for statements (4) and (6) will be the same as the one for statement (2).

$$\frac{\Psi; y_1 : \tau_1, \ldots, y_n \vdash s : ok}{\Psi \vdash \tau_1 y_1; \ldots; \tau_n y_n : \tau_n; s : ok} \text{ (T-Program)}$$

$$\frac{}{\Psi; \Gamma \vdash () : ok} \text{ (T-NoStmt)}$$

$$\frac{\Psi; \Gamma \vdash s : ok \quad \Psi; \Gamma \vdash s\_seq : ok}{\Psi; \Gamma \vdash s; s\_seq : ok} \text{ (T-Stmts)}$$

$$\frac{\texttt{label} \in \Psi}{\Psi; \Gamma \vdash \texttt{goto label} : ok} \text{ (T-GotoStmt)}$$

$$\frac{\Psi; \Gamma \vdash exp : \text{BOOL} \quad \texttt{label} \in \Psi}{\Psi; \Gamma \vdash \texttt{if}\ (exp)\ \texttt{goto label} : ok} \text{ (T-IfStmt)}$$

$$\frac{\Psi; \Gamma \vdash x : \tau \quad \Psi; \Gamma \vdash exp : \tau}{\Psi; \Gamma \vdash x\ =\ exp : ok} \text{ (T-AssignStmt)}$$

$$\frac{\Psi; \Gamma \vdash exp_1 : \text{INT} \quad \Psi; \Gamma \vdash exp_2 : \text{INT}}{\Psi; \Gamma \vdash exp_1\ <\ exp_2 : \text{BOOL}} \text{ (T-Condexp)}$$

$$\frac{\Psi; \Gamma \vdash exp_1 : \text{INT} \quad \Psi; \Gamma \vdash exp_2 : \text{INT}}{\Psi; \Gamma \vdash exp_1\ +\ exp_2 : \text{INT}} \text{ (T-AddExp)}$$

$$\frac{x : \tau \in \Gamma}{\Psi; \Gamma \vdash x : \tau} \text{ (T-Var)}$$

$$\frac{}{\Psi; \Gamma \vdash n : \text{INT}} \text{ (T-Int)}$$

$$\frac{}{\Psi; \Gamma \vdash \texttt{true} : \text{BOOL}} \text{ (T-True)}$$

$$\frac{}{\Psi; \Gamma \vdash \texttt{false} : \text{BOOL}} \text{ (T-False)}$$

Figure 4.7: *Typing Rules for* `MINI-JIMPLE`

$$\frac{}{\Gamma \vdash . : ok}\ \texttt{(T-StoreEmpty)}$$

$$\frac{\Gamma \vdash \mu \quad \Gamma(y) = \tau}{\Gamma \vdash (\mu, y) : ok}\ \texttt{(T-StoreVar)}$$

$$\frac{\Gamma \vdash \mu \quad \Gamma(y) = \tau \quad \Gamma \vdash v : \tau}{\Gamma \vdash (\mu, (y, v)) : ok}\ \texttt{(T-StoreVarVal)}$$

Figure 4.8: *Typing Rules for* `MINI-JIMPLE` *Local Variable Store*

$\Psi = \{\texttt{label\_1}, \texttt{label\_2}, \texttt{label\_3}\}$

Proof tree for statement (2) :

$$\frac{\dfrac{x : \text{INT} \in \ldots}{\Psi; \ldots \vdash x : \text{INT}}\ \texttt{(T-Var)} \quad \dfrac{\dfrac{}{\Psi; \ldots \vdash 2 : \text{INT}}\ \texttt{(T-Int)} \quad \dfrac{}{\Psi; \ldots \vdash 2 : \text{INT}}\ \texttt{(T-Int)}}{\Psi; \ldots \vdash 2 + 2 : \text{INT}}\ \texttt{(T-AssignStmt)}}{\dfrac{\Psi; x : \text{INT}; y : \text{INT}; \vdash x = 2 + 2 : ok}{\Psi \vdash \texttt{int}\ x, y;\ x = 2 + 2 : ok}\ \texttt{(T-Program)}}$$

Proof tree for statement (3) :

$$\frac{\dfrac{\texttt{label\_3} \in \Psi}{\Psi; x : \text{INT}; y : \text{INT}; \vdash \texttt{goto label\_3} : ok}\ \texttt{(T-GotoStmt)}}{\Psi \vdash \texttt{int}\ x, y;\ \texttt{goto label\_3} : ok}\ \texttt{(T-Program)}$$

Proof tree for statement (5) :

$$\frac{\dfrac{\dfrac{\vdots}{\Psi; \ldots \vdash x : \text{INT}} \quad \dfrac{\vdots}{\Psi; \ldots \vdash 4 : \text{INT}}}{\Psi; \ldots \vdash (x < 4) : \text{BOOL}}\ \texttt{(T-Condexp)} \quad \dfrac{\vdots}{\Psi; \ldots \vdash \texttt{goto label\_2} : ok}}{\dfrac{\Psi; x : \text{INT}; y : \text{INT}; \vdash \texttt{if}(x < 4)\ \texttt{goto label\_2} : ok}{\Psi \vdash \texttt{int}\ x, y;\ \texttt{if}(x < 4)\ \texttt{goto label\_2} : ok}\ \texttt{(T-Program)}}\ \texttt{(T-IfStmt)}$$

Figure 4.9: *Type Proof Trees for the Runnning Example*

4.5 Type Preservation

The type preservation proof gives an assurance that the program will be well-typed after each step of the abstract machine. It is important to prove the soundness of the type system in order to be sure that well-typed statements will not get stuck during evaluation until all statements have been evaluated and we reach the no statement marker. We present the proof and the required lemmas in this section.

Lemma 4.5.1 *Updating a store preserves store typing.*

This lemma states that if a store μ is well-typed under Γ, local variable x has type τ and value v also having type τ is assigned to x, the store still remains well-typed.

If

$\mathscr{D}$:$\Gamma \vdash \mu$:ok

$\mathscr{E}$:$\Gamma(x) = \tau$

$\mathscr{F}$:$\Psi;\Gamma \vdash$ v:τ

then

$\mathscr{G}$:$\Gamma \vdash [x \mapsto v]\ \mu$:$ok$.

Proof: By induction on the derivation $\mathscr{D}$. In the proof i.h. stands for induction hypothesis.

Case: $x \neq y$

$\mathscr{D}$ =

$$\frac{\overset{\mathscr{D}_1}{\Gamma \vdash \mu} \quad \Gamma(y) = \tau \quad \Gamma \vdash v' : \tau}{\Gamma \vdash (\mu, (y, v')) : ok}\ \texttt{(T-StoreVarVal)}$$

by i.h. on $\mathscr{D}_1$	$\Gamma \vdash [x \mapsto v]\mu : ok$
by rule `(T-StoreVarVal)`	$\Gamma \vdash ([x \mapsto v]\mu, (y, v')) : ok$
by definition of update	$\Gamma \vdash [x \mapsto v](\mu, (y, v')) : ok$

$\mathscr{D}$ =

$$\frac{\overset{\mathscr{D}_1}{\Gamma \vdash \mu} \quad \Gamma(y) = \tau}{\Gamma \vdash (\mu, y) : ok}\ \texttt{(T-StoreVar)}$$

by i.h. on $\mathscr{D}_1$	$\Gamma \vdash [x \mapsto v]\mu : ok$
by rule (T-StoreVar)	$\Gamma \vdash ([x \mapsto v]\mu), y : ok$
by definition of update	$\Gamma \vdash [x \mapsto v](\mu, y) : ok$

Case: $x = y$

$\mathscr{D}$ =

$$\frac{\overset{\mathscr{D}_1}{\Gamma \vdash \mu} \quad \Gamma(x) = \tau \quad \Gamma \vdash v' : \tau}{\Gamma \vdash (\mu, (x, v')) : ok} \text{ (T-StoreVarVal)}$$

by assumption	$\Psi;\Gamma \vdash v : \tau$
by rule (T-StoreVarVal)	$\Gamma \vdash (\mu, (x, v)) : ok$
by definition of update	$\Gamma \vdash [x \mapsto v](\mu, (x, v')) : ok$

$\mathscr{D}$ =

$$\frac{\overset{\mathscr{D}_1}{\Gamma \vdash \mu} \quad \Gamma(x) = \tau}{\Gamma \vdash (\mu, x) : ok} \text{ (T-StoreVar)}$$

by assumption	$\Psi;\Gamma \vdash v : \tau$
by rule (T-StoreVarVal)	$\Gamma \vdash (\mu, (x, v)) : ok$
by definition of update	$\Gamma \vdash [x \mapsto v](\mu, x) : ok$

Lemma 4.5.2 *Statements are well-typed if the store typing environment (*Γ*) is extended.*

This lemma states that a statement which is well-typed under the local variable typing environment Γ and is assigned *ok* will also be well-typed in Γ' which types all variables in Γ and possibly more. *s* will be assigned *ok* in Γ' as well.

If

$\mathscr{D} : \Psi;\Gamma \vdash s{:}ok$

$\mathscr{E} : \Gamma' \supseteq \Gamma$

then

$\mathscr{F} : \Psi;\Gamma' \vdash \text{s:}ok.$

Proof: By structural induction on the derivation $\mathscr{D}$. Only two cases are given here.

Case: $\mathscr{D}$ =

$$\frac{\overset{\mathscr{D}_1}{\Psi;\Gamma \vdash exp : \text{BOOL}} \quad \texttt{label} \in \Psi}{\Psi;\Gamma \vdash \texttt{if}\ (exp)\ \texttt{goto}\ \texttt{label} : ok}\ \texttt{(T-IfStmt)}$$

by i.h on $\mathscr{D}_1$	$\Psi;\Gamma' \vdash exp : \text{BOOL}$
by rule `(T-IfStmt)`	$\Psi;\Gamma' \vdash \texttt{if}\ (exp)\ \texttt{goto}\ \texttt{label} : ok$

Case: $\mathscr{D}$ =

$$\frac{\overset{\mathscr{D}_1}{\Psi;\Gamma \vdash x : \tau} \quad \overset{\mathscr{D}_2}{\Psi;\Gamma \vdash exp : \tau}}{\Psi;\Gamma \vdash x\ =\ exp : ok}\ \texttt{(T-AssignStmt)}$$

by i.h on $\mathscr{D}_1$	$\Psi;\Gamma' \vdash x : \tau$
by rule `(T-AssignStmt)`	$\Psi;\Gamma' \vdash x\ =\ exp : ok$

Theorem 4.5.3 *A well-typed sequence of statements (if it is not the end of statements marker) will take a step according to the evaluation rules and the resulting sequence of statements will also be well-typed.*

The theorem states that if a sequence of statements is well-typed, the store is well-typed, and the machine can take a step as specified by one of the evaluation rules, then the resulting sequence of statements will be well-typed and the resulting store will be well-typed. For this proof we consider the heap as well formed under Ψ.

If

$\Gamma \vdash \mu\text{:}ok$

$\Psi \vdash H\text{:}ok$

$\mathscr{D}\text{:}\Psi;\Gamma \vdash s\_seq\text{:}ok$

$\mathscr{E}\text{:}(H,s) \mid \mu \rightarrow (H,s') \mid \mu'$

then

$\mathscr{F}$:$\Psi;\Gamma \vdash s\_seq'$:$ok$

$\Gamma \vdash \mu'$:ok.

Proof: By following the evaluation rules for all possible sequences of statement.

Case: $\mathscr{D}$ =

$$\frac{\overset{\mathscr{D}_1}{\Psi;\Gamma \vdash \texttt{goto label} : ok} \quad \overset{\mathscr{D}_2}{\Psi;\Gamma \vdash s\_seq : ok}}{\Psi;\Gamma \vdash \texttt{goto label};s\_seq : ok}\ \texttt{(T-Stmts)}$$

$\mathscr{E}$ =

$$\frac{\overset{\mathscr{E}_1}{H(\texttt{label}) = s\_seq_2}}{(H,(\texttt{goto label});s\_seq \mid \mu) \rightarrow (H, s\_seq_2 \mid \mu)}\ \texttt{(E-Goto)}$$

by inversion on $\mathscr{D}_1$	$\texttt{label} \in \Psi$
by $\mathscr{E}_1$	$H(\texttt{label}) = s\_seq_2$
by $\Psi \vdash H : ok$, *heap is well formed*	$\Psi;\Gamma \vdash s\_seq_2 : ok$
by assumption	$\Gamma \vdash \mu : ok$

Case: $\mathscr{D}$ =

$$\frac{\overset{\mathscr{D}_1}{\Psi;\Gamma \vdash \texttt{if}\ (exp)\ \texttt{goto label} : ok} \quad \overset{\mathscr{D}_2}{\Psi;\Gamma \vdash s\_seq : ok}}{\Psi;\Gamma \vdash \texttt{if}\ (exp)\ \texttt{goto label};s\_seq : ok}\ \texttt{(T-Stmts)}$$

Subcase 1: $\mathscr{E}$ =

$$\frac{\overset{\mathscr{E}_1}{exp \rightarrow v}}{(H,(\texttt{if}\ (exp)\ \texttt{goto label};s\_seq) \mid \mu) \rightarrow (H,(\texttt{if}\ v\ \texttt{goto label};s\_seq) \mid \mu)}\ \texttt{(E-IfCond)}$$

by inversion on $\mathscr{D}_1$	$\Psi;\Gamma \vdash exp : \text{BOOL}$ *and* $\texttt{label} \in \Psi$
by type preservation lemma for expressions :	
if $\Psi;\Gamma \vdash exp : \tau$ *and* $exp \rightarrow v$, *then* $\Psi;\Gamma \vdash v : \tau$	
therefore	$\Psi;\Gamma \vdash v : \text{BOOL}$
by rule (T-IfStmt)	$\Psi;\Gamma \vdash \texttt{if}\ v\ \texttt{goto label} : ok$
by rule (T-Stmts)	$\Psi;\Gamma \vdash (\texttt{if}\ v\ \texttt{goto label};s\_seq) : ok$
by assumption	$\Gamma \vdash \mu : ok$

Subcase 2: $\mathscr{E}$ =

$$\frac{\overset{\mathscr{E}_1}{H(\texttt{label}) = s\_seq_2}}{(H, (\texttt{if true goto label}; s\_seq) \mid \mu) \rightarrow (H, s\_seq_2 \mid \mu)} \texttt{(E-IfTrue)}$$

by inversion on $\mathscr{D}_1$	$\Psi; \Gamma \vdash \texttt{true} : \texttt{BOOL}$ *and* $\texttt{label} \in \Psi$
by $\mathscr{E}_1$	$H(\texttt{label}) = s\_seq_2$
by $\Psi \vdash H : ok$, *heap is well formed*	$\Psi; \Gamma \vdash s\_seq_2 : ok$
by assumption	$\Gamma \vdash \mu : ok$

Subcase 3: $\mathscr{E}$ =

$$\frac{}{(H, (\texttt{if false goto label}; s\_seq) \mid \mu) \rightarrow (H, s\_seq \mid \mu)} \texttt{(E-IfFalse)}$$

by $\mathscr{D}_2$	$\Psi; \Gamma \vdash s\_seq : ok$
by assumption	$\Gamma \vdash \mu : ok$

Case: $\mathscr{D}$ =

$$\frac{\overset{\mathscr{D}_1}{\Gamma \vdash x = exp : ok} \quad \overset{\mathscr{D}_2}{\Psi; \Gamma \vdash s\_seq : ok}}{\Psi; \Gamma \vdash \Gamma \vdash x = exp; s\_seq : ok} \texttt{(T-Stmts)}$$

Subcase 1: $\mathscr{E}$ =

$$\frac{\overset{\mathscr{E}_1}{exp \rightarrow v}}{(H, (x = exp; s\_seq) \mid \mu) \rightarrow (H, (x = v; s\_seq) \mid \mu)} \texttt{(E-AssignExp)}$$

by inversion on $\mathscr{D}_1$	$\Psi; \Gamma \vdash x : \tau$ *and* $\Psi; \Gamma \vdash exp : \tau$
by type preservation lemma for expressions : *if* $\Psi; \Gamma \vdash exp : \tau$ *and* $exp \rightarrow v$, *then* $\Psi; \Gamma \vdash v : \tau$	
by rule `(T-AssignStmt)`	$\Psi; \Gamma \vdash x = v : ok$
by rule `(T-Stmts)`	$\Psi; \Gamma \vdash (x = v; s\_seq) : ok$
by assumption	$\Gamma \vdash \mu : ok$

Subcase 2: $\mathscr{E}$ =

$$\frac{}{(H, (x = v; s\_seq) \mid \mu) \rightarrow (H, s\_seq \mid [x \mapsto v]\mu)} \texttt{(E-AssignV)}$$

by $\mathscr{D}_2$	$\Psi; \Gamma \vdash s\_seq : ok$
by lemma 4.5.1	*store update is valid*

4.6 Summary

We explained the type system for MINI-JIMPLE in this chpater. The type system has two types, INT and BOOL, which do not carry any security information for the data in MINI-JIMPLE programs. In order to represent the security level of data we need a richer type system with security types. The security types consist of the type of the data and a security label indicating the security level at which the data may be used. Secure information flow can only be proven for programs with security types and such a type system can be built by extending the type system given in this chapter.

Chapter 5
Data-Flow Analysis for Secure Information Flow

This chapter describes a *context-sensitive inter-procedural* data-flow analysis which tracks information flow. The context-sensitive analysis was chosen even though it is very expensive because it is the most accurate of the inter-procedural analyses. This analysis demonstrates the best results we can possibly get from a data-flow analysis for information flow. It was implemented in Soot by extending the flow-analysis framework which facilitates the implementation of code analyses and optimizations.

A data-flow analysis analyzes each statement in a program and ascertains how data is being used by the statement [Muc97]. The rules for analyzing each statement depends on the kind of analysis and they vary from analysis to analysis. Data-flow analyses provide meaningful information regarding the code to the programmers and the information is also helpful in guiding optimizations. The details of the data-flow anlaysis work in this research are presented in this chapter and the next chapter (Chapter 6).

5.1 The Analysis

The information flow analysis has been implemented at the level of the `JIMPLE` IR (grammar given in Section 2.3). `JIMPLE` has fewer statements than actual Java and we specify rules to analyze each one. Our analysis successfully analyzes all statements in `JIMPLE`

including the ones that are in the clinit methods (which initialize the static fields). The flow analysis framework in Soot simplified the implementation of the analysis a great deal. The information flow analysis makes use of the *control-flow* information and the *points-to* information which is available in Soot. We designed and implemented two analyses: the first one is class-based explained in this chapter and the second one considers instances of classes (explained in Chapter 6).

The context-sensitive inter-procedural analysis starts at the main method of the application that is being analyzed. It analyzes the statements one after the other and being a context-sensitive analysis it considers the affect a method call would have on the data that is collected in the analysis. At a method-invocation the analysis follows the call and completely analyzes the invoked method statements before continuing to analyze the callee's remaining statements. In order to statically approximate the target methods of a call site the call graph has to be constructed by analyzing the receiver variable at each call site. Figure 5.1 gives an example of a call site in which variable *o* is the receiver whose type suggests the possible methods that could be invoked.

```
o.foo(arg_1,...,arg_n) //Example call site
```

Figure 5.1: *Call Site*

Soot computes a call graph which the analysis uses in determining the invoked methods statically. A precise call graph reduces the number of possible target methods and reduces the cost of the analysis. Therefore we use the most precise call graph available in Soot which is constructed by the class hierarchy analysis [DGC95] and is refined by the points-to information computed by Spark [LH03, Lho02]. The points-to analysis returns a set of objects to which a variable may point to which is useful in accurately determining the possibly types of the receiver variable.

The analysis makes use of the control-flow graph information available in Soot in order to know the continuous blocks of statements without any jumps or jump targets. The

mechanism for merging information at the beginning or end of a basic block of statements together with the rules for analyzing each statement defines a data-flow analysis.

5.1.1 General Rules for Jimple Constructs

The analysis is a data-flow analysis and its specification is presented by the following six rules.

1. Collects sets of secure (high) data. The high data may be local variables, static or instance fields, array variables or class names of exceptions in Soot.

2. A variable (local variables or fields) is considered high at a program point p if it is assigned a value which was computed using at least one secure variable or it was assigned any value in a high context on any path before p. An exception is high if the variable in the throw statement is high or the throw statement happens to be in a high context. In the case of local variable another condition is that it is not assigned a low value in between the assignment statement and point p and in the case of an exception is that it is not caught before the statement.

3. It is a forward analysis

4. The confluence operator is union which means that at a control-flow join the variables in the two joining paths are merged together by the union operator. This means that variables belonging to either path become part of the set after the join.

5. The following affect the high information set:

   - assignment statements with method invocation expressions, field references, array references and local variables;
   - throw statements for exceptions;
   - identity statements which assign parameter values to local variables at the beginning of methods and those which catch exceptions;
   - control flow statements: if and switch; and

- method invocation statements.

The data-flow equation for JIMPLE statements is as follows:

out(s) = gen(s) $\cup$ **(in(s) - kill(s))**

- out(s) is the set of highs after the statement
- gen(s) contains the data to be made high in the current statement
- in(s) is the set of highs before the statement
- kill(s) contains the data which is no longer high after the statement

We explain the rules for computing the gen and kill sets for JIMPLE statements and details of the algorithms which have been implemented to enforce the rules in Section 5.2.

6. Starting approximations are as follows:

- **out(start) =** $\{\}$. This is a safe approximation because at start no secure data has entered the application program.
- **out(all other statements) =** $\{\}$. This is an unsafe approximation because it would have been safe to say that all variables, fields, arrays and exceptions are at a high security level.

We are computing a least fixed point in this analysis.

5.1.2 Secure Information in the Program

Programming languages have different mechanisms for specifying secure data in programs. In this work we consider two security levels: a high H security level and a low L security level as given in Section 2.2.1. We have to make a choice to specify the source of secure data for our analysis. This will affect the output (that is the information flows and warnings) but it does not affect the basic running of the analysis.

We consider the main argument array as high. The return value of library calls is also considered high. The library call can be thought of as a possible high user input or a database access which returns a high data value.

5.2 Analysis Rules for Each Statement

In this section we present how the constructs are analyzed. The analysis takes into consideration the uses and definitions in each statement. Depending on the type of statement, variables, fields or classes in the case of exceptions are added to the set of highs, and local variables and classes of exceptions are removed from the set of highs. The fields of all application (user-defined) classes are considered globals. Even though the instance fields which belong to different instances of the same class are independent, this analysis makes no differentiation between them. A field is analyzed based on its name and class in which it is defined. Therefore static and instance fields are handled in the same manner.

The *inSet* is the set of high data before a statement. When a statement is analyzed according to the rule specified for the kind of statement the data that is to be added to the set of highs are added to the gen set and the data to be removed from the set of highs are added to the kill set. Once the statement has been analyzed the general flow equation:

out(s) = gen(s) $\cup$ **(in(s) - kill(s))**

is applied to compute the *outSet* which is the set of highs after analyzing the statement. All the sets in the implementations are flow sets which store Java objects just as Lists. Flow sets provide useful operations such as add, remove, union of two sets and intersections of two sets which are used in the analysis.

5.2.1 Analyzing an Assignment Statement

The analysis of the assignment statement can be broken down into three parts depending on what occurs on the left hand side and right hand side of the assignment statement. The three parts are dependent on the occurrence or lack of of invocation expressions on the right-hand side of the assignment statement and array lookups on either side of the assignment statement. The general rule applies to majority of the assignment statements.

Simple Assignment Statement

The simple assignment statement has neither an invoke expression nor an array expression. Some examples are given in Figure 5.2.

The gen set of an assignment statement consists of the variable (local or a instance or static field) which is assigned the value which is computed using a high variable. The kill set of an assignment statement is the variable (local or static field) which is assigned a value computed from variables which are not high before the statement.

The general rule to analyze the assignment statement is as follows:

$a = b$

In this case, if b is a secure variable, then a will be added to the gen set or if b is an insecure variable or constant value, then a will be added to the kill set.

$a = b + c$

In this case, if b or c or both are secure variables, then a will be in the gen set. If both b and c are insecure variables, then a will be in the kill set.

```
i2 = <Test: int j>;
i0 = 0;
i1 = lengthof r1;
```

Figure 5.2: *Simple Assignment Statement*

In the analysis, at each such assignment statement the variables used (they can be local variables or fields) are checked if they are present in the set of highs before the statement. Depending on the security level of the used variables, it is decided whether to add the defined variable (local variable or field) to the set of the high variables or not. Algorithm 1 presents the pseudocode of the algorithm employed in the implementation of the analysis. A flag is set if any of the uses in the statement is high and depending on that the variable being assigned is added to the gen or kill set. Only local variables or static fields can be

added to the kill set since we are not sure if there is only one instance of a class.

Algorithm 1: Analyzing an Assignment Statement Not in a High Context

input : AssignStatement, Gen Set, Kill Set
output: Changed Gen and Kill Sets

```
1  Uses ← AssignStatement.GetUses ;
2  if any use is high then
3      set a flag;
4  end
5  Defs ← AssignStatement.GetDefs ;
6  if flag is set then
7      GenSet.Add (Defs);
8  else
9      KillSet.Add (Defs);
10 end
```

Assignment Statement with Invocation on the Right Hand Side

The details of how an invocation is handled are presented in Section 5.2.6. An example of such a statement is given in Figure 5.3. The only difference between a simple assignment statement and this statement is that the right hand side is high if the return value of a non-void method is high or the receiver variable (variable $r5 in Figure 5.3) is high.

```
virtualinvoke $r5.<my: int setKK(int[],int)>(r3, $i10);
```

Figure 5.3: *Simple Assignment Statement with Invocation on RHS*

Assignment Statement with an Array Expression

According to the JIMPLE grammar, the array expression can be on the left hand side of an assignment statement in which an array location is being assigned a value of a local or a constant or it can be on the right-hand side of an assignment statement in which the value of an array location is assigned to local variable. In the case of arrays in JIMPLE, we use the *base variable* information of the array. The security level is enforced on the whole array because the index values in array reads and writes can not be ascertained statically if they are not constants. The use of any high value to initialize the array or assignment of a high value to any of the array's locations makes the array high. The array can never return to a low security value since assignment of a low value to one of the array locations can not be generalized to make the whole array low. In the code snippet in Figure 5.4, when a new array is initialized, the local variable r1 that first references it becomes the base variable. Further on when r2 also references the same array (after statement r2 = r1), any use of r2 will have the base variable as r2. r2 will be added to the set of high variables if r1 was high by the analysis of the simple assignment statement.

```
r1 = newarray (int)[8];
r2 = r1;
i3 = r2[0];
r2[5] = i5;
```

Figure 5.4: *Array Declaration and Use*

If an array expression occurs on the right-hand side (i3 = r2[0]; in Figure 5.4), the security level of the array index variable or the *base variable* is ascertained. If either of them is high, then a flag is set. Algorithm 2 presents the pseudocode of the analysis. On the other hand if an array expression occurs on the left-hand side (r2[5] = i5; in Figure 5.4), the security level of the uses are checked. If the uses are high or the array index variable of the array is high, then the *base variable* is added to the gen set. The pseudocode for the

analysis is presented in Algorithm 3.

Algorithm 2: Assignment Statement with Array Expression on RHS

input : AssignStatement, Gen Set, Kill Set

output: Changed Gen and Kill Sets

```
1  Uses ← AssignStatement.GetArrayExp ;
2  if array index is high or array base variable is high then
3      set a flag;
4  end
5  Defs ← AssignStatement.GetDefs ;
6  if flag is set then
7      GenSet.Add (Defs);
8  else
9      KillSet.Add (Defs);
10 end
```

Algorithm 3: Assignment Statement with Array Expression on LHS

input : AssignStatement, Gen Set, Kill Set

output: Changed Gen and Kill Sets

```
1  Uses ← AssignStatement.GetUses ;
2  if any use is high then
3      set a flag;
4  end
5  Defs ← AssignStatement.GetArrayExp ;
6  if flag is set or array index is high then
7      GenSet.Add (ArrayBaseVar);
8  else
9      DoNothing ;
10 end
```

5.2.2 Conditional Statements

Statements which have branch conditions and affect the control-flow are potential places where an implicit flow might occur. In order to successfully identify an information leak that might occur in the branch of conditionals we need to point out which statements are in the branch. The conditional expression is analyzed and it is ascertained as to whether a variable used in the conditional expression is high. If it is found that the conditional expression uses a secure variable, then all the statements in the branches need to be analyzed in a high context. In a high context any variable that is assigned to will be added to the set of highs even if a low value is being assigned to it in order to prevent implicit flows. The conditional statements encountered at the `JIMPLE` level are if-statements and switch-case statements. We explain in detail about how the analysis works for the if-statement and the switch-case statement is exactly the same except for the fact that it may have several branches depending on the number of cases.

Consider the program segment:

```
if (a > b)
then
(1)  x = 1;
else
(2)  x = 2;
```

In order to analyze the if-statement given above we require information that both statements (1) and (2) belong to the branches of the if-statement. The control-flow graph used in the analysis does not give this information directly. Therefore another analysis was used for the information. The relationship between branch-dependant statements and the conditional statement can be defined in terms of a postdominance relationship [1]. All the statements which do not post-dominate the conditional statement are branch-dependant excluding the statements in the program before the conditional statement and the statements which are in the program after the merge. We used the Backward Control Flow

[1] Statement s_1 postdominates statement s_2 if every possible execution path from s_2 to exit includes s_1.

Analysis [PV04] which gives exactly the information required. It gives a set of all branch-dependant statements of a conditional statement.

Figure 5.5 presents a control-flow graph with 10 nodes each representing a statement. Node 1 is a conditional-statement. Statements 6 and 9 are the only statements which post-dominate statement 1 since all paths through node 1 will definitely pass through nodes 6 and 9. The control-flow may or may not execute statements in nodes 2 thru 5 or 7 and 8. The statements 2 thru 5 are branch-dependant on node 1 but 7 and 8 are not since they are after the merge node 6 and node s is not branch-dependent since it is before the conditional statement.

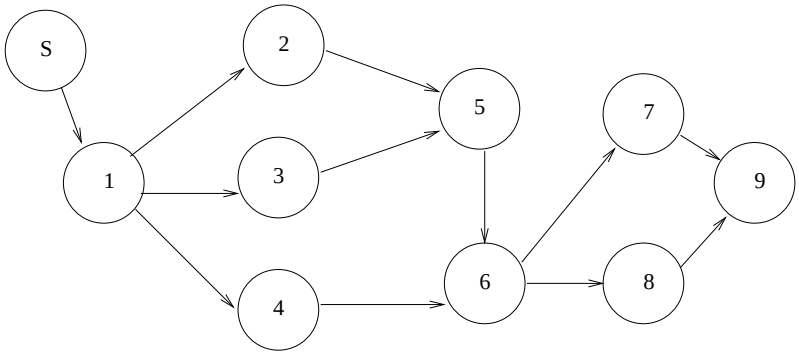

Figure 5.5: *Control-flow Graph*

Algorithm 4 gives the pseudocode of the analysis at a conditional statement of the kind *if* $(a > b)$. The analysis will first check if any of the used variables is in the set of high variables. If any one of them is, then the Backward Control Flow Analysis gives a set of all the branch-dependant statements which is stored in a global Hashtable. The statements in the set will be analyzed in a high context.

5.2.3 Tracking the Return Value of Non-void Methods

Non-void methods return some value which could also be at a high or low security level. Therefore the return statements (example in Figure 5.6) in `JIMPLE` are also analyzed. Algorithm 5 gives the pseudocode of how a return statement is analyzed in a non-void method. A flag is set if the return expression (i0 in Figure 5.6) in a return statement is found to be

Algorithm 4: Analyzing an If-statement in Jimple

input : IfStatement, Gen Set, Kill Set

output: Changed Gen and Kill Sets

```
1 Uses ← IfStatement.GetUses ;
2 if any use is high then
3     Stmts ← IfStatement.GetBranchDependantStmts ;
4     Store Stmts set in a global Hashtable;
5 end
```

at a high security level or the statement is analyzed in a high context. The information set in the flag is used by the callee method if the result of a non-void method is assigned to a local variable.

```
return i0;
```

Figure 5.6: *Return Statement Example*

Algorithm 5: Analyzing a Return Statement

input : ReturnStatement, Gen Set, Kill Set

output: Gen and Kill Sets

```
1 Uses ← AssignStatement.GetUses ;
2 if use is high or statement in high context then
3     set a flag indicating returned value is high;
4 end
```

5.2.4 Exceptions

In Java there are two kinds of exceptions: explicit and implicit. The implicit ones are generated by the JVM when there is some specific runtime problem in the execution of the code. Class Cast Exception is an example of an implicit exception. Implicit exceptions are potential places where an information leak might occur. Such exceptions are not handled in our analysis due to the resulting excessive conservativeness in the call graph. Every Bytecode instruction has the potential of throwing implicit exceptions and this would imply exception edges from pretty much every Bytecode.

There are two constructs in JIMPLE which are specific to explicit exceptions. The first is the throw statement which throws exceptions where as the second one is the catch statement. A catch statement is a kind of identity statement and they are the first statement in an exception handler code block.

Throw statement

In JIMPLE, exceptions are generated and then thrown by the statements as shown in Figure 5.7. The third statement in the figure, throw $r10, actually throws the exception. At such a statement, the analysis checks if the variable used (in this case $r10) is in the set of high variables, or the statement is being analyzed in a high context. If any of the two conditions is true, then the class of the exception is added to the set of highs.

```
$r10 = new java.lang.NumberFormatException;
specialinvoke $r10.<java.lang.NumberFormatException:
                    void <init>()>();
throw $r10;
```

Figure 5.7: *Throw statement*

Catch Statement

The catch statement is the first statement of a block of code that handles the exception. The statement before the caught exception identity statement is a label since all statements in the try statement that the catch covers jump to the label if an exception is generated. In Figure 5.8, an example of an identity statement with a caught exception is presented. At such a statement, the analysis finds out what is the class of the exception being caught. If the set of highs contains an exception that is the same or a subclass of the exception being caught at the identity statement, then the class of the exception is added to the kill set of the statement.

The merge operator of the analysis is union so if an exception is not caught along all branches of a control-flow with several branches, then it stays in the set after the join of all the branches. Some exceptions may be thrown in a method but not caught in the same method. They go beyond the boundary of the method and can be caught in the callee method and so they remain in the set of highs that are passed back to the callee method.

```
label2:
        $r12 := @caughtexception;
```

Figure 5.8: *Catch statement*

5.2.5 Assignment Statement in a High Context

It is possible for an assignment statement to be in the branch of a high-conditional and so it will be analyzed in a high context. The analysis rule of an assignment statement needs to be amended to take this into consideration. The statements which are branch-dependant on a high conditional expression will be analyzed in a high context as follows:

Case: $a = b$

In this case, a will be added to the gen set even if b is not in the set of highs.

Case: $a = b + c$

In this case, a will be added to the gen set even if both b and c are not in the set of highs.

Variable *a* could be a local variable, field (instance or static) or an array write expression. The right hand side of the assignment statement could be a local variable, field (instance or static), an array read expression, an invoke expression or any other expression. The pseudocode of the improved analysis which takes a high context into consideration is presented in Algorithm 6. The only change required is that it is also now checked if the statement belongs to a set of statements which need to be analyzed in a high context.

Algorithm 6: Analyzing an Assignment Statement in a High Context

input : AssignStatement, Gen Set, Kill Set
output: Changed Gen and Kill Sets

```
1  Uses ← AssignStatement.GetUses ;
2  if any use is high or statement is in a high context then
3      set a flag;
4  end
5  Defs ← AssignStatement.GetDefs ;
6  if flag is set then
7      GenSet.Add (Defs);
8  else
9      KillSet.Add (Defs);
10 end
```

5.2.6 Method Invocations

The analysis follows the call graph at a method invocation. At each call site, the receiver suggests the type of the object and the method name identifies the method invoked. It is possible that the receiver can be of more than one type. The call graph provides this information and in the case where more than one type is possible at runtime both methods are statically analyzed. The sets of highs after analyzing each method are merged. An example invocation statement is given in Figure 5.3.

Algorithm 7 presents the pseudocode to analyze method calls. The possible methods which could be called at a call site are obtained from the call graph. It is possible for

information passed through parameters of a call to be at a high security level. Therefore security level information for parameters is passed when the analysis is called on the invoked method. This information is utilized at the identity statements which are present at the beginning of each method in `JIMPLE`. These statements assign the value of the parameters to the locals. One identity statement is present for each parameter of a method. At each such statement, the security level information of the parameters is used to decide if the local should be added to the set of high variables. An example identity statement is given in Figure 5.9 which assigns the value of parameter 1 to local variable i.

Algorithm 7: Analyzing an Invoke Statement

input : InvokeStatement, Gen Set, Kill Set

output: Changed Gen and Kill Sets

1 MethodsInvoked ← InvokeStatement.`GetMethodsInvoked` from call graph;
2 Set security level of parameters;
3 Pass set of high fields as initial set of highs;
4 Analyze all MethodsInvoked ;
5 Merge Results of all methods possibly called;
6 GenSet.`Add` (All high fields and exception classes returned);
7 KillSet.`Add` (All high fields before call);

```
i0 := @parameter1: int;
```

Figure 5.9: *Identity Assignment*

In the analysis the fields are considered globals and so the high fields before the call are present in the initial set of highs for the analysis of the invoked method. The returned set of highs contains the fields which are high after the method has been analyzed and any uncaught high exception classes. The returned set of high fields is merged with the locals variables which were high before the call.

5.2.7 Method Call in a High Context

Invoke statements or assignment statements containing an invoke expression may occur in the branch of a conditional statement which branches on a high variable value or they might be in a high context. In such a case the invoked method is analyzed in a high context to prevent any implicit leak of information. In our analysis the receiver variable (variable $r5 in Figure 5.3) being present in the set of highs is not a reason to analyze the invoked method in a high context.

5.2.8 Limitations of the Analysis

Instance fields cannot be removed from the set of highs because we are not certain that we only have a single instance of a class. An instance field represents the field in all instances of the class in which it is defined and all the subclasses of that class. Therefore the first assignment of a high data to the field makes the field high for the rest of the analysis. In the analysis explained in the next chapter we make a distinction between fields of different instances of a class.

Arrays, once in the set of high information, are not removed from the set of highs because we cannot statically determine the index values in array references if they are not constant values. We consider each array as a single entity with regards to information flow.

In the current implementation of the analysis we handle recursion by pushing all the globals (fields of all application classes) to high when the analysis encounters recursion. A better approximation would be to compute a fixed point.

Library methods are not analyzed since the analysis did not run to completion when very small applications with library calls were analyzed. We include the libraries in our analysis by considering the return values from libraries as high and the high arguments to library methods are counted as warnings since library methods can possibly output high data to a user.

5.3 Proof of Monotonicity

The statement rules update the set of data collected monotonically in order to make sure that the data-flow analysis terminates. We get monotonicity because the rules deterministically analyze the statement depending on the *inSet* values. If we have sets of highs $S1$ and $S2$ where $S1$ is a subset of $S2$ and we analyze the same statement (for example $a = b + c$) with $S1$ as *inSet* and then $S2$ as *inSet* we end up with *outSets* $S1'$ and $S2'$ respectively. According to the analysis rule variable a will be added to the set of highs if either variable b or variable c is high. $S1'$ will still be a subset of $S2'$ after analyzing the statement because if variable a is added to $S1$ it will also be added to $S2$ but not necessarily the other way round. This is accurate because if variable b or c are in $S1$ they will also be in $S2$. However, it is possible that either b or c are in $S2$ but not in $S1$. In this case set a will be added to $S2$ but not to $S1$ when the statement is analyzed.

5.4 Summary

We have presented the details of the information flow analysis which considers all fields as globals in this chapter and the security level of fields belonging to different instances of an object is merged together. The next chapter describes the analysis which uses points-to information to differentiate between fields belonging to different instances of the same class.

Chapter 6
Data-Flow Analysis Using Spark Information

Spark [LH03, Lho02] is a package in Soot which measures the context-insensitive [1] points-to information for variables in a JIMPLE program. If a local variable used anywhere in JIMPLE code points to an object Spark gives the allocation site (one or many) whose instance the variable may be pointing to. This way we can differentiate between data for two different instances of the same class. In the code snippet given below the two statements instantiate two instances of the Object class in Java and are assigned to variables o1 and o2. The instance fields in o1 and o2 will be independent. Spark will provide information that o1 and o2 have different allocation sites.

```
Object o1 = new Object();
Object o2 = new Object();
```

The analysis described in Chapter 5 considered a single instance for all fields of a class. Essentially we made no differentiation between a static field and an instance field; we considered different instances of a field for the same class or its subclasses as the same. In this chapter we present the second analysis which uses Spark points-to information in order to differentiate between instance fields of different objects and between different array objects.

[1] In a context-insensitive analysis the analysis approximates the side-effects of the method invoked at a call site [Muc97]

6.1 Spark Points-to Information

The alias information from Spark is used to make a distinction between data of different instances of objects. This will impact the information flow analysis because now the first assignment of a high data to an instance field of a class will not make it high for the rest of the analysis. In fact now we only need to make the instance field of particular instance(s) high depending upon the information obtained from Spark. The code presented in Figure 6.1 has two allocation sites (labelled Allocation Site 1 and Allocation Site 2) for initializing instances of class Two. The instance field i for the class Two will be independent for object instances obj1 and obj2 and so the security level of field i in obj1 does not affect the security level of field i in obj2. In the previous analysis at statement S1 (obj1.i = x), if x is high then field i also becomes high. When the statement S2 is analyzed, field i remains high even though it is assigned a constant value 4. At statement S3, even though myInt is assigned the value of instance obj2's field i, it will still become high because the analysis cannot tell the difference between instance field i of obj1 and obj2. Spark can differentiate between the two instances obj1 and obj2. The Spark points-to analysis is used in the analysis to obtain the allocation sites of the objects pointed-to by variables obj1 and obj2. At statement S3 (myInt = obj2.i), Spark will give information that obj2's allocation site is Allocation Site 2 and since field i of obj2 is low, myInt will remain low. This helps in reducing the number of statements in the program where secure data is used. Assignment of field i of obj2 is actually safe but due to the conservative approximation of the first analysis it was considered unsafe.

Spark cannot always deterministically tell that a local points to a specific allocation site. In the code given in Figure 6.2, at statement S1 (obj1.i = x), obj1 may point to the Allocation Site 1 before the if-statement or the Allocation Site 2 in the branch of the if-statement. It cannot be confirmed which allocation site obj1 points to if boolean value of the conditional expression of the if-statement cannot be ascertained statically. In this case, the instance field i for both allocation sites would be made high if x is high.

Similar to the analysis in Chapter 5 a high field cannot be made low since we have a may points-to analysis and not a must points-to analysis in Spark. A may points-to analysis cannot tell that a local definitely points-to only one instance of a class. Such information

```
public class FirstTest{
    public static int x = 100;
    public static void main (String[] args){
        int myInt, myInt2;
        myInt2 = Two.j;
        Two obj1, obj2;
        obj1 = new Two(); //Allocation Site 1
        obj2 = new Two(); //Allocation Site 2
        obj1.i = x; //S1
        obj2.i = 4; //S2
        myInt = obj2.i; //S3
    }
}
class Two{
    public int i;
    public static int j = 50;
    public Two(){
    }
}
```

Figure 6.1: *Example Highlighting Usefulness of Spark*

comes from a must points-to analysis. Consider the example in Figure 6.3 in which we have Allocation Site 1 which is the only allocation site for initializing objects of class Two. Assume statement S1 assigns a high data value x to field i of obj1. At statement S2 field i is assigned a constant value which is at a low security level. Even though a low value is assigned to field i we cannot change the security level of the field to low. The points-to analysis will return just one allocation site (Allocation Site 1) for obj1 but we can observe that the allocation site is in a while loop and we do not have the information that it will be executed only once.

```
public class SecondTest{
    public static int x = 100;
    public static void main (String[] args){
        int myInt, myInt2;
        myInt2 = Two.j;
        Two obj1;
        obj1 = new Two(); //Allocation Site 1
        if(myInt2 > 100)
            obj1 = new Two(); //Allocation Site 2
        obj1.i = x; //S1
    }
}

class Two{
    public int i;
    public static int j = 50;
    public Two(){
    }
}
```

Figure 6.2: *More Than One Allocation Site*

6.2 Incorporating Points-to Information

In order to incorporate points-to information into the basic analysis described in Chapter 5 a number of changes had to be made to both the initial data structures and to the analysis rules.

6.2.1 Changes in the Data Structures Used

Changes to the domain of high data collected in the analysis necessitates some changes to initial data structures used. The flow sets from the previous analysis now contain high

```
public class SecondTest{
    public static int x = 100;
    public static void main (String[] args){
        int myInt, myInt2;
        myInt2 = 120;;
        Two obj1;
        while(myInt2 > 100){
            obj1 = new Two(); //Allocation Site 1
            myInt2−−;
        }
        obj1.i = x; //S1
        obj1.i = 5; //S2
    }
}

class Two{
    public Two(){}
}
```

Figure 6.3: *Allocation Site in a While Loop*

static fields, local variables and classes of high exceptions.

In order to track high instance fields we added a global Hashtable into the analysis. In our case since we need to differentiate between instance fields of objects instantiated at different allocation sites, the fields are the natural choice for the key and a List of allocation nodes (corresponding to allocation sites) for which the key (field) is high are the values.

We had to introduce another global data structure to keep a record of high instances of arrays. A List data structure was sufficient. Arrays are objects and they have allocation nodes for the statement where they were initialized. An array initialization statement in Java code (labelled ”Array initialization”) is presented in Figure 6.4. We only need to keep track of the allocation sites for the high arrays which can be stored in a List.

```
public class FirstTest{
    public static void main (String[] args){
        int[] myArray = new int[10]; //Array initialization
        myArray[1] = 5;
    }
}
```

Figure 6.4: *Array Initialization*

6.2.2 Analysis Rules For Expressions

The analysis rules for statements that have fields and arrays had to be adjusted to make use of the points-to information from Spark. We explain how each of the expressions is handled by our analysis in the following sections with the aid of example code snippets and algorithm descriptions.

Instance Field Read

Field accesses in JIMPLE occur on the right hand side of an assignment statement (example in Figure 6.5). In order to ascertain the security level of the field that is being accessed the we have to find out which allocation site r1 points-to. Once we have obtained the information about the allocation site(s) that r1 points to, we matched it with the allocation sites (instances of class Two) for which field i is high. If any of the allocation sites that r1 points-to has field i high, then a flag is set. If the flag is set, the local variable (in this case i2) is added to the gen set, otherwise, i2 is added to the kill set. The algorithm for analyzing instance field read is presented in Algorithm 8.

Instance Field Write

Instance field writes occur on the left hand side of the assignment statement in JIMPLE. The right hand side of the same assignment statement will have a local variable (as shown

```
i2 = r1.<Two: int i>;
```

Figure 6.5: *Example of Instance Field Read in Jimple*

Algorithm 8: Analyzing Instance Field Read

input : AssignStatement, Gen Set, Kill Set
output: Changed Gen and Kill Sets

```
1  InstanceField ← AssignStatement.GetInstanceField ;
2  LocalVariable ← AssignStatement.GetLocalVariable ;
3  AllocationNodes ← PointsToAnalysis.GetAllocNodes (LocalVariable);
4  if InstanceField is high for any AllocationNode that LocalVariable points-to
   then
5      set a flag;
6  end
7  Defs ← AssignStatement.GetDefs ;
8  if flag is set then
9      GenSet.Add (Defs);
10 else
11     KillSet.Add (Defs);
12 end
```

in Figure 6.6) or a constant. The decision to mark field j high will depend on the security level of variable i3. If i3 is high then field j will be marked high for all the instances of class Two which the variable r1 points-to. In the analysis the allocation sites of the instances are ascertained and they are added to the global Hashtable which records the high instance fields. The procedure is presented in Algorithm 9. If variable i3 is not in the set of highs nothing is done since we cannot move a field to low due to the fact that we are working with a may points-to analysis.

```
r1.<Two: int j> = i3;
```

Figure 6.6: *Example of Instance Field Write in Jimple*

Algorithm 9: Analyzing Instance Field Write

input : AssignStatement, Gen Set, Kill Set

output: Changed Global High Instance Field Hashtable

```
1  Uses ← AssignStatement.GetUses ;
2  if any use is high then
3      set a flag;
4  end
5  InstanceField ← AssignStatement.GetInstanceField ;
6  LocalVariable ← AssignStatement.GetLocalVariable ;
7  AllocationNodes ← PointsToAnalysis.GetAllocNodes (LocalVariable);
8  if flag is set then
9      mark InstanceField high for all AllocationNodes LocalVariable may
       point-to in global Hashtable;
10 end
```

Array Initialization

Arrays are just like objects and are treated as such by the points-to analysis. All variables which reference an array point to one or more allocation sites where the array being referenced was initialized. Array initialization expressions are present on the right hand side of assignment statements. The expressions use a local in the case of a single dimensional array or several locals in the case of multi-dimensional array initializations. In the example in Figure 6.7, a single dimensional array is initialized and the length is specified by the local variable i5. If i5 is found to be high then the allocation node for the array pointed to by local variable r1 is added to the *List* which contains the allocation sites of all the arrays

which are high. The left hand side of an array initialization expression is always assigned to a local. The functioning of the analysis for array initialization expressions is presented as pseudocode in Algorithm 10.

```
r1 = newarray (int)[i5];
```

Figure 6.7: *Example of Array Initialization in Jimple*

Algorithm 10: Analyzing Array Initialization

input : AssignStatement, Gen Set, Kill Set
output: Changed High Array Instances' List

```
1 Uses ← AssignStatement.GetUses ;
2 if any use is high then
3     set a flag;
4 end
5 LocalVariable ← AssignStatement.GetDef ;
6 if flag is set then
7     AllocationNode ← PointsToAnalysis.GetAllocNode (LocalVariable);
8     Add AllocationNode to List of High Array Instances;
9 end
```

Array Length Operation

This case is similar to the array initialization. The length expression occurs on the right hand side of an assignment statement and the result is assigned to a local variable. An example statement is presented in Figure 6.8. In order to ascertain the security level of the array referenced by variable r2 the allocation nodes which r2 may point-to are obtained by using the points-to analysis. If any of the allocation nodes are present in the *List* of high

array instances a flag is set and using the information from the flag the local variable (i5 in the example) is added to the gen set (if flag is true) or to the kill set (if flag is false). The procedure is described in Algorithm 11.

```
r2 = newarray (int)[i3];
i5 = lengthof r2;
```

Figure 6.8: *Example of Array Length Expression in Jimple*

Algorithm 11: Analyzing Array Length Expression

input : AssignStatement, Gen Set, Kill Set
output: Changed Gen and Kill Sets

```
1  Use ← AssignStatement.GetUse ;
2  if Use instanceof LengthExpr then
3      LocalVariable ← LengthExpr.GetLocalVariable ;
4      AllocationNodes ← PointsToAnalysis.GetAllocNodes (LocalVariable);
5  end
6  if any AllocationNode is in List for high Arrays then
7      set a flag;
8  end
9  Defs ← AssignStatement.GetDefs ;
10 if flag is set then
11     GenSet.Add (Defs);
12 else
13     KillSet.Add (Defs);
14 end
```

Array Read

The array location access expression is always on the right hand side of an assignment statement and the location value is assigned to a local variable as shown in Figure 6.9. In such an expression there is a base variable (r2 in Figure 6.9) which points-to the array object and an index variable (i8 in Figure 6.9) or constant which gives the index of the array location to access. The analysis has to check the security level of the array object as well as the index variable to decide whether to add i3 to the gen set or the kill set. If either the array object is high or the index variable is high then i3 will be added to the gen set. The security level of the array object is ascertained by using the points-to analysis to find out to which allocation node(s) r2 points-to. If any of the allocation node(s) is in the High List of arrays or the index variable is high before the statement the flag is set to true. This method for analyzing the statement is presented in Algorithm 12.

```
i3 = r2[i8];
```

Figure 6.9: *Example of Array Read in Jimple*

Array Write

The array location write expression is on the left hand side of an assignment statement and it is assigned the value of a local variable or a constant. i4 is the local variable whose value is assigned to the array r2 at index i0 in the example in Figure 6.10. The analysis decides to add the array allocation node pointed to by the base variable to the High List of arrays if either the value on the right hand side (i4 in Figure 6.10) of the assignment statement is high or the index variable (i0 in Figure 6.10) is high. If either or both of values is high a flag is set and then the allocation node(s) pointed-to by the array base variable (r2 in Figure 6.10) is added to the global *List* marking all high array instances. The analysis procedure is presented in Algorithm 13.

Algorithm 12: Analyzing Array Location Read

input : AssignStatement, Gen Set, Kill Set

output: Changed Gen and Kill Sets

```
1  ArrayExp ← AssignStatement.GetArrayExp ;
2  BaseVariable ← ArrayExp.GetBaseVariable ;
3  IndexVariable ← ArrayExp.GetIndexVariable ;
4  AllocationNodes ← PointsToAnalysis.GetAllocNodes (BaseVariable);
5  if ArrayInstance is high for any AllocationNode that BaseVariable points-to or
   IndexVariable is high then
6      set a flag;
7  end
8  Defs ← AssignStatement.GetDefs ;
9  if flag is set then
10     GenSet.Add (Defs);
11 else
12     KillSet.Add (Defs);
13 end
```

```
r2[i0] = i4;
```

Figure 6.10: *Example of Array Location Write in Jimple*

Algorithm 13: Analyzing Array Location Write

input : AssignStatement, Gen Set, Kill Set

output: Changed Global High Instance Array List

```
1  Use ← AssignStatement.GetUse ;
2  ArrayExp ← AssignStatement.GetArrayExp ;
3  IndexVariable ← ArrayExp.GetIndexVariable ;
4  if any use is high or IndexVariable is high then
5      set a flag;
6  end
7  BaseVariable ← ArrayExp.GetBaseVariable ;
8  AllocationNodes ← PointsToAnalysis.GetAllocNodes (BaseVariable);
9  if flag is set then
10     add all AllocationNodes BaseVariable may point-to in the global List;
11 end
```

Special Case for Main Method String Array Argument

The command-line arguments to the main method are considered high security in our analysis. We had to specify a special rule to analyze the identity statement which assigns the string array argument of the main method to a local variable. The only argument to a main method is the string array and so when the analysis encounters an identity statement with a parameter reference (example shown in Figure 6.11) while analyzing the main method it has to be the assignment of the string array argument to a local variable. The local variable on the left hand side of the identity statement (r0 in Figure 6.11) points-to the allocation node of the string array. The points-to analysis gives the information about the allocation node which is added to the List of high array instances. The pseudocode is presented in Algorithm 14. The initialization statement for the string array is in a method which is executed before the main method. We did not have to analyze that method with the initialization statement because the local variable on the left hand side of the identity statement in the main method points-to the allocation site and the identity statement is analyzed before any other statement in JIMPLE code.

```
public static void main(java.lang.String[])
{
    java.lang.String[] r0;
    r0 := @parameter0: java.lang.String[];
}
```

Figure 6.11: *Main Method String Array Argument*

Algorithm 14: Analyzing Identity Statement of Main Method

input : IdentityStatement, Gen Set, Kill Set

output: Changed High Array Instances' List

1 **if** *main method is being analyzed and identity statement has a parameter reference* **then**

2 LocalVariable ← IdentityStatement.`GetDef` ;

3 AllocationNode ← PointsToAnalysis.`GetAllocNode` (LocalVariable);

4 Add AllocationNode to List of High Array Instances;

5 **end**

6.2.3 Statements in High Contexts

It is possible that assignment statements that have an instance field reference or an array reference occur in the branch of a high conditional or a method which is analyzed in a high context. The algorithms in Section 6.2.2 will adjust accordingly if the statements are analyzed in a high context since now even if the right hand side data is not high the left hand side data will be added to the set of highs. The local variable of a primitive type on left hand side will be added to the gen set. In the case when the local variable references an object, then:

- for a field reference, the field is marked high for all the allocation node(s) that the local variable may point-to in the global Hashtable which tracks high instance fields;

and

- for an array reference, all allocation node(s) that the local variable may point-to are added to the global List which tracks the high array instances.

6.3 Summary

We explained the analysis which uses information obtained from the Spark points-to analysis in this chapter. First we gave the benefit of using points-to information in an information flow analysis and then we described the changes that we had to make in terms of the data structures used and the statement analysis rules to adapt the first analysis explained in Chapter 5. In the next chapter we present our experimental findings on the two analyses.

Chapter 7
Experimental Results

This chapter reports an empirical study on the information flow analyses. The first section describes the experimental model, the second section presents a view of the high data in a program in the Soot Eclipse Plug-in, the third section describes the metrics and the last section gives a brief overview of the benchmarks and the tabulated data with a discussion on the values calculated. A user guide is given in Appendix A where information on where to obtain Soot and the analysis code can be found.

7.1 Experimental Model

The goal of an information flow analysis is to track secure data in a program and identify places in the program where the data may leak to unwanted users. Figure 7.1 presents this idea. Before the assignment statement ($a = b + c$) variable b is high and it is used as an operand in the addition expression and the result is assigned to variable a. Since a is assigned the result of a computation which uses a secure data it is also added to the set of high data. The print statement after the assignment statement gives out the value stored in variable a to a user. a stores a high data value and it should only be given out to a user who has the right permission to observe secure data. Our analyses track the secure data and generate a warning if there is a possibility of giving out the data to a user.

In Java data is given out to a user in many ways some of which are standard output, graphical user interface or database writes. These all invoke a Java library method which

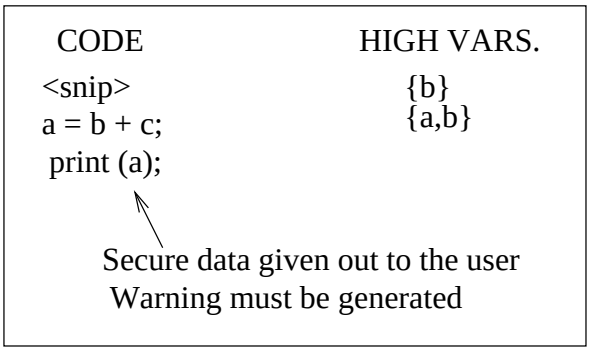

Figure 7.1: *Experimental Model*

calls a native method on the machine in the call chain to eventually give out the data to a user. Pursuing our goal of providing a practical solution to the information flow problem we tried to analyze all of Java library code and generate a warning when an actual native method is called with an argument which is high. We ran the simplest benchmark on the library but our analysis could not to completion due to lack of memory. Since the number of possible calling sequences are exponentially big in the range of 10^{11} we had to mark out the library code in our tests. In our tests we analyze the Java application classes and generate warnings whenever a Java library method is invoked and it is passed a high data value as argument. The analyses also count the number of information flows possible in a program either implicitly or explicitly at assignment statements.

The analyses keep track of the set of high data at each statement in a program during the inter-procedural analysis. If a statement is encountered more than once in the analysis the merge (union) set of the high data is stored each time. Once the analysis completes analyzing the program we have a set of highs associated with each statement in the program. We use the high set of data to calculate values for our metrics presented in Section 7.3. The metrics provide the following information about secure data:

- in the case of assignment statements whether they read or write secure data;
- in the case of catch statements for exceptions if they catch a high exception in a low context; and

- in the case of a library method invocations if high data is passed as argument.

7.2 Sample Run and Viewing Results in Eclipse

Eclipse [Ecl] is an open-source, extensible integrated development environment (IDE). Eclipse was designed as a plug-in framework where one can easily add new functionality. Soot has a plug-in [Lho05] for Eclipse and the result of the analysis can be viewed in it. Figure 7.2 presents a sample program. The code presented is Java and it only reflects how the analysis manages the set of high data at each statement. The set of secure variables is given after each statement in the code. The initial set is assumed to contain the private field *i* of the class Test. At each statement it can be seen that the set of high variables changes depending on the security level of the variables used and defined in the statement and the set of secure variables before the statement. The conditional expression in the if-statement is a high conditional since the variable *y* is high when it is evaluated. Therefore variable *x* in the assignment statement, which is in the branch of the if-statement, is added to the set of high data even though it is assigned a value of a constant which is at a low security level.

The same program's `JIMPLE` code is analyzed using our analysis assuming field *i* as the initial secure data value. The analysis collected the high data sets for each statement and the variables which can store a high data value are tagged. The tagged variables are given a dark background in the `JIMPLE` output which is viewed in the Soot Eclipse plug-in. A sample screen shot is given for the tagged `JIMPLE` code in Figure 7.3. The variables which are high at each statement in the `JIMPLE` code can be identified clearly.

7.3 Metrics

The two categories of the kinds of information flows are: explicit flows and implicit flows. Besides the explicit and implicit flows there are possible information flows due to exceptions and secure data being passed to libraries in our analyses. We define metrics which give numerical values for the kind of information flows and warnings which can occur in a program. There are no well known metrics to evaluate information flow in programs.

```
//The set of high data is given after each statement in curly brackets
public class Test {
    private static int i; Initial set: {i}
    public static void main(String[] args){ {i}
        int x,y,z; {i}
        x = i; {i,x}
        x = 1; {i}
        y = i; {i,y}
        z = 8; {i,y}
        if(y == z){ {i,y}
            x = 5; {i,y,x}
        }
    }
}
```

Figure 7.2: *Example Run of Analysis*

We specified eight metrics: six of which are specific to assignment statements in JIMPLE and one each counts an important aspect about exceptions and secure data leaking to library code. Numbers for the same metrics were counted for both our analyses, the first one explained in Chapter 5 and the second one explained in Chapter 6, to find out the affect of using points-to information to differentiate between instance data of two instances of a class has on the different kinds of information flows and warnings. Each of the defined metrics are now explained in turn.

Explicit Flow from a High to Low ($H \hookrightarrow L$)

This metric gives the number of statements which assign secure data to variables in a program which previously did not store secure data. Such statements allow secure data to spread in the program.

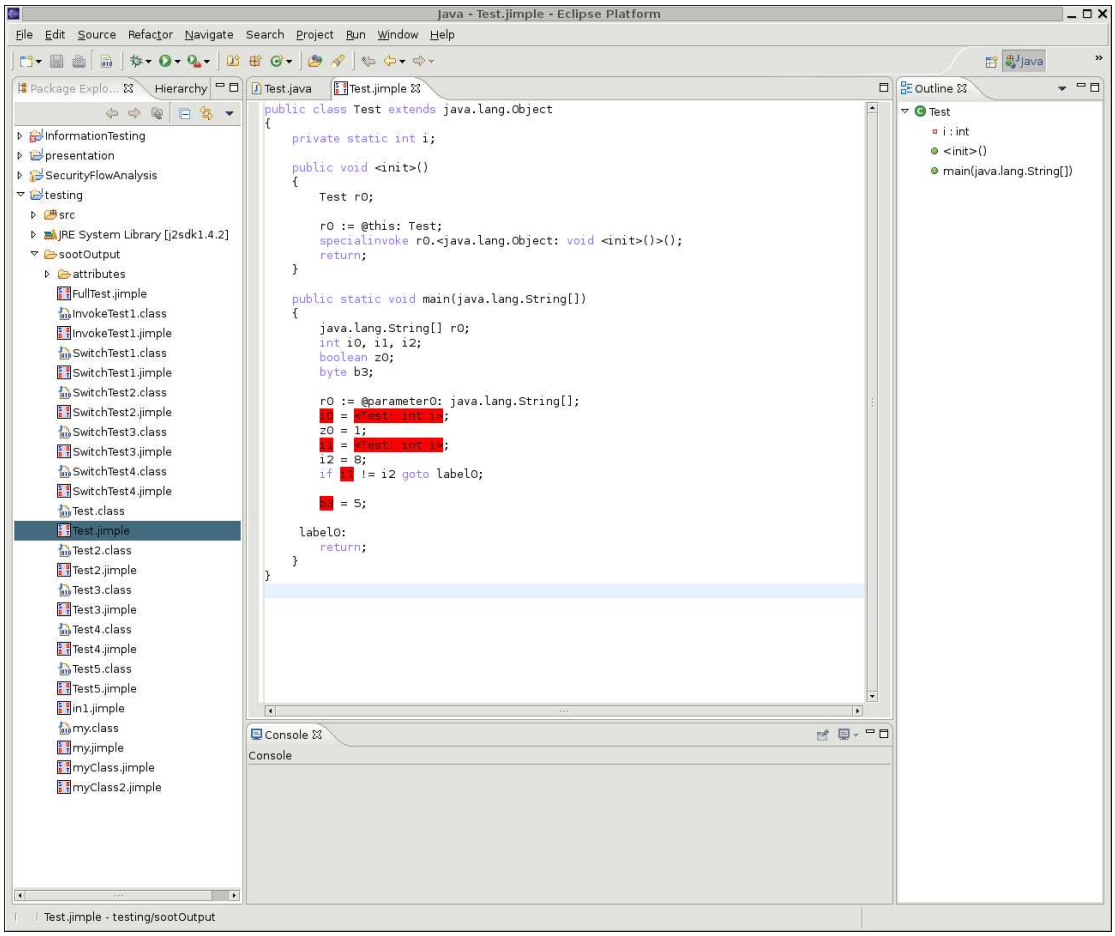

Figure 7.3: *Analysis Results in Eclipse*

Explicit Flow from a High to High ($H \hookrightarrow H$)

This flow occurs when the right hand side of an assignment statement has a high use but the left hand side is also high. A high numerical value for this flow suggests that secure data is present in most of the statements in a program.

Explicit Flow from a Low to High ($L \hookrightarrow H$)

In the case when the right hand side of an assignment has no high use it is not required to push the left hand side to high. When it is found to be high even though the right hand side is low a flow from a low to a high is added. This statement is safe and does not spread secure data in the program.

Explicit Flow from a Low to Low ($L \hookrightarrow L$)

This metric is quite straight forward. A safe assignment is counted in this case when there is no secure data on the left hand side or the right hand side of the assignment. High number of these statements in a program is a good sign as it suggests that most of the data in the program is not confidential.

Implicit Flow from a High to Low ($H \rightsquigarrow L$)

When an assignment statement is in the branch of a high conditional statement or the method is being analyzed in a high context all assignments are considered secure since partial information about a secure conditional expression can be possibly known by an unwanted user. Therefore the left hand side of an assignment is always marked secure. This metric counts the number of times the left hand side was low before the statement was analyzed and is pushed to high at the particular statement. A large number of these implicit flows suggest many conditional statements in the program which use secure data.

Implicit Flow from a High to High ($H \rightsquigarrow H$)

This metric counts those instances when the left hand side of an assignment statement, which happens to occur in the branch of a high conditional or a high context, is already high. High numbers of this flow in a program suggests that there is widespread use of secure data in the program.

Catch Exception in a Low Context

Explicit exceptions are not used commonly used by programs in a program to define control flow. They are used at specific places when it is important to catch an exception or reason about a certain condition in the code. However, they do occur and in our analyses we handle exceptions by adding the class of the exception to the set of high variables when a high exception is thrown. When an exception is caught in a low context it is possible that information may leak implicitly. This metric counts the number of times a high exception is caught in a low context.

Secure Data to Library Code

Library code is executed by way of a method invocation. Since our analyses only analyze application methods and data can be given out to a user by some library methods we make a note of all high data values that are passed to a library method. Every time a library method is invoked this metric counts the number of arguments which are high.

7.4 Experimental Results

This section presents the benchmarks on which the analyses were run and the results. These experiments were performed to ascertain the kind of information which is present in programs. They suggest how secure data flows through a program and how widespread is secure data throughout the program.

The different options which were tried on the analyses are as follows: library safe and unsafe, recursion taken into consideration or not and whether points-to information is used or not. In the remainder of this section we refer to the data-flow analysis which does not use points-to information (explained in Chapter 5) as the first analysis and the analysis which uses the points-to information provided by Spark (explained in Chapter 6) as the second analysis.

Library Safe and Unsafe Option

When the library is considered safe the return value of a library method call is assumed at a low security level whereas when the library is assumed to be unsafe the return value is considered high. If the return of an invocation is assigned to a local in the case when the library is assumed unsafe, the local which is assigned to becomes secure and this is a source of secure data coming into the program. We count the number of high data values which are passed as argument to library methods only when the libraries are considered unsafe.

Recursion Accounted for or Not

When recursion is encountered and it is accounted for in the first analysis all fields belonging to the application classes are made secure because we are not certain which fields might be touched in the recursive call. When recursion is accounted for in the second analysis all static and instance fields are pushed to high. In the case when it is not accounted for in both the analyses no fields are made high at a recursive call.

Points-To Information Used or Not

The second analysis uses the points-to information provided by Spark and is tested with either library safe or unsafe options. For both cases, in one run recursion is not accounted for and in the second it is. When the points-to information is utilized we are able to differentiate between the data stored in the instance fields for different objects.

7.4.1 Benchmarks

The benchmarks used to test our analyses are from the Optimizing Compilers class at McGill University. The benchmarks are moderate in size and the numbers generated for the metrics described in the previous section demonstrate the kind of data which is present in the programs. We ran our tests on six benchmarks which have varying properties. Some of them are intensive on object allocations while some have a lot of conditional statements. Three of them have recursive calls. The following are the benchmarks used:

- **PointsToGraph:** generates a points-to graph for some arbitrary variables and their allocation sites;
- **DFT:** performs Discrete Fourier Transform on sequences;
- **Coefficients:** is a library for matrix operations;
- **Froggy:** is an interpreter for a language called Froggy, which is a small but useful subset of Scheme;
- **Puzzle:** finds a solution to the sliding block puzzle problem by applying A-star search algorithm; and
- **Mersenne Prime:** takes a number n as input and provides the nth Mersenne Prime in the Mersenne Prime List.

7.4.2 Tabulated Results and Discussion

We tested a total of eight combinations of the three options and the results are given in this section. The tables give the percentage for each kind of explicit and implicit flow and a total count of the number of information flows which corresponds to the number of assignment statements in the program. The warnings table is given for the experiments in which the library was considered unsafe. None of the benchmarks has a catch statement which catches a high exception in a low context. The information on the machine specifications on which the tests were carried out can be found in Appendix A.

Library Safe without Recursion

Table 7.1 lists the results for the case of testing the data-flow analysis with the library safe option and skipping recursion without doing anything. All flows in all the benchmarks are explicit ones from low to low because there is no high data in the program other than the main string array which is not used in any benchmark other than Puzzle. In Puzzle the main argument introduces high data which is assigned to local variables and so it has 5% explicit flows other than low to low.

Benchmarks	$H \hookrightarrow L$	$H \hookrightarrow H$	$L \hookrightarrow H$	$L \hookrightarrow L$	$H \rightsquigarrow L$	$H \rightsquigarrow H$	Total Flows
PointsToGraph	0%	0%	0%	100%	0%	0%	289
DFT	0%	0%	0%	100%	0%	0%	1672
Coefficients	0%	0%	0%	100%	0%	0%	410
Froggy	0%	0%	0%	100%	0%	0%	272
Puzzle	4%	0%	1%	95%	0%	0%	246
MersennePrime	0%	0%	0%	100%	0%	0%	436

Table 7.1: *Library Safe without Recursion*

Library Safe with Recursion

Table 7.2 gives the result for the option when we considered library safe but at recursion all the fields belonging to the application classes were added to the high set. Now that there is another source of secure data in the programs we have several more flows in this case than the previous case in which we did not introduce any new high fields at a recursive method call. The results for DFT, Puzzle and MersennePrime remain the same since they have no recursion. Froggy has a lot of implicit flows ($H \rightsquigarrow L$) since it is an interpreter and has switch-case statements with several cases. This causes many statements to be in the branch of a conditional expression which branches on a high value. The presence of high data in the three benchmarks with recursion shows that there is a very large input of high data when we consider the conservative assumption that all fields become high at a recursive method call. This also highlights the need to deal with recursion in a better manner.

Library Unsafe without Recursion

The results for this option are presented in Table 7.3 and Table 7.4. In this case we consider the main string array as high and all returns from the library calls as high. Compared to the case with Library safe without Recursion all benchmarks report that high data is present in some statements. Most of the programs written in Java access library code

Benchmarks	$H \hookrightarrow L$	$H \hookrightarrow H$	$L \hookrightarrow H$	$L \hookrightarrow L$	$H \rightsquigarrow L$	$H \rightsquigarrow H$	Total Flows
PointsToGraph	3%	4%	1%	61%	17%	14%	289
DFT	0%	0%	0%	100%	0%	0%	1672
Coefficients	9%	2%	4%	57%	22%	6%	410
Froggy	9%	6%	1%	34%	42%	8%	272
Puzzle	4%	0%	1%	95%	0%	0%	246
MersennePrime	0%	0%	0%	100%	0%	0%	436

Table 7.2: *Library Safe with Recursion*

which introduces high data in this case. DFT reports the highest percent of implicit flows. This is due to the fact that it has many method calls in the branch statements of conditional expressions and the branch condition could have a high data value. Again Froggy has a high percentage of implicit flows. It is not surprising because Froggy is a subset of Scheme and interpreters for functional languages have a large number of switch-case statements. We also report warnings for data leaks to libraries for this case since the libraries are considered unsafe. Explicit exceptions are scarcely used and no benchmark reports the catch of a high exception in a low context.

Benchmarks	$H \hookrightarrow L$	$H \hookrightarrow H$	$L \hookrightarrow H$	$L \hookrightarrow L$	$H \rightsquigarrow L$	$H \rightsquigarrow H$	Total Flows
PointsToGraph	7%	8%	1%	37%	22%	25%	289
DFT	11%	4%	0%	24%	21%	40%	1672
Coefficients	10%	2%	1%	42%	23%	22%	410
Froggy	5%	3%	0%	45%	42%	5%	272
Puzzle	27%	7%	1%	42%	13%	10%	246
MersennePrime	6%	0%	0%	79%	10%	5%	436

Table 7.3: *Library Unsafe without Recursion*

Benchmarks	Warnings Exceptions	Warnings Data to Library
PointsToGraph	0	24
DFT	0	32
Coefficients	0	4
Froggy	0	8
Puzzle	0	15
MersennePrime	0	16

Table 7.4: *Library Unsafe without Recursion*

Library Unsafe with Recursion

Table 7.5 and Table 7.6 give the results for the case where high data is introduced by library calls as well as recursive calls. Compared to the previous case in which only library calls returned high data, the number of explicit flows $L \hookrightarrow L$ values goes down or stays the same for all benchmarks. In the case of PointsToGraph, Coefficients and Froggy it goes down by a big percent where as in the case of the other three benchmarks it stays the same since they have no recursive method call. Since more high data is entering the program when recursion is encountered more statements process high data and so the explicit flows $L \hookrightarrow L$ count decreases. In the case of Froggy the count for the number of warnings for high data given out to library increases from 8 in the previous case to 17 since Froggy is an interpreter for a functional language and has a lot of recursive calls to the evaluation function.

Points-To Library Safe without Recursion

The results for this option are presented in Table 7.7. As expected we find that most flows are explicit ones from low to low since there is no high data in the programs except the main string array argument. Puzzle has 4% explicit flows $H \hookrightarrow L$ since it uses the main

Benchmarks	$H \hookrightarrow L$	$H \hookrightarrow H$	$L \hookrightarrow H$	$L \hookrightarrow L$	$H \rightsquigarrow L$	$H \rightsquigarrow H$	Total Flows
PointsToGraph	10%	8%	1%	22%	23%	36%	289
DFT	11%	4%	0%	24%	21%	40%	1672
Coefficients	13%	2%	3%	36%	23%	23%	410
Froggy	10%	6%	1%	29%	44%	10%	272
Puzzle	27%	7%	1%	42%	13%	10%	246
MersennePrime	6%	0%	0%	78%	11%	5%	436

Table 7.5: *Library Unsafe with Recursion*

Benchmarks	Warnings Exceptions	Warnings Data to Library
PointsToGraph	0	24
DFT	0	32
Coefficients	0	4
Froggy	0	17
Puzzle	0	15
MersennePrime	0	16

Table 7.6: *Library Unsafe with Recursion*

string array argument at several places where as the other benchmarks have a 100% count for explicit flows $L \hookrightarrow L$.

Points-To Library Safe with Recursion

The results for this option are presented in Table 7.8. As expected the benchmarks with recursive method calls PointsToGraph, Coefficients and Froggy now have a drop in the explicit flows $L \hookrightarrow L$ since all fields become high at a recursive method call. An intersting thing to note in these results is that DFT also has flows other than explicit flows $L \hookrightarrow L$.

Benchmarks	$H \hookrightarrow L$	$H \hookrightarrow H$	$L \hookrightarrow H$	$L \hookrightarrow L$	$H \rightsquigarrow L$	$H \rightsquigarrow H$	Total Flows
PointsToGraph	0%	0%	0%	100%	0%	0%	289
DFT	0%	0%	0%	100%	0%	0%	1672
Coefficients	0%	0%	0%	100%	0%	0%	410
Froggy	0%	0%	0%	100%	0%	0%	272
Puzzle	4%	0%	0%	96%	0%	0%	246
MersennePrime	0%	0%	0%	100%	0%	0%	436

Table 7.7: *Points-To Library Safe without Recursion*

This is due to the fact that recursion in the library calls caused high data to enter into the program.

Benchmarks	$H \hookrightarrow L$	$H \hookrightarrow H$	$L \hookrightarrow H$	$L \hookrightarrow L$	$H \rightsquigarrow L$	$H \rightsquigarrow H$	Total Flows
PointsToGraph	3%	4%	1%	61%	17%	14%	289
DFT	10%	3%	2%	30%	19%	36%	1672
Coefficients	9%	3%	4%	56%	12%	16%	410
Froggy	9%	6%	1%	34%	42%	8%	272
Puzzle	4%	0%	0%	96%	0%	0%	246
MersennePrime	0%	0%	0%	100%	0%	0%	436

Table 7.8: *Points-To Library Safe with Recursion*

Points-To Library Unsafe without Recursion

Table 7.9 and Table 7.10 give the results for this case. The results have a variety of flows but they are comparable to the results of the first analysis considering library unsafe and not accounting for recursion. We see a drop in the high data present in the program for

benchmarks which allocate many object instances. Froggy and DFT have a high percentage of implicit information flows. Coefficients generates a lot of array instances and it has 6% more explicit flows $L \hookrightarrow L$ than the first analysis. We also see that in the case of Coefficients the warnings for high data leaking to library have also gone down from 4 to 3. Puzzle also generates a lot of objects in the Astar search and so we see 4% more explicit flows $L \hookrightarrow L$ than in the case of the first analysis. The results suggest that tracking data for different instances of fields and arrays helps in reducing high data in the program and number of possible information leaks to library for specific benchmarks.

Benchmarks	$H \hookrightarrow L$	$H \hookrightarrow H$	$L \hookrightarrow H$	$L \hookrightarrow L$	$H \rightsquigarrow L$	$H \rightsquigarrow H$	Total Flows
PointsToGraph	7%	7%	0%	34%	23%	29%	289
DFT	10%	3%	0%	27%	32%	28%	1672
Coefficients	9%	2%	1%	48%	19%	21%	410
Froggy	5%	4%	0%	44%	42%	5%	272
Puzzle	24%	7%	1%	46%	13%	9%	246
MersennePrime	6%	0%	0%	78%	11%	5%	436

Table 7.9: *Points-To Library Unsafe without Recursion*

Points-To Library Unsafe with Recursion

Table 7.11 and Table 7.12 give the results for this case. As expected, the percentage for explicit flows $L \hookrightarrow L$ drops in all cases compared to the previous case with points-to library unsafe and not accounting for recursion since high data also enters the program at a recursive method call. The results in this case have a variety of flows but they highlight an important point when compared with the results of the first analysis considering library unsafe and accounting for recursion. Two benchmarks Coefficients and Puzzle which have a lot of object instances have an increase in the percentage of explicit flows $L \hookrightarrow L$. This also highlights the benefit of tracking data for different instances of fields and arrays.

Benchmarks	Warnings Exceptions	Warnings Data to Library
PointsToGraph	0	24
DFT	0	36
Coefficients	0	3
Froggy	0	8
Puzzle	0	15
MersennePrime	0	16

Table 7.10: *Points-To Library Unsafe without Recursion*

Benchmarks	$H \hookrightarrow L$	$H \hookrightarrow H$	$L \hookrightarrow H$	$L \hookrightarrow L$	$H \rightsquigarrow L$	$H \rightsquigarrow H$	Total Flows
PointsToGraph	10%	8%	1%	22%	23%	36%	289
DFT	13%	3%	1%	22%	19%	42%	1672
Coefficients	12%	3%	4%	41%	19%	21%	410
Froggy	10%	6%	1%	29%	44%	10%	272
Puzzle	24%	7%	1%	46%	13%	9%	246
MersennePrime	6%	0%	0%	78%	11%	5%	436

Table 7.11: *Points-To Library Unsafe with Recursion*

7.5 Summary of Results

We present a novel way of quantitatively evaluating information flow in programs. We define metrics which are helpful in ascertaining the kind of data present in programs with respect to security. The only prior quantitative analysis for information flow on Java Bytecode by Genaim and Spoto [GS05] only measured the time it took for their analysis to examine the benchmarks.

In our study the numbers counted for the different kinds of information flows in the

Benchmarks	Warnings Exceptions	Warnings Data to Library
PointsToGraph	0	24
DFT	0	40
Coefficients	0	4
Froggy	0	17
Puzzle	0	15
MersennePrime	0	16

Table 7.12: *Points-To Library Unsafe with Recursion*

benchmark programs indicate that high data is present in many statements of programs. Points-to analysis information does impact the kind of data that is present in programs. The results suggest that tracking data with respect to different instances of objects does reduce high data in benchmarks with lots of array and object instances.

Chapter 8
Conclusions and Future Work

8.1 Conclusions

This thesis presented the ground work for a context-sensitive inter-procedural analysis for information flow on JIMPLE intermediate representation (IR) of Java Bytecode. We cover all of the single-threaded Java language at the level of the JIMPLE IR and give rules to analyze every statement. Algorithm design to implement the respective rules and choice of data structures is described. We gave two analyses: the first one considers all instances of classes' fields as the same in the analysis whereas the second analysis differentiates between the instance fields that belong to different instances of a class using the points-to information provided by Spark.

The implementation is modular and allows modifications to be easily introduced. The analyses require no programming overhead since no security annotations are required. The starting set of secure variables or the choice of secure data can be easily changed.

The thesis also gave an operational semantics and type system for MINI-JIMPLE and a type preservation proof for the type system. This work can be extended to formulate type preserving compilation for Java into JIMPLE intermediate representation. This is an important step in developing more formal but still practical models for information flow.

The primary aim of this work was to investigate the kind of information that can be obtained by practical, state-of-the-art techniques in program analysis. We measure success by counting the different kinds of possible information flows in a program. Using points-to

information improves information flow results, reducing the effect of conservative approximations necessary in a practical design.

We have demonstrated that information flow analysis on JIMPLE intermediate representation can be refined and further research can be based on it using more features available in the Soot framework. Similarly the subset of JIMPLE that we formalized can be extended to include more constructs in the JIMPLE language.

8.2 Future Work

This thesis presented two ideas and both can be improved upon in various ways. We suggest some possible avenues to take in the next two sections: first for the information flow analysis on JIMPLE and then for formalizing JIMPLE.

8.2.1 Information flow analysis

The current analysis includes rules to analyze all statements in the Jimple IR. However, several improvements are possible and the way a statement is examined and the method for marking secure data can be refined.

Context-sensitive points-to analysis

Spark computes context-insensitive points-to analysis. More accurate points-to analysis can be used in our analyses to see how it affects the information flows. The Paddle [Lho06] framework provides context-sensitive points-to and call graph analyses for Java. The refined call graph will also improve the information we get for the receiver in a virtual call.

Impact of compiler optimization on information flow counts

In a practical sense a program may have many expressions depending on the stage of compilation. It is common to optimize programs using a variety of complex techniques and it would be interesting to see the effect that some of them have on information flow.

Java Libraries

In the analyses described in this thesis the Java libraries are not directly analyzed and we consider the effects of assuming the library calls to be either safe or unsafe. A deeper investigation of the use of library code, as well as other issues such as use of native methods would be useful.

Programmer help

We have provided a simple mechanism for warning the user of information leaks. A direct Java to JIMPLE translation [Lho05] is now available in Soot which keeps the original variable names in Java for variables in JIMPLE. This can be used to give user-friendly warning messages to the user.

Recursion

Recursion is only handled in the first analysis in a very crude manner. Analyzing recursion in the analysis would require an inter-procedural fixed point to be computed. This is not a conceptually complicated notion, but was not implemented in this thesis due to technical complexity.

8.2.2 Formalizing Jimple

The subset of the JIMPLE IR we have chosen to formalize consists of the very basic statements in JIMPLE. Extension to all the object-oriented features of Java is highly desirable. A fully developed and formally proven type system for JIMPLE would enable further investigation from the perspective of security type systems.

Appendix A
User guide

- The benchmarks used to test the analyses can be found at: http://www.cs.mcgill.ca/~cs621/621benchmarks-2002.jar
- The Soot Framework code and documentation is available at: http://www.sable.mcgill.ca/soot/
- The analysis package is availabe at: http://www.sable.mcgill.ca/~aahmed12/analysis.tar. Download the analysis code and untar it in a folder. If Soot is setup the following command will run the analysis:

  > java -Xmx400m informationflowanalysis/Main -w -p cg.spark on -f jimple -inf sim -main-class <mainClass> -process-dir <directory name>

  There is one added option "-inf" to run the information flow analysis:

  - "-inf sim" runs the class based analysis.
  - "-inf spa" runs the instance object based analysis.

- The machine on which the benchmarks were run had a dual processor. The complete specifications are as follows:

  > cat /proc/cpuinfo

processor : 0
model name : AMD Athlon(tm) 64 X2 Dual Core Processor 3800+
cpu MHz : 2010.314
cache size : 512 KB
processor : 1
model name : AMD Athlon(tm) 64 X2 Dual Core Processor 3800+
cpu MHz : 2010.314
cache size : 512 KB

- The memory in the system was:

  > grep MemTotal /proc/meminfo

  MemTotal: 4046572 kB

Bibliography

[ABB06] Torben Amtoft, Sruthi Bandhakavi, and Anindya Banerjee. A logic for information flow in object-oriented programs. In *POPL '06: Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, Charleston, South Carolina, USA, 2006, pages 91–102. ACM Press, New York, NY, USA.

[ABF03] Marco Avvenuti, Cinzia Bernardeschi, and Nicoletta De Francesco. Java bytecode verification for secure information flow. *SIGPLAN Not.*, 38(12):20–27, 2003.

[BL75] D. Bell and L. LaPadula. Secure computer systems: Mathematical foundations and model. Technical report, Technical Report M74-244, Mitre Corporation, Belford, MA, 1975.

[BN02] Anindya Banerjee and David A. Naumann. Secure information flow and pointer confinement in a java-like language. In *Proceedings of the 15th IEEE Computer Security Foundations Workshop (CSFW'02)*, 2002, page 253. IEEE Computer Society, Washington, DC, USA.

[BN03] Anindya Banerjee and David A. Naumann. Using access control for secure information flow in a java-like language. In *Proc. 16th IEEE Computer Security Foundations Workshop*, 2003, pages 155–169.

[BRN06] Gilles Barthe, Tamara Rezk, and David Naumann. Deriving an information flow checker and certifying compiler for java. In *SP '06: Proceedings of the 2006 IEEE Symposium on Security and Privacy (S&P'06)*, 2006, pages 230–242. IEEE Computer Society, Washington, DC, USA.

[CT05] Juan Chen and David Tarditi. A simple typed intermediate language for object-oriented languages. In *POPL '05: Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, Long Beach, California, USA, 2005, pages 38–49. ACM Press, New York, NY, USA.

[DD77] Dorothy E. Denning and Peter J. Denning. Certification of programs for secure information flow. *Commun. ACM*, 20(7):504–513, 1977.

[Den76] Dorothy E. Denning. A lattice model of secure information flow. *Commun. ACM*, 19(5):236–243, 1976.

[DGC95] Jeffrey Dean, David Grove, and Craig Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In *ECOOP '95: Proceedings of the 9th European Conference on Object-Oriented Programming*, 1995, pages 77–101. Springer-Verlag, London, UK.

[Ecl] Eclipse. Platform Technical Overview, Technical Report, Object Technology International, 2003. `http://www.eclipse.org/`.

[Fen74] J. S. Fenton. Memoryless subsystems. *Computing Journal, 17(2):143–147*, May 1974.

[GHM00] Etienne Gagnon, Laurie J. Hendren, and Guillaume Marceau. Efficient inference of static types for java bytecode. In *SAS '00: Proceedings of the 7th International Symposium on Static Analysis*, 2000, pages 199–219. Springer-Verlag, London, UK.

[GM82] J. A. Goguen and J. Meseguer. Security policies and security models. In *Proc. IEEE Symp. on Security and Privacy*, 1982, pages 11–20.

[GS05] Samir Genaim and Fausto Spoto. Information flow analysis for java bytecode. In *Proc. of the Sixth International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI'05)*, 2005.

[HR98] Nevin Heintze and Jon G. Riecke. The slam calculus: programming with secrecy and integrity. In *POPL '98: Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, San Diego, California, United States, 1998, pages 365–377. ACM Press, New York, NY, USA.

[HS06] Sebastian Hunt and David Sands. On flow-sensitive security types. In *POPL '06: Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, Charleston, South Carolina, USA, 2006, pages 79–90. ACM Press, New York, NY, USA.

[IPW01] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight java: a minimal core calculus for java and gj. *ACM Trans. Program. Lang. Syst.*, 23(3):396–450, 2001.

[JL75] Anita K. Jones and Richard J. Lipton. The enforcement of security policies for computation. In *SOSP '75: Proceedings of the fifth ACM symposium on Operating systems principles*, Austin, Texas, United States, 1975, pages 197–206. ACM Press, New York, NY, USA.

[LH03] Ondřej Lhoták and Laurie Hendren. Scaling Java points-to analysis using Spark. In G. Hedin, editor, *Compiler Construction, 12th International Conference*, April 2003, volume 2622 of *LNCS*, pages 153–169. Springer, Warsaw, Poland.

[Lho02] Ondřej Lhoták. Spark: A flexible points-to analysis framework for Java. Master's thesis, McGill University, December 2002.

[Lho05] Jennifer Lhoták. Visualization tools for optimizing compilers. Master's thesis, McGill University, August 2005.

[Lho06] Ondřej Lhoták. *Program Analysis using Binary Decision Diagrams*. PhD thesis, McGill University, Jan 2006.

[LST99] Christopher League, Zhong Shao, and Valery Trifonov. Representing java classes in a typed intermediate language. In *ICFP '99: Proceedings of the fourth ACM SIGPLAN international conference on Functional programming*, Paris, France, 1999, pages 183–196. ACM Press, New York, NY, USA.

[LST02] Christopher League, Zhong Shao, and Valery Trifonov. Type-preserving compilation of featherweight java. *ACM Trans. Program. Lang. Syst.*, 24(2):112–152, 2002.

[MCG+99] Greg Morrisett, Karl Crary, Neal Glew, Dan Grossman, Richard Samuels, Frederick Smith, David Walker, Stephanie Weirich, and Steve Zdancewic. Talx86: A realistic typed assembly language. In *the 1999 ACM SIGPLAN Workshop on Compiler Support for System Software*, May 1999, pages 25–35.

[MI04] Kenji Miyamoto and Atsushi Igarashi. A modal foundation for secure information flow. In *Proceedings of Workshop on Foundations of Computer Security (FCS'04)*, July 2004, pages 187–203.

[Mil76] Jonathan K. Millen. Security kernel validation in practice. *Communications of the ACM*, 19(5):243–250, 1976.

[ML98] Andrew C. Myers and Barbara Liskov. Complete, safe information flow with decentralized labels. In *14'th IEEE Symp. Security and Privacy*, 1998, pages 186–197.

[Muc97] Steven S. Muchnick. *Advanced compiler design and implementation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1997.

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

[O'C99] Robert O'Callahan. A simple, comprehensive type system for java bytecode subroutines. In *POPL '99: Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, San Antonio, Texas, United States, 1999, pages 70–78. ACM Press, New York, NY, USA.

[Pie05] Benjamin C. Pierce, editor. *Advanced Topics in Types and Programming Languages*. The MIT Press, 2005.

[PS02] François Pottier and Vincent Simonet. Information flow inference for ml. In *POPL '02: Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, Portland, Oregon, 2002, pages 319–330. ACM Press, New York, NY, USA.

[PV04] Christopher J. F. Pickett and Clark Verbrugge. Compiler analyses for improved return value prediction. Technical Report SABLE-TR-2004-6, Sable Research Group, McGill University, Oct. 2004.

[SLM98] Zhong Shao, Christopher League, and Stefan Monnier. Implementing typed intermediate languages. In *ICFP '98: Proceedings of the third ACM SIGPLAN international conference on Functional programming*, Baltimore, Maryland, United States, 1998, pages 313–323. ACM Press, New York, NY, USA.

[SM03] A. Sabelfeld and A. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications, 21(1)*, 2003.

[Soo] Soot. a Java Optimization Framework. `http://www.sable.mcgill.ca/soot/`.

[UKY06] Hiroshi Unno, Naoki Kobayashi, and Akinori Yonezawa. Combining type-based analysis and model checking for finding counterexamples against non-interference. In *PLAS '06: Proceedings of the 2006 workshop on Programming languages and analysis for security*, Ottawa, Ontario, Canada, 2006, pages 17–26. ACM Press, New York, NY, USA.

[VR00] Raja Vallée-Rai. Soot: A Java bytecode optimization framework. Master's thesis, McGill University, October 2000.

[VRGH+00] Raja Vallée-Rai, Etienne Gagnon, Laurie J. Hendren, Patrick Lam, Patrice Pominville, and Vijay Sundaresan. Optimizing Java bytecode using the Soot framework: Is it feasible? In *Proceedings of the International Conference on Compiler Construction*, 2000, pages 18–34.

[VSI96] Dennis Volpano, Geoffrey Smith, and Cynthia Irvine. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(3):167–187, 1996.

[WSO+75] K. G. Walter, S. I. Schaen, W. F. Ogden, W. C. Rounds, D. G. Shumway, D. D. Schaeffer, K. J. Biba, F. T. Bradshaw, S. R. Ames, and J. M. Gilligan. Structured specification of a security kernel. In *Proceedings of the international conference on Reliable software*, Los Angeles, California, 1975, pages 285–293. ACM Press, New York, NY, USA.

[Zda04] Steve Zdancewic. Challenges for information-flow security. In *Proceedings of the 1st International Workshop on the Programming Language Interference and Dependence (PLID'04)*, 2004.