UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

DENYS HUPEL DA SILVA OLIVEIRA

Engineering Java Byte Code to Detect Exception Information Flow

Bachelor Thesis

Dr. Raul Fernando Weber
Advisor

Porto Alegre, July 2010

CIP – CATALOGING-IN-PUBLICATION

Oliveira, Denys Hupel da Silva

Engineering Java Byte Code to Detect Exception Information Flow / Denys Hupel da Silva Oliveira. – Porto Alegre: PPGC da UFRGS, 2010.

41 f.: il.

Final Report (Bachelor) – Universidade Federal do Rio Grande do Sul. Curso de Bacharelado em Ciência da Computação, Porto Alegre, BR–RS, 2010. Advisor: Raul Fernando Weber.

1. Information flow. 2. Java byte code. 3. Exception flow. 4. Static analysis. 5. Usage control. I. Weber, Raul Fernando. II. Título.

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
Reitor: Prof. Carlos Alexandre Netto
Vice-Reitor: Prof. Rui Vicente Oppermann
Pró-Reitora de Graduação: Profa. Valquiria Link Bassani
Diretor do Instituto de Informática: Prof. Flávio Rech Wagner
Coordenador do CIC: Prof. João César Netto
Bibliotecária-Chefe do Instituto de Informática: Beatriz Regina Bastos Haro

“The wisest mind has something yet to learn.”

— George Santayana

ACKNOWLEDGEMENTS

I would like to thank my family, who was always there for me, no matter good or bad times, always giving support and important advices.

Many thanks to my friends(old ones and some met at the university), who gave me great moments during my graduation, helped me when necessary, and also understood times that I was very busy with university stuff.

And I would also like to thank Raul Weber, who was my advisor in this thesis, and helped me whenever necessary.

CONTENTS

LIST OF ABBREVIATIONS AND ACRONYMS 6
LIST OF FIGURES 7
LIST OF TABLES 8
ABSTRACT 9
RESUMO 10
1 INTRODUCTION 13
2 BACKGROUND 14
2.1 Java exceptions 14
2.1.1 Checked exceptions 14
2.1.2 Unchecked exceptions 14
2.1.3 Exception handling 14
2.2 Information flow 16
2.2.1 Explicit flow 17
2.2.2 Implicit flow 17
2.2.3 Main and exception flow 18
2.3 Static/Runtime analysis 18
2.4 Jimple code 20
2.5 Code instrumentation 21
3 SOLUTION AND IMPLEMENTATION 22
3.1 Solution 25
3.1.1 Try blocks 26
3.1.2 Methods propagating exceptions to caller 27
3.2 Problem with post-dominators 30
4 RESULTS 32
5 CONCLUSION AND FUTURE WORK 39
REFERENCES 40

LIST OF ABBREVIATIONS AND ACRONYMS

JVM	Java Virtual Machine
CFG	Control Flow Graph

LIST OF FIGURES

Figure 2.1: Interprocedural control flow in Java exception-handling constructs.(SINHA, 2000) 16
Figure 2.2: Control-flow graph for explicit flow(listing 2.2) 17
Figure 2.3: Control-flow graph for implicit flow(listing 2.3) 19
Figure 2.4: Control-flow graph for exception flow(listing 2.4) 20

Figure 3.1: Post-dominator graph for explicit flow example 23
Figure 3.2: Control-flow graph with two branch points 24
Figure 3.3: Simplified CFG for figure 3.2 24
Figure 3.4: Post-dominator graph for figure 3.3 25
Figure 3.5: Control flow with exceptional and unexceptional edges 26

LIST OF TABLES

Table 2.1: Java exception table for listing 2.1 16

ABSTRACT

Data is increasingly added to computer systems every day. Much information the user inputs in programs or websites, are sensitive for the user, and protection of that data, giving access just to whom may concern, is a important security topic. Information flow analysis is a way to track the flow of variables in a system, and find points where information might be exposed somehow. Information can flow explicitly or implicitly in a program. (WILLENBROCK, 2009) proposes a implicit information flow analyzer, without regards to exception flow. In this thesis, a static analysis,i. e. , before runtime, is proposed to find out al points where information can be unvealed because of the ocasion(or not) of a checked exception in a Java program.

Keywords: Information flow, java byte code, exception flow, static analysis, usage control.

Detecção do fluxo de informações no contexto de exceções através da engenharia de byte code Java

RESUMO

Segurança de dados é um tópico muito importante em ciência da computação. Sistemas e sites da Internet retém cada dia mais informações pessoais sobre seus usuários. Muitos dos dados inseridos pelos usuários são confidenciais, e, ao inserí-los em um determinado sistema, eles esperam que estas informações estejam seguras, ou seja, algum mecanismo de controle de acesso deve ser aplicado nos dados. Para ilustrar a situação, considere o cenário do sistema de um hospital: o segurança não deve poder acessar nenhuma informação sobre os pacientes no sistema; o recepcionista, por outro lado, poderia ter acesso a informações básicas do paciente, como nome, endereço, data da última consulta. Enfim, dados sobre o histórico médico do paciente, só podem ser acessados por um médico.

Esse tipo de controle é muito importante para a segurança de informações, e uma maneira de recuperar dados confidenciais sem permissão, é através da análise do fluxo de informações no programa. Uma vez inserido ou carregado em um sistema, um valor sensível fica armazenado em uma variável, e essa variável deve ser protegida para que somente usuários com permissão de acesso à mesma, consigam obter seu valor. Entretanto, um valor não fica retido apenas na variável a qual foi atribuído, mas é repassado integralmente ou parcialmente para outras variáveis durante o ciclo de vida do programa. Um indivíduo com intenções de acessar informações confidenciais, pode então analisar o fluxo de informações do sistema, e descobrir o conteúdo de uma variável protegida, através da análise do conteúdo de uma variável não protegida.

Para evitar esse tipo de ataque, uma solução é analisar o fluxo de informações em um programa, e marcar todas as instruções e variáveis que de alguma forma são influenciadas por um dado confidencial, para que sejam também protegidas de maneira que não haja vazamentos de dados. Informações podem fluir explicita ou implicitamente em um programa.

O fluxo explícito de informações é definido por instruções de atribuição no código. Por exemplo, na instrução $a{=}b$, o valor de b é explicitamente atribuído na variável a.

No caso do fluxo implícito de informações, valores vazam não por serem diretamente atribuídos a outras variáveis, mas são inferidos após a análise de qual caminho o programa seguiu após um determinado ponto de ramificação. Ilustrando o exemplo, após testar se o valor de uma variável sensível é maior ou menor que 1000, o programa irá tomar um dos dois caminhamentos possíveis. Se em um dos caminhamentos uma flag(não sensível) é setada como verdadeira, e no outro caminhamento ela é definida como falsa, pode-se descobrir se o valor confidencial é maior ou menor que 1000, apenas checando se a flag terminou a execução como verdadeira ou falsa. Da mesma forma, informações podem ser adquiridas analizando se exceções ocorreram ou não em determinados pontos do programa.

O objetivo deste trabalho então é fazer uma análise estática do byte code de programas Java, definindo pontos onde informações podem vazar devido à ocorrência ou não de exceções. Análises estáticas são caracterizadas por ocorrerem antes da execução do código, e a análise proposta insere código instrumentado no programa analisado, de forma que uma futura análise em tempo de execução tenha informações sobre os trechos de código onde exceções podem ocorrer, e possa então fazer a propagação das marcas de variáveis sensíveis.

Stefan Willenbrock, em seu trabalho de graduação(WILLENBROCK, 2009), realizou um framework capaz de detectar fluxo implícito(e explícito) de informações estaticamente, e propagar marcas em tempo de execução, a partir das informações coletadas antes. Porém ele não considerou a ocorrência de exceções nos programas analisados, o que é muito importante, pois a grande maioria dos programas Java tem ibfluência no seu fluxo devido à ocorrência(ou não) de exceções durante sua execução. Este trabalho difere-se do trabalho de Stefan, pois considera o contexto de exceções, e não faz a propagação de marcas a partir das influências definidas na análise estática, e sim sinaliza os pontos onde a propagação de marcas deve iniciar e terminar, devido à possibilidade de um fluxo excepcional de informações.

Exceções em Java podem ser checadas ou não-checadas. As exceções não-checadas são aquelas que podem ocorrer em qualquer ponto do programa, e não precisam ser explicitamente lançadas ou capturadas. Exceções checadas, por outro lado, precisam ser explicitamente lançadas com o uso da instrução *throw*, e têm que ser capturadas por um block *catch* em algum ponto do programa. Este último tipo citado, é o tipo de exceções tratado neste trabalho.

Considerando então o escopo de exceções checadas, elas podem ocorrer dentro de um bloco *try*, ou dentro de um método capaz de propagar exceções para o método que o chamou. Neste contexto então, diz-se que o fluxo de informações a partir de instruções dentro de um bloco *try* ou dentro de um método capaz de propagar exceções pode ter dois possívei destinos: o fluxo excepcional, que é o bloco *catch* ou a propagação pro método chamador; e o fluxo normal, que é a execução da próxima instrução do programa.

Trechos de código onde exceções podem ocorrer, podem ser considerados trechos com fluxo de informações implícitas, pois são instruções que lançam ou não exceções, ou seja, são pontos de ramificação no código, podendo ter 2 destinos: excepcional ou não. Devido a este motivo, a análise feita neste trabalho lembra a análise feita para detectar fluxo implícito de informações, porém difere no fato que o ponto de ramificação e o ponto de junção de uma ramificação têm que ser descobertos, e então sinalizados de alguma forma.

À primeira vista, para sinalizar trechos de código onde existe fluxo de informações devido a ocorrência de exceções, poderiamos marcar como ponto de início a entrada de um bloco *try* ou o começo de um método com propagação de exceções; e como ponto final o fim do último block *catch* correspondente ao *try*, ou o final do método. Entretanto, este método não funciona, pois instruções que ocorrem antes de um bloco *try*, podem ser afetadas pelo mesmo. Por exemplo, se um bloco *try* está dentro de um bloco *if*, informações do bloco *else* podem ser descobertas devido à ocorrência de uma exceção no outro ramo do fluxo(bloco *if*). Outro fator importante a ser levado em conta, é que nem todas instruções dentro de um bloco try são influenciadas pelo lançamento de uma exceção, e deseja-se então fazer uma análise de forma a delimitar somente os trechos de código realmente afetados pelo fluxo excepcional, ou seja, somente as instruções que podem vazar alguma informação sensível devido à uma exceção lançada.

Ilustrando como foi feita a implementação do programa, pode-se ver abaixo um algo-

```
1 Encontrar a instrução throw
2 Detectar o escopo (bloco try, ou método capaz de propagar
    exceções)
3 Encontrar instrução de início das influências por exceções
    através de uma análise das instruções anteriores
4 Encontrar instrução de fim das influências por exceções
5 Instrumentar código, delimitando os pontos de início e fim
    de propagação das influências
```

Listing 1: Algoritmo super simplificado

ritmo super simplificado da solução.

Para definir os pontos de início e fim de influências, foi utilizado o conceito de pós dominante imediato de uma instrução. Uma instrução x pós dominante imediata de y, é aquela instrução que sempre ocorre após x, ou seja, se x é um ponto de ramificação, a instrução que se pode ter certeza que sempre ocorrerá após x é o ponto de junção da ramificação. Dessa forma, foram encontrados os pontos de junção de caminhos excepcionais e não excepcionais, portanto, o ponto onde influências devem parar a propagação.

Para concluir, alguns resultados de código instrumentado após a análise são apresentados no caítulo de resultados, e sugere-se ainda que este trabalho seja extendido para uma análise considerando também a ocorrência de exceções não checadas. O grande problema dessas exceções é que elas podem ocorrer em grande parte do programa, e a primeira vista, a grande maioria das instruções seriam afetadas por essas exceções, espalhando as influências de exceções por toda a análise. Portanto uma idéia inicial seria considerar primeiramente apenas algumas exceções não checadas, e analisar os resultados.

Palavras-chave: Fluxo de informações, java byte code, fluxo de exceções, análise estática, controle de uso.

1 INTRODUCTION

Data security is a very important topic in Computer Science. People is increasingly putting personal data on websites, companies have confidential data of their employees, hospitals have medical records of their patients, and so on. These information can be used to somehow harm someone, as these data are supposed to be confidential. To protect sensitive data, a control of who can access which information should be done, and this is used with access control. For instance, in a hospital, a security man should have no access to a patient's record, a receptionist could get information about dates of patient's last medical consultations, and finally the doctor would have access to the medical history of the person.

Whenever someone not supposed to acquire information about someone ends with that data in hands, it is said that there was a leak of information. An attacker is trying to obtain information of a system would study the system to identify leak points. First thing to do to protect confidential data that enters in a system, is specify which data is sensitive, which are not, and put some protection mechanism in every points using that values in the system. An attacker would then analyze the program and try to get the data from sensitive variable, but will not succeed, right? No. During the runtime of a program, classified values flow to many destinations. Consequently, to really protect given data, it is essential to analyze the flow of information in a program, and taint all variables that could have some information from a sensitive input.

The purpose of this thesis is to observe the information flow in Java byte code, and point out code sections where information could leak as a consequence of an exception being raised or not.

The rest of the thesis structure is divided as follows:

- Chapter 2, Background, gives some general specifications of Java exceptions, information flow, and types of analysis that can be done.
- Chapter 3, Solution and Implementation, describes what is done to implement a framework capable of detecting exception information flow.
- Chapter 4, Results, shows how code looks like after running the analysis.
- Chapter 5, Conclusion and Future Work, presents the final conclusions and possibilities extensions with future work.

2 BACKGROUND

This chapter gives brief descriptions of some concepts used along this thesis, so the user gets some background about the thema. It gives an overview of how Java deals with exceptions, how information can flow during execution of programs, and ways to analyze the flow of information.

2.1 Java exceptions

Exceptions are a way to indicate that something not normal happened during the program execution, or, ”when a program violates the semantic constraints of the Java programming language, the Java virtual machine signals this error to the program as an exception” (GOSLING, 2005,p.297).

When an exception condition is achieved during a method execution, an exception object (exceptions are represented as objects in Java) is thrown, and a handler should catch this object, in order to treat the problem. The handler can be on the method body itself, or inside one of the method callers. After the exception is thrown, each method will check if a handler exists, and if not, it throws the exception object to the next method in the method invocation stack, looking for someone that could deal with the exception. If no handler is found, the object is caught by the Java runtime environment.

Exceptions are subclasses of Throwable, and are divided in two groups: checked and unchecked.

2.1.1 Checked exceptions

Checked exceptions are all exceptions that are explicitly thrown and need to have a handler declared somewhere. For these exceptions, the Java compiler checks for handlers in the program during compile time.

2.1.2 Unchecked exceptions

The exceptions that are not necessarily thrown in the code, and can have no handler, are runtime exceptions, and are called unchecked exceptions.

2.1.3 Exception handling

Whenever a checked exception is thrown during runtime, it has to be caught and treated.

To define handlers for an exception, it is necessary to enclose in a *try block*, all instructions that can throw an object, or specify a *throws* statement with the exception type in the method definition, so this object will try to find a handler inside the caller method.

```
1  public void methodWithExceptions(int val) throws Exception{
2    try{
3      if ( val > 10 ){
4        throw new MyException();
5      }else if ( val > 0 ){
6        throw new MyOtherException();
7      }else{
8        throw new Exception();
9      }
10   }catch( MyException me ){
11     doSomething();
12   }catch( MyOtherException moe ){
13     doAnotherThing();
14   }finally{
15     //always executed
16     doNecessaryThings();
17   }
18   System.out.println(''End of method'');
19 }
```

Listing 2.1: Method throwing exceptions

A *try block* is followed by one or more *catch blocks*, a *finally block*, or both. Catch blocks are handlers that deal with the exception object itself, so they are executed just in cases where the program raises an exception. Each handler has a type, and gets all exceptions that match its exception type or subtypes. When an exception is raised, all subsequent instructions inside the try block are aborted,and the flow is redirected to the first handler that matches the exception type(the other catch blocks will not be executed then, even if there is one that handles the same exception type), so it is important to write first the most specific handlers, and then the more comprehensive ones. Finally blocks are pieces of code that are always executed, no matter if an exception is thrown or not. That means, if a try block finishes normally, flow goes to finally block, and if the flow goes to a handler, after the handler the flow goes also to finally.

Listing 2.1 shows an example where once a value entered the function, flow can go to different points. If *val>10*, control flow goes to the first handler(line 10), and later executes the finally block(line 14). When $0 < val <= 10$, the second catch block is executed(line 12), and then the finally block. To finish, in case $val <= 0$, flow goes to finally block, and later is redirected to the caller method, throwing an exception of type *java.lang.Exception*.

Figure 2.1 exemplifies all possibilities of control flow(considering the possibility of exceptions) inside and between methods.

Also, the JVM creates an exception table for every handler, i. e., catch block inside the code, and it works as follows: for each *catch* clause of each *try block*, there is a line in the exception table describing the line where try block starts(from), the line where it finishes(to), then the line where catch block starts(target), and last, the type of exception this handler deals with(Type). Table 2.1 shows the exception table for listing 2.1.

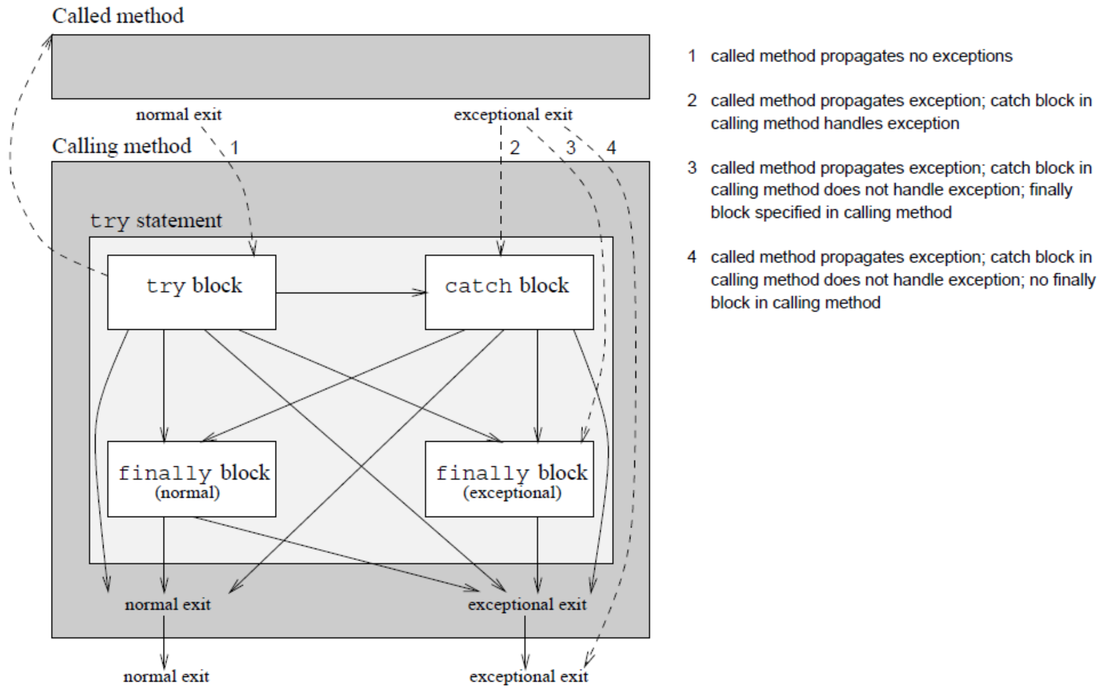

Figure 2.1: Interprocedural control flow in Java exception-handling constructs.(SINHA, 2000)

Table 2.1: Java exception table for listing 2.1

From	To	Target	Type
3	9	10	java.lang.MyException
3	9	12	java.lang.MyOtherException

2.2 Information flow

Information flow is related to what happens to information once they are entered in a program. During runtime, values inputed in the program are reassigned to other variables. When this happens, we say that information is flowing from one variable(e.g. input) to another, and the latter variable now has information from the former one.

So, once some information is inputed in the program(from now on, program input will be referred as *source*), it can go to many different destinations, i.e. several variables can have its value, and during that, this data could pass through insecure channels, where some third party one can monitor information. These insecure channels are called *sinks*.

In practice there are situations where sensitive data enter the program through sources, and that data should not be unveiled anywhere, i.e. the data cannot pass through sinks, or, if it does, it should be protected anyhow. A daily example of sensitive source is a *password*, which the user enters in a website everyday, trusting that value is secure inside the website and no one will retrieve it. A way to give this reliability to the user is to make an information flow analysis, tracking all flows possible to sensitive data, and later ensure confidentiality to all affected variables with criptografy, for instance.

In respect to information flow analysis, there are two different ways data can flow from one variable to another, and they will be discussed next: explicit and implicit flow.

```
1 public static void main(String[] args){
2   String name = args[0];
3   int salary = Integer.valueOf(args[1]);
4   int bonus = Integer.valueOf(args[2]);
5
6   int earnings = salary + bonus;
7   System.out.println(name+"␣got␣"+earnings+"this␣month.");
8 }
```

Listing 2.2: Code with explicit flow

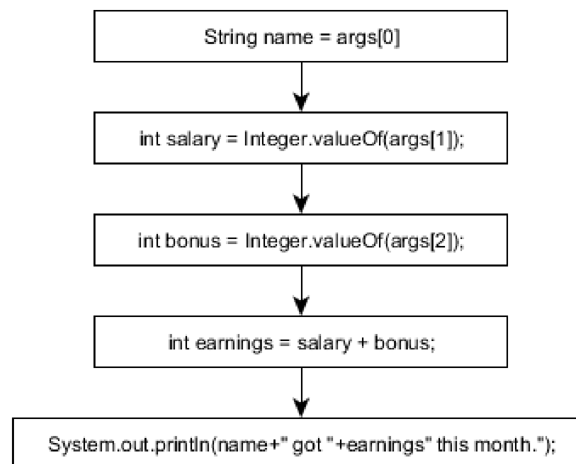

Figure 2.2: Control-flow graph for explicit flow(listing 2.2)

2.2.1 Explicit flow

We say that there is an explicit flow of information from variable *a* to *b* when *a* appears in the right hand side of an assignment to *b*. Listing 2.2 shows this clearly: there is a method *main* that receives as parameter a list of arguments *args*; then, inside the method body, at first 3 variables are declared: *name*, *salary*, and *bonus*; these variables receive the values from *args*, in form of assignment. So, information is flowing from *args* to these variables, e.g. the content of *args[1]* explicit flows to new variable *salary*, i.e *salary* now has the value of *args[1]*. If we take a look in line 6, data from *salary* and *bonus* are explicit flowing to *earnings*. The control flow happening in listing 2.2, is representerd as a control flow graph(CFG) in figure 2.2.

2.2.2 Implicit flow

On the other hand is the implicit flow, where data is not explicit flowing from variable *a* to *b*, but one can guess the content of *a* by looking at *b*. Checking out listing 2.3, let's say some attacker wants to discover the salary of someone, but variables *salary* and *args* are somehow protected(e.g. encrypted). Taking closer atention to the code, if the attacker checks for the value of *payTaxes*, he then finds out part of the data(sometimes the entire data) that lies inside *salary*. In this case it is said information implicit flows from *salary* to *payTaxes*.

Figure 2.3 shows the CFG of listing 2.3, and we can see more clearly how control

```
1  public static void main(String[] args){
2    String name = args[0];
3    int salary = Integer.valueOf(args[1]);
4    boolean payTaxes = false;
5
6    if ( salary > 2000 ){
7      payTaxes = true;
8    }else{
9      payTaxes = false;
10   }
11
12 }
```

Listing 2.3: Code with implicit flow

flow branches at *if* instruction and later joins back together at *return*. Everytime we have branches like this, one is actually executed during runtime and is called *executed branch*, whereas the other stays idle and is named *non-executed branch*. Someone trying to obtain information from the program, would choose one of the branches to monitor and check wheter its instructions were executed or not. We can conclude that no matter which branch is taken, information can be acquired, and to prevent this, both branches should be considered while running an information flow analysis.

2.2.3 Main and exception flow

During runtime, a program usually has some expected paths to follow, e.g. when a user enters a username and password, the expected flows are: go to next page logged with the user data, or go back to login page because username or password were incorrect. This flow, which is the sequence of instructions that would normally be executed one after another, is referred as the *main flow*. Sometimes though, different paths are triggered after an unexpected behavior or error in runtime; these paths are called *exception flow*.

Exception flow is included in the context of implicit flow, since depending of the occurrence of an exception or not, flow can take two different branches, and an attacker can discover secret data just by analyzing which branch was taken.

Listing 2.4 shows an example of an exception flow. In this example, *secret* is sensitive information, and to guess its value, it is only necessary to check which flow the program took during runtime. If we take a look at, there is a method that throws an exception whenever *secret* is *false*, so one could guess that if code in the catch block was executed, that means an exception happened, and *secret==false*. Otherwise, if line 6 was executed, exception was not raised, and *secret==true*.

CFG in figure 2.4 shows that after *methodThatThrowsExceptionWhenSecretIsFalse* there are 2 possible branches, one taking the main flow, and one following the exceptional edge; later flow is joined back at *return* instruction.

2.3 Static/Runtime analysis

The analysis of information flow can be done in two different phases: before, and during program execution. The former approach is called *static analysis*, the latter, *runtime*

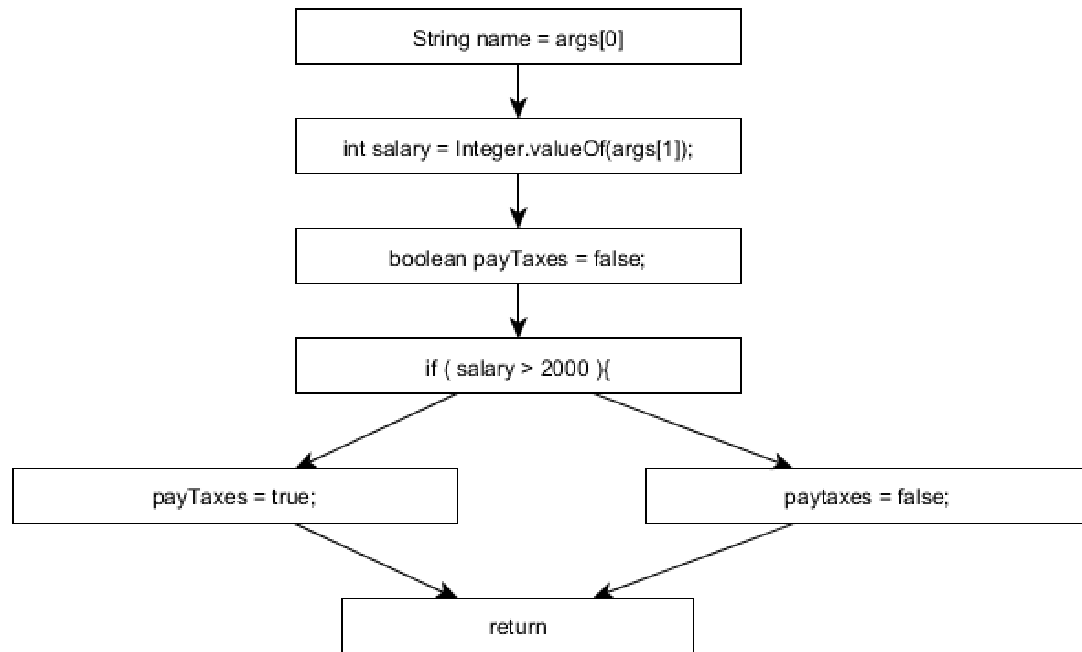

Figure 2.3: Control-flow graph for implicit flow(listing 2.3)

```
1  public static void main(String[] args){
2    boolean secret = false;
3
4    try{
5      methodThatThrowsExceptionWhenSecretIsFalse(secret);
6      System.out.println(''Everything fine.'');
7    }catch ( Exception e ){
8      System.out.println(''Exception catched.'');
9    }
10 }
```

Listing 2.4: Code with exception flow

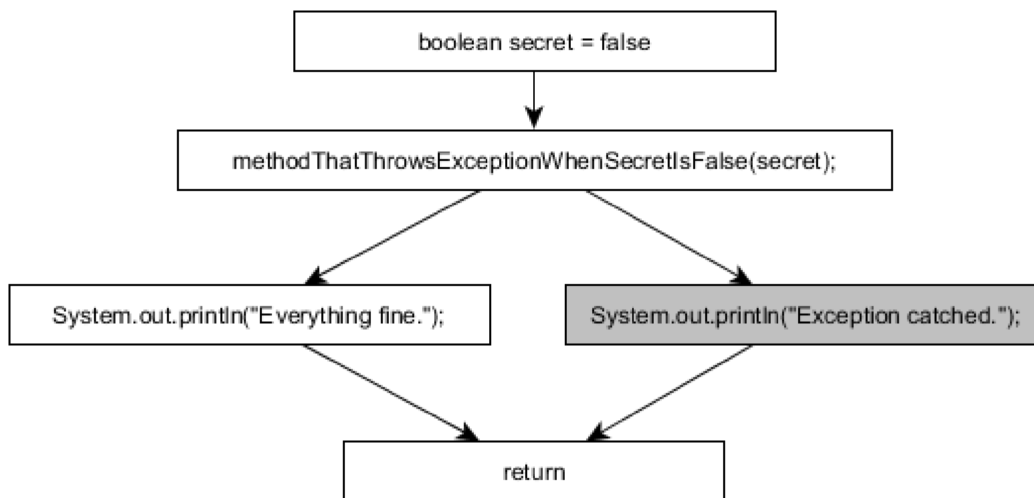

Figure 2.4: Control-flow graph for exception flow(listing 2.4)

```
1 public static void HelloWorld(String[] args){
2   System.out.println("Hello␣World!");
3 }
```

Listing 2.5: HelloWorld example in Java

analysis.

As static code analysis is done before program runtime, no inputs are given, and instructions are analyzed without being executed. Runtime code analysis, on the other hand, runs during program execution, and analysis results may vary depending on the program sources. It is good to use a static analysis to detect defects in the code(and sinks), and this type of analysis is much faster than the runtime, because computations do not need to be recalculated after each executed instruction in the program. But runtime analysis is good to watch how flow is actually being executed and what happens after each statement, so it gives more precise results, because it is based on actual inputs and also heap status.

2.4 Jimple code

Instead of using pure Java Byte Code, it was opted for using an intermediate representation of Java programs. This representation is part of SOOT framework (SOOT, 2010) and is called Jimple.

Jimple is a representation between Java byte code and Java high level code. It converts stack-based byte code into 3-address code, making it easier for the developer to handle the code and write an analysis, since Jimple has only 15 possible operations(Java byte code has over 200 operations). Below is an example of a Hello World program in Java(Listing 2.5) and in correspondent Jimple code(Listing 2.6).

```
1  public static void HelloWorld(java.lang.String[]){
2    java.lang.String[] r0;
3    java.io.PrintStream $r1;
4
5    r0 := @parameter0: java.lang.String[];
6    $r1 = <java.lang.System: java.io.PrintStream out>;
7    virtualinvoke $r1.<java.io.PrintStream:
8         void println(java.lang.String)>("Hello World!");
9    return;
10 }
```

Listing 2.6: HelloWorld example in Jimple code

2.5 Code instrumentation

To instrument code is to insert instructions inside a system in order to monitor components in the same system. Instrumenting code is useful, for instance , to insert during static analysis, method calls that will be executed during runtime to perform the runtime analysis.

3 SOLUTION AND IMPLEMENTATION

To design a framework capable of detect exceptional flow in the Java byte code, it is important to understand the concepts explained in the background chapter, and look after an approach to achieve this goal. To create the framework proposed, Java byte code is read with SOOT, translated to an intermediate representation (Jimple), then analyzed to define important points, and finally new instrumented code is inserted in these points, in order to signalize them during runtime.

SOOT is a Java optimization framework that provides four intermediate representations for analyzing and transforming Java byte code. These representations are: Baf, Jimple, Shimple and Grimp. In this thesis only Jimple will be considered, so this is the last time we hear about the others.

First, let's talk about Willenbrock's framework, described in his bachelor thesis (WILLENBROCK, 2009). Willenbrock implemented a framework to detect implicit information flow in Java byte code. His work uses static and runtime analysis to detect implicit and explicit information flow in unexceptional contexts. He starts with a static analysis that inspects the call graph created from the method body. With the call graph in hands, every unit(unit is how SOOT represents each node in the CFG) is examined so influences from one unit to another can be detected, i.e., detect information flowing from unit *A* to unit *B*, and maybe later information flowing from *B* to unit *C*; in this case it is said that there was information flow from unit *A* to unit *C*. This influence propagation is based mainly on information from *sources* entered in the method body, as these values are the ones which may possibly bring sensitive information to the method. Units with sensitive information should be called *tainted*, and the objective of propagating influences is to successfully taint all units that may contain some sensitive data.

All these influences are saved into serialized files–so they can be reloaded later– and code is instrumented with the insertion of method calls to methods in the framework responsible for runtime analysis. These code will be executed later during runtime, and will exchange taint along the analyzed code, based on influences saved before and also actual input, that have definitions of tainted sources entering the program.

The main objective of my work is to statically analyze code, and use instrumentation to insert check points delimiting where flow can have normal and exceptional destination, and where code can just flow unexceptionally. With this information, a later work can be done to integrate with Willenbrock's framework, and use taint propagation in both exceptional and main flow.

In this framework, just Java checked exceptions are analyzed, because to handle unchecked exceptions a much more detailed analysis should be done, considering all possible runtime exceptions that could be raised by each instruction, and taint propagation would probably spread very fast all over the program. As said before, checked exceptions

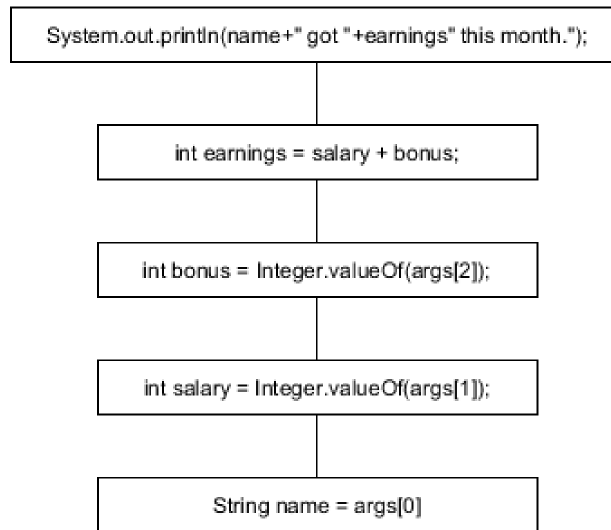

Figure 3.1: Post-dominator graph for explicit flow example

are explicit thrown in Java. They can occur inside a try block with handlers to catch the exception object, and these exceptions can also propagate to caller method, until a handler is found.

Thinking at a first glance, the task of an exceptional analysis may look very simple: in methods that cannot propagate exceptions to caller, put a marker signalizing the entry point of a try block to signalize possible exceptional flow, and another mark at the end of last handler of same try block to signalize the end of possible exceptional flow; and assume that all units inside methods with exception propagation could have an exceptional edge in the CFG, consequently the hole method would be tainted. This approach is not wrong, but has two big problems: first, it is very innacurate, since many instructions will always execute inside a try block, no matter if an exception happens or not, so unnecessary exceptional edges will be considered, and these instructions would later be marked as tainted without really needing to. Second problem, that happens because of first one, is taint spread: statements tainted without need would also propagate tainting to normal nodes.

To solve these problems, a better approach was used, and to understand it, it is necessary to explain a little about *post-dominators* in graph theory.

The definition of post-dominators, according to (CHEN, 1997) is: ”For any nodes x and y, $x \neq y$, in a control flow graph, y post-dominates x if every path from x to n_x passes through y.”. Figure 3.1 shows the post-dominator graph for the explicit flow example in figure 2.2. This example is very simple, there are no branches in the CFG, thus the post-dominator graph is basically the same as CFG, but upside-down. Let's use another example to show better how it works: figure 3.2 has a more complex CFG, with two branch points: ”if(salary>2000)” and ”if('peter'.equals(name))”. To simplify this CFG, the first three boxes can be seen as just one big block, because their flow is well-defined, i.e., is always the same. And to make this graph even easier, we can forget about the actual statements in the nodes, and just consider the CFG behavior. This new graph is shown in figure 3.3.

Now each node is analyzed, and a post-dominator graph is generated according to the definition given above. For instance, the post-dominators for node 4 are that nodes that

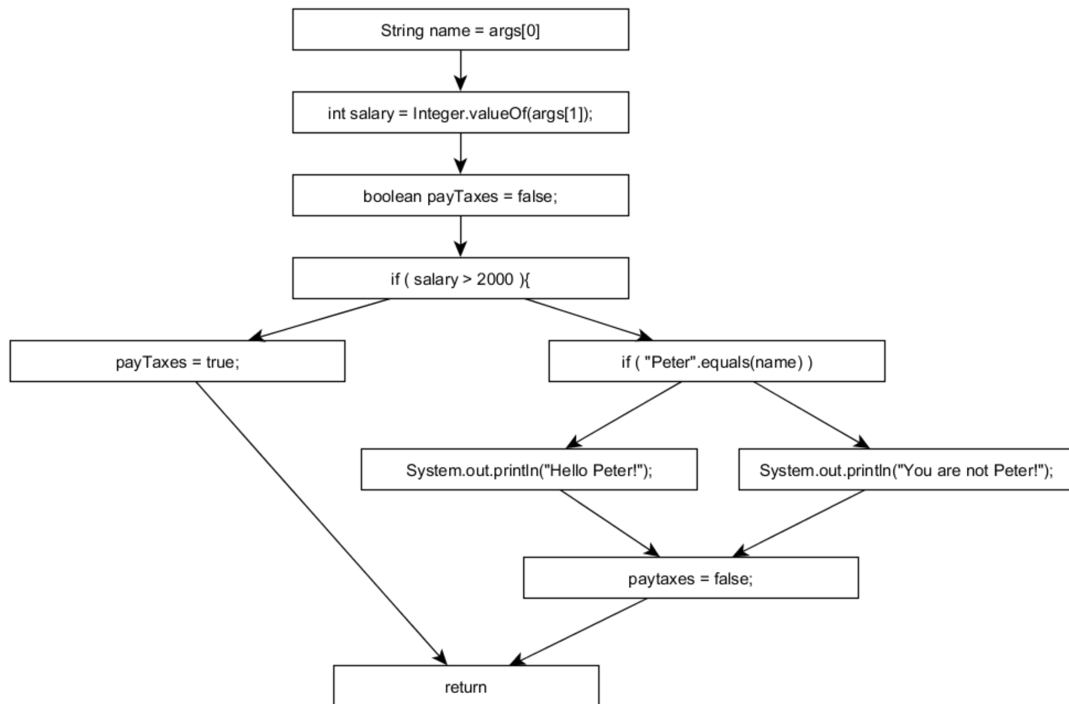

Figure 3.2: Control-flow graph with two branch points

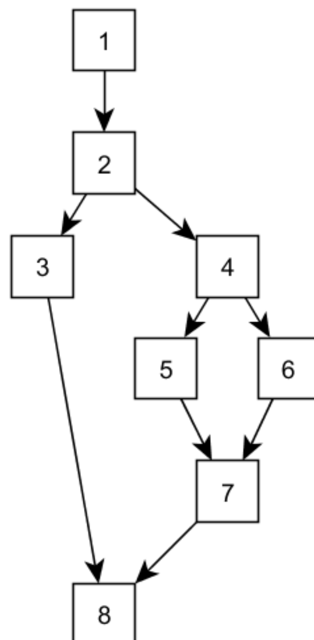

Figure 3.3: Simplified CFG for figure 3.2

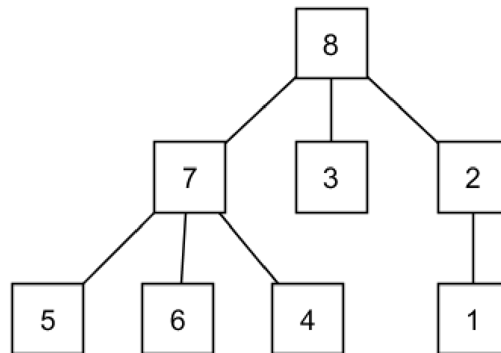

Figure 3.4: Post-dominator graph for figure 3.3

are always executed after node 4: these are 7 and 8. Another example, post-dominators of node 1 are nodes 2 and 8. The complete post-dominator graph is illustrade in figure 3.4.

In the same paper, the following is defined as the immediate post-dominator: ”If a node y is a post-dominator of x such that y is post-dominated by any post-dominator of xm we say y immediately post-dominates x.”. The immediate post-dominator is basically the first post-dominator of a node, and this is the key node to make implicit flow analysis. These nodes are important because they are the join point of 2 or more branches. Whenever a branch point is reached during an analysis, there is implicit flow going on in all its branches, so influences are calculated and propagated. The immediate dominator of these branches is the point where all flows go back to just one, and this point is set to stop the influences propagation. This was the approach used by Willenbrock in his implicit flow analysis, where he considered that branches could only start with an *if* statement. But what does this have to do with exceptional analysis? Adding exceptional edges to the CFG causes implicit flow to happen on every unit capable of throwing an exception: next node can be one in exceptional or unexceptional flow.

With that in mind, knowing the immediate post-dominator of a instruction inside a try-block leads to knowing when the handler block ends, and flow joins back to just one main flow. That is essential to stop propagation in exceptional analysis.

3.1 Solution

Now that some considerations have been made, the approach used in this framework to get a more precise analysis is explained.

First of all, two CFGs are loaded with SOOT: CompleteUnitGraph and BriefUnitGraph. These structures are defined in (JORGENSEN, 2003):

> **CompleteUnitGraph** is a CFG including a node for each unit, and incorporating edges representing exceptional control flow. Soot represents exception table entries by ”traps”, which comprises a pair of unit references delimiting the protected area, a reference to the initial unit in the exception handler, and the type of the catch parameter. A CompleteUnitGraph contains edges to each trap handler from every unit protected by the trap, as well as from the predecessors of the first trapped unit.
>
> **BriefUnitGraph** is a CFG including a node for each unit, but containing no edges for exeptional control flow.

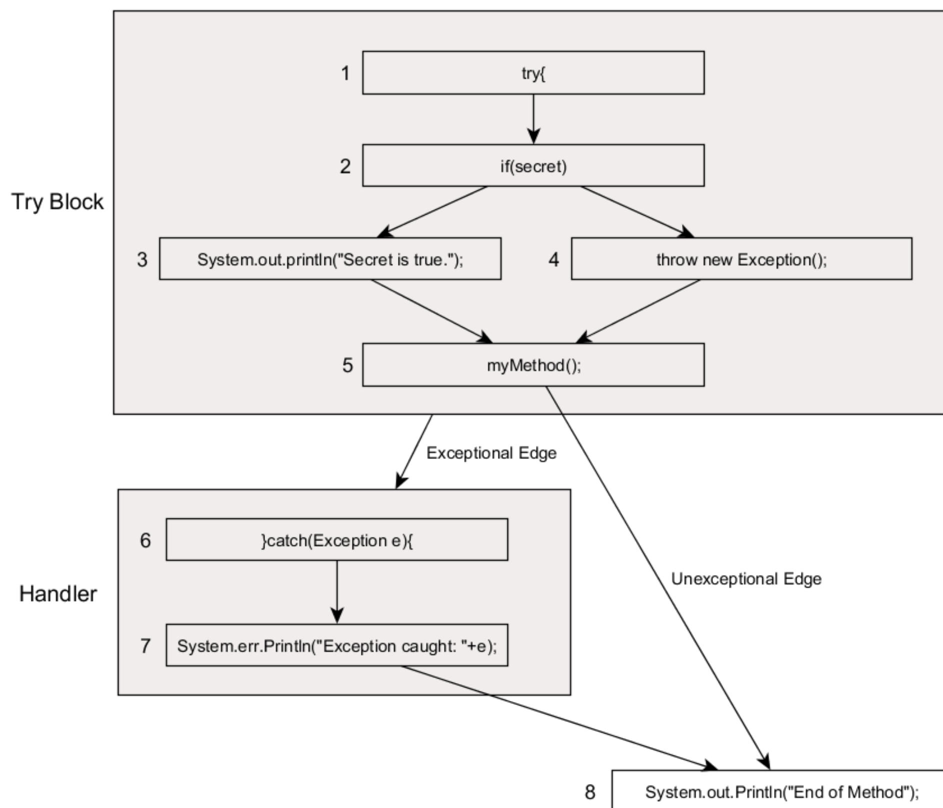

Figure 3.5: Control flow with exceptional and unexceptional edges

The former structure considers exceptional edges, so it is useful to calculate immediate post-dominators for try blocks, i.e., the first statement after the handler blocks. The latter structure is also used in the analysis because it does not consider exceptional edges, so it can get for instance a join point of a branch statement inside a try block. Both cases are explained next with the help of figure 3.5. The figure has a try block, which every instruction can have two destinations at least: its normal edge, and the exceptional edge going out of try block towards the handler. If no exception is thrown inside the try block, flow goes through *Unexceptional Edge* to unit number 8. On the other hand, if an exception is raised anywhere inside try block, flow goes through *Exceptional Edge* to the handler block. If a CompleteUnitGraph is used to represent the try block, the immediate post-dominator for every unit will be *unit 8*, as it is the statement that will always execute after the handler. If a BriefUnitGraph is used instead, it is possible to point out that *unit 5* is the immediate post-dominator for *unit 2*.

3.1.1 Try blocks

In methods without possibility of exception propagation to callers, the target of the analysis are especially the *try blocks*, and some instructions before the try block, when it is already inside a branch in the flow. The instructions inside these blocks(and also information about handler blocks and exception types) are get with help of the Java exception table, and these instructions are the only capable of throwing checked exceptions in this method. But the goal of our analysis is to identify only units that can really influence somehow the flow of information in the presence of exceptions, so some criteria must be applied to differ units that may influence or not a future taint propagation.

```
1  private void tryExample( boolean flag ){
2    try{
3      someMethod();
4      if(flag)
5        System.out.println(''Flag is true.'');
6      else{
7        throw new Exception();
8      }
9    }catch ( Exception e ){
10     System.out.println(''Exception caught.'');
11   }
12 }
```

Listing 3.1: Simple try block example

These criteria are based on the algorithm in listing 3.4. The idea behind the algorithm, as said before, is that not all units influence the information flow when an exception is raised. Consider listing 3.1: in this method, line 3 will always execute, and let's say the method *someMethod* will always return without any exception propagation. As this instruction always executes, and there is no exceptional edge, we can infer there is no implicit flow here. Going on to line 4, a branch point is introduced: flow can go to line 5 or line 7; if the condition is true, normal flow is actived, else, flow goes to it's exceptional edge, as a new exception is throw. With this example it is clear that the only units influenced by the presence of an exception are the units after the branch point, because one of the branches raises an exception. Everything that comes before this branch point is considered irrelevant to the exceptional analysis, and is not important anymore.

Back to explaining the algorithm, it's main idea is: find the *throw instruction*, and propagate influences *towards* until the join point for normal and exceptional flow, which is usualy the end of a handler block, and also *backwards* until a *key point*. These *key points* can be a branch point of an if statement, a join point after an if statement, or a method call with possibility of exception propagation. The three situations are shown in listings 3.1, 3.2, and 3.3.

3.1.2 Methods propagating exceptions to caller

When analyzing methods that have a "throws" declaration in it's signature, this method can propagate an exception to it's caller, so the algorithm has to be changed. The new algorithm(listing 3.5) is slightly different from the previous one. When there is no try block, it acts just as the other algorithm, but the end point is the end of the method. In cases where there are try blocks inside the method, the try algorithm above is used, but for each unit capable of throwing an exception, i.e., a throw statement or a method call, the type of the exception is observed, in order to guarantee that the exception object will be caught by the handler block. If the object is caught by the handler, influence propagation for flow inside this try block will end at the handler block. On the other hand, when there is no handler for exception type, the propagation of influences will be done until the end of the method, starting at the starting point. Type polymorphism cases might be a problem here, because the exception object can be of an unexpected type. Some extra checking should be done at runtime to guarantee correct taint propagation in these context(polymorphism), and this is an extension sugestion to current work, since this framework just works with

```
1  private void tryExample( boolean flag ){
2    try{
3      someMethod();
4      if(flag)
5        System.out.println(''Flag is true.'');
6      else{
7        System.out.println(''Flag is false.'');
8      }
9      // join point, units from here on
10     // are influenced by exceptions
11     someMethod();
12     throw new Exception();
13   }catch ( Exception e ){
14     System.out.println(''Exception caught.'');
15   }
16 }
```

Listing 3.2: Key point is IF join point

```
1  private void tryExample( boolean flag ){
2    try{
3      System.out.println(''Hi.'');
4      int a = 5;
5      // this method can throw an exception, so
6      // all units from here on are influenced
7      methodWithExceptionPropagation(flag);
8      a=10;
9      anotherMethod(a);
10   }catch ( Exception e ){
11     System.out.println(''Exception caught.'');
12   }
13 }
```

Listing 3.3: Key point is method call

```
1  propagate=false
2
3  for Unit u in UnitChain
4    if startUnit is null
5      startUnit = u
6    if endUnit is null
7      endUnit = getTryImmediateDominator()
8
9    if (!propagate)
10     if u is methodCall
11       if insideIf
12         if methodThrowsException
13           propagate = true
14       else
15         startPoint = u
16
17     if u is ifStatement
18       if !insideIf
19         insideIf = true
20         startPoint = u
21
22     if u is ifJoinPoint
23       insideIf = false
24       startPoint = u
25
26     if u is throwStatement
27       propagate = true
28
29 // propagation start and end points
30 setInfluences(startUnit, endUnit)
```

Listing 3.4: Algorithm for units inside try block

static analysis so far.

3.2 Problem with post-dominators

While using SOOT to get the immediate post-dominators of the statements, there were some cases where it would get unexpected results, like in listing 4.1, the immediate post-dominator for line 5 should be line 9, but as the last unit before the handler is a throw instruction, SOOT considers that the next statement would be always the handler, which is not true, since when the condition is false, no exception is thrown. To make everything work as expected, a little modification is done before actual analysis: the throw instructions that could lead to this problem are transformed into a method call to *throwException*, a method that simply receives an exception object and rethrows it, so no semantics change is done in the program analyzed(listing 3.6).

```
1  propagate=false
2
3  for Unit u in UnitChain
4    if startUnit is null
5      startUnit = u
6    if endUnit is null
7      endUnit = lastInstructionInMethod
8
9    if (!propagate)
10     if u is insideTryBlock
11       /*use Try algorithm, BUT check exception
12       types, to see if exception is really handled
13       by try block, otherwise propagation goes
14       outside try block */
15
16     else
17       if u is methodCall
18         if insideIf
19           if methodThrowsException
20             propagate = true
21         else
22           startPoint = u
23
24       if u is ifStatement
25         if !insideIf
26           insideIf = true
27           startPoint = u
28
29       if u is ifJoinPoint
30         insideIf = false
31         startPoint = u
32
33       if u is throwStatement
34         propagate = true
35
36 // propagation start and end points
37 setInfluences(startUnit, endUnit)
```

Listing 3.5: Algorithm for units inside method with ”throws” in it’s signature

```
1  public void throwException( Exception e ) throws Exception{
2    throw e;
3  }
```

Listing 3.6: Method that receives an exception and rethrows it

4 RESULTS

This chapter shows the resulting code after using the framework on some methods. The resulting codes were read from the class file generated, using a Java decompiler, JD-GUI (JDGUI, 2010). To help finding inserted code, some comments were added, pointing these instrumented instructions.

First example, listing 4.1, is the simplest case, where the hole try block should propagate influences. A *throw* statement is found in line 5, the start point is defined as line 4 (throug backwards propagation), and the end propagation marker is inserted after handler, in line 9. The result code is in listing 4.2.

```
1  public static void simpleTry2(){
2    int a = 1;
3    try{
4      if(a==2)
5        throw new Exception();
6    }catch(Exception e){
7      System.out.println("Exception␣caught.");
8    }
9    a=3;
10   System.out.println("a="+a);
11 }
```

Listing 4.1: Simple example

Listing 4.4 shows a case where a try block is entered, flow branches in line 5, joins

```
1  public static void simpleTry2()
2  {
3    int i = 1;
4    try
5    {
6      // instrumented code -> start propagation here
7      ExceptionHandler.startPropagation();
8      if (i == 2)
9      {
10       Exception localException1 = new java/lang/Exception;
11       localException1.<init>();
12       // method call to throwException
```

Listing 4.2: Instrumented simple example

```
      throwException(localException1);
    }
  }
  catch (Exception localObject1)
  {
    Object localObject1 = localException2;
    localObject1 = System.out;
    ((PrintStream)localObject1).println("Exception␣caught.");
  }
  // instrumented code -> end propagation here
  ExceptionHandler.endPropagation();
  int j = 3;
  PrintStream localPrintStream = System.out;
  Object localObject2 = new java/lang/StringBuilder;
  ((StringBuilder)localObject2).<init>();
  localObject2 = ((StringBuilder)localObject2).append("a=");
  localObject2 = ((StringBuilder)localObject2).append(j);
  localObject2 = ((StringBuilder)localObject2).toString();
  localPrintStream.println((String)localObject2);
}
```

Listing 4.3: Instrumented simple example(Continued)

back in line 10, and branches again in line 11. Analyzing this code, a intruction throwing an exception can just be seen in line 14. This istruction is inside the branch starting in line 11, so, everything coming before this branch point is not sensitive to our purpose. In the instrumented result(listing 4.5), we can see that propagation begins just from the second branch point, as first one could throw no exception, and ends after the catch block.

```
public static void tryAnd2If(){
  int a = 1;
  try{
    a=3;
    if (a==2)
      System.out.println("a␣is␣2!");
    else
      System.out.println("No␣exception");
    a=4;
    if (a==4)
      System.out.println("a␣is␣4!");
    else
      throw new Exception();
    a=5;
  }catch(Exception e){
    System.out.println("Exception␣caught.");
  }
  a=6;
}
```

Listing 4.4: Try block with two branch points

```
public static void tryAnd2If(){
  int i = 1;
  try{
    i = 3;
    PrintStream localPrintStream1;
    if (i == 2)
    {
      localPrintStream1 = System.out;
      localPrintStream1.println("a␣is␣2!");
    }else{
      localPrintStream1 = System.out;
      localPrintStream1.println("No␣exception");
    }
    int j = 4;
    // instrumented code -> start propagation here
    ExceptionHandler.startPropagation();
    Object localObject1;
    if (j == 4)
    {
      localObject1 = System.out;
      ((PrintStream)localObject1).println("a␣is␣4!");
    }
    else
    {
      localObject1 = new java/lang/Exception;
      ((Exception)localObject1).<init>();
      // method call to throwException
      throwException((Exception)localObject1);
    }
    int k = 5;
  }
  catch (Exception localObject2)
  {
    Object localObject2 = localException;
    localObject2 = System.out;
    ((PrintStream)localObject2).println("Exception␣caught.");
  }
  // instrumented code -> end propagation here
  ExceptionHandler.endPropagation();
  int l = 6;
  PrintStream localPrintStream2 = System.out;
  Object localObject3 = new java/lang/StringBuilder;
  ((StringBuilder)localObject3).<init>();
  localObject3 = ((StringBuilder)localObject3).append("a=");
  localObject3 = ((StringBuilder)localObject3).append(l);
  localObject3 = ((StringBuilder)localObject3).toString();
  localPrintStream2.println((String)localObject3);
}
```

Listing 4.5: Instrumented try block with two branch points

```
1  public static void ifBeforeTry2(){
2    int a = 1;
3    if (a==1){
4      a=2;
5      System.out.println(a);
6    }else{
7      System.out.println("else");
8      try{
9        if (a==4){
10         System.out.println("a is 4!");
11       }else{
12         throw new Exception();
13       }
14       a=3;
15
16     }catch(Exception e){
17       System.err.println("EXCEPTION");
18     }
19     a=4;
20   }
21   System.out.println("END");
22 }
```

Listing 4.6: Branch point before try block

Next test case, listing 4.6, is a very interesting one, because the try block(starting at line 8) is already inside a branch, so, during the static analysis, when the *throw* instruction is found, backward propagation starts, and it finds out that this unit is inside a branch outside the try block. In line 3, a branch started and has not yet been closed. This makes the statement in line 3 to be the startin point to our propagation. Considering that this propagation starts before the try block, the end point will not necessarily be located after the exception handlers, but after the join point of the mentioned branch(line 20).

The earlies methods were all methods that could not propagate exceptions to their caller, so analysis was just for their try blocks. Next two examples are based on methods capable of throw an exception to their caller in the stack.

As explained earlier, methods propagating exception to callers have to be analyzed differentily, because the handler may not be inside method body, but in a caller method of the call stack. In these cases, the end marker for propagation in this method will be the method's end point, thus, once the start propagation point started, propagation will usually happen until the end of the method.

In listing 4.8, there is a method with "throws ArrayIndexOutOfBoundsException" declaration in its signature. First statement, *System.out.println* cannot throw an exception in this example, and at line 5, there is a call to a method potentially raising an exception of type *ArrayIndexOutOfBoundsException*. Knowing this, the call to *m1* is considered as a branch point, and it could possible leak some information, so influence propagation is set to start before line 5, until the end of the method.

A last case to show, listing 4.10, is a method also capable of throwing an exception to its caller. But taking a closer look, the only unit raising an exception is inside a try

```
1  public static void ifBeforeTry2(){
2    PrintStream localPrintStream1 = 1;
3    // instrumentedcode -> start propagation here
4    ExceptionHandler.startPropagation();
5    PrintStream localPrintStream2;
6    if (localPrintStream1 == 1)
7    {
8      localPrintStream1 = 2;
9      localPrintStream2 = System.out;
10     localPrintStream2.println(localPrintStream1);
11   }else{
12     localPrintStream2 = System.out;
13     localPrintStream2.println("else");
14     try
15     {
16       localPrintStream2 = localPrintStream1;
17       int i = (byte)localPrintStream2;
18       Object localObject;
19       if (i == 4)
20       {
21         localObject = System.out;
22         ((PrintStream)localObject).println("a␣is␣4!");
23       }
24       else
25       {
26         localObject = new java/lang/Exception;
27         ((Exception)localObject).<init>();
28         // method call to throwException
29         throwException((Exception)localObject);
30       }
31       int j = 3;
32     }
33     catch (Exception localException2)
34     {
35       Exception localException2 = localException1;
36       PrintStream localPrintStream3 = System.err;
37       localPrintStream3.println("EXCEPTION");
38     }
39     int k = 4;
40   }
41   // instrumented code -> end propagation here
42   ExceptionHandler.endPropagation();
43   PrintStream localPrintStream4 = System.out;
44   localPrintStream4.println("END");
45 }
```

Listing 4.7: Intrumented branch point before try block

```
1 private static void m2()
2   throws ArrayIndexOutOfBoundsException{
3   System.out.println("Entered␣method␣m2");
4   // this method throws an ArrayIndexOutOufBoundsException
5   m1();
6 }
```

Listing 4.8: Method capable of propagate exception

```
1  private static void m2()
2    throws ArrayIndexOutOfBoundsException
3  {
4    PrintStream localPrintStream = System.out;
5    localPrintStream.println("Entered␣method␣m2");
6    // instrumented code -> start propagation here
7    ExceptionHandler.startPropagation();
8    // this method throws an ArrayIndexOutOufBoundsException
9    m1();
10   // instrumented code -> end propagation here
11   ExceptionHandler.endPropagation();
12 }
```

Listing 4.9: Intrumented method capable of propagate exception

block. With this in mind, the exception object is analyzed, and its type is compared to the types handled by the catch blocks(in this case, there is just one handler block). We can see that the exception thrown in line 6 will be always caught in line 8, as the handler gests exception of the same type. As no more exception objects are thrown in this method, the propagation stays just inside the try block, starting in line 5, and ending in line 10.

```java
private static void m5(int a)
  throws IndexOutOfBoundsException{
  System.out.println("Entered␣method␣m3");
  try{
    if (a==1){
      throw new ArrayIndexOutOfBoundsException();
    }
  }catch(ArrayIndexOutOfBoundsException e){
    System.out.println("Caught␣exception␣in␣m5");
  }
  System.out.println("Leaving␣m3.");
}
```

Listing 4.10: Try block inside method with ”throws”

```java
private static void m5(int paramInt)
  throws IndexOutOfBoundsException
{
  Object localObject1 = System.out;
  ((PrintStream)localObject1).println("Entered␣method␣m3");
  try
  {
    // instrumented code -> start propagation here
    ExceptionHandler.startPropagation();
    if (paramInt == 1)
    {
      localObject1 = new java/lang/ArrayIndexOutOfBoundsException;
      ((ArrayIndexOutOfBoundsException)localObject1).<init>();
      // method call to throwException
      throwException((Exception)localObject1);
    }
  }
  catch (ArrayIndexOutOfBoundsException localObject2)
  {
    localObject2 = localArrayIndexOutOfBoundsException;
    localObject2 = System.out;
    ((PrintStream)localObject2).println("Caught␣exception␣in␣m5");
  }
  // instrumented code -> end propagation here
  ExceptionHandler.endPropagation();
  Object localObject2 = System.out;
  ((PrintStream)localObject2).println("Leaving␣m3.");
}
```

Listing 4.11: Instrumented try block inside method with ”throws”

5 CONCLUSION AND FUTURE WORK

Working with Java Byte Code analysis and instrumentation is important to improve security in Java programs. Willenbrock showed in his bachelor thesis an approach to implicit information flow analysis, in order to taint pieces of code that might reveal some sensitive information given by the user. But his thesis did not deal with exceptions happening along the code. The majority of Java programs use exception handling to deal with unnexpected behaviour in runtime, so in this thesis was presented a way to analyze information flow in the presence of checked exceptions.

The most important goal during this analysis was not to retrieve all instructions enclosed in a situation where an exception could happen and mark these statements as influences to a taint propagation, but to minimize this group of instruction to just the ones that really could leak some data to an attacker.

Along the thesis were present notions of exceptions, how information can flow in a Java program, and ideas of how to get just influencing statements, with the least false positives as possible. Later, algorithms to statically analyze the code, marking start and end points to future taint propagation were explained, and some results of these algorithms being applied in dummy methods were showed in previous chapter.

Some ideas of future work were also presented, for instance, an analysis of polymorphism cases can be done at runtime, because an exception could be at runtime, instantiated with a different type than foreseen in static analysis, and influences would have to be updated. Another work to be done is integrate the framework with Willenbrock's, to use his taint propagation.

Including unchecked exceptions to the analysis is also an important topic, but as almost everywhere in the program a runtime exception can happen, an idea would be to at first use just a few runtime exception types, like supposing that all array accesses could throw and *ArrayIndexOutOfBoundsException* for instance, to watch the impact this would cause in taint propagation.

REFERENCES

CHANG, B. et al. Interprocedural Exception Analysis for Java. In: Symposium on Applied Computing, 2001. **Proceedings...** Las Vegas, United States, 2001. p.620–625.

CHEN, T. Y.; CCHEUNG Y. T.. Structural Properties of Post-Dominator Trees. In: AUSTRALIAN SOFTWARE ENGINEERING CONFERENCE, ASWEC '97, 1997, Sydney, Australia. **Proceedings...** Proceedings of Australian Software Engineering Conference ASWEC 97, pp. 158.

CHOI, J. et al. Efficient and Precise Modeling of Exceptions for Analysis of Java Programs. In: Workshop on Program Analysis for Software Tools and Engineering, 1999. **Proceedings...** Toulouse: ACM, 1999. p.21–31.

GOSLING,J. et al. **The Java™Language Specification**. 3rd ed. Boston: Addison Wesley, 2005. 649p.

HARROLD, M. J.; SOFFA, M. L. . Interprocedural Data Flow Testing. In: International Symposium on Software Testing and Analysis, 1989. **Proceedings...** New York: ACM, 1989. p.158–167.

JDGUI. **Java Decompiler project**. Available at: <http://java.decompiler.free.fr/>. Accessed in June, 2010.

JO, J.; CHANG, B.. Constructing Control Flow Graph for Java by Decoupling Exception flow from Normal Flow. **Lecture Notes in Computer Science**, Heidelberg, n.3043. p.106–113, 2004.

JORGENSEN, J.. **Improving the precision and correctness of exception analysis in Soot**. 2003. 70 p. Sable Technical Report. McGill University, School of Computer Science, Canada.

MALAYERI, D.; ALDRICH, J. . Practical Exception Specifications. **Lecture Notes in Computer Science**, Germany, n.4119, p.200–220, 2006.

NEHMER, N.. An Exception Handling Framework. In: European Conference on Object-Oriented Programming 2008, Doctoral Symposium, Cyprus, July 2008.

ROBILLARD, M. P.; MURPHY, G. C.. **Analyzing Exception Flow in Java Programs**. 1999. 16p. Technical Report – University of British Columbia, Vancouver, Canada.

ROBILLARD, M. P.; MURPHY, G. C.. Static Analysis to Support the Evolution of Exception Structure in Object-Oriented Systems. **ACM Transactions on Software Engineering and Methodology (TOSEM)**, New York, v.12, Issue 2, p.191–221, 2003.

SABELFELD, A.; MYERS, A. C.. Language-Based Information-Flow Security. **IEEE Journal on Selected Areas in Communications**, v.1, n.21, p.5–19, 2003.

SIMPSON, P.. **Information Flow Control Security in Java Virtual Machines**. 2007. 58p. Master Thesis (Master in Computer Science) –Vrije University, Faculty of Exact Sciences, Amsterdam, Netherlands.

SINHA, S.; HARROLD, M. J.. Analysis and Testing of Programs With Exception-Handling Constructs. **IEEE Transactions on Software Engineering**, v.26, n.9, sep.2000.

SINHA, S.; ORSO A.; HARROLD, M. J.. Automated Support for Development Maintenance and Testing in the presence of Implicit Control Flow. In: International Conference on Software Engineering, 26., 2004. **Proceedings...** Washington: IEEE Computer Society, 2004. p.336–345.

SINHA, S.; HARROLD, M. J.. Criteria for Testing Exception Handling Constructs in Java Programs. In: IEEE International Conference on Software Maintenance, 1999. **Proceedings...** Washington: IEEE Computer Society, 1999. p.265.

SMITH, G.. On the Foundations of Quantitative Information Flow. In: International Conference on Foundations of Software Science and Computational Structures: Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, 12., 2009. **Proceedings...** York: Springer-Verlag, 2009. p.288–302.

SOOT. **SOOT: a Java Optimization Framework**. Available at: <http://www.sable.mcgill.ca/soot/>. Accessed in June, 2010.

WILLENBROCK,S.. **Engineering Java Byte Code to Detect Implicit Information Flow**. 2009. 74p. Bachelor Thesis (Bachelor in Computer Science) – Technische Universität Kaiserslautern, Kaiserslautern, Germany.