Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Information Flow Analysis on the Java Byte Code Level

Master Thesis

Jan-Filip Zagalak
<jzagalak@ethz.ch>

Supervisor: Prof. Dr. Alexander Pretschner, Matus Harvan

Information Security Group
Department of Computer Science
ETH Zurich

March 18, 2009

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Abstract

This work presents an information flow framework for analyzing Java byte code. Explicit and implicit information flows are traced using a hybrid analysis i.e. a combination of static (before execution) and dynamic analysis (during execution). A security monitor ensures that information flows according to the declarations in a security policy, which specifies proper and improper information flow. Technically, the information flow analysis is integrated into Java input programs through byte code instrumentation. The instrumented byte code performs information flow tracing. This framework is independent from Java runtime systems and compilers. In order to allow information flow tracing to be more exact in array data structures, single array elements are analyzed.

I hereby want to thank Professor Alexander Pretschner for giving me the opportunity of writing this master thesis under his wings. Matus Harvan for supporting this project and for helping with words and deeds. Jan Alsenz, Matthias Büchler, Peter Heinrich and Doctor Werner Dietl for their great help and constructive conversations. Robert Weiser, one of the greatest "LaTeX-wizards", for valuable advises and dirty tricks.

Special thanks go to my mother, brother and my girlfriend, for keeping me grounded and for their great efforts in my interest.

In loving memory of my father Bolesław Jerzy Zagalak, who once brought home a computer from his laboratory and . . .

Contents

1 Introduction

Controlling usage of sensitive information as digital data was always an important topic. Nowadays its importance is increasing even further, since the world is getting more connected and information exchange becomes easier and happens faster. Knowing that sensitive information given to another entity is treated according to the belief of its appropriate owner, is requested increasingly. With further spreading of information technology to different contexts this topic won't stagnate but its importance will rise even more.

Reaching for absolute information security will stand as a wonderful, even if not attainable, goal, but one can always try to come approximately near to it. This work is intended to perform a trial in this very direction through seek for existing information leaks.

Information flow takes place whenever variables contained in the program source code are written. In every assignment statement information flows from the right-hand side to the left-hand side expression. If a variable b is assigned to a variable a ($a = b$), a contains the same information as b, hence the information contained in b flows to a. This kind of information flow is referred to as explicit. Information can also flow implicitly in conditional statements. Both kinds of information flows are described in chapter 2. Knowledge about which information flows to which destination is a key requirement for usage control. Analyzing information flows can provide this kind of knowledge, hence it is a foundation of every usage control system. Such an analysis can be made on different levels like system call level (e.g. Windows API calls), CPU/executable level (e.g. x86 ASM), window server level (e.g. X11) or runtime system level (e.g. Java virtual machine)

Problem Statement. This work tackles the problem of information flow analysis of Java programs on byte level.

Solution. Because many of daily used Java programs are shipped without any source code, the Java byte code representation is used as only input for the analysis. Explicit and implicit information flows are analyzed using a hybrid analysis i.e. a combination of static (before execution) and dynamic analysis (during execution). Implicit information flow is only handled for the executing branch using a label for the program counter. A security policy identifies sensitive information and declares, which information flows are considered as proper and improper. The security monitor enforces this policy, hence makes sure that sensitive information is not leaking. The security monitor and the information flow analysis are integrated in the input application through Java byte code instrumentation. The original byte code is extended in order to allow information flow to be traced. This approach works for arbitrary Java byte code (including handwritten byte code) on any Java virtual machine.

Contribution. This work sheds some light on details of the implementation of such an information flow tracing framework. Unlike other work in this context, every step towards realization is being discussed. Subproblems are presented together with corresponding solutions and algorithms. A thorough evaluation presents performance measurements and brings out which applied concepts did work and which didn't.

The rest of this thesis is organized as follows. Chapter 2 raises the topic of information flow, introducing basic concepts of and ways to analyze it. Chapter 3 gives a general overview of the design and discusses some implementation details. There are three chapters dedicated to problems worth mentioning. Each of these chapters starts with the problem statement and then presents a particular solution. The first (chapter 4) of these three chapters applies oneself to the instrumentation of arbitrary Java byte code. The second one (chapter 5) deals with information flow analysis of array data structures. The last one (chapter 6) is about finding join points in control flow graphs. Then chapter 7 presents an evaluation and some results of this work. Chapter 9 concludes this thesis by summarizing the key facts and points towards future work.

2 Background

Computer programs are full of data i.e. information in digital form. They mostly consume information as input and generate new information as output. In-between consumption and generation the information doesn't stay in the same place all the time. There is always a flow of information, it moves or just flows between program parts. Answering questions about localization of information is really though. The simplest but most honest answer would be "*your information x was once in y, but now it can be everywhere*". It's clear that people with a delicate sense for security or privacy don't feel very comfortable with such an answer.

This chapter discusses information flow basics and different kinds of flows. Subsequently some methods are presented to follow information through all parts of the program. Concepts to prohibit unwanted information flows are also discussed[1].

2.1 Explicit Information Flow

The simplest form of information flow can be observed in an assignment expression like $a = b$. The semantics of this expression is that the value of the right-hand side variable b is assigned to the left-hand side variable a[2]. From the perspective of information flow a contains now the same value or more general the same information as b. Therefore information flows from b to a, which can be formally annotated as $b \mapsto a$ [5]. This form of information flow is called **explicit**. Of course the expressions involved in the assignment operation can be more complex and can contain several variables like $a = b + c * d - e$. The observed explicit information flow is $b, c, d, e \mapsto a$. a contains a mixture of all the information flowing from b, c, d, e. In general it is possible to define a quantitative measurement [9] and to assess how much information flows from each single variable compared to the overall information flow. This work treats all kind of flows equally and relinquishes quantitative conclusions. Formal information flow annotations will be separated from pseudo code with the $\|$-symbol. Figure 2.1 shows annotated example statements.

$$a = b + c * d - e; \quad \| \, b, c, d, e \mapsto a$$
$$f = d * 3 - g + b^e; \| \, d, g, b, e \mapsto f$$

Figure 2.1: Program statements with annotated explicit information flow.

[1] a good introduction to information flow can be found in [3]

[2] instead of variable, the term entity is often used in other publications

2.1.1 Method Calls

If the right-hand side expression contains a method call then the return value of this method flows into the left-hand side variable. The last statement is valid but more details about the flow can be found by inspecting the method. Generally speaking a method is a sequence of statements encapsulated into a method and located elsewhere in the program text segment. Information can flow through such a method and leave the method through a return statement. Such return statements contain variables. To gather more information flow details about method calls, the statements of the method (in the callee) can be inspected together with the variables involved in the return statement. These details can then be used to annotate the statement from where method is called (in the caller). Figure 2.2 declares a method $foo1()$, which is called and annotated in Figure 2.3 to demonstrate the annotation of return statements.

$$\begin{aligned} &\text{method } foo1() : \\ &\qquad \ldots \\ &\qquad return\ q; \end{aligned}$$

Figure 2.2: Example method $foo1()$.

$$\begin{aligned} a &= b + c * d - e; \quad \| \ b, c, d, e \mapsto a \\ f &= a - foo1(); \quad \| \ a, q \mapsto f \end{aligned}$$

Figure 2.3: Method call of method $foo1()$ with annotated explicit information flow.

The method $foo1()$ returns the value or just the information contained in the variable q[3]. The call of the method $foo1()$ in Figure 2.3 could be annotated in more details because it is known that $foo1()$ in Figure 2.2 returns the variable q. Therefore q flows into the variable f. As already mentioned before a method call transfers control to another location in the text segment, where the annotation of explicit flows can be continued.

2.1.2 Method Arguments

A method call not only returns information, it may also consume it through method arguments. The caller of the method passes information to the method through method arguments[4]. Inside the method the arguments flow to the variables used in the statements of the method or even to arguments of further method calls. The method $foo1()$ from Figure 2.2 is

[3] variable scoping is ignored here, (for simplification) it is assumed that each variable exists only once in the whole program

[4] for simplicity, other kind of information consumption like global variables, fields etc. are ignored

extended as method $foo2(arg_1,\ arg_2)$ depicted in Figure 2.4, it takes two arguments now. The method is called and annotated in Figure 2.5.

$$
\begin{aligned}
&\text{method } foo2(arg_1,\ arg_2): \\
&\qquad\qquad u = arg_1;\ \|\ arg_1 \mapsto u \\
&\qquad\qquad v = arg_2;\ \|\ arg_2 \mapsto v \\
&\qquad\qquad q = u + v; \|\ u, v \mapsto q \\
&\qquad\qquad return\ q;
\end{aligned}
$$

Figure 2.4: Example method $foo2(arg_1,\ arg_2)$.

$$
\begin{aligned}
&a = b + c * d - e;\ \|\ b, c, d, e \mapsto a \\
&f = foo2(a, b);\quad \|\ a \mapsto arg_1,\ b \mapsto arg_2,\ arg_1 \mapsto u,\ arg_2 \mapsto v,\ u, v \mapsto q,\ q \mapsto f
\end{aligned}
$$

Figure 2.5: Method call of $foo2(arg_1,\ arg_2)$ from Figure 2.4 with annotated explicit information flow.

The information flow annotation in Figure 2.5 is much more detailed due to the ”inspection” of the called method. The annotations are quite cluttered at the last line, section 2.2 introduces other ways of representation of information flow. At this point it should be stated that such detailed annotations of information flow can only be realized if the method can be inspected i.e. the text segment of the method can be analyzed. **This is not always the case.**[5]

2.2 Information Flow Graphs

The information flow annotations introduced in 2.1 are transitive. It means that if $a \mapsto b$ and $b \mapsto c$ then $a \mapsto c$. Hence the single flows per line/statement can be combined to get a bigger picture of all flows. Figure 2.6 shows an example, where information flow from Figure 2.5 is combined for the variable e.

It is possible to state that information contained in variable e flows over a couple of other variables (a, arg_1, u, q) to the variable f. Hence somebody interested in how the information contained in variable e spreads over program parts i.e. variables - can now get this information.

An information flow graph is a graphical representation of information flows. **Graph nodes** represent variables, whereas **directed edges** represent a flow of information from a source variable to a destination variable. An information flow graph is depicted in Figure 2.7. It shows all the information flows contained in the example of Figure 2.5. It is clear that

[5] discussed later in 2.9.3

$$
\begin{aligned}
e &\mapsto a \\
a &\mapsto arg_1 \\
arg_1 &\mapsto u \\
u &\mapsto q \\
q &\mapsto f \\
\Rightarrow e &\mapsto f
\end{aligned}
$$

Figure 2.6: Information flow combination for variable e from Figure 2.5. **Note:** The big arrow in the last line doesn't represent a logic implication, it is used to mark the result of the combinations of information flows.

also the transitive information flows depicted in Figure 2.6 are represented by paths in the information flow graph.

An information flow graph is always a snapshot of information flow at a certain point in runtime (at a certain line number). In this work graphs will be usually constructed after the last line of the program. The collection of information flows of variables and the construction

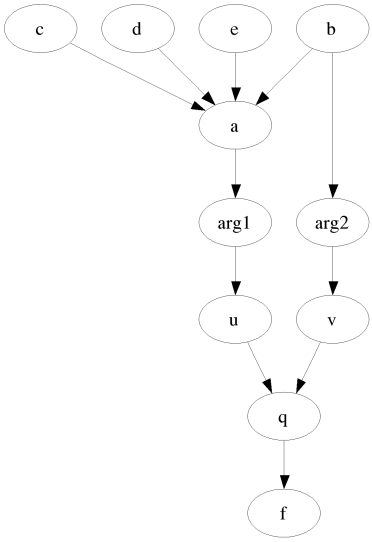


Figure 2.7: Information flow graph of example in Figure 2.5. The flow $e \mapsto f$ from Figure 2.6 can be recognized as path from e to f.

of flow graphs helps to answer questions like *"to what variables did the information contained in variable x flow to"*. Later in this chapter information sinks 2.6 and sources 2.4 will be discussed, which are also represented as nodes in the information flow graph.

2.3 Information Flow Analysis

An information flow graph (see 2.2) is constructed out of the collected information flow data, gathered by applying a so-called **information flow analysis**. It traces the flow of information throughout the whole program. This work uses the term **information flow tracing** for the appliance of an information flow analysis to an input program.

An analysis can be run *statically* or *dynamically* on a program. A static analysis inspects a program without running it. Because some runtime dependent aspects are missing, the analysis has to make some assumptions. A dynamic analysis on the other hand inspects the program during runtime. It is executed in parallel and monitors the behaviour of the program with respect to information flow. The difficulty lies in integrating such an analysis into a program. An analysis may trace all kind of variables or simply focus on certain ones.

2.4 Information Sources

So far the notion of explicit information flow together with a formal notation (see 2.1) and flow graphs (see 2.2) were introduced. Section 2.3 introduced information analysis as a means to information flow tracing. This section focus on how information is used and where information is coming from. Information flow tracing can be used to analyze information flows of a certain variable. The following list represents some use cases of information consuming programs.

1. The user starts an email client, enters his password to check his emails.
2. The user starts a tax declaration software, enters his earnings and prints out his tax declaration.
3. The user starts a messenger application, enters his credentials and starts chatting.

In all use cases information is entered by the user and consumed by the application. This user input may be more ”valuable” to the user than other ordinary data. For each use case an informal policy concerning the handling of the user input data by the application, emphasizes the meaning of ”valuable” data.

1. The password entered into the email client shall not be transmitted over the network without encryption.
2. The tax data entered into the tax declaration software shall not be transferred over the network.
3. The user credentials entered into the messenger application shall not be saved plain in the filesystem.

From this informal policies it can be seen that firstly there is information, which is important/valuable for the user. In this work this kind of information is called **sensitive information**. Secondly, it can be seen that there are some destinations that the user don’t want his sensitive information to flow to.

Before methods are discussed to prohibit certain information flows the identification and tracing of sensitive information has to be discussed. Sensitive information can be traced

with the help of information flow analysis (see 2.3). The analysis can start to trace from the variables containing the sensitive information[6]. Therefore they represent starting points of the information flow analysis. This starting points are so-called **sources**. Generally speaking the initial information flow is emitted by a source. A source can be declared on different levels of abstraction e.g. a variable, the return value of a method. In this work a source represents always a certain field. Further details can be found in 3.5.1.

The results of the analysis can be combined and visualized with an information flow graph (see 2.2). The roots of this graph are the starting points i.e. the sources - that have been declared to carry sensitive information. The sources in the flow graph in Figure 2.8 are colored blue. All nodes that are reachable from a source in the flow graph are variables,

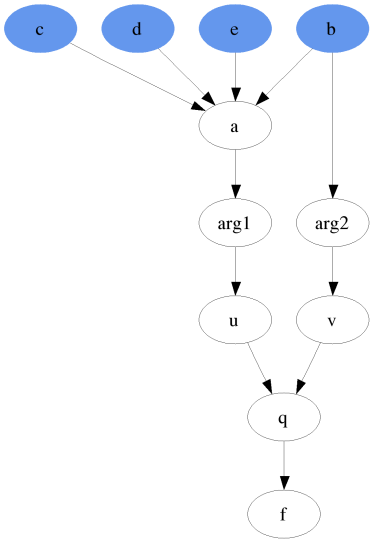


Figure 2.8: Information flow graph, information **sources** are colored blue.

which contain sensitive information from the source.

2.5 Tainted Variables

2.4 defined the term source for variables carrying sensitive information or simply starting points for information flow analysis (see 2.3). A variable that contains sensitive information is referred to be a **tainted** variable. Therefore all variables, which are on paths starting at sources in information flow graphs, are tainted variables. Hence the variables $a, arg_1, u, q, f, arg_2, v$ are tainted variables in Figure 2.8. Actually, sources are also tainted variables, sometimes they are referred to as *initially tainted*, but the term source has much more meaning and will be used instead.

The term **taint propagation** is used to describe the flow of tainted variables. If sensitive information from a tainted variable flows to another variable, this variable becomes tainted too, and therefore the taint has propagated.

[6] more precisely: the first variable that was assigned this sensitive information

2.6 Information Sinks

So far information sources (see 2.4) as starting points and tainted variables (see 2.5) as variables containing sensitive information were defined. What is missing so far is a description of the ”dangerous” destinations for information flows, where information shall not flow to. A destination can be seen as an output channel to a potentially hostile environment, where the flow of the information given to the output channel may no longer be traced nor controlled. Such destinations/output channels are called **sinks**. A sink may also be defined on different kind of levels of abstraction e.g. a certain method, a variable or a whole context (e.g. filesystem, network). In this work a sink represents always a certain method. Further details can be found in 3.5.2. The corresponding sinks for the use cases in 2.4 are:

1. unencrypted network channels
2. all network channels
3. unencrypted output channels to the filesystem

Sinks are also nodes in an information flow graph (see 2.2). A sink represents an output channel to an environment, where information flow tracing isn't possible any more, hence it is a leaf of the information flow graph.

2.7 Control of Information Flow

So far information flow basics were introduced together with information flow analysis. Now it is time to discuss ways to control certain flows of information. Before concrete methods are discussed, several new terms have to be introduced first.

As presented in the use cases in 2.4 certain flows of sensitive information shall not occur. If sensitive information flows to a sink the information is leaking. In order to prohibit information leaks a ”big brother” is needed that keeps watching the information flows.

2.7.1 Security Monitor

A security monitor is some kind of a ”big brother”. Generally speaking a security monitor (in this context) intercepts potentially ”dangerous” information flows and checks if the flows, which are going to happen, do leak information or not. Thus a particular operation of a program, that is going to leak information, can be blocked or logged depending on the enforcement mechanism of the security monitor. Interception means that the security monitor is called before the current operation is carried out. The operation is investigated and checked for information leaks. If the operation isn't leaking sensitive information the security monitor returns control to the program and the operation is carried out. The definition of a program operation depends on the level of abstraction. It may be a method call, an API-call, an interpreted instruction or a machine instruction. A security policy tells the security monitor, which kinds of operations are allowed and which aren't. The security monitor makes sure that the security policy isn't violated by prohibiting unwanted operations, in this case operations that leak sensitive data to sinks. More details are presented in 3.3.

2.7.2 Prohibit Information Leaks

Figure 2.9 depicts an example program. The variables $userpwd$, $earnings$ contain sensitive information and hence there are declared to be sources. The method $send_{network}(arg)$ can't be investigated in more details but it is identified as an output channel, which may not consume sensitive data. Therefore the method is declared as sink. The security monitor avoids leaks of sensitive information by simply prohibiting calls of the $send_{network}(arg)$ method with tainted arguments i.e. variables containing sensitive information (see 2.5).

```
1:   userpwd   = "secretstr"; % source_1
2:   earnings  = 101'000;     % source_2
3:   isrich    = false;       %  init boolean var: is rich
4:   taxdisc   = false;       %  init boolean var: gets tax discount
5:   tmp_1     = userpwd;     || userpwd ↦ tmp_1
6:   cond_1    = earnings;    || earnings ↦ cond_1
7:   tmp_2     = tmp_1;       || tmp_1 ↦ tmp_2
8:   tmp_3     = tmp_2;       || tmp_2 ↦ tmp_3
9:   if( cond_1 > 100'000 )
     {
10:    isrich = true;
     }
11:  else
     {
12:    taxdisc = true;
     }
13:  send_network(isrich);
14:  send_network(taxdisc);
15:  send_network(tmp_3);
```

Figure 2.9: Prohibiting information leakage, example program.

Figure 2.10 shows the corresponding information flow graphs. The graph is created after the last line of the program was executed. The security monitor intercepts every method call. Information flow tracing provides data about information flow that helps to decide on the method call.

During runtime line 13 and 14 are intercepted by the security monitor. The argument given to the $send_{network}(arg_1)$ method has to be checked for tainting. As can be seen in the flow graph neither $isrich$ is tainted at line 13 nor $taxsdisc$ at line 14 is tainted. Hence, the call of the method is granted by the security monitor.

At line 15 the security monitor checks (the same for) variable tmp_3. The flow from the source $userpwd$ over the variables tmp_1, tmp_2 to the variable tmp_3 is being noticed[7]. Hence, tmp_3 is tainted i.e. contains sensitive information - initially emitted by the source

[7] path found from $userpwd$ to $tmp3$

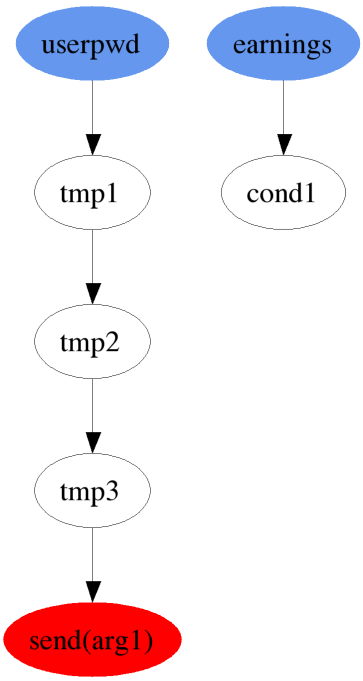


Figure 2.10: Information flow graphs of the example 2.9. The sources $earnings$, $userpwd$ are variables and they are colored blue. In the example the method $send_{network}(arg1)$ was declared as sink. In the graph the name $send(arg1)$ is used for the sink node, which is colored red.

$userpwd$. The call of this method would leak sensitive information therefore the security monitor prohibits this call to avoid an information leak.

2.8 Implicit Information Flow

The example in 2.7.2 demonstrated how to prohibit unwanted information flows to sinks. In this example the security monitor prohibited leakage of sensitive information contained in the sources $earnings$ and $userpwd$. This section analyses the example in more details.

2.8.1 Executing Branch

Information can flow in less obvious forms than explicitly. The conditional branch in the example of Figure 2.10 is analyzed in more details here.

The calls at line 13, 14 are granted by the security monitor because the arguments aren't tainted. At runtime the variable $cond_1$ at line 9 is tainted. It contains the value $101'000$ from the source $earnings$, hence the condition is evaluated to *true*, hence the if-branch is executed and the $isrich$ variable switches from *false* to *true*.

So far no explicit information flows are registered from a source to the variable $isrich$, hence it isn't marked as tainted. Later at line 13 the $isrich$ variable is consumed by the sink $send_{network}(arg_1)$ and sent to an output channel. So did this program finally leak sensitive

information? Unfortunately yes! The value of $isrich$ at line 13 is *true*, which was initialized to *false* at line 3. Hence it can be deduced that the variable $isrich$ was written during runtime[8]. And because of that, it is clear whether the if-branch was executed. Hence the condition of the if-statement was evaluated to *true*.

The final deduction is that because the logical expression $cond_1 > 100'000$ was *true*, $cond_1$ was bigger than $100'000$. The flow graph shows that information flows from the source $earnings$ to $cond_1$. Hence, sensitive information in $earnings$ leaked somehow over the variable $cond_1$ to the variable $isrich$. This flow of information is less obvious than explicit information flow, because it occurs implicitly. This kind of information flow is referred to as **implicit information flow of the executing branch**. Figure 2.11 shows the implicit information flows of the example in Figure 2.9

$$\begin{aligned} earnings &\mapsto_{explicit} cond_1 \\ cond_1 \quad &\mapsto_{implicit} isrich \\ earnings &\Rightarrow isrich \end{aligned}$$

Figure 2.11: Implicit information flow of example in Figure 2.9.

2.8.2 Non-Executing Branch

Unfortunately there are even more information leaks in the example of Figure 2.9. In 2.8.1 the if-statement was analyzed in more details and only the branch executed at runtime was inspected. Here the branch, which wasn't executed, shall be analyzed.

As we have seen so far the calls at line 13, 14 are granted by the security monitor, because the arguments aren't tainted. At runtime the variable $cond_1$ at line 9 is tainted. It contains the value $101'000$ from the source $earnings$, therefore the condition is evaluated to *true*, and therefore the if-branch is executed and the $isrich$ variable switches from *false* to *true*. This has of course another effect or has precisely not an effect, which is also observable.

At line 4 the value of $taxdisc$ is initialized to *false*. This variable changes its value only, if the else-branch is executed at runtime. Because in the example the if-branch is executed instead, the value of $taxdisc$ stays untouched. $taxdisc$ is still *false* when it is written to the output channel at line 14. Subsequently, it can be deduced that $taxdisc$ isn't written during runtime[9]. And because the value isn't written it's clear that the else branch is not executed, hence the condition of the if-statement evaluates to *true*.

The final deduction is the same as before in 2.8.1. The logical expression $cond_1 > 100'000$ is *true*, $cond_1$ is bigger than $100'000$. The flow graph shows that information flows from the source $earnings$ to the variable $cond_1$. Hence, sensitive information in $earnings$ leaked again somehow over the variable $cond_1$ to the variable $taxdisc$. This flow also occurs implicitly. This kind of information flow is referred to as **implicit information flow of the non-executing branch**. Figure 2.11 shows all implicit information flow occurring in the example depicted in Figure 2.9.

[8] elsewise the variable would still have the initial value *false*

[9] elsewise the variable would still have the initial value *true*

$$
\begin{aligned}
earnings &\mapsto_{explicit} cond_1 \\
cond_1 &\mapsto_{implicit} isrich \\
earnings &\Rightarrow isrich
\end{aligned}
$$

$$
\begin{aligned}
earnings &\mapsto_{explicit} cond_1 \\
cond_1 &\mapsto_{implicit} taxdisc \\
earnings &\Rightarrow taxdisc
\end{aligned}
$$

Figure 2.12: Implicit information flow of example in Figure 2.9.

2.8.3 Bottom Line

The bottom line of these investigations is that implicit information flow occurs in conditional statements, where tainted variables are involved in the conditional expression. Information from the condition flows to all variables contained in the **executing branch**, and to all written variables in the **non-executing branch**.

If at runtime a branch is executed, variables in the executed branch are written and therefore receive implicit information flow about the evaluation of the condition. On the other hand if at runtime a branch is not executed, variables in the not executed branch are not written and therefore have the value same as before the conditional statement. This is of course again observable, therefore they too receive implicit information flow about the evaluation of the condition.

The implicit flow of information didn't reveal all of the sensitive information. The source $earnings$ contains the sensitive information $101'000$, what leaked over the conditional branch and could be deduced is that $cond_1$ was *true* and hence $earnings$ was bigger than $100'000$. The exact value didn't leak at all, but a certain portion of it did[10] Note that being this a work that doesn't quantify information flow, every kind of information flow is treated equally.

2.9 Prohibiting Implicit Information Leaks

As discussed in 2.8 information may not only leak due to explicit but also to implicit information flow. This section presents a method to prohibit even implicit information leaks. The key is that the information flow analysis must be able to recognize implicit information flows as well. The security monitor simply applies the same strategy as before.

2.9.1 Executing Branch

To recognize implicit information flow in executing branches a program counter is needed (pc-label, for short). A program counter always points to the instruction which is executed next. An additional label is attached to the program counter. The pc-label may either be

[10] even without much theory an information flow can be recognized: Does a burglar need to know about the exact balance of his victim or is it enough for him to know that the victim is rich?

tainted or not tainted as with common variables so far. The pc-label is involved in the evaluation of a statement with respect to information flow. Generally speaking if the pc-label is tainted implicit information flow from executing branches flows also to written variables in the statements. A pc-label is said to be tainted i.e. the label is escalated - whenever a conditional expression containing tainted variables is evaluated. On the other hand a pc-label is untainted i.e. the label is de-escalated - whenever the join point (see 2.9.1.1) of the conditional statement is reached.

2.9.1.1 Join Point

Given a directed, cyclic graph representing control flow, the flow of control splits in two separate branches at conditional statements. Which of the branches is actually executed at runtime depends on the condition. The two separated branches will join back in exactly one node. This node is referred to as the join point of the corresponding conditional node. This topic is discussed in more details in 6.1.

2.9.1.2 PC-Label in Action

The pc-label can be implemented with a counter, whenever it is escalated the value is incremented, de-escalating decrements the counter. A counter bigger than zero means pc-label is tainted. Figure 2.13 shows the example presented in 2.7.2 with pc-label annotation ($pc_{false}, 0$ means pc-label not tainted). The condition of the if-statement at line 9 contains a tainted variable, hence the pc-label is escalated. As consequence the variable $isrich$ at line 10 is tainted because the pc-label is tainted at this point. Subsequently the implicit information flow of the executing branch is properly detected. Line 14 is the join point of the conditional statement, the pc-label is de-escalated at this line. Thanks to the pc-label the information flow to the $isrich$ variable could be recognized and depicted in the corresponding information flow graph in Figure 2.14. Hence, the security monitor can detect the implicit information flow from $cond_1$ to $isrich$ and prohibit the information leak caused by the call to $send_{network}(isrich)$ at line 15.

2.9.1.3 Nested Conditional Statements

It might look far to complicated to use a counter as pc-label to solve this problem, but there may be nested statements with varying condition taints. With this approach nested conditional statements can be handled properly. Figure 2.15 contains a more complex example with nested conditional statements, which could be handled correctly thanks to the counter.

2.9.2 Non-Executing Branch

In order to trace implicit information flow of the non-executing branch a lot more work has to be done. The principle is to include *compensating code* in the respectively other branch. If the branch is not executed, the implicit information flow from the non-executed branch is handled by the compensating code (described later in 2.9.2.1), which now is executed in the other executed branch. Figure 2.16 depicts this approach. If the if-branch (BB2) is executed at runtime the non-executing else-branch (BB3) is compensated by the code in BB6. On the other hand if the else-branch (BB3) is executed at runtime the non-executing if-branch

```
1:        userpwd = "secretstr";                    % pc_{false,0}
2:        earnings =101'000;                        % pc_{false,0}
3:        isrich =    false;                        % pc_{false,0}
4:        taxdisc =   false;                        % pc_{false,0}
5:        tmp_1 =     userpwd;                      % pc_{false,0}
6:        cond_1 =    earnings;                     % pc_{false,0}
7:        tmp_2 =     tmp_1;                        % pc_{false,0}
8:        tmp_3 =     tmp_3;                        % pc_{false,0}
9:        if( cond_1 > 100'000 )                    % pc_{true,1}, escalation
10:       {
11:          isrich = true;                         % pc_{true,1}
12:       }
11:       else
12:       {
13:          taxdisc = true;
14: JP_1  }                                         % pc_{false,0}, de-escalation
15:       send_network(isrich);                     % pc_{false,0}
16:       send_network(taxdisc);                    % pc_{false,0}
17:       send_network(tmp_3);                      % pc_{false,0}
```

Figure 2.13: PC-Label in action. If-branch is executed at runtime. $pc_{[true|false],i}$ the pc-label is either tainted (*true*) or not tainted (*false*), i is the counter mentioned above. JP_i is a join point of the i-th conditional branch. **Note:** Escalation at line 9, De-escalation at join point at line 14. Due to the pc-label the variable $isrich$ gets tainted at line 11.

(BB2) is compensated by the code in BB7. It's clear that the compensating code has to be inserted before the jump instruction (here *goto*). If nested branches have same join points, the compensating code insertion gets more complicated. The same is true for loops. A loop body has to be compensated if the body wasn't executed.

2.9.2.1 Compensating Code

Compensating code taints all the variables, which would have been written in the non-executing branch. To create compensating code for a branch, the branch has to be analyzed statically, because the code can't be executed due to potential side-effects that may change the program state. The static analysis determines all variables that are written. Actually all side-effects of the branch have to be determined because they are observable. Because the analysis is static (see also 2.3) some runtime dependent aspects are missing and hence assumptions are made. For object-oriented method calls, the called method depends on the runtime type of the receiver (see 3.4.3 for more details). Because only the static type of the receiver is known, one approach could be to conservatively compensate all possible methods that could be called.

If the condition in the example of Figure 2.9 is tainted and evaluates to true (at runtime), the compensating code for the else branch would taint the $taxdisc$ variable.

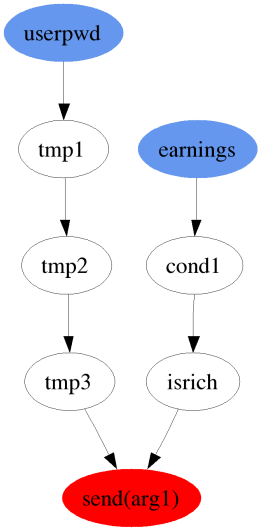


Figure 2.14: Information flow graph corresponding to Figure 2.13. The implicit information flow of the executing branch from $earnings$ to $isrich$ is recognized now. The sources $earnings$, $userpwd$ are variables and they are colored blue. In the example the method $send_{network}(arg1)$ was declared as sink. In the graph the name $send(arg1)$ is used for the sink node, which is colored red. $tmp3$ and $isrich$ are both called with the same sink $send(arg1)$, hence both variables flow to the sink.

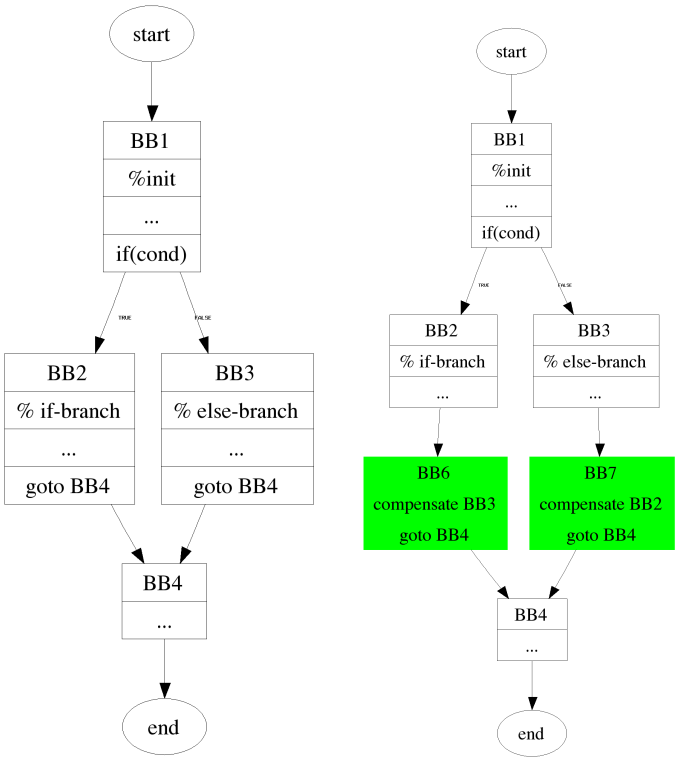
2.9.3 Control Borders

It is not always possible to investigate all of the program code as in 2.1.1 and to run the information flow analysis (see 2.3) on it. A statement may contain a library call that uses functionality on lower abstraction levels e.g. a system call, or the method is using code of a different programming language e.g. native methods in Java. In such cases analysis isn't complete, a border is met that can't be passed. Either the analysis assumes what happens beyond the border (e.g. if any argument provided to the library call is tainted, the return value is tainted too) or a higher analysis instance recognizes the situation and invokes another analysis that fits to the new environment beyond the border (e.g. an information flow analysis for C is invoked for native Java methods). Cross-border information flow analysis tends to be much more complex, because data has to be exchanged, and finding a common representation of concrete programming language mechanisms is often quite difficult. A higher instance has to control all the analysis on the different levels and has to keep track on the overall information flow.

It is clear that if information flow analysis isn't working beyond borders, neither does control. Therefore, information that flows beyond borders should be regarded as "lost" concerning control on the usage, because the security monitor can no longer guarantee the proper usage. To handle such borders all border-crossing methods can be declared as sinks and the appropriate handling can be specified for each sink.

```
1:          % init, pc_{false,0}
2:          if( cond_1 ) %pc_{false,0}
3:          {
3:              % ifbranch_1, pc_{false,0}
4:              if( cond_2 ) % pc_{true,1}, escalation
5:              {
6:                  % ifbranch_2, pc_{true,1} (all written variables are tainted)
7: JP_2         }% pc_{false,0}, de-escalation
8:              % ifbranch_1 continued, pc_{false,0}
9:              if( cond_3 ) % pc_{false,0}
10:             {
11:                 % ifbranch_3, pc_{false,0}
12:                 if( cond_4 ) % pc_{true,1}, escalation
13:                 {
14:                     % ifbranch_4, pc_{true,1} (all written variables are tainted)
15:                     if( cond_5 ) % pc_{true,1}
16:                     {
17:                         %ifbranch_5, pc_{true,1} (all written variables are tainted)
18: JP_5                }
19:                     % ifbranch_4 continued, pc_{true,1} (all written variables are tainted)
20: JP_4            } % pc_{false,0}, de-escalation
21: JP_3        }
22:             if( cond_6 ) % pc_{true,1}, escalation
23:             {
24:                 % ifbranch_6, pc_{true,1} (all written variables are tainted)
25:                 if( cond_7 ) % pc_{true,2}, escalation
26:                 {
27:                     % ifbranch_7, pc_{true,2} (all written variables are tainted)
28:                     if( cond_8 ) % pc_{true,3}, escalation
29:                     {
30:                         % ifbranch_8, pc_{true,3} (all written variables are tainted)
31: JP_8                }% pc_{true,2}, de-escalation
32:                     % ifbranch_7 continued, pc_{true,2} (all written variables are tainted)
33: JP_7            } % pc_{true,1}, de-escalation
34:                 % ifbranch_6 continued, pc_{true,1} (all written variables are tainted)
35: JP_6        } % pc_{false,0}, de-escalation
36:         % ifbranch_1 continued, pc_{false,0}
37: JP_1 } % pc_{false,0}
38: % end, pc_{false,0}
```

Figure 2.15: pc-Label in Action. All bold variables are tainted. In this example all conditional statements are evaluate to *true* at runtime. JP_i is a join point of the i-th conditional branch. $pc_{[true|false],i}$ the pc-label is either tainted (*true*) or not tainted (*false*), i is the counter mentioned above. **Note**: **line 17**: condition isn't tainted but all written variables in the body are because of the pc-label. **line 32**: pc-label is still *true* after de-escalation because $counter > 0$. **line 35**: pc-label is **false** here because the $counter == 0$.



(a) Control Flow Graph, variable in conditional statement *cond* is tainted.

(b) Control Flow Graph, added compensating code to both branches.

Figure 2.16: Compensating Code insertion to control implicit information flow of non-executing branches.

3 Design and Implementation

This chapter gives an overview of the framework design, the technical details of its implementation, the used technologies and the applied concepts. Some of the topics presented here are deepened in the problem chapters 4, 5 and 6 dedicated to a certain problem statement and its solution. Some of the points discussed here are evaluated in chapter 7.

3.1 Overview

The kind of information flow analysis (see 2.3) implemented in this framework is referred to as hybrid i.e. a combination of static (ahead of execution) and dynamic (concurrent with execution). The input application i.e. the Java byte code program that shall be analyzed - is instrumented (see 3.4) using the Byte Code Engineering Library (BCEL), which is part of the Apache Jakarta Project. A security policy specifies the parameters of the information flow analysis for the particular input application. The source code of the application isn't required. This framework is completely independent from the compiler used to produce the byte code. Any Java byte code verifiable by Java virtual machines can be processed. The same is valid for handwritten byte code. This framework is working standalone, no integration of components into libraries or the virtual machine is needed. It is working with any Java compatible virtual machine. The process of analyzing the input application with respect to information flow works in two phases:

1. **instrumentation-phase**
   The input application is instrumented according to the security policy (see 3.2). The original byte code is changed in such a way that information flow analysis is integrated and can trace information flow throughout the application runtime. The modified byte code is written back to the class files of the application.

2. **tracing-phase**
   The input application is executed. The security monitor (see 3.3) is loaded and initialized according to the security policy. Control is then passed to the application. During execution every information flow sensitive operation is traced by the framework's flow analysis. The security monitor enforces that information flows only as specified by the security policy, the application doesn't leak sensitive information.

The workflow in Figure 3.1 shows how this information flow framework is used. In the first phase the instrumenter of the framework is used to instrument the input application. The settings for this phase are also located in the security policy. In the tracing phase the information flow analysis component of the framework analyzes and traces the flow of information and the security monitor enforces the security policy.

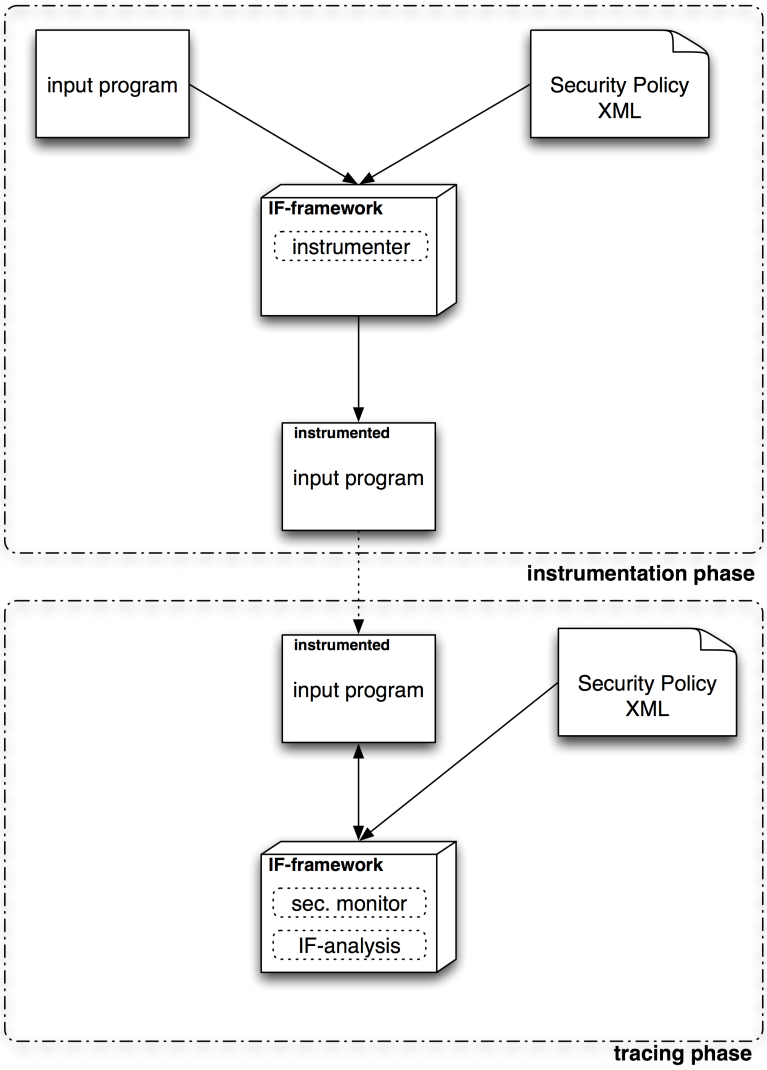


Figure 3.1: Workflow of the information flow framework. **Instrumentation phase:** The instrumenter component of the information flow framework (IF-framework, for short) processes the input program using settings from the security policy. **Tracing phase:** The instrumented program is executed, the IF-framework is initialised (the security policy is being parsed, the security monitor is set up) and control is returned to the input program. The information flow analysis component of the IF-framework traces all information flows. The security monitor is active and enforces the security policy, information leaks are prohibited.

3.2 Security Policy

The security policy defines the terms of usage. It defines proper and improper usage of information. The security monitor (see 3.3) enforces this policy during input program runtime. The security policy is specified in a xml document. It declares information flow sources (described in 3.5.1) and sinks (described in 3.5.2). It serves also as configuration file where

parameters as the following can be set as well:

- application starting points
  see 3.2.1 for details and declaration examples
- security monitor mode
  see 3.3 for available modes, is represented through a XML-attribute in the policy
- instrumentation mode
  see 3.4.4 for available modes, is represented through a XML-attribute in the policy
- information sources
  see 3.5.1 for details and declaration examples, see 2.4 for background on sources
- information sinks
  see 3.5.2 for details and declaration examples, see 2.6 for background on sinks
- level of logging
  sets which parts of the framework shall emit output, is represented through XML-attributes in the policy
- activate graphical output
  sets the output to be represented graphically, is represented through a XML-attribute in the policy
- activate debug mode
  sets the added instructions to be surrounded with NOPs [1] to be recognized better, is represented through a XML-attribute in the policy

Before processing the xml file is validated against a xml schema. As both phases (see 3.1) represent separated runtimes ,the security policy has to be processed in each phase. To assure that the xml file used for the tracing phase is the same used during the instrumentation phase a hidden hint file containing the filename is created in the filesystem[2].

3.2.1 Starting Points

A starting point represents the entry point of the input application. In general, this is a main method located in a certain class. There may be several starting points per application. In the instrumentation-phase (see 3.1) a reachability analysis is used to compute the reachable code of the application. To reduce the instrumentation overhead only reachable byte code is instrumented. The arguments provided to the starting point i.e. the entry point method - may already be a source of sensitive information. Therefore, tainting information about this arguments can be provided for every starting point. Figure 3.2 shows an example of a starting point definition where an array element is declared to carry sensitive information i.e. is a source[3].

[1] no operation instruction, doesn't influence the execution

[2] of course this solution isn't secure but this work's aim isn't to fulfil any security requirements

[3] tainting of single array elements is discussed in details in 5

```
<startingpoint>
   <classname>tests.EvilApplication</classname>
   <methodname>main</methodname>
   <methodsignature>([Ljava/lang/String;)V</methodsignature>
   <mainArgumentTaint mainArgumentCount="1">
      <argsArrayTaint arrayLength="2" arrayReferenceTaint="false">
         <arrayCell isTainted="true"/>
         <arrayCell isTainted="false"/>
      </argsArrayTaint>
   </mainArgumentTaint>
</startingpoint>
```

Figure 3.2: Example of a starting point definition in the security policy xml file. First array argument element is tainted i.e. is a information flow source.

3.3 Security Monitor

The security monitor enforces that the information used by the input application flows accordingly to the security policy (see 3.2). If sensitive information is going to flow to a sink (definition see 2.6, further details in 3.5.2) the security monitor prohibits this flow to avoid an information leak. Therefore, the monitor enforces the proper use of information, in other words it controls its usage. The security monitor is attached to the input application and avoids information leaks to sinks by intercepting calls of sinks i.e. methods. The information flow analysis provides the needed taint information to allow a decision to be made about the call. The semantics of the original application stays untouched. If the system is seen as a final state machine no new states or transitions are added. A security monitor checks every state transition before it is executed. If a state transition breaks the security policy the transition is prohibited[4]. The state transitions allowed by the security monitor have the same effects on the system state as before the application was instrumented.[5]

There are two selectable modes of operation for the security monitor.

- **Log-only**
  The security monitor logs every security policy (see 3.2) violation of an illegal information flow. It prints out a summary when execution of the tracing-phase (see 3.1) ends.

- **Exit-on-violation**
  The security monitor terminates the execution of the application on security policy violations and prints out information about the security violation.

3.4 Instrumentation Process

The instrumentation process is started in the instrumentation-phase (see 3.1). Two modes of instrumentation (see 3.4.4) are supported. Work items, which should be processed are kept in a process queue. This work items of the process are instrumentation jobs. An instrumentation job represents a single method that shall be instrumented. The queue is processed sequentially job after job. An instrumentation job is specified by the following attributes:

[4] depends on the selected mode of the security monitor

[5] there is no proof of semantic preservation in this thesis

- unique job id
- full qualified class name[6] (e.g. bytecode.instrument.InstrumentManager)
- method name (e.g. main)
- method signature (in Java byte code syntax e.g. $([Ljava.lang.String;)V$)

Initially for every starting point (see 3.2.1) defined in the security policy an instrumentation job is created and added to the process queue. An instrumentation job is processed as follows:

1. the corresponding class file is opened and parsed
2. the text segment i.e. the particular Java byte code implementation -, which matches the method name and signature is extracted from the class file[7]
3. the text segment is instrumented sequentially starting from the first instruction
4. the modified text segment is written back to the class file in the filesystem

Instrumentation jobs in the process queue are processed sequentially one after the other. To make sure no method is instrumented twice a separate book keeping is kept aside.

Instrumenting a method means altering its implementation not on the source but on the byte code level. Hence not the source of a method but only the class file is needed. Instrumenting Java byte code is similar to a binary rewrite of machine code. Altering a Java method means either adding new Java byte code instructions, removing certain instructions or changing existing instructions. Of course the altered byte code has still to pass the verification process of the Java virtual machine at startup, hence the modifications have to conform to this verification guidelines.

This work integrates information flow analysis into an input program. The instrumented byte code performs information flow tracing. As stated in 3.3 the security monitor enforces proper information flows without changing the semantics of the input program. Thus the original input program or more precisely the original text segment should only be extended to achieve the goals without altering the original semantics. These extensions simply implement something similar to a context switch in an operation system. Obviously no processes are switched, control goes simply from the input program to the information flow framework, where the information flow of the current byte code instruction of the input program is analyzed. On analysis completion control is returned to the input program. ”Switching” from the input program to the information flow framework is realized through the insertion of an invoke instruction - a so-called **hook** - calling the framework. A static method call is used because no receiver object is needed for the call, providing it would add complexity to this instrumenting schema, because the receiver object would have to be created before, what would increase the amount of newly inserted instructions. To speed up execution in the framework, for each byte code instruction a corresponding **handler** exists in the framework. The inserted hook directly invokes the right handler in the framework. The handler runs the information flow analysis, then returns to the input program.

[6] package names separated with dots, .class suffix omitted

[7] details omitted here, see 3.4.2 for further details

As mentioned in 3.3 the security monitor prohibits information leaks by intercepting calls of sinks. This interception is implemented with a handler for invoke instructions, which calls the security monitor.

This Java byte code instrumentation is done statically, ahead of execution. The mentioned code extensions are hooks to handlers in the information flow framework. To extract the needed information from the input program and to provide them to the framework over the hook, information is cloned and transferred through the arguments of the hook i.e. static invoke instruction. Chapter 4 explains this principle in more details. Figure 3.3 contains an instrumentation example.

```
; Jasmin source
;   package names and line numbers omitted
;   pseudo handler class names used
;   original instructions are annotated

; original byte code
iload_2
iaload
iadd

;#######################################

; instrumented byte code
iload_2           ;original
dup
iconst_2
invokestatic ;IFTracer.load_intType_Handler:(II)V
dup2
dup2
iaload
invokestatic ;IFTracer.iaload_Handler:(Ljava.lang.Object;II)V
iaload            ;original
iadd              ;original
dup
iconst_2
iconst_1
ldc_w ; "iadd"
invokestatic ;IFTracer.arithmeticInstr_Handler:(IIILjava.lang.String;)V
```

Figure 3.3: Instrumentation example. Original and instrumented Java byte code. Hooks to handlers through static invoke instructions, arguments of handlers pushed onto stack with additional instructions (iconst, dup, ...).

3.4.1 Reachability Analysis

As mentioned before (see 3.1) only reachable code of the application is actually instrumented to reduce the instrumentation overhead. The general problem of dead code elimination i.e. removing code that isn't reachable - is undecidable. The reachability analysis used in this work, solves only an approximation of this general problem. First of all exceptions are ignored and loops are assumed to always terminate. A method is reachable from another method, if control is transferred to this method through a method call. Method calls in Java byte code are represented by invoke instructions. Hence, the method text segment i.e. the particular Java byte code implementation - can be statically analyzed by simply identifying and following all

invoke instructions. The analysis is applied to every instrumentation job (see 3.4). For each such invoke instruction a new instrumentation job is created with the according attributes (see 3.4) and added to the process queue. Special cases of handling invoke instructions are discussed in the subsequent sections: 3.4.2 discusses how inheritance has to be handled and 3.4.3 explains the concept of polymorphism and how dynamic binding of methods is treated concerning instrumentation.

3.4.2 Inheritance

Java supports the concept of inheritance which generally spoken allows to reuse functionality from other existing classes. This useful feature applied very often in Java applications wants to be handled with special care. Because a particular method is not implemented but inherited from another class, the implementation of this method is located in the inheriting class, the so-called ancestor. Therefore, the text segment i.e. the particular Java byte code implementation - of the particular method specified in the instrumentation job (see 3.4) isn't necessarily located in the class file specified by the class name. To solve this issue the correct ancestor class has to be determined and the text segment taken from its class file. At the beginning of the instrumentation-phase (see 3.1) a type graph of the whole virtual machine class path is constructed. A graph node is a particular class and contains all implementing methods i.e. all methods that are really implemented and not inherited. An edge between two nodes represents the inheritance relationship. The ancestor node inherits its methods to the descendant node. This relationship is transitive. Because Java doesn't support multiple inheritance a descendant node may only have one ancestor node. The root of the type graph is the $Ljava.lang.Object$ class. Currently interfaces aren't handled. With help of this type graph the correct ancestor can be determined and the proper implementation can be loaded for instrumentation. Figure 3.4 shows an example type graph.

3.4.3 Polymorphism

Dynamic method binding is a concept applied in every object oriented language. Generally speaking the called method doesn't depend on the static type of the receiver object but on the dynamic type only known at runtime. Therefore several candidate methods for one particular method call could exist. Figure 3.5 contains a small example.

If only the method that matches the static type were considered in the instrumentation process (see 3.4), the tracing of information flow in the analysis would contain gaps because control flow could reach not instrumented methods. To avoid these gaps every method call is treated with respect to polymorphism.

As described in 3.4.1 for every invoke instruction a new instrumentation job is created. To cover polymorphism the type graph described in 3.4.2 is used to determine all matching polymorphic candidates for the method call. According to the concept of polymorphism program parts working with supertype objects work as well with subtype objects. Therefore all descendants of the current type graph node are traversed and every polymorphic match is added as a new instrumentation job. It is assumed that the class path is immutable during runtime and no classes are dynamically loaded over the network. Java allows overriding methods to have return type covariance i.e. overriding method return values may also be a subtype of the original method to be overridden e.g. $B :: foo()Ljava.lang.String;$ of class

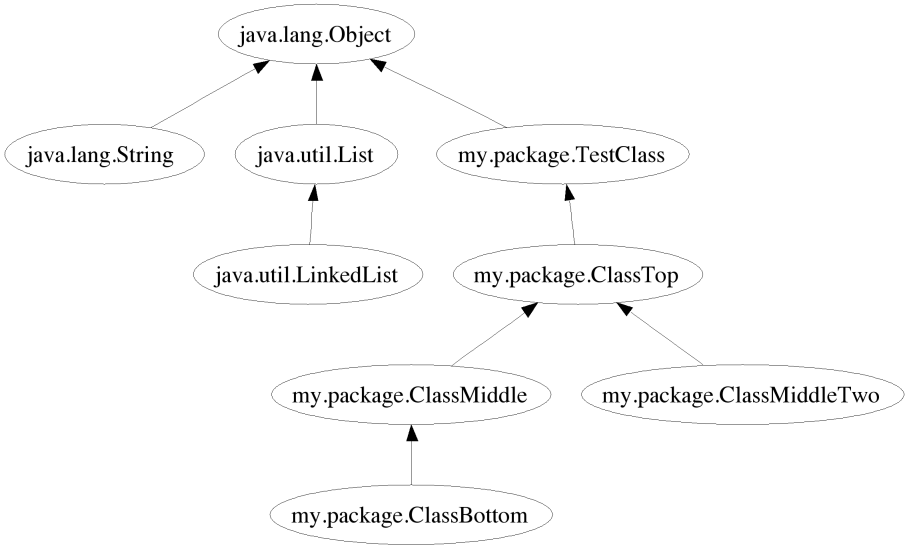


Figure 3.4: An example type graph. The directed edges represent the *extends*-relation in Java. Each node (class) contains all methods that are implemented by the node itself, hence they can be found in the corresponding class file.

```
// MyClassB extends MyClassA, MyClassB overrides foo()V
MyClassA instA = new MyClassA();
MyClassB instB = new MyClassB();
instA.foo(); // MyClassA::foo() called
instB.foo(); // MyClassB::foo() called
instA = instB;
instA.foo(); // MyClassB::foo() called
```

Figure 3.5: Example of dynamic binding.

B overrides $A :: foo()Ljava.lang.Object;$ of class A, if B extends A[8]. Currently this Java feature isn't handled by the framework, method signatures have to exactly match. A similar concept exists for fields i.e. class member - access. The current implementation doesn't handle them with respect to polymorphism, this issue can be avoided by using the concept of getter and setter methods[9]. Further limitations exist concerning Java visibility modifiers (e.g. protected, private, ...).

3.4.4 Instrumentation Modes

There are two selectable modes for the instrumentation-phase (see 3.1).

- **Explicit-only**

[8] $Ljava.lang.String;$ is a subtype of $Ljava.lang.Object;$

[9] adding the support for fields to the framework doesn't need any new concepts or data structures

The application is instrumented in order to integrate an explicit information flow analysis to trace explicit information flow (see 2.1 and 2.7).

- **Explicit-and-ImplicitONE**
  The application is instrumented in order to integrate an explicit information flow analysis together with an analysis for implicit information flow of the executing branch (see 2.8 and 2.9).

Currently there is no mode for integration of an analysis, which traces implicit information flows of the non-executing branch (see 2.8.2). With a selectable instrumentation mode the framework is much more adaptable to the range of real world applications and experiments, examining differences among the applied analysis, can be generated more easily.

3.4.5 Wrap-up

The instrumentation process is executed in the instrumentation-phase (see 3.1) before the actual information flow tracing in the tracing-phase.

The work items are instrumentation jobs representing single methods. Jobs are held in a queue and processed sequentially. The integration of the information flow framework is realized through hooks to handlers, which trace information flow. Java inheritance and dynamic binding of methods have to be treated with special care. Currently the analysis of explicit and implicit information flow of the executing branch can be integrated, implicit information flow of the non-executing branch can't be traced. At startup the instrumentation process completes the following tasks:

- load instrumentation setting from security policy
- build type graph of virtual machine class path
- create initial instrumentation jobs for each starting point (see 3.2.1)

Subsequently for each instrumentation job:

- load correct class file using the type graph (see 3.4.2)
- run reachability analysis (see 3.4.1), create further instrumentation jobs for each reachable method, handle polymorphic method calls (see 3.4.3)
- (in Explicit-and-ImplicitONE mode only) parse method text segment and construct control flow graph
- (in Explicit-and-ImplicitONE mode only) compute the post-dominator tree (see 6.2.5)
- (in Explicit-and-ImplicitONE mode only) identify all join points (see 2.9.1.1 and 6.2.6)
- (in Explicit-and-ImplicitONE mode only) insert code at conditional statements for for pc-label escalation (see 2.9.1.2)
- (in Explicit-and-ImplicitONE mode only) insert code at join points for pc-label de-escalation (see 2.9.1.2)
- write the instrumented method back to the class file[10]

[10] more detailed: to improve I/O performance all class files are kept in memory and are written back only at the end of the instrumentation process

3.5 Information Flow

This section describes how certain information flow concepts are implemented. A general overview over the topic of information flow can be found in chapter 2.

3.5.1 Source

As described in 2.4 a source represents a starting point of information flow. Sources can be specified in the security policy (see 3.2). Currently a source may be an object field i.e. a class member - or as mentioned in 3.2.1 a starting point method argument. A field source definition has the following attributes:

- full qualified class name[11] (e.g. bytecode.instrument.InstrumentManager)
- field name (e.g. taintedField)

An example of a source definition is shown in Figure 3.6. An example for a definition of a starting point method, which contains an argument that is a source, can be seen in Figure 3.2. The return value of a method could also be seen as a source of information flow. Currently the framework doesn't support such a source definition. This source declarations are static.

```
<source>
     <field>
        <classname>StandaloneArrayTest_1</classname>
        <fieldname>a_tainted_int</fieldname>
    </field>
</source>
```

Figure 3.6: Example of a sink declaration in the xml security policy.

At runtime there may exist several instances of a class. The information flow analysis simply checks all existing runtime instances against the static source declarations. If a field matches, it is marked to contain sensitive information, this work uses the term statically tainted for this case. Further taint propagation (see 2.5) flowing from such a statically tainted field i.e. source - is referred to be dynamically tainted.

3.5.2 Sink

As presented in 2.6 a sink represents an output channel. Sinks are mapped to Java methods. Usually, sinks are methods that neither can be investigated nor instrumented, and therefore they represent a "border" (see 2.9.3) for the information flow analysis. The assumption made here is that a sink may consume information only through method arguments. A call to a sink is intercepted by the security monitor (see 3.3). The security monitor fetches the taint information about the actual arguments from the information flow analysis. The reference taint information is obtained from the security policy (see 3.2). Actual taint is compared to the reference taint information. If there is one single missmatch the policy is violated. Figure 3.7 depicts a method call evaluation of the security monitor. The security monitor reacts to violations according to its mode of operation. Because information marked as sensitive is going

[11] package names separated with dots, .class suffix omitted

to leak through an argument that may not contain sensitive information. Sink arguments that may accept sensitive information are declared accordingly in the security policy. Usually, the implicit first argument $arg0$ is the reference to the receiver object, hence the security policy discards it and starts with $arg1$.
There are two ways of defining a sink in the security policy:

$call_1$ and $call_2$ are both calls of the same sink method consuming three arguments $(arg_1,\ arg_2,\ arg_3)$, 0 means not tainted, 1 means tainted:

	arg_1	arg_2	arg_3
$reference$ taint:	1	1	0
$call_1$ taint:	0	0	1
$call_2$ taint:	0	1	0

Figure 3.7: Example of taint comparison. The reference taint from the security policy allows the first and second argument to be tainted. $call_2$ is a call of the sink. The sink was called with the second argument being tainted. The security monitor allows $call_2$. $call_1$is also a call of the sink. The sink was called with the third argument being tainted. The reference taint from the security policy doesn't allow the third argument to be tainted. The security monitor prohibits $call_1$ because it violates the security policy.

- **plain sink definition:**
  a particular method specified through the following attributes
  - full qualified class name[12] (e.g. bytecode.instrument.InstrumentManager)
  - method name (e.g. main)
  - method signature (in Java byte code syntax e.g. $([Ljava.lang.String;)V$)
  - taint setting for the method arguments
    specifies which argument may be tainted in order to be allowed by the security monitor
- **pattern sink definition:**
  all methods matching a regular expression e.g. $java\backslash.lang\backslash..\star$, may not be called with any tainted arguments

An example showing both kind of sink definitions can be seen in Figure 3.8.

3.5.3 Taint Book Keeping

The general purpose of a taint book keeping is to record the incoming results form the information flow analysis (see 2.3). The security monitor sends queries to the taint book keeping to gather taint information needed for its decision making.

There are two kinds of taint entries, one for dynamic and one for static taint. **Static** taint (see 3.5.1) is declared in form of information sources in the security policy. **Dynamic** taint

[12] package names separated with dots, .class suffix omitted

```
<plainSinkDef>
      <classname>java.io.BufferedWriter</classname>
      <methodname>write</methodname>
      <methodsignature>(Ljava/lang/String;)V;</methodsignature>
      <allowedArgumentTaint argumentCount="1">
         <argumentTaint canBeTainted="false"/>
      </allowedArgumentTaint>
</plainSinkDef>

<patternSinkDef>
      <regexp>java\.net\..*</regexp>
<patternSinkDef>
```

Figure 3.8: Example of a sink declaration in the xml security policy.

results from taint propagation (see 2.5)[13]. In this work, only class fields can be declared as sources (see 3.5.1). A source i.e. a static taint - is represented by a book keeping entry containing the class name and the field name. Dynamic taint emerges from taint propagation (see 2.5) when information flows from a tainted variable to another variable at runtime. The latter variable (often referred to as entity) is a runtime instance and becomes tainted too. Therefore this kind of taint is called **dynamic**. A dynamic taint book keeping entry[14] consists of the field name and the object reference. This allows to distinguish between different instances of the same class.

So far book keeping entries within class scope have been discussed. To save taint information about local variables a local taint book keeping is used. Local variables in Java consist only of local variable indices ($0 - 0xff$), this is the only attribute, which has to be saved. The range of local variables is statically declared in each method header[15]. Therefore the local book keeping can be implemented as an array fitting into the range of the indices. Each method has its own local variables, hence the local taint book keeping is implemented as a stack of frames (similar to the virtual machine operand stack) and each frame has an array mentioned earlier. If a method ends, the frame is popped from the stack, if a new method is called, a new frame is pushed onto the stack.

The whole taint book keeping is implemented using lists and hash maps from the *java.util* package.

3.5.4 Inter-Method Information Flow Tracing

In order to trace information flow throughout methods the taint information has to be exchanged inter-procedural between methods. 3.5.3 described how class- and local-scope variables are handled. During method invocation method arguments are mapped to the local variables of the callee (first argument is first local variable, ...), hence the taint information about the arguments in the caller has to be transferred to the local variables in the callee. There are some approaches adding additional fields for this purpose, which are used as exchange container. Here a static class is used as exchange container. The class is designed in frames similar to the virtual machine operand stack. Each method has its own frame. An exchange works as follows. The caller stores taint information about each argument of the method to

[13] an exception are class fields declared as *static*

[14] array field book keeping entries are described in chapter 5

[15] in Jasmin the *.limit locals* statement is used

be called in the exchange class. It also provides unique identification attributes (classname, methodname, methodsignature, ...) about the callee. The callee is instrumented in such a way, that one of the first initialization tasks is the one which collects the taint information about the arguments (from the callee perspective they are local variables) from the exchange class. The callee has to provide his unique identification attributes to the exchange class. The exchange class checks, if the taint information are meant for this callee by comparing the stored attributes to the one provided by the callee[16]. The advantage of this additional checks is, that the right taint information reaches the right call target. Exchange mismatches can be detected. The same principle is applied to return values of methods.

[16] this check of course respects polymorphism(see 3.4.3)

4 Instrumenting Arbitrary Java Bytecode

This chapter deals with the problem of instrumenting arbitrary byte code i.e. code not necessarily generated by a compiler. An example shows why handwritten byte code doesn't fit into a general approach good enough for instrumenting compiler-generated code. The concept of a shadow stack is presented as solution to this problem. A summary with a short evaluation completes this chapter.

4.1 Problem Statement

If sensitive information flows to a sink, the information is leaking (see 2.7). To prohibit information leaks a security monitor (see 2.7.1 and 3.3) intercepts calls of sinks[1] and gathers taint information (see 2.5) about method arguments from the taint book keeping (see 3.5.3). The reference taint information are obtained from the security policy. Actual taint is compared to the reference taint (see 3.5.2). In order to provide taint information, the information flow analysis has to be integrated in the input program through byte code instrumentation (for an overview of instrumentation see section 3.4).

A method call consumes its method arguments from the operand stack. Hence the arguments have to be pushed before the actual method invocation. The top stack item is the last argument, the subjacent item is the penultimate argument and so on. Figure 4.1 shows the virtual machine operand stack layout of a method call. A Java byte code invoke instruction

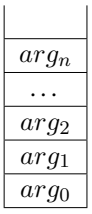


Figure 4.1: Virtual Machine operand stack layout for method calls. For dynamic methods arg_0 is the receiver object reference.

represents a method call of a certain method. The method arguments are pushed by certain instructions immediately preceding the invoke instruction. Given an input program generated by a compiler[2] the following simple approach could be used to determine the "argument pushing instructions". The argument count ($argc$, for short) can be extracted from the method signature and then the $argc$ preceding instructions before the invoke instruction are selected. For a sequence of instructions this means:

[1] in this work sinks are declared at the level of methods, for more details see 2.6 and 3.5.2

[2] more detailed: Java sources compiled to Java byte code

$$\overbrace{instr_1,\ instr_2,\ \ldots,\ instr_{argc},}^{selected\ instructions} \quad \underbrace{instr_{invoke}}_{consuming\ argc\ arguments}\ ,\ \ldots$$

This selected instructions together with the invoke instruction could then be instrumented in order to integrate the information flow analysis (see 2.3). The assumption here is that every other information flow carrying instruction is already properly instrumented and hence the information flow is traced without gaps from all sources to these method arguments. It is clear that such an approach could be realized with a very simple instrumentation schema and would require only few additional instructions.

The drawback is that it is depending on the order of the instructions and therefore on the strategy of the code generator used by the Java compiler. Dealing with **arbitrary code not generated by a compiler breaks this approach** as the example code in Figure 4.2 depicts.

```
; Jasmin source

aload_3      ; argument 0, receiver object
iconst_2     ; argument 1, integer constant
aload_0
getfield myField ; argument 2, string array class member
             ; start of interleaving block
   iconst_2
   iconst_2
   iadd
   istore 5
             ; end of interleaving block

invokevirtual ; method foo.MyClass.myMethod1(I[Ljava.lang.String;)V
return
```

Figure 4.2: Counter example. Simple approach is broken. Instructions in the interleaving block are selected wrongly.

The code is perfectly valid and accepted by Java virtual machine verifiers. The pitfall lies in the implicit assumptions on the order of the instructions. The computation in the interleaving block immediately preceding the method call leaves the operand stack in exactly the same state as if the interleaving block would have been omitted. The mentioned approach doesn't work because the approach selects the three[3] instructions preceding the invoke instructions and those aren't the ones which are responsible for pushing the method arguments. Thus, a more sophisticated approach is needed in order to handle arbitrary Java byte code.

4.2 Solution

A particular solution to this problem statement is described here. The easiest way of solving this problem would be to intercept the method call, freeze the virtual machine, and inspect the operand stack just before the call is executed. The arguments for the method call lie on the stack and they could be checked against the taint bookkeeping (see 3.5.3). A security monitor could then decide how to handle this particular call. Unfortunately, with an instrumenting approach there is no direct access to virtual machine components like the operand stack.

[3] the receiver object is also counted, it doesn't appear in the method signature

Subsequently, the only way to obtain this kind of information is to mirror or ”ape” the virtual machine. A so-called **shadow stack** is used to mirror the operand stack of the virtual machine. Each instruction that has impact on the operand stack's state is replayed on the shadow stack. Therefore the shadow stack is a clone of the operand stack.

To simplify the taint information retrieval and alleviate debugging the shadow stack items carry taint and meta-information which aren't included in a simple operand stack item. A particular instruction is replayed by the following instrumenting schema. Firstly all necessary meta-information (like local variable index, field name or type information) are collected and pushed onto the virtual machine operand stack using appropriate push instructions that are inserted before the instruction we are going to instrument. A method call to a static method consumes those data items from the operand stack as arguments. This static call invokes a handler, which replays then the particular instruction we are instrumenting on the shadow stack. This call is realized through the insertion of a so-called **hook** (see 3.4) calling this particular handler. A static method call is used because no receiver object is needed for the call, providing it would add complexity to this instrumenting schema because the receiver object would have to be generated before and that would increase the amount of newly inserted instructions. Every hook insertion in this framework uses static method calls.

Figure 4.3 contains an example of arbitrary Java byte code. The code is instrumented according to the presented approach using a shadow stack. Thanks to the shadow stack, the

```
; Jasmin source
;   package names and line numbers omitted
;   pseudo handler class names used
;   original instructions are annotated
;   myMethod1(I)V is declared as sink in the security policy

; original byte code
aload_3
iconst_2
invokevirtual ; foo.MyClass.myMethod1(I)V  ;call to sink

;#######################################

; instrumented byte code
aload_3      ; argument 0, receiver object
      dup
      iconst_3
      invokestatic ; IFTracer.loadHandler(Ljava.lang.Object;I)V ;shadowstack update
iconst_2     ; argument 1, integer constant
      dup
      invokestatic ; IFTracer.constHandler(I)V   ;shadowstack update
      invokestatic ; IFTracer.secMonitor()
invokevirtual ; foo.MyClass.myMethod1(I)V   ;call to sink
return
```

Figure 4.3: Original and instrumented Java byte code. Hooks to shadow stack updating handlers are inserted. The hook before the invoke instruction, which is calling a sink, invokes the security monitor, which checks the taint information of the method arguments kept by the shadow stack.

state of the virtual machine's operand stack can be retrieved even without direct access to it.[4]

[4]Note that the example above is slightly simplified. In the implementation of the information flow framework, the security monitor isn't invoked directly. A hook to a invoke instruction handler is inserted instead and it calls the security monitor

4.3 Summary

The presented approach doesn't rely on a particular code generation strategy of a certain Java compiler. Every byte code, no matter if compiler-generated or handwritten, can be instrumented in order to replay every instruction on a **shadow stack**, which basically is a clone of the immutable virtual machine operand stack.

When method calls are intercepted, the shadow stack can be used to check the taint information of the arguments consumed by the method, by simply examining the corresponding shadow stack items without any computation.

This approach is independent of the virtual machine. The shadow stack simply mirrors the current operand stack state of the virtual machine. The additional meta-information (like field name, typename or local variable index) provided by the shadow stack enriches the output of the system and helps to debug knotty errors related to corrupt operand stacks. This is also useful when certain information flow traces should be printed out to the user as human-readable logs. The implementation of the shadow stack contains consistency checks. The result of every *push* or *pop* operation is compared to the one of the real operand stack using Java *assert* statements. Hence shadow stack inconsistencies can be detected.

An important disadvantage is the amount of needed additional instructions inserted during the instrumentation process (see 3.4). Each argument that is provided to the handler has to be pushed onto the stack through an additional push instruction, besides the hook in form of an invoke instruction. Currently, the amount of used arguments is greater than actually needed, the additional unnecessary arguments are those carrying the mentioned taint and meta-information and the reference values for the consistency checks. This additional arguments enlarging the amount of additional instructions, could be avoided by simply adding a flag deciding if they are needed or not. Further comments can be found in 7.3.

5 Information Flow Analysis of Arrays

This chapter deals with the problem of tracing information flow throughout array data structures. First a problem statement introduces this topic and the levels of granularity are discussed. Then the solution is presented together with the selected level of granularity. A short summary sums up the findings.

5.1 Problem Statement

An array is a data structure that represents a group of elements. An individual array element can be accessed directly. Each array element has an index, used as key for the element access. Information can also be stored in such data structures. Therefore, information can flow to array elements. Hence an information flow analysis has to consider arrays too, in order to provide a information flow analysis without gaps.

The information flow analysis can operate on different levels of granularity. An array can be seen as one unit that either contains sensitive information or not i.e. the array is tainted (see 2.5 for tainted variables) or not. A finer grained analysis pays attention to the single array element. The applied level of granularity decides on the accuracy of the information flow analysis. A rough-grained level of granularity tends to produce too conservative decisions i.e. a particular flow can be declared to leak information where actually no information leaked. The result is that label creep i.e. the propagation of tainted entities floods the system - occurs much more probable.

An information flow analysis that checks array data structures only with respect to the array reference is rougher grained than one that checks single array elements. In a scenario where a tainted value is written into an array at a certain index the taint book keeping registers the whole array as tainted. A subsequent read of an array element at any index (even one where no tainted value are stored) yields a tainted value.

With an information flow analysis that handles single array elements, the same scenario results in more precise decisions, where only reads to indices of tainted array elements are actually tainted. In order to realize such a fine-grained information flow analysis for array data structures any instruction that deals with arrays needs to be instrumented.

5.2 Solution

A particular solution of this problem statement is described here. The following instructions dealing with array data structures were identified: creation of an array

- read access to a particular array element with a particular index
- write access to a particular array element with a particular index

- array length queries e.g. $array.length$

One of the byte code instructions that reads a particular array element is the *AALOAD*-instruction[1]. Figure 5.1 shows the outlook of operand stack for such an instruction. The

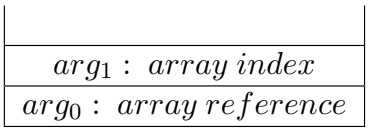


Figure 5.1: Operand stack layout of the *AALOAD* instruction

shadow stack (see 4.2), is a clone of the virtual machine operand stack, therefore we can retrieve the array reference and array index from it. The information flow analysis for array instructions uses the following instrumentation schema. An invoke instruction to a static method call - a so-called hook - is inserted. This hook calls a particular array load handler, which obtains the mentioned index and reference from the shadow stack and updates the taint book keeping. The same schema is applied to array store instructions like e.g. *AASTORE*.

5.2.1 Taint Book Keeping

The taint book keeping (see 3.5.3) treats array references as unique keys for particular arrays. For each stored array reference a taint structure is stored. It contains as many elements as the array, and each element in this taint structure corresponds to the array element and represent its current taint state (tainted or not tainted).

This design is consequently applied wherever arrays are used as data structures. Therefore method calls using arrays as arguments are working as well. In the security policy (see 3.2) a sink (see 2.6) can be defined at this level of granularity
e.g. $my.packagage.myFancyMethod(String[]\ arg_1)$ is a sink and a call of this method is only granted if the 3rd array element of the first argument isn't tainted. The policy has even a declaration for the *args* arguments of the main method (so-called starting points 3.2.1). This allows the framework to be used with programs that take sensitive data as command line input arguments.

5.2.2 Local and Global Arrays

An important point is the difficulty to distinguish between arrays with local and global i.e. class - scopes. Local variables declared in Java sources are mapped to byte code local variables with a unique ID between 0 and $0xff$. A global variable i.e. a class member also called a field - keeps the identifier declared in the Java source. Distinguish between a field reference that is an array and one that is not an array is quite difficult. Figure 5.3 depicts a write to a local and a global array. As it can be seen the code operating on the array field (a_global_ar) doesn't differ in the array instruction *IASTORE* but only in the way the array reference is stored and loaded.

In the case of the local array the *ALOAD* and *ASTORE* instructions are used in the case of the global array i.e. the array field - the *GETFIELD* and *PUTFIELD* instructions are

[1] details about Java byte code instructions see [10]

used. Therefore to distinguish local and global arrays the way the array references had been manipulated has to be inspected.

The framework solves this issue by separating global and local array taint book keeping. Those instructions loading references are instrumented and hooks to special handlers are inserted. A first version used *Java reflection* to detect array references, but after some performance test, this turned out to be a huge bottleneck. This part was replaced with a static analysis, solving this problem. More details about this issue are shown in 7.7.4.

If the loaded array reference has global scope the global taint book keeping for this reference is imported into the local array taint book keeping. All array manipulations take effect on the local array taint book keeping. If the array reference is written back using a field instruction it is clear the reference becomes global scope and therefore the local array taint book keeping is written back to the global array book keeping. If array references are only stored and loaded with the local instruction the array is treated as local and the local array taint is not written back.

```
int[] a_local_ar = new int[10];
a_local_ar[5] = 999;

a_global_ar = new int[10]; // declared as private int[] field
a_global_ar[3] = 666;
```

Figure 5.2: Java source code, manipulations on a global and a local array

```
25: bipush 10
27: newarray int ; new array created
29: astore_2
30: aload_2
31: iconst_5    ; index 5
32: sipush 999  ; new value 999
35: iastore     ; a_local_ar[5] = 999

36: aload_0     ; this reference
37: bipush 10
39: newarray int ; new array created
41: putfield    ; write field a_global_ar:[I
44: aload_0
45: getfield    ; read field a_global_ar:[I
48: iconst_3    ; index 3
49: sipush 666  ; new value 666
52: iastore     ; a_global_ar[3] = 666
```

Figure 5.3: Corresponding bytecode of Java source in Figure 5.2, the local array is mapped to the local variable index 2.

5.2.3 Information Flow in Arrays

Handling arrays at the level of single array elements widens the kinds of occurring information flows. Examples are the creation of a new array with the size variable being tainted or the querying of a tainted array reference for the current array length. The decision if the newly created array reference or the array length is tainted, should be dynamically adjustable.

Therefore, such information flow analysis decisions are not hardcoded in the handler but the security monitor is consulted to decide on particular array access. The security monitor can have several decision strategies that may be exchanged even during runtime. The different decision strategies can be selected in the security policy.

5.3 Summary

This approach provides information flow tracing throughout array data structures at the granularity level of single array elements. This enables security policies to be written at a very detailed level. A certain method can be defined as sink. Calls of this sink are only granted if the runtime taint information of single array elements are allowed according to the sink declaration e.g. *a call of this method is only granted if the 3rd array element of the first argument isn't tainted*.

Such fine-grained policies make the information flow analysis much more precise and help avoiding conservative analysis decisions what subsequently decreases the probability of label creep. The possibility to define custom information flow analysis strategies to decide on array access, adds to this point.

This approach profits from the shadow stack (described in 4.2), which contains the used taint information. Except the hook to the array taint handler, no additional instructions than those used to enable the shadow stack are needed. There is however some additional overhead in the array taint handler concerning the computations in the array taint book keeping facility.

6 Join Point Identification

In order to prohibit information to leak over implicit information flows, conditional statements have to be handled with special care. Implicit information flow in executing branches can be controlled with a pc-label described in 2.9.1. A security label for the program counter is escalated at tainted conditional branches and de-escalated at the corresponding join point. This chapter introduces the problem of identifying join points. Dominators are introduces together with all needed definitions. The immediate postdominator is then presented as a solution to this problem. Because of the analogy between dominators and postdominators, the definitions and algorithms for dominators are simply adapted to postdominators. A summary forms the end of this chapter.

6.1 Problem Statement

Given is a directed, cyclic graph representing control flow of Java byte code. Control flow in a control flow graph splits in two separate branches at conditional statements. Which of the branches is actually executed at runtime depends on the condition. If exceptions are disregarded and loops assumed to always terminate, the two separated branches will join back in exactly one node. This node is referred to as the join point of the corresponding conditional node[1]. Because conditional branches can be nested, several conditional nodes can have the same join point. To illustrate this fact Figure 6.1 and 6.2 compare two Java sources and the corresponding Java byte code control flow graphs both having different kinds of nested conditional branches. 6.1(a) and 6.2(a) have the same join point. 6.1(b) and 6.2(b) have two separate join points.

Loop expressions have also to be considered. A loop statement is mapped to a conditional statement i.e. the loop head - checking the loop condition. If the condition is true a jump to the loop body is performed. Figure 6.3 and 6.4 compare two Java sources and the corresponding Java byte code control flow graphs both having different kinds of loop expressions.

As for conditional branches a join point for loop expressions is a node, where the earlier separated conditional branches join back again. In a join point of a loop expression the following two branches meet:

1. loop body executed 0 times, condition never true

2. loop body executed at least 1 time, condition true at least once

[1] a node means a basic block in the CFG, a node of a conditional statement means the node or basic block, which contains this particular statement

```
int a = 100;
int res;
if (a <= 10)
{ // if-branch_1
      res = 2;
      if ( a < 0 )
      { // if-branch_2
        res = 21;
      }
      else
      { // else-branch_2
        res = 22;
      }
 // cont. if-branch_1

}
else
{ // else-branch_1
        res = 1;
}
res *=2;
```

(a) nested conditional statements with same join point

```
int a = 100;
int res;
if (a <= 10)
{ // if-branch_1
      res = 2;
      if ( a < 0 )
      { // if-branch_2
        res = 21;
      }
      else
      { // else-branch_2
        res = 22;
      }
 // cont. if-branch_1
      res += 20;
}
else
{ // else-branch_1
        res = 1;
}
res *=2;
```

(b) nested conditional statements with separate join points

Figure 6.1: Comparison of Java source with nested conditional branches

If the loop never ends or an exception is thrown the join point will never be reached. **This case is disregarded here**. As can be seen from the loop examples, determining join points isn't straightforward especially for loop expressions.

6.2 Solution

First of all the term join point should be clarified. A formal definition for join points will be presented. But before that the preliminary definitions should be given here.

6.2.1 Dominator

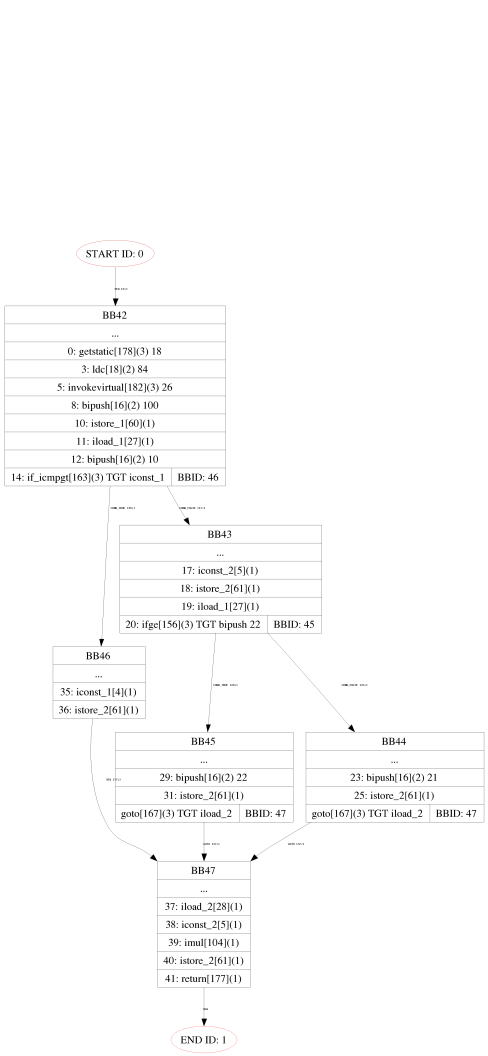
The following quotation defines the term dominator:

> We say node d of a flow graph dominates node n, written $d\ dom\ n$, if every path from the initial node of the flow graph to n goes through d. Under this definition, every node dominates itself, and the entry of a loop $[\ldots]$ dominates all nodes in the loop [1, p.602, sec. 10.3].

6.2.2 Dominator Tree

The following quotation defines the term dominator tree:

> A useful way of presenting dominator information is in a tree, called the dominator tree, in which the initial node is the root, and each node d dominates only its descendants in the tree [1, p.602, sec. 10.3].



(a) CFG of 6.1(a), $BB47$ same join point for $BB42$ and $BB43$

(b) CFG of 6.1(b), $BB41$ is join point for $BB35$, $BB40$ is join point of $BB36$

Figure 6.2: Comparison of control flow graphs corresponding to 6.1. **Note:** Due to compiler optimization the generated byte code can differ in the conditional jump instructions, the conditions are simply negated and hence the if and else branches are exchanged. This doesn't change the semantics!

Figure 6.5 shows an example. 6.5(a) is an example control flow graph and 6.5(b) is the corresponding dominator tree.

6.2.3 Immediate Dominator

The following quotation defines the term immediate dominator:

```
int res = 0;
for ( int i = 0; i < 10; i++ )
{
        res++;
}
```

(a) Simple Loop.

```
int res = 0;
int res2 = 100;
for ( int i = 0; i < 10; i++ )
{
        res++;
        for ( int j = 100; j > 0; j— )
        {
                res2 —;
        }
}
```

(b) Nested Loop.

Figure 6.3: Comparison of Java sources with loop expressions.

> The existence of dominator trees follows from a property of dominators; each node has a unique immediate dominator m that is the last dominator of n on any path form the initial node to n. In terms of the dom relation, the immediate dominator m has that property that if $d \neq n$ and $d\ dom\ n$, then $d\ dom\ m$ [1, p.602, sec. 10.3].

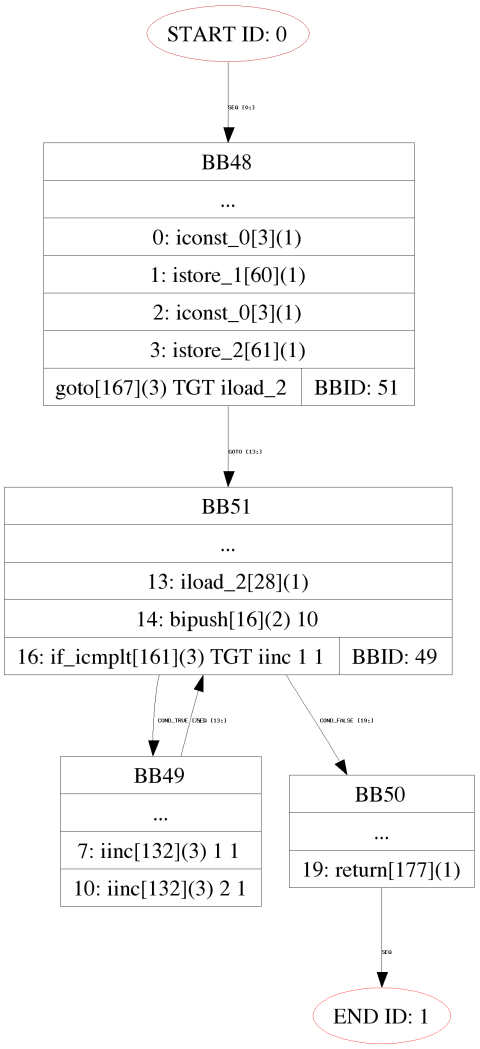
6.2.4 Dominator Algorithm

One way of computing dominators is to start from an approximation and to refine it step by step until a stopping criterion is met. These kind of algorithms are referred to as fix point computations that converge towards a fix point. For each graph node of the control flow graph the dominators have to be calculated. The set of dominators for a node n i.e. $D(n)$ - contains all nodes dominating n i.e. $\forall d \in D(n) \cdot d\ dom\ n$. The algorithm uses predecessors p_i of the current node n. Here these predecessors are all nodes of the control flow graph, which have a path to the node n or with other words, all nodes for which n is reachable. A good initial approximation is crucial factor for such computations. The root node of the control flow graph is called n_0. It is clear that $D(n_0)$ contains only n_0[2]. For all other nodes the initial approximation of $D(n)$ contains all nodes of the control flow graph. The principle of the algorithm is the following:

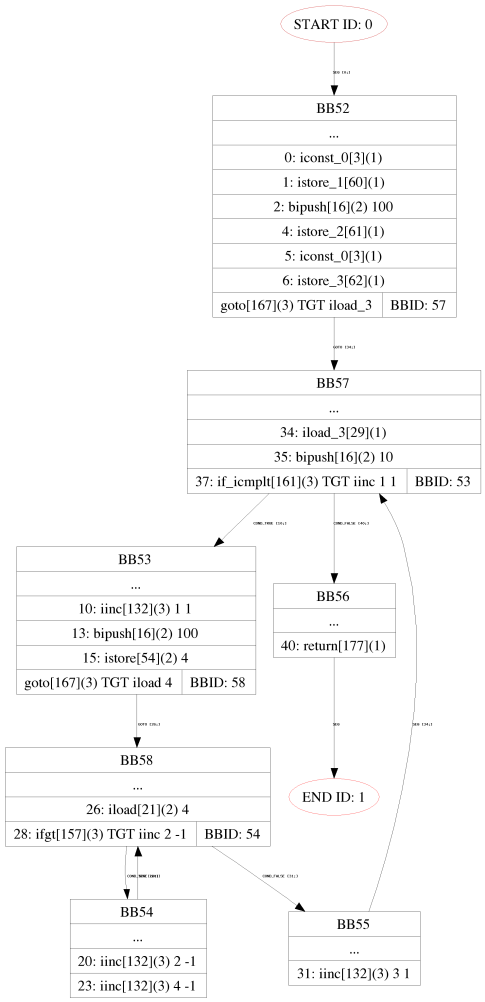
> $[\ldots]$ based on the principle that if $p_1, p_2, \ldots, p_k$ are all the predecessors of n, and $d \neq n$, then $d\ dom\ n$ if and only if $d\ dom\ p_i$ for each i.$[\ldots]$ we take an approximation to the set of dominators of n an refine it by repeatedly visiting all the nodes in turn. In this case, the initial approximation we choose has the initial node dominated only by the initial node, and everything dominates everything besides the initial node [1, p.670, sec. 10.9].

Figure 6.6 contains the dominator algorithm from [1, p.671, sec. 10.3] in pseudocode. In the first iteration the only changes of $D(n)$ sets will occur for those nodes having the root node as predecessor. These changes will propagate wavelike to the other nodes until no further changes are made. The initial approximation is finite because the amount of nodes in a control flow graph is finite. The computation converges because the new $D(n)$ at line 6 is always a subset of the old, right-hand site $D(n)$ and therefore the new set gets smaller towards the empty set.

[2] according to 6.2.1 each node is dominated by itself and n_0 has no predecessors (def. of root node)



(a) CFG of 6.3(a), $BB51$ is join point for $BB50$

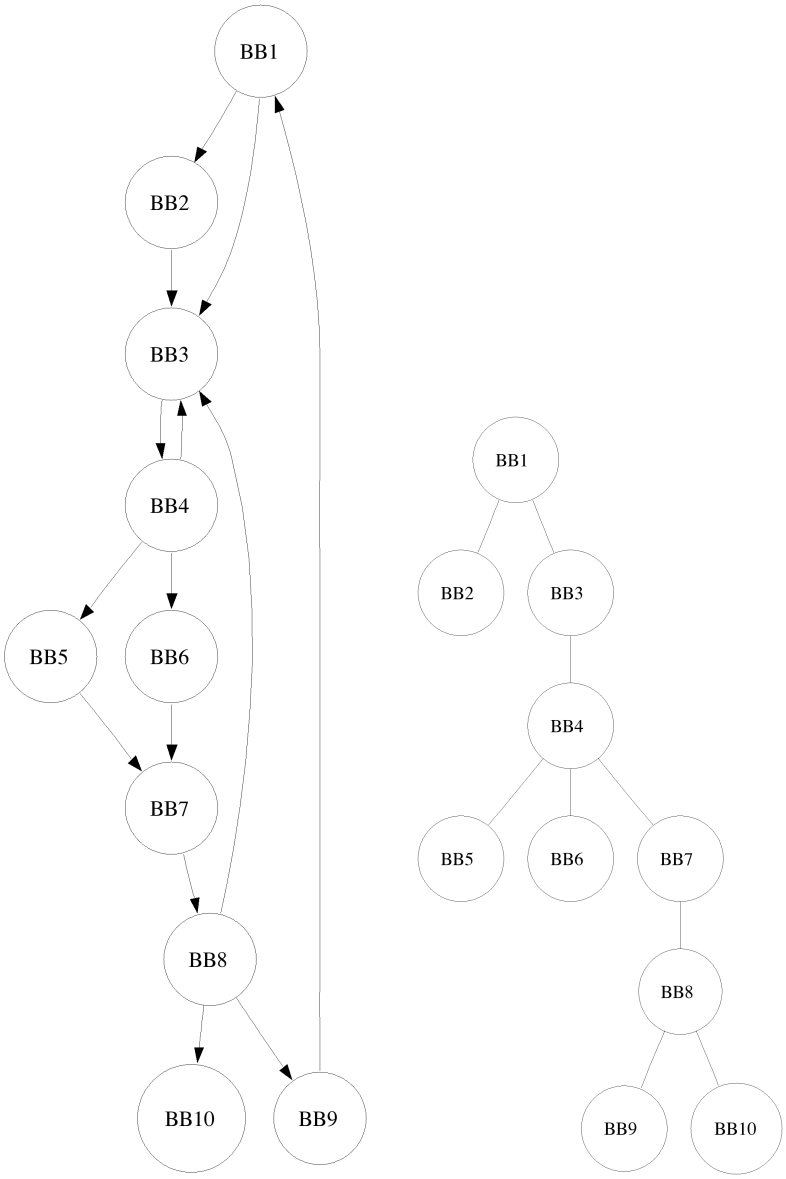


(b) CFG of 6.3(b), $BB55$ is join point for $BB58$, $BB56$ is join point for $BB57$

Figure 6.4: Comparison of control flow graphs corresponding to 6.3. **Note:** Due to compiler optimization the generated byte code can differ in the conditional jump instructions, the conditions are simply negated and hence the if and else branches are exchanged. This doesn't change the semantics!

6.2.4.1 Complexity

The algorithm has a time bound of $O(N^2)$, it is not the fastest algorithm. There are plenty of algorithms with better time bounds but they are more complex, difficult to implement and often there is a certain node count threshold from where on the algorithm starts to perform better. Only a few known algorithms are mentioned here for the interested readers (E means the amount of edges and N of nodes). Tarjan[12] proposed an algorithm with a time bound of $O(NlogN + E)$. Later Tarjan and Lengauer[8] developed an algorithm that runs in



(a) Control flow graph (b) Corresponding dominator tree

Figure 6.5: Control flow graph and dominator tree example from [1, p.603, sec. 10.3]. **Note:** Due to compiler optimization the generated byte code can differ in the conditional jump instructions, the conditions are simply negated and the if and else branches are exchanged. This doesn't change the semantics!

$O(E * log(N))$. An even faster algorithm and a good summary on the history of dominator algorithms were presented by Cooper et al.[4].

1: $D(n_0) := \{n_0\}$;
2: for n in $N - \{n_0\}$ do
3: $D(n) := N$;
$\backslash\star$ end of initialization $\star\backslash$
4: while changes to any $D(n)$ occur do
5: for n in $N - \{n_0\}$ do
6: $D(n) := \{n\} \bigcup \bigcap_{p \ a \ predecessor \ of \ n} D(p)$;

Figure 6.6: Dominator algorithm in pseudocode.

6.2.5 Postdominator

As announced in the beginning of 6.2 a formal definition of join points should be given. It will be shown that a join point of a conditional statement is equal to its **immediate postdominator**. The following quotation gives a first general statement about this new term.

> The immediate postdominator of an instruction (ipd, for short) in the control flow graph of the program represents the first instruction common to all the execution paths starting at such instruction.[2]

The first common instruction of a conditional statement in a control flow graph is exactly what needs to be identified as a join point. Unfortunately, the term join point is used in different contexts[3] and can have slightly different meanings. To avoid ambiguity and confusion the term **immediate postdominator** will be used instead from now on.

There is a strong analogy between dominator and postdominator. Subsequently, the dominator related definitions will be adapted for the postdominator. The following list contains all the definitions presented so far adapted to the postdominator:

- **Postdominator**
  The definition from 6.2.1 changes to:
  We say node d of a flow graph *postdominates* node n, written $d \ pdom \ n$, if every path from n to the *final* node of the flow graph goes through d. Every node *postdominates* itself.

- **Postdominator Tree**
  The definition from 6.2.2 changes to:
  A useful way of presenting dominator information is in a tree, called the *postdominator* tree, in which the *final* node is the root, and each node d *postdominates* only its descendants in the tree.

- **Immediate Postdominator**
  The definition from 6.2.3 changes to:
  Each node has an unique *immediate postdominator* m that is the *first postdominator* of

[3] e.g. in aspect oriented programming

1: $PD(n_{final}) := \{n_{final}\}$;
2: for n in $N - \{n_{final}\}$ do
3: $PD(n) := N$;
\⋆ end of initialization ⋆\
4: while changes to any $PD(n)$ occur do
5: for n in $N - \{n_{final}\}$ do
6: $PD(n) := \{n\} \bigcup \bigcap_{p \ a \ successor \ of \ n} PD(p)$;

Figure 6.7: Postdominator algorithm in pseudocode.

n on any path form n to the final node. In terms of the $pdom$ relation, the *immediate postdominator* m has that property that if $d \neq n$ and $d \ pdom \ n$, then $d \ pdom \ m$.

- **Postdominator Algorithm**
  The set of postdominators for a node n i.e. $PD(n)$, contains all nodes that postdominate n i.e. $\forall d \in PD(n) \ \cdot \ d \ pdom \ n$. The algorithm uses successors s of the current node n. These successors are all nodes of the control flow graph, which are on all paths from the node n to the final node or with other words all nodes, which are reachable from n. The computation principle (see 6.6) stays the same. The initial approximation changes to:
  The final node of the control flow graph is called n_{final}. It is clear that $PD(n_{final})$ contains only n_{final}[4]. For all other nodes the initial approximation of $PD(n)$ contains all nodes of the control flow graph. Figure 6.7 contains the postdominator algorithm in pseudocode.

The convergence criteria and the performance for the postdominator algorithm is the same as for the dominator algorithm. Dominator definitions can also be found in e.g., [2], [12].

6.2.6 Join Point Identification

As presented in 6.2.5 the join point of a conditional statement is equal to its immediate postdominator. Therefore, the problem of identifying join points is reduced to the problem of finding immediate postdominators. This work uses the postdominator algorithm presented in 6.7 which is based on the dominator algorithm (see 6.6). A postdominator tree is constructed from the resulting postdominator sets i.e. $PD(n), \ n \in N$. The immediate postdominator is the parent node i.e. the direct ancestor - of the node containing the conditional statement in the postdominator tree. Figure 6.9 and 6.10 shows an example of a dominator tree and a control flow graph with annotated join points.

The Postdominator tree 6.9 and control flow graph 6.10 are built for the example in 6.8. **Note:** Due to compiler optimization the generated byte code can differ in the conditional jump instructions, the conditions are simply negated and hence the if and else branches are exchanged. This doesn't change the semantics!

[4] according to 6.2.5 each node is postdominated by itself and n_{final} has no successors (def. of final node)

```
int a = 100;
int res;
if ( a > 10 )
{
res = 1;
}
else
{
    res = 2;
    if ( a < 0 )
    {
        res = 21;
    }
    else
    {
        res = 22;
    }
}

if ( res > 0 )
{
    for ( int i = 0; i < 100; i++ )
    {
        a = res + 10;
    }
}
else
{
    a = a*2;
    res = res - a;
}
res = res + a;
```

Figure 6.8: Java source code example

6.2.7 Loops

Figure 6.4 presented a loop example and 6.1 stated that identifying join point isn't straightforward especially for loops. The definitions and algorithms presented in this chapter work for all kind of control flow graphs of Java byte code. It might be confusing that the back-pointer in a loop doesn't have to be handled exceptionally. The only point where back-pointers play a special role, is in the algorithm (see 6.7) where successors are computed (see line 6 in 6.7). To avoid that cycles are visited infinitely, nodes are marked as visited. The algorithm iterates sequentially over the nodes of the control flow graph and back-pointers don't need any

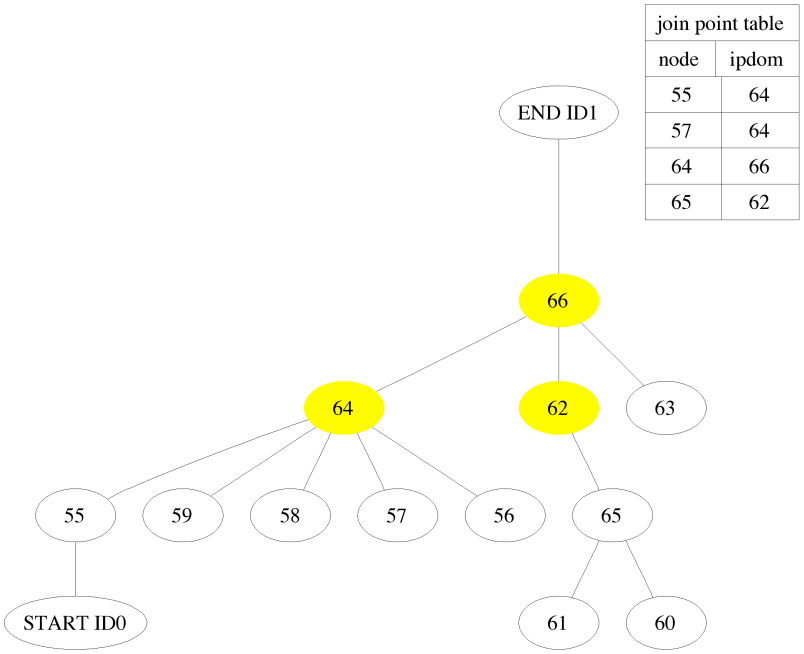


Figure 6.9: Postdominator Tree, immediate postdominator (ipdom, for short) marked yellow, join point table with node ipdom correspondence.

exceptional handling[5].

6.3 Summary

In a control flow graph the immediate postdominator of a conditional statement is the point where conditional branches join back again. All needed definitions of postdominator related terms were presented as analogies of dominator terms. The dominator algorithm was slightly changed to yield postdominators. The final identification of a join point i.e. immediate postdominator - is proceeded in the postdominator tree. There, the parent node i.e. the direct ancestor - of the conditional branch containing node is the wanted immediate postdominator. The used algorithms are the easiest to understand, but not the fastest ones. The trees are needed in the instrumentation phase 3.1, hence execution time in the tracing phase isn't influenced.

[5]There are papers discussing the general role of back-pointers or fronds, as they are often referred to, in the dominator relation. In certain problem statements fronds don't influence the dominator relation. Tarjan presented an algorithm [12] to compute dominators in directed graphs. This algorithm reduces directed, cyclic graphs to directed, acyclic graphs using dominator-preserving transformations. Tarjan provided a proof that these transformations have no effect on the dominator relation.

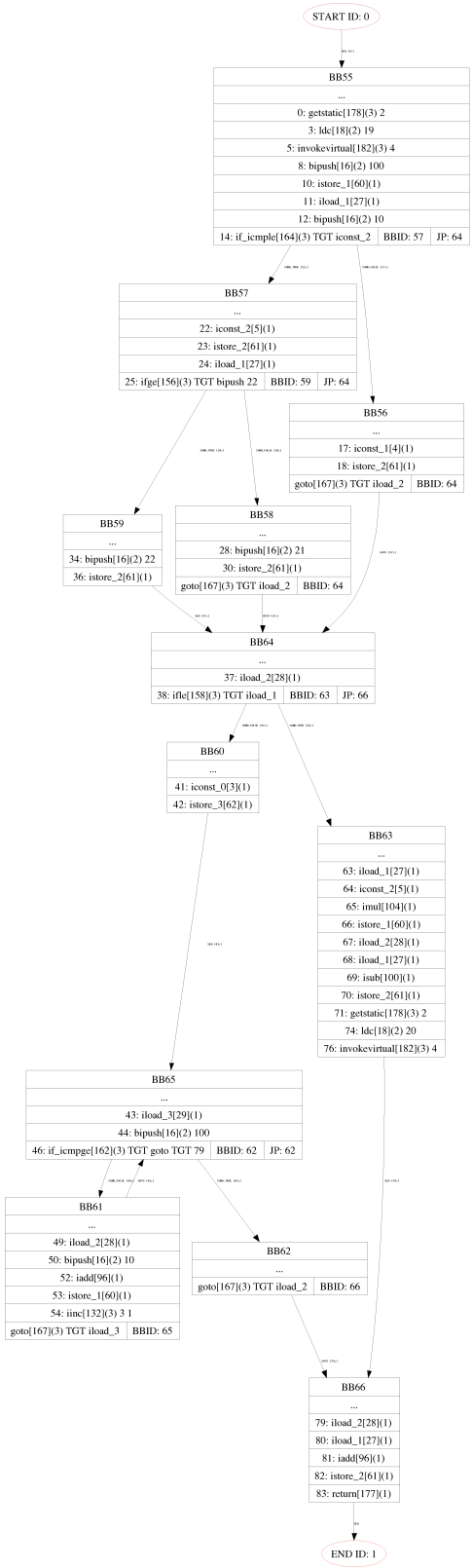


Figure 6.10: Control flow graph with join points (JP, for short).

7 Evaluation

In this chapter the implemented framework is evaluated. The achieved results are presented together with encountered throwbacks. Every point is commented listing advantages and disadvantages. Suggestions for improvements are mentioned too.

7.1 Java Byte Code Instruction Set Coverage

Unfortunately not all 256 Java byte code instruction could be handled by the current implementation. Therefore a smaller subset is defined, which represents the most used instructions in simple example programs. Details about Java byte code instructions can be found in[10]. An overview over the Java virtual machine can be found in [13].

7.1.1 Supported Instruction Subset

The following listing contains the handled instruction set. These instructions are handled by the current implementation:

- local variable instructions:
  ILOAD, ILOAD$_{<n>}$, ALOAD, ALOAD$_{<n>}$, ISTORE, ISTORE$_{<n>}$, ASTORE, ASTORE$_{<n>}$
- constant instructions:
  ICONST, LDS, ACONST_NULL, BIPUSH, SIPUSH
- field instructions:
  GETFIELD, PUTFIELD
- static field instructions:
  GETSTATIC, PUTSTATIC
- conditional branch instructions:
  all conditional jumps like *IF_ICMPEQ, ...*
- unconditional jump instructions:
  GOTO
- invoke instructions:
  INVOKEVIRTUAL, INVOKESPECIAL, INVOKESTATIC
- array instructions:
  AALOAD, AASTORE, ARRAYLENGTH, ANEWARRAY, NEWARRAY
- return instructions:
  ARETURN, IRETURN, RETURN, RET

- arithmetic instructions:
  all integer arithmetic instructions like *IADD, IINC, IMUL, ...*
- stack instructions:
  DUP, POP

7.1.2 Unsupported Instructions

The following instructions aren't handled yet. There are missing in the current implementation. Programs containing these instructions can't be instrumented without gaps in the information flow analysis (see 2.3). The missing instructions are:

- local variable instruction for floats, doubles and long integers:
  CALOAD, CASTORE, DALOAD, DASTORE,...
- conversion instructions:
  e.g. convert double to float *D2F*, ...
- arithmetic instructions:
  all arithmetic instructions operating on doubles, floats, long integers e.g. *DDIV,...*
- conditional branch instructions:
  TABLESWITCH, LOOKUPSWITCH
- special double, float, long integer instructions:
  DACMPG, DNEG
- constant instructions:
  FCONST_<F>
- invoke instructions:
  INVOKEINTERFACE
- return instructions:
  DRETURN, FRETURN, LRETURN
- stack instructions:
  DUP2, DUP2_X1, DUP2_X2, DUP_X, DUP_X2, SWAP
- array instructions:
  MULTIANEWARRAY
- other:
  ATHROWBREAKPOINT, WIDE, MONITORENTER, MONITOREXIT

Lots of the missing instructions are versions of the instructions in the supported subset, which are based on other primitive types like long integers, floats or doubles. Writing handler code for these versions shouldn't be a big deal, because of the similarity to the integer based instructions. In case of the instructions based on doubles, two instead of one stack element are used. But there are also other types of instruction handlers that need more time to be implemented.

7.2 Information Flow Analysis

Information flow analysis (see 2.3) of Java byte code could be successfully applied to input programs using only instructions contained in the subset defined in 7.1. Information flow could be traced over the distinct sequences of instructions. The test suite contains test cases where rather long information flow sequences were tested. An example for such a sequence is e.g. field $\mapsto$ local variable $\mapsto$ stack item $\mapsto$ method argument $\mapsto$ polymorphic method call $\mapsto$ local variable $\mapsto$ array element $\mapsto$ library call $\mapsto$ return value. The taint book keeping recorded the tainted variables resulting from such taint propagation.

7.2.1 Library Calls

In 2.9.3 library calls represent control borders. Actually, this is only true if the called library method can't be instrumented. Instrumenting libraries is slightly different from ordinary code, because libraries are expected to be at certain class path locations. Either the original library files are exchanged with the instrumented ones, or the library class path is changed at start-up. A full instrumentation of a library wasn't achieved. The reduced instruction set (see 7.1) is too small. The complexity of library classes was underestimated. Even a $Ljava.util.LinkedList$; implementation uses instructions that aren't in the instruction set.

The lack of exception handling turned out to be the biggest problem, nearly every library class deals with it. In order to overcome this problem (first of all) the supported instruction set has to be extended. Adding exception handling would not only mean to instrument the *ATHROW* instruction but to extend the control flow graph (and also the byte code parser) and to adjust the information flow analysis to trace information flow throughout exceptions and exception catch-blocks.

Because library methods couldn't be instrumented, they were declared as sinks in the security policy. Sink declarations (see 3.5.2) in the security policy allow to define reference taint for each argument separately. The granularity of such definitions turned out to be fine enough to write tests using such calls. Because the called library methods weren't instrumented, certain assumptions about the information flow had to be made e.g. string concatenation "abc"+"'def' = "abcdef", the argument string ("def") and the receiver object ("abc") flows only to the return value ("abcdef").

Information leaks could be detected in example programs, with some restrictions e.g. all Java library methods for writing data to the filesystem were declared as sinks. File writing attempts with input data marked as tainted could be successfully prohibited.

7.2.2 Inter-Method Information Flow Tracing

The design details about how taint information are exchanged between methods are discussed in 3.5.4. The frame design and the additional checks allowed to recognize defects in the implementation very early. A defect in the exchange between two methods doesn't necessarily lead to an error but the analysis deliver s wrong results. Finding such defects can only be done manually by evaluating information flow traces, which is quite tedious. These checks saved lots of time. A disadvantage is of course the performance. The checks need time and the access to data in the proper frame is also time intensive. In the worst case a check has to compare all polymorphic candidates (see 3.4.3) in the type graph(see 3.4.2). To improve the performance these checks could be controlled over flags. Because the checks are not invoked

by additional instructions in the byte code, but are started by the exchange class, this could be realized very easily.

7.2.3 Array Tainting

The information flow analysis on array element level described in chapter 5 opened up a whole bunch of new features. First of all, the information flow tracing of single array elements is working pretty fine. What is missing, is the support for multi-dimensional arrays. Because arrays could be tainted on single element level, the security policy sink declarations could be done on a much finer level of granularity e.g. *calls of a sink may be prohibited only if the third array element in the second method argument is tainted*. A further feature is the starting point declaration (see 3.2.1). The main arguments may of course also represent sources of sensitive information. The possibility of declaring starting points with the corresponding taint information allows to run input applications using command line arguments. This feature increases the usability and broadens the range of appliance to real-world applications.

The integration of the information flow tracing is quite smooth but the taint book keeping for the arrays is quite complex due to the problem of distinguishing between local and global arrays (see 5.2.2). Further studies of this particular problem could help to find a simpler solution. Achieved improvements of performance are presented in 7.7.4).

The applied design (see 5.2.3) to allow different array information flow strategies to be implemented and selected in the security policy, makes integration of other information flow analysis of arrays very comfortable and easy.

7.3 Shadow Stack

The shadow stack is working properly. Not only the ordinary stack items are displayed but additional meta information. This turned out to be extremely helpful in tedious debugging cases. The virtual machine doesn't print out much information about the location of an error in the byte code source. Additional byte code outline tools like eclipse plugins turned out to have troubles displaying instrumented code[1]. Using traces from the shadow stack helped to reconstruct the stack state before the actual error. It was also a good way to delve into the inner life of the virtual machine and to see how the instructions are carried out and impact on the stack.

The biggest disadvantage is the enormous code expansion, which influences the performance. As mentioned in chapter 4 every additional portion of meta information has to be pushed onto the stack in order to arrive to the shadow stack. Another problem are complex instructions that don't simply impact on top stack elements, but rather on references. An example is the object construction. Figure 7.1 depicts an object creation example. Figure 7.2 shows the corresponding virtual machine operand stack layouts. The *NEW* instruction simply creates a not initialized reference. The constructor allocates memory for the object and initializes it. Afterwards, the reference points to the initialized memory i.e. the created object in the heap.

[1] only a couple of freely available tools were tested

```
Object objref = new Object(); // local variable
```

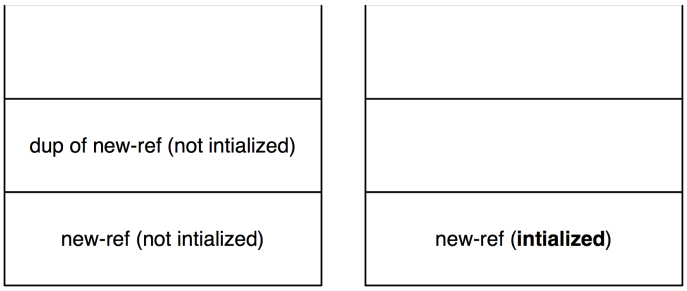
(a) Example object creation in Java.

```
;      Jasmin source

0:  new          ; Ljava.lang.Object
3:  dup
4:  invokespecial ; Ljava.lang.Object."<init>":()V
7:  astore_1
```

(b) Corresponding Java byte code.

Figure 7.1: Object creation in Java and in Java byte code. The call of $< init > ()V$ initializes an object reference.



(a) Virtual machine operand stack after line 3 and before call to $< init > ()V$ at line 4.

(b) Operand stack after call to $< init > ()V$ at line 4.

Figure 7.2: Virtual Machine operand stack before and after the constructor call of $< init > ()V$. The *new-ref (not initialized)* in 7.2(a) is duplicated (using the *DUP* instruction) before the call to $< init > ()V$ because it consumes the top of the stack as receiver object. $< init > ()V$ initializes the memory to which the receiver *dup of new-ref (not initialized)* points to. Hence the *new-ref (not initialized)* stack item, which lays underneath the receiver object is initialized as well, because it is the same reference. After the call this reference is initialized as shown in 7.2(b).

The shadow stack would have to use a point-to / alias analysis to determine, which stack items needs updating. Running such an analysis on the whole stack for each constructor call would certainly worsen the performance. Therefore, the current implementation doesn't update such new-references.

7.4 Join Point Identification

Chapter 6 discussed ways to find join points. The immediate postdominator (see 6.2.5) is calculated using a slightly changed dominator algorithm. The algorithm has a time bound of $O(n^2)$ (n are the nodes or rather basic blocks of the control flow graph). Faster algorithms exist but they are far more complex. The computations are done in the instrumentation phase (see 3.1), hence the time to build the control flow graph, the postdominator tree and to find the immediate postdominator has to be paid once ahead of execution. During the tracing phase this kind of data structures aren't needed.

7.5 Compatibility

The framework is completely independent o Java virtual machines. Input programs are provided as compiled Java source code or as handwritten byte code[2]. The test suite (see 7.6) was executed with the Sun™Java Virtual Machine v1.6/v.1.5 under Linux and Mac OS X. More details about executed tests can be found in 7.7.

7.6 Testing

A test suite was built containing nearly 200 unit tests. In order to evaluate test cases automatically a test oracle has to determine the correctness. Because some of the test values are runtime dependent, they can't be defined in testcases. An example is the taint book keeping presented in 3.5.3. There, a dynamic taint entry contains an object reference, which is only known at runtime. To solve this problem a callback is inserted in each test case and a second taint book keeping is created containing the expected results. The callback provides the missing runtime values to the second taint book keeping. Hence the test oracle can use it to automatically judge the test results.

With growing testsuite the implementation process became more efficient. The test suite was especially used for regression testing. Work in the context of byte code instrumenting is quite complex. Especially inserting instructions to prepare the operand stack for a method call turned out to be quite error-prone. Having a good test coverage on instruction handlers is a key issue in such low level projects. Performance tests are discussed in 7.7.

7.7 Performance

This section presents the results of performance assessment of the information flow framework. The execution time is measured. First the measuring methods are discussed and then the test setup. Finally, the test results are presented.

7.7.1 Measuring Methods

To measure the performance of the information flow framework two measuring tools are used. The *time – time command execution* (**time**, for short)[3] - is used to measure the total execution time of a Java test program. To measure execution time from the inside of the Java program, the system variable *System.currentTimeMillis()* (**java time**, for short) is used. Figure 7.3 depicts how the execution time of the original program is measured.

time is used to measure the whole execution of the original input program, this includes the virtual machine initialization and tear down. **java time** starts measuring at the beginning and stops at the end of the *main()*-method, this is the pure execution time without any virtual machine initialization or tear down.

Figure 7.4 depicts how the execution time of the instrumented input program in the tracing phase (see 3.1) of the information flow framework is measured. The instrumented input program measured with **time** yields the total execution time including virtual machine (vm,

[2] compiled with the Jasmin compiler

[3] used the BSD version under MacOS X, and the GNU version under Linux

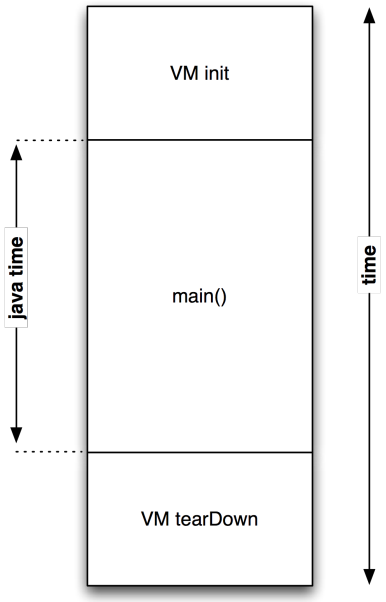


Figure 7.3: Measuring execution time of the original input program using **java time** and **time**.

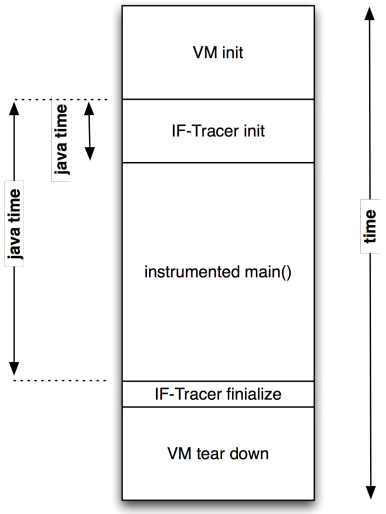


Figure 7.4: Measuring execution time of the instrumented input program in the tracing phase of the information flow framework (IF-Tracer, for short). **java time** is used twice here. Once to measure the IF-Tracer initialization and once to measure the whole execution time of the instrumented program including IF-Tracer init.

for short) initialization (init, for short) and tear down. In this phase the information flow framework (IF-Tracer, for short) is initialized first. The security policy file is loaded and parsed, the security monitor is created and initialised. Subsequently control is returned to the instrumented input program. After the termination of the input program's *main()*-method,

the finalizer of the IF-Tracer prints out a summary (policy violations, log, ...) and cleans up the IF-Tracer. The whole java program ends after the finalizer ends. **java time** is used twice here, once to only measure the initialisation of the IF-Tracer and once to measure the complete computation time of the instrumented input program without IF-Tracer finializer (and without vm init and tear down).

7.7.2 Test Setup

The performance test uses four different input programs. The following four sorting algorithms are used: Bubblesort, Insertionsort, Mergesort, Quicksort. In a test round an input program sorts each input set. The following input sets are used:

- **random**
  random values within a specified range, starting from a specified seed, generated with *java.util.Random*

- **constant**
  the whole set consists of the same value

- **ascending**
  ascending sequence starting at a specified value

- **descending**
  descending sequence starting at a specified value

The generation of the input set happens inside of the input program's *main()* method. The cardinalities of the input sets are: 1000, 2000, 2500. In a test round the original untouched input program sorts the mentioned input sets. Then the original input program is instrumented and started again to sort the input sets. Each test round is executed 10 times. The execution time is measured using methods presented in 7.7.1. From the data collected from the 10 test rounds, the mean is taken for each measured time value.

The machines used to run the tests are the following:

- MacBookAlu5,1
  Intel Core 2 Duo, 2.4 GHz, 3 MB L2 Cache, 4GB RAM
  Mac OS X 10.5.6 (9G55), Sun^{TM}Java SE Runtime Environment 1.6.0.07

- MacBook4,1
  Intel Core 2 Duo, 2.4 GHz, 3 MB L2 Cache, 2GB RAM
  Mac OS X 10.5.6 (9G55), Sun^{TM}Java SE Runtime Environment 1.5.0

- IBM Thinkpad R32
  Intel 1.8 GHz, 512MB RAM
  Ubuntu Linux, Sun^{TM}Java SE Runtime Environment 1.5.0

The security policies for the input programs are all identical containing one declared source used once in each input set. The instrumentation mode (see 3.4.4) is set to "Explicit-and-ImplicitONE". The security monitor mode (see 3.3) is set to "log-only".

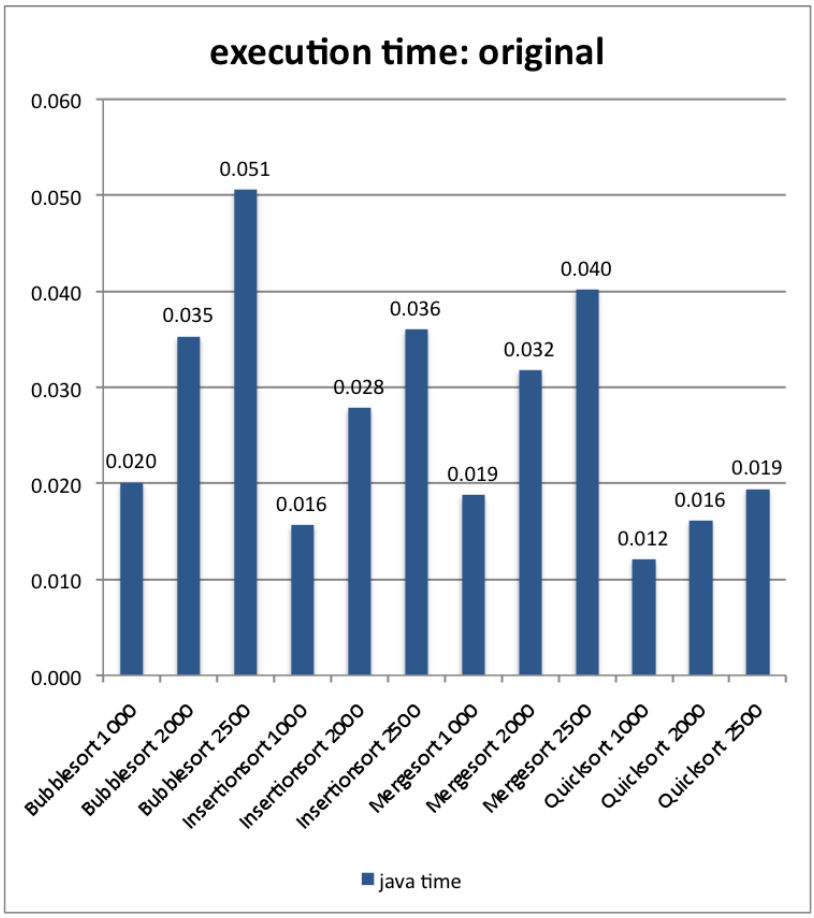


Figure 7.5: Execution time of original input programs, measured with **java time**.

7.7.3 Test Results

The presented results are those from the first machine mentioned above. The results from the other machines were quite similar concerning the computed factors[4]. All time values are in seconds.

7.7.3.1 Execution time of original input programs

Figures 7.5 and 7.6 contain the execution times of each original input program.

7.7.3.2 Execution time of instrumentation phase

Figure 7.7 contains the execution times of the instrumentation phase for each input program.

[4] it is clear that the execution times of the last machine were much higher, but the overhead factors were similar

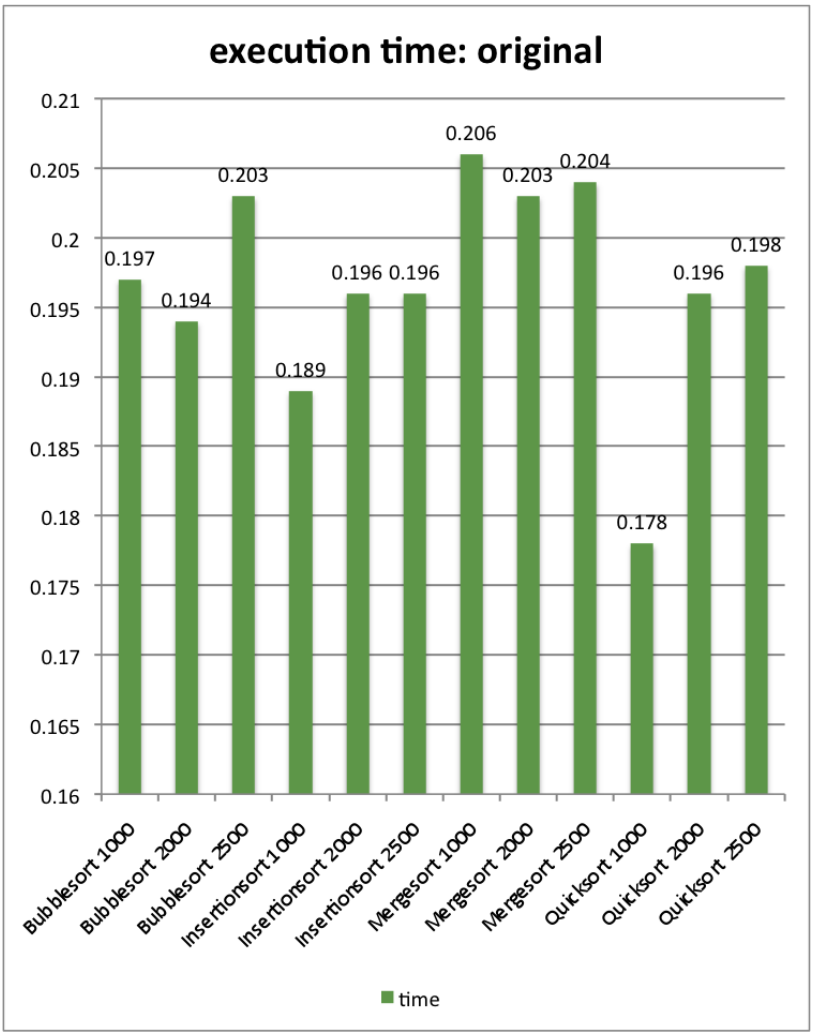


Figure 7.6: Execution time of original input programs, measured with **time**.

7.7.3.3 Execution time of tracing phase

Figures 7.8 and 7.9 contain the execution times of the tracing phase for each instrumented input program. The **java time** execution times are divided into IF-Tracer initialisation and instrumented input program computation (see 7.7.1). The results show that Quicksort has a bigger overhead compared to Mergesort. The instrumented Mergesort performs better than Quicksort.

7.7.3.4 Overhead Factors

The overhead factor is defined here as

$$\frac{exectime_{instrumented}}{exectime_{original}}$$

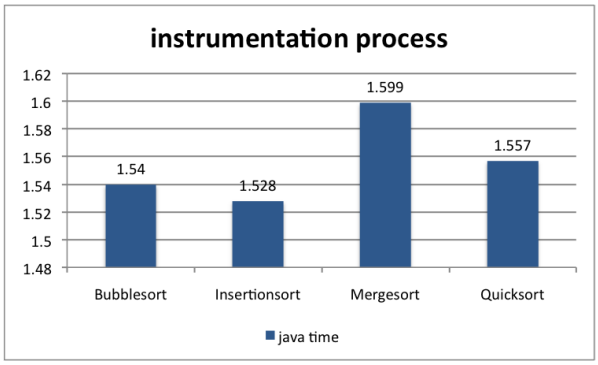


Figure 7.7: Execution time of the instrumentation phase of the IF-Tracer, measured with **java time**. The amount of input values didn't influence the execution time of the instrumentation process.

Figures 7.10 and 7.11 contain the overhead factors for each input program. The results show that Mergesort has the smallest overhead factor.

7.7.4 Bottlenecks

Earlier performance tests delivered much larger overhead values. Profiling the whole information flow framework with a profiler[5] revealed that much computation time is spent in *Java Reflection* methods. The *isArray()* method turned out to be very expensive. This method was used by get/put-field instruction handlers to distinguish between array and non-array field references at runtime. A couple of runtime checks used this method as well.

After outlining this bottleneck, a statical analysis was implemented and it replaced the runtime *Java Reflection* calls. During the instrumentation phase each reference used by get/put-field instructions is analyzed for being an array or a non-array reference type (see chapter 5). An "ordinary" field reference is hooked with the usual field handler (now without calls to *Java Reflection*), an array reference is hooked with a (new) array reference handler.

The checks weren't removed but flagged (could be activated if needed).

The removal of *Java reflection* calls had a big impact on performance concerning execution time. Hence, such functionality should only be used, if all other means refuse to work.

The whole taint book keeping is using lists and hash maps from the *java.util* package. Querying and updating this book keeping is very expensive. A query or an update of this book keeping is done by nearly every handler. Handling a put/get-field instruction turned out to be the most expensive operation of this framework. The corresponding handler does the most updates and queries on the book keeping. Therefore the Bubblesort input program shows the biggest overhead factor, because this algorithm has the highest field accesses frequency. An improvement of the book keeping performance would surely influence the overhead factors.

[5] Apple's Shark 4.6.1 (227)

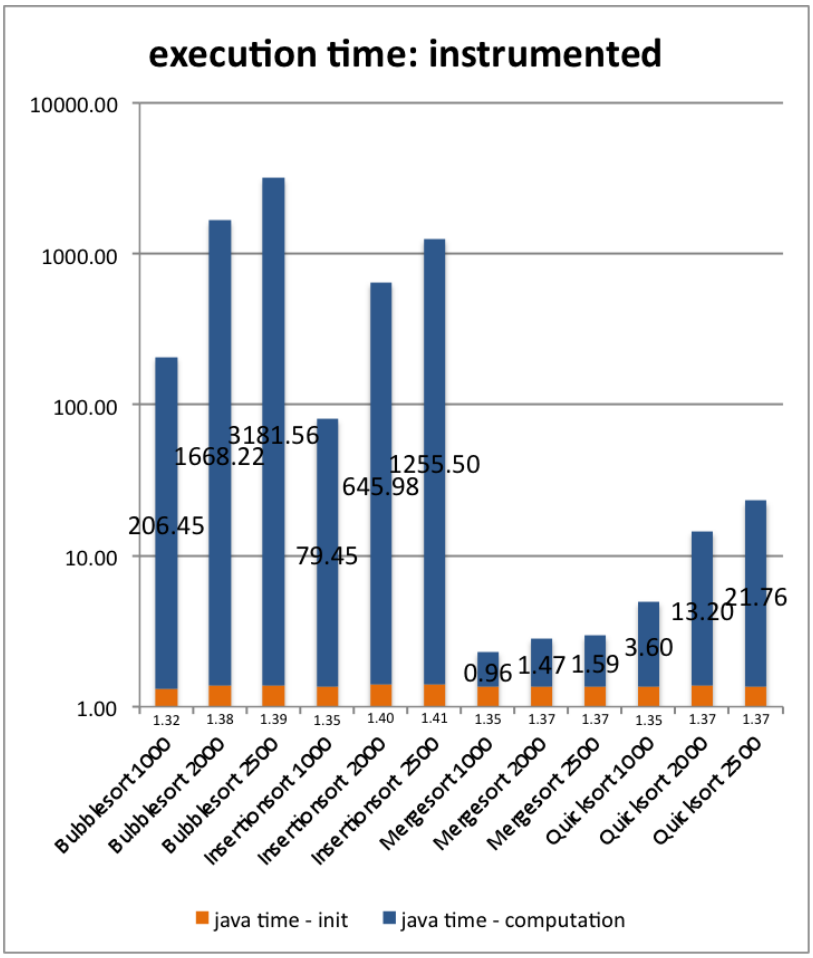


Figure 7.8: Execution time of the instrumented input program during the tracing phase of the IF-Tracer, measured with **java time**. **java time** results are split in IF-Tracer init and computation time. **Note:** Y-axis scale is logarithmic.

7.7.5 Wrap-up

For every test input set all instrumented input programs delivered the same (output) results. The input sets returned sorted in ascending order and the summary of the security monitor showed that taint propagation reached all input items. In other words: at startup only one input element of the input set is tainted and at the end all elements are tainted. This is the consequence of implicit information flow of the executing branch. All input programs contain if-statements comparing input set elements. If at least one of the involved variables is tainted, information flows implicitly to all variables that are written in the executing branch (see 2.8).

The results in 7.7.3 show how much overhead has to be paid for information flow analysis and the control of information flow by a security monitor. With growing amount of input values, the overhead doesn't grow linearly. There is also a big implementation dependency, certain instructions are less expensive than others. The factors are far from what could be tolerated by users of such systems.

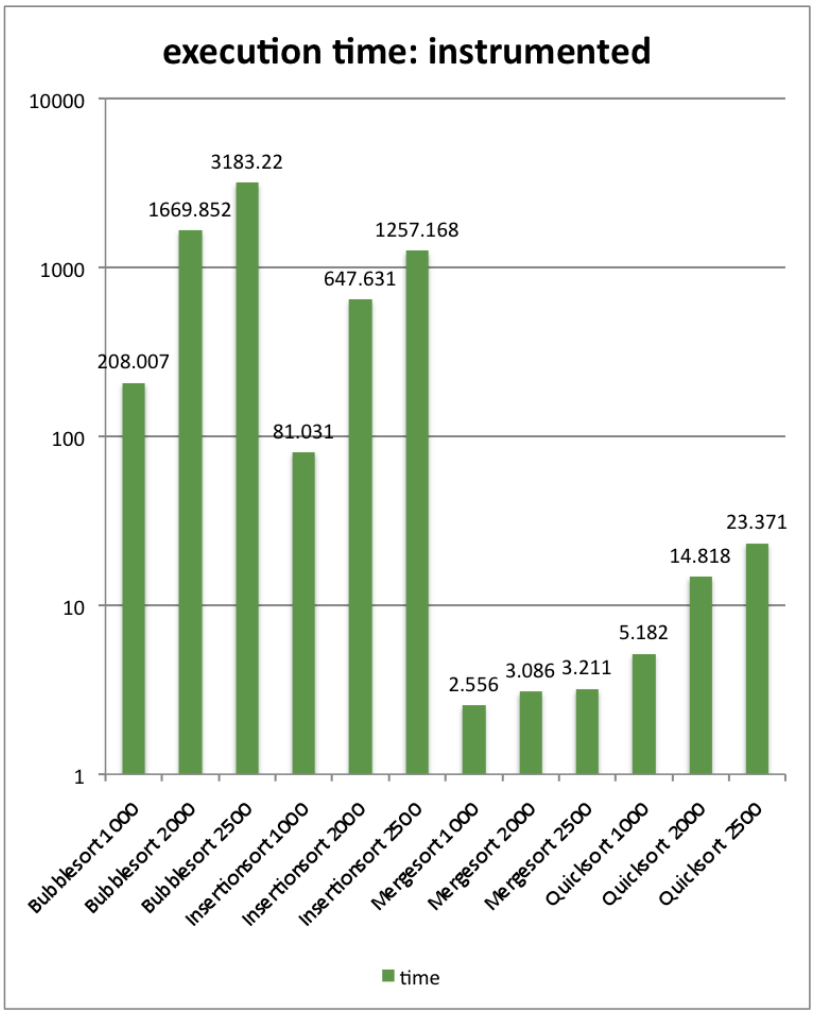


Figure 7.9: Execution time of the instrumented input program during the tracing phase of the IF-Tracer, measured with **time**. **Note:** Y-axis scale is logarithmic. The **java time** for the IF-Tracer initialization is shown below the X-axis.

7.8 Code Expansion

This section compares the original to the instrumented class files of the input programs (the same used in 7.7). Due to Java byte code instrumentation, the class file of the instrumented input program grows bigger, in this work this is called code expansion. Figure7.12 depicts a table comparing the original line count (lc, for short) of the class file to the instrumented one. The lc factor shows the enlargement of the instrumented code compared to the original one. The resulting code expansion factor is between 4 and 5.

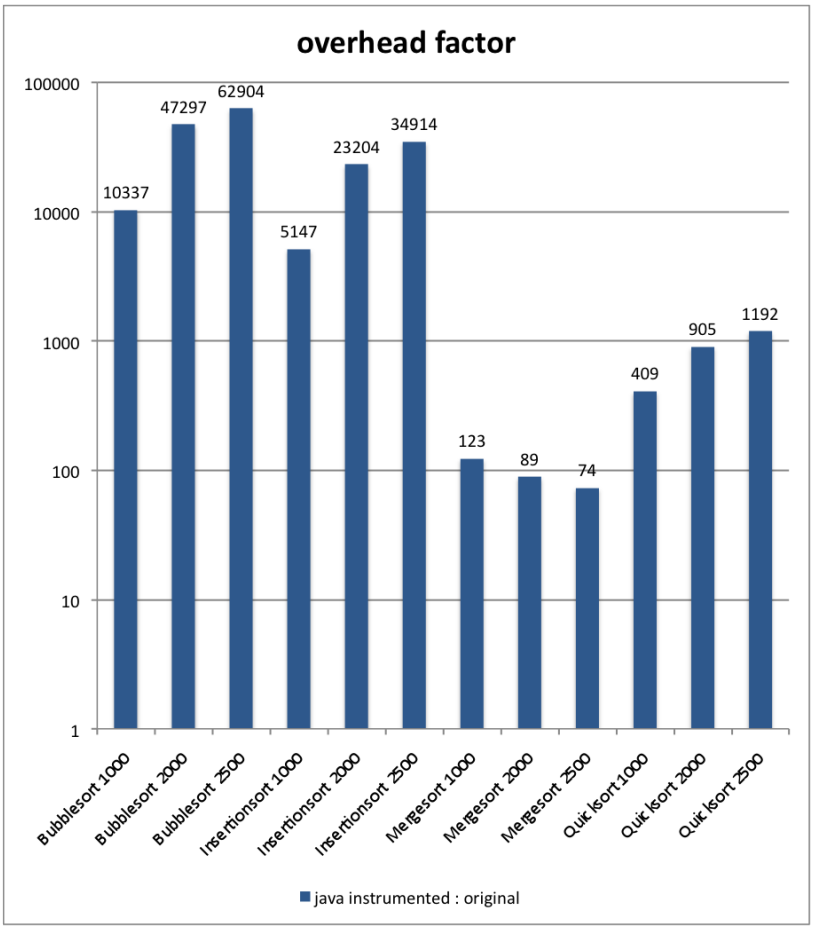


Figure 7.10: Overhead factor of each input program of **java time** measurements. **Note:** Y-axis scale is logarithmic.

7.9 Summary

The supported Java byte code instruction set is far too small to instrument real world input programs or Java libraries. Supporting Java exceptions turned out to be an essential key feature for an information flow framework. The lack of this feature in this work is a big drawback, because programs using exceptions can't be instrumented.

Tainting arrays on the level of single array elements (see chapter 5) allow to write more fine-grained security policies. It broadened the scope of runnable input programs, because standard input from the command line[6] could be declared as tainted e.g. *./testprogram 3 "sensitive data"* - second argument declared as tainted. Such fine-grained policies make the information flow analysis much more precise and can avoid conservative analysis decisions, that decrease the probability of label creep.

The shadow stack approach allows to stay independent from Java compilers. Arbitrary Java byte code can be handled. It also helps in tedious debugging scenarios. But more

[6] *main(args)* argument

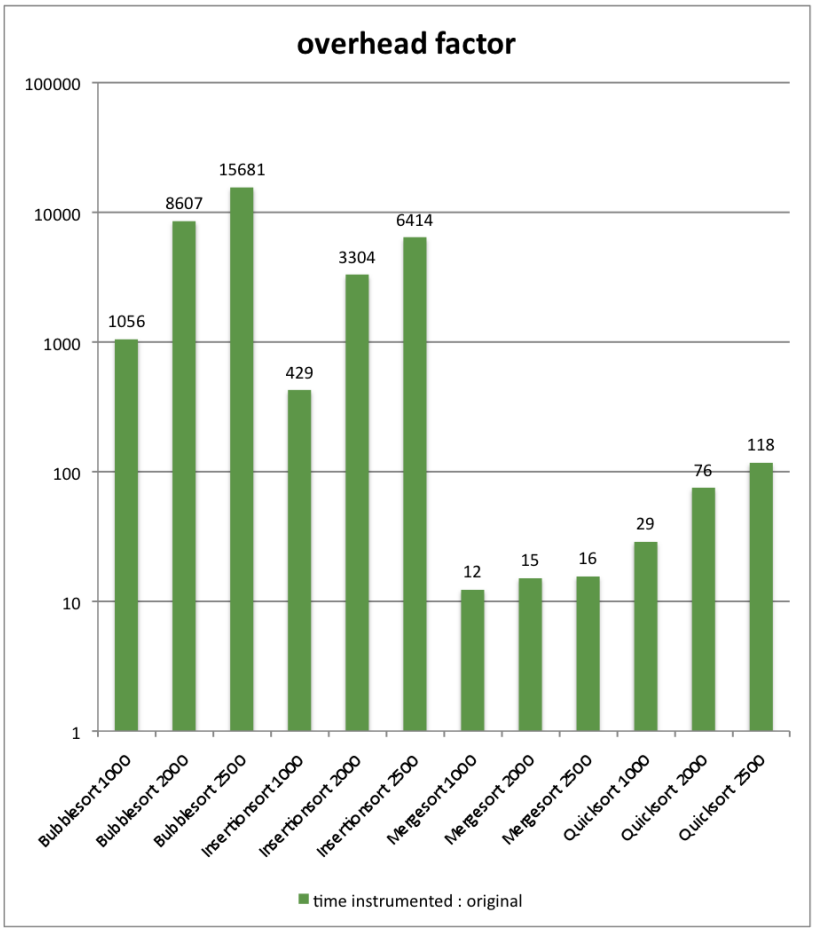


Figure 7.11: Overhead factor of each input program of **time** measurements including Java initialization and tear down. **Note:** Y-axis scale is logarithmic.

input program:	**Bubblesort**	**Mergesort**	**Insertionsort**	**Quicksort**
lc original:	238	181	274	293
lc instrumented:	998	883	1130	1267
lc factor:	4.19	4.88	4.12	4.32

Figure 7.12: Code expansion of instrumented class files.

additional instructions are needed, this in turn increases the code expansion.

Identifying join points is essential if implicit information flows have to be handled. The used algorithms are simple but slow.

The implemented information flow framework is totally independent from Java compilers or virtual machines. The price that has to be paid for this feature are huge overhead factors and code expansions. More efficient book keeping data structures would diminish the overhead.

8 Related Work

In [7] Vivek Haldar, Deepak Chandra and Michael Franz developed a plug-in for a JVM that adds mandatory access control (MAC) checking access to Java objects. They use a pure dynamic analysis through instrumentation and ignore implicit information flow. Each object carries a label that indicates the current state of the encapsulated data with respect to sensitivity. Sensitive data is tagged with a secret label, other data has a public label. Sinks are called output channels, the labels of those objects reflect what level of data is permissible to output on that channel. Writes from objects with a secret label to output channels with a public label are prohibited. A secret label propagates to another object whenever a read operation takes places on an object that has a secret label. The label of the object that reads this secret data becomes also secret. The initial tagging of all objects is done through interception of object creation instructions like *new*. The security policy is specified in Java source code, this code is called back by the security monitor and implements the enforcement mechanism. Compared to the default Java access control system the policies are more fine-grained. Nevertheless, the granularity of tagging whole objects comes with difficulties emerging as soon as an object contains mixed data i.e. the object has read public and secret labeled data - the secret label overwhelms and therefore non-sensitive writes of public data to channels are prohibited wrongly. Not much details on the implementation and the used algorithms were published by the authors.

In [6] a tainted value is input that isn't trustworthy. This work deals with problems concerning web services and user input. It focus on identifying, tracking and preventing improper use of such untrusted data with a pure dynamic approach on strings only. Instead of a policy a specification identifies sources of tainted strings and sensitive methods that shouldn't be called with tainted input. The definitions of sources and sinks are slightly adapted to the web context of the work. Taint propagation is quite simple because it is closed under the class of operations on strings like concatenation, case conversion etc.. To overcome the problem that tainted entities spread very fast over the whole system (often referred to as label creep) and making it unusable, a simple mechanism for un-tainting tainted strings is used. The following heuristic is used: A string that is undergoing an operation that does matching and checking in the $java.lang.String$ library package is being "sanitized" and may be untainted. Of course trust is shifted back to the user assuming he does some meaningful checks on the string (e.g. regular expression matching to cut out "evil" characters"). In this work there is no real "sanitizing". A tainted variable is un-tainted if the variable is overwritten with a value that isn't tainted. [6] instrumented only the string library to trace taint propagation. Further Java classes are instrumented at load-time through a custom class loader.

In [3] Deepak Chandra and Michael Franz used a combination of static (ahead of execution) and dynamic analysis (concurrent with execution) techniques that they named hybrid. It it a framework for the Java Virtual Machine based on Soot. The VM is IBM's JikesRVM. Due to some changes to the virtual machine the approach isn't completely independent from the

vm. Policies are specified in Java source code, the policy may me changed at runtime. The framework traces information flow at the level of entities i.e. variables. They use the same principle of security labels which are attached to the entities. According to this work any call to a sink with an argument more restrictive than the specified label in the policy is blocked by the security monitor. Handling implicit information flow of executing branches, was also implemented with a pc-label, smilar to this work. They also use the immediate postdominator of the conditional branch to declassify the pc-label but they use a backward dataflow analysis to compute the new value. No further details are written about this topic. Implicit information flow from the non-executing branch is handled by a static analysis, which determines the compensating code that is inserted in the executing branch to avoid leaks of this kind of flow. Again there are missing details about the used analysis. Polymorphism is handled at least for fields. It isn't written if dynamic method binding is handled. The arrays are handled very conservative compared to this project. An array has one label for all of its elements. In this work information flow analysis traces single array elements.

Thrishul [11] uses also a hybrid approach, instead of an external component it is fully integrated into a virtual machine (here it is a Kaffe JVM). Nevertheless, instrumentation is still necessary in some cases. The basic units involved in information flow are also variables but there are called class members. They use also the pc-label approach for implicit information flow tracing. Exceptions are mostly handled with static analysis, a basic block in the control flow graph gets an additional edge to the basic block containing the catch block code. Due to the fact that in some cases the catch block is unknown until it is executed, this case is handled through best effort. The direct integration of a virtual machine significantly improves the performance. Instrumentation approaches have far more overhead. More precise comparisons to this work isn't possible due to the lack of details in the publication.

9 Conclusions

This work integrates information flow analysis in Java input applications through Java byte code instrumentation. The analysis allows to trace information flows throughout runtime. Explicit and implicit information flows (only for executing branches) can be traced. The implemented information flow framework instruments Java class files before they are executed in the instrumentation phase and writes them back to the file system. Execution of instrumented input programs starts the information flow tracing phase of the framework. A security monitor is initialized according to the security policy provided as XML-document. The security policy declares sources and sinks. The security monitor assures that sensitive information isn't flowing to sinks.

On one hand instrumentation guarantees independence from virtual machine environments and compilers, on the other hand a big price has to be paid concerning code expansion and performance overhead.

Due to the lack of direct access to virtual machine components they have to be replayed on own data structures. Here a so-called shadow stack approach is used to mirror the virtual machine operand stack. This approach is independent too and works with arbitrary Java byte code that passes Java verification. The integration of meta-information in the shadow stack alleviates tedious debugging scenarios.

The appliance of a combined static (ahead of execution) and dynamic (concurrent with execution) information flow analysis (hybrid, for short) outperforms a single analysis approach. Static analysis can reduce runtime overhead. For every Java byte code instruction a hook to a matching handler can be inserted, hence less case distinction has to be made at runtime. Replacing the runtime case distinction for array and non-array field references using *Java reflection* with a static analysis ahead of execution, immensely improved performance. The dynamic analysis adds back accuracy to information flow tracing that is diluted by pure static analysis. Certain program aspects are only known at runtime and can only be traced by a dynamic analysis. The dynamic analysis handles the dynamically bounded method calls. Taint propagates only through methods that are actually called during runtime and not to all polymorphic candidate methods determined statically.

A pc-label is used to trace implicit information flow of executing branches. The pc-label is escalated at conditional statements containing tainted variables and de-escalated at corresponding join points. The problem of identifying join points in control flow graphs has been reduced to the problem of finding immediate postdominators. A simple algorithm solving this problem has been presented. From the computed postdominator sets, the postdominator tree is constructed, which allows to determine the immediate postdominators very easily. The used algorithm has a time bound of $O(nodes^2)$, which is quite slow. The computations needed to construct the postdominator tree are done ahead of execution in the instrumentation phase, hence they don't impact on the performance during the tracing phase.

Arrays are handled at the granularity of single array elements. Subsequently taint infor-

mation can be provided for every array element. This broaden the scope of executable input programs, because single standard input arguments can be declared to carry sensitive information. But also security policies can be written at a more fine grained level. Information sources and sinks can be declared much more precisely. Of course this improves information flow analysis, which gets much more precise and avoids conservative analysis decisions decreasing the probability of label creep. The taint book keeping uses ordinary *java.util* data structures, which increase the overhead as the results of performance tests presented. The performance overhead doesn't counter-weigh the security benefits.

The set of supported Java byte code instructions is far too small. Especially the lack of exception handling is a great drawback. Being able to handle Java exceptions has been identified as a must for future information flow frameworks. Due to this fact and the limited set of supported byte code instructions no Java libraries have been instrumented.

Beyond control borders information flow analysis isn't possible anymore. The borders here are quite limiting, because neither library calls nor *Java native calls* can be traced. Of course more such borders exist e.g. data base interactions, socket connections etc.. Being able to trace information flows without gaps is essential, especially if usage control is coming into play. As a matter of fact, such borders need to be crossed over.

9.1 Future Work

Broaden the supported Java byte instruction set and adding exception handling will allow to experiment with larger input programs and libraries. The information flow analysis will be extended to cover implicit information flow of non-executing branches.

As mentioned before the performance overhead of the implemented information flow framework isn't acceptable. Improving the performance of the book keeping data structures would clearly diminish the overhead, but if a pure instrumentation approach can reach an acceptable overhead is rather questionable. There is related work claiming to have accomplished this aim, but not enough details on the implementation is being presented to really reconstruct this achievements in real-world scenarios. Hence, before no stone is left unturned to improve the performance of the current framework, the following question shall be answered first: "*How important is independence of runtime environments and compilers? And are users up to wait longer for this independence?*" An extensive examination of alternative approaches will show how big the differences in performance overheads really are. One important alternative that should be checked surely a Java virtual machine extension, implementing information flow analysis. This would probably decrease overhead, because additional code isn't interpreted as Java byte code, but runs directly in the virtual machine. The complexity of the implementation would be diminished, due to the higher level of abstraction (Java source code, instead of Java byte code), progress could be achieved much faster. Of course such an approach cuts back on independence from runtime environments.

To overcome the information flow tracing borders mentioned above[1], future work shouldn't be bounded to only one level of abstraction. In other words, several information flow tracing frameworks will be developed, each is adapted to a certain level e.g. Java byte code, C, X11, ASM, ... If a running analysis meets a border were further tracing isn't possible, it reports to a higher instance supervising the overall information flow. This instance could then delegate

[1] see also 2.9.3

further tracing to a matching framework e.g. Java byte code analysis meets a native call, the higher instance activates a framework usable with the C language to continue tracing. Such a cross-border information flow analysis approach tends to be more complex and needs a representation of information and data that is abstract enough to be exchanged between different analysis instances on different abstraction levels. Of course a security policy should also be declared in a more abstract fashion in order to be applied on different levels.

Bibliography

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers*. Addison Wesley, January 1986.

[2] Cinzia Bernardeschi, Nicoletta De Francesco, and Luca Martini. Efficient bytecode verification using immediate postdominators in control flow graphs: Extended abstract. In Robert Meersman and Zahir Tari, editors, *OTM Workshops*, volume 2889 of *Lecture Notes in Computer Science*, pages 425–436. Springer, 2003.

[3] D. Chandra and M. Franz. Fine-grained information flow analysis and enforcement in a java virtual machine. pages 463–475, Dec. 2007.

[4] Keith D. Cooper, Timothy J. Harvey, and Ken Kennedy. A simple,fast dominance algorithm.

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

[6] Vivek Haldar, Deepak Chandra, and Michael Franz. Dynamic taint propagation for java. In *ACSAC '05: Proceedings of the 21st Annual Computer Security Applications Conference*, pages 303–311, Washington, DC, USA, 2005. IEEE Computer Society.

[7] Vivek Haldar, Deepak Chandra, and Michael Franz. Practical, dynamic, information-flow for virtual machines. *Technical Report TR*, 05-2, 2005.

[8] Thomas Lengauer and Robert Endre Tarjan. A fast algorithm for finding dominators in a flowgraph. *ACM Trans. Program. Lang. Syst.*, 1(1):121–141, 1979.

[9] Stephen McCamant and Michael D. Ernst. Quantitative information flow as network flow capacity. *SIGPLAN Not.*, 43(6):193–205, 2008.

[10] Jon Meyer and Troy Downing. *Java virtual machine*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 1997.

[11] Srijith K. Nair, Patrick N. D. Simpson, Bruno Crispo, and Andrew S. Tanenbaum. A virtual machine based information flow control system for policy enforcement. *Electron. Notes Theor. Comput. Sci.*, 197(1):3–16, 2008.

[12] Robert Endre Tarjan. Finding dominators in directed graphs. *SIAM J. Comput.*, 3(1):62–89, 1974.

[13] Bill Venners. *Inside the Java Virtual Machine*. McGraw-Hill, Inc., New York, NY, USA, 1996.