San Jose State University
SJSU ScholarWorks

Master's Projects

Master's Theses and Graduate Research

Spring 5-22-2017

Dynamic Information Flow Analysis in Ruby

Vigneshwari Chandrasekaran
San Jose State University

Recommended Citation

Chandrasekaran, Vigneshwari, "Dynamic Information Flow Analysis in Ruby" (2017). *Master's Projects*. 520.
DOI: https://doi.org/10.31979/etd.kz95-t8bz
https://scholarworks.sjsu.edu/etd_projects/520

Dynamic Information Flow Analysis in Ruby

A Project

Presented to

The Faculty of the Department of Computer Science

San José State University

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

by

Vigneshwari Chandrasekaran

May 2017

The Designated Project Committee Approves the Project Titled

Dynamic Information Flow Analysis in Ruby

by

Vigneshwari Chandrasekaran

APPROVED FOR THE DEPARTMENT OF COMPUTER SCIENCE

SAN JOSÉ STATE UNIVERSITY

May 2017

Dr. Thomas Austin	Department of Computer Science
Dr. Robert Chun	Department of Computer Science
Dr. Jon Pearce	Department of Computer Science

ABSTRACT

Dynamic Information Flow Analysis in Ruby

by Vigneshwari Chandrasekaran

With the rapid increase in usage of the internet and online applications, there is a huge demand for applications to handle data privacy and integrity. Applications are already complex with business logic; adding the data safety logic would make them more complicated. The more complex the code becomes, the more possibilities it opens for security-critical bugs. To solve this conundrum, we can push this data safety handling feature to the language level rather than the application level. With a secure language, developers can write their application without having to worry about data security.

This project introduces dynamic information flow analysis in Ruby. I extend the JRuby implementation, which is a widely used implementation of Ruby written in Java. Information flow analysis classifies variables used in the program into different security levels and monitors the data flow across levels. Ruby currently supports data integrity by a tainting mechanism. This project extends this tainting mechanism to handle implicit data flows, enabling it to protect confidentiality as well as integrity. Experimental results based on Ruby benchmarks are presented in this paper, which show that: This project protects confidentiality but at the cost of 1.2 - 10 times slowdown in execution time.

ACKNOWLEDGMENTS

I would like to express my gratitude to my advisor Dr. Austin for his support, guidance and motivation throughout this thesis. Also, I would like to thank my committee members Dr. Chun and Dr. Pearce for their invaluable time and support. And last but not least, thanks to my family and friends, especially my friend, Venkatesh Ramanujam, for inspiring me to learn and love programming.

TABLE OF CONTENTS

CHAPTER

APPENDIX

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

Introduction

With the luxury of software aided automation, there comes a risk of having security issues. We are often the target of hackers, who gain valuable information about us by exploiting flaws in the software that we use. This has led to the invention of a lot of security software to safeguard the information underneath. Though this has been effective to some extent, the hackers keep inventing techniques that bypass the protection that we have. One common area of interest for attackers is stealing secret information, like login credentials, credit card information, innovation details, and top-secret government information.

Introducing information flow analysis to a language helps protect confidentiality and data integrity. Information flow can be defined as the transfer of information from one variable `a` to another variable `b` during execution. Every variable is associated with a security level. Transfer of information from a high level to a low level leads to confidentiality or integrity violations. In reference to the scope of this project, there are two security levels: secret and public. If there is a data flow from secret to public, it is considered as a confidentiality violation. This project focusses on ensuring confidentiality in JRuby. Whenever there is a data flow from secret to secret, public to public or, public to secret, the execution is allowed. If the data flow occurs from a secret level to a public level, the execution is not permitted, and the program throws exception to preserve confidentiality.

1.1 Motivation

Humans are prone to mistakes. This seems especially true when we consider writing software code. Processes such as code reviews can help, but we can still not be sure that the code is bug free. By extension, when there is a bug, there is a possibility of software instability that can be utilized in a malicious way. Many software vulnerabilities like buffer overflows, cross-site scripting, and SQL injection are due to the improper handling of external data injected into the software system. In general, these problems are the result of external input from an untrusted source propogated through the program to affect part of the code dealing with sensitive data. There have been several approaches to solve this issue caused by improper handling of information flows [1].

Per Qin et al. [1] proposed using security tools to detect information flow violations. Publicly known software such as LibSafe can find only about 30% of the software security violations. In addition, [1] does not divulge any additional information regarding how the attack was carried out or what were the steps involved in the attack. So, to complement the security tools, ACLs were introduced. Though they worked well with restricting data access to privileged users, they did not deal with the problem of tainted flow propagation. Although techniques like mandatory access control prevent problems with memory leaks, the approach is too slow to use it in real time [2].

Alternatively, privacy and integrity of the sensitive data can be statically checked for information flow that is susceptible to an external attack. But the main disadvantage of this attack is that even if a branch would not be executed in runtime, it will still be analyzed, leading to more false positives.

1.2 Solution - Dynamic Information Flow Analysis

Information flow analysis is the classification of variables used in the program into different security levels and monitoring the data flow across levels at runtime.

For instance, in Figure 1 there are two variables: `a` and `b`. `a` is marked as public and `b` is marked as secret variable. If the transfer of information is from `b` to `a`, it is considered a confidentiality violation. This data flow is prevented through information flow analysis, thereby eliminating confidentiality violations [3].

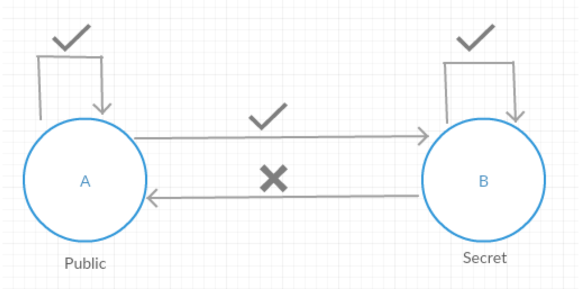


Figure 1: Information flow across different security levels.

There are two types of information flows.

1. Explicit flows: Secret leaking directly to a public variable

2. Implicit flows: Secret leaking to a public variable through program control flow

Explicit flows happen whenever there is an assignment statement, IO read/write, or function calls with a return statement involving secret data. Figure 2 shows explicit information flow.

```
username = readFromConsole()     // IO Read
print getBalance(username);
//Secret information is retrieved with a public data

// The getBalance code looks like this..
double getBalance(username) { // Gets a public data
return balanceList.get(username);  // Returns a secret
    information
}
```

Figure 2: Explicit Flow - Code Snippet

An implicit flow from *a* to *b* can be defined as a data flow from a random variable c to b with the data flow occurring at runtime based on a decision made by `a`. Figure 3 shows an implicit information flow from `balance` to `rFlag`.

Information flow analysis is of two variants:

```
double balance = secret(25000) // Secret data
bool rFlag = false // Public − Minimum balance required
   flag
if balance >= 5000 // Implicit flow
  rFlag = true   // rFlag can partially reveal the
     balance information
end
```

Figure 3: Implicit Flow - Code snippet

1. Static information flow analysis

2. Dynamic information flow analysis

Static information flow analysis is a language based approach that ensures information flow policies through a technique like type checking. If a program has any policy violations, it means there are improper information flows that could cause security violations and hence will not be permitted for execution. This technique works at compile time and requires the programmer to use certain annotations to mark public and secret context so that the type checking can be carried out. There is little runtime overhead since everything is done before execution. But on the downside, though static analysis can cover all paths of execution, it is a poor fit for dynamically typed languages like Ruby. For instance, the *eval* in Ruby allows the programmers to compile and evaluate a string at runtime. This dynamic evaluation can lead to false alarms in static analysis, since the type of expression will not be known until runtime.

Dynamic information flow analysis is a method that involves tagging the data at runtime with information flow labels. Whenever secret data gets written/leaked to a public variable, the execution can be terminated to avoid confidentiality violations [4]. This process incurs some runtime overhead as the cost paid for preventing confidentiality violations [5].

The confinement problem [3] is producing public results that depend on private data. For instance, if a person visits a website that calculates how much of a tax refund he is eligible for, it gets sensitive data from the user to calculate results. The downside is that there has to be an assurance that the data has not been written to any of the server variables to maintain confidentiality. Dynamic information flow analysis helps in addressing the confinement problem.

1.3 Information Flow Analysis in JRuby

This paper introduces dynamic information flow analysis to JRuby, a Java Implementation of Ruby. There are three reasons for choosing this topic.

1. Much of the current software executes code on a server which might not be completely trustworthy. This leads to a need for data safety at runtime. By adding memory safety to the language rather than the application, we provide a more systemic defense against data leaks [4].

2. There can be instances where users trust the server that it does not persist users' data. Yet, the server may depend on untrustworthy library code to complete execution. There can be no guarantee that this third-party code does not misuse the data. This looks similar to the confused deputy problem which puts a serious threat on the reliability of the server.

3. JRuby currently supports a dynamic tainting mechanism. This allows us to concentrate on dynamic information flow handling for implicit flows and provides a platform for runtime performance comparison between the JRuby baseline and JRuby with dynamic information flow analysis.

My implementation only handles various forms of *if* statements but additional constructs can be extended from this in a straightforward manner. This project presents experimental results of small benchmark programs run on unmodified JRuby and JRuby with dynamic information flow analysis. In terms of performance, the baseline code is 1.2 - 10 times faster than the modified version. But on the other hand, the modified code catches implicit data leaks whereas the baseline allows data leak across security levels. Chapter 5 discusses the execution results of benchmark programs and the time taken for execution. In this test, all the data flows were valid. For data leaks, there cannot be any performance comparison, since the modified code crashes the execution right away.

The organization of this paper proceeds as follows. The next section discusses the literature survey of previous researches done in achieving confidentiality. Chapter 3 talks about JRuby architecture. Chapter 4 explains the implementation details of this project. Chapter 5 presents the performance results and chapter 6 concludes this paper with a discussion on future scope of this project.

CHAPTER 2

Literature Review

This chapter discusses information flow analysis and previous research works towards achieving it. The papers being discussed in this chapter are the inspiration and starting point for this thesis.

2.1 Background and Motivation

Many current security practices aim at protecting confidentiality and integrity of data by the computing systems. Confidentiality is ensuring that the sensitive data does not flow into inappropriate domains, whereas integrity is ensuring that the data is not written from inappropriate domains. This thesis concentrates on ensuring confidentiality through dynamic information flow analysis.

Access control was introduced to prevent users from accessing a file, thereby achieving confidentiality. However, once the file is read, the propagation of data is not monitored by access control. Hence there can be data leaks. Similarly, cryptographic encryption ensures only the person who has the decryption key can have access to the data. But it fails to make sure the data is not leaked afterwards. These limitations led to the need for data monitoring throughout the process.

To address the data propagation issue, information flow analysis is introduced at the language level. Sabelfeld and Myers [6] provides a summary of prior research on information flow analysis.

2.2 Information Flow Analysis

Consider two security levels: secret and public. There are two variables `s` and `p`. `s` belongs to secret level and `p` belongs to public level. Whenever there is transfer of data across security levels, we call it information flow. In reference to our context, a data flow from secret to public, it is an information leak (or) confidentiality breach. Monitoring data flow across the security levels is called as information flow analysis.

2.3 Information Flow Analysis Approaches

There are two approaches to information flow analysis: static and dynamic. Static approach involves mechanisms that deal with certifying a program at compile-time. On the otherhand, dynamic approach makes a call to terminate the program whenever it finds data flow across unacceptable levels.

2.3.1 Static Information Flow Analysis

Denning [3] presents a static approach which certifies a program as valid only after it complies with the policies defined. They consider a programming language with simple data types: integer, boolean, and file. A program is certified as secure only when there is no data flow from `x` to `y`, unless specified in the policy. This approach has been implemented for assignment statements, I/O, control structures, procedure calls, and exception handling. Certification semantics are defined and incorporated to the analysis phase of a compiler. The compiler verifies the program against the standard defined in the semantics. Lampson [7] identified three channels through which data can propogate. *Legitimate channels* are the outputs of a program. *Storage channels* are storage objects where the data gets written during the execution. *Covert channels* are unintendend channels through which the data can leak; it can be program running time, noise, or power consumption. The first two channels are covered in [3] whereas the covert channels fall beyond the scope. It should also be noted that it is impossible to eliminate the covert channels completely in distributed systems with plenty of shared resources.

Type checking can be used to achieve information flow analysis. The Jif compiler [8] has implemented the type-checking approach. In this approach, every expression has a static label associated with it along with its data type. For example, an expression with type *int* has a label. The label specifies how the value can be used.

Since labels are static, type-checking is done at compile time. It ensures there is no improper information flows at runtime.

Type-checking also supports handling implicit flows through the use of a program counter. The program counter is associated with a label *pc*. The program is certified as secure only if the label of the assigned variable is equal or higher than *pc*. Consider a conditional statement as shown in the code snippet 4. The variable *s* has a label *secret* and *p* has a label *public*. The *pc* is also *secret* to match the current branching conditional statement. The assignment inside the control structure will not be permitted because the label of assigned variable is not as restrictive as that of program counter.

```
s = 1
p = 0
if s == 1
    p = 1
end
```

Figure 4: An implicit flow.

Volpano et al. [9] have developed a type-system based on Denning's linformation flow analysis and proved its correctness. Although there are plenty of research works that prove static analysis for achieving confidentiality, the static approach is not suitable for dynamically-typed languages like Ruby, Perl, and JavaScript. The static approach is even more difficult in cases of dynamic code evaluation supported through constructs like *eval* methods. To handle data safety in a dynamically-typed language, dynamic analysis is preferred because of its flexibility on applying policies at runtime.

2.3.2 Dynamic Information Flow Analysis

King et al. [10] investigated explicit and implicit flows identified by a static information analysis algorithm. Though it catches every path that leads to confidentiality violations, it suffers from an extremely high number of false alarms. Most of these false alarms are due to unchecked exceptions in implicit flows, thereby proving the need for more practical analysis at runtime.

Austin and Flanagan [4] presented dynamic information flow analysis using universal and sparse labelling strategies. In universal labelling, every value is tagged with a information flow label. At runtime, whenever a resultant variable has a public label, it is made sure that it could not have been influenced by value that has a secret label. Label locality is where most or all of the data items in a data structure. In this case, associating every value with a label will incur unnecessary overhead at runtime. To avoid the label locality problem, [4] introduces sparse labelling where the information label will be introduced to values only for values those get transferred across information flow domains. This approach reduces a significant amount of runtime overhead associated with the universal labelling strategy. Experiments done in this paper proved that sparse labelling has a substantial speedup over universal labelling.

In another work of Austin and Flanagan [11], the focus was made on implicit flows and letting most of the programs to complete execution while preventing data leaks. This paper uses the permissive-upgrade strategy, where a security label P is introduced to track partially-leaked data. Such data containing private information, in some instances can be labeled as public in certain executions. This partial leak gets converted into total leak if this data is used in any conditional statement and the execution halts. To solve this conundrum, privatization operation is used. A privatization operation converts public and partially leaked data to private. As a ground rule, introducing privatization at sensitive uses of partially leaked data eliminates many stuck executions (i.e. program crashes).

Both these papers address the issues and solutions associated with JavaScript or any other browser-based programming language. This is because of the exponential increase in the usage of Internet applications and JavaScript is predominantly used for developing such applications.

Apart from information flow analysis, Devriese and Piessens [12] have presented a *secure multi-execution* strategy, where a program is executed several times, once per security level. This approach guarantees noninterference since the outputs at a level cannot possibly depend on inputs from higher levels. The downside of this approach is that it suffers a high cost in terms of CPU time and memory. *Faceted values* have been introduced to reduce this cost issue [13]

CHAPTER 3

JRuby Architecture

This chapter explores how JRuby works and its comparison with the traditional Matz's Ruby Interpreter (MRI). I use an example based on strings to explain the work flow in JRuby.

3.1 Ruby vs JRuby

Ruby interprets the input code using native machine language since it is based on C. On the other hand, JRuby compiles the input code to Java bytecode, that will run on a standard JVM. Unlike Ruby that interprets the script directly, JRuby is actually a compiler. Hence, ahead of execution, JRuby compiles the input script to bytecode. The JVM interprets this bytecode and then executes it.

```
sum = 1 + 2
print sum.to_s()
```

Figure 5: Script: ToString.rb

Consider the simple script given in Figure 5. This script converts an integer to its string representation. JRuby has several layers of abstraction, in addition to the bytecode generation. The rest of this chapter explains how a Ruby script gets transformed into Java bytecode, how the JRuby library is being used in the script, and the JVM interpretation of the bytecode.

3.2 JRuby - Java Bytecode

Once the *jruby* command is executed, JRuby takes the file name as the parameter, constructs an abstract syntax tree (*AST*), and generates the bytecode representation of the same. From the bytecode, a *class* file is generated. Methods to be used inside the script will be discussed in a short while discussing bytecode generation.

Figure 6 presents the functional architecture of JRuby.

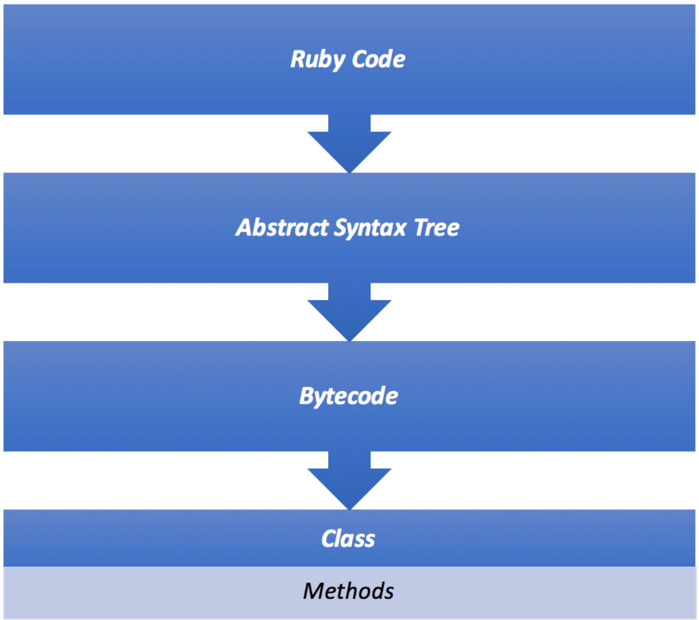
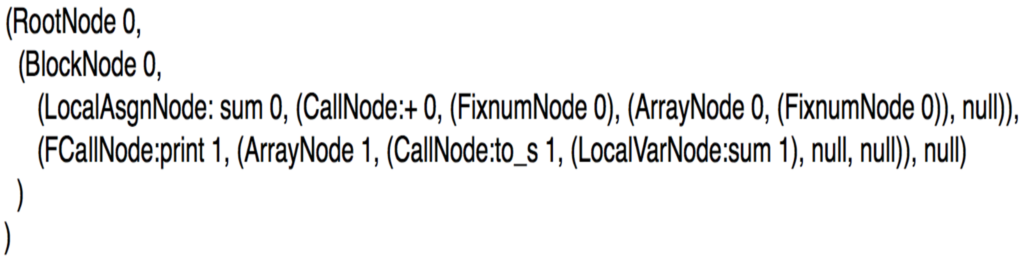


Figure 6: JRuby Architecture

3.2.1 AST generation



```
(RootNode 0,
 (BlockNode 0,
   (LocalAsgnNode: sum 0, (CallNode:+ 0, (FixnumNode 0), (ArrayNode 0, (FixnumNode 0)), null)),
   (FCallNode:print 1, (ArrayNode 1, (CallNode:to_s 1, (LocalVarNode:sum 1), null, null)), null)
 )
)
```

Figure 7: AST - ToString.rb

The JRuby parser generates an AST for the script. Figure 7 shows the *ast* generated for the *ToString.rb* script shown in Figure 5.

On a closer look at the AST, it can be noted that several nodes are created to represent the script. `RootNode` is the reference/parent node. `BlockNode` is created for every block in the script. Since the example script has a single block, this node can be of less use in the discussion.

`LocalAsgnNode` is created for the first statement: *sum = 1 + 2*. `FixnumNode` is used for numerical constants. A `CallNode` with a parameter `"+"` denotes the arithmetic operation.

`FCallNode: print` is created for the second statement: *print sum.to_ s()*. `LocalVarNode` retrieves the local variable named *sum*. `CallNode` represents the *to_ s* function call. `FCallNode` makes a call to the *print* method.

Each of these nodes is defined as a separate class in JRuby. Appendix A lists nodes defined in JRuby. This rich set of modularized and readable nodes helped in the implementation of this project. The AST gets complicated with complex logic and more blocks. This *ast* is the input for bytecode generation.

3.2.2 Bytecode generation

Since JRuby is Java based, the libraries are in `class` files compiled from the actual definition. For instance, the string representation is defined in a Java class named `RubyString.java`. This class is responsible for all string functions. JRuby uses annotations to link to Ruby-equivalent methods.

Figure 8 is a snippet from the RubyString class. The annotation `@JRubyMethod` tells the JRuby execution engine to link this method when `to_s` is referenced in a Ruby script.

The input script would contain code that refers to the library functions. For example `CallNode` makes calls to internal functions. During execution, those methods are required. Bytecode generated by JRuby is platform independent, as it is one of the key properties of Java. JRuby, on compilation, generates bytecode that can be

```
@JRubyMethod(name = {"to_s", "to_str"})
@Override
public IRubyObject to_s() {
  Ruby runtime = getRuntime();
  if (getMetaClass().getRealClass() != runtime.
     getString()) {
        return strDup(runtime, runtime.getString());
  }
  return this;
}
```

Figure 8: JRuby - to_s definition

interpreted by the JVM.

Figure 9 shows the bytecode generated for the script `ToString.rb`. I have edited the bytecode for readability without modifying the overall meaning. `ALOAD` and `ASTORE` statements are loading and storing the values from register, just like any other assembly code. Statements like `INVOKESTATIC` and `INVOKEVIRTUAL` makes calls to library methods. `INVOKESTATIC` is used when the method to be called is determined at compile time. `INVOKEVIRTUAL` is used when the method to be called in determined during runtime.

For instance, `INVOKEVIRTUAL` in line `L1` is used for assigning the computed value to `sum`. Since the reference to `sum` would not be created until runtime, the method to be called is determined at runtime. Similarly, in `L2`, retrieving value from `sum` can be determined only during runtime.

For execution, the bytecode from libraries is also required. Bytecode of the `to_s` method of `RubyString` is presented in Figure 10

3.2.3 Bytecode Execution

The execution is complete when the bytecode is interpreted by the JVM. Figure 11 shows the bytecode interpretation by the JVM.

```
L1
LINENUMBER 1 L1
ALOAD 0
ALOAD 2
ALOAD 0
INVOKESTATICToString.fixnum0 (Lorg/jruby/runtime/
   ThreadContext;)Lorg/jruby/RubyFixnum;
INVOKESTATIC ToString.invokeOtherOneFixnum1:+
ASTORE 11
ALOAD 7
ALOAD 11
INVOKEVIRTUAL org/jruby/runtime/DynamicScope.
   setValueZeroDepthZeroVoid
LINENUMBER 2 L2
ALOAD 0
ALOAD 2
ALOAD 7
INVOKEVIRTUAL org/jruby/runtime/DynamicScope.
   getValueZeroDepthZero ()
INVOKESTATIC ToString.invokeOther2:to_s
ASTORE 12
ALOAD 0
ALOAD 2
ALOAD 2
ALOAD 12
INVOKESTATIC ToString.invokeOther3:print
ASTORE 13
ALOAD 13
ARETURN
```

Figure 9: Script Bytecode

3.2.3.1 JIT Compiler

A JIT (Just In Time) compiler is available inside the JVM. As the name implies, it provides one more compilation step for some portions of code, identified as "hotspots". The JVM finds out which portions of the code would be used frequently and labels them as "hotspots". The JIT compiles the bytecode associated with the hotspot into

```
 public org.jruby.runtime.builtin.IRubyObject to_s();
Code:
0: aload_0
1: invokevirtual #45  // Method getRuntime:()Lorg/jruby/
   Ruby;
4: astore_1
5: aload_0
6: invokevirtual #62  // Method getMetaClass:()Lorg/jruby
   /RubyClass;
9: invokevirtual #149  // Method org/jruby/RubyClass.
   getRealClass:()Lorg/jruby/RubyClass;
12: aload_1
13: invokevirtual #64  // Method org/jruby/Ruby.getString
   :()Lorg/jruby/RubyClass;
16: if_acmpeq      29
19: aload_0
20: aload_1
21: aload_1
22: invokevirtual #64  // Method org/jruby/Ruby.getString
   :()Lorg/jruby/RubyClass;
25: invokevirtual #124  // Method strDup:(Lorg/jruby/Ruby
   ;Lorg/jruby/RubyClass;)Lorg/jruby/RubyString;
28: areturn
29: aload_0
30: areturn
```

Figure 10: RubyString - to_s Bytecode

native machine language, speeding up execution.

A simple tweak to the sample code in Figure 12 is done to trigger the JIT to identify a hotspot. The modified code contains an arbitrary loop which ensures the code gets executed multiple times.

Setting the runtime flag `-J-XX:+PrintCompilation` during execution displays all the methods that were compiled to native machine code from Java bytecode. The command used to display the hotspots is shown below:

```
$ ./bin/jruby -J-XX:+PrintCompilation ToString.rb > hotspot.txt
```

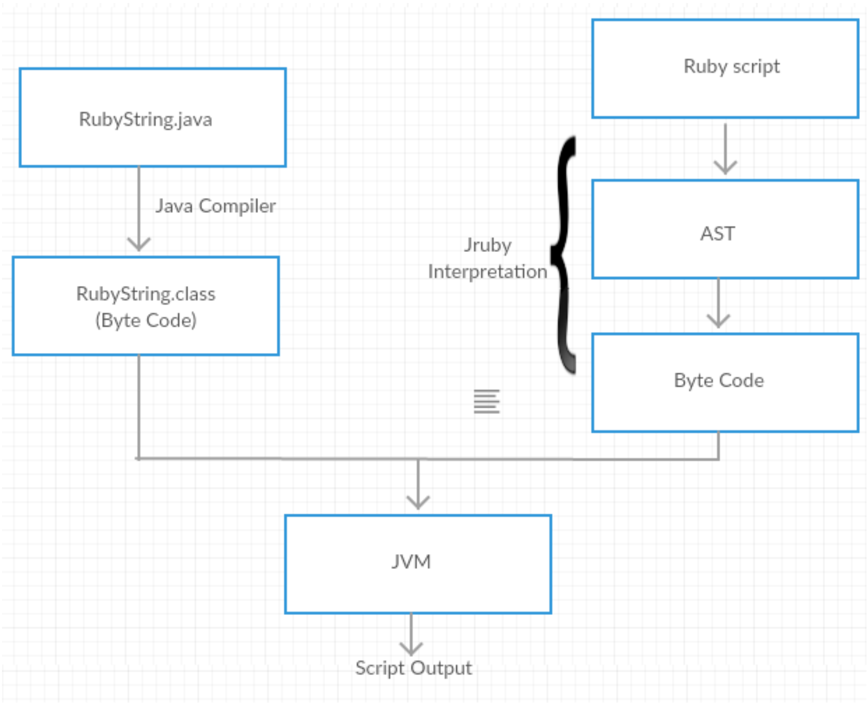


Figure 11: Execution

```
i = 0
while i < 10000 do
  s = "1".to_s()
  i = i + 1
end
```

Figure 12: Induced Hotspot

The output file *hotspot.txt* lists various methods that were identified by the JIT. Without the loop introduced in Figure 12, *hotspot.txt* did not have `to_s()`. After introducing the loop, the JIT added `to_s()` to the list of methods converted to native machine language as shown below:

```
1813 2936  3  org.jruby.RubyString$INVOKER$i$0$0$to_s::call (8 bytes)
1814 2937  3  org.jruby.RubyString::to_s (31 bytes)
```

CHAPTER 4

Implementation

The implementation is based on monitoring implicit data flows to track information flow at runtime. This implementation is focussed only on Ruby Strings. This can be extended to other datatypes in a straightforward manner. This chapter presents code development process, approaches used, and challenges faced during the implementation.

4.1 Software Criteria

The following software was used for configuring the JRuby development environment.

- JDK 1.8
- Maven 3.3.9
- Apache ANT 1.8

4.2 Characteristics of JRuby's architecture

The implementation of this project is closely coupled with the following three characteristics of JRuby.

- Multi-layer Abstraction
- Dynamic class generation
- Code optimization

These characteristics are explained to help understand the implementation of this project.

4.2.1 Multi-layer Abstraction

As shown in the previous chapter, the call stack for a simple code is quite large. For instance, a program with a single *print* statement involves 27 function calls. The multi-layer abstraction in JRuby provides modularity. For example, JRuby has a rich set of instruction classes which corresponds to Java bytecode. To add a new type of instruction, it is enough to modify the instruction definition layer. Once defined confirming to the standards required, the instruction can be included at any part of bytecode generation. The execution of the instruction is automatically taken care by the subsequent layers by calling the handlers provided within the instruction definition.

Figure 13 shows the new instruction, *IfEnd* created for this project. The use of this instruction is discussed in the section 4.3.3. Following are required to be followed by the new class created:

- Must have `IS - A` relationship with `Instr` class.
- Must implement `clone()` method.
- Must implement `visit()` method.

The `IfEnd` class inherits `NoOperandInstr` which means the instruction does not have any operands. The `FixedArityInstr` interface is a marker interface which instructs JVM that this instruction has a fixed number of operands. `IfEnd.ifEnd` is placed wherever this instruction is required. The `visit()` method is the handler method that will be called when the bytecode contains `IF_END`.

4.2.2 Dynamic class generation

The Java Virtual Machine executes the instructions from class file loaded by the class loader. To comply with this, as a next step to bytecode generation, JRuby generates a class file dynamically. A method handle is created for the method, `Ruby$script` which is the point of entry, similar to a main method in Java. Figure 14 shows the entire process from bytecode generation to obtaining a handle for the `Ruby$script` method.

- Bytecode obtained from the AST is stored in the variable `bytecode` in line 1.
- In line 2, a class handle is created from the bytecode generated in line 1.

```
package org.jruby.ir.instructions;

import org.jruby.ir.IRVisitor;
import org.jruby.ir.Operation;
import org.jruby.ir.transformations.inlining.CloneInfo;
/**
* Created by vigneshwarichandrasekaran on 4/5/17.
* Dynamic Information Flow Analysis
*/
public class IfEnd extends NoOperandInstr implements
   FixedArityInstr {
  public static IfEnd ifEnd = new IfEnd();

  private IfEnd() {
    super(Operation.IF_END);
  }

  @Override
  public Instr clone(CloneInfo ii) {
    return this;
  }

  @Override
  public void visit(IRVisitor visitor) {
    visitor.ifEndInstr(this);
  }
}
```

Figure 13: If End Instruction

- In line 3 & 4, a handle to the method `Ruby$script` is created.

4.2.3 Code Optimization

During bytecode generation, deadcode is eliminated. *Peephole optimization* [14] is being done to eliminate useless instructions. Every instruction has a flag `isDead`. During the compiler optimization pass, this flag is set to true if this instruction is eligible for elimination. In the final pass, the marked instructions are excluded from bytecode generation. When adding instructions to handle implicit flows, it was

```
bytecode = visitor.compileToBytecode(scope, context);

Class compiled = visitor.defineFromBytecode(scope, bytecode, classLoader);

Method compiledMethod = compiled.getMethod("RUBY$script", ThreadContext.class, StaticScope.class, IRubyObject.class, IRubyObject[].class, Block.class, RubyModule.class, String.class);

_compiledHandle = MethodHandles.publicLookup().unreflect(compiledMethod);
```

Figure 14: Dynamic Class Generation

important to ensure that they would not be deleted by the optimizer.

4.3 Implicit Flow Monitor

To handle implicit flows, this project altered the *if-else* construct in JRuby. The project aims at terminating the execution when a data leak happens, such as the leak shown in Figure 15.

4.3.1 Data Leak

In Figure 15, `rFlag` is a public flag, and `balance` is sensitive data. At the end of execution, by observing `rFlag`, one can guess the range of `balance`. To protect confidentiality, this project changes the byecode formation of the *if-else* construct, and also uses the existing tainting mechanism.

```
double balance = secret(25000) // Secret data
bool rFlag = false // Public - Minimum balance required flag
if balance >= 5000 // Implicit flow
  rFlag = true   // rFlag can partially reveal the balance
end
```

Figure 15: Implicit Flow

4.3.2 If-else construct

The if-else statement is built as show in the Figure 16. The condition gets evaluated first. If the condition is evaluated to true, then the program control jumps to a label that corresponds to the *then* statement. Otherwise, the program counter gets incremented normally. The *else* block follows immediately after condition evaluation and it contains a *jump* instruction to a label that occurs right after the if-else construct. Pseudo-code representation of JRuby's if-else construct is shown in Figure 16.

```
        decision = build(if_cond_expr)
        JMP(decision, true, L2)
L1:     result = build(else body)
        JMP L3  #L3 is the label of instruction after if-else
            construct
L2:     result = build(thenbody)
L3:     #other statements
```

Figure 16: If-else construction

The if-else construct in JRuby has 5 parts. They are as follows:

- condition evaluation
- then block
- then block exit
- else block
- else block exit

4.3.3 Dynamic Information Flow

The *then block exit* and *else block exit* contain instructions of what has to be done after the respective body execution. The implict data flow monitor requires a watchman to monitor the security level of the assigned variable in data assignment statements inside *then* and *else* blocks. If the assigned variable's security level is not as restrictive as that of the variable in the conditional expression, the execution is terminated. This project introduces a flagging mechanism and a watchman at the entry and exit points of *then* and *else* blocks.

As a last step to the evaluation of conditional expression, a flag is assigned to the resultant boolean value. This flag reflects the security level of variables involved in the conditional expression. Inside *then* or *else* block, if there is an assignment statement, the assigned variable's security level is compared with the flag introduced. If the assigned variable's security level is lower than the flag, the execution is terminated. Otherwise, the execution is allowed safely. The flag is reverted to false at the exit of the corresponding block. The flag revert operation is introduced because of the following reasons.

- To ensure the rest of the assignment statements following the *if-else* construct get executed safely.
- To provide a template to accomodate nested structures.

The flag revert operation required a new instruction to be introduced in JRuby as an exit monitor to the *then* and *else* constructs. It was straightforward to introduce a flag at the entry of *if* block, which was done as the last step in condition evaluation. However, reverting had to be done at the end of this control structure.

The *if* block can:

- be empty
- contain a then block
- contain an else-if ladder
- contain a then-else block

To accomodate any type of *if* block structure, this project introduced a new instruction template, shown in Figure 13. This instruction marks the *if* exit point. When the bytecode gets generated, this instruction is made available to be used to mark any exit criteria from the *if* block.

4.4 Compiler Optimization

This project was implemented and tested with an exclusion of one of the compiler optimization passes. During the creation of a new instruction, the instruction was overlooked during the compiler optimization. To incorporate the newly created instruction for this project, the optimization was disabled.

CHAPTER 5

Experimental Results

This chapter presents the analysis and experimental results of this project. To evaluate the performance cost of this project, two implementations of JRuby are used: Unmodified JRuby (referred to as baseline), and JRuby with dynamic information flow analysis.

5.1 System Configuration

The tests were ran on a MacBook Pro with a 2.5 GHz Intel Core i7 quad-core processor, 16 gigabytes of RAM, and running OS X version 10.12.2.

5.2 Performance Comparison

Performance comparison of the implementations were done on the benchmark programs [15] listed in Table 1.

Table 1: Performance Comparison

Benchmark	Baseline	Modified JRuby	Ratio
fannkuch-redux	0.3 s	3 s	10
mandelbrot	1.62 s	10.6 s	6.6
pidigits	7.66 s	12.47 s	1.7
n-body	1.26 s	3.66 s	2.99
binary-trees	90 s	140.35 s	1.6
crypto - MD5	1.63 s	6.2 s	3.9
matrix multiplication	1.06 s	5.45 s	5.2
fasta	40.2 s	48.8 s	1.21
quick sort	6.22 s	32 s	5.14
merge sort	9.92 s	46.58 s	4.7

Since the compiler optimization had been excluded in this project, it had been ignored in the baseline considered to have a fair comparison. The baseline is unmodified otherwise. While these results show that there is a 1.2x - 10x slowdown in performance, this project protects confidentiality by sealing data leaks through implicit flows.

The Ratio column in Table 1 shows the slowdown of this project over the baseline. The fasta, pidigits, and binary-trees benchmarks had very few *if-else* constructs, which is evident in the Ratio column. Tests showed that the slowdown in execution was proportional to the usage of *if-else* construct and assignment statements inside it.

5.3 Incremental Performance Testing

Appendix B shows the script for the test case if-else-10K shown in Table 2. The if-else-10K has two *while* loops. The first loop has a simple *if-else* construct which involves secret operation, whereas the second loop does the identical operation over public data. This test case involves 10,000 secret operations and 9,999,000 public operations. The remaining test cases in Table 2 are similar to if-else-10K with an exception of incremental 10 percent increase in the ratio of secret operation to public operation, keeping the total number of operations constant across all the testcases. The Execution Time column shows that the execution time increases when there is a significant increase in the number of secret operations.

Table 2: Incremental Testing

No. of secret operations / 1B	Execution Time
if-else-10K	357 s
if-else-100K	428.9 s
if-else-1M	447.44 s
if-else-10M	501.62 s
if-else-100M	529.58 s
if-else-1B	565.65 s

CHAPTER 6

Conclusion and Future work

This project presents an implicit data flow monitor in JRuby that tracks information flow across security levels at runtime, and protects confidentiality. The performance results show that this project has some higher performance overhead than the baseline in order to prevent data leaks.

Extending this project to accomodate the following is the future scope of this project.

- Additional data types: In addition to Strings, the project can be extended to other data types such as: boolean, numbers, and objects.
- Additional language constructs: Extending this project to looping constructs, case statements and function calls is straightforward.
- Additional principals: Along with the simple *secret-public* lattice, this project can be extended to a lattice with more security levels [16].

LIST OF REFERENCES

[1] F. Qin, C. Wang, Z. Li, H.-s. Kim, Y. Zhou, and Y. Wu, ‘‘Lift: A low-overhead practical information flow tracking system for detecting security attacks,’’ in *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO 39. Washington, DC, USA: IEEE Computer Society, 2006, pp. 135--148. [Online]. Available: http://dx.doi.org/10.1109/MICRO.2006.29

[2] A. C. Myers, ‘‘Jflow: Practical mostly-static information flow control,’’ in *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL ’99. New York, NY, USA: ACM, 1999, pp. 228--241. [Online]. Available: http://doi.acm.org/10.1145/292540.292561

[3] D. E. Denning and P. J. Denning, ‘‘Certification of programs for secure information flow,’’ *Commun. ACM*, vol. 20, no. 7, pp. 504--513, July 1977. [Online]. Available: http://doi.acm.org/10.1145/359636.359712

[4] T. H. Austin and C. Flanagan, ‘‘Efficient purely-dynamic information flow analysis,’’ in *Proceedings of the ACM SIGPLAN Fourth Workshop on Programming Languages and Analysis for Security*, ser. PLAS ’09. New York, NY, USA: ACM, 2009, pp. 113--124. [Online]. Available: http://doi.acm.org/10.1145/1554339.1554353

[5] W. Chang, B. Streiff, and C. Lin, ‘‘Efficient and extensible security enforcement using dynamic data flow analysis,’’ in *Proceedings of the 15th ACM Conference on Computer and Communications Security*, ser. CCS ’08. New York, NY, USA: ACM, 2008, pp. 39--50. [Online]. Available: http://doi.acm.org/10.1145/1455770.1455778

[6] A. Sabelfeld and A. C. Myers, ‘‘Language-based information-flow security,’’ *IEEE J.Sel. A. Commun.*, vol. 21, no. 1, pp. 5--19, Sept. 2006. [Online]. Available: http://dx.doi.org/10.1109/JSAC.2002.806121

[7] B. W. Lampson, ‘‘A note on the confinement problem,’’ *Commun. ACM*, vol. 16, no. 10, pp. 613--615, Oct. 1973. [Online]. Available: http://doi.acm.org/10.1145/362375.362389

[8] A. C. Myers, L. Zheng, S. Zdancewic, S. Chong, and N. Nystrom, ‘‘Jif: Java information flow,’’ *Software release. Located at http://www.cs.cornell.edu/jif*, 2001.

[9] D. Volpano, C. Irvine, and G. Smith, ‘‘A sound type system for secure flow analysis,’’ *J. Comput. Secur.*, vol. 4, no. 2-3, pp. 167--187, Jan. 1996. [Online]. Available: http://dl.acm.org/citation.cfm?id=353629.353648

[10] D. King, B. Hicks, M. Hicks, and T. Jaeger, ‘‘Implicit flows: Can't live with 'em, can't live without 'em,’’ in *International Conference on Information Systems Security*. Springer, 2008, pp. 56--70.

[11] T. H. Austin and C. Flanagan, ‘‘Permissive dynamic information flow analysis,’’ in *Proceedings of the 5th ACM SIGPLAN Workshop on Programming Languages and Analysis for Security*, ser. PLAS '10. New York, NY, USA: ACM, 2010, pp. 3:1--3:12. [Online]. Available: http://doi.acm.org/10.1145/1814217.1814220

[12] D. Devriese and F. Piessens, ‘‘Noninterference through secure multi-execution,’’ in *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, ser. SP '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 109--124. [Online]. Available: http://dx.doi.org/10.1109/SP.2010.15

[13] T. H. Austin and C. Flanagan, ‘‘Multiple facets for dynamic information flow,’’ in *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '12. New York, NY, USA: ACM, 2012, pp. 165--178. [Online]. Available: http://doi.acm.org/10.1145/2103656.2103677

[14] W. M. McKeeman, ‘‘Peephole optimization,’’ *Commun. ACM*, vol. 8, no. 7, pp. 443--444, July 1965. [Online]. Available: http://doi.acm.org/10.1145/364995.365000

[15] ‘‘Ruby jruby measurements (64-bit ubuntu quad core) | computer language benchmarks game,’’ accessed 2017-03-05. [Online]. Available: http://benchmarksgame.alioth.debian.org/u64q/measurements.php?lang=jruby

[16] D. E. Denning, ‘‘A lattice model of secure information flow,’’ *Commun. ACM*, vol. 19, no. 5, pp. 236--243, May 1976. [Online]. Available: http://doi.acm.org/10.1145/360051.360056

APPENDIX A

Types of Nodes

- AliasNode
- AndNode
- ArgsCatNode
- ArgsNode
- ArgsPushNode
- ArgumentNode
- ArrayNode
- AssignableNode
- AttrAssignNode
- BackRefNode
- BeginNode
- BignumNode
- BinaryOperatorNode
- BlockAcceptingNode
- BlockArgNode
- BlockNode
- BlockPassNode
- BreakNode
- CallNode
- CaseNode
- ClassNode
- ClassVarAsgnNode
- ClassVarDeclNode
- ClassVarNode
- Colon2ConstNode
- Colon2ImplicitNode
- Colon2Node
- Colon3Node
- ComplexNode
- ConstDeclNode
- ConstNode
- DAsgnNode
- DNode
- DRegexpNode
- DStrNode
- DSymbolNode
- DVarNode
- DXStrNode
- DefNode
- DefinedNode
- DefnNode
- DefsNode
- DotNode
- EncodingNode
- EnsureNode
- EvStrNode

- FCallNode
- FalseNode
- FileNode
- FixnumNode
- FlipNode
- FloatNode
- ForNode
- GlobalAsgnNode
- GlobalVarNode
- HashNode
- IArgumentNode
- IScopedNode
- IScopingNode I
- fNode
- InstAsgnNode
- InstVarNode
- InvisibleNode
- IterNode
- KeywordArgNode
- KeywordRestArgNode
- LambdaNode
- ListNode
- LiteralNode
- LocalAsgnNode
- LocalVarNode
- Match2CaptureNode
- Match2Node
- Match3Node
- MatchNode
- MethodDefNode
- ModuleNode
- MultipleAsgnNode
- NewlineNode
- NextNode
- NilImplicitNode
- NilNode
- Node
- NonLocalControlFlowNode
- NthRefNode
- NumericNode
- OpAsgnAndNode
- OpAsgnConstDeclNode
- OpAsgnNode
- OpAsgnOrNode
- OpElementAsgnNode
- OptArgNode
- OrNode
- PostExeNode
- PreExe19Node
- PreExeNode
- RationalNode
- RedoNode

- RegexpNode
- RequiredKeywordArgumentValueNode
- RescueBodyNode
- RescueModNode
- RescueNode
- RestArgNode
- RetryNode
- ReturnNode
- RootNode
- SClassNode
- SValueNode
- SelfNode
- SplatNode
- StarNode
- StrNode
- SuperNode
- SymbolNode
- TrueNode
- UndefNode
- UnnamedRestArgNode
- UntilNode
- VAliasNode
- VCallNode
- WhenNode
- WhenOneArgNode
- WhileNode
- XStrNode
- YieldNode
- ZArrayNode
- ZSuperNode

APPENDIX B

Test case: if-else-10K

```
start = Time.now
i = 0
mid = 10000                 # varies
max = 1000000000

while i < mid do                        # secret
    balanceFile = "bal.txt".taint
    flag = "0".taint
    if balanceFile == "bal.txt"
        flag = "1"
    end
    i = i + 1
end

while i < max do                        # public
    balanceFile = "publicFile.txt"
    flag = "0"
    if balanceFile == "publicFile.txt"
        flag = "1"
    end
    i = i + 1
end
```

```
finish = Time.now
execTime = finish - start
print "\r\n"
print execTime
```