Compiler Design of a Policy Specification Language for Conditional Gradual Release

2018

Manasa Kashyap Harinath
University of Central Florida

COMPILER DESIGN OF A POLICY SPECIFICATION LANGUAGE FOR CONDITIONAL GRADUAL RELEASE

by

MANASA KASHYAP HARINATH
B.E. Sir M Visvesvaraya Institute of Technology 2013

A thesis submitted in partial fulfillment of the requirements
for the degree of Master of Science
in the Department of Computer Science
in the College of Engineering and Computer Science
at the University of Central Florida
Orlando, Florida

Summer Term
2018

Major Professor: Gary T. Leavens

ABSTRACT

Securing the confidentiality and integrity of information manipulated by computer software is an old yet increasingly important problem. Current software permission systems present on Android or iOS provide inadequate support for developing applications with secure information flow policies. To be useful, information flow control policies need to specify declassifications and the conditions under which declassification must occur. Having these declassifications scattered all over the program makes policies hard to find, which makes auditing difficult. To overcome these challenges, a policy specification language, 'Evidently' is discussed that allows one to specify information flow control policies separately from the program and which supports conditional gradual releases that can be automatically enforced. I discuss the Evidently grammar and modular semantics in detail. Finally, I discuss the implementational details of Evidently compiler within the Xtext language development environment and the implementation's enforcement of policies.

I dedicate this thesis to my parents Gayathri Tekal Ramarao and Harinath Kashyap Bheema Rao who have loved me unconditionally and whose good examples have taught me to work hard for the things that I aspire to achieve.

ACKNOWLEDGMENT

I take this opportunity to express my great appreciation towards Professor Dr. Gary T. Leavens for his invaluable and great advice and support. Without his advice and guidelines, I couldn't accomplish this work. Dr. Leavens not only helped me in going through this research and accomplish the work, but even more importantly, showed me how to look from different perspectives into the same problem. I would also like to thank the members of my thesis committee, Professors Damla Turgut and Liqiang Wang for their advice and guidance during the entire process. Special thanks to my best friend, Siddarth Vellakovil Rajamani, who has been a constant source of support and encouragement during the challenges of graduate school and life.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF ACRONYMS

ANTLR	Another Tool for Language Recognition
AST	Abstract Syntax Tree
CGR	Conditional Gradual Release
EMF	Eclipse Modelling Framework.
IDE	Integrated Development Environment
JDK	Java Development Kit
JML	Java Modeling Language

CHAPTER ONE: INTRODUCTION

In software systems that operate on data with different sensitivity levels, the challenge is to provide security assurance by controlling the information flow within the system. Access control has been a tool to prevent secure information from being disseminated. Access control verifies the program's access rights at the point of access, and either grants or denies permission to access the secure information.

Information flow control monitors the flow of information through the program thus ensuring its information security. An ideal secure system must possess two major properties: Confidentiality and Integrity. *Confidentiality* suggests that no secure information is released to insecure program locations and *Integrity* suggests that no insecure program location may influence the values stored at the secure program locations. *Noninterference* is a security property that specifies constraints on the information between program locations.

A system with perfect Confidentiality and Integrity is not always ideal when it comes to real world applications. For example, consider a system that provides a login prompt to a user. If the user enters the incorrect login credentials, the system must deny the user access. However, this action reveals something about what values the username and password are not and therefore leaks information. Such a system does not have the noninterference property. Since the noninterference property is not practical in real systems, some amount of release of the information must be allowed. Programs that release information are said to *declassify* that information.

Therefore, the user (designer) must answer the questions such as: how much information should be released? What information should be released? Under what constraints information should be released? The answer to these questions are specified by information flow control policy specifications.

"Evidently" is a specification language designed for specifying information flow control policies. In Evidently, the Information Flow Control policy specifications are separated from the actual program. Separating the policy specifications from the implementation of a system permits the policy to be modified without changing its underlying implementation. Additionally, when policy specifications are separated from the program code, policy scattering and tangling of policies throughout the program code can be avoided. This feature of Evidently makes auditing easier.

Tangling occurs when the module in the application includes repeated code for each type of Conditional gradual release, in many modules. *Scattering* occurs when the information flow control policies are mixed with the business logic of the application.

Problem and its importance

As discussed, monitoring information flow between program locations is an inevitable aspect of information flow control policies. Chong and Myers [1] have described a declassification policy language where policies are written along with the program code. This means the expressions for the declassifications are interwoven with the program code, making auditing difficult. Additionally, when the user changes the program code, care must be taken to examine the complete program to verify all the declassification expressions. Similarly, in other specification languages such as, JIF [4], JFlow [5], JRif [6] and Paragon [7] the declassifications rules are specified along with the program code.

Denning [2] presents a compiler-time methodology that verifies the program for secured information flow, before the program is even executed. However, statically-typed policy specification is not always recommended. Real computing systems have information flow that vary dynamically and that cannot be completely determined at compile time. Zheng

and Myers [3] proposed a policy specification language, λ_{DSec} to securely monitor and manipulate information with dynamic secure labels.

In λ_{DSec} , the dynamic labels can be used as type annotations to support static analysis. The type system of λ_{DSec} enforces noninterference ensuring secure information flow control.

Evidently is a specification policy language that describes information flow control policies and declassification separate from the program code thereby avoiding scattering and tangling. Writing policies separately from the program code enables the policies to be reused. In Evidently, the security labels are designed in the models using flowpoints which is point of contact through which the policy interacts with the program code. Therefore, if the program code changes, only the model need to be modified.

In this thesis, Evidently language is discussed in detail describing Evidently semantics and constructs. An implementation of Evidently is discussed, which describes enforcement of information flow policies for programs written in the Java.

Background on Evidently

In Evidently, the security policy is defined by labelling each location in the program code with a value that represents the security level of that location. For instance, a program location with a high security property might be labelled as ‘H’ and a program location with a lower security property might be labelled as ‘L’. The declassification rules apply constraints on the information flow from ‘H’ to ‘L’.

Declassification

Evidently uses Conditional Gradual Release, for declassification, i.e., rather than the delimited style of declassification, wherein declassifications are arbitrarily powerful

relabelings. In Evidently, policies may express the conditions for declassification. In Evidently, the declassification properties of Java program are specified by policies. The policies answers questions such as: 'what' information is being released?, 'where' is this information released to in the program code?, and 'when' is this information allowed to flow?

To be concrete, let us consider a hypothetical example of a free HBO subscription account. This account holds good for a month, after which, the account is terminated. Suppose, the user who still has a valid free HBO subscription, wants to watch an episode of their favorite series. Let us analyze the application of policies through this example.

What. While specifying information flow control policy, *what* indicates the information that is being allowed to flow. In our example, the "what" could be the episode of a particular series that the user requests to watch, i.e., a video. The "what" could also be more fine-grained—for example, even though, the user has a free subscription and can watch that episode, there might be a constraint on the duration of the episodes that user is allowed to watch. This information flow and its constraints are specified by Evidently abstractions called 'Models' and 'Properties' which will be discussed in the next chapter.

Where. In the context of declassification, "where" can refer to policies that describe release to locations in the code. These locations are represented by variables or fields or even various security levels. In our example, where suggests the security level say, from the DATABASE of HBO to *NETWORK*. In Evidently, the "where" feature is represented by an abstraction called 'Levels' which will be discussed in the next chapter.

When. In Evidently, this dimension can be specified using properties in the policies. The 'When' dimension permits declassification under certain arbitrary conditions. Property specifications will be discussed in detail in the next chapter. Considering our example of HBO subscription, the episode will be allowed to be viewed by the user while he has a valid

subscription. Once the subscription expires, the user will not be able to watch the episode. In Evidently, the when dimension is encoded by combining properties inside the policy.

Overview of the thesis

Chapter Two explains more details about Evidently syntax and semantics. Chapter Three discusses the implementation details of the Evidently compiler and enforcement of policies. Evidently is implemented using Language Development framework Xtext. Chapter Four also provides a brief insight about the Future work and Summary of the paper. The Appendix contains the compiler code to the compiler.

CHAPTER TWO: SECURITY POLICIES USING EVIDENTLY

As discussed, Evidently is a policy specification language with centralized declassification rules that are not scattered throughout a program. This enables the reusability of the policies. When policies are scattered throughout the program code, any modification done to the program, might change the semantics of the policies. To achieve centralization of policies in Evidently, the user describes an abstraction of the program, which can be used throughout the policy. Evidently provides four abstractions namely: Models, Lenses, Projections, and Levels.

Models, Projections, and Levels are used in describing a Policy, which is the fourth and main abstraction. We discuss the properties and functionalities of all these abstractions starting with models, which describes the program locations that user wants to monitor information flow.

Models

Models in Evidently contain references to the data locations that the user wishes to write declassification rules about. A model may define single or multiple data locations by declaring flowpoints. Using these flowpoints, models might also contain properties to specify constraints on the flowpoints. We discuss flowpoints and properties in detail in the next section. The syntax of Models in Evidently is provided in the figure 1.

Flowpoints

flowpoints are defined in the models to refer to data locations in a program's code. As discussed, in Evidently, the primary functionality of models is to identify data locations in the

program code. A model might contain one or more flowpoints. Evidently allows users to be specific or generic by providing a pointcut-like-predicates and operators for describing sets of data locations. Figure 2 provides a brief description of these predicates.

Predicate Expressions and Predicate Operators

The flowpoints are comprised of predicate expressions. Predicates are combined to form predicate expressions that describes a set of data locations. The predicate expressions may be joined using && and || operator. The && operator denotes the intersection of two sets of data locations and || denotes their union. Figure 1 shows an example:

```
flowpoint adminMode:boolean = {
              within(AsmTrial) && field("adminMode")
       }
```

Figure 1: Predicate Expressions and Predicate operators

Model ::= **model** *Id* { *MBElems* }

MBElems ::= *MBElem* | *MBElems MBElem*

MBElem ::= *Use* | *Flowpoints* | *Property*

Flowpoints ::= **flowpoints** *Id* : *ProgType*={ *EvExp* }
| **flowpoints** *Id*={ *EvExp* }

EvExp ::= (*EvExp*)
| *EvPredExt* (*MethDescs*)
| *EvScopePred* (*Id*)
| !*EvExp* | *EvExp* && *EvExp* | *EvExp* || *EvExp*

EvPredExt ::= **execution** | **resultof** | **this** | **cflow**

EvScopePred ::= **within** | **typeof** | **named** | **field**

MethDescs ::= *MethDesc* | *MethDescs*, *MethDesc*

MethDesc ::= *IdOrStar* | *IdOrStar* (*MethFormalDescs*)
| *TypeOrStar IdOrStar* (*MethFormalDescs*)

IdOrStar ::= *Id* | *

TypeOrStar ::= *ProgType* | *

MethFormalDescs ::= ... | *TypeOrStar**,

Property ::= **property** *Id* : *ProgType EqSpec* { *PropBody* }
| **property** *Id EqSpec* { *PropBody* }
| **property** *Id PFormals* : *ProgType EqSpec* { *PropBody* }
| **property** *Id PFormals EqSpec* { *PropBody* }

EqSpec ::= = *Specification* | =

Specification ::= *Requires Assignable Ensures*
| *Assignable Ensures*
| *Requires Ensures*
| *Ensures*

Requires ::= **requires** *Exp*;

Assignable ::= **assignable** *Exp*;

Ensures ::= **ensures** *Exp*;

PropBody ::= *Exp*

PFormals ::= (*Formals*)

Formals ::= *Formal* | *Formals*, *Formal*

Formal ::= *Id* : *ProgType* | *Id*

Figure 2: Syntax of Model in Evidently

Predicate Name	Meaning
execution	The set of data locations within the execution of the supplied argument
resultof	The set of data locations that are the result of the argument's execution
this	The set of data locations where the **this** keyword produces specified type
cflow	The set of data locations within the control flow of the supplied argument
within	The set of data locations contained within the supplied namespace.
typeof	The set of data locations that have the type of the argument
named	Program identifier matches the supplied identifier
field	The Set of data locations where the object field name matches the supplied argument

Figure 3: Flowpoint predicates

Properties

In models, *properties* specify the attributes of the model that are supposed to be released by the policy. They also denote how much of this information is released by the policy. This feature is enabled using projections which will be discussed in the next section.

Specification of Properties

Having only properties will not make information flow completely secure. For example, if we have a method which displays only last 4 digits of credit card number and we

have defined properties to ensure this flow in the models. Suppose, if the method changes, it might degrade the quality of security provided by the policy. To overcome this challenge, there are specifications attached to the properties. These specifications are called pre- and post-conditions specifications. In Evidently, pre-, and post- conditions are achieved using **requires** keyword and **ensures** keywords respectively.

Levels

In previous sections, we have been using generic parameters H and L to represent high security and low security labels. In Evidently, these security labels are specified using the abstraction *levels*. Once defined, these levels are recognized within a policy which we will discuss in the next section. A level may be defined as a sink, which denotes the program locations to which information may flow to, or a source, which denotes program locations from which information may flow. The levels we show below are derived from Sparta [8]. Figure 3 shows the syntax of levels in Evidently.

```
Levels          ::= levels Id { LevelsSpec }
LevelsSpec      ::= LevelElem
                  | LevelsSpec LevelElem
LevelElem       ::= LevelKind {LevBodyElems}
LevelKind       ::= sink
                  | source
                  | both
LevBodyElems    ::= Id
                  | LevBodyElems , Id
```

Figure 4: Syntax of Levels in Evidently

Policies

In Evidently, *policies* are responsible for bringing all abstractions together to express security specifications of a program. In this section we will discuss the policy features of Evidently as well as show examples of how to encode common security properties in Evidently policies. In Evidently, the release keyword is used to specify the actual flow. A release statement in the policy contains a property or flowpoint and set of templates F -> T which denoted the generic information flow. The set of templates, F->T represent the *what* and *where* dimensions respectively. 'F' and 'T' may be instantiated with any two declared labels. Evidently supports parametric polymorphism, similar to Java's generics. The set of templates, F->T is used within the release tuple. The *when* dimension is denoted by the keywords, **when** and **unless.**

The Root Policy

In Evidently, the information flow policy of an application is not written within the program's code. However, to support better modularity, Evidently abstractions such as models,

levels, projections and policies may be written in multiple files. To combine all these elements together, our system requires a root policy file, *policy.epl*, which is the starting point into an applications information flow policy. Within this file the user may import other policies, and set the enforcement level of the various policies. We show the syntax of policy files in Evidently in Figure 4.

Policy	::=	**policy** *Id* { *PoBElems* }
		| **policy** *Id GParams* { *PoBElems* }
GParams	::=	⟨ *Ids* ⟩
Ids	::=	*Id* | *Ids*, *Id*
PoBElems	::=	*PoBElem* | *PoBElems PoBElem*
PoBElem	::=	*Use* | *Release* | *EnforceDirective*
Use	::=	*UsePolicy*
		| *UseElem*
UsePolicy	::=	**use policy** *Id*;
		| **use policy** *Id* **as** *Id*;
		| **use policy** *Id GArgs*;
		| **use policy** *Id GArgs* **as** *Id*;
UseElem	::=	**use model** *Id*;
		| **use model** *Id* **as** *Id*;
GArgs	::=	⟨ *GenericArgs* ⟩
GenericArgs	::=	*GenericArg* | *GenericArgs*, *GenericArg*
GenericArg	::=	*Id* | {*Ids*} | ⋆
Release	::=	**release** (*EvidentlyId*, *LevelFlow*) ={*RTimes*}
		| **release** (⋆ , *LevelFlow*) ;
LevelFlow	::=	*GenericArg* –> *GenericArg*
RTimes	::=	*RTime* | *RTimes RTime*
RTime	::=	**when** { Exp } | **unless** { Exp }
EnforceDirective	::=	**enforce** *EnforceMode Id*;
EnforceMode	::=	**static** | **runtime**

Figure 5: Syntax of Policy in Evidently

CHAPTER THREE: EVIDENTLY COMPILER IMPLEMENTATION

Introduction

Implementation of Evidently for the Java language is discussed in this section. I used the Xtext language development tool [9] to implement Evidently. Prior to using Xtext, Evidently's grammar was implemented using Jastadd [10]. In this method, the Lexer and Parser are written using JFlex [11] and Beaver [12] respectively. JastAdd, a meta compilation system that supports Reference Attribute Grammars was used to implement Evidently.

Overview of Xtext

Xtext is an ideal language development workbench, to implement Evidently. In Xtext, the grammar is specified in the '.xtext' file. From this specification, Xtext automatically generates the parser and lexer in Java. Like ANTLR, Xtext only needs the grammar specification to create the AST and corresponding Java classes. To implement type checking, validation, and code generation, Xtext provides a structured language called 'Xtend' [13]. In our implementation, the validation checks, type checking and code generation are written using Xtend.

Architecture

In this section we discuss the implementational details of Evidently compiler. First, we discuss how the grammar is written using Xtext DSL, Validations using Xtend and, describe

how to enforce the information flow control policies during runtime.

Grammar

In Xtext, the grammar is written in the '.xtext' file. The Evidently grammar is written in 'Evidently.xtext. The first rule in the grammar defines where the parser starts and the type of the root element of the grammar and the AST. For Evidently, '*PolicyFile'* is the root element of the AST which contains one or more Evidently abstractions such as Levels, Models and Policy. The complete grammar of Model, Level and Policy is described in the Appendix.

The parser algorithm of Xtext does not deal with left recursive rules. A rule is said to be left recursive when the rule's non-terminal (each rule in a CFG has a non-terminal on the left hand side) refers to itself. To avoid left recursion one must use "left factoring" to remove left recursion. For example, consider the left-recursive rule A -> A a | b. This rule can be replaced by adding another rule say, A', as follows:

A-> b A'

A' -> a A' | (empty)

Cross References

To refer to another element in the grammar, Xtext allows cross references using EMF. The Eclipse Modelling Framework (EMF) [14] is a set of plug-ins available for Eclipse IDE that allows the developer to create the meta-model of the application. Generally, EMF is used to define these meta-models of the application and generate corresponding Java implementation classes.

In Evidently, the root policy, 'policy.epl' generally contains references to models, policies and the levels which are handled using the cross-reference property of Xtext. In EMF,

the element type is not any primitive data type but, is a reference. This reference is an instance of *EReference* (from EMF). Xtext provides a mechanism to store all the cross-referencing objects using a structure called, 'Index'. This index store stores only the meta-data of the object using *IEObjectDescription* elements. Xtext use namespaces to allow elements with the same name in different namespaces. To achieve this functionality, Xtext provides the type *QualifiedName*. Below is a code snippet from Evidently.xtext (Appendix) to demonstrate the usage of namespace with cross reference.

```
UseElem:
    'use' kind='model' name=[Model|QualifiedName] ';' // ';'
    | 'use' kind='model' name=[Model|QualifiedName] 'as' alias=ID ';';
```

Figure 6: Example of Cross reference and Namespace in Evidently

Validations

Parsing is only the first stage of language development and overall correctness of an Evidently policy is not completely determined during parsing. Having constraint checks during the parsing process makes grammar specifications more complex and difficult to understand. Hence, it is better to do as little as possible in the grammar and as much as possible in *Validation.* In the implementation, all the validations are written in the 'EvidentlyValidator.xtend' file. Xtext performs validation by invoking all the methods annotated with *@Check.* Inside these methods, we carry out the semantic checks on the Evidently elements and if the check fails, an *error* method is called. The *error* method is passed with the following: 1) A *message* for the error 2) Information about the EObject against which the error was reported. The source code for all the validations is provided in the Appendix. An example method is showing in Figure 8 which checks for duplicate models.

```
@Check
def checkDuplicateModelsInFiles(Model m ) {
    print("hey")
    val modelName= m.fullyQualifiedName
    print (modelName)
    m.getVisibleClassesDescriptions.forEach[
        desc |
        if (desc.qualifiedName == modelName &&
                desc.EObjectOrProxy != m     &&
                desc.EObjectURI.trimFragment != m.eResource.URI) {
            error(
                "The type " + m.fullyQualifiedName+ " is already defined",
                EvidentlyPackage::eINSTANCE.model_Name,DUPLICATE_CLASS)
            return
        }
    ]
}
```

Figure 7 : Checking for Duplicate Models

Policy Enforcement

To demonstrate the runtime enforcement of policies, we have implemented Evidently to work with the Java language. This section provides details on Evidently policy enforcement during runtime.

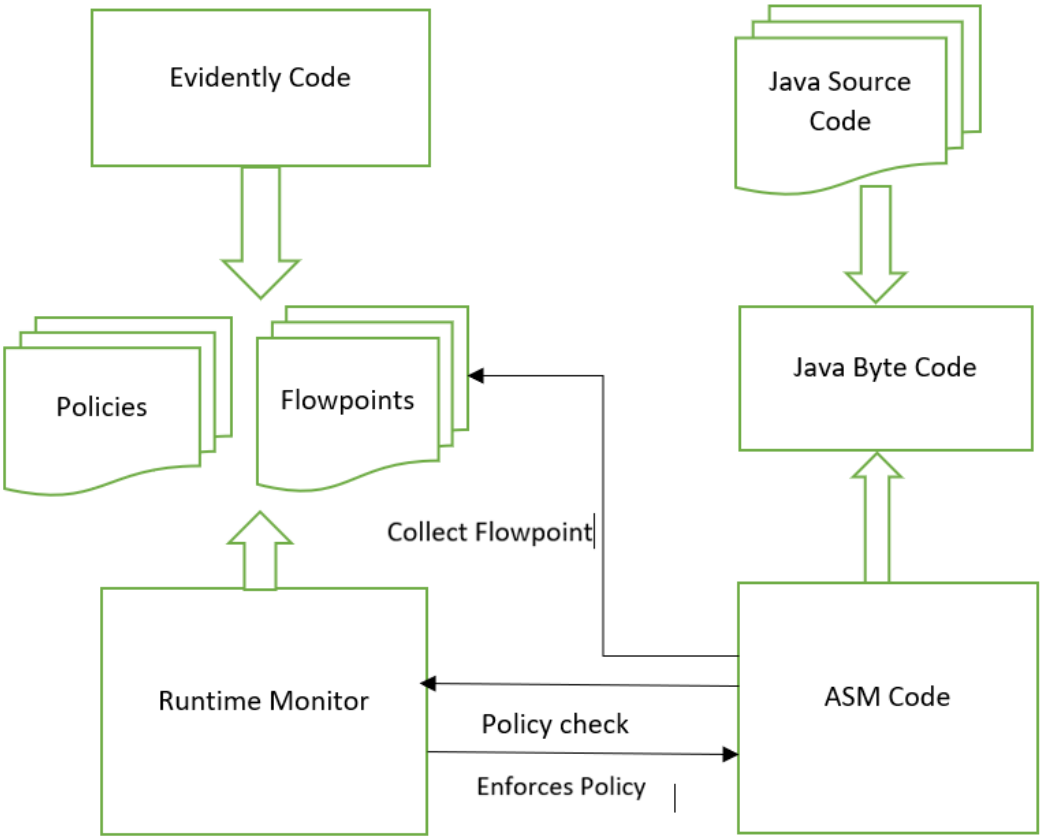

Figure 8: Enforcement of Policies

Figure 9 gives an overview of our implementational details of the Evidently policy enforcement. Evidently makes use of ASM tool to perform additional bytecode generation. ASM [15] is a tool for the Java language designed for runtime class transformation and generation. In Evidently, ASM transforms the Java program by instrumenting bytecodes as per the underlying Evidently policy.

In Evidently, flowpoints describe the data locations in the Java program and are an integral part of the models. A model can contain one or more flowpoints. We used Xbase to map flowpoint constructs of the Evidently model to the corresponding Java model elements. This mapping is specified by implementing an **IJvmModelInferrer** interface. Since we use Xbase in our grammar, Xtext automatically generates an Xtend stub class,

EvidentlyJvmModelInferrer in the jvmmodel sub-package. The EvidentlyJvmModelInferrer has an **infer** method to create Java model elements, associate them to Evidently elements, and pass them to the **acceptor** which implements the mapping. The **JvmTypesBuilder** is an extension that provides an API to create Java model elements such as: toClass, toMethod, toField and so on. The generated flowpoint class contains two methods, getField() and getCodeClass() that returns the name of data field in the target Java program and its corresponding class respectively.

We leveraged the ASM tool to identify the data locations to instrument the policy specifications. The target source code is compiled into Java bytecode and the ASM tool uses this bytecode to inject additional bytecode to the target source code. The ASM tool checks all the class variables and method variables and verifies if there are flowpoints associated with the fields. We handle the policy enforcement by instrumenting the code with special calls to our runtime monitor. The runtime monitor checks the generated policy and corresponding flowpoints and generates appropriate code to instrument into the target source code.

The runtime monitor checks the generated policy file for information to be released and conditions under which it should be released. The 'information' to be released is the Flowpoint and the condition is specified within the 'when' clause. If the condition for the information release is not met, an exit routine is called, which stops the information from being released.

```
public class Test {

		public static int secureNumber = 10;
		public static boolean adminMode=false;

		public static void add(){


			boolean isOk= true;

			if(isOk){
			//Exit code should be instrumented here
				//We should not allow this
				System.out.println(secureNumber);
			}
			else{
				//No need to add anything here
				System.out.println("Not an admin");
			}
		}
		public static void main(String[] args) {
			add();
		}

}
```

Figure 9: Java source code

Figure 8 shows a sample Java source code, we wish to write a policy for. We should be able write a policy in Evidently which restricts the release of 'secureNumber', if 'adminMode' is false. Figure 9 shows the corresponding policy example.

```
model AdminAccess {

	flowpoint adminMode:boolean = {
		within(AsmTrial) && field("adminMode")
	}i

	flowpoint secureNumber:int = {
		within(AsmTrial) && field("secureNumber")
	}
}

policy ReleaseSecureNumberToAdmin {

	use model AdminAccess;


	release(AdminAccess.secureNumber, X->Y){
		when {
			adminMode == true
		}
	}
}
```

Figure 10: Evidently policy

We define a model 'AdminAccess' with two flowpoints, 'adminMode' and 'secureNumber'. Furthermore, we define ReleaseSecureNumberToAdmin policy, to use AdminAccess model which allows the policy to access all the flowpoints and properties in the model. The **release** statement specifies the flowpoint (what) that is to be released based on the condition that is specified by **when** clause. The second parameter within the release tuple (F->T), specifies **where** dimension. In Figure 8, System.out.println(secureNumber); is executed only when the adminMode is 'true'. Otherwise, an exit statement is called before 'secureNumber' is accessed.

CHAPTER FOUR: CONCLUSION

Evidently is a specification language that defines centralized information flow control policies separate from the program code. This avoids policies being scattered throughout the program code thereby making auditing easier. I discussed how the Evidently grammar and Validation rules are written with XText language development tool. Finally, I discussed how policies are enforced during runtime using the concepts of bytecode instrumentation.

Future Work

The current implementation is targeted to monitor the information flow of assignments in the program code. The implementation can be extended to track and control the information flow during method calls, and conditional statements. Currently, Evidently is implemented to control information flow in a program written in Java Language. The implementation can be extended to control information flow in the program written in various other object-oriented programming language such as Python.

APPENDIX: CODE FOR THE COMPILER

Grammar

```
grammar org.evidently.Evidently with org.eclipse.xtext.xbase.Xbase

generate evidently "http://www.example.org/evidently/Evidently"

import "http://www.eclipse.org/emf/2002/Ecore" as ecore
import "http://www.eclipse.org/xtext/common/JavaVMTypes" as jvmTypes

PolicyFile:
    (elements+=FileElem)*;

ProgId:
    ID;

EvidentlyId:
    ID ('.' ID)*;

BooleanLiteral:
    "true" | "false";


ProgType:
    {ProgType}type="byte" prime=ProgTypePrime
    |{ProgType}type="String" prime=ProgTypePrime
    |{ProgType} type="short" prime=ProgTypePrime
    |{ProgType} type="int" prime=ProgTypePrime
    |{ProgType} type="long" prime=ProgTypePrime
    |{ProgType} type="float" prime=ProgTypePrime
    |{ProgType} type="double" prime=ProgTypePrime
    |{ProgType} type="char" prime=ProgTypePrime
    |{ProgType} type="boolean" prime=ProgTypePrime
    |{ProgType} type="void" prime=ProgTypePrime
    |{ProgType} type="File" prime=ProgTypePrime
    |{ProgType} javaType=[jvmTypes::JvmType|QualifiedName] prime=ProgTypePrime // takes the place of ProgId rules.
;
```

Figure 11: Evidently Grammar part a

```
35
36 ProgTypePrime:
37     ("[" "]" ProgTypePrime)?;
38
39 FileElem:
40     ImportLevels
41     |Policy
42     |Model
43     |Levels
44     |AxiomsImpl
45     ;
46
47 AxiomsImpl:
48     axioms=Axioms
49     |imports=XImportSection
50 ;
51
52 ModelImpl:
53     model=Model
54     |imports=XImportSection
55 ;
56
57 ImportLevels:
58     'using' 'levels' importedNamespace=[Levels|QualifiedNameWithoutWildcard]
59     ;
60
61 Policy:
62     'policy' name=ID '{' (elements+=PoBElem)* '}'
63     | 'policy' name=ID params=GParams '{' (elements+=PoBElem)* '}';
64
65 GParams:
66     '<' paramOne+=ID (',' params+=ID)* '>';
67
```

Figure 12: Evidently Grammar part b

```
PoBElem:
    use=Use
    | release=Release
    | enforce=EnforceDirective;

Use:
    usePolicy=UsePolicy
    | useModel=UseElem;

    //USe namespace
UsePolicy:
    'use' 'policy' policy=[Policy|QualifiedName] ';'
    | 'use' 'policy' policy=[Policy|QualifiedName] 'as' alias=ID ';'
    | 'use' 'policy' policy=[Policy|QualifiedName] args=GArgs ';'
    | 'use' 'policy' policy=[Policy|QualifiedName] args=GArgs 'as' alias=ID ';'
    ;

GArgs:
    "<" args+=GenericArg (',' args+=GenericArg)* ">";


GenericArgRule:
    name=ID
    | '*' name="*"
    | "{" names+=ID (',' names+=ID)* '}'
;

UseElem:
    'use' kind='model' name=[Model|QualifiedName] ';' // ';'
  //| 'use' kind='model' importedNamespace=QualifiedNameWithoutWildcard ';'
    | 'use' kind='model' name=[Model|QualifiedName] 'as' alias=ID ';';

QualifiedNameWithoutWildcard:
    QualifiedName '.*'?;
```

Figure 13: Evidently grammar part c

```
104
105 PropertyOrFlowpoint:
106     Flowpoints | Property;
107
108 Release:
109     'release' '(' what=[PropertyOrFlowpoint|QualifiedNameWithoutWildcard] ',' flow=LevelFlow ')' '=' '{' (releases+=RTime)* '}'
110     |'release' '(' '*' ',' flowlevel=LevelFlow ')' ';';
111
112 LevelFlow:
113     from=GenericArgRule "->" to=GenericArgRule;
114
115 RTime:
116     type='when' '{' condition=XExpression '}'
117     | type='unless' '{' notCondition=XExpression '}';
118
119
120 EnforceDirective:
121     'enforce' mode=EnforceMode policy=ID;
122
123 EnforceMode:
124     'static' | 'runtime';
125
126 Model:
127     'model' name=ID '{' (elements+=MBElem)* '}';
128
129 MBElem:
130     modeluse=Use
131     |flow= Flowpoints
132     |property= Property
133     ;
134
135 Flowpoints:
136     'flowpoints' name=ID ':' type=ProgType '=' '{' body=EvBody '}'
137     | 'flowpoints' name=ID '{' body=EvBody '}';
138
139 EvBody:
```

Figure 14: Evidently Grammar part d

```
139 EvBody:
140     EvExpression;
141
142 EvExpression:
143     EvMulOrDiv;
144
145 EvMulOrDiv returns EvExpression:
146     ExpPlus (
147         {EvMulOrDiv.left=current} op=('*'|'/')
148         right=ExpPlus
149     )*
150 ;
151 ExpPlus returns EvExpression:
152   ExpMinus ({ExpPlus.left=current} '+' right=ExpMinus)*;
153
154 ExpMinus returns EvExpression:
155  EvMod ({ExpMinus.left=current} '-' right=EvMod)*;
156
157 EvMod returns EvExpression:
158     EvAnd ({EvMod.left=current} '%' right=EvAnd)*;
159
160 EvAnd returns EvExpression:
161     EvOr ({EvAnd.left=current} '&&' right=EvOr)*;
162
163 EvOr returns EvExpression:
164     Comparison ({EvOr.left=current} '||' right=Comparison)*;
165
166 Comparison returns EvExpression:
167     Equality (
168         {Comparison.left=current} op=(">="|"<="|">"|"<")
169         right=Equality
170     )*
171
```

Figure 15: Evidently Grammar part e

```
;
Equality returns EvExpression:
    EvExpBase ({Equality.left=current} op=("=="|"!=")
        right=EvExpBase
    )*
;
EvExpBase returns EvExpression:
    '(' exp=EvExpression ')'
    | predicate=EvPredExtended '(' description=MethDescs ')'
    | predicate=EvScopePred '(' id= EvidentlyId')'
    | '!' EvMulOrDiv
    |atomic=Atomic;

Atomic returns EvExpression:
    {IntConstant} value=INT |
    {StringConstant} value=STRING |
    {BoolConstant} value=('true'|'false') |
    {VariableRef} variable=[Formal]
    ;


EvPredExtended:
    'execution' | 'resultof' | 'this' | 'cflow';

EvScopePred:
    'within' | 'typeof' | 'named' | 'field';

MethDescs:
    elements+=MethDesc (',' elements+=MethDesc)*;

MethDesc:
    name=IdOrStar
    | name=IdOrStar '(' signature=(MethFormalDescs)? ')'
    | type=TypeOrStar name=IdOrStar '(' signature=(MethFormalDescs)? ')';

```

Figure 16: Evidently Grammar part f

```
206
207 IdOrStar:
208     EvidentlyId
209     | '*';
210
211 TypeOrStar:
212     {TypeOrStar} '*'
213     | ProgType;
214
215 MethFormalDescs:
216     desc='...'
217     | descs+=TypeOrStar (',' descs+=TypeOrStar)*;
218
219 Property:
220     'property' var=Formals (spec=Specification)? '{' (expression=XExpression) '}'
221     | 'property' propertyName=ID formals=PFormals (':' type=ProgType)? (spec=Specification)? '{' (expression=XExpression) '}';
222
223 Axioms:
224     arr='axioms' '{' elements+=AxElems '}'
225 ;
226
227 AxElems:
228     package=PackagedMethod ':' level=[Levels|QualifiedName]
229 ;
230
231 PackagedMethod:
232      {PackageMethod} package= IdOrStar '.' method=IdOrStar '(' signature=(MethodLevelDescs)? ')'
233     | {UnnamedPackageMethod} method=IdOrStar '(' signature=(MethodLevelDescs)? ')'
234 ;
235
236 MethodLevelDescs:
237
238     desc='...'
239     | descs+=IdOrStar (',' descs+=IdOrStar)*
240 ;
```

Figure 17: Evidently Grammar part g

```
Specification:
    {Specification} (requires+=Requires)+ (assignable+=Assignable)* (ensures+=Ensures)+
    | {Specification} (assignable+=Assignable)+ (ensures+=Ensures)+
    | {Specification} (ensures+=Ensures)+;

Requires:
    'requires' expression=XExpression ';';

Assignable:
    'assignable' expression=XExpression ';';

Ensures:
    'ensures' expression=XExpression ';';

PFormals:
    '(' formal=Formals ')';

Formals:
    elements+=Formal (',' elements+=Formal)*;

Formal:
    name=ID ':' type=ProgType;

Levels: 'levels' name=ID '{' (elements+=LevElem)* '}';

LevElem:
    kind=LevelKind '{' elements=(LevelBodyElems)? '}'
;

LevelKind:
    'sink' | 'source' | 'both'
;
```

Figure 18: Evidently Grammar part h

```
LevelBodyElems:
    elements+=LevelBodyElem (',' elements+=LevelBodyElem)*
;
LevelBodyElem:
    name=ID
;
GenericArg:
    arg=[LevelBodyElem|QualifiedName]
    | '*' star="*"
    | '{' elements+=[LevelBodyElem|QualifiedName] (',' elements+=[LevelBodyElem|QualifiedName])* '}'
;
```

Figure 19: Evidently Grammar part i

Validation Rules

```
 4 package org.evidently.validation
 5 import org.evidently.evidently.Levels
36
37 /**
38  * This class contains custom validation rules.
39  *
40  * See https://www.eclipse.org/Xtext/documentation/303_runtime_concepts.html#validation
41  */
42 class EvidentlyValidator extends AbstractEvidentlyValidator {
43
44     @Inject ResourceDescriptionsProvider rdp
45
46     @Inject IContainer$Manager cm
47
48     @Inject extension IQualifiedNameProvider
49
50     public static val DUPLICATE_CLASS = "org.Evidently.DuplicateClass"
51     public static val DUPLICATE_NAME = "org.Evidently.DuplicateName"
52
53
54     def getVisibleClassesDescriptions(EObject o) {
55         o.getVisibleEObjectDescriptions(EvidentlyPackage::eINSTANCE.fileElem)
56     }
57
58     def getVisibleEObjectDescriptions(EObject o, EClass type) {
59         o.getVisibleContainers.map[ container |
60             container.getExportedObjectsByType(type)
61         ].flatten
62     }
63
```

Figure 20: Validation Rules part a

```
def getVisibleContainers(EObject o) {
    val index = rdp.getResourceDescriptions(o.eResource)
    val rd = index.getResourceDescription(o.eResource.URI)
    if (rd !== null)
        cm.getVisibleContainers(rd, index)
    else
        emptyList
}

@Check
def checkDuplicateModelsInFiles(Model m ) {
    print("hey")
    val modelName= m.fullyQualifiedName
    print (modelName)
    m.getVisibleClassesDescriptions.forEach[
        desc |
        if (desc.qualifiedName == modelName &&
                desc.EObjectOrProxy != m     &&
                desc.EObjectURI.trimFragment != m.eResource.URI) {
            error(
                "The type " + m.fullyQualifiedName+ " is already defined",
                EvidentlyPackage::eINSTANCE.model_Name,DUPLICATE_CLASS)
            return
        }
    ]
}
```

Figure 21: Validation Rules part b

```
72
73    @Check
74    def checkDuplicateModelsInFiles(Model m ) {
75        print("hey")
76        val modelName= m.fullyQualifiedName
77        print (modelName)
78        m.getVisibleClassesDescriptions.forEach[
79            desc |
80            if (desc.qualifiedName == modelName &&
81                    desc.EObjectOrProxy != m     &&
82                    desc.EObjectURI.trimFragment != m.eResource.URI) {
83                error(
84                    "The type " + m.fullyQualifiedName+ " is already defined",
85                    EvidentlyPackage::eINSTANCE.model_Name,DUPLICATE_CLASS)
86                return
87            }
88        ]
89    }
90    @Check
91    def checkDuplicatePolicyInFiles(Policy p ) {
92
93        val policyName= p.fullyQualifiedName
94
95        p.getVisibleClassesDescriptions.forEach[
96            desc |
97            if (desc.qualifiedName == policyName &&
98                    desc.EObjectOrProxy != p     &&
99                    desc.EObjectURI.trimFragment != p.eResource.URI) {
00                error(
01                    "The type " + p.name+ " is already defined",
02                    EvidentlyPackage::eINSTANCE.policy_Name,DUPLICATE_CLASS)
03                return
04            }
05        ]
06    }
```

Figure 22: Validation Rules part c

```
def checkDuplicateLevelsInFiles(Levels l ) {

    val policyName= l.fullyQualifiedName

    l.getVisibleClassesDescriptions.forEach[
        desc |
        if (desc.qualifiedName == policyName &&
                desc.EObjectOrProxy != l     &&
                desc.EObjectURI.trimFragment != l.eResource.URI) {
            error(
                "The type " + l.name+ " is already defined",
                EvidentlyPackage::eINSTANCE.levels_Name,DUPLICATE_CLASS)
            return
        }
    ]
}
@Check
def checkDuplicateAxiomsInFiles(Axioms a ) {

    val axiomsName= a.fullyQualifiedName

    a.getVisibleClassesDescriptions.forEach[
        desc |
        if (desc.qualifiedName == axiomsName &&
                desc.EObjectOrProxy != a     &&
                desc.EObjectURI.trimFragment != a.eResource.URI) {
            error(
                "The type " + a.arr+ " is already defined",
                EvidentlyPackage::eINSTANCE.axioms_Arr,DUPLICATE_CLASS)
            return
        }
    ]
}
```

Figure 23: Validation Rules part d

```
def static containingPolicy(EObject e) {
    e.getContainerOfType(typeof(Policy))
}
def static containingModel(EObject e) {
    e.getContainerOfType(typeof(Model))
}

def static containingGA(EObject e) {
    e.getContainerOfType(typeof(GenericArg))
}
def static containingLevel(EObject e) {
    e.getContainerOfType(typeof(Levels))
}


@Check
def void checkNoDuplicateVariable(UseElem vardecl) {
    val list = vardecl.containingPolicy.elements

    for(PoBElem eachPolicy:list){

     val duplicate=eachPolicy.use.getAllContentsOfType(typeof(UseElem)).findFirst[
            (it != vardecl && it.name == vardecl.name) ||
            (it.name.name == it.alias)||(it.alias == vardecl.name.name)
        ]
        if (duplicate !== null)
        error("Duplicate variable declaration '" + vardecl.name.name + "'",
            EvidentlyPackage::eINSTANCE.useElem_Name,
            DUPLICATE_NAME
        )
    }

    }
```

Figure 24: Validation Rules part e

```
@Check
def void checkForDuplicateFlowpoints( Flowpoints fp){

    val list = fp.containingModel.elements

    for(MBElem elem: list){

 val duplicate=elem.getAllContentsOfType(typeof(Flowpoints)).findFirst[
        (it != fp && it.name == fp.name)
    ]
    if (duplicate !== null)
    error("Duplicate variable declaration '" + fp.name + "'",
        EvidentlyPackage::eINSTANCE.flowpoints_Name,
        DUPLICATE_NAME
    )
    }
}
@Check
def void checkForDuplicateProperty(Property p){

    val list = p.containingModel.elements
    for(MBElem elem: list){

         val duplicate=elem.getAllContentsOfType(typeof(Property)).findFirst[
        (it != p && it.propertyName == p.propertyName)
    ]
    if (duplicate !== null){
    error("Duplicate variable declaration '" + p.propertyName + "'",
        EvidentlyPackage::eINSTANCE.property_PropertyName,DUPLICATE_NAME)
        }
        }
    }
```

Figure 25: Validation Rules part f

LIST OF REFERENCES

[1] S. Chong and A. C. Myers, "Security policies for downgrading," Proceedings of the 11th ACM conference on Computer and communications security - CCS 04, 2004.

[2] Certification of Programs for Secure Information Flow http://www.cs.cornell.edu/andru/cs711/2003fa/reading/denning77.pdf

[3] Zheng, L., & Myers, A. C. (n.d.). Dynamic Security Labels and Noninterference (Extended Abstract). Formal Aspects in Security and Trust IFIP International Federation for Information Processing, 27-40. doi:10.1007/0-387-24098-5_3

[4] Java + information flow. https://www.cs.cornell.edu/jif/

[5] Andrew C. Myers. JFlow: Practical Mostly-static Information Flow Control. In Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '99, pages 228–241, New York, NY, USA, 1999. ACM. doi:10.1145/292540.292561.

[6] Elisavet Kozyri, Owen Arden, Andrew C. Myers, and Fred B. Schneider. JRIF: Reactive Information Flow Control for Java. February 2016.

[7] Niklas Broberg, Bart van Delft, and David Sands. Paragon for practical programming with information-flow control. In APLAS, volume 8301, pages 217–232. Springer, 2013.

[8] Michael D. Ernst, Ren´e Just, Suzanne Millstein, Werner Dietl, Stuart Pernsteiner, Franziska Roesner, Karl Koscher, Paulo Barros Barros, Ravi Bhoraskar, Seungyeop Han, Paul Vines, and Edward X. Wu. Collaborative Verification of Information Flow for a High-Assurance App Store. In Proceedings of the 2014 ACM SIGSAC

Conference on Computer and Communications Security, pages 1092–1104, New York, NY, USA, 2014. ACM. doi:10.1145/2660267.2660343.

[9] Xtext language development tool: https://www.eclipse.org/Xtext/documentation/index.html

[10] G. Hedin. Tutorial: An Introductory Tutorial on JastAdd Attribute Grammars GTTSE III, 166-200, LNCS 6491, 2011. http://dx.doi.org/10.1007/978-3-642-18023-1_4.

[11] JFlex, lexical analyzer generator for Java: http://jflex.de/manual.html

[12] Beaver - a LALR Parser Generator: http://beaver.sourceforge.net/

[13] Building Domain-specific Languages with Xtext and Xtend: https://blogs.itemis.com/en/building-domain-specific-languages-with-xtext-and-xtend

[14] Eclipse Modeling Framework (EMF) – Tutorial: http://www.vogella.com/tutorials/EclipseEMF/article.html

[15] "Using the ASM framework to implement common bytecode transformation patterns", E. Kuleshov, AOSD.07, March 2007, Vancouver, Canada.