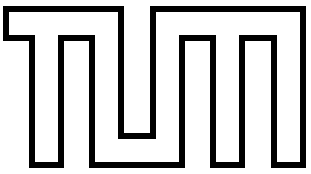


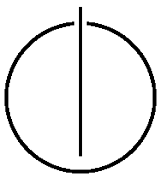
FAKULTÄT FÜR INFORMATIK

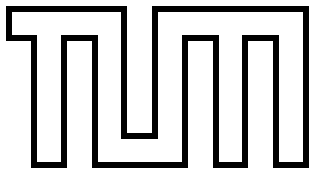
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Master's Thesis

Semi-Automated Detection of Sanitization, Authentication and Declassification Errors in UML State Charts.

Md Adnan Rabbi





FAKULTÄT FÜR INFORMATIK

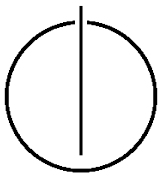
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Master's Thesis

Semi-Automated Detection of Sanitization, Authentication and Declassification Errors in UML State Charts.

Halbautomatische Erkennung von Sanitisierungs,Authenifizierungs und Deklassifierungsfehlern in UML-Zusantsdiagrammen.

Author:	Md Adnan Rabbi
Supervisor:	Prof. Dr. Claudia Eckert
Advisor:	MSc. Paul-loan Muntean
Date:	November 15, 2015



Ich versichere, dass ich diese Diplomarbeit selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel verwendet habe.

München, den 13. September 2015 Md Adnan Rabbi

Acknowledgments

If someone contributed to the thesis... might be good to thank them here.

Abstract

An abstracts abstracts the thesis!

Contents

Outline of the Thesis

Part I: Introduction and Theory

CHAPTER 1: INTRODUCTION

This chapter presents an overview of the thesis and it purpose. Furthermore, it will discuss the sense of life in a very general approach.

CHAPTER 2: BACKGROUND INFORMATION

This chapter describes the background and the essential theory to establish the research.

Part II: Implementation and Analysis

CHAPTER 3: CHALLENGES AND ANNOTATION LANGUAGE EXTENSION

This chapter presents the challenges and annotation language extension for the system.

CHAPTER 4: IMPLEMENTATION

This chapter presents the implementation of the system.

CHAPTER 5: EXPERIMENTS

This chapter presents the different application area of the system.

Part III: Conclusion and Future Work

CHAPTER 6: CONCLUSION AND FUTURE WORK

This chapter presents the conclusion of the whole research along with future work intentions.

Part I.

Introduction and Theory

1. Introduction

Security is one of the major important factor in software development. To develop a secure system is difficult task. Only adding some information flow restrictions is not sufficient. Information flow vulnerabilities detection in code and UML state charts is not well known. It is particularly one of the challenging issue. Actually there is no common annotation language for annotating UML state charts and source code with information flow security constraints such that errors can be detected also when code is not available. Also there are no automated checking tools which can reuse the annotated constraints in early stages of software development phase to check for information flow errors. It is important to specify security constraints as early as possible in the software development phase in order to avoid later costly repairs or exploitable vulnerabilities.

A solution for tagging sanitization, declassification and authentication in source code is based on libraries which contain all needed annotations attached to function declarations. This approach plays an important role mainly for static analysis bug detection techniques where the information available during program run-time. Detection of information flow vulnerabilities uses dynamic analysis techniques , static analysis techniques and hybrid techniques which combine static and dynamic approaches. The static techniques need to know when to use sanitization , declassification and authentication functions. Data sanitization has been studied in the context of architectures for high assurance systems, language-based information flow controls and privacy-preserving data publication [6]. A global policy of noninterference which ensures that high-security data will not be observable on low-security channels. Because noninterference is typically too strong a property, most programs use some form of declassification to selectively leak high security information[8].Declassification is often expressed as an operation within a given program. Authentication is the way through which the users gets access to a system. In this research main focus are these three types of functionalities which are sanitization, declassification and authentication errors in UML state charts.

Web applications are often implemented by developers with limited security skills and that's why they contain vulnerabilities. Most of these vulnerabilities come from the lack of input validation. That is, web applications use malicious input as part of a sensitive operation, without having properly checked or sanitized the input values prior to their use. Another function is declassification. We all know that computing systems often deliberately release (or declassify) sensitive information. A main security concern for systems permitting information release is whether this release is safe or not. Is it possible that the attacker compromises the information release mechanism and extracts more secret information than intended? Now-a-days computing systems release sensitive information by classifying the basic goals according to what information is released, who releases information, where in the system information is released and when information can be released. in case of authentication, it is the mechanism actually which confirms the identity of users trying to access a system(application, login verification into a system, database access etc.).

It is important to develop techniques and tools which can detect information flow type of errors before software developers or programmers develop their production code. Information flow errors in UML models and code are introduced by software developers or programmers who are sometimes unaware or blind while developing software. This type of vulnerabilities are hard to detect because static code analysis techniques need previous knowledge about what should be considered a security issue. Code annotations which are added mainly during software development [6] can be used to provide additional knowledge regarding security issues. On the other hand code annotations can increase the number of source code lines by 10%. In order to detect information flow vulnerabilities software artifacts have to be annotated with annotations attached to public data, private data and to

system trust boundaries. Next, annotated artifacts have to be made tractable by tools which can use the annotations and check if information flow constraints hold or not based on information propagation techniques.

Static Checking is a promising research area which tries to cope with the shortage of not having the program run-time information. During extended static analysis additional information is provided to the static analysis process. This information can be used to define trust boundaries and tag variables. Textual annotations are usually manually added by the user in source code. At the same time annotations can be automatically generated and inserted into source code . Static Checking can be used to eliminate bugs in a late stage of the software project when code development is finished. Tagging and checking for information exposure bugs during the design phase would eliminate the potential of implementing software bugs which can only be removed very costly after wards. Thus security concerns should be enforced into source code right after the conceptual phase of the project.

It can be said that annotations can cover design decisions and enhance the quality of source code. Annotations are necessary in order to do Static Checking and the user needs a kind of assistance tool that helps selecting the suited annotation based on the current context.At the same time adding annotations to reusable code libraries reduces even more the annotation burden since libraries can be reused, shared and changed by software development teams.

2. Background Information

2.1. Sanitization

Sanitization is the process of removing sensitive information from a document or other message or sometimes encrypting messages, so that the document may be distributed to a broader audience. Sometimes sanitization can be called as an operation that ensures that user input can be safely used in an SQL query. Web applications use malicious input as part of a sensitive operation without having properly checked or sanitized the input values from the user. Previous research on vulneribility analysis has mostly focused on identifying cases which web applications directly uses external input for critical operations. It is suggested that always use proper sanitization method to validate external input values from the user for any application.For example, user inputs must always flow through a sanitizing function before flowing into a SQL query or HTML, to avoid SQL injection or cross-site scripting vulnerabilities.

Reflection of security breaches are very significant for high assurance system. For examples of this type of systems are aircraft navigation, where a fault could lead to a crash, various control systems which has critical infrastructure , where an error could cause toxic waste to leak, and weapons targeting, where an inaccuracy could result in severe collateral damage. In such operational environments, the impact is virtually irreversible and must therefore be prevented even if it is likely to occur with low probability.It's always good that transforming information to a form which is suitable for release or sanitize the information by redacting some portions of it.

Some basic purpose of sanitization are given below:

- Remove malicious elements from the input.
- To identify the set of parameters and global variables which must be sanitized before calling functions.
- It is acceptable to first pass the untrusted user input through a trusted sanitization function.
- Any user input data must flow through a sanitization function before it flows into a SQL query.
- Confidential data needs to be cleansed to avoid information leaks.
- Most paths that go from a source to a sink pass through a sanitizer.
- Developers typically define a small number of sanitization functions in libraries.

2.2. Declassification

Information security has a challenge to address: enabling information flow controls with expressive information release (or declassification) policies. In a scenario of systems that operate on data with different sensitivity levels, the goal is to provide security assurance via restricting the information flow within the system. Practical security-typed languages support some form of declassification through which high-security information is allowed to flow to a low-security system or observer.

United States Federal Trade Commission reveals the damage that is continually caused by electronic information leakage. In protecting sensitive information, including everything from credit

card information to military secrets to personal, medical information, there is a highly need for software applications with strong, confidentiality guarantees. Security-typed languages promise to be a valuable tool in making provably secure software applications. In such languages, each data item is labeled with its security policy. In practical security-typed languages support some form of declassification, in which high-security information is permitted to flow to a low-security receiver/observer

To declassify information means lowering the security classification of selected information. Sabelfeld and Sands [16] identify four different dimensions of declassification, what is declassified, who is able to declassify, where the declassification occurs and when the declassification takes place.

Myers and Liskov introduced the decentralized label model [14], describing how labels could be applied to a programming language and then used to check information flow policy compliance in distributed systems. The framework includes a declassify function for downgrading data if the owners policies allow. The model allows principals to define their own downgrading policies.

2.2.1. Dimensions of declassification

Classification of the basic declassification goals according to four axes: what information is released, who releases information, where in the system information is released and when information can be released.

- What : Selective or Partial information flow policies [3, 4, 11, 7] regulate what information may be released. Partial release guarantees that only a part of a secret is released to a public domain. Partial release can be specified in terms of precisely which parts of the secret are released. This is useful, for example, when partial information about a credit card number or a social security number is used for logging.

- Who : In a computing system it is essential to specify who controls information release . Ignoring the issue of control opens up attacks where the attacker hijacks release mechanisms to launder secret information. Myers and Liskov decentralised label model [13] security labels with explicit ownership information. According to this approach, information release of some data is safe if it is performed by the owner who is explicitly recorded in the data security label. This model has been used for enhancing Java with information flow controls [12] and has been implemented in the Jif compiler [15].

- Where : In a system information Where is an important aspect of information release.One can ensure that no other part can release further information. by delegating particular parts of the system to release information.Declassification via encryption is not harmful as long as the program is, in some sense, noninterfering before and after encryption. A combination of “where” and “who” policies in the presence of encryption has been recently investigated by Hicks et al.[9]

- When : The fourth dimension of declassification is “when” information should be released.The work of Giambiagi and Dam [29] focuses on the correct implementation of security protocols. Here the goal is not to prove a noninterference property of the protocol, but to use the components of the protocol description as a specification of what and when information may be released.Chong and Myers security policies [2] address when information is released.By annotating variables this is achieved .

2.3. Authentication

Authentication is the mechanism which confirms the identity of users trying to access a system. For a user to be granted access to a resource, they must first prove that they are who they claim to be.

Generally this is handled by passing a key with each request (often called an access token, User verification using user id and password). The system or server verifies that the access token or user id and password is genuine, that the user does indeed have the required privileges to access the requested resource and only then the request granted.
Also authentication can be defined as it is the process by which the system validates a user's logon information. A user's name and password are compared to an authorized list and if the system detects a match then access is granted to the extent specified in the permission list for that user.

One familiar use of authentication and authorization is access control. A computer system that is supposed to be used only by those authorized must attempt to detect and exclude the unauthorized. Common examples of access control involving authentication include:

- A computer program using a blind credential to authenticate to another program.
- Logging in to a computer.
- Using an Internet banking system.
- Withdrawing cash from an ATM and more

2.4. Detecting Information Flow Errors During Design:

If a step of function call like authentication, sanitization or declassification is missing inside the program then this can lead to software vulnerabilities. In the figure 2.1 left side picture depicted that it has three functions. Among then func2() is named either sanitization/declassification/authentication function. Which means in this scenario there will be no error regarding sanitization/declassification/authentication function. On the other hand the right side picture represents there is a missing function of sanitization/declassification/authentication function. That's why it is the buggy path of UML state charts during design stage of software life cycle.

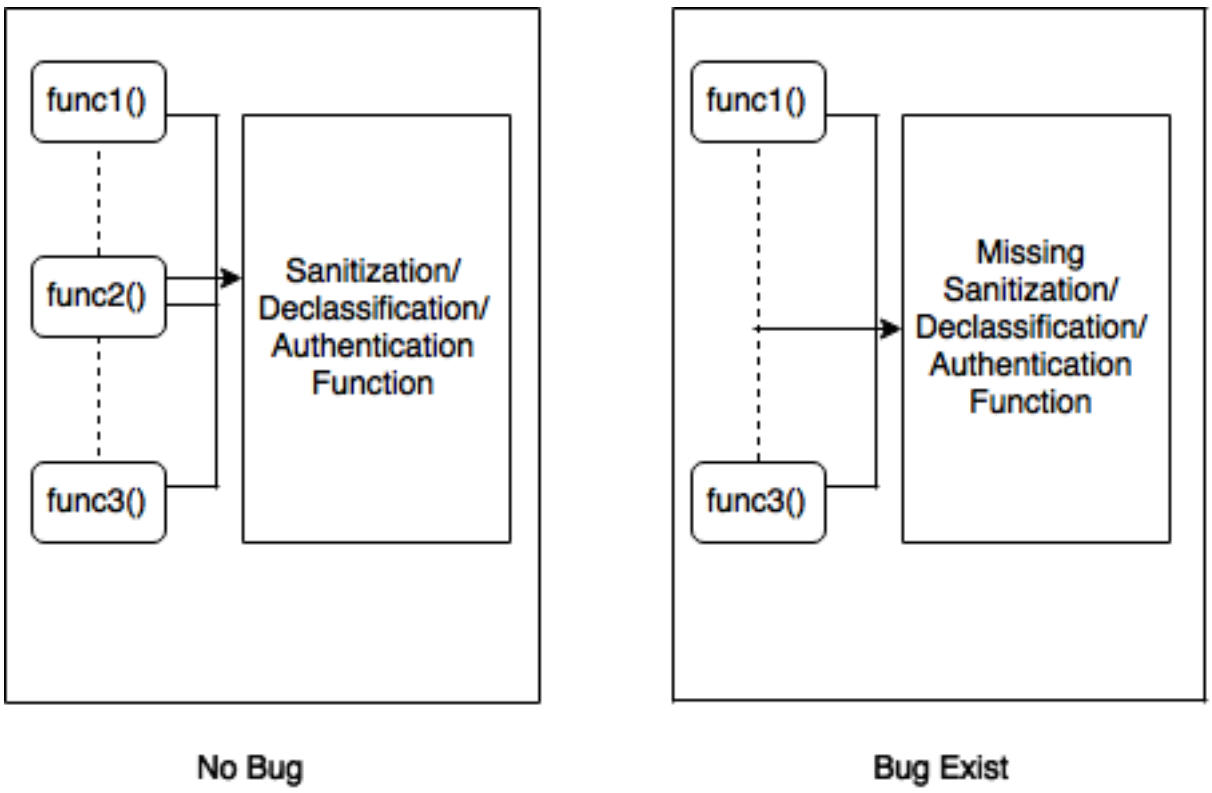


Figure 2.1.: Information flow errors during design

2.5. Detecting Information Flow Errors During Coding:

Figure 2.2 depicts two explicit information flows according to A lattice model of secure information flow of Denning [5] contained in two systems (system 1) and (system 2) where each of the flows starts with statement variable a and ends with leaving the system. The the variable declaration up to outside the system represent C language statements. System 1 is depicted in left side containing the flow from the source to the sink and leaving the system indicated with circles at the top and bottom of each of the two information flows. A source is any function or programming language statement which provides private information through a system boundary. A sink can be a function call or programming language statement which exposes private information to the outside of the system through a system boundary. A system boundary can be a statement, function call, class, package or module. In figure 2.2 the source and sink represent C language statements where information enters and respectively leaves system 1 or system 2. The variable a was tagged with label "H" (confidential) as it inserts confidential information into system 1. The arrows represent the passing of the confidential label "H" between the program statements. When a variable labeled with "H" is about to leave system 1 or system 2 without passing through either authentication/declassifiaction/sanitization function then a bug report should be created.In figure 2 right side system has a bug because it passes a secured/confidential information without passing through authentication/declassifiaction/sanitization function. These functions either authenticated,declassified or sanitized secured/confidential information and makes the variable label as "L" to leave the system. But in the left part of the picture there is no bug as in this system, secured/confidential information and the variable which is labeled with "H" passes through either authentication/declassifiaction/sanitization function.

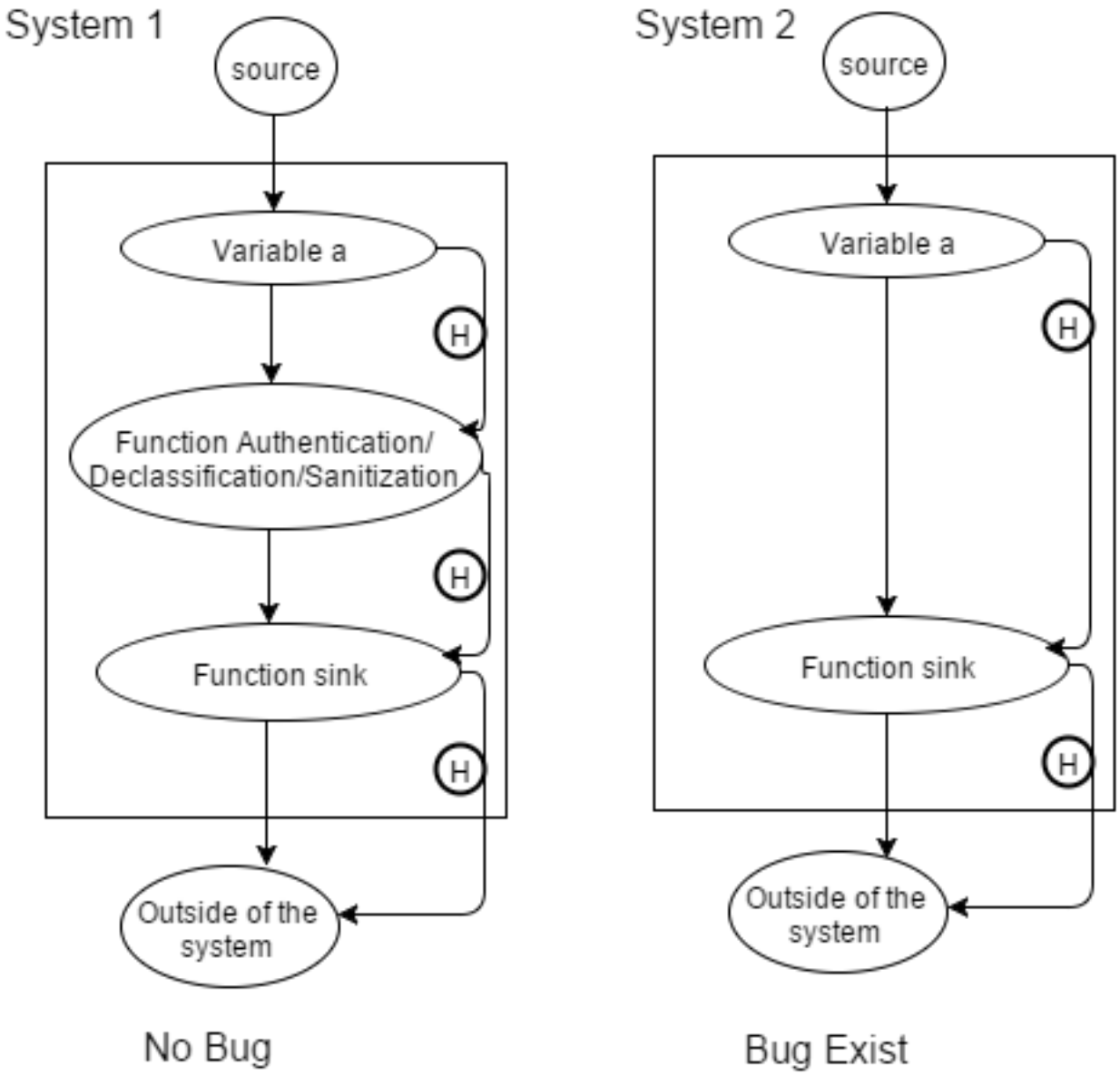


Figure 2.2.: Information flow errors during coding

Part II.

The Second Part

3. Challenges and Annotation Language Extension

3.1. Challenges and Idea

To develop the system eclipse xtext, eclipse xtend and static analysis engine named smtcodan(which is developed in Java to detect C and C++ vulneribilities) are used. For building the source code annotation editor eclipse xtext is used. Modeling the source as UML Statechart opensource platform YAKINDU SCT editor is used. Inside YAKINDU sct editor to genrate the .c(c file) and .h(header file) eclipse xtend is used mainly for genrating code files from statechart. Let's see in briefly what is xtext, xtend and how it works.

- Xtext : Xtext is a framework for development of programming languages and domain specific languages.According to the "eclipse.org/Xtext", it covers all aspects of a complete language infrastructure, from parsers, over linker, compiler or interpreter to fully-blown top-notch Eclipse IDE integration. It comes with great defaults for all these aspects which at the same time can be easily tailored to your individual needs. Here is an example of Xtext file:

```
grammar org.xtext.example.mydsl.MyDsl with
org.eclipse.xtext.common.Terminals

generate myDsl "http://www.xtext.org/example/mydsl/MyDsl"

Model:
messages+=Message*;

Message:
'Hello' name=ID '!';
```

  This language allows to write down a list of messages. The following would be proper input messages which are allowed to write:

```
Hello User!
Hello World!
```

- Xtend : According to the "eclipse.org/Xtend", Xtend is a statically-typed programming language which translates to comprehensible Java source code. Syntactically and semantically Xtend has its roots in the Java programming language but improves on many aspects such as- Extension methods, Lambda Expressions, Active Annotations, Operator overloading, Powerful switch expressions, Multiple dispatch, Template expressions etc. Xtend has zero interoperability issues with Java: Everything you write interacts with Java exactly as expected. At the same time Xtend is much more concise, readable and expressive. Its small library is just a thin layer that provides useful utilities and extensions on top of the Java Development Kit (JDK). Here is an example of Xtend file:

```
package example
import java.util.List
class A {
def greetToAll(List<String> names) {
    for(name: names) {
        println(name.helloMessage)
    }
}
```

```
def helloMessage(String name) {
    'Hello ' + name + '!'
    }
}
```

Xtend provides type inference, the type of name and the return types of the methods can be inferred from the context. Classes and methods are public by default, fields private. Semicolons are optional.

The example also shows the method helloMessage called as an extension method, like a feature of its first argument. Extension methods can also be provided by other classes or instances.

Previous annotation language grammer has been extended more to detect implicit and explicit information flow bugs in UML state charts and C code. The purpose of the same annotation language can be used to add information flow constraints to UML state charts and code in order to detect information flow errors.

The challenge was addressed by extending the annotation language containing textual annotations which can be used to annotate source code and UML state charts which are backward compatible.The single-line annotations have the same as previous consisting start tag "//@" and the multi-line annotations have the start tag "/*@" and the end tag "@*/" .

Some challenges throughout the approach are- converting textual comments into annotations objects, introducing syntactically correct annotations into files, how to use the same annotation language in order to annotate UML state charts and source code, dealing with scattered annotations and attaching annotations to the right function declaration or variable.

The eclipse xtext based grammar is used to parse the whole C/C++ language. The C/C++ source code file extensions (.h, .hh, .hhh, .hxx, .c, .cpp) and UML state chart annotation box (graphical boxes which can be attached to different parts of a UML state chart diagram) can be annotated with policy language restrictions. The obtained CORE model (a one to one mapping from xtext grammar to the ECORE grammar representation) that can be reused for integrating the policy language into an UML state chart editor. Treating the annotation tags as EObjects created new possibilities for annotating UML models.The policy language grammar has about 420 lines of code with code comments included. Source code generation is also supported by using eclipse xtend, ANTLR and .mwe2 files. To parse other programming languages as well this annotation language parser can be used.The result is an extensible policy language and a highly reusable source code implementation as well as source code generator that can easily be used for annotating models and source files.

3.2. Annotation Language Tags

3.3. Language Implementation Process

The process depicted in figure 3.1 was used in order to implement annotation language. Figure 3.1 depicts the annotation language implementation process. The process is comprised of the following steps: At first, the .xtext file containing the language grammar was extended following the requirements. Next the grammar file is compiled and software artifacts are generated. After editing the .mwe2 file then compile it. The result of compiling is: a parser, a lexer and class bindings between these two (lexer and parser) and the grammar ECore model. The generated parser, lexer and the bindings were reused inside static analysis engine and in the UI source file editor. After opening and editing a source file with the editor,the file can be parsed and the annotations can be automatically loaded and used inside checkers.

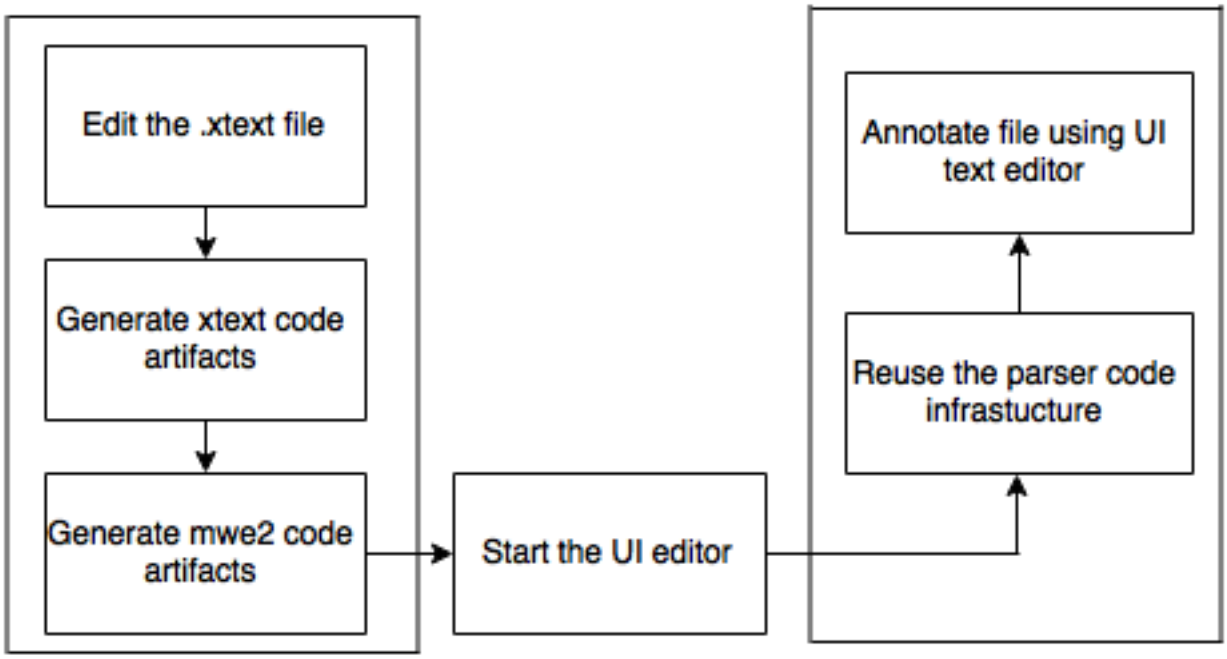


Figure 3.1.: Annotation language design process

4. Implementation

4.1. Overview of System Architecture:

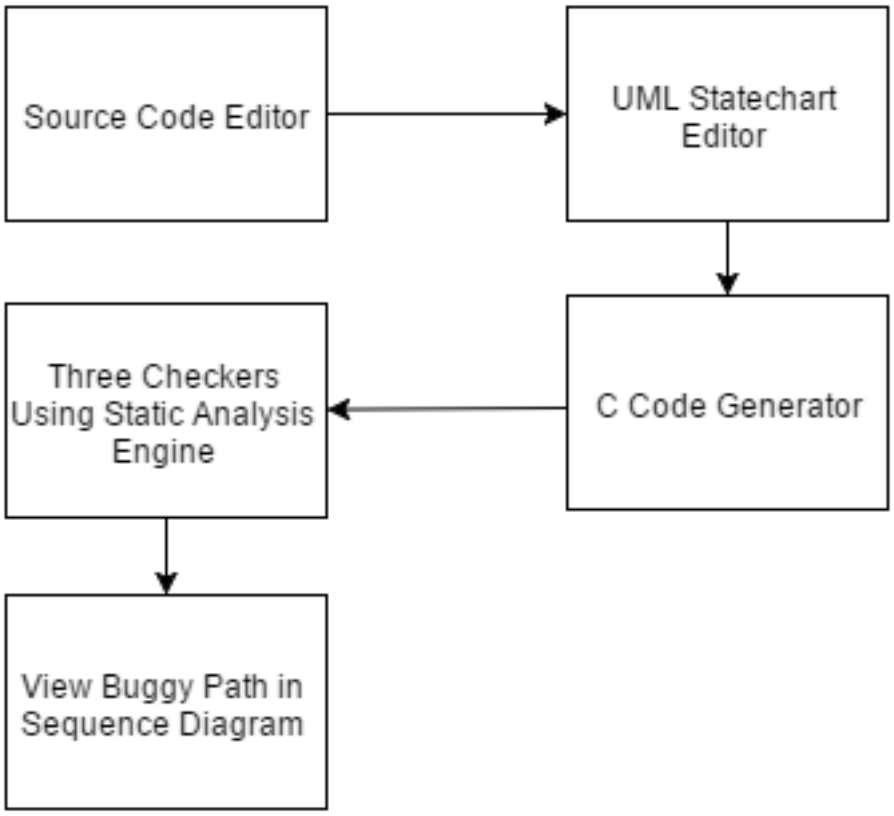


Figure 4.1.: System Architecture

Figure 4.1 depicts the complete system architecture. First, the source code editor is developed using Eclipse Xtext. By this editor one can easily annotate the source code of C/C++. Even it is possible to annotate C/C++ header files. For information flow vulnerabilities detection in C/C++ code annotation technique has chosen which is easy to extend and backward compatible. Then for modeling purpose Yakindu SCT editor has chosen to model the C/C++ code into state charts to detect the bug during design stage of software development life-cycle. Inside the Yakindu SCT editor the annotation language grammar has also included using Eclipse Xtext. So, that user can easily annotate the state charts to detect the information flow vulnerabilities. Afterwards the C code generator has extended inside the Yakindu SCT editor using Eclipse Xtend. After modeling the C code files in Yakindu SCT editor user can easily generate the code using C code generator. Through this generator two files will be generated. One file has .c extension and another file has .h extension. Inside those files annotation has also included. Those annotations are helpful to detect the information flow errors. After generating the code files using static analysis engine named smtcodan three checkers have included to detect authentication, declassification and sanitization function missing vulnerabilities. Then to view the buggy path in sequence diagram a sequence diagram generator has created. That is the end of complete system architecture of this system.

4.2. The Grammar of Annotation Language:

The grammar of annotation language is represented as in Extended Backus Naur Form (EBNF) in figure 4.1 . The following type face conventions were used: Italic font for non-terminals, bold typewriter font for literal terminals including keywords.

Ann_Lang ::= HeaderModel*;

H_Model ::= *S_L_Anno*; ;single line comment rule
| *M_L_Anno*; ;multi line comment rule
| *Func_Decl*; ;function declaration rule
| *Attr_Def*; ;variable declaration rule

S_L_Anno ::= **"//@ @function "**, Func_Type, [**H** | **L**];
| **"//@ @parameter "**, p_Name, Sec_Type, Var_Type, [**H** | **L**];
| **"//@ @variable "**, v_Name, Sec_Type, [**H** | **L**];
| **"//@ @preStep "**, pr_s_Name, [**H** | **L**];
| **"//@ @postStep "**, po_s_Name, [**H** | **L**];

M_L_Anno ::= [**"/*@ "**], ["* "], Func_Ann, (**" @*/"**)
| ("*"), [" "]*, (**"@*/"**);

Func_Ann ::= **"@function "**, Func_Type, [**H** | **L**];
| **"@parameter "**, p_Name, Sec_Type, Var_Type, [**H** | **L**];
| **"@preStep "**, pr_s_Name, [**H** | **L**];
| **"@postStep "**, po_s_Name, [**H** | **L**];

Func_Type ::= **authentication**;
| **declassification**;
| **sanitization**;
| **sink**;
| **source**;
| **trust_boundary**;

Sec_Type ::= **confidential**;
| **source**;

Var_Type ::= **authenticated**;
| **declassified**
| **sanitized**;

Figure 4.2.: Light-weight annotation language grammar excerpt

Annotation language grammar has two grammar rules `S_L_Anno` and `M_L_Anno` used for defining security annotations. The `Func_Decl` and `Attr_Definition` rules are used to recognize C or C++ function declarations and variable. The `Var_Type` rule is used for variable type which is either for authenticated or declassified or sanitized variable. `Sec_Type` rule is used for type of security whether a variable is confidential or not. In `Func_Decl` rule the type function is declared. A function can be either one of this like authentication, declassification, sanitization, source or sink.

4.3. UML State Chart Editor:

A set of formal representation of UML statecharts is presented in this section. The state identifier and event are represented as S and Event respectively both as set types. For simple specification, the basic set types are used. In the definition of a transition from one state to another the guard is defined as a Boolean type. According to F Alhumaidan state based static and dynamic formal analysis of UML state diagrams [1] , a state can have three possible values that are active, passive or null represented as Active, Passive and null respectively. The type of state can be simple, concurrent, non-concurrent, initial or final.

[S, Event]
Boolean ::= *True* | False;
Status ::= *Active* | Passive | Null;
Type ::= *Simple* | Concurent | Nonconcurent | Initial | Final;

Figure 4.3.: UML Statechart Formal Representation

In modeling using sets, it's not imposing any restriction upon the number of elements and a high level of abstraction is supposed. Further, it's not insist upon any effective procedure for deciding whether an arbitrary element is a member of the given collection or not. As a consequent, sets S and Event are sets over which cannot define any operation of set theory. For example, cardinality to know the number of elements in a set cannot be defined. Similarly, the subset, union, intersection or complement operations over the sets are not defined.

The state diagram is a collection of states related by certain types of relations. In the definition of a state, state identifier, its type, status and set of regions is required.Region is defined as a power set of sequence of states. The state is represented by a schema which consists of four components described above. All these components are encapsulated and put in the Schema State given below. The invariants over the schema are defined in the second part of schema.

State
name : *S*
type : *Type*
status : *Status*
regions : *seq Regions*
regions = *1* type= *Simple*
regions = *1* type= *Nonconcurrent*
regions = *1* type= *Concurent*

Figure 4.4.: UML statechart some other formal representation

Invariants:

- If there is no region in a state inside the state diagram, then it is a simple state.
- If there is exactly one region in a state then it is termed as non-concurrent composite state.
- If there are two or more regions in a state then it is concurrent composite state.

The collection of states is represented by the schema States which consists of four variables. The mapping substates from State to power set of State describes type of a state.
Invariants:

- The start state is not in the collection of states.
- The start state is not the target state.

States
start : *State*
states : *State*
states : *State*
substates : *State* State;
target : *State*
start : *states*
start : *target*
start : *dom* substates
states
s : *State* s *dom* substates s *states*
s : *State* s *states* s *start* s *target* s.typ
Simple s *dom* substates
target *states*
target *dom* substates

Figure 4.5.: UML Statechart Formal Representation some parts

- The start state does not belong to domain of substates mapping that is it has no sub-state.
- The set of states is non-empty.
- For any state, s, if it is in the states and is not the start or target state and not the simple state then it belongs to domain of sub-states.
- The target state does not belong to states.
- The target state of the state diagram does not belong to domain of the sub-states.

UML state chart editor has been extended based on the open source Yakindu SCT [10]framework. The existing language grammar with annotation language grammar has extended in order to support new set of tags. Furthermore, an annotation proposal filter implemented which was used to filter out the annotation language tags of the Yakindu SCT language grammar.

To extend the Yakindu SCT editor here it has been decided to represent the statements like variable declaration, function calling as state and transitions are represent as move from one statement to another. A rectangular box can be attached with transitions where annotation can be written as per requirements. So for the developed system the UML statechart formal representation is as like this:

States
start : *State*
states : *State*
states : *State*
substates : *State* State;
target : *State*
start : *states*
start : *target*
transition : *annotation**
target : *states*

Figure 4.6.: UML Statechart Formal Representation for this System

4.4. Source Code Editor:

The source code editor has extended which offers annotation language proposals which are context sensitive with respect to the position of the currently edited syntax line.Editor suggestions work only if the whole file is parsed without errors. Editor was developed using Eclipse Xtext.

As per requirements previous annotation language grammar which is written in xtext language has been extended. Extra annotation have included like “authenticated”, “declassified”, “sanitized”, “sanitization”, “declassification”, “authentication”. Some part of the code snippet of extended xtext grammar is given below.

```
/**
* @FunctionAnnotation :used for function annotations
*/
FunctionAnnotation returns FunctionAnnotation:
{FunctionAnnotation}(
result += '@function' functionType=FunctionType (' ')? (level =('H'|'L'))?
    ((name0=ID))? ((nameComment=ID))? ('\n' | '\r')?
// supported without space before confidential and sensitive
| '@parameter' parameter=ID (name0=ID)? (securityType=SecurityType)?(' ')? (level =('H'|'L'))? ('True'|'False')?
    (variableTyp=VariableType)? ((name1=ID))? ((nameComment=ID))? ('\n' | '\r')?
;

/**
* @SingleLineAnnotation :used for adding single line annotations
*/
SingleLineAnnotation returns SingleLineAnnotation:
{SingleLineAnnotation}(
result+= '//@ @function' functionType=FunctionType (' ')? (level =('H'|'L'))? ((name0=ID))?
    ((nameComment=ID))? ('\n' | '\r')*
// supported without space before confidential and sensitive
| '//@ @parameter' parameter=ID (securityType=SecurityType)? (' ')? (level =('H'|'L'))? ('True'|'False')?
    (variableTyp=VariableType)? ((nameComment=ID))? ('\n' | '\r')?
| '//@ @variable' variable=ID (securityType=SecurityType)? (' ')? (level =('H'|'L'))? ('True'|'False')?
    ((nameComment=ID))? ('\n' | '\r')?

;

/**
* @FunctionType :annotaions types for functions
*/
enum FunctionType: declassification
| sanitization
| authentication
| sink
| source
| trust_boundary
;


/**
* @Variable Type :annotaions types for function parameters
*/
enum VariableType: declassified
| sanitized
| authenticated
;
```

4.5. C Code Generator:

C code generator has extended based on Eclipse EMF and xTend which is used to generate the state chart execution code containing the previously added security annotations from UML state charts. The code generator outputs two files per UML state chart (one .c and one .h file). Generated annota-

tions can reside in both header file and source code file. Previously annotated UML state chart states are converted to either C function calls or C variables declarations, both have been previously annotated. We use the available state chart execution flow functionality which is responsible for traversing the UML state chart during state chart simulation. The UML state chart will be traversed by the code generation algorithm and code is generated based on the mentioned state chart execution flow. The generated code will contain at least one bad path (contains a true positive) and a good path (contains no bug) per UML state chart if those paths were previously modeled inside the UML state chart.

Algorithm 4.1 *C Code Generator*

Input: Statechart
Output: .c and .h files

1: ***function*** GENERATETYPESH*(sc)* ▷ *Where sc - statchart*
2: $defgenerateFile_1(testModule.h, typesHAnnotationContent(sc))$ *Where def - function declaration in xtend*
3: $defgenerateFile_2(testModule.c, typesCAnnotationContent(sc))$
4: ***end function***
5: ***function*** TYPESHANNOTATIONCONTENT*(sc)*
6: ***for*** $s : getFileContent(sc).entrySet$ ***do***
7: ***if*** $!s.key.contains('//@@variable')$ ***then***
8: $s.key$
9: ***else if*** $s.value.contains('(')$ ***then***
10: $void < s.value >;$
11: ***end if***
12: ***end for***
13: ***end function***
14: ***function*** TYPESCANNOTATIONCONTENT*(sc)*
15: ***for*** $s : getFunctionContent(sc).entrySet$ ***do***
16: ***if*** $(!s.value.contains('authentication')and(!s.value.contains('declassification'))$
17: $and(!s.value.contains('sanitization')))$ ***then***
18: $void < s.value >$
19: ***end if***
20: ***end for***
21: ***for*** $region : sc.regions$ ***do***
22: ***if*** $region.name.equalsIgnoreCase('bad_path()'$ ***then***
23: $void < region.name >$
24: ***for*** $s : getBadPathContent(sc).entrySet$ ***do***
25: ***if*** $s.key.contains('//@@variable')$ ***then***
26: $s.key$
27: $s.value$
28: ***end if***
29: ***if*** $s.value.contains('(')$ ***then***
30: $s.value;$
31: $s.value$
32: ***end if***
33: ***end for***
34: ***end if***
35: ***if*** $region.name.equalsIgnoreCase('good_path()')$ ***then***
36: $void < region.name >$
37: ***for*** $s : getGoodPathContent(sc).entrySet$ ***do***
38: ***if*** $s.key.contains('//@@variable')$ ***then***
39: $s.key$
40: ***end if***
41: $s.value;$

```
42:             end for
43:         end if
44:     end for
45: end function
```

```
def HashMap<String, String> getFileContent(Statechart sc) {
    var fileContent = <String, String>newHashMap
    for( region : sc.regions){
        for(vertex : region.vertices)  {
            if  (!( vertex.name.nullOrEmpty)){
                for( transition  : vertex.incomingTransitions) {
                    fileContent.put(transition.specification,vertex.name)
                }

            }

        }

    }
return fileContent
}

def HashMap<String, String> getFunctionContent(Statechart sc) {
    var functionContent = <String, String>newHashMap
    for( region : sc.regions){
        for(vertex : region.vertices. filter [eClass.name.contentEquals('State')])  {

            if  ( (vertex.name.contains('(')) && (!(vertex.name.nullOrEmpty))){
                functionContent.put(vertex.name,vertex.name)
            }
        }
    }
    return functionContent
}

def HashMap<String, String> getBadPathContent(Statechart sc) {
    var badfunctionContent = <String, String>newHashMap
    var String newName

        for( region : sc.regions){

            if(region.name.equalsIgnoreCase('bad_path()')){
                for(vertex : region.vertices. filter [eClass.name.contentEquals('State')]){

                    if (!( vertex.name.contains('(')) && (!(vertex.name.nullOrEmpty))){
                        for( transition  : vertex.incomingTransitions) {
                        badfunctionContent.put(transition.specification,vertex.name)
                        }
                    }
                    if ((vertex.name.contains('(')) && (!(vertex.name.nullOrEmpty))){
                        if  ( (vertex.name.contains('char ')) ){
                            newName=vertex.name.replaceAll('char *','')
                            if(newName.contains('*'))
                                newName=newName.replaceAll('\\*','' )
                            badfunctionContent.put(newName,newName)
                            }
                        else
                            badfunctionContent.put(vertex.name,vertex.name)
                    }
                }

            }
```

```
        }
    return badfunctionContent
}

def String getVariableName(Statechart sc){
    var String variablename
        for( region : sc.regions){
            for(vertex : region.vertices. filter [eClass.name.contentEquals('State')])  {

                if  (!( vertex.name.contains('(')) && (!(vertex.name.nullOrEmpty))){
                    for( transition : vertex.incomingTransitions) {
                        variablename= vertex.name.replaceAll('char *','')
                        if (variablename.contains('*'))
                        variablename=variablename.replaceAll('\\*','')
                    }
                }
            }

        }
    return variablename
}

def HashMap<String, String> getGoodPathContent(Statechart sc) {
    var goodfunctionContent = <String, String>newHashMap
    var String newName

        for( region : sc.regions){
            if (region.name.equalsIgnoreCase('good_path()')){

                val choiceState=0;
                val increment=1;

                for(vertex : region.vertices. filter [eClass.name.contentEquals('Choice')]){
                    val sum=choiceState+increment;
                    for( transition : vertex.incomingTransitions) {
                        System.out.println(" ********* "+"if\n"+sum);

                    }
                }

                for(vertex : region.vertices. filter [eClass.name.contentEquals('State')])  {


                    for(invertex : vertex.parentRegion.vertices. filter [eClass.name.contentEquals('State')])
                        {
                            if (!( vertex.name.contains('(')) && (!(vertex.name.nullOrEmpty))){
                                for( transition : vertex.incomingTransitions) {
                                    goodfunctionContent.put(transition.specification,vertex.name)

                                }
                            }
                            if ((vertex.name.contains('(')) && (!(vertex.name.nullOrEmpty))){

                                if  ( (vertex.name.contains('char '))){

                                    newName=vertex.name.replaceAll('char *','')
                                    newName=newName.replaceAll('\\*','')
                                    goodfunctionContent.put(newName,newName)
                                    }
                                    else
                                    goodfunctionContent.put(vertex.name,vertex.name)
                                    }

                    }
```

```
                }
            }
        }
    return goodfunctionContent
}
```

4.6. Three Checkers in Static Analysis Engine

Static analysis refers analyzing code without executing it. Generally it is used to find bugs or ensure conformance to coding guidelines. The classic example is a compiler which finds lexical, syntactic and even some semantic mistakes.Static analysis tools should be used when they help maintain code quality. If they're used, they should be integrated into the build process, otherwise they will be ignored.Some characteristics of static analysis tools are:

- Identify anomalies or defects in the code.
- Analyze structures and dependencies.
- Help in code understanding.
- To enforce coding standards.

For this system static analysis engine "smtcodan" has been used. Inside the engine to detect the information flow vulnerabilities required classes like AuthenticationFunctionChecker.java, DeclassificationFunctionChacker.java, SanitizationFunctionChecker.Java, Authentication_gen.java, Declassification_gen.java, Sanitization_gen.java files are included. These files are included in order to detect authentication, declassification and sanitization function missing bug detection in C code. For these three types of function detection here it has been used as library functions in C programming language. In order to detect the information flow vulnerabilities three models have been included such as Authentication_gen.java, Declassification_gen.java, Sanitization_gen.java. In the generated .c file there exist these three kinds of methods without signature which is given in figure . As they have no method body that's why they are acting as library function in "smtcodan" static analysis engine. Inside the engine three function signature will act as keyword like authentication, declassification and sanitization function.

From C code generator generated .c and .h file with annotation should act as input for "smtcodan" static analysis engine. Engine parses the code with annotation. The authentication, declassification and sanitization function all makes the high secured variable or confidential variable as low and according to the policy they passes the information from the sender to the receiver in a secured way. While implementing the checkers, information flow restriction has followed. If any of the C files are not following the secure information flow then bug should be triggered as either authentication , declassification or sanitization function missing function.

4.7. View Buggy Path in Sequence Diagram

Through the static analysis engine buggy path can be found as a list of string. Inside the list there are function calls, separate statements like if statements, switch-case statements, variable declaration, assignment of variables etc. of programming language (like C,C++). Then to view the path using java a sequence diagram is generated. now it easier to trace the buggy path by viewing generated sequence diagram. One sample example of the buggy path is given in figure 4.1.

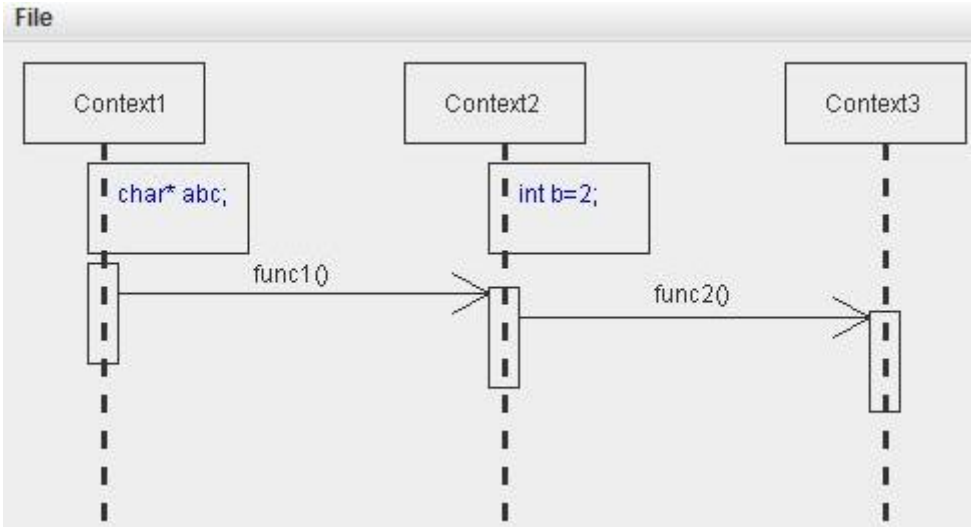


Figure 4.7.: Error trace path in sequence diagram

5. Experiments

6. Related Work

There are many annotation languages proposed until now for extending the C type system [9], [13], [29], [30], [57] to be used during run-time as a new language run-time for PHP and Python [61] to annotate function interfaces [13], [29], [57], to annotate models in order to detect information flow bugs [24] to annotate source code files [46], [47], [56] or to annotate control flows [13], [15], [29].The following annotation languages have made significant impact: Microsoftś SAL annotations [29] helped to detect more than 1000 potential security vulnerabilities in Windows code [3]. In addition, several other annotation languages including FlowCaml [50], Jif [7], Fable [55], AURA [22] and FINE [54] express information flow related concerns.

UMLSec [23] is a model-driven approach that allows the development of secure applications with UML. Compared with our approach, UMLSec does neither automatic code generation nor the annotations can be used for automated constraints checking.

The detection of information flow errors [31] can be addressed with dynamic analysis techniques [2], [16], [48], static analysis techniques [17], [41], [51], [58], [60] (similar to our approach with respect to static analysis of code and tracking of data information flow) and hybrid techniques which combine static and dynamic approaches [38]. Also, extended static checking [10] (ESC) is a promising research area which tries to cope with the shortage of not having certain program run-time information. The static code analysis techniques need to know which parts of the code are: sinks, sources and which variables should be tagged. A solution for tagging these elements in source code is based on a pre-annotated library which contains all the needed annotations attached to function declarations. Leino [27] reports about the annotation burden as being very time consuming and disliked by some programming teams.

The studies rely on manually written annotations while our annotation language is integrated into two editors which are be used to annotate UML state charts and C code by selecting annotations from a list and without the need to memorize a new annotation language.

Recently taint modes integrated in programming languages as Camlbased FlowCaml [52], Adabased SPARK Examiner [5] and the scripting. However, none of these annotation and programming languages have support for introducing information flow restrictions in both models and the source code. Splint [14], Flawfinder [59] and Cqual [49] are used to detect information flow bugs in source code and come with comprehensive user manuals describing how the annotation language can be used in order to annotate source code. iFlow [24] is used for detecting information flow bugs in models and is based on modeling dynamic behavior of the application using UML sequence diagrams and translating them into code by analyzing it with JOANA [25]. In comparison with our approach these tools do not use the same annotation language for annotating UML models and code. Thus, a user has to learn to use two annotation languages which can be perceived to be a high burden in some scenarios.

Heldal et al. [18], [19] introduced an UML profile that incorporates a decentralized label model [40] into the UML. It allows the annotation of UML artifacts with Jif [42] labels in order to generate Jif code from the UML model automatically. However, the Jif-style annotation already proved to be non-trivial on the code level [45], while [19] notes that the actual automatic Jif code generation is still future work. These approaches can not be used to annotate both UML models and code. Moreover, these approaches lack of tools for automated checking of previously imposed constraints.

7. Conclusion and Future Work

A keyword-based annotation language that can be used out of the box for annotating UML state charts and C code in two software development phases by providing two editors for inserting security annotations in order to detect information flow bugs automatically. It's evaluated on some sample programs and showed that this approach is applicable to real life scenarios.
It's a light-weight annotation language usable for specifying information flow security constraints which can be used in the design and coding phase in order to detect information flow bugs.
In future it can be extended for source code editor as a pop-up window based proposal editor used to add/retrieve annotation to/from a library. The definition of new language annotation tags should be possible from the same window by providing two running modes (language extension mode and annotation mode). The envisaged result is to reduce the gap between annotations insertion/retrieval and the definition of new language tags. This would help to create personalized annotated libraries which can be collaboratively annotated if needed.

Appendix

A. Detailed Descriptions

Here come the details that are not supposed to be in the regular text.

Bibliography

[1] Fahad Alhumaidan et al. State based static and dynamic formal analysis of uml state diagrams. *Journal of Software Engineering and Applications*, 5(07):483, 2012.

[2] Stephen Chong and Andrew C Myers. Security policies for downgrading. In *Proceedings of the 11th ACM conference on Computer and communications security*, pages 198–209. ACM, 2004.

[3] Ellis Cohen. Information transmission in computational systems. In *ACM SIGOPS Operating Systems Review*, volume 11, pages 133–139. ACM, 1977.

[4] Ellis S Cohen. Information transmission in sequential programs. pages 297–335, 1978.

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

[6] Ashish Gehani, David Hanz, John Rushby, Grit Denker, and Rance DeLong. On the (f) utility of untrusted data sanitization. In *MILITARY COMMUNICATIONS CONFERENCE, 2011-MILCOM 2011*, pages 1261–1266. IEEE, 2011.

[7] Roberto Giacobazzi and Isabella Mastroeni. Adjoining declassification and attack models by abstract interpretation. In *Programming Languages and Systems*, pages 295–310. Springer, 2005.

[8] Boniface Hicks, Dave King, Patrick McDaniel, and Michael Hicks. Trusted declassification:: high-level policy for a security-typed language. In *Proceedings of the 2006 workshop on Programming languages and analysis for security*, pages 65–74. ACM, 2006.

[9] Boniface Hicks, David King, and Patrick McDaniel. Declassification with cryptographic functions in a security-typed language. *Network and Security Center, Department of Computer Science, Pennsylvania State University, Tech. Rep. NAS-TR-0004-2005*, 2005.

[10] itemis AG. Yakindu SCT Open-Source-Tool , https://code.google.com/a/eclipselabs.org/p/yakindu/.

[11] Rajeev Joshi and K Rustan M Leino. A semantic approach to secure information flow. volume 37, pages 113–138. Elsevier, 2000.

[12] Andrew C Myers. Jflow: Practical mostly-static information flow control. In *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 228–241. ACM, 1999.

[13] Andrew C Myers and Barbara Liskov. *A decentralized model for information flow control*, volume 31. ACM, 1997.

[14] Andrew C Myers and Barbara Liskov. Protecting privacy using the decentralized label model. volume 9, pages 410–442. ACM, 2000.

[15] Andrew C Myers, Lantian Zheng, Steve Zdancewic, Stephen Chong, and Nathaniel Nystrom. Jif: Java information flow. *Software release. Located at http://www. cs. cornell. edu/jif*, 2005, 2001.

[16] Andrei Sabelfeld and David Sands. Declassification: Dimensions and principles. volume 17, pages 517–548. Citeseer, 2009.