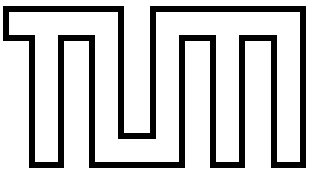


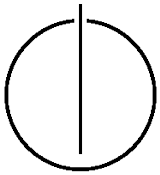
FAKULTÄT FÜR INFORMATIK

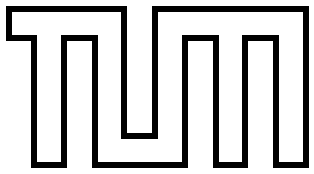
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Master's Thesis

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

Md Adnan Rabbi





FAKULTÄT FÜR INFORMATIK

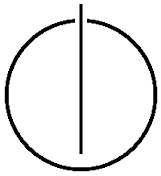
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

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 Muntean
Date:	October 15, 2015



I assure the single handed composition of this masters thesis only supported by declared resources.
Ich versichere, dass ich diese Masterarbeit selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel verwendet habe.

München, den 15 October 2015 Md Adnan Rabbi

Acknowledgments

I would like to express my deepest appreciation to all those who provided me the possibility to complete this thesis. A special gratitude and thanks I give to my supervisor Prof. Dr. Claudia Eckert and my advisor, MSc. Paul Muntean, whose guidance, stimulating suggestions and encouragement, helped me in all the time of research and writing of this thesis. Without their cooperation in the last 6 months, this thesis could not be done smoothly.
Last but not the least, I wish to thank my family for their support and encouragement throughout my Master study.

Abstract

Information flow vulnerabilities detection with static code analysis techniques is challenging because code is usually not available during the software design phase and previous knowledge about what should be annotated and tracked is needed. To detect information flow errors in UML state charts and C code are not easy task as they can cause data leakages or unexpected program behavior. In this research it is proposed that textual annotations used to introduce information flow constraints in UML state charts and code which are afterwards automatically loaded by information flow checkers that check if imposed constraints hold or not. The experimental results on selected sample scenarios shows that this approach is effective and can be further applied to other types of UML models and programming languages as well, in order to detect different types of vulnerabilities.

The contributions of this thesis is the development of a system for semi-automated detection of sanitization, authentication and declassification errors in UML state Charts. A novel light-weight security annotation language used in order to define information flow constraints regarding authentication, declassification and santization function errors in UML state charts and source code. Annotation language editors designed as Eclipse plug-ins which is used to edit UML state charts and source code files. Developed Source code generator as Eclipse plug-in which is used to generate C code with header files from UML State chart. And finally experimented automatic loading and usage of textual annotations inside 3 new checkers.

Contents

List of Figures

List of Tables

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 important factor in software development. To develop a secure system is not an easy 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 now-a-days. 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 [32]. 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 [41]. Declassification is often expressed as an operation within a given program. Authentication is the way through which the users get 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 [14] 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.

Current standard security practices do not provide substantial assurance that the end-to-end behavior of a computing system satisfies important security policies such as confidentiality.An end-to-end confidentiality policy might assert that secret input data cannot be inferred by an attacker through the attacker's observations of system output; this policy regulates information flow. Conventional security mechanisms such as access control and encryption do not directly address the enforcement of information-flow policies. Recently, a promising new approach has been developed: the use of programming-language techniques for specifying and enforcing information-flow policies. The past three decades of research on information-flow security, particularly focusing on work that uses static program analysis to enforce information-flow policies.

Protecting the confidentiality of information manipulated by computing systems is a long-standing yet increasingly important problem. There is little assurance that current computing systems protect data confidentiality and integrity; existing theoretical frameworks for expressing these security properties are inadequate and practical techniques for enforcing these properties are unsatisfactory. Language-based mechanisms are especially interesting because the standard security mechanisms are unsatisfactory for protecting confidential information in the emerging, large networked information systems. Military, medical and financial information systems, as well as web-based services such as mail, shopping, and business-to-business transactions are applications that create serious privacy questions for which there are no good answers at present.

The standard way to protect confidential data is access control: some privilege is required in order to access files or objects containing the confidential data. Access control checks place restrictions on the release of information but not its propagation. Once information is released from its container, the accessing program may, through error or malice, improperly transmit the information in some form. It is unrealistic to assume that all the programs in a large computing system are trustworthy; security mechanisms such as signature verification and antivirus scanning do not provide assurance that confidentiality is maintained by the checked program. To ensure that information is used only in accordance with the relevant confidentiality policies, it is necessary to analyze how information flows within the using program; because of the complexity of modern computing systems, a manual analysis is infeasible.

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.

In summary the contribution for this research are:

- A novel light-weight security annotation language used to define information flow constraints regarding authentication, declassification and santization function errors in UML state charts and source code.

- Annotation language editors designed as Eclipse plug-ins which is used to edit UML state charts and source code files.

- Source code generator developed as Eclipse plug-in which is used to generate C code with header files from UML State chart.

- Experiments are presented in experiments section based on automatic loading and usage of textual annotations inside 3 new checkers.

2. Literature Review

Although the difficulty of strongly protecting confidential information has been known for some time, the research addressing this problem has had relatively little impact on the design of commercially available computing systems. These systems employ security mechanisms such as access control, capabilities, firewalls and antivirus software; it is useful to see how the these standard security mechanisms fall short. Access control is an important part of the current security infrastructure. For example, a file may be assigned access-control permissions that prevent users other than its owner from reading the file; more precisely, these permissions prevent processes not authorized by the file owner from reading the file. However, access control does not control how the data is used after it is read from the file. To soundly enforce confidentiality using this access-control policy, it is necessary to grant the file access privilege only to processes that will not improperly transmit or leak the confidential data. But these are precisely the processes that obey a much stronger information-flow policy. Access-control mechanisms cannot identify these processes; therefore, access control, while useful, cannot substitute for information-flow control.

Other common security enforcement mechanisms such as firewalls, encryption and antivirus software are useful for protecting confidential information. However, these mechanisms do not provide end-to-end security. For example, a firewall protects confidential information by preventing communication with the outside. In practical systems, however, firewalls permit some communication in both directions [11]; whether this communication violates confidentiality lies outside the scope of the firewall mechanism. Similarly, encryption can be used to secure an information channel so that only the communicating endpoints have access to the information. However, this encryption provides no assurance that once the data is decrypted, the computation at the receiver respects the confidentiality of the transmitted data. Antivirus software is based on detecting patterns of previously known malicious behavior in code and, thus, offers limited protection against new attacks.

However, many intuitively secure programs do allow some release, or declassification, of secret information (such as- password checking, information purchase, and spreadsheet computation). Noninterference fails to recognize such programs as secure. In this respect, many security type systems enforcing noninterference are impractical. On the other side of the spectrum are type systems designed to accommodate some information leakage. However, there is often little or no guarantee about what is actually being leaked. As a consequence, such type systems are vulnerable to laundering attacks, which exploit declassification mechanisms to reveal more secret data than intended. To bridge this gap, Sabelfeld and Myers [73] introduces a new security property, delimited release, an end-to-end guarantee that declassification cannot be exploited to construct laundering attacks. In addition, a security type system is given that straightforwardly and provably enforces delimited release.

If a user wishes to keep some data confidential, he or she might state a policy stipulating that no data visible to other users is affected by confidential data. This policy allows programs to manipulate and modify private data, so long as visible outputs of those programs do not improperly reveal information about the data. A policy of this sort is a noninterference policy [35], because it states that confidential data may not interfere with (affect) public data. An attacker (or unauthorized user) is assumed to be allowed to view information that is not confidential (that is public). The usual method for showing that noninterference holds is to demonstrate that the attacker cannot observe any difference between two executions that differ only in their confidential input [36]. Noninterference

can be naturally expressed by semantic models of program execution. This idea goes back to Cohen's early work on strong dependency [18], [20]. McLean [58] argues for noninterference for programs in the context of trace semantics. However, neither work suggests an automatic security enforcement mechanism.

The type-checking approach has been implemented in the Jif compiler [16, 62]. In the type-checking approach, every program expression has a security type with two parts: an ordinary type such as int, and a label that describes how the value may be used. Unlike the labels used by mandatory access-control mechanisms, these labels are completely static: they are not computed at run time. Because of the way that type checking is carried out, a label defines an information-flow policy on the use of the labeled data. Security is enforced by type checking; the compiler reads a program containing labeled types and in type checking the program, ensures that the program cannot contain improper information flows at run time. The type system in such a language is a security-type system that enforces information-flow policies.

This chapter described the basic of sanitization, declassification and authentication mechanism in software and web application. Also here the purpose and requirements of those three functions (sanitization, declassification and authentication) are presented. At last of this chapter the mechanism of detecting information flow errors during design and code also presented.

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 vulnerability 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.

Three of the top five most common website attacks are SQL injection, cross-site scripting (XSS), and remote file inclusion (RFI). The root cause of these attacks is common: input sanitization. All three exploits are leveraged by data sent to the web server by the end user. When the end user is a good guy, the data he sends the server is relevant to his interaction with the website. But when the end user is a hacker, he/she can exploit this mechanism to send the web server input which is deliberately constructed to escape the legitimate context and execute unauthorized actions.

Input sanitization describes cleansing and scrubbing user input to prevent it from jumping the fence and exploiting security holes. But thorough input sanitization is hard. While some vulnerable sites simply don't sanitize at all, others do so incompletely, lending their owners a false sense of

security.

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.
- Prevent web attacks using input sanitization.

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 [74] 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 [66], 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.

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 [19, 21, 46, 33] 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 [65] 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 [64] and has been implemented in the Jif compiler [67].

- 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. [42]

- When: The fourth dimension of declassification is "when" information should be released.The work of Giambiagi and Dam [34] 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 [17] address when information is released.By annotating variables this is achieved.

For a given model, the "what" and "when" dimensions seem relatively straightforward to define formally. The "what" dimension abstracts the extensional semantics of the system; the "when" dimension can be distinguished from this since it requires an intensional semantics that (also) models time, either abstractly in terms of complexity or via intermediate events in a computation. The "who" and "where" dimensions are harder to formalize in a general way, beyond saying that they cannot be captured by the "what" and "when" dimensions.

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.

Protecting confidential data in computing environments has long been recognized as a difficult and daunting problem. All modern operating systems include some form of access control to protect files from being read or modified by unauthorized users. However, access controls are insufficient to regulate the propagation of information after it has been released for processing by a program. Similarly, cryptography provides strong confidentiality guarantees in open, possibly hostile environments like the Internet, but it is prohibitively expensive to perform nontrivial computations with encrypted data. Neither access control nor encryption provide complete solutions for protecting confidentiality.

A complementary approach, proposed more than thirty years ago, is to track and regulate the information flows of the system to prevent secret data from leaking to unauthorized parties. This can be done either dynamically, by marking data with a label describing its security level and then propagating those labels to all derivatives of the data, or statically, by analyzing the software that processes the data to determine whether it obeys some predefined policy with respect to the data. Arguably, a mostly static approach (perhaps augmented with some dynamic checks) is the most promising way of enforcing information-flow policies.

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
- Assurance of identity of person or originator of data.

A variety of authentication technologies have hit the market over the years.To better understand what they are and how they compare to the user name/password combination, it helps to be familiar with the standard authentication factors something you know, something you have and something you are and how each technology leverages them to power its authentication capabilities.

- Something you know is a bit of knowledge committed to memory, such as a password or an answer to a secret question.
- Something you have is an item that is owned or carried, such as a smart card or similar hardware device.
- Something you are is a physical attribute that can be identified, such as a fingerprint or voice.

According to the networkworld [68] now-a-days seven strong authentication methods are:

- Computer recognition software.
- Biometrics
- E-mail or SMS one-time password (OTP).
- One Time Password (OTP) token.
- Out of band.
- Peripheral device recognition.

- Scratch-off card.

2.4. Static Code Analysis

Static program analysis is the analysis of computer software that is performed without actually executing programs (analysis performed on executing programs is known as dynamic analysis). In most cases the analysis is performed on some version of the source code, and in the other cases, some form of the object code.The term is usually applied to the analysis performed by an automated tool, with human analysis being called program understanding, program comprehension, or code review. Software inspections and Software walkthroughs are also used in the latter case. .Static analysis, also called static code analysis, is a method of computer program debugging that is done by examining the code without executing the program. The process provides an understanding of the code structure, and can help to ensure that the code adheres to industry standards. Automated tools can assist programmers and developers in carrying out static analysis. The process of scrutinizing code by visual inspection alone (by looking at a printout, for example), without the assistance of automated tools, is sometimes called program understanding or program comprehension [3].

The principal advantage of static analysis is the fact that it can reveal errors that do not manifest themselves until a disaster occurs weeks, months or years after release. Nevertheless, static analysis is only a first step in a comprehensive software quality-control regime. After static analysis has been done, dynamic analysis is often performed in an effort to uncover subtle defects or vulnerabilities. In computer terminology, static means fixed, while dynamic means capable of action and/or change. Dynamic analysis involves the testing and evaluation of a program based on execution. Static and dynamic analysis, considered together, are sometimes referred to as glass-box testing [3].

Static code analysis advantages are:

- It can find weaknesses in the code at the exact location.
- It can be conducted by trained software assurance developers who fully understand the code.
- It allows a quicker turn around for fixes.
- It is relatively fast if automated tools are used.
- Automated tools can scan the entire code base.
- Automated tools can provide mitigation recommendations, reducing the research time.
- It permits weaknesses to be found earlier in the development life cycle, reducing the cost to fix.

According to the GCN [1] static code analysis limitations are:

- It is time consuming if conducted manually.
- Automated tools do not support all programming languages.
- Automated tools produce false positives and false negatives.
- There are not enough trained personnel to thoroughly conduct static code analysis.

- Automated tools can provide a false sense of security that everything is being addressed.
- Automated tools only as good as the rules they are using to scan with.
- It does not find vulnerabilities introduced in the runtime environment.

2.5. Information Flow Vulnerabilities

Information flow means transmission of information from one ”place” to another. Securing the data manipulated by computing systems has been a challenge in the past years. Several methods to limit the information disclosure exist today, such as access control lists, firewalls, and cryptography. However, although these methods do impose limits on the information that is released by a system, they provide no guarantees about information propagation. For example, access control lists of file systems prevent unauthorized file access, but they do not control how the data is used afterwards. Similarly, cryptography provides a means to exchange information privately across a non-secure channel, but no guarantees about the confidentiality of the data are given once it is decrypted.

In low level information flow analysis, each variable is usually assigned a security level. The basic model comprises two distinct levels: low and high, meaning, respectively, publicly observable information, and secret information. To ensure confidentiality, flowing information from high to low variables should not be allowed. On the other hand, to ensure integrity, flows to high variables should be restricted.For example, considering two security levels L and H (low and high), if L less equal to H, flows from L to L, from H to H, and L to H would be allowed, while flows from H to L would not [80].

A system is secure with respect to confidentiality should arise from a rigorous analysis showing that the system as a whole enforces the confidentiality policies of its users. This analysis must show that information controlled by a confidentiality policy cannot flow to a location where that policy is violated. The confidentiality policies we wish to enforce are, thus, information-flow policies and the mechanisms that enforce them are information-flow controls. Information-flow policies are a natural way to apply the well-known systems principle of end-to-end design [75] to the specification of computer security requirements; therefore, we also consider them to be specifications of end-to-end security. In a truly secure system, these confidentiality policies could be precisely expressed and translated into mechanisms that enforce them. However, practical methods for controlling information flow have eluded researchers for some time.

Information flow can be classified as explicit (information flow that arises explicitly, due to assignment statements), and implicit (flow that arises implicitly, due to conditional statements). In [55], proposed a new general-purpose static analysis for the inference of explicit information flow. This analysis is light-weight,works directly on Java programs before program execution, and does not require annotations by the programmer. It can be incorporated in program understanding and verification tools and help verify in a practical manner the confidentiality and integrity of sensitive program data. We believe that our work, although limited in the sense that it does not handle implicit flow, is a step forward towards the use of static analysis for the purposes of reasoning about information flow; it may help advance the use of static analysis in tools for understanding and verification of security properties.

2.6. 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 them 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.

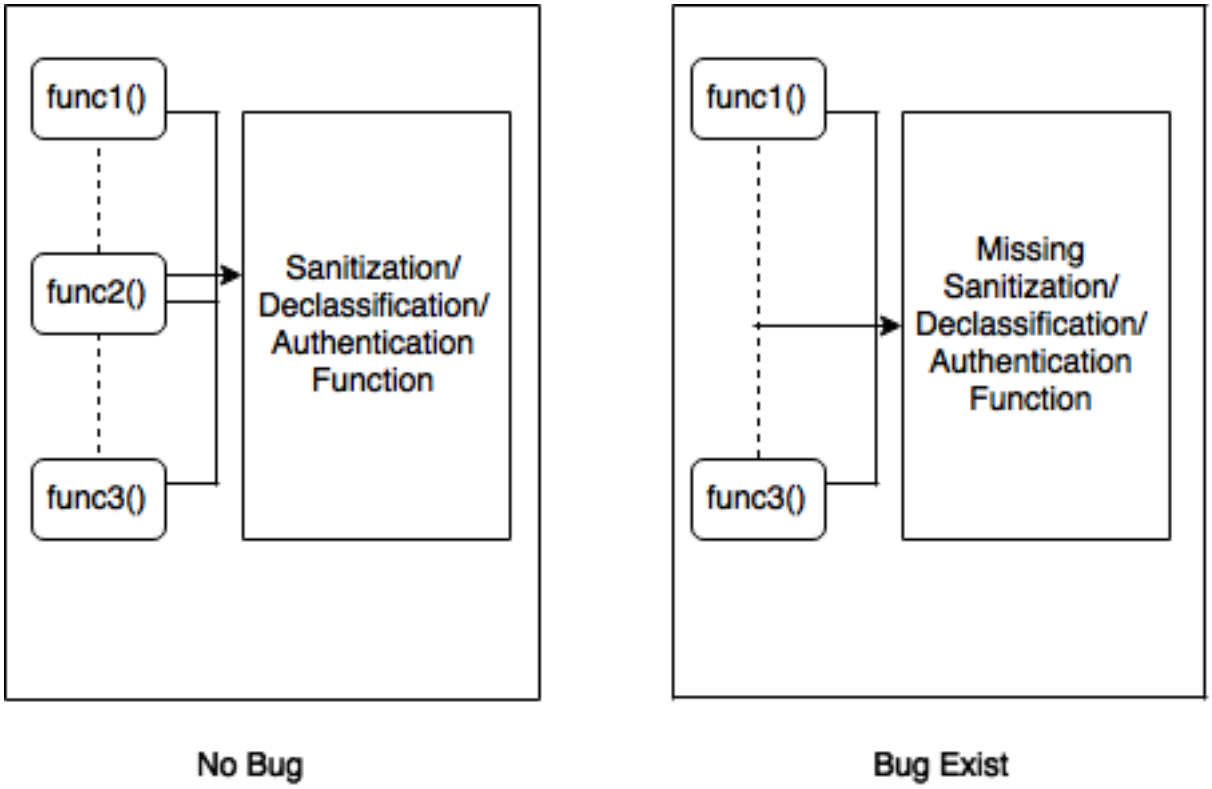


Figure 2.1.: Information flow errors during design

2.7. 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 [24] 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.

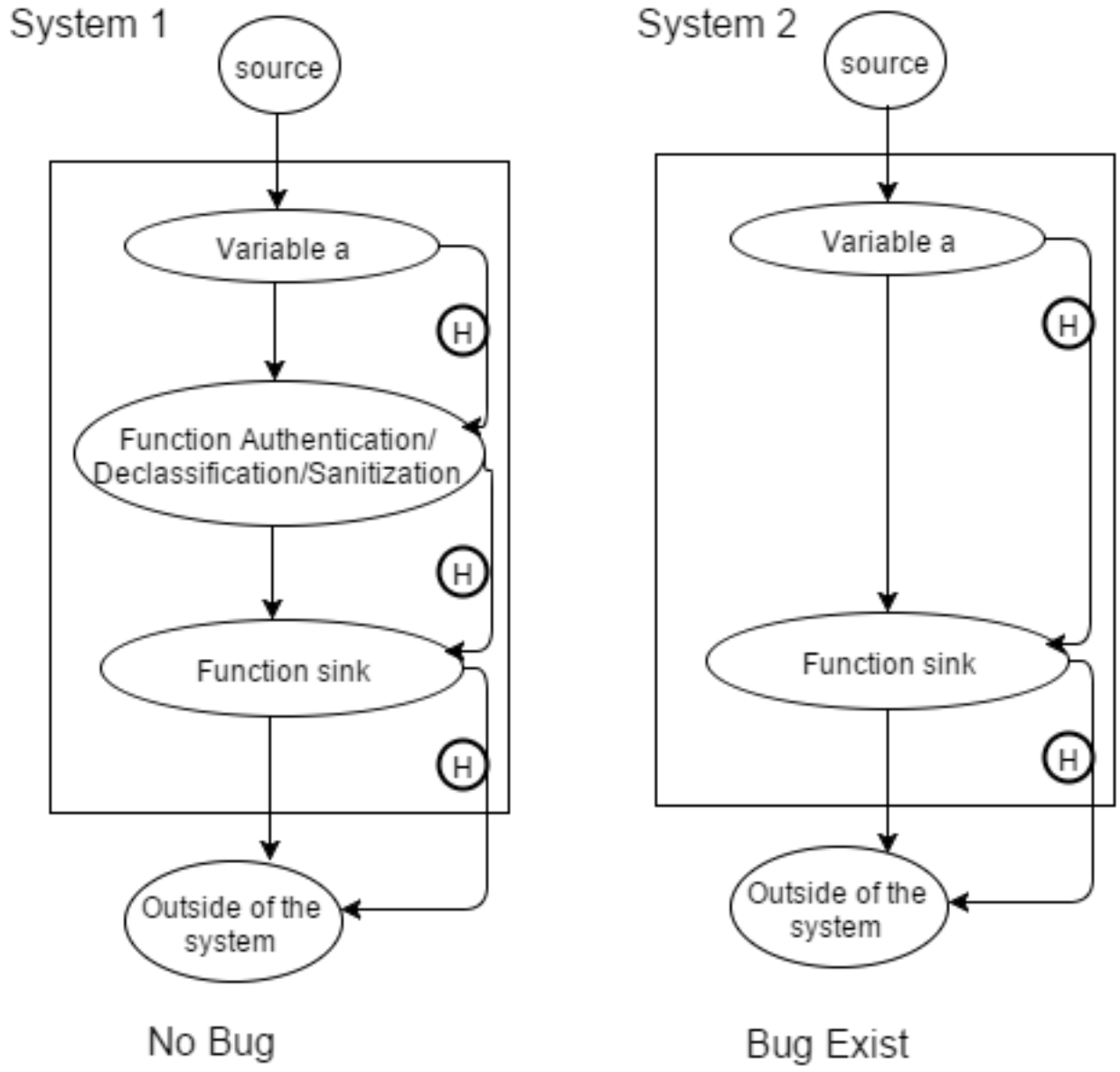


Figure 2.2.: Information flow errors during coding

Part II.

The Second Part

3. Challenges and Annotation Language Extension

Main goal was to overcome the challenge of not being able to detect implicit and explicit information flow bugs in UML state charts and C code. An annotation language which can be used to annotate UML state charts and code by inserting information flow restrictions during two software development phases (design and coding). The insight was that 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.

In this chapter the challenges and how the annotation language has extended are described.

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++ vulnerabilities) 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 [27], 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!
```

How Xtext Works: Xtext provides user with a set of domain-specific languages and modern APIs to describe the different aspects of user's programming language. Based on that information it gives user a full implementation of that language running on the JVM. The compiler components of user's language are independent of Eclipse or OSGi and can be used in any Java environment. They include such things as the parser, the type-safe abstract syntax tree (AST), the serializer and code formatter, the scoping framework and the linking, compiler checks and static analysis aka validation and last but not least a code generator or interpreter. These runtime components integrate with and are based on the Eclipse Modeling Framework (EMF), which effectively allows user to use Xtext together with other EMF frameworks like for instance the Graphical Modeling Project GMF.

In addition to this nice runtime architecture, user will get a full blown Eclipse-IDE specifically tailored for user's language. It already provides great default functionality for all aspects and again comes with DSLs and APIs that allow to configure or change the most common things very easily. And if that's not flexible enough there is Guice to replace the default behavior with user's own implementations.

Domain-Specific Language: A Domain-Specific Language (DSL) is a small programming language, which focuses on a particular domain. Such a domain can be more or less anything. The idea is that its concepts and notation is as close as possible to what you have in mind when you think about a solution in that domain.

The opposite of a DSL is a so called GPL, a General Purpose Language such as Java or any other common programming language. With a GPL you can solve every computer problem, but it might not always be the best way to solve it.

Imagine you want to remove the core from an apple. You could of course use a Swiss army knife to cut it out, and this is reasonable if you have to do it just once or twice. But if you need to do that on a regular basis it might be more efficient to use an apple corer.

There are a couple of well-known examples of DSLs. For instance SQL is actually a DSL which focuses on querying relational databases. Other DSLs are regular expressions or even languages provided by tools like MathLab. Also most XML languages are actually domain-specific languages. The whole purpose of XML is to allow for easy creation of new languages. Unfortunately, XML uses a fixed concrete syntax, which is very verbose and yet not adapted to be read by humans. Into the bargain, a generic syntax for everything is a compromise.

Xtext is a sophisticated framework that helps to implement your very own DSL with appropriate IDE support. There is no such limitation as with XML, you are free to define your concrete syntax as you like. It may be as concise and suggestive as possible being a best match for your particular domain. The hard task of reading your model, working with it and writing it back to your syntax is greatly simplified by Xtext.

Users of Xtext: Xtext is used in many different industries. It is used in the field of mobile devices, automotive development, embedded systems or Java enterprise software projects and game development. People use Xtext based languages to drive code generators that target Java, C, C++, C sharp, Objective C, Python, or Ruby code. Although the language infrastructure itself runs on the JVM, you can compile Xtext languages to any existing platform. Xtext based languages are developed for well known Open-Source projects such as Maven, Eclipse B3, the Eclipse Webtools platform or Google's Protocol Buffers and the framework is also widely used in research projects.

- Xtend : According to the [26], Xtend is a statically-typed programming language which trans-

lates 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).

Java Interoperability: Xtend, like Java, is a statically typed language. In fact it completely supports Java's type system, including the primitive types like int or boolean, arrays and all the Java classes, interfaces, enums and annotations that reside on the class path.

Java generics are fully supported as well: You can define type parameters on methods and classes and pass type arguments to generic types just as you are used to from Java. The type system and its conformance and casting rules are implemented as defined in the Java Language Specification.

Resembling and supporting every aspect of Java's type system ensures that there is no impedance mismatch between Java and Xtend. This means that Xtend and Java are 100% interoperable. There are no exceptional cases and you do not have to think in two worlds. You can invoke Xtend code from Java and vice versa without any surprises or hassles. As a bonus, if you know Java's type system and are familiar with Java's generic types, you already know the most complicated part of Xtend.

The default behavior of the Xtend to Java compiler is to generate Java code with the same language version compatibility as specified for the Java compiler in the respective project. This can be changed in the global preferences or in the project properties on the Xtend Compiler page (since 2.8). Depending on which Java language version is chosen, Xtend might generate different but equivalent code. For example, lambda expressions are translated to Java lambdas if the compiler is set to Java 8, while for lower Java versions anonymous classes are generated.

Type Inference: One of the problems with Java is that you are forced to write type signatures over and over again. That is why so many people do not like static typing. But this is in fact not a problem of static typing but simply a problem with Java. Although Xtend is statically typed just like Java, you rarely have to write types down because they can be computed from the context.

Consider the following Java variable declaration:

```
final LinkedList<String> list = new LinkedList<String>();
```

The type name written for the constructor call must be repeated to declare the variable type. In Xtend the variable type can be inferred from the initialization expression:

```
val list = new LinkedList<String>
```

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

Table 3.1 contains in this section: the annotation language target types, the annotation tags which can be used in combination with the tag @function, the tag @parameter can be used to annotate the function parameter as authinticated/declassified/santized H/L and the tag @variable used to annotate the variable of C/C++ code with confiential H/L which are used to tag public and private variables.

Annotation Type	Annotation Tag	Description
@function	sink	uses information
	source	source provides information
	authentication	authentication authenticates information
	declassification	declassification declassifies information
	sanitization	sanitization sanitizes information
	trust_boundary	trust_boundary is a trust-boundary
@parameter	authenticated H/L	authenticated High/Low tags
	declassified H/L	declassified High/Low tags
	sanitized H/L	sanitized High/Low tags
@variable	confidential H/L	confidential High/Low tags
	source H/L	source High/Low tags
@preStep	preStep	previous function call name
@postStep	postStep	next function call name

Table 3.1.: Security language annotation tags

The tag @variable which can be used only inside single line annotations whereas @parameter is used only in multi line annotations. The tags were defined and implemented iteratively based on the work flow presented in figure 3.1 and by using the eclipse xtext [27] language definition grammar.

For detecting of authentication, declassification and sanitization errors new function tags included like authentication, declassification and sanitization function type. Also for parameter new tag type of parameter included such as authenticated, declassified and sanitized. Still H/L tags for parameter exists in the annotation tags for parameter to define that which type of parameter is this either “High” or “Low”. High means that this parameter is highly confidential or secured and low means that this parameter is not highly secured. The tag preStep used to annotate the previous function call name and tag postStep is used for next function call name. In Table 3.1 the new tags for annotation language grammar has given with previous annotation tags.

3.3. Annotation Language Implementation Process

To implement the annotation language Eclipse Xtext [27] was used. Xtext is a sophisticated framework that helps to implement own DSL(Domain Specific Language) with appropriate IDE support. There is no such limitation as with XML, user’s are free to define user’s concrete syntax as user like. It may be as concise and suggestive as possible being a best match for user’s particular domain. The hard task of reading user model, working with it and writing it back to user’s syntax is greatly simplified by Xtext. Xtext relies heavily on Eclipse Modeling Framework (EMF) internally, but it can also be used as the serialization back-end of other EMF-based tools.

Xtext provides a lot of generic implementations for language’s infrastructure but also uses code generation to generate some of the components. Those generated components are for instance the parser, the serializer, the inferred Ecore model (if any) and a couple of convenient base classes for content assist etc.The generator also contributes to shared project resources such as the plugin.xml, MANIFEST.MF and the Guice modules. Xtext’s generator uses a special DSL called MWE2 - the modeling workflow engine to configure the generator. MWE2 allows to compose object graphs declaratively in a very compact manner. The nice thing about it is that it just instantiates Java classes and the configuration is done through public setter and adder methods as one is used to from Java Beans encapsulation principles.

Xtext itself and every language infrastructure developed with Xtext is configured and wired-up using dependency injection. Xtext may be used in different environments which introduce different constraints. Especially important is the difference between OSGi managed containers and plain vanilla Java programs. To honor these differences Xtext uses the concept of ISetup (src)-implementations in normal mode and uses Eclipse's extension mechanism when it should be configured in an OSGi environment.

The Modeling Workflow Engine 2 (MWE2) is a rewritten backwards compatible implementation of the Modeling Workflow Engine (MWE). It is a declarative, externally configurable generator engine. Users can describe arbitrary object compositions by means of a simple, concise syntax that allows to declare object instances, attribute values and references. One use case - that's where the name had its origins - is the definition of workflows. Such a workflow consists usually of a number of components that interact with each other. There are components to read EMF resources, to perform operations (transformations) on them and to write them back or to generate any number of other artifacts out of the information. Workflows are typically executed in a single JVM. However there are no constraints the prevent implementors to provide components that spawn multiple threads or new processes.

Xtext ships with a default set of predefined, reasonable and often required terminal rules. The grammar for these common terminal rules is defined as follows:

```
grammar org.eclipse.xtext.common.Terminals
hidden(WS, ML_COMMENT, SL_COMMENT)
import "http://www.eclipse.org/emf/2002/Ecore" as ecore
terminal ID :
'^'?('a'..'z'|'A'..'Z'|'_')('a'..'z'|'A'..'Z'|'_'|'0'..'9')*;
terminal INT returns ecore::EInt:
('0'..'9')+;
terminal STRING :
'"' ( '\\'('b'|'t'|'n'|'f'|'r'|'u'|'"'|"'"|'\\') | !('\\'|'"') )* '"' |
"'" ( '\\'('b'|'t'|'n'|'f'|'r'|'u'|'"'|"'"|'\\') | !('\\'|"'") )* "'";
terminal ML_COMMENT :
'/*' -> '*/';
terminal SL_COMMENT :
'//' !('\n'|'\r')* ('\r'? '\n')?;
terminal WS :
(' '|'\t'|'\r'|'\n')+;
terminal ANY_OTHER:
.;
```

In order to implement the annotation language grammer in this research it was required to extend the terminal rule ML_COMMENT and SL_COMMENT. After extending these two rules it looks like this:

```
/**
* @SL_COMMENT :all strings which follow // | || | } will be a single line
    comment
*/
terminal SL_COMMENT  : '//'!('@') !('\n'|'\r')* ('\n'|'\r')*
// '}' can be used optional to disable the method bodyes together with
    multiline line {} comment
//    | '}'      !('\n'|'\r')* ('\n'|'\r')*
;

/**
* @ML_COMMENT :@/* multiline comment excluding @ from inside
*         :{} multi line comment
*/
```

```
terminal ML_COMMENT : '/*' !('@') -> !('@')'*/' !('\n'|'\r')* ('\n'|'\r')*
// '{' -> '}' can be used optional to disable the method bodyes together with
    single line { comment
   //    | '{' -> '}'            ('\n'|'\r')?
   ;
```

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.

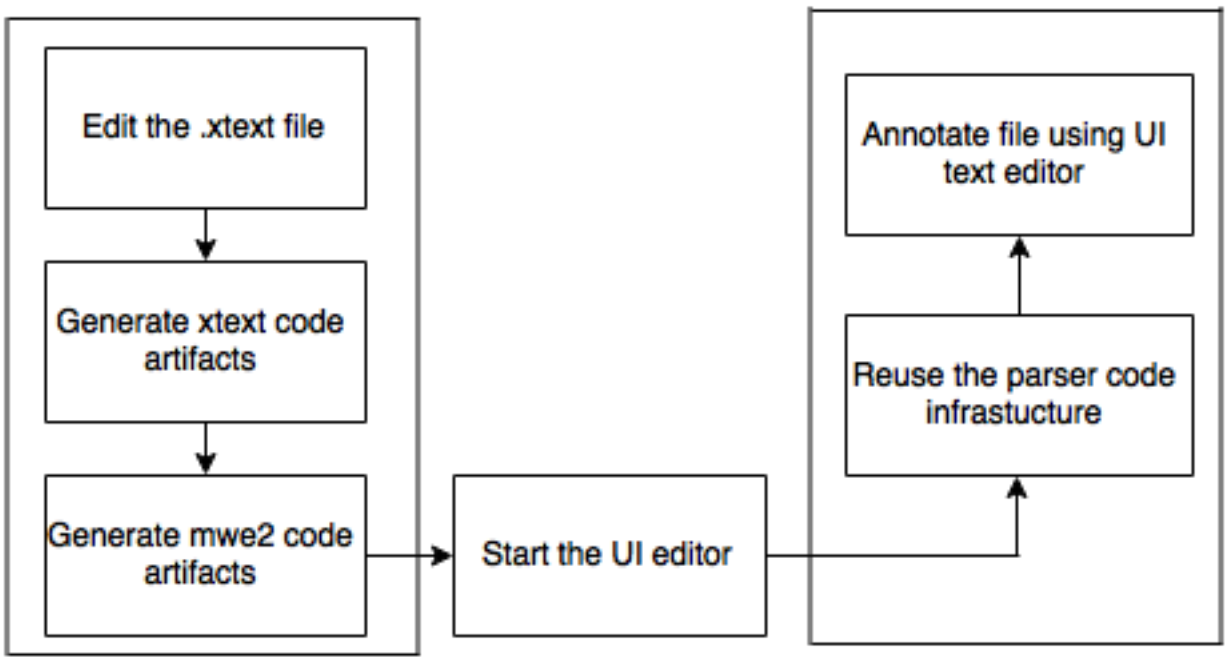


Figure 3.1.: Annotation language design process

4. Implementation

This chapter focus mainly about the software implementation for semi-automated detection of sanitization, authentication and declassification errors in UML state Charts. Used Eclipse xtext to develop source code editor, Eclipse xtend to develop source code generator, YAKINDU SCT editor for modeling C/C++ programs as UML statchart and extended static analysis engine named smtcodan using Java, Graphics2D and Jframe.

4.1. Overview of System Architecture

A system architecture or systems architecture is the conceptual model that defines the structure, behavior and more views of a system. An architecture description is a formal description and representation of a system, organized in a way that supports reasoning about the structures and behaviors of the system.

It can be said that the system architecture is (similar to the one of a building architecture) a global model of this system consisting of:

- a structure
- properties (of various elements involved)
- relationships (between various elements)
- behaviors and dynamics
- multiple views of the system (complementary and consistent).

System Architecture is based on 9 fundamental principles:

- The objects of the reality are modelled as systems
- A system can be broken down into a set of smaller subsystems, which is less than the whole system
- A system must be considered in interaction with other systems
- A system must be considered through its whole lifecycle
- System can be linked to another through an interface, which will model the properties of the link
- A system can be considered at various abstraction levels, allowing to consider only relevant properties and behaviors

- A system can be viewed according to several layers (usually three: its sense, its functions, and its composition)
- A system can be described through interrelated models with given semantics (properties, structure, states, behaviors, datas, etc)
- A system can be described through different viewpoints corresponding to various actors concerned by the system.

The architecture of our system is given below:

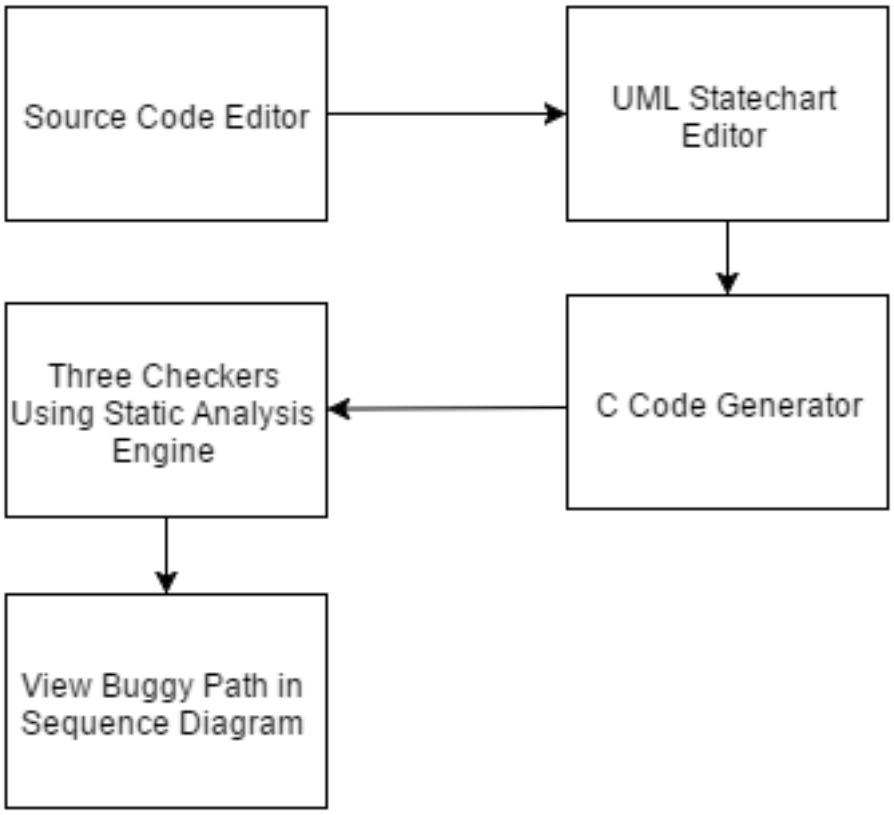


Figure 4.1.: System Overview

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 this annotation technique has chosen which is easy to extend and backward compatible. This editor has developed as an Eclipse plug-in. If this plug-in exist in eclipse then user can easily annotate C/C++ source code files and header files by pressing the keys ctrl+space in keyboard. Then for modeling purpose open source tool Yakindu SCT editor [2] 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. Inside the static analysis engine according to the requirements new modules

are added. 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

Annotation language is developed using Eclipse xtext [27]. The goal is to overcome the challenge of not being able to detect implicit and explicit information flow bugs in UML state charts and C code. That's why an annotation language has chosen which can be used to annotate UML state charts and code by inserting information flow restrictions during two software development phases (design and coding). Our insight is that 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 grammar of annotation language is represented as in Extended Backus Naur Form (EBNF) in Figure 4.2. The following type face conventions were used: Italic font for non-terminals, bold typewriter font for literal terminals including keywords.

All main rules are included under `H_Model`. This `H_Model` rule contains all rules like `S_L_Anno`, `M_L_Anno`, `Func_Ann` and `Attr_Def`. Annotation language grammar has two grammar rules named `S_L_Anno` and `M_L_Anno` used for defining security annotations. `S_L_Anno` is used for single line annotation rule. Because of this rule one can easily annotate the variables of C/C++ language in a single line. The `M_L_Anno` rule is used for multiline annotation rule. Normally multiline rule annotation is required for function annotation in C/C++ function declaration. The `Func_Ann` 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_Type` rule the type of function is declared. A function can be either one of this like authentication, declassification, sanitization, source or sink.

4.3. UML State Chart Editor

UML state machine diagrams depict the various states that an object may be in and the transitions between those states. In fact, in other modeling languages, it is common for this type of a diagram to be called a state-transition diagram or even simply a state diagram. A state represents a stage in the behavior pattern of an object and like UML activity diagrams it is possible to have initial states and final states. An initial state, also called a creation state, is the one that an object is in when it is first created, whereas a final state is one in which no transitions lead out of. A transition is a progression from one state to another and will be triggered by an event that is either internal or external to the object.

The following are the basic notational elements that can be used to make up a diagram:

- Filled circle, representing to the initial state.
- Hollow circle containing a smaller filled circle, indicating the final state (if any).
- Rounded rectangle, denoting a state.
- Arrow, denoting transition.

Ann_Lang ::= HeaderModel*;

H_Model ::= *S_L_Anno*; ;single line comment rule
| *M_L_Anno*; ;multi line comment rule
| *Func_Ann*; ;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

A set of formal representation of UML state charts 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 [4] , 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.

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

Invariants:

- The start state is not in the collection of states.
- The start state is not the target state.
- 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 [2]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 [27].

As per requirements previous annotation language grammar which was written in xtext language has been extended. Extra annotation have included like “authenticated”, “declassified”, “sanitized”, “sanitization”, “declassification”, “authentication”. Mainly FunctionAnnotation, FunctionType and SingleLineAnnotation rules are extended. A new rule is added which is enumeration type. The rule

name is VariableType. Inside the VariableType new attributes are included like declassified, sanitized and authenticated. New function types are added like declassification, sanitization and authentication function inside FunctionType rule. Inside the FunctionAnnotation and SingleLineAnnotation rules there exist an annotation for parameter name @parameter. Inside @ parameter declaration new attribute is added named VariableType. 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
;
```

From the above xtext code snippet previous annotation grammar has FunctionAnnotation, SingleLineAnnotation rules. According to the requirements previous rules are extended. Inside the rules of FunctionAnnotation and SingleLineAnnotation new enumeration types are added. Enumeration

type name is VariableType. The new enumeration type rule has attributes declassified, sanitized and authenticated. Also the rule of enumeration FunctionType extended by adding new type of funtion like authentication, declassification and sanitization.

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 annotations 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.

The algorithm is given below how the C code generator has been created. The input of the algorithm for code generator is UML statechart. In eclipse xtend [26] function can be declared as "def". Here the generateTypeH function requires the input of UML statechart. The plug-in named "MyC" uses eclipse xtend to parse the UML statechart. Inside this function there another two functions named "typesHAnnotationContent" to generate header file of c(extension .h) and "typesCAnnotationContent" to generate source code file of c(extension .c).Function typesHAnnotationContent generate the required contents for C header file mostly function signature and annotation of the function which exist in the UML statechart. One sample example of a header file is given below-

```
/*@ @function authentication
* @parameter a L @*/
void authentication(char *a);

/*@ @function source
* @parameter a L @*/
void logIn(char *a);
```

Function typesCAnnotationContent generate the required contents for C source file. This file contains the annotation only for variable declaration. The function annotation is normally located at header file. In this file other code is as normal as C syntax. All functions,statements, variable declaration ar similar to C programming language syntax. Some of the contents of the C file comes from another xtend file named "Naming.xtend" which contains in "MyC" project folder of YAKINDU Sct Editor. some part of code snippet from "Naming.xtend" file is given below:

Algorithm 4.1 *C code generator*

Input: Statechart
Output: .c and .h files

1: ***function*** GENERATETYPESH(*sc*) ▷ *Where sc - statchart*
2: $def generateFile_1(testModule.h, typesHAnnotationContent(sc))$ *Where def - function declaration in xtend*
3: $def generateFile_2(testModule.c, typesCAnnotationContent(sc))$
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
```

Below from the xtend code snippet it can be seen that from the state chart xtend can easily accessible the contents of the state chart which is designed in the modeling stage. For example in case of the function "getFunctionContent" it parses the function names and put it inside a hash map. It parses those as a function who is a state and contains first bracket like "(". Inside the hashmap it puts the function name and function annotation. In case of the function "getBadPathContent" which returns the content for the bad path. From the content of state chart there is a region which named is "bad_path()". From that region this function parses the comments from each transition and gets the name of each state. Then it puts those contents into a hash map. Through iterating that map according to the algorithm it puts some part of contents in C header file and some part in source code

file. The purpose of function “getGoodPathContent” is to get all the required content from the good path(which is not buggy path). The parameter of this function is the state chart. In the modeling phase it has been declared as a region named ”good_path()”. This “getGoodPathContent” function starts parsing the contains from good path then traverse the whole good path and get all the required contents. After that this function puts the content into a hash map. This hashmap also contains the function name, function annotation, variable declaration which has annotation. Then according to the algorithm by iterating through the hash map generates the required files by putting the contents in proper place.

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

```
                                    for(transition : vertex.incomingTransitions) {
                                        goodfunctionContent.put(transition.specification,ver

                                    }
                                }
                                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. Inside the sequence diagram all function calls are included inside rectangular box attached to the head of timeline. To switch one timeline to another a function call required. Like in the figure 4.7 from function call int main() there is a transition to move from int main() to good_path(). Above the transition there also exist the function name for which it goes from one function call to another. Here for example to move from int main() to good_path() there is a transition and above that transition function call good_path() is attached. This means through calling the good_path() function buggy path goes into good_path() function from int main(); Inside the sequence diagram the statements before the function call are attached to the timeline as blue color. Those statements are just the statements before a function of the analyzed C/C++ source code file. Inside the box of the blue color texts there exist "ln:" which means the line number of the statement in the analyzed file and "fn:" means the file name of analyzed file.Now it is easier to trace the buggy path by viewing generated sequence diagram.

One sample example of the part of a buggy path is given in figure 4.7.

The process of creating sequence diagram using Java programming language is given below as an algorithm representation. To develop the sequence diagram a separate class is included inside the smtcodan project. The class name is SequenceDiagramGenerator. Inside this class the drawSequence-Diagram function creates the diagram. The function input parameter is a list of IASTNode which is delivered from the smtcodan project packages. To draw the diagram BufferedImage, Graphics2D, JPanel, JFrame classes were used. At first need to declare object each of these(BufferedImage, Graphics2D, JPanel, JFrame) classes. Then iterating through the list of IASTNodes set all the statements except function call with line number and file name. There is a class which is responsible to draw the visual things of the sequence diagram named MyCanvasDraw. Afterwards need to create an object of this class. This class has a list. The list of this class has to be set with the list of all statements, line number and file name. Then it will draw the diagram according to the list. To view this diagram need to set the JFrame object and make it visible. Inside the JFrame JScrollpane object also added to make the frame scrollable. For saving the diagram as an image save as option also included. Through the menu bar user can easily save the sequence diagram as an image. By default it will save the image as in .jpg format. But one can easily save this image in other format like .png,.bmp,.jpeg etc. Exit option is also included inside the JFrame through which user can exit the current window.

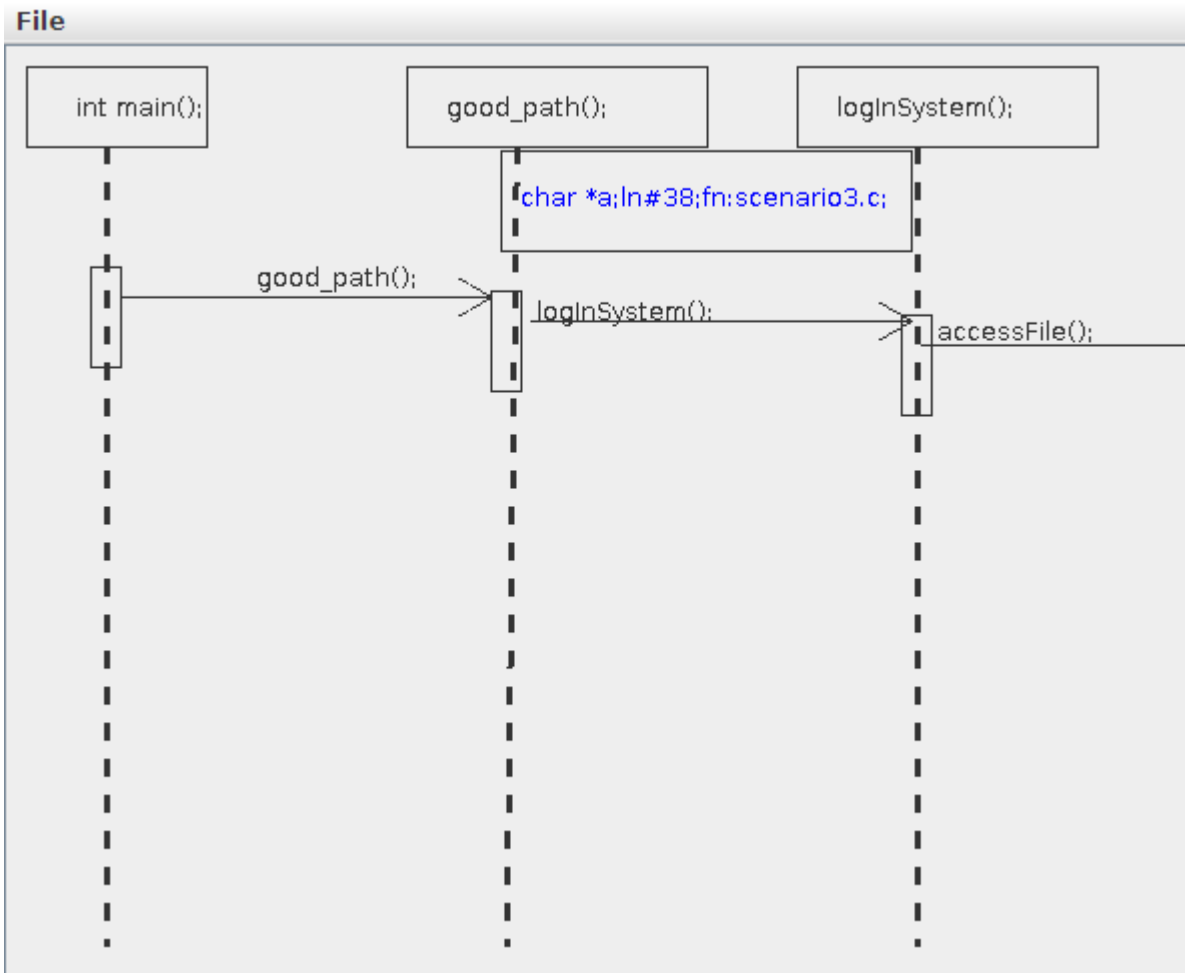


Figure 4.7.: Error trace path in UML sequence diagram

Algorithm 4.2 *Sequence diagram generator*

Input: List of statements and function call
Output: Sequence diagram in a Jframe

1: ***function*** DRAWSEQUENCEDIAGRAM*(*$ArrayList < IASTNode > statementsList$*)*
2: $frame < -JFrame$
3: $image < -BufferedImage$
4: $graphics2D < -Graphics2D$
5: $topPanel < -JPanel$
6: $mcd < -MyCanvasDraw$
7: ***for*** $i : statementsList.size()$ ***do***
8: ***if*** $!statementsList.get(i).getRawSignature().toString().contains("(")$ ***then***
9: $intj = i;$
10: ***if*** $j <= statementsList.size() - 2$ ***then***
11: do
12: $add(linenumber)$
13: $add(filename)$
14: $while(!statementsList.get(j).getRawSignature().toString().contains("("))$
15: $mcd.buggyPathList.add(allStatements);$ ▷ *Where allStatements - statements with containing line number and file name also*
16: ***end if***
17: ***end if***
18: ***end for***
19: $set < -JScrollPane(property)$
20: $mcd.paint(graphics2D)$
21: $topPanel.add(mcd);$
22: $menuBar < -JMenuBar$
23: $menuFile < -JMenu$
24: $menuFileExit < -JMenuItem$
25: $menuSaveAs < -JMenuItem$
26: $menuSaveAs.addActionListener < -fileSaveandexit$
27: $set < -frame(properties = visibility, menubar)$
28: ***end function***

5. Experiments

In this chapter experimental results are presented for semi-automated detection of sanitization, authentication and declassification errors in UML state Charts. For three separate problems such as authentication,declassification and sanitization three sample programs selected in C language. Then according to the system these three scenarios are modeled in YAKINDU SCT editor. After this source code file generated and analyzed using static analysis engine. Detailed process how the system works and what is the outcome of the system are given below.

5.1. Authentication Scenario

To understand the authentication scenario a simple example has chosen. The scenario contains a user account creation inside database to access the contents of the database. To access the database contents user have to be authorized by the database administrator or from who is responsible to do the authorization. User creation inside a database now-a-days normally does the database administrator. When database administrator creates the user then user can access the contents of the database otherwise not. In the following Java example the method createUser is used to create a DBAccess object for a database management application.

```
public DBAccess createUser(String userName, String userType,
 String userPassword) {

     DBAccess access = new DBAccess();
     access.setUserName(userName);
     access.setUserType(userType);
     access.setUserPassword(userPassword);

     return access;
}
```

However, there is no authentication mechanism to ensure that the user creating this database user account object has the authority to create new user access. Some authentication mechanisms should be used to verify that the user has the authority to create database access objects. The following Java code includes a boolean variable and method for authenticating a user. If the user has not been authenticated then the createUser will not create the database access object.

```
private boolean isUserAuthentic = false;

// authenticate user,
// if user is authenticated then set variable to true
// otherwise set variable to false
public boolean authenticateUser(String username, String password) {
...
}

public DBAccess createUser(String userName, String userType,
```

```
String userPassword) {
       DBAccess access = null;

       if (isUserAuthentic) {
       access.setUserName(userName);
       access.setUserType(userType);
       access.setUserPassword(userPassword);
     }
   return access;
}
```

To model this kind of scenario in UML statechart considering that for C/C++ application.Figure 5.1 represents an authentication scenario. In this scenario a user *A* wants to access a database. At first he/she has to provide his/her user id, password, account number. Then he/she sends request to access database. The database administrator creates a user using his/her id,name and password based on their policy. According to the policy user can either view or access the database or not. If database server doesn't have some secured policy like validation,encryption, decryption or authentication methodology then hacker may easily break the application and receives the confidential information or data from the database server. In Figure 5.1 depicts a highly secured variable "char *a" which is initially annotated as H. The annotation is attached with the state named "char *a". Annotation are included on the transition from state "accessDatabase()" to "char *a". On the transition annotation is look like this "//@@ variable a H False". The variable "char *a" passes through a function named authentication. This authentication function is represented as a state in the statechart as like "void declassification(char *a)". Annotation of this state "void declassification(char *a)" is "/*@ @function authentication * @parameter a H authenticated @*/" which is attached with the transition from "char *a" to "void declassification(char *a)". This function makes the high secured variable as low by following the policy language. After passing these function the variable "char *a" is annotated with L and authenticated. Now it can be passed to other system or release information to the authenticated user. While moving from "void declassification(char *a)" to "void logIn(char *a)" state there is another annotation which contains annotation "/*@ @function sink * @parameter a L @*/". This logIn function is a sink type function and it expects is one of the parameter is low. If the parameter doesn't come through the authentication function then it remains H(High). Then bug should be triggered. Here in this scenario inside the bad_path() region bug should be triggered because there is no authentication function exist. But if the parameter comes through the authentication function then the parameter will change to L(low). In this scenario inside the good_path() region there is no bug because it has the authentication function and the variable "char *a" passes through authentication function.

5.2. Declassification Scenario

Noninterference is typically too strong a property, most programs use some form of declassification to selectively leak high security information when performing a password check or data encryption. Unfortunately, such a declassification is often expressed as an operation within a given program, rather than as part of a global policy, making reasoning about the security implications of a policy more difficult. For application programmers need to prevent a range of problems, from SQL injection and cross-site scripting, to inadvertent password disclosure and missing access control checks.Adding declassification function is one of the possibility for an application to detect information flow vulnerabilities. It requires few changes to the existing application code and an assertion of functions such as declassification, sanitization and authentication can reuse existing code and data structures.

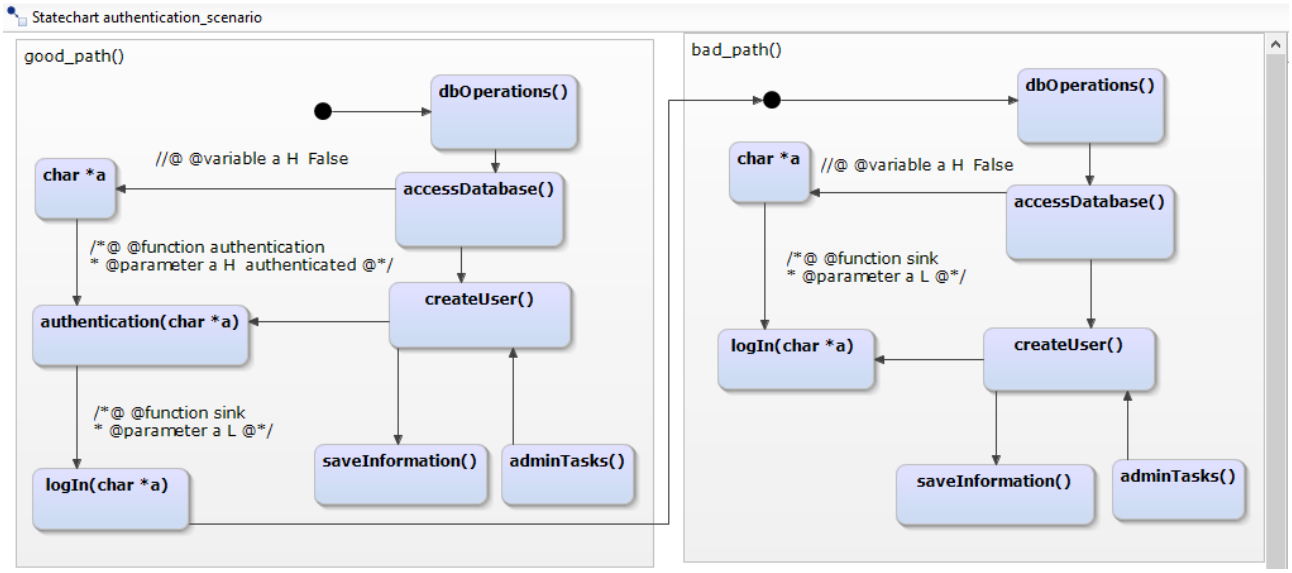


Figure 5.1.: UML Statechart Modeling for Authentication Scenario

For the declassification scenario, a user *A* wants to access his bank account. Every bank has their own policy to their customer who can access their account information. After giving the password, account number, user name user *A* send his request to the bank server to view the account information. The bank server has his own policy according to their requirements. Through that policy bank server verifies the user *A* may be by following the basic declassification goals according to four axes like what information is released, who releases information, where in the system information is released and when information can be released.

Figure 5.2 represents a declassification scenario. In this scenario a user *A* wants to access his bank account. At first he/she has to provide his/her user id, password, account number. Then he/she sends request to access his/her account. The bank server has their own policy who can access the account details. According to the policy user can either view his account details or not. If bank server doesn't have some secured policy like encryption, decryption or declassification methodology then hacker may easily break the application and receives the confidential information or data. In Figure 5.2 depicts a highly secured variable "char *a" which is initially annotated as H. State "char *a" has an incoming transition from state "logInSystem()". That transition contain annotation like this "//@ @ variable a H False". Variable "char *a" passes through a function named declassification. This declassification function is represented as a state in the statechart as like "void declassification(char *a)". This declassification function has an annotation which is like this "/*@ @function declassification * @parameter a H @*/". Declassification function makes the high secured variable as low by following the policy language. After passing these function the variable "char *a" is annotated with L and declassified. Now it can be passed to other function or release information to the verified user. Here in declassification scenario there is a accessAccount function call. The state "accessAccount(char *a)" expect a parameter which is L(low). If the parameter is low then there would be no bug. But here in case of bad_path() region there is no declassification method that's why the variable "char *a" remains H(high). In case of bad_path() region the state "accessAccount(char *a)" gets the parameter as H(high) that's why bug should be triggered here.

After finishing the modeling of declassification scenario then through the C code generator C code files are generated which consist two files(.c and .h file). Inside those files annotations are exist. Then through static analysis engine named "smtcodan" analyze the code and detect the information flow

vulnerabilities if they exist inside the generated files.

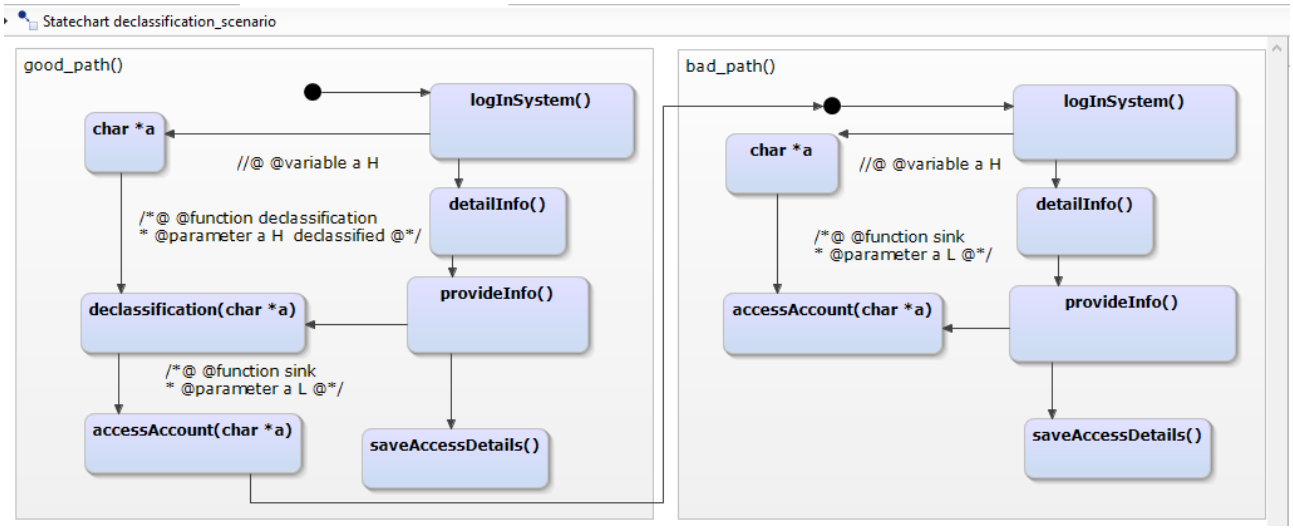


Figure 5.2.: UML Statechart Modeling for Declassification Scenario

5.3. Sanitization Scenario

Web applications are often implemented by developers with limited security skills.As a result, they contain vulnerabilities. Most of these vulnerabilities stem 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. In the past research on vulnerability analysis has mostly focused on identifying cases in which a web application directly uses external input in critical operations. However, little research has been performed to analyze the correctness of the sanitization process. Secured web application helps to prevent the bad guys from gaining unauthorized access to your application or website data. It helps you keep your data's integrity and ensures availability as needed. Sql injection and XSS attacks are common attacks now-a-days. to prevent this kind of attacks need to use sanitization methods and validate user input properly.
This example code intends to take the name of a user and list the contents of that user's home directory. It is subject to the first variant of OS command injection.Example language is in PHP.

```
$userName = $_POST["user"];
$command = 'ls -l /home/' . $userName;
system($command);
```

The userName variable is not checked for malicious input. An attacker could set the userName variable to an arbitrary OS command such as:

```
;rm -rf /
```

Then that would produce a result like this-

```
ls -l /home/;rm -rf /
```

Since the semi-colon is a command separator in Unix, the OS would first execute the ls command, then the rm command, deleting the entire file system.

The previous example was given for PHP language. In C language here it has been selected that user "A" wants to access some file from server pc. So, he needs to give the file name. If he wants to access some exe file then bug should be triggered or if he inserts some OS command injection type of statement then bug should be triggered. That's why the user input should pass through a sanitization method to sanitize the input parameter.

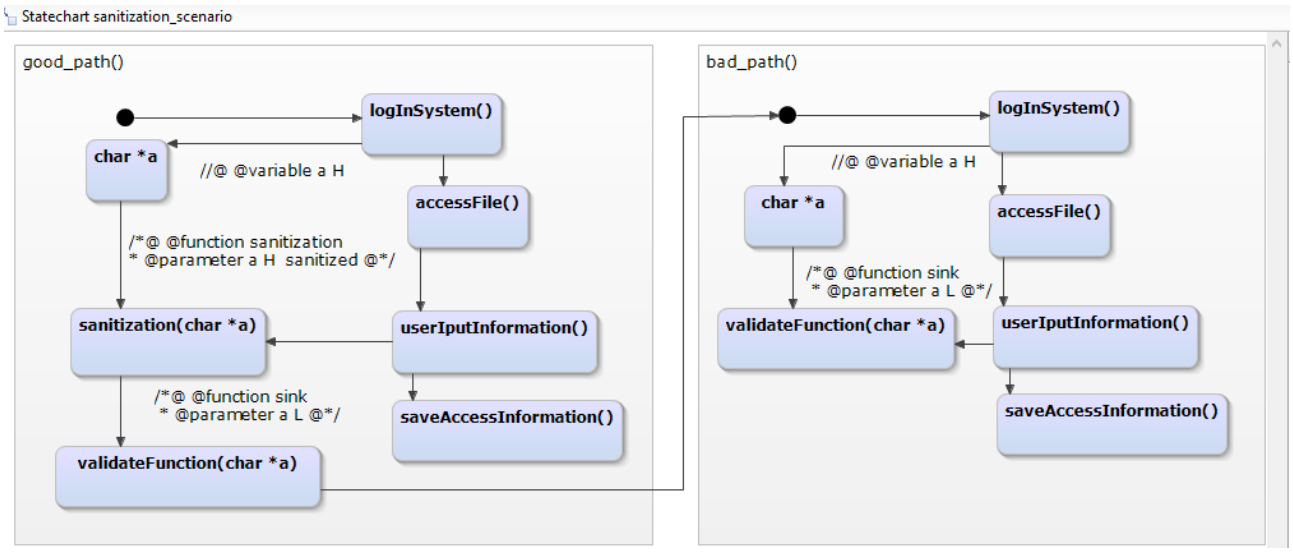


Figure 5.3.: UML Statechart Modeling for Sanitization Scenario

Figure 5.3 depicts a sanitization scenario. In this scenario a user *A* wants to access file from a server. At first he/she has to provide file name. Then he/she sends request to access his/her desire file. The server has their policy who can access the file. According to the policy if the server has proper sanitization methods then by sanitizing the user input server either allow or disallow the user to access the file. If the server doesn't have some secured policy like encryption, decryption or sanitization methodology then hacker may easily break the application and receives the confidential information or data, even execute the .exe files or may do some OS command injection. In Figure 5.3 shows a highly secured variable "char *a" which is initially annotated as H. It passes through a function named sanitization which also represented as a state named "void sanitization(char *a)". This function makes the high secured variable as low by following the policy language. After passing these function the variable "char *a" is annotated with L and sanitized. Now it can be passed to other function or release information from the system. In good_path() region there exist sanitization method. So the method make the variable "char *a" from H(high) to L(Low). So the method validateFunction gets the parameter as L(low). So there would be no bug in good_path() region. But in bad_path() region there is no sanitization method. That's why validateFunction get's the parameter as H(high). So there should be bug triggered.

5.4. Checkers in Static Analysis Engine

Static Code Analysis (also known as Source Code Analysis) is usually performed as part of a Code Review (also known as white-box testing) and is carried out at the Implementation phase of a Security Development Lifecycle (SDL). Static Code Analysis commonly refers to the running of Static Code Analysis tools that attempt to highlight possible vulnerabilities within 'static' (non-running) source code by using techniques such as Taint Analysis and Data Flow Analysis.

A lot of the defects that are present in a program are not visible to the compiler. Static code analysis is a way to find bugs and reduce the defects in a software application.In the 1970's, Stephan Johnson, then at Bell Laboratories, wrote Lint, a tool to examine C source programs that had compiled without errors and to find bugs that had escaped detection.There are many ways to detect and reduce the number of bugs in a program. For instance in Java, JUnit is a very useful tool for writing tests. Overtime research proved that analyzing the code (especially code reviews) is the best way to eliminate bugs. This is not always possible because it is very hard to train people and get them together to study and identify problems in programs. Furthermore it is almost impossible to use code inspections on project's complete code base. Most errors fall into known categories, as people tend to fall into the same traps repeatedly. Therefore a static analyzer or checker is a program written to analyze other programs for flaws.

Static code analysis, also commonly called white-box testing, looks at applications in non-runtime environment. This method of security testing has distinct advantages in that it can evaluate both web and non-web applications and through advanced modeling, can detect flaws in the software's inputs and outputs that cannot be seen through dynamic web scanning alone. In the past this technique required source code which is not only unpractical as source code often is unavailable but also insufficient.

Some tools are starting to move into the Integrated Development Environment (IDE). For the types of problems that can be detected during the software development phase itself, this is a powerful phase within the development lifecycle to employ such tools, as it provides immediate feedback to the developer on issues they might be introducing into the code during code development itself. This immediate feedback is very useful as compared to finding vulnerabilities much later in the development cycle.The UK Defense Standard 00-55 requires that Static Code Analysis be used on all safety related software in defense equipment.

False Positives: A static code analysis tool will often produce false positive results where the tool reports a possible vulnerability that in fact is not. This often occurs because the tool cannot be sure of the integrity and security of data as it flows through the application from input to output.

False positive results might be reported when analyzing an application that interacts with closed source components or external systems because without the source code it is impossible to trace the flow of data in the external system and hence ensure the integrity and security of the data.

False Negatives: The use of static code analysis tools can also result in false negative results where vulnerabilities result but the tool does not report them. This might occur if a new vulnerability is discovered in an external component or if the analysis tool has no knowledge of the runtime environment and whether it is configured securely.

Some tools for static code analysis are given below:

- In .NET programming language: .NET Compiler Platform, CodeIt.Right, CodeRush, FxCop, NDepend, StyleCop etc.
- In case of C, C++: BLAST, Cppcheck, Clang, Coccinelle, Fluctuat etc.
- For Java: Checkstyle, FindBugs, IntelliJ IDEA, Sonargraph, Soot, Squale etc.
- For JavaScript: JSLint, JSHint.
- In Objective-C: Clang.

Types of Static Code Checkers: According to the [9] static code analyzers come in different flavors, analyzers that work directly on the program source code and analyzers that work on the compiled byte code. Each type has its own advantages. When analyzing directly the program code, the source code static analyzer checks directly the source program code written by the programmer. Compilers optimize code, and the resulting byte code might not mirror the source.On the other hand, working on byte code is much faster, which is very important on projects that contain for instance more than one hundred thousand lines of code.

Even though each code checker works differently, most of them share some basic traits.Static checkers read the program and build an abstract representation of it that they are using for matching the error patterns they recognize. On the other hand static checkers perform some kind of data-flow analysis, trying to infer the possible values that variables might have at certain points in the program. When a program accepts input, there is a possibility that this input can be used to subvert the system. Buffer overflows have been over time hacker's favorite inroad. Nowadays SQL injection seems to have taken the top place for program sore spots. Therefore data-flow analysis is very important for vulnerability checking. It is essential to be able to trace the input flow from users through the program. There is no code checker that can ever assure developers that a program is correct. Such guarantees aren't possible.In fact, no code checker is complete or sound. A complete code checker would find all errors, while a sound code checker would report only real errors and no false positives.

The percentage of false to true positives indicates if a code checker is suitable for different programs. It is recommended for developers to examine a checker's behavior in their work before committing to it in the whole project.

Human fallibility is somewhat predictable, but code checkers cannot take all possible bugs into account. Most of the checkers gives programmers the option to define their own rules for the checker to use. In this way, if developers are certain or can predict that they are particularly prone to some kinds of bugs, they can guard against them by writing custom bug detectors.

In this research through static code analysis engine named smtcodan was used as the checker to detect the bug in source code files. After the generation of C/C++ code files, those files are included inside the eclipse C/C++ project. This project should be included in the running eclipse instance. Then after running the project as C/C++ code analysis the bug will be detected if any bugs are available inside the project. Figure 5.4 depicts the bug reports obtained by running the checkers in parallel on the generated programs. Here for example the sanitization scenario has selected to analyze. At first the generated files are included inside the project named "SantizationScenario". Then the project was run as C/C++ code analysis. Here the circled numbers in Figure 5.4 indicate the following: number 1 indicates the analyzed programs (generated programs), number 2 presents bug reports generated by one of the checker for the analyzed programs and number 3 shows the bug location (line number 69) in source code file scenario3.c by clicking on the second bug report (Missing Sanitization function Bug Detected). The bug reports depicted in Figure 5.4 with number 2 , confirm that bug is successfully detected and no false positives and false negatives were generated. Beside the number 2 there also exist more information like file name, line number of the bug location and file path.

When bug found inside the source file, if user click the bug location then easily can view the bug message.Below in the figure 5.4 it is presented how the message will appear.In the scenario3.c file after clicking the bug symbol in line number 69 the message is appeared as like this sanitization function missing bug detected.

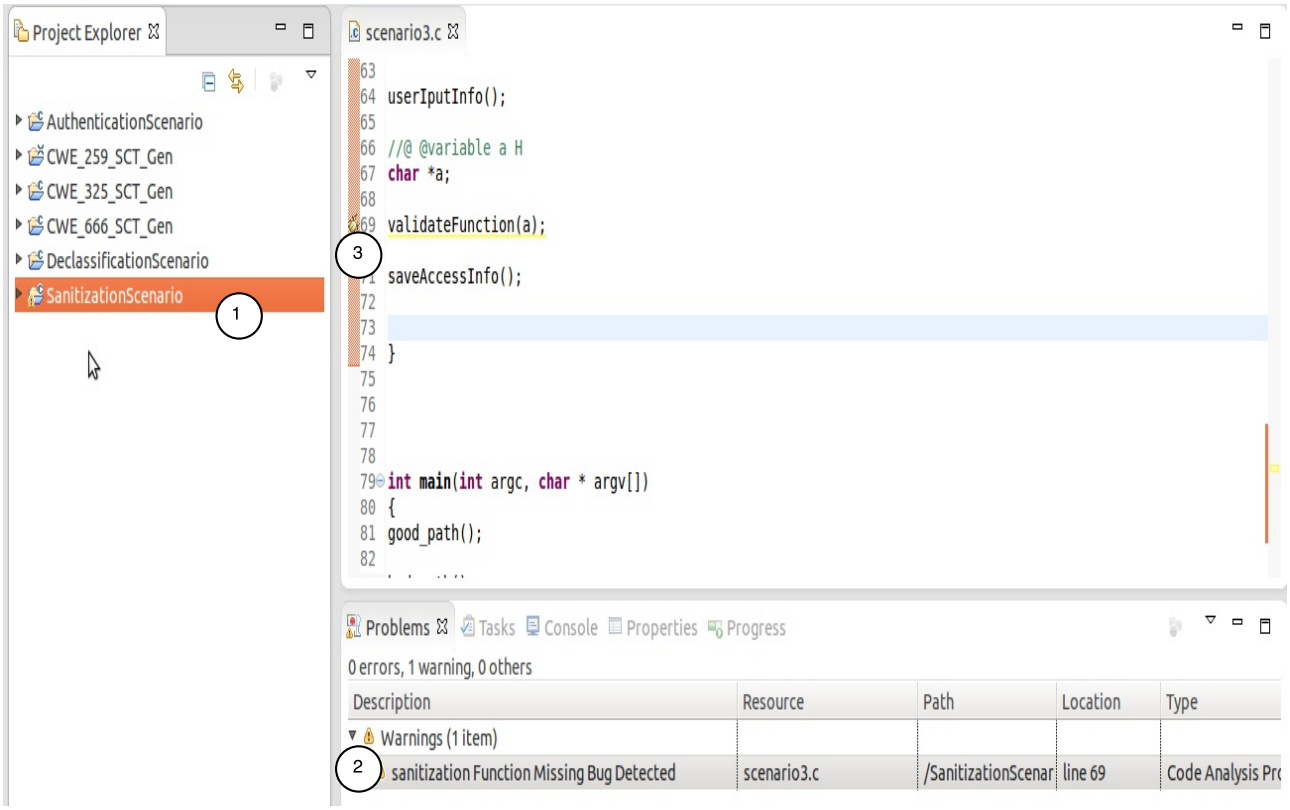


Figure 5.4.: Bug Reports in Checker

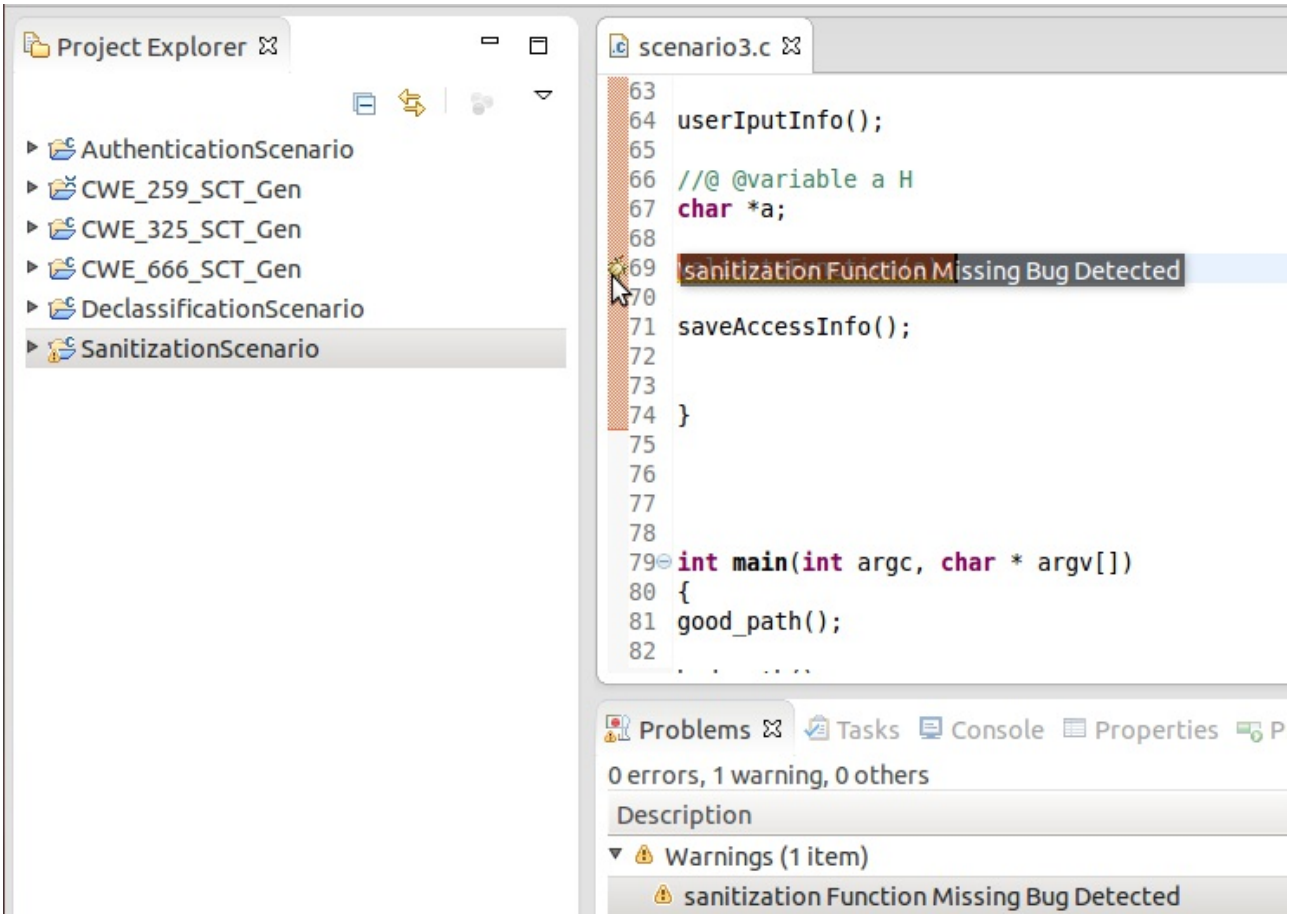


Figure 5.5.: Bug Reports in Checker with Message

6. Related Work

There are many annotation languages proposed until now for extending the C type system [22], [28], [59], [60], [84] to be used during run-time as a new language run-time for PHP and Python [90] to annotate function interfaces [28], [59], [84] to annotate models in order to detect information flow bugs [50] to annotate source code files [70], [71], [83] or to annotate control flows [28], [30], [59].The following annotation languages have made significant impact: Microsoftś SAL annotations [59] helped to detect more than 1000 potential security vulnerabilities in Windows code [7]. In addition, several other annotation languages including FlowCaml [77], Jif [16], Fable [82], AURA [45] and FINE [81] express information flow related concerns.

Saner [83], a novel approach to the evaluation of the sanitization process in web applications. The approach relies on two complementary analysis techniques to identify faulty sanitization procedures. Saner [83] introduce a dynamic analysis technique that is able to reconstruct the code that is responsible for the sanitization of application inputs, and then execute this code on malicious inputs to identify faulty sanitization procedures. By applying it to real-world applications, identified novel vulnerabilities that stem from incorrect or incomplete sanitization.

A simple idea named trusted declassification [41] in which special declassifier functions are specified as part of the global policy. In particular, individual principals declaratively specify which declassifiers they trust so that all information flows implied by the policy can be reasoned about in absence of a particular program. They formalize their approach for a Java like language and prove a modified form of noninterference which they call noninterference modulo trusted methods. They have implemented their approach as an extension to Jif and provide some of their experience using it to build a secure e-mail client.

Using RESIN [91], Web application programmers can prevent a range of problems, from SQL injection and cross-site scripting, to inadvertent password disclosure and missing access control checks. Adding a RESIN assertion to an application requires few changes to the existing application code, and an assertion can reuse existing code and data structures. For instance, 23 lines of code detect and prevent three previously-unknown missing access control vulnerabilities in phpBB, a popular Web forum application. Other assertions comprising tens of lines of code prevent a range of vulnerabilities in Python and PHP applications. A prototype of RESIN incurs a 33% CPU overhead running the HotCRP conference management application.

UMLSec [49] 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.

Darvas, Hähnle, and Sands [23] used a self-compositional approach to prove secure information flow properties for Java CARD programs. They used an interactive approach instead of an automatic approach. Barthe, D'Argenio, and Rezk coined the term "self-composition" in their paper [10]. Their paper is mostly theoretical results on characterizing various secure information flow problems, including non-deterministic and termination-sensitive cases, in a self-compositional framework.

Barthe, D'Argenio, and Rezk in the same paper showed that their selfcompositional framework can handle delimited information release as originally proposed by Sabelfeld and Myers [73]. Li and

Zdancewic's recently proposed relaxed non-interference [54] is equivalent to delimited information release when strengthened with semantic equivalence. Relaxed non-interference is arguably a more natural formulation of information downgrading than delimited information release. This research suggests a promising practical approach of natural formulation of information downgrading.

The detection of information flow errors can be addressed with dynamic analysis techniques [5], [31], [72], static analysis techniques [37], [62], [78], [85], [89] (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 [61]. Also, extended static checking [25] (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 [53] reports about the annotation burden as being very time consuming and disliked by some programming teams.

Security concerns are a major disincentive for use of the cloud, particularly for companies responsible for sensitive data. DIFC [6] is most appropriately integrated into a PaaS cloud model which can be tested by augmenting existing open source implementations such as VMware CloudFoundry and Red Hat OpenShift. DIFC has been used to protect user data integrity and secrecy. In order to apply these techniques to a cloud environment a number of challenges need to be overcome. These include: selecting the most appropriate DIFC model; policy specification, translation, and enforcement; audit logging to demonstrate compliance with legislation and for digital forensics.

There exist several approaches that are focused on the detection of "taint-style" vulnerabilities (such as XSS or SQL injections), which frequently occur in web applications. Huang et al. [43] adapted parts of the techniques used in CQual to develop an intraprocedural analysis for PHP programs. In [44], the same authors present an alternative approach that is based on bounded model checking. Whaley and Lam [86] described an interprocedural, flow-insensitive alias analysis for Java applications. Their analysis is based on binary decision diagrams, and was used by Livshits and Lam [57] for the detection of taint-style vulnerabilities.In [8], their approach is based on Pixy [48, 47], an open source static PHP analyzer that uses taint analysis for detecting XSS vulnerabilities.

A good overview of static analysis approaches applied to security problems is provided in [15]. Simple lexical approaches employed by scanning tools such as ITS4 and RATS use a set of predefined patterns to identify potentially dangerous areas of a program [88]. While a significant improvement on Unix grep, these tools, however, have no knowledge of how data propagates throughout the program and cannot be used to automatically and fully solve taint-style problems. A few projects use path-sensitive analysis to find errors in C and C++ programs [12, 38, 56]. While capable of addressing taint-style problems, these tools rely on an unsound approach to pointers and may therefore miss some errors. The WebSSARI project uses combined unsound static and dynamic analysis in the context of analyzing PHP programs [43]. WebSSARI has successfully been applied to find many SQL injection and cross-site scripting vulnerabilities in PHP code.

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 [79], Adabased SPARK Examiner [13] 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 [29], Flawfinder [87] and Cqual [76] 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 [51] 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 [52]. 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. [39], [40] introduced an UML profile that incorporates a decentralized label model [40] into the UML. It allows the annotation of UML artifacts with Jif [63] 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 [69], while [40] 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. Limitations

The main limitations of this system are given below:

- In UML state chart user can design function calls and statements of C/C++ language.
- Currently user have to model a real life system using region name good_path() and bad_path() or one of them in UML state chart editor. Otherwise code generator will not work.
- Now code generator generates only two files. One is .c file and another one is .h file.
- Code generator works with YAKINDU SCT editor.
- Static analysis engine works only in Ubuntu operating system.
- Code generator, UML statechart modeling developed in windows operating system.
- The simulation is not currently not working in selected scenarios in UML state chart editor.
- Now source, sink, authentication, declassification and santization types of function can be annotated.

8. 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.

Another avenue of future work lies in expanding the policy language. It is currently very simple, but could be more expressive. For example, constraints could be added to indicate negative information flows. Policy analyses could also be used to determine whether separation of duties is maintained between two principals. When integrity is added to policy language, it could be expanded with robustness constraints.It is a general problem in language based security that there is too little experience with security-typed programming to help guide such research as designing the best form of declassification. We hope that our implementation of this mechanism will help to promote more practical experience with declassifiers which will better inform future research.

Functions are used to declassify sensitive data because they are trusted to release information. Our work introduces a security-typed language. We annotate functions with security levels. Functions may be annotated with a high security level; this indicates they are trusted and permits them to serve as a safe mechanism for declassification.

Web applications perform a lot of critical tasks and handle sensitive information. Even though there have been a number of research efforts to identify the use of unvalidated input in web applications, little has been done to characterize how sanitization is actually performed and how effective it is in blocking web-based attacks. In case of desktop applications it is not easy to handle sensitive information. To handle these sensitive information and invalidate insecure input data a good approach to the evaluation of the sanitization process has been developed. Future work will focus on the analysis of type-based validation procedures.

Mechanisms such as access control, encryption, firewalls, digital signatures and antivirus scanning do not address the fundamental problem: tracking the flow of information in computing systems. Run-time monitoring of operating-systems calls is similarly of limited use because information-flow policies are not properties of a single execution; in general, they require monitoring all possible execution paths. On the other hand, there is clear evidence of benefits provided by language-based security mechanisms that build on technology for static analysis and language semantics.The benefits of security-type systems and semantic-based security models and emphasize the compositional nature of both.Type systems are attractive for implementing static security analyses. It is natural to augment type annotations with security labels. Type systems allow for compositional reasoning, which is a necessity for scalability when applied to larger programs.Semantics-based models are suitable for describing end-to-end policies such as noninterference and its extensions. These models allow for a precise formulation of the attacker's view of the system. This view is described as a relation on program behaviors where two behaviors are related if they are not distinguishable by the attacker. Attackers of varying capabilities can be modeled straightforwardly as different attacker views, and correspond to different security properties.A number of further advantages are associated

with both security-type systems and semantics based security. Compositionality is especially valuable in the context of security properties.Compositionality greatly facilitates correctness proofs for program analyses. If the recent progress in language-based techniques for soundly enforcing end-to-end confidentiality policies continues, the approach may soon become an important part of standard security practice.

It is a general problem in language based security that there is too little experience with security-typed programming to help guide such research as designing the best form of declassification. We hope that our implementation of this research mechanism will help to promote more practical experience with declassifiers which will better inform future research.

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. Appendix

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

Bibliography

[1] GCN, http://gcn.com/articles/2009/02/09/ static-vs-dynamic-code-analysis.aspx.

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

[3] Searchwindevelopment Techtarget, http://searchwindevelopment.techtarget.com/definition/static-analysis.

[4] 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.

[5] T. Avgerinos, S.K. Cha, B. L. T. Hao, and D. Brumley. AEG: Automatic Exploit Generation. *Proceedings of the Network and Distributed System Security Symposium (NDSS 11)*, February 2011.

[6] Jean Bacon, David Eyers, Thomas F Pasquier, Jaskirat Singh, Ioannis Papagiannis, and Peter Pietzuch. Information flow control for secure cloud computing. *Network and Service Management, IEEE Transactions on*, 11(1):76–89, 2014.

[7] T. Ball, B. Hackett, S. Lahiri, and S. Qadeer. Annotation-based property checking for systems software. Technical report, Microsoft, May 2008.

[8] Davide Balzarotti, Marco Cova, Vika Felmetsger, Nenad Jovanovic, Engin Kirda, Christopher Kruegel, and Giovanni Vigna. Saner: Composing static and dynamic analysis to validate sanitization in web applications. In *Security and Privacy, 2008. SP 2008. IEEE Symposium on*, pages 387–401. IEEE, 2008.

[9] Alexandru G Bardas. Static code analysis. *Journal of Information Systems & Operations Management*, 4(2):99–107, 2010.

[10] Gilles Barthe, Pedro R D'Argenio, and Tamara Rezk. Secure information flow by self-composition. In *Computer Security Foundations Workshop, 2004. Proceedings. 17th IEEE*, pages 100–114. IEEE, 2004.

[11] Don Box, David Ehnebuske, Gopal Kakivaya, Andrew Layman, Noah Mendelsohn, Henrik Frystyk Nielsen, Satish Thatte, and Dave Winer. Simple object access protocol (soap) 1.1, 2000.

[12] William R Bush, Jonathan D Pincus, and David J Sielaff. A static analyzer for finding dynamic programming errors. *Software-Practice and Experience*, 30(7):775–802, 2000.

[13] R. Chapman and A. Hilton. Enforcing Security and Safety Models with an Information Flow Analysis Tool. *ACM SIGAda*, 24(4), 2004.

[14] Brian Chess and Gary McGraw. Static analysis for security. *IEEE Security & Privacy*, (6):76–79, 2004.

[15] Brian Chess and Gary McGraw. Static analysis for security. *IEEE Security & Privacy*, (6):76–79, 2004.

[16] S. Chong, A. C. Myers, N. Nystrom, L. Zheng, and S. Zdancewic. Jif: Java + information flow, July 2006. Software release.

[17] 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.

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

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

[20] Ellis S Cohen. Information transmission in sequential programs. *Foundations of Secure Computation*, pages 297–335, 1978.

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

[22] J. Condit, M. Harren, Z. R. Anderson, D. Gay, and G. C. Necula. Dependent types for low-level programming. *ESOP*, 2007.

[23] Ádám Darvas, Reiner Hähnle, and David Sands. A theorem proving approach to analysis of secure information flow. In *Security in Pervasive Computing*, pages 193–209. Springer, 2005.

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

[25] D. L. Detlefs, K. R. M. Leino, G. Nelson, and J. B. Saxe. Extended Static Checking . *Compaq SRC Research Report 159*, 1998.

[26] Eclipse. xtend Documentation. Technical report, Eclipse, http://www.eclipse.org/xtend/documentation/.

[27] Eclipse. xText Documentation. Technical report, Eclipse, iTemis, http://www.eclipse.org/Xtext/documentation.html.

[28] D. Evans. Static detection of dynamic memory errors. *PLDI*, 1996.

[29] D. Evans and D. Larochelle. Improving Security Using Extensible Lightweight Static Analysis. *IEEE Software*, Jan/Feb 2002.

[30] D. Evans and D. Larochelle. Splint - Manual, http://www.splint.org/manual/html/sec8.html.

[31] J. S. Fenton. Memoryless subsystems. *Computer Journal*, 17(2):143–147, May 1974.

[32] 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.

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

[34] Pablo Giambiagi and Mads Dam. On the secure implementation of security protocols. In *Programming Languages and Systems*, pages 144–158. Springer, 2003.

[35] Joseph A Goguen and José Meseguer. *Security policies and security models*. IEEE, 1982.

[36] Joseph A Goguen and José Meseguer. Unwinding and inference control. In *Security and Privacy, 1984 IEEE Symposium on*, pages 75–75. IEEE, 1984.

[37] M. Guarnieri, P. El Khoury, and G. Serme. Security vulnerabilities detection and protection using Eclipse. *ECLIPSE-IT 2011, 6th Workshop of the Italian Eclipse Community*, September 2011.

[38] Seth Hallem, Benjamin Chelf, Yichen Xie, and Dawson Engler. *A system and language for building system-specific, static analyses*, volume 37. ACM, 2002.

[39] R. Heldal and F. Hultin. Bridging model-based and language-based security. *Computer Security - ESORICS 2003*, 2808:235–252, 2003.

[40] R. Heldal, S. Schlager, and J. Bende. Supporting Confidentiality in UML: A Profile for the Decentralized Label Model. Technical report, TU Munich Technical Report TUM-I0415, 2004.

[41] 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.

[42] 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.

[43] Yao-Wen Huang, Fang Yu, Christian Hang, Chung-Hung Tsai, Der-Tsai Lee, and Sy-Yen Kuo. Securing web application code by static analysis and runtime protection. In *Proceedings of the 13th international conference on World Wide Web*, pages 40–52. ACM, 2004.

[44] Yao-Wen Huang, Fang Yu, Christian Hang, Chung-Hung Tsai, Der-Tsai Lee, and Sy-Yen Kuo. Verifying web applications using bounded model checking. In *Dependable Systems and Networks, 2004 International Conference on*, pages 199–208. IEEE, 2004.

[45] L. Jia, J. Vaughan, K. Mazurak, J. Zhao, L. Zarko, J. Schorr, and S. Zdancewic. Aura: A programming language for authorization and audit . *ICFP*, 2008.

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

[47] Nenad Jovanovic, Christopher Kruegel, and Engin Kirda. Precise alias analysis for static detection of web application vulnerabilities. In *Proceedings of the 2006 workshop on Programming languages and analysis for security*, pages 27–36. ACM, 2006.

[48] Nenad Jovanovic, Christopher Kruegel, and Engin Kirda Pixy. A static analysis tool for detecting web application vulnerabilities (short paper). In *Proceedings of the 2006 IEEE symposium on Security and Privacy, Washington, DC, IEEE Computer Society*, pages 258–263, 2010.

[49] J. Juerjens. *Secure systems development with UML*. Springer Verlag, 2005.

[50] K. Katkalov, K. Stenzel, M. Borek, and W. Reif. Model-Driven Development of Information

Flow-Secure Systems with IFlow. *ASE Science Journal*, 2(2), 2013.

[51] K. Katkalov, K. Stenzel, M. Borek, and W. Reif. Model-Driven Development of Information Flow-Secure Systems with IFlow. *ASE Science Journal*, 2(2), 2013.

[52] KIT. JOANA (Java Object-sensitive ANAlysis) - Information Flow Control Framework for Java . *KIT*, [online] http://pp.ipd.kit.edu/projects/joana/.

[53] K. Rustan M. Leino. Extended Static Checking: a Ten-Year Pesrsective. *Proceeding Informatics - 10 Years Back. 10 Years Ahead*, pages 157–175, January 2001.

[54] Peng Li and Steve Zdancewic. Downgrading policies and relaxed noninterference. In *ACM SIGPLAN Notices*, volume 40, pages 158–170. ACM, 2005.

[55] Yin Liu and Ana Milanova. Static analysis for inference of explicit information flow. In *Proceedings of the 8th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 50–56. ACM, 2008.

[56] V Benjamin Livshits and Monica S Lam. Tracking pointers with path and context sensitivity for bug detection in c programs. In *ACM SIGSOFT Software Engineering Notes*, volume 28, pages 317–326. ACM, 2003.

[57] V Benjamin Livshits and Monica S Lam. Finding security vulnerabilities in java applications with static analysis. In *Usenix Security*, pages 18–18, 2005.

[58] John McLean. Proving noninterference and functional correctness using traces. Technical report, DTIC Document, 1992.

[59] Microsoft. MSDN run-time library reference - SAL annotations, http://msdn.microsoft.com/en–us/library/ms235402.aspx, 2014.

[60] Sun Microsystems. Lock_Lint - Static data race and ceadlock detection tool for C, http://developers.sun.com/sunstudio/articles/locklint.html.

[61] S. Moore and S. Chong. Static analysis for efficient hybrid information-flow control . *CSF '11 Proceedings of the IEEE 24th Computer Security Foundations Symposium*, pages 146–160, 2011.

[62] A. C. Myers. JFlow: Practical Mostly-Static Information Flow Control. *Proceedings of the 26th ACM Symposium on Principles of Programming Languages (POPL '99)*, January 1999.

[63] A. C. Myers and B. Liskov. A decentralized model for information flow control. *Proceedings of the sixteenth ACM symposium on Operating systems principles, ser. SOSP '97.*, pages 129–142, 1997.

[64] 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.

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

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

[67] 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.

[68] Networkworld. networkworld Documentation. Technical report, Networkworld, http://www.networkworld.com/article/2296774/access-control/seven-strong-authentication-methods.html.

[69] Sören Preibusch. Information flow control for static enforcement of user-defined privacy policies. In *Policies for Distributed Systems and Networks (POLICY), 2011 IEEE International Symposium on*, pages 133–136. IEEE, 2011.

[70] D. S. Rosemblum. Towards a Method of Programming with Assertions. *ACM*, (1), January 1992.

[71] D. S. Rosenblum. A practical approach to programming with assertions. *IEEE Transactions on software engineering*, 21, January 1995.

[72] A. Sabelfeld and A. Russo. From dynamic to static and back: Riding the roller coaster of information-flow control research. *International Conference on Perspectives of System Informatics*, 2009.

[73] Andrei Sabelfeld and Andrew C Myers. A model for delimited information release. In *Software Security-Theories and Systems*, pages 174–191. Springer, 2004.

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

[75] Jerome H Saltzer, David P Reed, and David D Clark. End-to-end arguments in system design. *ACM Transactions on Computer Systems (TOCS)*, 2(4):277–288, 1984.

[76] U. Shankar and et al. Detecting Format-String Vulnerabilities with Type Qualifiers. *10th USENIX Security Symposium*, August 2001.

[77] V. Simonet. *FlowCaml in a nutshell*. In G. Hutton, ed. APPSEM-II, 2003.

[78] V. Simonet. The Flow Caml System: documentation and user's manual . Technical report, INRIA, July 2003.

[79] V. Simonet. The Flow Caml System: documentation and user's manual . Technical report, INRIA, July 2003.

[80] Geoffrey Smith. Principles of secure information flow analysis. In *Malware Detection*, pages 291–307. Springer, 2007.

[81] N. Swamy, J. Chen, and R. Chugh. Enforcing Stateful Authorization and Information Flow Policies in FINE . *In proceedings of ESOP 2010: 19th European Symposium on Programming*, March 2010.

[82] N. Swamy, B. J. Corcoran, and M. Hicks. Fable: A language for enforcing user-defined security policies . *In S&P*, 2008.

[83] L. Tan, Y. Zhou, and Y. Padioleu. aComment:Mining Annotations from Comments and Code to Detect Interrupt Related Concurency Bugs. *ACM 978-1-4503-0445-0/11/05*, May 2011.

[84] L. Torvalds. Sparse - A sematic parser for C, http://www.kernel.org/pub/software/devel/sparse.

[85] D. Volpano, G. Smith, and C. Irvine. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(3):167–187, 1996.

[86] John Whaley and Monica S Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *ACM SIGPLAN Notices*, volume 39, pages 131–144. ACM, 2004.

[87] D. A. Wheeler. *Flawfinder*, http://www.dwheeler.com/flawfinder/.

[88] John Wilander and Mariam Kamkar. A comparison of publicly available tools for static intrusion prevention. 2002.

[89] X. Xiao and et al. Transparent Privacy Control via Static Information Flow Analysis . Technical report, Microsoft, August 2011.

[90] A. Yip, X. Wang, N. Zeldovich, and M. F. Kaashoek. Improving Application Security with Data Flow Assertions. *SOSP'09*, Oct. 2009.

[91] Alexander Yip, Xi Wang, Nickolai Zeldovich, and M Frans Kaashoek. Improving application security with data flow assertions. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 291–304. ACM, 2009.