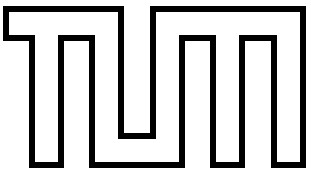


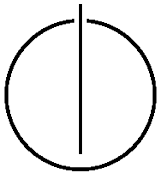
FAKULTÄT FÜR INFORMATIK

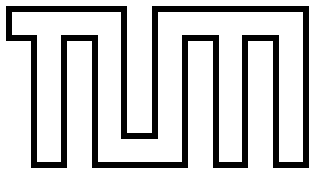
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Master Thesis

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

Md Adnan Rabbi





FAKULTÄT FÜR INFORMATIK

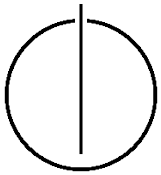
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Master 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:	M.Sc. Paul Muntean
Date:	November 15, 2015



I assure the single handed composition of this master's 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 November 2015 Md Adnan Rabbi

Acknowledgments

I would like to express my deepest appreciation to all those who helped me to complete this thesis. A special gratitude and thanks goes to my supervisor Prof. Dr. Claudia Eckert and my advisor, M.Sc. 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 codes are usually unavailable 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 has been 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 of the selected sample scenarios show 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.

This thesis contributes to the development of a system for semi-automated detection of sanitization, authentication and declassification errors in UML state charts. A light-weight security annotation language is used in order to define information flow constraints regarding authentication, declassification and santization function errors in UML state charts and source code. Annotation language editor is designed as eclipse plug-in 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 three 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's purposes.

CHAPTER 2: LITERATURE REVIEW

This chapter describes the literature and the essential theories needed for our 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 areas of the system.

CHAPTER 6: RELATED WORK

This chapter presents the related work of this research.

CHAPTER 7: LIMITATIONS

This chapter presents the limitations of our approach.

Part III: Conclusion and Future Works

CHAPTER 8: CONCLUSION AND FUTURE WORKS

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 factors in software development. Developing secure system is not an easy task and only adding some information flow restrictions is not sufficient. Now-a-days detection of information flow vulnerabilities in code is particularly one of the challenging issues. 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 when code is not available. In addition, there are no automated checking tools which can reuse the annotated constraints in early stages of software development phase to check information flow errors. It is important to specify security constraints as early as possible in the software development phase in order to avoid the later stage costly repairs or exploitable vulnerabilities.

A solution for tagging sanitization, declassification and authentication in source code is based on libraries which contain all required annotations attached to function declarations. This approach plays an important role mainly for static analysis bug detection techniques where the information is available during program run-time. Detection of information flow vulnerabilities uses dynamic analysis techniques, static analysis techniques and hybrid techniques combining both static and dynamic approaches. The static technique needs 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 [31]. A global policy of noninterference which ensures that high security data will not be observable on low-security channels. This is because noninterference is typically a very strong 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, the 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. Computing systems often deliberately release (or declassify) sensitive information. The main security concern for systems permitting information release is whether this release is safe or not. 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 which actually 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 as a security issue. Code annotations which are added mainly during software development [15] can be

used to provide additional knowledge regarding security issues. 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. Furthermore, 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 added manually by the user in the 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 the later stage of the software project when code development is finished. Tagging and checking for information exposure bugs during the design phase would eliminate software bugs which can be very expensive afterwards. 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 which is 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.

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 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 used program.

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 annotation burden since libraries can be reused, shared and changed by software development teams.

In summary the contribution of this research are:

- A 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 editor designed as eclipse plug-in 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.

- Three checkers are implemented inside our static analysis engine.

- Textual annotations are added inside three scenarios and experimented with our static analysis engine (with three new checkers).

2. Literature Review

The difficulty of strongly protecting confidential information has been known for some time. The research addressing this problem have had relatively little impacts 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 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, can not act as a 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, firewalls permit some communication in both directions [12]; 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 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 secured. In this respect, many security type systems enforcing noninterference are impractical. 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 [74] introduces a new security property, delimited release and 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 [34] 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 [35]. Noninterference can be naturally expressed by semantic models of program execution. This idea goes back to Cohen's early work on strong dependency [19, 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 [18, 63]. 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 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 describes the basics of sanitization, declassification, authentication, information flow vulnerabilities and static code analysis mechanism in software and web applications. Also the purpose and requirements of these three functions: sanitization, declassification and authentication are presented here. At the end of this chapter, the mechanism of detecting information flow errors during design and code are 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 where web applications directly uses external input for critical operations. It is suggested to 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 person, the data he sends to 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.

Cross-site scripting (XSS) attacks (the most common vulnerability) [4] may occur when a web application accepts data originating from a user and sends it to another user's browser without first validating or encoding it. For example, suppose an attacker embeds malicious JavaScript code into his or her profile on a social web site. If the site fails to validate such input, that code may execute in the browser of any other user who visits that profile.Malicious file executions happen when a web

application improperly trusts input files or uses unverified user data in stream functions, thereby allowing hostile content to be executed on the server.

SQL injection vulnerabilities arise in applications where elements of a SQL query originate from an untrusted source. Without precautions, the untrusted data may maliciously alter the query, resulting in information leaks or data modification. The primary means of preventing SQL injection are sanitization and validation, which are typically implemented as parameterized queries and stored procedures.

Suppose a system authenticates users by issuing the following query (listing 2.1) to a SQL database. If the query returns any results, authentication succeeds; otherwise, authentication fails.

Listing 2.1: SQL query for user authentication

```
SELECT * FROM db_user WHERE username='USERNAME' AND
password='PASSWORD'
```

An attacker can substitute arbitrary strings for USERNAME and PASSWORD. In that case, the authentication mechanism can be bypassed by supplying the following USERNAME with an arbitrary password (listing 2.2):

Listing 2.2: SQL query after substitution by attacker

```
'validuser' OR '1'='1
```

The authentication routine dynamically constructs the following query (listing 2.3):

Listing 2.3: Full SQL query after substitution by attacker

```
SELECT * FROM db_user WHERE username='validuser' OR '1'='1' AND
   password='PASSWORD'
```

If validuser is a valid user name, this SELECT statement yields the validuser record in the table. The password is never checked because username='validuser' is true; consequently, the items after the OR are not tested. As long as the components after the OR generate a syntactically correct SQL expression, the attacker is granted the access of validuser. Similarly, an attacker could supply the following string for PASSWORD with an arbitrary username (listing 2.4):

Listing 2.4: SQL query for password with an arbitrary username

```
' OR '1'='1
```

Now producing the following query (listing 2.5):

Listing 2.5: Query after SQL injection

```
SELECT * FROM db_user WHERE username='USERNAME' AND password='' OR '1'='1'
```

'1'='1' always evaluates to true, causing the query to yield every row in the database. In this scenario, the attacker would be authenticated without needing a valid username or password.

Web applications receive remotely supplied (and potentially malicious) user data via HTTP request parameters. In PHP, a variety of security mechanisms exist that can be applied by developers in order to defuse user input for sensitive operations. However, different markup contexts of different operations require different security mechanisms. User input is always received as data of type

string in PHP applications. If this input is processed in a security sensitive operation of the web application, a vulnerability can occur. For example, a dynamically built SQL query with embedded user input may lead to a SQL injection vulnerability. In order to prevent such a critical vulnerability, the user input has to be sanitized or validated beforehand. For this purpose, a security mechanism is applied between the user input (source) and the sensitive operation (sink) such that malicious data cannot reach the sensitive operation. Some sanitization and validation mechanisms have to be applied carefully to the context of the markup. Additionally, a security mechanism can be applied path sensitively. During input sanitization the data is transformed as such that harmful characters are removed or defused. The advantage of this approach is that relevant parts of the data stay intact, while only certain characters are removed or replaced. This way, the application can proceed with the sanitized data without a request for resubmission.

The symbols meta information about sanitization is inferred from PHP operators and built-in functions within the AST. For this purpose, a whitelist of sanitizers is used and complex sanitizers are simulated carefully [22]. The built-in function htmlentities() sanitizes against double quotes and less-than characters by default. Thus, the corresponding sanitization tags for double quoted attributes or HTML elements are added to the argument's data symbol. If the mechanism sanitizes against all types of injection flaws, such as an integer typecast, a universal sanitization tag is added. In order to track multiple levels of escaping and encoding, each data symbol has an escaping and encoding stack that is modified by certain built-in functions, such as addslashes() and urlencode(). String and regular expression replacements are evaluated against a configured list of characters that are required for exploitation of specific markup contexts. If a character is replaced, the according sanitization tag is added to the data symbol.

In many cases, the data is passed directly to a component in a different trusted domain. Data sanitization is the process of ensuring that data conforms to the requirements of the subsystem to which it is passed. Sanitization also involves ensuring that data conforms to security-related requirements regarding leaking or exposure of sensitive data when output across a trust boundary. Sanitization may include the elimination of unwanted characters from the input by means of removing, replacing, encoding or escaping the characters. Sanitization may occur following input (input sanitization) or before the data is passed across a trust boundary (output sanitization). Data sanitization and input validation may coexist and complement each other. Many command interpreters and parsers provide their own sanitization and validation methods. When available, their use is preferred over custom sanitization techniques because custom-developed sanitization can often neglect special cases or hidden complexities in the parser. Another problem with custom sanitization code is that it may not be adequately maintained when new capabilities are added to the command interpreter or parser software.

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 do not sanitize at all, others do so incompletely, leading their owners a false sense of security.

Some basic purposes 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.

Sanitize the input by removing suspicious tags: This is a naive approach because it's almost impossible to cover everything. This regex function removes the obvious code to inject unwanted scripting which is given below (listing 2.6).

Listing 2.6: Regex function to inject unwanted scripting

```
public static String sanitize(String string) {
return string
   .replaceAll("(?i)<script.*?>.*?</script.*?>", "") // case 1: script tags
      are removed.
   .replaceAll("(?i)<.*?javascript:.*?>.*?</.*?>", "") // case 2: javascript
      call are removed.
   .replaceAll("(?i)<.*?\\s+on.*?>.*?</.*?>", ""); // case 3: remove on*
      attributes like onLoad or onClick.
}
```

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 et al. [76] identify four different dimensions of declassification, what is declassified, who is able to declassify, where the declassification occurs and when the declassification takes place.

While the security research community has recognized the importance of the problem, the state-of-the-art in information release comprises a fast growing number of definitions and analyses for different kinds of information release policies over a variety of languages and calculi. The relationship between different definitions of release is often unclear and the relationships that do exist between methods are often inaccurately portrayed. This creates hazardous situations where policies provide only partial assurance that information release mechanisms cannot be compromised.

For example, consider a policy for describing what information is released. This policy stipulates

that at most four digits of a credit card number might be released when a purchase is made (as often needed for logging purposes). This policy specifies what can be released but says nothing about who controls which of the numbers are revealed. Leaving this information unspecified leads to an attack where the attacker launders the entire credit card number by asking to reveal different digits under different purchases.

Myers et al. [67] introduced the decentralized label model, 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, 20, 32, 45] 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 et al. decentralized label model [66] 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 [68].

- 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. [40]

- When: The fourth dimension of declassification is “when” information should be released. The work of Giambiagi et al. [33] 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 et al. 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

Malicious applications targeting financial account information have increased dramatically over the last few years. The number of online applications is growing everyday. The ease of use of the Internet and the growing number of users are making a perfect target for criminals. Attacking thousands of users is achievable with only one click. The methods used by these criminals vary immensely, but they have one thing in common: they are getting more and more sophisticated. With these increasing threats, governments are issuing stronger legislations and companies are realizing that their current systems can not thwart current attacks anymore. To counter these threats, current authentication systems have to be adopted. Not only the criminal side has made advances in the last years. The security industry developed new mechanisms and protection systems to thwart even the most sophisticated attacks.

Before describing the process of the authentication, it is important to explain some terms. Now-a-days, AAA is often used. AAA stands for Authentication, Authorization and Accounting. It is also important to know the differences between those terms:

Authentication: the confirmation that a user is who it is claiming to be.

Authorization: the process to determine whether the user has the authority to issue certain commands.

Accounting: measuring the resources a user consumes during access.

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 is granted.

Now-a-days following authentication, a user must gain authorization for doing certain tasks. After logging into a system, for instance, the user may try to issue commands. The authorization process determines whether the user has the authority to issue such commands. Simply put, authorization is the process of enforcing policies: determining what types or qualities of activities, resources or services a user is permitted. Usually, authorization occurs within the context of authentication. Once users are authenticated, they may be authorized for different types of access or activity.

According to Viega et al. [89] the processes of how to avoid the top ten software security flaws are:

- Earn or give, but never assume, trust.
- Use an authentication mechanism that cannot be bypassed or tampered with.
- Authorize after you authenticate.
- Strictly separate data and control instructions and never process control instructions received from untrusted sources.
- Define an approach that ensures all data are explicitly validated.
- Use cryptography correctly.

- Identify sensitive data and how they should be handled.
- Always consider the users.
- Understand how integrating external components changes your attack surface.
- Be flexible when considering future changes to objects and actors.

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 such as 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 are- 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, assurance of identity of person or originator of data and more.

Challenges in secure Authentication: The security of an application is always a trade off between a high level of security and more usability. The more security is added to an authentication system (pass phrases instead of passwords, multiple authentication tokens), the lower will be the acceptance rate of the users and the usability will decrease. It is a big challenge to find the most secure authentication system which is accepted by the users.

Users always want new applications and features with easy to use interfaces. At the same time they are worried about the increasing dangers. Moreover, new legislations are pushing manufactures and companies to protect the privacy of their clients.The increasing mobility of the users is another important factor. The users want to access their applications not only with their desktop at home, but also in their office, on vacation with their PDA and everywhere with their cell phone. These broad demands from the users create a wide range of attack vectors. Phishing, identity theft, spyware, malware, keyloggers, javascript attacks and generally untrusted consumer platforms all make the traditional means of password based authentication ever more complicated. According to the networkworld [69] now-a-days seven strong authentication methods are- computer recognition software, biometrics, e-mail or SMS, one-time password (OTP) token, peripheral device recognition, scratch-off card etc.

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 later 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].

A variety of static analysis approaches have been proposed to automatically identify security vulnerabilities in PHP applications based on insufficient sanitization and validation. Zheng et al. introduced path-sensitive static analysis for PHP applications with Z3-str [103]. They leveraged a modified version of the Z3 SMT solver that is also capable of analyzing strings. Shar et al. proposed static code attributes for predicting SQLi and XSS vulnerabilities [80, 81]. Yu et al. built an automata-based string analysis tool called STRANGER [101] based on the static code analysis tool Pixy [46]. STRANGER detects security vulnerabilities in PHP applications by computing possible string values using a symbolic automata representation of common string functions, including escaping and replacement functions. Later, they automatically generated sanitization statements for detected vulnerabilities by using regular expression replacements [102]. Balzarotti et al. combined static and dynamic analysis techniques to identify faulty custom sanitization routines [9]. The static analysis component of their tool called Saner extends Pixy and analyzes string modification with automata, while the dynamic component verifies analysis results to reduce false positives.

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.
- It can scan the entire code base.
- It can provide mitigation, reduce 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.
- It does not support all programming languages.
- It can produce false positives and false negatives.
- There are not enough trained personnel to thoroughly conduct static code analysis.
- It can provide a false sense of security that everything is being addressed.
- It is 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. 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.

Information-flow violations [25] comprise the most serious security vulnerabilities in today's web applications. In fact, according to the Open Web Application Security Project (OWASP) [4], they constitute the top six security problems. Automatically detecting such vulnerabilities in real-world web applications may be difficult due to their size and complexity. Manual code inspection is often ineffective for such complex programs and security testing may remain inconclusive due to insufficient coverage.

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 [83].

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 that are expected to be enforced, 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 [77] to the specification of computer security requirements. In a truly secured 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 this research [54], 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 to verify in a practical manner the confidentiality and integrity of sensitive program data. This research 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

The systems development life cycle (SDLC), also referred to as the software/application development life-cycle. It is a term which is used to describe a process for planning, creating, management, testing and deploying an information system. The systems development life-cycle concept applies to a range of hardware and software configurations, as a system can be composed of hardware only, software only or a combination of both.

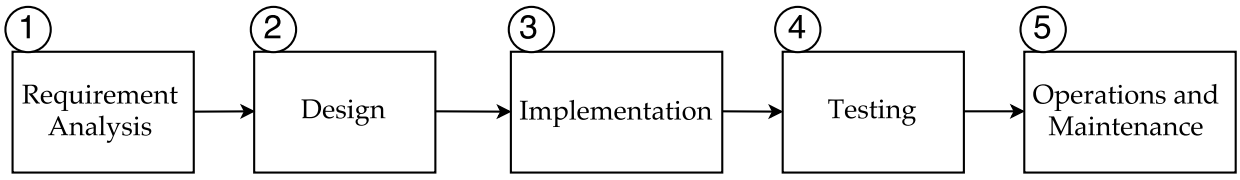


Figure 2.1.: Software development life cycle

Software organization follows SDLC (software development life-cycle) to build a software project. SDLC consists of a detailed plan describing how to develop, maintain, replace and alter or enhance specific software. The life cycle defines a methodology for improving the quality of software and the overall development process. Almost every software development life cycle model has following five phases:

① **Requirement gathering and analysis:** This is the first phase of software development life cycle which is presented in Figure 2.1 as number ①. Sometimes it is also known as feasibility study. In this stage of the software development life cycle, the development team visits the customer and studies their requirements and system. The team investigate the need for possible software automation in the given system/software. By the end of the requirement analysis or feasibility study, the team creates a document that holds the different specific recommendations for the candidate system/software. It also includes the personnel assignments, costs, project schedule, target dates , requirements etc. The requirement gathering process is intensified and focused specially on software. To understand the nature of the software/system to be built, the system engineer or analyst must understand the information domain for the software, as well as required function, behavior, performance and interfacing. The essential purpose of this phase is to find the need and to define the problem that needs to be solved.

The goal of requirement analysis is to determine where the problem is and also it is an attempt to fix the system requirements. This step involves breaking down the system in different pieces to analyze the situation, analyzing project goals, breaking down what needs to be created and attempting to engage users so that definite requirements can be defined perfectly.

② **Design:** This is the sencond phase in the software development life cycle. It is marked as number ② in Figure 2.1. In this phase, the software's overall structure, the software development process and its nuances are defined. In terms of the client/server technology, the number of tiers needed for the package architecture, the database design, the data structure design etc. are all defined in design phase. A software development model is thus created. Analysis and Design are very crucial in the whole development cycle. Any glitch in the design phase could be very expensive to solve in the later stage of the software development. Much care is taken during this phase. The logical system of the product is developed in this phase.

System features in detail, functional hierarchy diagrams, screen layout diagrams, tables of business rules, business process diagrams, pseudo-code and a complete entity-relationship diagram with a full data dictionary of the desired system are described in this stage of software development life cycle. These design elements are intended to describe the system in sufficient detail, such that skilled developers and engineers may develop and deliver the system with minimal additional input design.

③ **Implementation or coding:** Implementation and coding is the third phase of software development life cycle which is denoted as number ③ in Figure 2.1. To develop the software/system the design must be translated into a machine-readable form. The code generation step performs this task. If the design is performed in a detailed manner, code generation can be accomplished without much complication. Programming tools e.g. compilers, interpreters, debuggers etc. are used to generate the code. Different high level programming languages e.g. C, C++, Pascal, Java are used for coding. With respect to the type of application/software , the right programming language is chosen.

④ **Testing:** Testing is the fourth phase of software development life cycle which is marked as number ④ in Figure 2.1. After the code generation, the software program testing begins. Various kind of testing methodologies are available to unravel the bugs that were committed during the previous phases. Different testing tools and methodologies are already available. Some companies build their own testing tools that are tailor made for their own development operations. Below are the types of testing which are so popular to test the software/system:

- Unit testing.
- System testing.
- Integration testing.
- Black-box testing.
- White-box testing.
- Regression testing.
- Automation testing.
- User acceptance testing.
- Software performance testing.
- Path testing.

⑤ **Operations and maintenance:** This is the last phase of software development life cycle which is denoted as number ⑤ in Figure 2.1. The deployment of the system includes changes and enhancements before the decommissioning of the system. Maintaining the system is an important aspect of

SDLC (Software/System Development life Cycle). As key personnel change positions in the organization, new changes will be implemented. There are two approaches to system development- the traditional approach (structured) and object oriented. Information Engineering includes the traditional system approach, which is also called the structured analysis and design technique. The object oriented approach views the information system as a collection of objects that are integrated with each other to make a full and complete information system.

It would be easier for the developer if information flow errors can be detected in the early stage of software development life cycle. In this research a method has been proposed to detect the information flow vulnerabilities in the earlier stage of software development like in designing stage. The process in which way the information flow in designing phase has been detected is described below.

If a step of function call e.g. authentication, sanitization or declassification is missing inside the program then this can lead to software vulnerabilities. In Figure 2.2 the left side picture depicts 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 a missing function of sanitization/declassification/authentication function. Inside the Figure the bug is represented as a red cross sign like ⓧ. It is the buggy path of UML state charts during design stage of software life cycle.

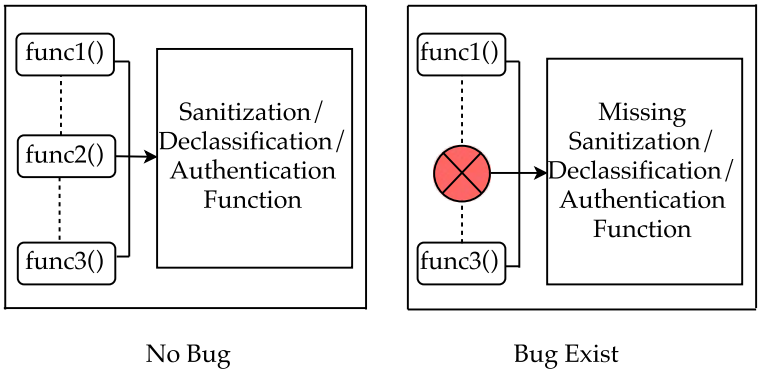


Figure 2.2.: Information flow errors during design

2.7. Detecting Information Flow Errors During Coding

Another way to detect the information flow vulnerabilities in this research is in coding stage of software development life cycle. Here mainly has been detected by static analyzing the code through the engine named smtcodan. How the checker in the static analysis engine has been implemented is given in the implementation chapter of this research. The visual representation of how the bug has been detected is given below.

Figure 2.3 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 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 program-

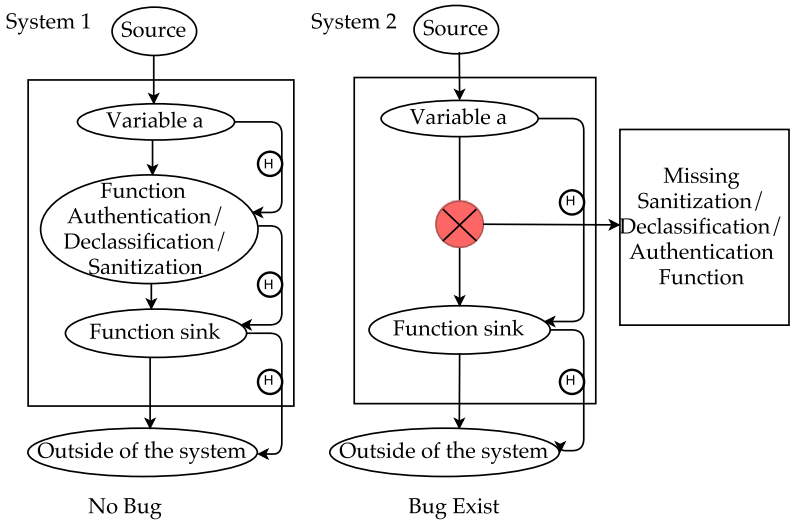


Figure 2.3.: Information flow errors during coding

ming 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 2.3 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 generated. In Figure 2.3 right side system has a bug because it passes a secured/confidential information without passing through authentication/declassifiaction/sanitization function. The bug place is represented as red color cross symbol (Ⓧ) which is showing the missing of those functions. 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.

Part II.

Implementation and Analysis

3. Challenges and Annotation Language Extension

The main goal of this research is 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 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.

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

3.1. Challenges and Ideas

To develop the system eclipse xtext, eclipse xtend and static analysis engine named smtcodan(which is developed in Java to detect C and C++ vulneribilities) are used. For building the source code annotation editor eclipse xtext is used. To model the source code as UML Statechart opensource platform YAKINDU SCT editor is used. Inside YAKINDU sct eclipse xtend is used mainly for genrating source code files (.c and .h files) from statechart. What is xtext, xtend and how it works are described below.

① **Xtext :** Xtext is a framework for development of programming languages and domain specific languages. According to the [98], 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 individual user needs.

Here is an example of xtext code (listing 3.1):

Listing 3.1: Xtext code example

```
grammar org.xtext.example.mydsl.MyDsl with
org.eclipse.xtext.common.Terminals
generate myDsl "http://www.xtext.org/example/mydsl/MyDsl"
Model:
messages+=Message*; //messages containing list of messages
Message:
'Hello' name=ID '!'; // After Hello one can add anything and then '!'
   symbol.
```

The above language allows user to write down a list of messages. The proper input messages which are allowed to write anything like 'Hello User!' or 'Hello World!'.

How Xtext Works: Xtext provides a user with a set of domain-specific languages and modern APIs to describe the different aspects of users programming language. Based on that information it gives

to the user a full implementation of that language running on the JVM. The compiler components of users 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 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, a 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 users 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 user can solve every computer problem, but it might not always be the best way to solve it.

Imagine a user want to remove the core from an apple. User could of course use a Swiss army knife to cut it out and this is reasonable if user have to do it just once or twice. But if user needs 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 user's very own DSL with appropriate IDE support. There is no such limitation as with XML, users are free to define users concrete syntax as users 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, user 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.

② **Xtend :** According to [97], Xtend is a statically-typed programming language which translates to comprehensible Java source code. Syntactically and semantically Xtend has its roots in the Java programming language but improves on many aspects such as- Extension methods, Lambda Expressions, Active Annotations, Operator overloading, Powerful switch expressions, Multiple dispatch, Template expressions etc. Xtend has zero interoperability issues with Java: everything users 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 in xtend. User can define type parameters on methods and classes and pass type arguments to generic types just as users 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 user does not have to think in two worlds. User can invoke Xtend code from Java and vice versa without any surprises or hassles. As a bonus, if user knows Java's type system and are familiar with Java's generic types, he/she already knows 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. 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, user rarely have to write types down because they can be computed from the context.

Consider the following Java variable declaration (listing 3.2):

Listing 3.2: 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 code (listing 3.3):

Listing 3.3: Xtend Code Example

```
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 have 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 have been 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

In this section Table 3.1 contains 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. The tag @variable can be used only inside single line annotations whereas @parameter is used only in multi line annotations. The tags are defined and implemented iteratively based on the work flow presented in Figure 3.1 and by using the eclipse xtext [98] language definition grammar.

For detecting authentication, declassification and sanitization errors new function tags are included like authentication, declassification and sanitization function type. Also for parameter new tag type of parameter is 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 [62] annotation tags.

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

Table 3.1.: Security language annotation tags

3.3. Annotation Language Implementation Process

To implement the annotation language, Eclipse Xtext [98] 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, users are free to define users concrete syntax as users like. It may be as concise and suggestive as possible being a best match for users particular domain. The hard task of reading user model, working with it and writing it back to users 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

that 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 (listing 3.4) is defined as follows:

Listing 3.4: Common terminal Rules in Xtext

```
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 grammar in this research it is required to extend the terminal rule ML_COMMENT and SL_COMMENT. After extending these two rules looks like this (listing 3.5):

Listing 3.5: Singleline and Multiline Comments Rule in Xtext

```
/**
* @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 bodies together with
    single line { comment
   //    | '{' -> '}'            ('\n'|'\r')?
   ;
```

The process depicted in Figure 3.1 is used in order to implement annotation language. The process comprises 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 need to compile it. The result of compilation 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.

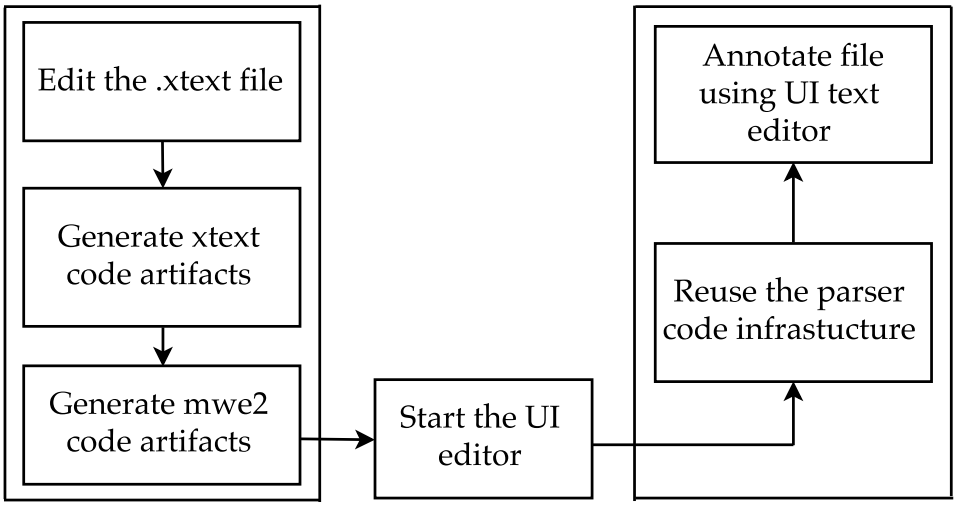


Figure 3.1.: Annotation language design process

4. Implementation

This chapter focuses 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 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. System architecture is similar to one of the building architecture and it's a global model of this system consisting of a structure, properties (of various elements involved), relationships (between various elements), behaviors and dynamics and multiple views of the system (complementary and consistent).

System Architecture is based on nine fundamental principles:

- The objects of the reality are modeled 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, data's 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:

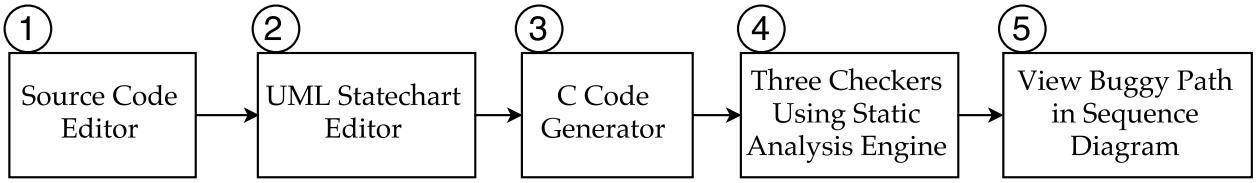


Figure 4.1.: System overview

Figure 4.1 depicts the complete system overview. At first which is presented as number ①, the source code editor is developed using Eclipse Xtext. By this editor one can easily annotate the source code of C/C++. It is possible to annotate C/C++ header files. For information flow vulnerabilities detection in C/C++ code, this annotation technique has been chosen which is easy to extend and backward compatible. This editor has been developed as an Eclipse plug-in. If this plug-in exists in eclipse then a 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 been chosen to model the C/C++ code into state charts to detect the bug during design stage of software development life-cycle which is depicted as number ② in the Figure 4.1. Inside the Yakindu SCT editor the annotation language grammar has also been included using Eclipse Xtext. So, a user can easily annotate the state charts to detect the information flow vulnerabilities. Afterwards the C code generator has been extended inside the Yakindu SCT editor using Eclipse Xtend. This C code generator is represented as number ③ in the Figure 4.1. 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. This three checkers are represented as number ④ phase in the Figure 4.1. 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 been created. Sequence diagram generator to view buggy path is the number ⑤ and last phase of the system. That is the end of complete system architecture of this system.

4.2. The Grammar of Our Annotation Language

The goal of the annotation language is to convey library-specific information to the compiler in a simple declarative manner. While it is clear that more sophisticated specifications could support more sophisticated optimizations, our goal is to show that a few simple annotations can enable many useful optimizations. Simplicity is important because we expect our language users to be library experts who do not necessarily have expertise in compilers or formal specifications.

An annotation is meta-data (e.g., a comment, explanation, presentational markup) attached to text, image, or other data. Often, annotations refer to a specific part of the original data. Markup languages like XML and HTML annotate text in a way that is syntactically distinguishable from that text. They can be used to add information about the desired visual presentation, or machine-readable semantic information. If annotations are to be machine-readable, they must have a well-defined syntax. Annotations also need a well-defined semantics. Quite a few specification languages have been

Category	Parameter Annotation	Description
Input to called function	_In_k	Data is passed to the called function, and is treated as read-only
Input to called function, and output to caller	_Inout_	Usable data is passed into the function and potentially is modified.
Output to caller	_Out_	The caller only provides space for the called function to write to. The called function writes data into that space.
Output of pointer to caller	_Outptr_	Like Output to caller. The value that's returned by the called function is a pointer.

Table 4.1.: Four basic kinds of parameters for Standard Annotation Language (SAL)

defined over the past several decades.

A special case is the Java programming language, where annotations can be used as a special form of syntactic metadata in the source code. Classes, methods, variables, parameters and packages may be annotated. The annotations can be embedded in class files generated by the compiler and may be retained by the Java virtual machine and thus influence the run-time behavior of an application. It is possible to create meta-annotations out of the existing ones in Java.

The ”annotate” function (also known as ”blame” or ”praise”) used in source control systems such as Git, Team Foundation Server and Subversion determines who committed changes to the source code into the repository. This outputs a copy of the source code where each line is annotated with the name of the last contributor to edit that line (and possibly a revision number). This can help establish blame in the event a change caused a malfunction, or identify the author of brilliant code.

For example, Standard Annotation Language or SAL [71] is a meta-language that can help static analysis tools, such as analyze switch in Visual Studio 2005 Team System and Visual Studio 2005 Team Edition for Developers, find bugs including security bugs in your C or C++ code at compile time.

Using SAL is relatively easy. User simply add annotations to user's function prototypes that describe more contextual information about the function being annotated. This can include annotations to function arguments and to function return values. The initial focus of SAL is to annotate functions that manipulate read and write buffers.

These four basic annotations in table 4.1, can be made more explicit in various ways. By default, annotated pointer parameters are assumed to be required, they must be non-NULL for the function to succeed. The most commonly used variation of the basic annotations indicates that a pointer parameter is optional, if it's NULL, the function can still succeed in doing its work. These annotations helps to identify possible uninitialized values and invalid null pointer uses in a formal and accurate manner. Passing NULL to a required parameter might cause a crash, or it might cause a ”failed” error code to be returned. Either way, the function cannot succeed in doing its job.

The main benefit of SAL is that user can find more bugs with just a little bit of upfront work. The process of adding SAL annotations to existing code can also find bugs as the developer questions the assumptions previously made about how the function being annotated works. By this as a developer

adds annotations to a function, he/she must think about how the function works in more detail than simply assuming it was written correctly. This process finds assumption flaws. Any bugs found in SAL annotated functions tend to be real bugs, not false positives, which has the benefit of speedier bug triage and code fixes.

In this research, annotation language is developed using Eclipse xtext [98]. 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 is why an annotation language has been 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). The idea 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 have been used: Italic font for non-terminals, bold typewriter font for literal terminals including keywords.

Ann_Lang ::= HeaderModel*;

H_Model ::= *S_L_Anno*; ;single line comment rule
| *M_L_Anno*; ;multi line comment rule
| *Func_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**; ▷ Newly added annotations to annotate variable types
| **declassified**;
| **sanitized**;

Figure 4.2.: Light-weight annotation language grammar excerpt [62]

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. Inference Rules for Secure Information Flows

The goal is to prevent the information flow from H (high security level, private) variables to L (low security level, public) variables through trust boundaries. The inference rules are implemented inside our static analysis engine which can handle pointers. Considering the following C if statement, if

$$a(L) \leq b(H)$$

then{} else{}, where variable *a* has attached the label L and variable *b* has attached the label H. There could be implicit (the variables inside the *then* or *else* branch do not depend on the values of *a* or *b*) and explicit (the variables inside the *then* or *else* branch depend on the values of *a* or *b*) flows between variables contained in the *then* or *else* as follows: L to L, H to H, L to H and H to L. If a variable labeled H is used afterwards inside a trust boundary then a information flow leakages should be reported and a bug report should be created. This situation marks as a forbidden flow which we try to detect.

(A) (data types)	$\tau ::= \text{H} \mid \text{L} \mid \text{PreStep} \mid \text{PostStep}$
(B) (phrase types)	$\rho ::= \tau \mid \tau\ var \mid \tau\ cmd$

Figure 4.3.: Secure typing system specialized on trust boundaries

Figure 4.3 presents the typing system on which our information flow inference rules, depicted in Figure 4.4, are based on. In the first row of Figure 4.3,(A), we define the following data types: H and L used to attach private and public labels to program variables (High/private and Low/public) and PreStep and PostStep used to attach function call ordering labels to previous and post function calls. Figure 4.3, (B) , presents three types of phrases on which our inference rules are based.

Listing 4.1: Sanitization function behavior to maintain secure information flow

```
//@ @variable a H;
char *a;

/*@ @function sanitization
* @parameter a H sanitized @*/;
sanitization(a);

/*@ @function sink
* @parameter a L @*/;
validateFunction(a);
```

Figure 4.4 depicts secure information flow inference rules which are based on the Denning [24] lattice model and Volpano et al. [91]. We used only two security levels (L and H) which correspond to 0 and 1 whereas one could use multiple levels if required, (e.g., [. . . , –3, –2, –1, 0, 1, 2, 3, . . .]). The expression $\gamma \vdash e : H$, (2), means that expression e has security level H (High). $\gamma \vdash e : \tau\ H$, (3) , means

✓⓪ (INT) $\gamma \vdash n : \mathrm{L}$

✓① (VAR) $\gamma \vdash x : \mathrm{H}\ var \quad \text{if } \gamma(x) = \mathrm{H}\ var$

✓② (R-VAL) $\dfrac{\gamma \vdash e : \mathrm{H}\ var}{\gamma \vdash e : \mathrm{H}}$

✓③ (F-CALL-P) $\dfrac{\gamma \vdash e : \tau\ (\mathrm{H})}{\gamma \vdash e : \tau_r\ (\mathrm{L})}$ ▷ Function: authentication, declassification or sanitization

④ (ASSIGN) $\dfrac{\gamma \vdash e : \mathrm{H}\ var \quad \gamma \vdash e' : \mathrm{H}}{\gamma \vdash e := e' : \mathrm{H}\ cmd}$

⑤ (COMPOSE) $\dfrac{\gamma \vdash c : \mathrm{L}\ cmd \quad \gamma \vdash c' : \mathrm{H}\ cmd}{\gamma \vdash c; c' : \mathrm{H}\ cmd}$

⑥ (IF) $\dfrac{\gamma \vdash e : \mathrm{H} \quad \gamma \vdash c : \mathrm{H}\ cmd \quad \gamma \vdash c' : \mathrm{H}\ cmd}{\gamma \vdash if\ e\ then\ c\ else\ c' : \mathrm{H}\ cmd}$

Figure 4.4.: Typing rules specialized to L, H for secure explicit and implicit information flow(✓implemented)

that if a function call (authentication, declassification or sanitization function) was tagged with the parameter label H and then the label of the parameter is replaced with L after execution (e.g., `char *a` initially annotated with label H, but `sanitization(a)` makes the variable `char *a` as L after execution and passes to other function which is safe (code example presented in listing 4.1)).

The inference rules presented in Figure 4.4 describe how the label(s): ⓪ L is attached to an integer value, ① H is attached to a variable, ② H is passed during a return statement, ③ is used to pass the parameter label H and makes the parameter as L between the parameters of a function call, ④ H is passed during an assignment statement , ⑤ L and H are passed during a composition statement, ⑥ is passed during an `if` statement.

4.4. UML State Chart Editor

UML is a standard language for specifying, visualizing, constructing, and documenting the artifacts of software systems. UML was created by Object Management Group and UML 1.0 specification draft was proposed to the OMG in January 1997. UML provides elements and components to support the requirement of complex systems. UML follows the object oriented concepts and methodology. So object oriented systems are generally modeled using the pictorial language. UML diagrams are drawn from different perspectives like design, implementation, deployment etc. UML can be defined as a modeling language to capture the architectural, behavioral and structural aspects of a system.

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.

Statechart diagram defines the states of a component and these state changes are dynamic in nature. So its specific purpose is to define state changes triggered by events. Events are internal or external factors influencing the system. Statechart diagrams are used to model states and also events operating on the system. When implementing a system it is very important to clarify different states of an object during its life time and statechart diagrams are used for this purpose. When these states and events are identified they are used to model it and these models are used during implementation of the system. The main uses of UML state chart are:

- To model object states of a system.
- To model reactive system. Reactive system consists of reactive objects.
- To identify events responsible for state changes.
- To forward and reverse engineering.

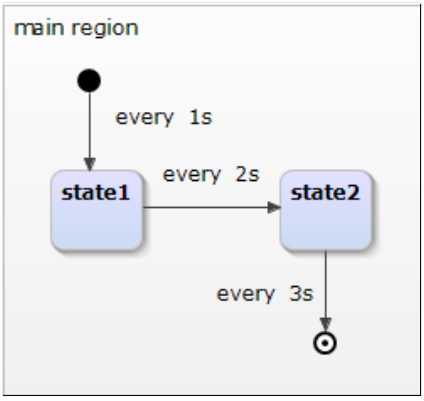


Figure 4.5.: Simple UML state chart

Below are the basic notational elements that can be used to make up a diagram represented in Figure 4.3.

- 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.

Statechart diagram is used to describe the states of different objects in its life cycle. So the emphasis is given on the state changes upon some internal or external events. These states of objects are important to analyze and to implement them accurately. Following are the main purposes of using Statechart diagrams:

- To model dynamic aspect of a system.
- To model life time of a reactive system.

- To describe different states of an object during its life time.
- To define a state machine to model states of an object.

A set of formal representation of UML state chart 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 [5] , 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 4.6.: UML statechart formal representation

In modeling using sets, it is 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 consequence, 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 4.7.: More UML statechart 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 sub states from State to power set of State describes type of a state.

Invariants:

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 4.8.: UML statechart formal representation for more parts

- 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 it has no sub-state.
- 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 been 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 e.g. 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 4.9.: UML statechart formal representation for this research

4.5. Source Code Editor

A source code editor is a text editor. It is a program which is designed specifically for editing source code of computer programs. It may be built into an integrated development environment (IDE) or web browser. To simplify and speed up input of source code such as syntax highlighting, indentation, autocomplete and bracket matching functionality, source code editors features have designed. Source code editors are the most fundamental programming tool. Some well-known source code editors are Gedit, IntelliJ, NetBeans, Notepad++ and more.

For this research, previous source code editor [62] has been 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 has been developed using Eclipse Xtext [98].

As per requirements previous annotation language grammar [62] which was written in xtext language has been extended. Extra annotation have included e.g. “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 e.g. declassified, sanitized and authenticated. New function types are added e.g. 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. The code snippet of extended xtext grammar is given in Appendix A.1.

From the xtext code snippet previous annotation grammar [62] has FunctionAnnotation(line number 113), SingleLineAnnotation(line number 128) 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 which is declared in line number 172 in A.1. The new enumeration type rule has attributes declassified, sanitized and authenticated. Also the rule of enumeration FunctionType(line number 154 in A.1) extended by adding new type of funtion like authentication, declassification and sanitization.

4.6. C Code Generator

C code generator [62] has been 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. The available state chart execution flow functionality has been used 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 4.1 which is given below is representing how the C code generator has developed. The input of the algorithm for code generator is UML statechart. In eclipse xtend [97] function can be declared as “def”. Inside the algorithm 4.1 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 of C programming language is given below-

Listing 4.2: C header file codes with annotation

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

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

Inside the algorithm 4.1 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 are similar to C programming language syntax. Inside the function typesHAnnotationContent at first need to get the function annotation as we placed the function signature and function annotation inside the header file. That's why we declared a method named getFileContent who returns a hashmap which contains annotations and statements e.g. variable declaration and function signatures. By iterating the hashmap need to make a check if current statement is not a variable annotation then we placed the annotation and function signatures inside the C header file.

Function typesCAnnotationContent is responsible to generate the C source code. Inside this function to get all function content there is a method called getFunctionContent which returns a hashmap with all function signatures and annotation. By iterating that hashmap the required function signatures are placed inside the C code file. Those functions that have annotations only signatures and blank function bodies are placed inside the C source code file. Because the annotations of the functions mainly placed inside the header file. Now in the modeling stage the system was designed with two regions. One region name is good_path() and another one is bad_path(). To get the content of good_path() and bad_path(), two methods have created inside the Naming.xtend file named getGoodPathContent() and getBadPathContent(). Those two methods also returned two hashmaps respectively. One hashmap contains the contents of good_path() region and another hashmap contains the contents of bad_path() region. Both hashmaps contains the function signatures, statements of C/C++ language and annotations. According to the design in the modeling stage the required contents are placed inside the body of good_path() and bad_path(). Normally inside the method of good_path() and bad_path() only the variable annotations exist and other statement e.g. function calling, variable declarations etc. The annotations for functions are placed inside the header files of C/C++ language.

Some of the contents of the header file, C file comes from another xtend file named "Naming.xtend". The methods such as getFileContent, getFunctionContent, getGoodPathContent() and getBadPathContent() all are implemented inside the "Naming.xtend" file. The code snippet from C code generator has given in Appendix A.2 and A.3.

Algorithm 4.1 *C code generator*

Input: Statechart
Output: .c and .h files

```
1:  function GENERATETYPESH(sc)                                   ▷ Where sc - statchart
2:      defgenerateFile_1(testModule.h, typesHAnnotationContent(sc))   ▷ def= function declaration
3:      defgenerateFile_2(testModule.c, typesCAnnotationContent(sc))   ▷ C file generator
4:  end function
5:  function TYPESHANNOTATIONCONTENT(sc)                          ▷ Method for header file generator
6:      for s : getFileContent(sc).entrySet do                    ▷ Iterating hashmap
7:          if !s.key.contains('//@@variable') then               ▷ Checking not variables
8:              s.key
9:          else if s.value.contains('(') then                    ▷ Checking functions
10:             void < s.value >;
11:         end if
12:     end for
13: end function
14: function TYPESCANNOTATIONCONTENT(sc)                          ▷ Method for C file generator
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                ▷ Checking except three types of function
18:             void < s.value >
19:         end if
20:     end for
21:     for region : sc.regions do
22:         if region.name.equalsIgnoreCase('bad_path()' then     ▷ Getting the contents of bad path
23:             void < region.name >
24:             for s : getBadPathContent(sc).entrySet do
25:                 if s.key.contains('//@@variable') then        ▷ Checking variable annotations
26:                     s.key
27:                     s.value;
28:                 end if
29:                 if s.value.contains('(') then                 ▷ Checking function declarations
30:                     s.value;
31:                 end if
32:             end for
33:         end if
34:         if region.name.equalsIgnoreCase('good_path()') then   ▷ Getting the contents of good path
35:             void < region.name >
36:             for s : getGoodPathContent(sc).entrySet do        ▷ Get statements and comments
37:                 if s.key.contains('//@@variable') then        ▷ Check for variable and get the comments
38:                     s.key
39:                 end if
40:                 s.value;
41:             end for
42:         end if
43:     end for
44: end function
```

From xtend code snippet in Appendix (A.2), xtend can easily access 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 which is a state and contains first bracket e.g. "(". 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 whose name 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 hashmap 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 contents from good path then traverse the whole good path and get all the required contents. After that this function puts the content into a hashmap. This hashmap also contains the function name, function annotation, variable declaration which has annotations. Then according to the algorithm by iterating through the hashmap generates the required files by putting the contents in proper place.

4.7. Three Checkers in Static Analysis Engine

Static analysis refers to 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 are 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 research, static analysis engine "smtcodan" has been used. Inside the engine, in order 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 bug 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. 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, the 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 pass 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.

4.8. View Buggy Path in UML Sequence Diagram

The Sequence Diagram models the collaboration of objects based on a time sequence. It shows how the objects interact with others in a particular scenario of a use case. A popular use for them is to document the dynamics in an object-oriented system. For each key collaboration, diagrams are created that show how objects interact in various representative scenarios for that collaboration. An important characteristic of a sequence diagram is that time passes from top to bottom : the interaction starts near the top of the diagram and ends at the bottom. Some of the components of sequence diagram [90] are described below:

- **Actor:** An Actor models a type of role played by an entity that interacts with the subject (by exchanging signals and data), but which is external to the subject (in the sense that an instance of an actor is not a part of the instance of its corresponding subject). Actors may represent roles played by human users, external hardware, or other subjects. Note that an actor does not necessarily represent a specific physical entity but merely a particular facet (role) of some entity that is relevant to the specification of its associated use cases. Thus, a single physical instance may play the role of several different actors and, conversely, a given actor may be played by multiple different instances.

- **Call Message:** A message defines a particular communication between Lifelines of an Interaction. Call message is a kind of message that represents an invocation of operation of target lifeline.

- **Create Message:** A message defines a particular communication between Lifelines of an Interaction. Create message is a kind of message that represents the instantiation of (target) lifeline.

- **Destroy Message:** A message defines a particular communication between Lifelines of an Interaction. Destroy message is a kind of message that represents the request of destroying the lifecycle of target lifeline.

- **Duration Message:** A message defines a particular communication between Lifelines of an Interaction. Duration message shows the distance between two time instants for a message invocation.

- **Found Message:** A found message is a message where the receiving event occurrence is known, but there is no (known) sending event occurrence.

- **LifeLine:** A lifeline represents an individual participant in the Interaction.

- **Lost Message:** A lost message is a message where the sending event occurrence is known, but there is no receiving event occurrence. We interpret this to be because the message never reached its destination.

- **Message:** A message defines a particular communication between Lifelines of an Interaction.

- **Return Message:** A message defines a particular communication between Lifelines of an Interaction. Return message is a kind of message that represents the pass of information back to the caller of a corresponded former message.

- **Note:** A note (comment) gives the ability to attach various remarks to elements. A comment carries no semantic force, but may contain information that is useful to a modeler.

- **Send Message:** A message defines a particular communication between Lifelines of an Interaction. Send message is a kind of message that represents the start of execution.

- **Sequence Message:** A message defines a particular communication between Lifelines of an Interaction. Sequence message is a kind of message that represents the need of performing actions in sequence.

- **Frame:** A frame represents an interaction, which is a unit of behavior that focuses on the observable exchange of information between connect able elements.

- **Concurrent:** A concurrent represents a session of concurrent method invocation along an activation. It is placed on top of an activation.

- **Constraint:** A condition or restriction expressed in natural language text or in a machine readable language for the purpose of declaring some of the semantics of an element.

- **Continuation:** A Continuation is a syntactic way to define continuations of different branches of an Alternative CombinedFragment. Continuation is intuitively similar to labels representing intermediate points in a flow of control.

- **Gate:** A Gate is a connection point for relating a Message outside an InteractionFragment with a Message inside the InteractionFragment.

- **Alternative Combined Fragment:** A combined fragment defines an expression of interaction fragments. A combined fragment is defined by an interaction operator and corresponding interaction operands. Through the use of CombinedFragments the user will be able to describe a number of traces in a compact and concise manner.

- **Recursive Message:** A message defines a particular communication between Lifelines of an Interaction. Recursive message is a kind of message that represents the invocation of message of the same lifeline. Its target points to an activation on top of the activation where the message was invoked from.

- **Self Message:** A message defines a particular communication between Lifelines of an Interaction. Self message is a kind of message that represents the invocation of message of the same lifeline.

- **Terminate Message:** A message defines a particular communication between Lifelines of an Interaction. Terminate message is a kind of message that represents the termination of execution.

- **Interaction Use:** An InteractionUse refers to an Interaction. The InteractionUse is a shorthand for copying the contents of the referred Interaction where the InteractionUse is. To be accurate the copying must take into account substituting parameters with arguments and connect the formal gates with the actual ones.

- **Time Constraint:** A TimeConstraint defines a Constraint that refers to a TimeInterval.

- **Uninterpreted Message:** A message defines a particular communication between Lifelines of an Interaction. Uninterpreted message is a kind of message that represents an uninterpreted call.

In this research we used sequence diagram to view the buggy path. In our case, the sending messages are function calls. One function call used to move from one lifeline to another lifeline. We

used statements (with line number and file name) before function call into the lifeline which will help the user to trace the buggy path easily.

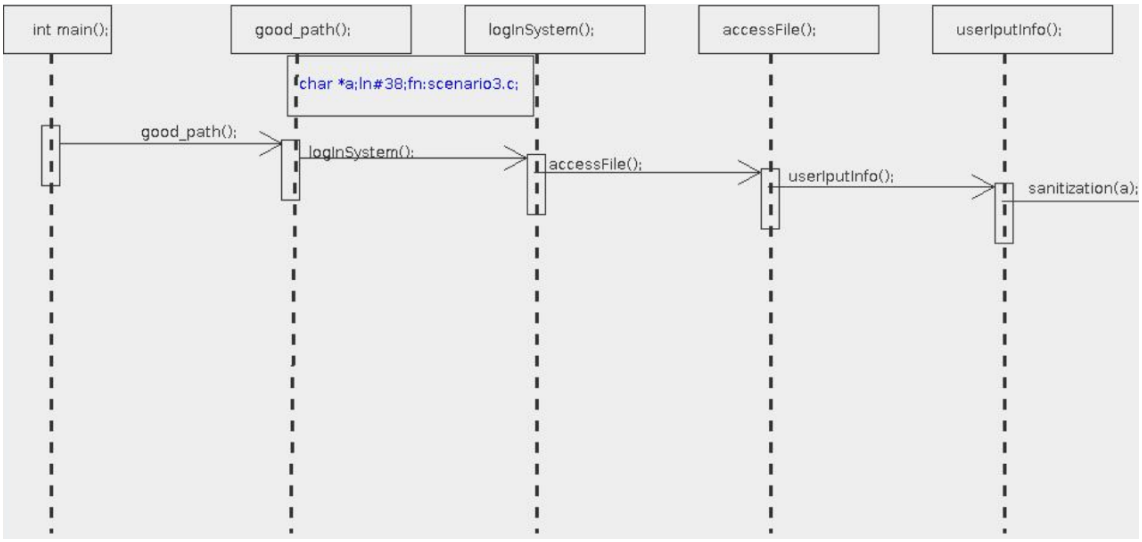


Figure 4.10.: Error trace path in UML sequence diagram

Through the static analysis engine a 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 of programming language (e.g. 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 lifeline. To switch one lifeline to another a function call required. Like in the Figure 4.8 from function call int main() there is a transition to move from int main() to good_path(). Above the transition there also exists 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 lifeline 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 has given in Figure 4.8.

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 it is required to declare object for each of these (Buffered-Image, 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 it is required 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 is 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.

Algorithm 4.2 *Sequence diagram generator*

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

1: ***function*** DRAWSEQUENCEDIAGRAM($ArrayList < IASTNode > statementsList$)
2: $Frame(object)$ ▷ *Frame object initialization*
3: $BufferedImage(object)$ ▷ *BufferedImage object initialization*
4: $Graphics2D(object)$ ▷ *Graphics2D object initialization*
5: $Panel(object)$ ▷ *Panel object initialization*
6: $MyCanvasDraw(object)$ ▷ *MyCanvasDraw object initialization*
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: $add(ln)$ ▷ *ln=line number*
13: $add(fn)$ ▷ *fn=file name*
14: $while(!statementsList.get(j).getRawSignature().toString().contains("("))$
15: $mcd.buggyPathList.add(allStatements);$ ▷ *Where allStatements - statements of C*
16: ***end if***
17: ***end if***
18: ***end for***
19: $set < -ScrollPane(property)$ ▷ *Setting ScrollPane property*
20: $mcd.paint(graphics2D)$ ▷ *Where mcd = MyCanvasDraw object*
21: $topPanel.add(mcd)$ ▷ *Where topPanel = Panel object*
22: $menuBar < -MenuBar$ ▷ *Adding menu bar*
23: $menuFile < -Menu$
24: $menuFileExit < -MenuItem$ ▷ *Adding exit menu*
25: $menuSaveAs < -MenuItem$ ▷ *Adding save as menu*
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 are selected in C language. Then according to the system these three scenarios are modeled in YAKINDU SCT editor. After this source code files are 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 the user has to be authorized by the database administrator or from someone who is responsible to do the authorization. User creation inside a database now-a-days is normally done by 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 (listing 5.1) the method createUser is used to create a DBAccess object for a database management application.

Listing 5.1: Java Code Example Create User Method for Database Access

```
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 who is 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 (listing 5.2) 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.

Listing 5.2: Java Code Example for Authentication Scenario

```
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;
}
```

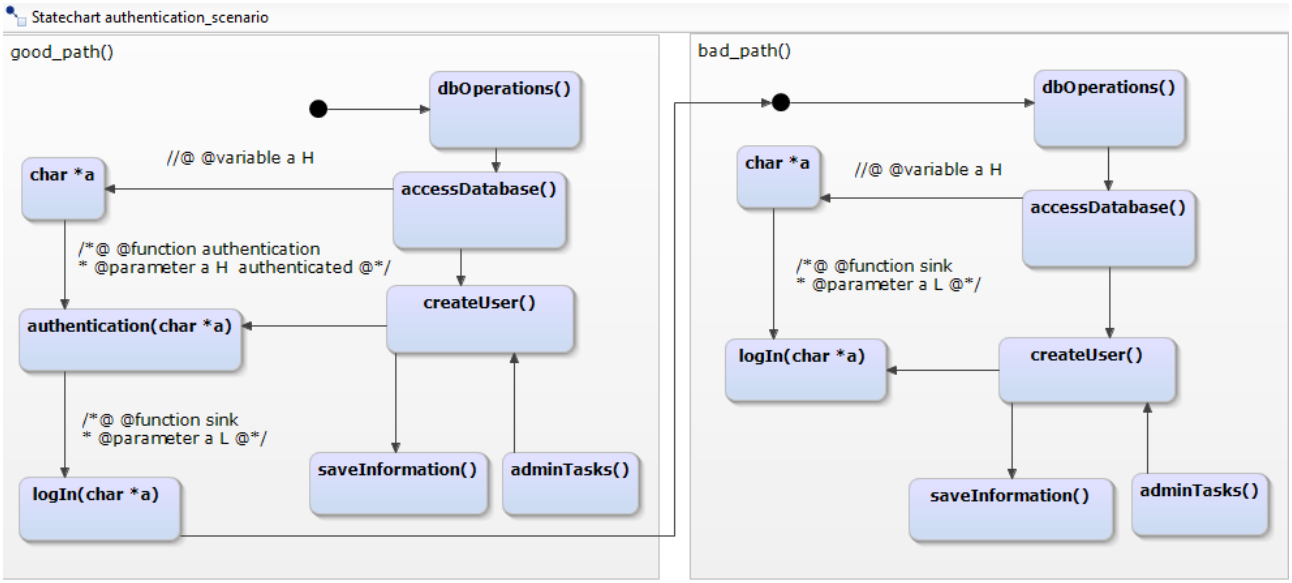


Figure 5.1.: UML statechart modeling for authentication scenario

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 does not have some secured policy like validation, encryption, decryption or authentication methodology then hacker may easily break the application and receive the confidential information or data from the database server. 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 authentication(char *a)". Annotation of this state "void authentication(char *a)" is "/*@ @function authentication * @parameter a H authenticated @*/" which is attached with the transition from "char *a" to "void authentication(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 systems or release information to the authenticated user. While moving from "void authentication(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 that one of the parameter is low. If the parameter does not come through the authentication function then it remains H(High). Then a bug should be triggered. Here in this scenario inside the bad_path() region a bug should be triggered because there is no authentication function exists. 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 strong 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 a part of a global policy, making reasoning about the security implications of a policy more difficult. 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.

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 *A* sends 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.

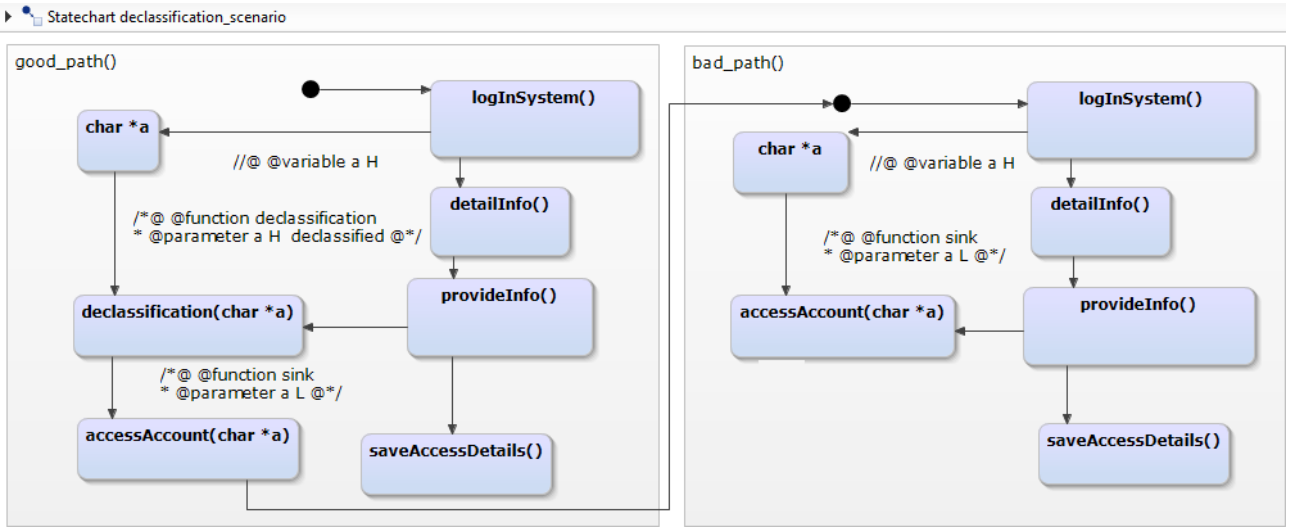


Figure 5.2.: UML statechart modeling for declassification scenario

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 does not have some secured policy e.g. encryption, decryption or declassification methodology then hacker may easily break the application and receive the confidential information or data. 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 state chart 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. Here in

case of bad_path() region there is no declassification method that is 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 is why bug should be triggered here. But for the good_path() region there is no bug because it has the declassification function and the variable “char *a” passes through declassification function. The variable “char *a” becomes H(high) to L(low).

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 exists. Then through static analysis engine named ”smtcodan” the code is analyzed and the information flow vulnerabilities are detected if they exist inside the generated files.

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 to keep data integrity and ensures availability as needed. Sql injection and XSS attacks are common attacks now-a-days. To prevent this kind of attacks it is necessary 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 subjected to the first variant of OS command injection. Example (listing 5.3) language is in PHP.

Listing 5.3: PHP example to list the content of users directory

```
$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 (listing 5.4) such as:

Listing 5.4: Arbitary OS command

```
;rm -rf /
```

Then that would produce a result like this (listing 5.5)-

Listing 5.5: Result after adding arbitary OS command

```
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 has been given for PHP language. In C language here it has selected that user "A" wants to access some file from server computer. 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 is why the user input should pass through a sanitization method to sanitize the input parameter.

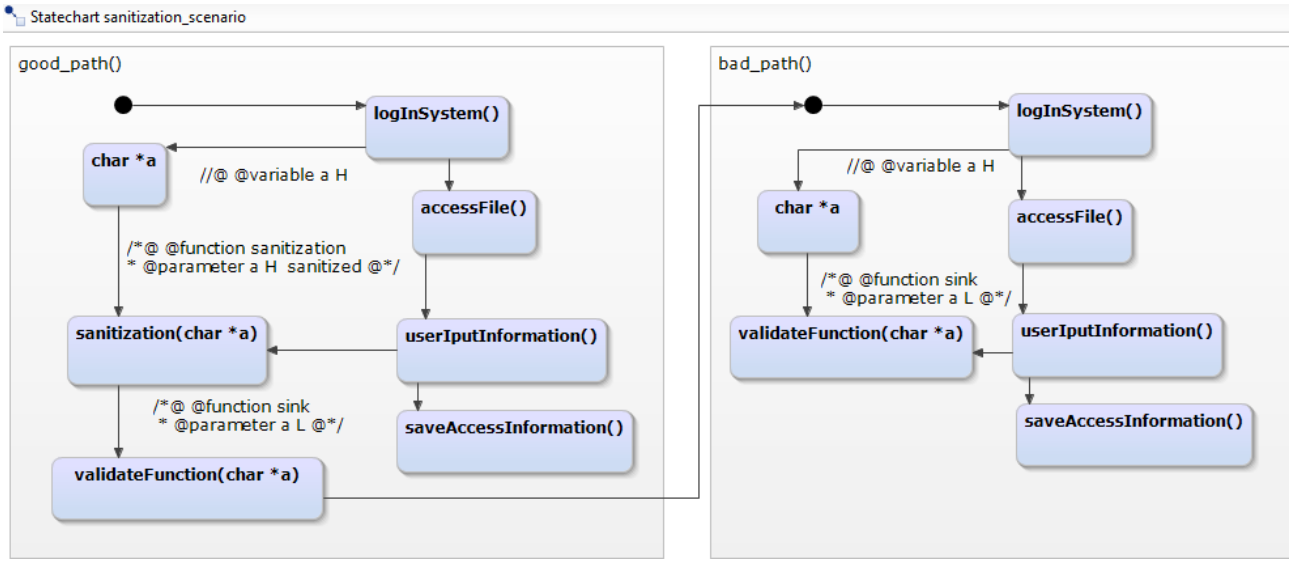


Figure 5.3.: UML statechart modeling for sanitization scenario

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 the file name. Then he/she sends request to access his/her desired file. The server has it's 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 does not have some secured policy e.g. 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 is why validateFunction gets the parameter as H(high). So the bug should be triggered there.

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.

There are many good reasons to use static code analysis in a project, one of them is thorough analysis of your code, without executing them. Static analysis scans all code. If there are vulnerabilities in

the distant corners of your application, which are not even used, then also static analysis has a higher probability of finding those vulnerabilities. Second benefit of using static code analysis is- one can define project specific rules, and they will be ensured to follow without any manual intervention. If any team member forgets to follow those rules, they will be highlighted by static code analyzer like fortify or find-bugs. Third major benefit of static code analysis is that they can find the bug early in development cycle, which means less cost to fix them. All these advantage of static code analyser can be best utilized only if they are part of build process.

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 impractical 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.

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 Alexandru G Bardas [10], 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 are not 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.

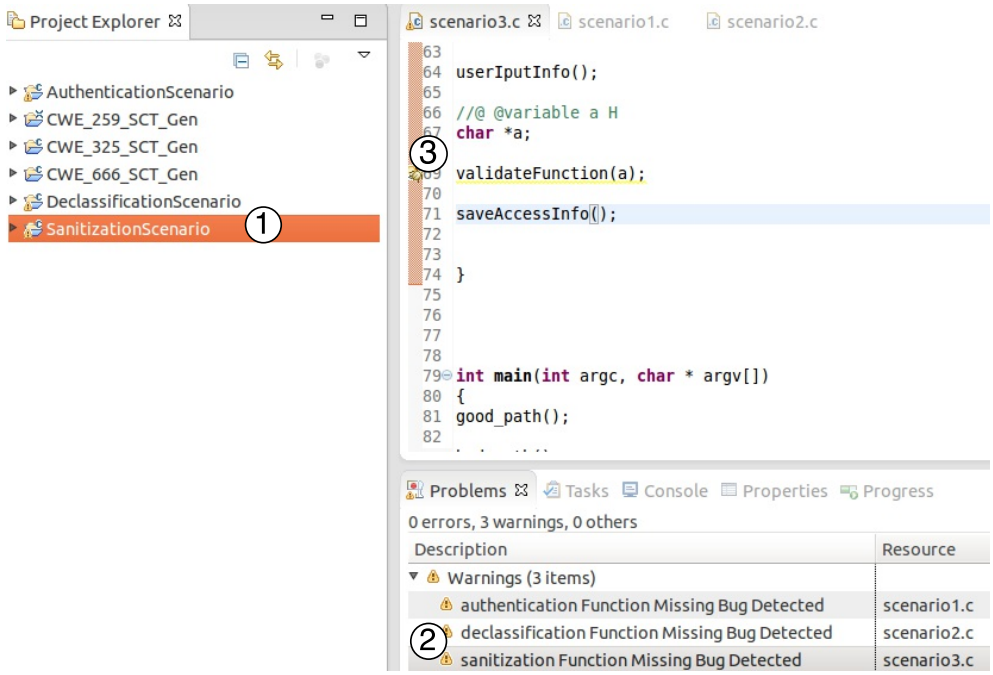


Figure 5.4.: Bug reports in checker

In this research, static code analysis engine named smtcodan has been 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 has been 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. 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 exists more information such as file name, line number of the bug location and file path.

When a bug is found inside the source file, if user click the bug location then they can easily view the bug message. In Figure 5.5 it is presented how the message will appear. In the scenario3.c file, after clicking the bug symbol in line number 69 the message will be appeared as like this - ”sanitization Function Missing Bug Detected”.

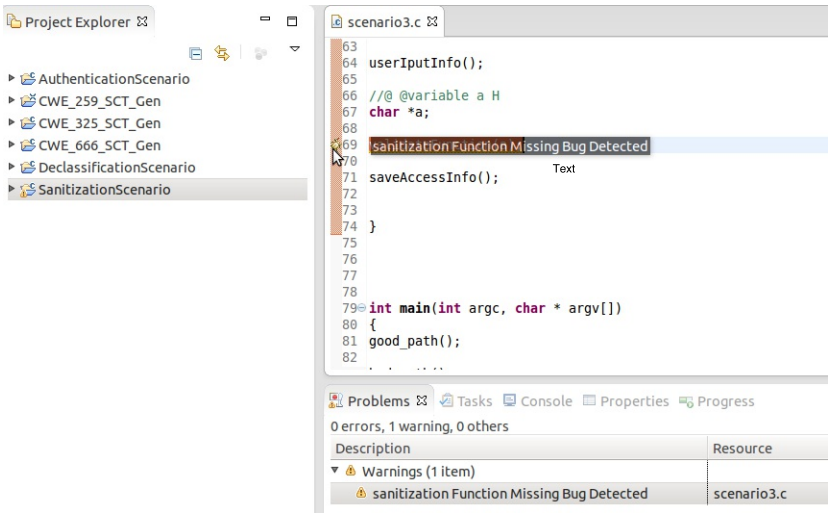


Figure 5.5.: Bug reports in checker with message

5.5. Error Trace in UML Sequence Diagram

To trace the buggy path here in this research sequence digram have used. Function calls, variable declaration and other statements of C/C++ language have included inside the sequence diagram. How we developed it that was described in implementation chapter.

After the bug detection in static analysis engine a buggy path can be found as a list of string if information flow bug exist in the source code files. To generate the sequence diagram a new class is added inside the smtcodan engine. The class name is SequenceDiagramGenerator. Inside the list which is returned from the smtcodan static analysis engine there exist function calls, separate statements e.g. if statements, switch-case statements, variable declaration, assignment of variables etc. of programming language (e.g., C/C++). Then to view the path using java a sequence diagram is generated through the class of SequenceDiagramGenerator. Inside the sequence diagram all function calls are included inside rectangular box attached to the head of timeline/lifeline. To move from one lifeline to another a function call required. Like in the Figure 5.6 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 lifeline 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.

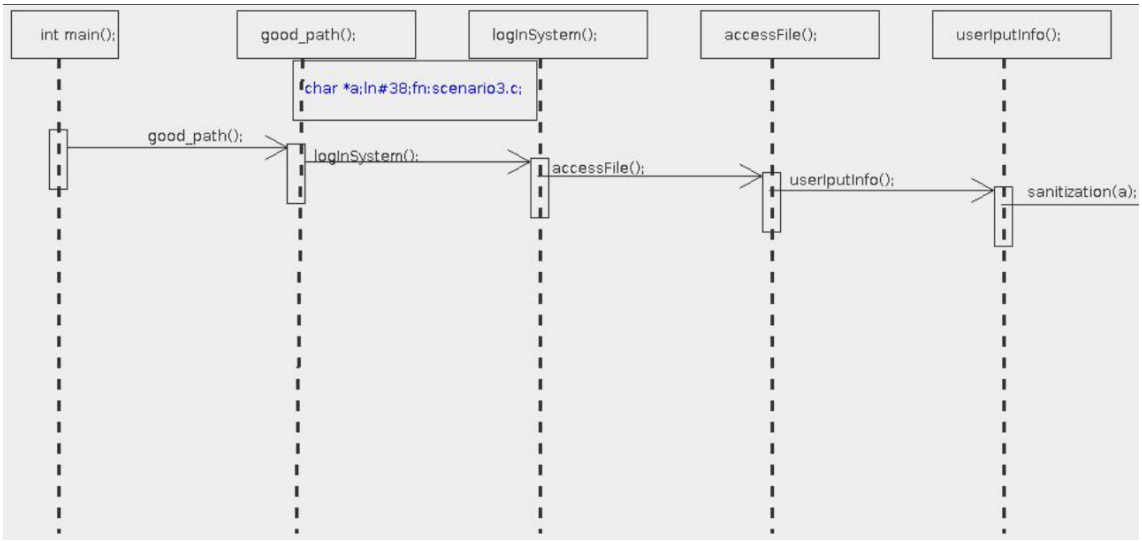


Figure 5.6.: Sequence diagram representation of buggy path

6. Related Works

There are many annotation languages proposed until now for extending the C type system [21], [27, 71, 59, 87] to be used during run-time as a new language run-time for PHP and Python [99] to annotate function interfaces [27, 71, 87] to annotate models in order to detect information flow bugs [50] to annotate source code files [72, 73, 86] or to annotate control flows [27, 29, 71]. The following annotation languages have made significant impact: Microsoftś SAL annotations [71] helped to detect more than 1000 potential security vulnerabilities in Windows code [8]. In addition, several other annotation languages including Jif [18], AURA [44], FlowCaml [82], FINE [84] and Fable [85] express information flow related concerns.

Saner [86], 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 [86] 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 [100], 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.

Static code analysis is a way to find bugs and reduce the defects in a software application. In the 1970's, Stephan Johnson 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. 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.

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.

TAJ [88], an approach to taint analysis suitable for industrial applications. An experimental evaluation indicates that the hybrid algorithm TAJ uses for slice construction is an attractive compromise between context-sensitive and context-insensitive thin slicing. TAJ is able to perform effective taint analysis in a limited budget, improving performance without significantly degrading accuracy.

Darvas et al. [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 et al. coined the term "self-composition" in their paper [11]. 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.

Wassermann et al. [60] string-analysis algorithm to syntactically isolate tainted substrings from untainted substrings in PHP applications. They labeled non-terminals in a Context-Free Grammar (CFG) with annotations reflecting taintedness and untaintedness. Their expensive, yet elegant mechanism is applied to detect both SQL injection and XSS vulnerabilities.

McCamant et al. [57] take a quantitative approach in information flow: instead of using taint analysis, they cast information- flow security to a network-flow-capacity problem, and describe a dynamic technique for measuring the amount of secret data that leaks to public observers.

Barthe et al. [11] showed that their self compositional framework can handle delimited information release as originally proposed by Sabelfeld et al. [74]. Li et al. recently proposed relaxed non-interference [53] 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 [6, 30, 75], static analysis techniques [36, 63, 91, 96] (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 [26] (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 [52] 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 [7] 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. [42] adapted parts of the techniques used in CQual to develop an intraprocedural analysis for PHP programs. In [43], the same authors presented an alternative approach that is based on bounded model

checking. Whaley et al. [93] described an interprocedural, flow-insensitive alias analysis for Java applications. Their analysis is based on binary decision diagrams and was used by Livshits et al. [56] for the detection of taint-style vulnerabilities. In [9], their approach is based on Pixy [47, 48], an open source static PHP analyzer that uses taint analysis for detecting XSS vulnerabilities.

Weinberger et al. empirically studied present sanitization approaches against XSS in web application frameworks [92]. They analyzed the availability of sanitization approaches for different HTML markup contexts for five PHP frameworks. Furthermore, eight PHP applications were studied for the usage of various markup contexts. A templating framework was proposed by Samuel et al. that uses type qualifiers to automate context-sensitive XSS sanitization [78].

A good overview of static analysis approaches applied to security problems are provided in [16]. 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 [95]. 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 [13, 37, 55]. 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 [42]. 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 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 [82], Adabased SPARK Examiner [14] 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 [28], Flawfinder [94] and Cqual [79] 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 [50] 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 [51]. 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. [38, 39] introduced an UML profile that incorporates a decentralized label model into the UML. It allows the annotation of UML artifacts with Jif [65] 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 [70], while [39] notes that the actual automatic Jif code generation is still a 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:

- **Function calls and statements are possible to model in state chart editor:** Now in UML state chart the user can model function calls and statements e.g. variable declaration of C/C++ language. But it is not possible to model switch-case statements, loops etc. In future it may be included.

- **Region names are fixed:** To model the source code user have to include some regions as per the characteristics of state chart. Inside the region the user can add states, transition, composite states, final state, initial state etc. But according to this research now user can declare two regions. 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. With these two region user can model the source code of a real system as a UML state chart.

- **Generator generates two files:** After modeling the source code as UML state charts need to generate the code from the model. The generator was built with Eclipse xtend. Generator included inside YAKINDU SCT Editor.Now code generator generates only two files. One is .c file and another one is .h file.

- **Generator included inside YAKINDU SCT editor:** As an open source tool here for this research we chose YAKINDU SCT Editor to model the source code into state charts. Inside the YAKINDU SCT Editor code generator also exist for C and C++ language. We developed the source code generator according to the requirements inside Myc package for C code generation. Now code generator works with YAKINDU SCT editor.

- **Bindings with operating system:** Code generator, UML statechart modeling developed in windows operating system. Both of these are now working only in windows operating system.

- **Simulation not working:** The simulation for statemachine is working for normal state chart. But the simulation of UML statechart is not currently working in selected scenarios in state chart editor.

- **Fixed function names:** For this research we had to add some more type of functions e.g., authentication, declassification and santization to annotate the UML state chart as well as source code. These three types of function have been included inside the annotation language grammar. Now source, sink, authentication, declassification and santization types of function can be annotated. So, user's can now be able to annotate only these type of functions not more than that.

Part III.

Conclusion and Future Work

8. Conclusion and Future Work

In the conclusion and future work we will present the conclusion and possible future work of this research.

8.1. Conclusion

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 is evaluated on some sample programs and showed that this approach is applicable to real life scenarios.

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. 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.

Last but not least, it is a system which is usable for specifying information flow security constraints which can be used in the design and coding phase in order to detect information flow bugs.

8.2. Future Work

In future this system can be extended for source code editor as a pop-up window based proposal editor 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.

Currently code generator, UML statechart modeling developed only for windows operating system. In future code generator, state chart editor can be implemented for other operating systems. 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 and sanitizers.

Appendix

A. Appendix

A.1. Appendix A: Source Code Editor with Annotation Language

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

/**
  [xText Grammar description. Logger]

  [Other notes: used for adding annotations to .h, .hh, .hhh, .hxx, .c, .cpp, .C and .Cpp files"]
  This grammar is intended to be used for annotating the above mentioned header files
**/
generate myDsl "http://www.xtext.org/example/mydsl/MyDsl"

/**
 * @AnnotationLanguage: top root node of the annotation language
 */
AnnotationLanguage:
   element+= HeaderModel*
;

/**
 * @SingleLineAnnotation: entity used for single line annotations
 * @MultilineAnnotation : entity used for multi line annotations
 * @MethodHeader  : entity used for recognizing anykind of C/C++ headers
 * @AttributeDefinition : entity used for recognizing anykind of variable definition
 */
HeaderModel:
  headers+= SingleLineAnnotation
        | MultilineAnnotation
        | MethodHeader
        | AttributeDefinition
;

/**
 * @AttributeDefinition : entity used for recognizing anykind of statement which begins with the simbol #
 */
AttributeDefinition:
  {AttributeDefinition}(attribute_def+=(SYMBOLS)?(' '?)*extension+=KeyWord (extension_2+=ExpressionAttribute))('\n'|'\r')*
;

/**
 * @ExpressionAttribute : atribute of the AttributeDefinition
 */
ExpressionAttribute returns Ref:
   EntityRef(({Expression.ref=current} (symbols_attr+=SYMBOLS) tail=EntityRef)*)
;
```

```
44
45  /**
46   * @MethodHeader :this recognizes anykind of method headers
47   */
48  MethodHeader returns MethodHeader :
49     {MethodHeader}(
50       ((name0=SYMBOLS)?(' '?'*'* ' '* ID ' '?)*
51         name1=SYMBOLS exp+=Expression name2=SYMBOLS)
52       ((name3=SYMBOLS)?(ID?)(name4=SYMBOLS)?(ID?)(name5=SYMBOLS)?(ID?))
53         name6=SYMBOLS?
54     ) ('\n' | '\r')?
55  ;
56
57  /**
58   * @Expression :used for recognizing expresions inside a MethodHeader
59   *            :it contains one recursion on the current
60   */
61  Expression returns Ref:
62     EntityRef(({Expression.ref=current} (symbols+=SYMBOLS) tail=EntityRef)*)
63  ;
64
65  /**
66   * @EntityRef :@Expression contains @EntityRef, this is a list of entitys
67   */
68  EntityRef returns Ref:
69     {EntityRef} (entity+=SpecialExpression)*
70  ;
71  /**
72   * @IDSpace :contains a left recursion on the currrent
73   *        :used for identityfing expressions with a space in front
74   */
75  IDSpace:
76    EntityRef ({IDSpace.left=current} (' ')* right=SpecialExpression)*
77  ;
78
79  /**
80   * @SpecialExpression :expressions containing stars
81   */
82  SpecialExpression:
83    {Entity}(rules+=
84      ID
85      | '**'((name0=SYMBOLS)?)(ID)?
86      | name1=SYMBOLS(name2=SYMBOLS)?(name3=SYMBOLS)?(name4=SYMBOLS)?(ID)?
87      | INT
88    )
89  ;
90
91  /**
92   * @SpaceID :used for recognizing spaces followed be ID
93   */
94  SpaceID:
95    {SpaceID}(expr+= (' ')* ID?)*
96  ;
97
98  /**
99   * @MultilineAnnotation :used for adding multiline annotations
100  */
101 MultilineAnnotation returns MultilineAnnotation:
102   {MultilineAnnotation}(
103             rule+= ('/*@ ')? ('* ')? functionAnnotation=FunctionAnnotation
                 ('\n')? (' @*/')? (name0=SYMBOLS)?
```

```
104                   | ('*') ' ' ' ' ('@*/') (name1=SYMBOLS)? ('\n' | '\r')?
105     )
106   ;
107
108   /**
109    * @FunctionAnnotation :used for function annotations
110    */
111   FunctionAnnotation returns FunctionAnnotation:
112     {FunctionAnnotation}(
113                       result += '@function ' functionType=FunctionType (' ')?
                                        (level =('H'|'L'))? ((name0=SYMBOLS))?
                              ((nameComment=ID))? ('\n' | '\r')?
114                             // supported without space before confidential and
                                    sensitive
115                             | '@parameter ' parameter=ID (name0=SYMBOLS)?
                                    (securityType=SecurityType)?(' ')? (level
                                    =('H'|'L'))? ('True'|'False')?
                                    (variableTyp=VariableType)? ((name1=SYMBOLS))?
                                    ((nameComment=ID))? ('\n' | '\r')?
116                             //for annotating pre and post functions
117                             | '@preStep ' preStep=ID (name0=SYMBOLS)? (' ')?
                                              (level =('H'|'L'))?
                                    ((name2=SYMBOLS))? ((nameComment=ID))? ('\n' |
                                    '\r')?
118                             | '@postStep ' postStep=ID (name0=SYMBOLS)? (' ')?
                                             (level =('H'|'L'))? ((name3=SYMBOLS))?
                                    ((nameComment=ID))? ('\n' | '\r')?
119                             | '@return'                              (' ')?
                                                    (level =('H'|'L'))?
                                           ('\n' | '\r')?
120     )
121   ;
122
123   /**
124    * @SingleLineAnnotation :used for adding single line annotations
125    */
126   SingleLineAnnotation returns SingleLineAnnotation:
127     {SingleLineAnnotation}(
128                       result+= '//@ @function ' functionType=FunctionType (' ')?
                              (level =('H'|'L'))? ((name0=SYMBOLS))?
                              ((nameComment=ID))? ('\n' | '\r')*
129                             // supported without space before confidential and
                                    sensitive
130                             | '//@ @parameter ' parameter=ID
                                    (securityType=SecurityType)? (' ')? (level
                                    =('H'|'L'))? ('True'|'False')?
                                    (variableTyp=VariableType)? ((nameComment=ID))?
                                    ('\n' | '\r')?
131                             | '//@ @variable ' variable=ID
                                    (securityType=SecurityType)? (' ')? (level
                                    =('H'|'L'))? ('True'|'False')? ((nameComment=ID))?
                                    ('\n' | '\r')?
132                             //for annotating pre and post functions
133                             | '//@ @preStep ' preStep=ID (' ')?
                                    (level =('H'|'L'))? ((name2=SYMBOLS))?
                                    ((nameComment=ID))? ('\n' | '\r')*
134                             | '//@ @postStep ' postStep=ID (' ')?
                                    (level =('H'|'L'))? ((name3=SYMBOLS))?
                                    ((nameComment=ID))? ('\n' | '\r')*
135     )
136   ;
```

```
137
138 /**
139  * @AnnotationType :annotation types
140  *              :annotations can addres whole functions or parameters of a
         function
141  */
142 enum AnnotationType:
143               function
144              |  parameter
145              | preStep
146              | postStep
147 ;
148
149 /**
150  * @FunctionType :annotaions types for functions
151  */
152 enum FunctionType: declassification
153               | sanitization
154               | authentication
155               | sink
156               | source
157               | trust_boundary
158 ;
159
160 /**
161  * @SecurityType :annotations types for parameters
162  */
163 enum SecurityType: confidential
164              | sensitive
165 ;
166
167 /**
168  * @VariableSituation :annotaions types for function parameters
169  */
170 enum VariableType: declassified
171               | sanitized
172               | authenticated
173 ;
174
175 /**
176  * @KeyWord :list of C/C++ keywords
177  */
178 KeyWord returns KeyWord:
179   {KeyWord}(
180        rule=
181             '__BEGIN_DECLS'
182             |'__BEGIN_NAMESPACE_STD'
183             |'__BEGIN_NAMESPACE_C99'
184             |'__END_DECLS'
185             |'__END_NAMESPACE_STD'
186             |'__END_NAMESPACE_C99'
187             |'__USING_NAMESPACE_STD'
188             |'define'
189             |'ifndef'
190             |'undef'
191             |'ifdef'
192             |'if'
193             |'include'
194             |'include_next'
195             |'pragma'
196             |'else'
```

```
197             |'elif'
198             |'error'
199             |'typedef'
200             |'class'
201             |'endif'
202             |'source'
203   )
204 ;
205
206 /**
207  * @SYMBOLS :all available C/C++ symbols
208  */
209 SYMBOLS:
210 {SYMBOLS}(symbols+=
211     ','
212   | '.'
213   | '..'
214   | '...'
215   | '

216 ' // it is in math.h
217   | ';'
218   | ' '
219   | '*'
220   | '* '
221   | '['
222   | ']'
223   | '\n'
224   | '('
225   | ')'
226   | '>>'
227   | '<<'
228   | '>'
229   | '<'
230   | '^'
231   | '+'
232   | '-'
233   | '/'
234   | BackSlash
235   | '%'
236   | '|'
237   | '->'
238   | '<-'
239   | '='
240   | '?'
241   | '!'
242   | DoubleQuote
243   | SingleQuote
244   | ':'
245   | '&'
246   | '~'
247   | '#'
248   | CURLY_OPEN
249   | CURLY_CLOSE
250   | INT
251   | name0=KeyWord //used to bypass the reserved xText keyword source, source
        can be used a function call in C/C++ headers or source files
252  )
253  ;
254
255 /**
```

```
256  * @StructDefinition :used for identifying structs
257  *                    :this is future work. This can be used in the future
258  *                    :when {} is removed as multiline comment and the body of
259  *                    :the struct will be available
260  */
261 StructDefinition:
262   'typedef ' ID (name0=SYMBOLS) name1=ID CURLY_OPEN
263     attr+=ID*
264   CURLY_CLOSE (name2=SYMBOLS) name3=ID (name4=SYMBOLS)?
265 ;
266
267 /**
268  * @SingleQuote :declaration of ' and avoinding overriding the terminal STRING
269  */
270 SingleQuote:
271    MY_STRING
272  ;
273
274 /**
275  * @DoubleQuote :declaration of " and avoinding overriding the terminal STRING
276  */
277 DoubleQuote:
278   STRING | DOUBLE_DQ_STRING
279  ;
280
281 /**
282  * @BackSlash :declaration of \ and avoinding overriding the terminal STRING
283  */
284 BackSlash:
285   STRING | MY_BACKSLASH
286  ;
287
288 /**
289  * @MY_BACKSLASH :double backslash
290  */
291  terminal MY_BACKSLASH: '\\' ;
292
293  /**
294  * @DQ_STRING :double quote declaration
295  */
296  terminal DOUBLE_DQ_STRING : "'\"' ~('\"')* '\"'";
297
298  /**
299  * @SQ_STRING :single quote declaration
300  */
301  terminal DOUBLE_SQ_STRING : "'\'' ~('\'')* '\''";
302
303  /**
304  * @CURLY_OPEN :open curly bracket declaration
305  */
306  terminal CURLY_OPEN: '{';
307
308  /**
309  * @CURLY_CLOSE :close curly bracket declaration
310  */
311  terminal CURLY_CLOSE: '}';
312
313  /**
314  * @MY_STRING :modified string terminals, it allows anykind of symbols inside
       the single and double quotations \
315  */
```

```
terminal MY_STRING :
       '"' ( '\\' | !('\\'|'"') )* '"' |
       "'" ( '\\' | !('\\'|"'") )* "'"
    ;

 /**
 * @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')?
 ;
```

A.2. Appendix B: C Code Generator (Part 1)

```
package Myc

import org.yakindu.sct.model.sexec.ExecutionFlow
import org.yakindu.sct.model.sgraph.Statechart
import org.yakindu.sct.model.sgraph.Choice
import org.eclipse.xtext.generator.IFileSystemAccess
import com.google.inject.Inject
import org.yakindu.sct.model.sgen.GeneratorEntry
import org.yakindu.sct.generator.core.impl.SimpleResourceFileSystemAccess
import org.eclipse.core.resources.ResourcesPlugin
import org.eclipse.core.runtime.Path
import org.eclipse.emf.ecore.EClass

class Types {

  @Inject extension Naming
  @Inject extension GenmodelEntries
  var String variableName
// var fileContent = <String, String>newHashMap

  def generateTypesH(ExecutionFlow flow, Statechart sc, IFileSystemAccess fsa,
     GeneratorEntry entry) {

    if (fsa instanceof SimpleResourceFileSystemAccess &&
      !exists(flow.testModule.h, fsa as SimpleResourceFileSystemAccess)) {
      fsa.generateFile(flow.testModule.h,
          flow.typesHAnnotationContent(sc,entry))
      fsa.generateFile(flow.testModule.c,
          flow.typesCAnnotationContent(sc,entry))
    }
```

```
28    }
29
30  def typesCAnnotationContent(ExecutionFlow flow, Statechart sc ,GeneratorEntry
        entry)'''
31
32  #import "flow.testModule.h"
33  #pragma comment(lib, "advapi32")
34
35  #define HASH_INPUT "ABCDEFG123456" /* INCIDENTAL: Hardcoded crypto */
36  #define PAYLOAD "plaintext"
37
38  FOR s: getFunctionContent(sc).entrySet
39  // extern void s.value {}
40    IF(!s.value.contains('authentication')&&(!s.value.contains('declassification'))
41    &&(!s.value.contains('sanitization')))
42              void s.value {}
43    ENDIF
44  ENDFOR
45
46  FOR region : sc.regions
47
48    IF ( region.name.equalsIgnoreCase('bad_path()'))
49      void region.name{
50        FOR s: getBadPathContent(sc).entrySet
51          IF s.key.contains('//@ @variable')
52          s.key
53          s.value;
54          ENDIF
55          IF s.value.contains('(')
56            s.value;
57          ENDIF
58        ENDFOR
59      }
60    ENDIF
61
62    IF ( region.name.equalsIgnoreCase('good_path()'))
63      void region.name{
64        FOR s: getGoodPathContent(sc).entrySet
65          IF s.key.contains('//@ @variable')
66            s.key;
67          ENDIF
68          s.value;
69        ENDFOR
70      }
71    ENDIF
72  ENDFOR
73
74  int main(int argc, char * argv[])
75  {
76    FOR region : sc.regions
77       IF region.name.contains('good_path()')||region.name.contains('bad_path()')
78         region.name;
79      ENDIF
80    ENDFOR
81   return 0;
82  }
83
84  def typesHAnnotationContent(ExecutionFlow flow, Statechart sc ,GeneratorEntry
        entry)'''
85      FOR s: getFileContent(sc).entrySet
86                IF s.value.contains('(') && s.key.contains('@')
```

```
87                s.key;
88                void s.value;
89              ENDIF
90      ENDFOR
91
92  def protected exists(String filename, SimpleResourceFileSystemAccess fsa) {
93      val uri = fsa.getURI(filename);
94      val file = ResourcesPlugin.getWorkspace().getRoot()
95             .getFile(new Path(uri.toPlatformString(true)));
96      return file.exists;
97    }
98  }
```

A.3. Appendix C: C Code Generator (Part 2)

```
1   package Myc
2
3   import com.google.inject.Inject
4   import org.eclipse.emf.ecore.EObject
5   import org.yakindu.sct.generator.core.types.ICodegenTypeSystemAccess
6   import org.yakindu.sct.model.sexec.ExecutionFlow
7   import org.yakindu.sct.model.sexec.Step
8   import org.yakindu.sct.model.sexec.naming.INamingService
9   import org.yakindu.sct.model.sgraph.Event
10  import org.yakindu.sct.model.sgraph.Scope
11  import org.yakindu.sct.model.sgraph.State
12  import org.yakindu.sct.model.sgraph.Choice
13  import org.yakindu.sct.model.sexec.ExecutionChoice
14  import org.yakindu.sct.model.stext.naming.StextNameProvider
15  import org.yakindu.sct.model.stext.stext.EventDefinition
16  import org.yakindu.sct.model.stext.stext.InterfaceScope
17  import org.yakindu.sct.model.stext.stext.InternalScope
18  import org.yakindu.sct.model.stext.stext.OperationDefinition
19  import org.yakindu.sct.model.stext.stext.VariableDefinition
20  import org.yakindu.sct.model.sgen.GeneratorEntry
21  import org.yakindu.sct.model.sexec.TimeEvent
22  import java.util.List
23  import org.yakindu.sct.model.sgraph.Statechart
24  import java.util.HashMap
25  import org.eclipse.emf.ecore.EClass
26  import org.yakindu.sct.model.sexec.transformation.StatechartExtensions
27  import org.yakindu.sct.model.sexec.transformation.StextExtensions
28  import org.yakindu.sct.model.sexec.transformation.SexecExtensions
29  import org.yakindu.sct.model.sexec.SexecFactory
30  import org.yakindu.sct.model.sexec.ExecutionNode
31  import org.yakindu.sct.model.sgraph.Pseudostate
32
33  class Naming {
34
35    @Inject extension Navigation
36    @Inject extension ICodegenTypeSystemAccess
37    @Inject private StextNameProvider provider
38    @Inject extension INamingService
39    @Inject GeneratorEntry entry
40    @Inject extension GenmodelEntries
41
42    def HashMap<String, String> getFileContent(Statechart sc) {
```

```
43      var fileContent = <String, String>newHashMap
44      for( region : sc.regions){
45        for(vertex : region.vertices) {
46             if (!(vertex.name.nullOrEmpty)){
47               for(transition : vertex.incomingTransitions) {
48                 if(transition.specification.nullOrEmpty)
49                   fileContent.put(vertex.name,vertex.name)
50                 if((!transition.specification.contains('//@ @variable')) &&
                       !(transition.specification.nullOrEmpty))
51                   fileContent.put(transition.specification,vertex.name)
52             }
53           }
54         }
55       }
56    return fileContent
57    }
58    //getFunctionContent returns the function content
59    def HashMap<String, String> getFunctionContent(Statechart sc) {
60      var functionContent = <String, String>newHashMap
61      for( region : sc.regions){
62        for(vertex : region.vertices.filter[eClass.name.contentEquals('State')])
              {
63             if ( (vertex.name.contains('(')) && (!(vertex.name.nullOrEmpty))){
64                  functionContent.put(vertex.name,vertex.name)
65           }
66         }
67       }
68    return functionContent
69    }
70    // getBadPathContent returns the content from bad path region
71    def HashMap<String, String> getBadPathContent(Statechart sc) {
72      var badfunctionContent = <String, String>newHashMap
73      var String newName
74      for( region : sc.regions){
75         if(region.name.equalsIgnoreCase('bad_path()')){
76           for(vertex :
                region.vertices.filter[eClass.name.contentEquals('State')]){
77             if(!(vertex.name.contains('(')) && (!(vertex.name.nullOrEmpty))){
78               for(transition : vertex.incomingTransitions) {
79                 badfunctionContent.put(transition.specification,vertex.name)
80                }
81               }
82             if((vertex.name.contains('(')) && (!(vertex.name.nullOrEmpty))){
83                  if ( (vertex.name.contains('char '))){
84                       newName=vertex.name.replaceAll('char *','')
85                       if(newName.contains('*'))
86                          newName=newName.replaceAll('\\*','')
87                      badfunctionContent.put(newName,newName)
88                  }
89                  else
90                    badfunctionContent.put(vertex.name,vertex.name)
91              }
92            }
93         }
94       }
95    return badfunctionContent
96    }
97    // getVariableName returns the exact variable name which required to static
         analysis engine
98    def String getVariableName(Statechart sc){
99      var String variablename
```

```
100     for( region : sc.regions){
101       for(vertex : region.vertices.filter[eClass.name.contentEquals('State')])
              {
102           if (!(vertex.name.contains('(')) && (!(vertex.name.nullOrEmpty))){
103                for(transition : vertex.incomingTransitions) {
104                          variablename= vertex.name.replaceAll('char *','')
105                          if(variablename.contains('*'))
106                            variablename=variablename.replaceAll('\\*','')
107           }
108         }
109       }
110     }
111   return variablename
112   }
113   // getGoodPathContent returns the contents from good path region
114   def HashMap<String, String> getGoodPathContent(Statechart sc) {
115     var goodfunctionContent = <String, String>newHashMap
116     var String newName
117     for( region : sc.regions){
118       if(region.name.equalsIgnoreCase('good_path()')){
119             val choiceState=0;
120             val increment=1;
121
122             for(vertex :
                  region.vertices.filter[eClass.name.contentEquals('Choice')]){
123              val sum=choiceState+increment;
124               for(transition : vertex.incomingTransitions) {
125                 System.out.println("*********"+"if\n"+sum);
126               }
127             }
128           for(vertex :
                region.vertices.filter[eClass.name.contentEquals('State')]) {
129               for(invertex : vertex.parentRegion.vertices.filter
130             [eClass.name.contentEquals('State')])
131               {
132                 if(!(vertex.name.contains('(')) &&
                        (!(vertex.name.nullOrEmpty))){
133                   for(transition : vertex.incomingTransitions) {
134                        goodfunctionContent.put
135                        (transition.specification,vertex.name)
136                   }
137                 }
138               if((vertex.name.contains('(')) &&
                      (!(vertex.name.nullOrEmpty))){
139
140                 if ( (vertex.name.contains('char '))){
141                      newName=vertex.name.replaceAll('char *','')
142                      newName=newName.replaceAll('\\*','')
143                     goodfunctionContent.put(newName,newName)
144                  }
145                  else
146                    goodfunctionContent.put(vertex.name,vertex.name)
147             }
148           }
149         }
150       }
151     }
152    return goodfunctionContent
153   }
154 }
```

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] OWASP. Technical report, http://www.owasp.org.

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

[6] 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.

[7] J. Bacon, D. Eyers, T. F. Pasquier, J. Singh, I. Papagiannis, and P. Pietzuch. Information flow control for secure cloud computing. *IEEE Transactions on Network and Service Management*, 11(1):76–89, 2014.

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

[9] D. Balzarotti, M. Cova, V. Felmetsger, N. Jovanovic, E. Kirda, C. Kruegel, and G. 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.

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

[11] G. Barthe, P. R. D'Argenio, and T. Rezk. Secure information flow by self-composition. In *Computer Security Foundations Workshop, 2004. Proceedings. 17th IEEE*, pages 100–114. IEEE, 2004.

[12] D. Box, D. Ehnebuske, G. Kakivaya, A. Layman, N. Mendelsohn, H. F. Nielsen, S. Thatte, and D. Winer. Simple object access protocol (soap) 1.1, 2000.

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

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

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

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

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

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

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

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

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

[22] J. Dahse and T. Holz. Simulation of built-in php features for precise static code analysis. In *Symposium on Network and Distributed System Security (NDSS)*, 2014.

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

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

[25] D. E. Denning and P. J. Denning. Certification of programs for secure information flow. *Communications of the ACM*, 20(7):504–513, 1977.

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

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

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

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

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

[31] A. Gehani, D. Hanz, J. Rushby, G. Denker, and R. DeLong. On the (f) utility of untrusted data sanitization. In *MILITARY COMMUNICATIONS CONFERENCE, 2011-MILCOM 2011*, pages 1261–1266. IEEE, 2011.

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

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

[34] J. A. Goguen and J. Meseguer. *Security policies and security models*. IEEE, 1982.

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

[36] 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.

[37] S. Hallem, B. Chelf, Y. Xie, and D. Engler. *A system and language for building system-specific, static analyses*, volume 37. ACM, 2002.

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

[39] 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.

[40] B. Hicks, D. King, and P. 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.

[41] B. Hicks, D. King, P. McDaniel, and M. 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] Y. Huang, F. Yu, C. Hang, C. Tsai, D. Lee, and S. 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.

[43] Y. Huang, F. Yu, C. Hang, C. Tsai, D. Lee, and S. Kuo. Verifying web applications using bounded model checking. In *Dependable Systems and Networks, 2004 International Conference on*, pages 199–208. IEEE, 2004.

[44] 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.

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

[46] N. Jovanovic, C. Kruegel, and E. Kirda. Pixy: A static analysis tool for detecting web application vulnerabilities. In *Security and Privacy, 2006 IEEE Symposium on*, pages 6–pp. IEEE, 2006.

[47] N. Jovanovic, C. Kruegel, and E. 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] N. Jovanovic, C. Kruegel, and E. K. 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] KIT. JOANA (Java Object-sensitive ANAlysis) - Information Flow Control Framework for Java . *KIT*, [online] http://pp.ipd.kit.edu/projects/joana/.

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

[53] P. Li and S. Zdancewic. Downgrading policies and relaxed noninterference. In *ACM SIGPLAN Notices*, volume 40, pages 158–170. ACM, 2005.

[54] Y. Liu and A. 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.

[55] V. B. Livshits and M. 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.

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

[57] S. McCamant and M. D. Ernst. Quantitative information flow as network flow capacity. *ACM SIGPLAN Notices*, 43(6):193–205, 2008.

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

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

[60] Y. Minamide. Static approximation of dynamically generated web pages. In *Proceedings of the 14th international conference on World Wide Web*, pages 432–441. ACM, 2005.

[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] P. Muntean, A. Rabbi, A. Ibing, and C. Eckert. Automated detection of information flow vulnerabilities in uml state charts and c code. In *IEEE International Conference on Software Quality,Reliability and Security*. IEEE, 2015.

[63] 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.

[64] A. 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] 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.

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

[67] A. C. Myers and B. Liskov. Protecting privacy using the decentralized label model. volume 9, pages 410–442. ACM, 2000.

[68] A. C. Myers, L. Zheng, S. Zdancewic, S. Chong, and N. Nystrom. Jif: Java information flow.

Software release. Located at http://www. cs. cornell. edu/jif, 2005, 2001.

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

[70] 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.

[71] Microsoft Run-Time Library Reference. MSDN run-time library reference - SAL annotations, https://msdn.microsoft.com/en-us/library/vstudio/ms235402(v=vs.100).aspx.

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

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

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

[75] 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.

[76] A. Sabelfeld and D. Sands. Declassification: Dimensions and principles. volume 17, pages 517–548. Citeseer, 2009.

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

[78] M. Samuel, P. Saxena, and D. Song. Context-sensitive auto-sanitization in web templating languages using type qualifiers. In *Proceedings of the 18th ACM conference on Computer and communications security*, pages 587–600. ACM, 2011.

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

[80] L. K. Shar and H. B. K. Tan. Predicting common web application vulnerabilities from input validation and sanitization code patterns. In *Automated Software Engineering (ASE), 2012 Proceedings of the 27th IEEE/ACM International Conference on*, pages 310–313. IEEE, 2012.

[81] L. K. Shar, H. B. K. Tan, and L. C. Briand. Mining sql injection and cross site scripting vulnerabilities using hybrid program analysis. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 642–651. IEEE Press, 2013.

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

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

[84] 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.

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

[86] 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.

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

[88] O. Tripp, M. Pistoia, S. J. Fink, M. Sridharan, and O. Weisman. Taj: effective taint analysis of web applications. *ACM Sigplan Notices*, 44(6):87–97, 2009.

[89] J. Viega and G. McGraw. *Building secure software: how to avoid security problems the right way*. Pearson Education, 2001.

[90] Visual-paradigm. Visual-paradigm. Technical report, Visual-paradigm, http://www.visual-paradigm.com/VPGallery/diagrams/Sequence.html.

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

[92] J. Weinberger, P. Saxena, D. Akhawe, M. Finifter, R. Shin, and D. Song. A systematic analysis of xss sanitization in web application frameworks. In *Computer Security–ESORICS 2011*, pages 150–171. Springer, 2011.

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

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

[95] J. Wilander and M. Kamkar. A comparison of publicly available tools for static intrusion prevention. 2002.

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

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

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

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

[100] A. Yip, Xi Wang, N. Zeldovich, and M. F. 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.

[101] F. Yu, M. Alkhalaf, and T. Bultan. Stranger: An automata-based string analysis tool for php. In

Tools and Algorithms for the Construction and Analysis of Systems, pages 154–157. Springer, 2010.

[102] F. Yu, M. Alkhalaf, and T. Bultan. Patching vulnerabilities with sanitization synthesis. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 251–260. ACM, 2011.

[103] Y. Zheng and X. Zhang. Path sensitive static analysis of web applications for remote code execution vulnerability detection. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 652–661. IEEE Press, 2013.