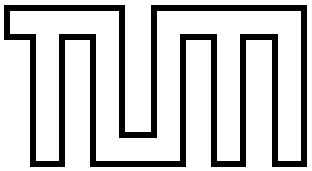


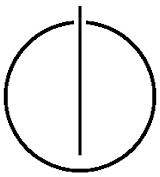
FAKULTÄT FÜR INFORMATIK

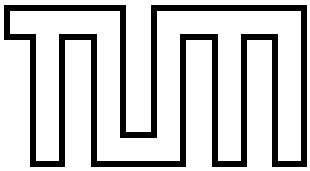
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Master's Thesis

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

Md Adnan Rabbi





FAKULTÄT FÜR INFORMATIK

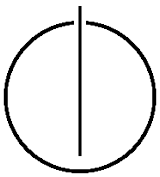
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Master's Thesis

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

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

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



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

München, den 29. August 2015 Md Adnan Rabbi

Acknowledgments

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

Abstract

An abstracts abstracts the thesis!

Contents

Outline of the Thesis

Part I: Introduction and Theory

CHAPTER 1: INTRODUCTION

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

CHAPTER 2: BACKGROUND INFORMATION

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

Part II: Implementation and Analysis

CHAPTER 3: CHALLENGES AND ANNOTATION LANGUAGE EXTENSION

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

CHAPTER 4: IMPLEMENTATION

This chapter presents the implementation of the system.

CHAPTER 5: EXPERIMENTS

This chapter presents the different application area of the system.

Part III: Conclusion and Future Work

CHAPTER 6: CONCLUSION AND FUTURE WORK

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

Part I.

Introduction and Theory

1. Introduction

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

A solution for tagging sanitization, declassification and authentication in source code is based on libraries which contain all needed annotations attached to function declarations. This approach plays an important role mainly for static analysis bug detection techniques where the information available during program run-time. Detection of information flow vulnerabilities uses dynamic analysis techniques , static analysis techniques and hybrid techniques which combine static and dynamic approaches. The static techniques need to know when to use sanitization , declassification and authentication functions. Data sanitization has been studied in the context of architectures for high assurance systems, language-based information flow controls and privacy-preserving data publication [1]. A global policy of noninterference which ensures that high-security data will not be observable on low-security channels. Because noninterference is typically too strong a property, most programs use some form of declassification to selectively leak high security information[2].Declassification is often expressed as an operation within a given program. Authentication is the way through which the users gets access to a system. In this research main focus are these three types of functionalities which are sanitization, declassification and authentication errors in UML state charts.

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

by tools which can use the annotations and check if information flow constraints hold or not based on information propagation techniques.

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

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

1.1. Latex Introduction

There is no need for a latex introduction since there is plenty of literature out there.

2. Background Information

2.1. Sanitization

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

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

Some basic purpose of sanitization are given below:

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

2.2. Declassification

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

To declassify information means lowering the security classification of selected information. Sabelfeld and Sands [4] identify four different dimensions of declassification, what is declassified, who is able to declassify, where the declassification occurs and when the declassification takes place.

Myers and Liskov introduced the decentralized label model [3], describing how labels could be applied to a programming language and then used to check information flow policy compliance in distributed systems. The framework includes a declassify function for downgrading data if the owners policies allow. The model allows principals to define their own downgrading policies.

2.3. Authentication

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

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

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

2.4. Detecting Information Flow Errors During Design:

If a step of function call like authentication, sanitization or declassification is missing inside the program then this can lead to software vulnerabilities.

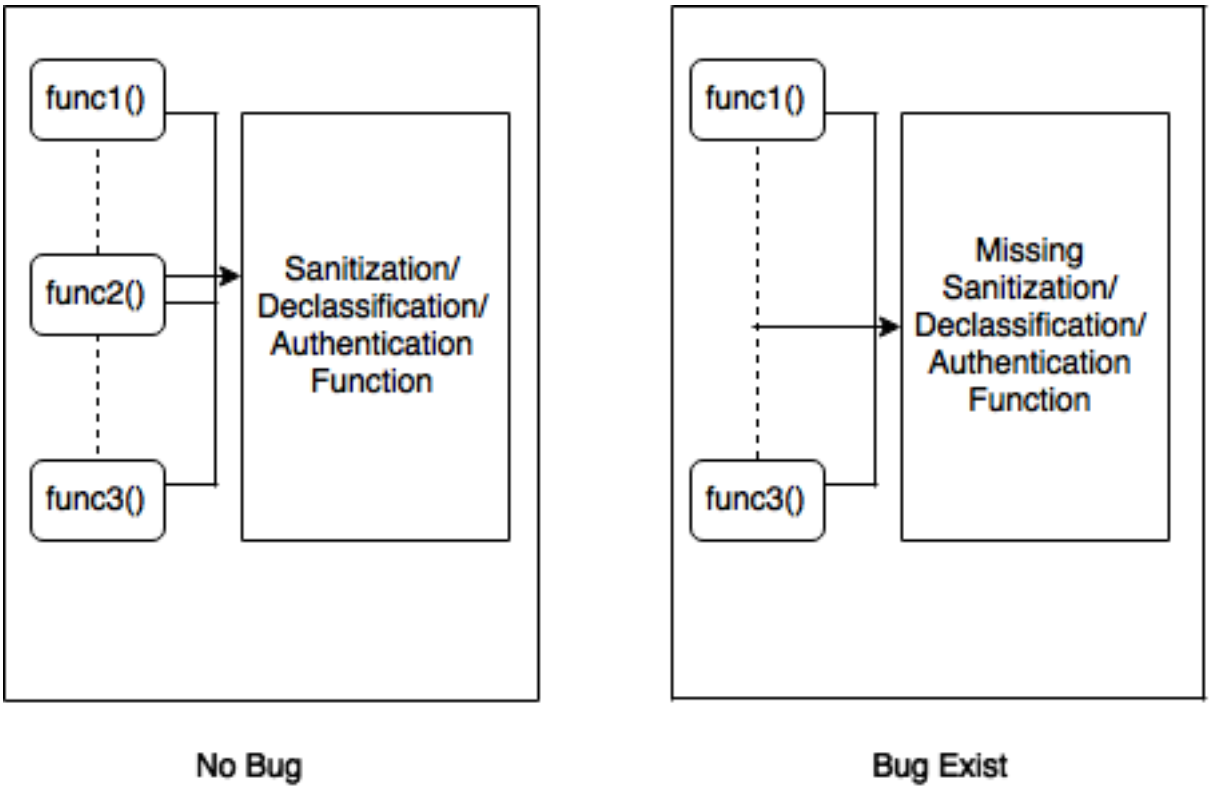


Figure 2.1.: Information flow errors during design

2.5. Detecting Information Flow Errors During Coding:

Part II.

The Second Part

3. Challenges and Annotation Language Extension

3.1. Challenges and Idea

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

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

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

The 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 400 LOC with code comments included. Source code generation is also supported by using 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 that can easily be used for annotating models and source files.

3.2. Annotation Language Tags

3.3. Language Implementation Process

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

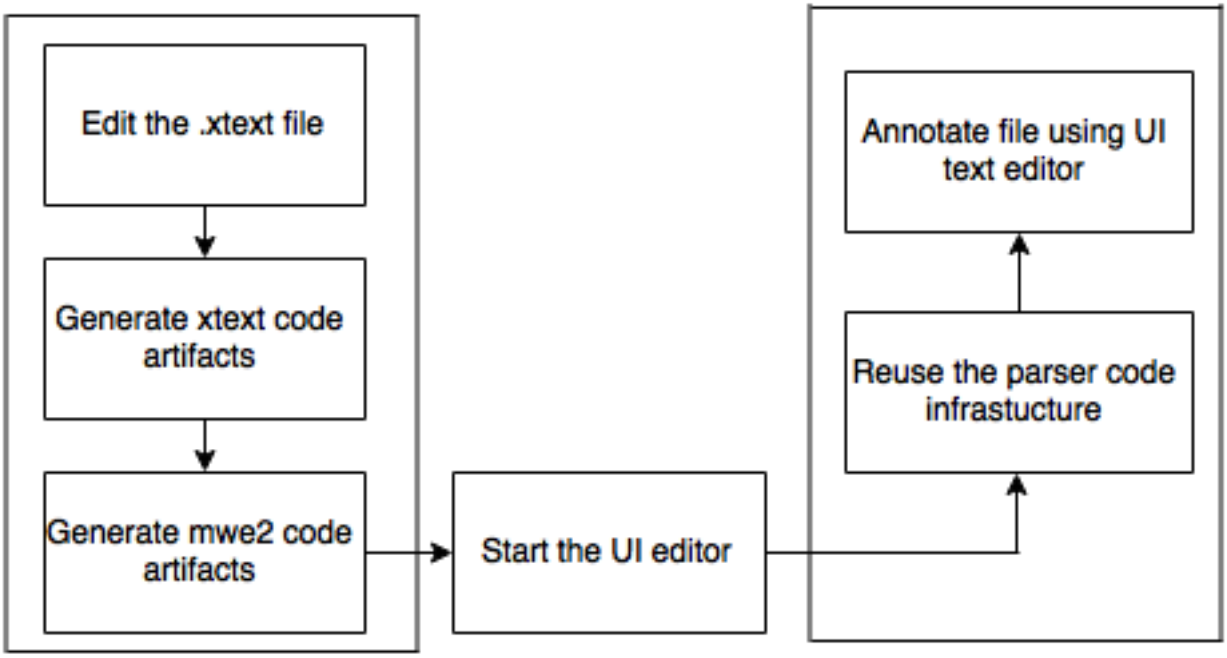


Figure 3.1.: Annotation language design process

4. Implementation

4.1. Complete Workflow of the System:

4.2. The Grammar of Annotation Language:

As per requirements previous annotation language grammar which is written in xtext language has been extended. Extra annotation have included like ”@return”, ”True”, ”False”, ”sanitization”, ”declassification”, ”authentication”. Some the code snippet of extended xtext grammar is given below.

```
/**
* @FunctionAnnotation :used for function annotations
*/
FunctionAnnotation returns FunctionAnnotation:
{FunctionAnnotation}(
result += '@return' (' ')? (level=('H'|'L'))? ('\n' | '\r')?
);

/**
* @SingleLineAnnotation :used for adding single line annotations
*/
SingleLineAnnotation returns SingleLineAnnotation:
{SingleLineAnnotation}(
result+= '//@ @parameter 'parameter=ID(securityType=SecurityType)?
    (' ')? (level =('H'|'L'))? ('True'|'False')? ((nameComment=ID))?
    ('\n' | '\r')?
);

/**
* @FunctionType :annotaions types for functions
*/
enum FunctionType: declassification
| sanitization
| authentication
```

4.3. UML State Chart Editor:

UML state chart editor has been extended based on the open source Yakindu SCT [21]framework. The existing language grammar with annotation language grammar has extended in order to support new set of tags. Furthermore, an annotation proposal filter implemented which was used to filter out the annotation language tags of the Yakindu SCT language

grammar.

4.4. Source Code Editor:

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

4.5. C Code Generator:

C code generator has extended based on Eclipse EMF and xTend which is used to generate the state chart execution code containing the previously added security annotations from UML state charts. The code generator outputs two files per UML state chart (one .c and one .h file). Generated annotations can reside in both header file and source code file. Previously annotated UML state chart states are converted to either C function calls or C variables declarations, both have been previously annotated. We use the available state chart execution flow functionality which is responsible for traversing the UML state chart during state chart simulation. The UML state chart will be traversed by the code generation algorithm and code is generated based on the mentioned state chart execution flow. The generated code will contain at least one bad path (contains a true positive) and a good path (contains no bug) per UML state chart if those paths were previously modeled inside the UML state chart.

4.6. View Buggy Path in Sequence Diagram

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

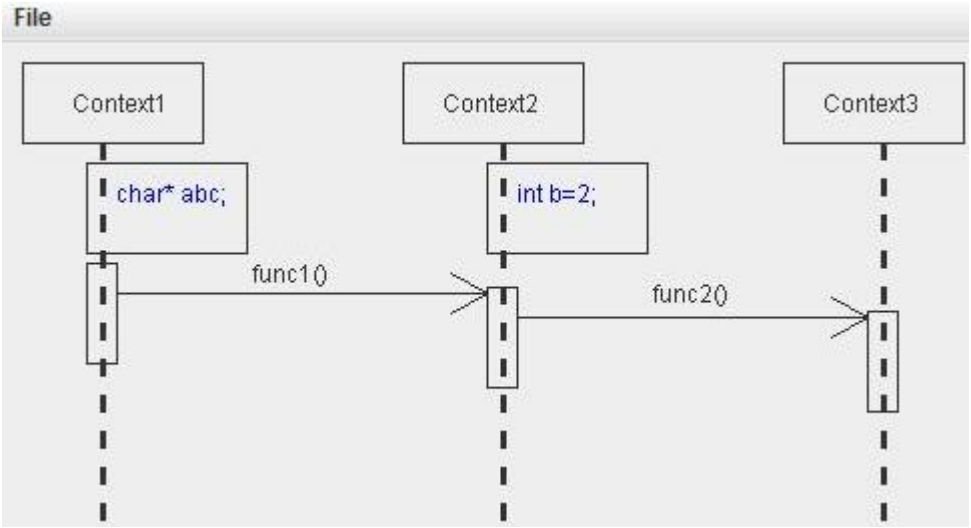


Figure 4.1.: Error trace path in sequence diagram

5. Experiments

6. Related Work

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

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

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

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

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

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

7. Conclusion and Future Work

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

Appendix

A. Detailed Descriptions

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

Bibliography

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

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

[3] Andrew C Myers and Barbara Liskov. Protecting privacy using the decentralized label model. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 9(4):410–442, 2000.

[4] Andrei Sabelfeld and David Sands. Declassification: Dimensions and principles. *Journal of Computer Security*, 17(5):517–548, 2009.