TUM

TECHNISCHE UNIVERSITÄT MÜNCHEN
INSTITUT FÜR INFORMATIK

Automatic Security Checks on the Model Level

Paul Muntean und Alexander Malkis

TUM-I1657

SIBASE
Technische Universität München
Chair for IT Security
Workgroup TP 5.1

SIBASE Report

TP 5.1 – AP 5.1.3: Automatic Security Checks on the Model Level

Paul Muntean (TUM), and
Alexander Malkis (TUM)

SIBASE
Technische Universität München
Chair for IT Security
Workgroup TP 5.1

SIBASE Report

TP 5.1 – AP 5.1.3: Automatic Security Checks on the Model Level

Editor: Paul Muntean, and
Alexander Malkis
(paul@sec.in.tum.de, and
malkis@sec.in.tum.de)

Authors:	Paul Muntean, TUM
	Alexander Malkis, TUM
TP Responsible:	Alexander Malkis
Version:	15 November 2016
Submission date:	15 November 2016

Funded by BMBF under grant number 01IS13020

Abstract

The working package AP 5.1.3 deals with descriptions of security requirements on the modeling level of UML statecharts as well as with automatic checking of UML statecharts against such descriptions. We use textual annotations to introduce information-flow constraints in UML statecharts. The constraints concern mainly authentication, declassification, and sanitization errors. The annotations are automatically loaded by information-flow checkers that check whether the imposed constraints hold or not. For the purpose of checking, the UML statecharts are transformed into C source code, and errors traces are presented as UML sequence diagrams. Together with the checkers, we developed an annotation language editor, a UML statechart editor and a source code generator. All the implementation uses Eclipse Modeling Framework. The experimental results show that this approach is effective and could potentially be further applied to other types of UML models and to programming languages other than C in order to detect different types of vulnerabilities.
Our results use and build upon SIBASE working packages 5.1.2, 5.1.4 and 5.2.1.

Version	Date	Author	Comment
0.1	31.07.2016	P. Muntean	Initial draft
1.0	15.11.2016	A. Malkis	Submission

Contents

List of Figures

Listings

List of Tables

List of Acronyms

ANTLR	ANother Tool for Language Recognition
API	Application Programming Interface
AST	Abstract Syntax Tree
ATM	Automated Teller Machine
ECore	EMF File Extension
DSL	Domain-Specific Language
EBNF	Extended Backus-Naur Form
EMF	Eclipse Modeling Framework
ESC	Extended Static Checking
GMF	Graph Modeling Framework
GUI	Graphical User Interface
GPL	General Purpose Language
HTML	HyperText Markup Language
HTTP	HyperText Transfer Protocol
IDE	Integrated Development Environment
JDK	Java Development Kit
JVM	Java Virtual Machine
MWE	Modeling Workflow Engine
NVD	National Vulnerability Database
PaaS	Platform as a Service
OMG	Object Management Group
OS	Operating System
OSGi	Open Services Gateway initiative
OTP	One-Time Password
OWASP	Open Web Application Security Project
PDA	Personal Digital Assistant
PHP	PHP: Hypertext Preprocessor
SAL	Standard Annotation Language
SDLC	Software Development Life Cycle
SMS	Short Message Service
SMT	Satisfiability Modulo Theory
SQL	Structured Query Language
SQLi	SQL injection
TUM	Technische Universität München
UML	Unified Modeling Language
XML	Extensible Markup Language
XSS	cross-site scripting
xTend	A programming language on top of Java VM, see [28]
xText	A framework for developing programming languages and DSLs, see [29]

1 Introduction

The US National Vulnerability Database (NVD) [68] lists 6626 common vulnerabilities and exposures in the last 12 months from which 682 (10.3%) are information leaks caused by inappropriate handling of information flow in software applications. Inappropriate handling of information flow can cause a wide range of problems such as information flow leakages/disclosure, weird program behavior and weaker cryptographic algorithm encryption, to name just a few.
These types of vulnerabilities are introduced during design, architecture or coding phase and can be potentially exploited if not discovered early. Information flow vulnerabilities in UML models and code are introduced by software designers or programmers who are sometimes "blind" with respect to the fact that they are trained to focus point-wise (one code line and one data flow at a time). For this reason appropriate techniques and tools can be of great help.
Information flow 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 [13] can be used to provide such additional knowledge. However, code annotations can increase the number of source code lines by 10% [49].
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 use information propagation techniques in order to detect information flow violations.
The detection of information flow vulnerabilities in code and UML statecharts is challenging and not well addressed. Foremost, there is no common annotation language for annotating UML statecharts and source code with information flow security constraints such that vulnerabilities can be detected also when code is not available. Second, there are no automated checking tools which can reuse the annotated constraints in early stages of software development to check for information flow vulnerabilities. We think that it is important to specify security constraints as early as possible in order to avoid later costly repairs or exploitable vulnerabilities.
In this paper, we address this open problem by providing:

- an extension of a light-weight security annotation language,
- an editors used to edit UML statecharts,
- an editor used to edit source code files and
- information flow checkers.

The checkers can automatically load and use code annotations in order to detect explicit and implicit information flow [19] vulnerabilities based on Extended Static Checking (ESC) [20] in UML statecharts and C code. In summary, our main contributions are:

- The extension of our annotation language with new annotation language tags.

- An UML statechart editor used to create and annotate UML statechart models.
- A source code generator used to generate C source code from UML statechart models.
- A source code editor used to further refine generated source code (if needed).
- Three static code analysis checkers used to detect declassification, authentication and sanitization bugs in UML statecharts.

The rest of this paper is organized as follows: § 2 presents background knowledge needed to understand the rest of this paper, § 3 presents challenges, ideas and our annotation language tags set, § 4 briefly presents our implementation, § 5 highlights experiments performed with our tools, § 6 presents related work, and § 7 concludes by showing future steps.

2 Background

This chapter presents the basics of user input sanitization in § 2.1, declassification of confidential information in § 2.2, authentication of user access in § 2.3, information flow vulnerabilities in § 2.5, which includes detecting of information flow errors during design phase in § 2.5.1 and detecting of information flow errors during coding in § 2.5.2.

2.1 Sanitization of User Input

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 [78]. Sanitization ensures that user input can be safely used in internal program operations such as, e.g., SQL queries. For example, a web application not employing sanitization could be attacked by feeding them with untypical input which will become part of a sensitive operation inside the web application such that this operation will have a malicious effect different from the originally intended one.
According to [95], three of the top five most common web-site attacks are SQL injection, cross-site scripting (XSS) and remote file inclusion (RFI). The root cause of the three attacks is common: the lack of input sanitization. Each of the three exploits is leveraged by input sent to a web server by an end user. When the end user acts legitimately, the data he/she sends is related to his/her interaction with the web-site. But when the end user acts as attacker, he/she exploits this mechanism by sending the input that is deliberately constructed to escape the legitimate context and carry out unauthorized actions.
On the one hand, sanitization is needed for data from less-trusted domain before a transfer to a component in another, more-trusted domain [55]. Especially data based on user input has to be cleaned in order to prevent an attacker exploiting a security hole. On the other hand, input sanitization is not a panacea: it can give a false sense of security to the web server owners since input sanitization prevents just one of many sources of vulnerabilities. E.g., sanitizing just the input does not prevent leaking sensitive information through the output and other types of attacks.
Overall, properly applied sanitization enjoys the following properties (cf. [54, 102]): (i) it removes malicious elements from the input, (ii) parameters and global variables which must be sanitized before calling useful functions are identified, (iii) for the correct functioning of the application it is acceptable to modify the input (in case it is believed to be malicious) by passing the untrusted user data through a trusted sanitization function, (iv) any user input data flows through a sanitization function before going into a SQL query, (v) output data is also cleansed to avoid leaks of confidential information, (vi) on the code level, most control flow paths that go from a source to a sink pass through a sanitizer, (vii) developers typically define a small number of well-thought sanitization functions in libraries, calling them on need. Summarizing, well-thought sanitization is the one of the simple and effective means of preventing attacks on

web applications.

2.2 Declassification of Confidential Information

Declassification of information deals with lowering the security classification of selected information such that it becomes secure to further expose this information to a third party. Sabelfeld et al. [76] identify four different dimensions of declassification: (i) what is declassified, (ii) who is able to declassify, (iii) where the declassification occurs, and (iv) when the declassification takes place. According to [76], the *what* and *when* dimensions can be formalized relatively straightforwardly. The *what* dimension abstracts the black-box, input/output view of the system. As opposed to it, the *when* dimension concerns a white-box, internal view of the system: it requires some notion of time in the semantics. The *who* dimension is formalized by attaching ownership information to data. The *where* dimension reduces to formalizing so-called intransitive noninterference, constrained noninterference, non-interference modulo trusted functions, etc.
Also others characterized declassification using these four dimensions: (i) *what* information is released [15, 16, 30, 42], (ii) *who* releases information [14, 65, 66], (iii) *where* in the system information is released [37], and (iv) *when* information can be released [31].
Myers et al. [67] introduced a type system for information flow for an imperative programming language Jif equipped with a compiler. They describe an execution platform which ensures that the types get preserved across applications running on the platform. In the context of confidentiality, their information-flow model permits specifying, roughly speaking, a hierarchy of principals, who can own the data or read the data, and information-flow constraints as types. These constraints represent, somewhat simplifying, a relation denoting which principal can read which variable at which control flow point. The principals may declassify data they (fully or partially) own in a fine-grained way. Jif allows the coders to specify the aforementioned constraints; the compiler checks that the program's transition relation does not violate the constraints and that the principals do not declassify more than they have the right to. In effect, type checking ensures that confidential data is distributed only to the specified readers up to correct declassification.

2.3 Authentication of User Access

Authentication deals with confirming the identity of the users who try to access a certain system. In order for a user to get access to a resource, the user must first prove that he/she is who he/she claims to be. This is typically handled by passing a key with each request (often called an access token; it is ordinarily generated after the user provides his/her id and the password) [21].
In order to clarify the difference between *authentication*, *authorization* and *accounting* we provide their definitions:

Authentication: confirming that a user is who he/she is claiming to be [3],

Authorization: determining whether the user has the rights to perform certain actions or access a service [4], and

Accounting: measuring the resources consumed by a user during access, potentially followed by billing [89].

Arce et al. [2] propose a process on how to avoid the top ten software security flaws. They provide a list of rules to follow. For us, the most relevant rule is "authorize after your authenticate".

2.4 Static Code Analysis

Static program-analysis is used to mechanically examine computer programs without actually running them. On the contrary, the analysis performed on running programs is known as dynamic analysis. Usually, the static program-analysis is performed on the source code, slightly less often on the object code, even less often on executable binaries.
Analyzing the program both statically and dynamically is sometimes referred to as glass-box testing [85]. One of the main advantages of the static code-analysis is high path coverage and that it can reveal errors before they manifest. Static program-analysis has its advantages and limitations [40]. One advantage is that if the code is formally proven correct (and if the proof is correct), then the code need not be tested for the proven properties.

2.5 Information Flow Vulnerabilities

Information flow can be classified as *explicit* (information flow that arises explicitly, due to e.g., assignment statements) and *implicit* (flow that arises implicitly, e.g., due to conditional statements) [51]. Information flow vulnerabilities arise when information which is considered confidential is leaked outside a system.
A software system is considered secure with respect to confidentiality if it enforces the confidential policies of its users [74]. The analysis which checks if a system can be considered confidential needs to ensure that there is no data flow which violates one of the previously enforced security policies. A firmly secured system can be obtained by precisely expressing confidentiality policies and translating them into mechanisms that enforce these policies.
On the one hand, confidentiality requires that data flows from private to public variables should be restricted [99]. On the other hand, integrity requires that flows to private variables should be restricted. Automating these requirements demands that each program variable is labeled by a certain security level. The security levels can vary. The basic model comprises only two distinct levels: L for (Low, public) and H for (High, private). These two labels mean publicly observable information and secret/private information, respectively.

2.5.1 Vulnerabilities Detection During Design

Figure 2.1 depicts the general software development life-cycle used to build software.

Figure 2.1: Software development life-cycle.

This representation provides a guideline on how to develop software in general and consists of:

① requirements gathering and analysis,

② design and modeling,

③ implementation or coding,

④ testing and

⑤ operation and maintenance.

(Typically, ⑤ is followed by ① for a new product or a new product version resulting from requirements which changed in the meantime—hence the name "cycle".)
Figure 2.2 depicts an example of a software bug which can be addressed during design phase by adding a call to a function from one of the following classes: *authentication*, *sanitization* or *declassification*.

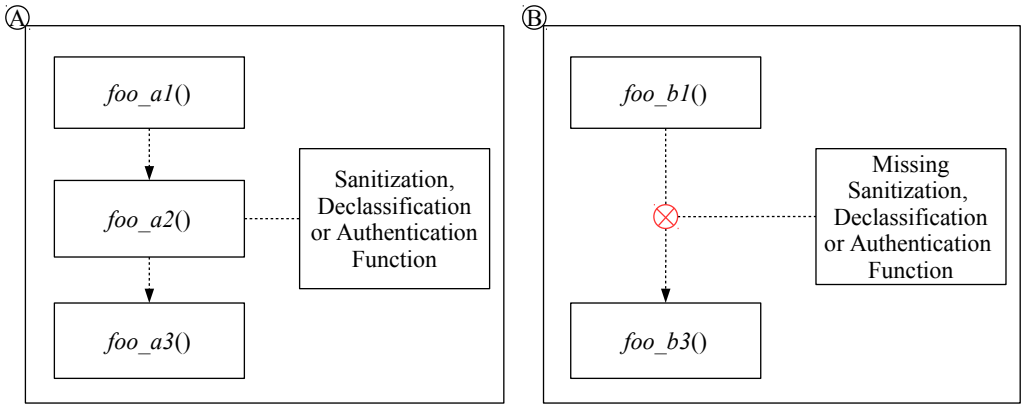

Figure 2.2: Information flow errors during design.

In this treatise we present a method used to detect information flow vulnerabilities during phase ②. Missing function calls to sanitization, declassification or authentication functions are typical causes of security-related vulnerabilities. Figure 2.2 depicts two scenarios (Ⓐ and Ⓑ): the right one with a bug, and the left one without a bug. The left-hand side (scenario Ⓐ) function *foo_a2*() employs one of the *sanitization*, *declassification* or *authentication* functions before calling *foo_a3*(), whereas in the right-hand side (scenario Ⓑ) the data does not flow first through the previously mentioned functions before calling function *foo_b3*(). The bug, which is a missing call to one of the previously mentioned functions, is indicated as ⊗ in Figure 2.2.
This bug can be avoided during the design phase by carefully modeling the data flow with an UML statechart diagram, for example. We achieve this by adding data flow annotations/assertions to the diagrams which are checked automatically later.

2.5.2 Vulnerabilities Detection During Coding

Information-flow vulnerabilities can be detected during the coding phase as well. Next, we present how an information flow error can be detected in source code.
Figure 2.3 depicts two information flows (Ⓢ1 and Ⓢ2) according to the lattice model for secure information flow described by Denning [19].
Figure 2.3 depicts two software sub-systems, *System 1*, Ⓢ1 and *System 2*, Ⓢ2 containing two data flows which do not interact with each other. Each of the data flows in the two systems starts at the top of the Ⓢ1 and Ⓢ2 sub-systems by first declaring variables *a* and *b*, respectively, and assigning some input data to them. This piece of data goes through a security-related function

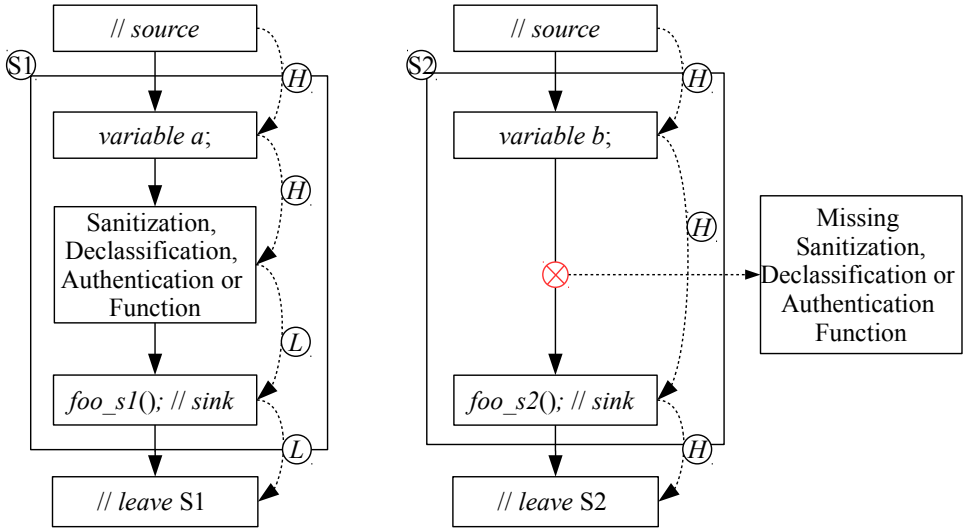

Figure 2.3: Information flow errors during coding.

in (S1) and is processed then by *foo_s1*(), after which the result of processing is output. In (S2), no security is ensured before the data is processed by *foo_s2*(). After *foo_s1*() and *foo_s2*(), the data leaves the system at "// *leave* S1" and "// *leave* S2", respectively.

Note that a *source* is any function or programming language statement which provides private (confidential) information through a system boundary. A *sink* can be a function call or any other programming language statement which exposes private information to the outside of the system through a system boundary. A *system boundary* can be a statement, function call, class, package or module. The exact definition of the three concepts depends on the project.

System 1 is depicted in the left hand side of Figure 2.3 containing the flow from the source (// *source*) to the *sink*; leaving the system is indicated with "// *leave* S1" at the bottom of this information flow. Consider the case that the variable *a* is assigned confidential data from the source and that the variable *a* is tagged with label H (High, confidential). This simply means that variable *a* introduces confidential information into the *System 1*. The dashed arrows represent the passing of the confidentiality label (which is H or L here) between the program statements. The data is passed through an *authentication*, *declassification* or *sanitization* function, whose output is no more considered confidential. When data from a variable labeled with L is about to leave *System 1*, it is compliant with our security requirements, because only non-confidential values are allowed to leave the system.

System 2 is depicted in right hand side of Figure 2.3 containing a data flow similar to the data flow depicted in *System 1*. We observe that data leaves (S2) without its label being lowered from H to L; thus there is an information flow bug. One possible position of this bug in the code is indicated with ⊗ in Figure 2.3. Here, the bug is caused by data (originally obtained via a variable *b*) which left *System 2* but was not previously passed through a *sanitization*, *declassification* or *authentication* function.

3 Annotation Language Challenges

This chapter presents challenges and ideas needed in order to develop our system in § 3.1, the annotation language tags of our annotation language in § 3.2, and the annotation language extension process § 3.3 used to add or refine the grammar of our annotation language. The last step is essential since the grammar of our annotation language can not be debugged as usual programs. make

3.1 Challenges and Ideas

We briefly list the main challenges of our approach in order to give an overall picture of the used technologies how these can be used together. The main challenge of our approach is to develop a new system based on Eclipse xText [29], Eclipse xTend [28], YAKINDU SCT [1] and the static analysis engine Smtcodan [63].
These tools need to be used in order to create two editors (one source code editor and one UML state chart editor), a C source code generator, and three information flow checkers.
Next we will describe what xText, xTend and YAKINDU SCT are and how these work.

3.1.1 xText

XText is a framework for development of programming languages and domain specific languages. According to [29], it covers all aspects of a complete language infrastructure. Listing 3.1 depicts an example of xText code.

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.
```

XText provides several DSLs used to describe different aspects of a programming language. The xText-based implementations are running on top of Java JVM. The compiler components are not dependent on Eclipse OSGi. They include:

- a parser,
- the type-safe abstract syntax tree (AST),
- a serializer,
- a code formatter,
- a linking and scoping framework,

- compiler checks,
- static analysis validation,
- a code generator and
- an 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 such as the Graphical Modeling Project GMF.
We want to use an xText-based grammar in order to parse the whole C/C++ language in order to obtain the ECore 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 statechart editor.
For C source code generation we want to use xTend, ANTLR and .mwe2 files. To parse other programming languages as well our annotation language a parser can be used. The desired results are an extensible policy language and a highly reusable source code implementation as well as a C source code generator that can easily be used for annotating models and source files.

3.1.2 xTend

The source code written in the statically-typed programming language xTend can be easily converted into Java code. XTend is based on Java but has many improvements such as:

- extension methods,
- lambda expressions,
- active annotations,
- operator overloading,
- powerful switch expressions,
- multiple dispatch and
- template expressions.

XTend has zero interoperability issues with Java: everything users write interacts with Java exactly as expected.
XTend can seamlessly interact with Java code. The developers of xTend claim that their programming language is much more concise and readable. At the same time they mention that the library which xTend provides can easily interact with the Java JDK.

3.1.3 Type Inference

Java has a limitation w.r.t. type inference: the user is forced to write a lot of type signatures. For this reason static typing is disliked. XTend is statically typed just as Java, but by using xTend users have to rarely write types because these can be deduced from the context. The type of a name and the return types of methods can be inferred from the context. Classes and methods are public by default and fields are private.
The challenge w.r.t. type inference is that we want to make useful user editing suggestions when editing source code files of UML state charts. For this reason the type inference capabilities offered by xTend where used.

3.1.4 Yakindu SCT

YAKINDU Statechart Tools (SCT) is an open-source tool which uses the concept of state machines for specification and development of reactive, event-driven systems. It contains a graphical user interface (GUI) for editing statecharts. It provides validation, code generation and simulation capabilities. The main features are: support for code generation (C/C++ and Java), a simulation engine used to execute models, state machines can be syntactically and semantically validated, and integration of graphical and textual modeling.

3.2 Language 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.

Table 3.1 depicts the new tags which were added to the annotation language [64]. Table 3.1 depicts the annotation language target types and the annotation tags which can be used in combination with the tags: @*function*, @*parameter*, @*variable*, @*preStep* and @*postStep*.
These can be used to annotate the function parameter with *authenticated*, *declassified* or *sanitized* having H or L. The tag @*variable* is used to annotate the variable of C/C++ code with *confidential*; H or L 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 xText language definition grammar.
For detecting *authentication*, *declassification* and *sanitization* errors new function tags are included such as *authentication*, *declassification* and *sanitization*. Also, for parameters new tag types are included such as *authenticated*, *declassified* and *sanitized*. These parameter can be also appended with H (High) or L (Low) tags. High means that the parameter is confidential or secured and low means that the parameter is not secured.
The tag @*preStep* is used to annotate the previous expected function call name and the tag @*postStep* is used to annotate the next expected function call name. These tags can be used in a *chain* of function calls for example.

3.3 Language Extension Process

We implemented the new annotation language tags based on xText. XText can be used to generate different components. The generated components are: the parser, the serializer, the inferred *ECore* model, and classes for content assist.
The generator contributes to the Guice modules, plugin.xml and Manifest.mf files. XText generator uses a special DSL called MWE2. This file is used to configure the code generator. MWE2 is used to create object graphs in a declarative and condensed way. Next, Java classes are instantiated, getter and setter methods are used to make the configuration.

Listing 3.2: Single-line and multi-line comment rules defined in xText.

```
/** @SL_COMMENT :all strings which follow // | || | } will be a
    single-line comment */
terminal SL_COMMENT  : '//'!('@') !('\n'|'\r')* ('\n'|'\r')*
// '}' can be used optionally to skip method bodies during parsing
// together with multi-line line {} comment
//    | '}'      !('\n'|'\r')* ('\n'|'\r')*
;
/** @ML_COMMENT :@/* multi-line comment excluding @ from inside
*         :{} multi line comment */
terminal ML_COMMENT : '/*' !('@') -> !('@')'*/' !('\n'|'\r')*
    ('\n'|'\r')*
// '{' -> '}' can be used optionally to skip method bodies
//     | '{' -> '}'            ('\n'|'\r')?
;
```

Listing 3.2 depicts the *ML_COMMENT* (multi-line) and *SL_COMMENT* (single-line) grammar rules, which were used to add single and multi-line comments to our annotation language. The new annotation language tags were added to the annotation language presented in AP 5.1.2 [62].

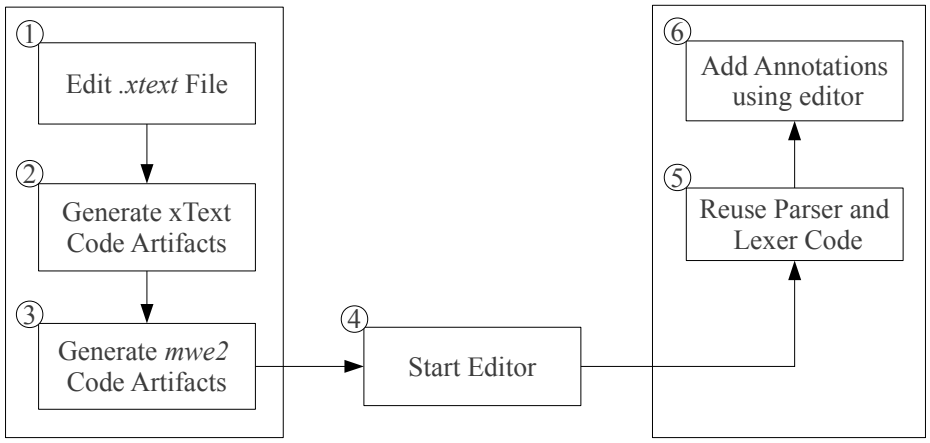

Figure 3.1: Annotation language design process.

Figure 3.1 depicts the process used in order to implement our annotation language. The process is comprised of the following steps:

① The *.xtext* file containing the language grammar is edited with new grammar components.

② The grammar file is compiled and software artifacts are generated.
③ After editing the *.mwe2* file it is compiled; the result of compilation is: a parser, a lexer and class bindings between these the lexer, parser and the grammar *ECore* model.
④ The editor will be started. This reuses the previously generated code artifacts.
⑤ The generated parser, lexer and the bindings will be reused inside our static analysis engine and in our source file editor.
⑥ After the new lexer, parser and *ECore* model were generated we can edit the UML statechart or the source code file with the new annotation language features which were just added.

4 Implementation

This chapter presents the implementation details concerning

- the system architecture of our system in § 4.1,
- our annotation language grammar in § 4.2,
- the inference rules used inside our checkers in § 4.3,
- our UML statechart editor in § 4.4,
- our source code editor in § 4.5,
- our C code generator in § 4.6,
- our three static analysis checkers which are used to detecting information flow bugs in §,4.7, and
- how a buggy program path can be viewed as an UML sequence diagram in § 4.8.

4.1 System Architecture

Figure 4.1 depicts the work-flow for using our approach.

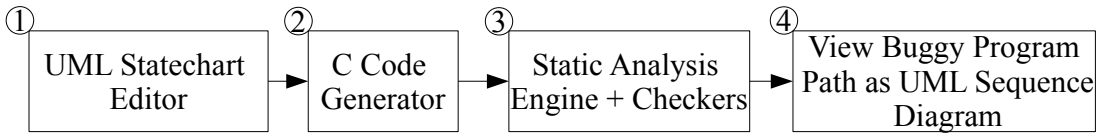

Figure 4.1: System overview

First, for annotating UML statecharts, we developed an UML statechart editor, shown as ① in Figure 4.1. For handling UML statecharts the open source framework Yakindu SCT has been chosen. The goal is to model C/C++ source code into UML statecharts. In this way bugs can be detected during software design phase. Inside the Yakindu SCT editor the annotation language grammar has also been included using xText. The user can easily annotate UML statecharts in order to detect the information flow vulnerabilities.
Second, the C code generator has been developed inside the Yakindu SCT editor using xTend. The C code generator is represented as ② in Figure 4.1. After modeling the C code files in Yakindu SCT editor, the user can generate the code using the C code generator. The generator can (for now) generate two types of files. One C source file having the .c extension and another C header file having the .h extension. Inside those files the previously added annotations will be included when code is generated. These annotations will be used afterwards in order to detect information flow errors. There is an option to further annotate the generated code. For this purpose we developed a source code editor. The source code editor can be used for source code editing; this editor is based on xText. The source code editor is not depicted the Figure 4.1 but can be used after step ② to annotate C/C++ source code and header files. This editor has been developed as an Eclipse plug-in. The plug-in has to be available (previously copied in

the corresponding folder) in Eclipse such that then user can easily annotate C/C++ source code files and header files. The annotation suggestions are displayed by pressing the combination Ctrl+Space on the keyboard.
Third, we used our static analysis engine, Smtcodan, inside the three checkers (under ③ in Figure 4.1) we developed. The first checker is used for detecting authentication errors, the second one for detecting declassification errors and the third one for detecting sanitization errors. Finally, for an improved viewing of a buggy program path a UML sequence diagram generator has been developed. It is shown as ④ in Figure 4.1. The goal of developing a better visual representation is to help the user to speed up finding the location of a bug in the generated source code.

4.2 Language Grammar

Figure 4.2 depicts the grammar of our annotation language in Extended Backus Naur Form (EBNF) (see [64] for mode details).

Ann_Lang	::=	HeaderModel*;
H_Model	::=	*S_L_Anno*; ;single-line comment rule
	|	*M_L_Anno*; ;multi-line comment rule
	|	*Func_Ann*; ;function declaration rule
	|	*Attr_Def*; ;variable declaration rule
S_L_Anno	::=	**"//@ @function "**, Func_Type, [**H** | **L**];
	|	**"//@ @parameter "**, p_Name, Sec_Type, Var_Type, [**H** | **L**];
	|	**"//@ @variable "**, v_Name, Sec_Type, [**H** | **L**];
	|	**"//@ @preStep "**, pr_s_Name, [**H** | **L**];
	|	**"//@ @postStep "**, po_s_Name, [**H** | **L**];
M_L_Anno	::=	[**"/*@ "**], ["* "], Func_Ann, (**" @*/"**)
	|	("*"), [" "]*, (**"@*/"**);
Func_Ann	::=	**"@function "**, Func_Type, [**H** | **L**];
	|	**"@parameter "**, p_Name, Sec_Type, Var_Type, [**H** | **L**];
	|	**"@preStep "**, pr_s_Name, [**H** | **L**];
	|	**"@postStep "**, po_s_Name, [**H** | **L**];
Func_Type	::=	**authentication**;
	|	**declassification**;
	|	**sanitization**;
	|	**sink**;
	|	**source**;
	|	**trust_boundary**;
Sec_Type	::=	**confidential**;
	|	**source**;
Var_Type	::=	**authenticated**;
	|	**declassified**;
	|	**sanitized**;

Figure 4.2: Light-weight annotation language grammar.

The following type face conventions have been used: an italic font for non-terminals and a bold typewriter font for literal terminals including keywords. We included all our main grammar rules under *H_Model*. The rule for *H_Model* contains rules for *S_L_Anno*, *M_L_Anno*, *Func_Ann* and *Attr_Def*. The annotation language grammar has two grammar rules for *S_L_Anno* and *M_L_Anno* used for defining security annotations. The rule for *S_L_Anno* defines single-line annotations; the rule for *M_L_Anno* defines multi-line annotations. Usually multi-line rule annotations are required for annotating C/C++ function declarations. The nonterminal *Func_Ann* corresponds to annotations of C or C++ function declarations. The nonterminal *Attr_Def* corresponds to declarations of variables within the annotation language. (A rule generating *Attr_Def* has been slightly refactored since its introduction in [62]. The details of the rule are too low-level; we are not going to mention them here.) The nonterminal *Var_Type* corresponds to annotating the variable type. This can be either *authenticated*, *declassified* or *sanitized*. The rule for *Sec_Type* is used to annotate the security type of a variable. The rule for *Func_Type* is used to set the type of a function. A function can be tagged as *authentication*, *declassification*, *sanitization*, *source* or *sink*.

4.3 Inference Rules

The aim of the used inference rules is to define a policy whose goal is to prevent the information flow from H (high security level, private) variables to L (low security level, public) variables across trust boundaries. The inference rules are implemented inside our static analysis engine (which, by the way, can handle pointers). Considering the following C *if* statement:

$$\textit{if } a(\mathsf{L}) \leq b(\mathsf{H}) \textit{ then } \ldots \textit{ else } \ldots,$$

where the label L is attached to the variable a and the label H is attached to the variable b. 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 a forbidden flow which we want to detect.

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

Figure 4.3: Secure typing system.

Figure 4.3 depicts the typing system on which our information flow inference rules, depicted in Figure 4.4, are based on. In the row Ⓐ of Figure 4.3, we define the following data types: H and L are used to attach private and public labels to program variables (High/private and Low/public) and *PreStep* and *PostStep* are used to attach function call ordering labels to previous and post function calls. The row Ⓑ of Figure 4.3 presents three types of phrases on which our inference rules are based.
Figure 4.4 depicts secure information flow inference rules which are based on the Denning [19] lattice model and Volpano et al. [92].

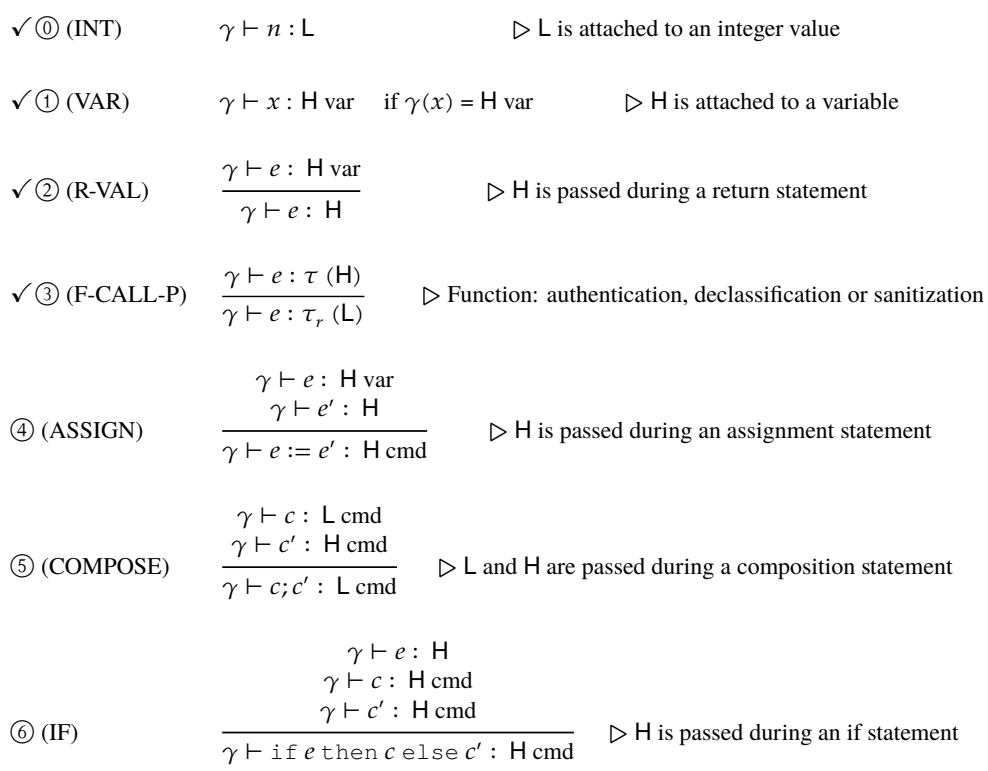

Figure 4.4: An excerpt of typing rules for secure explicit and implicit information flow (✓ means implemented).

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 $\gamma \vdash e : \mathsf{H}$ in ② means that an expression e has security level H (High). The expression $\gamma \vdash e : \tau\ (\mathsf{H})$ in ③ means that if before a function call (*authentication*, *declassification* or *sanitization* function) a variable was tagged with the parameter label H, then after the function call in the symbolic execution the label of the parameter is replaced with L. (E.g., a value stored at address a initially annotated with label H, and *sanitization(a)* leads to marking the new value stored at the same address as L. This new value can be passed to another, safe function.)
Figure 4.4 depicts an excerpt of the used inference rules and how the label(s) are used: ⓪ L is attached to an integer value, ① H is attached to a variable, ② H is passed during a return statement, ③ lowers the label of a variable in memory if it is passed as an input-output parameter of a call to a special security-related function, ④ H is passed during an assignment statement, ⑤ L and H are passed during a composition statement, ⑥ H is passed during an *if* statement. Note that not all inference rules were implemented inside our tools, only the ones marked with (✓).

4.4 UML Statechart Editor

Figure 4.5 depicts an exemplary UML statechart diagram which was modeled inside our UML statechart editor.

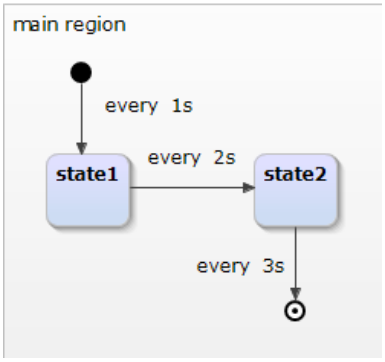

Figure 4.5: UML statechart diagram view in our editor.

Such a diagram in general contains the following components:

- a filled circle, representing the initial state,
- a hollow circle containing a smaller filled circle inside, indicating the final state (if any),
- rounded rectangles, denoting states, and
- arrows, denoting transitions.

Our UML statechart editor is used for creating and editing UML statecharts with the help of our annotation language. Inside the editor it is possible to define UML states. State changes are triggered by events. Events are internal or external factors influencing the system.
Our UML statechart editor can be used to:

- model certain dynamic aspects of a system,
- model the life time of a reactive system,
- describe different states of an object during its life time and
- define a state machine to model the states of an object.

4.5 Source Code Editor

We developed besides the UML statechart editor a textual editor for annotating generated source code files. The main goal of this editor is to allow the addition of new textual annotations into source code in order to fine-tune the already generated annotations. Our source code editor is built upon the Eclipse IDE.
To simplify and speed up editing with our source code editor the following features were added:

- syntax highlighting,
- indentation,
- auto-completion, and
- bracket matching (starting/ending) functionality.

In this research we extended the previous source code editor [64]. The previous editor version offered 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.
The previously available grammar [64] has been extended. The following annotation language tags have been included:

- *authenticated*,

- *declassified*,
- *sanitized*,
- *sanitization*,
- *declassification* and
- *authentication*.

The rules accepting *Func_Ann* (function annotation), *Func_Type* (function type) and *S_L_Anno* (single line annotation) have been extended. A new enumeration type rule for *Var_Type* (variable type) was added. Inside this rule new attributes were included, such as *declassified*, *sanitized* and *authenticated*. New function types were also added, namely *declassification*, *sanitization* and *authentication* functions inside the rule for *Func_Type*. Inside the rules accepting *Func_Ann* and *S_L_Anno* now admit an annotation for a parameter name *@parameter*. Inside a *@parameter* declaration a new attribute was added, namely *Var_Type*.
These new language features appear as code editing suggestions inside our source code editor during interactive editing.

4.6 C Code Generator

We developed a new C code generator based on the C code generator presented in [64]. Eclipse EMF and xTend are used to generate the UML statechart execution code containing the previously added security annotations from UML statecharts.
The code generator outputs two files per UML statechart (a .c file and a .h file). The generated annotations can reside in both header and source code files. Previously annotated UML statechart states are converted to either C function calls or C variables declarations, both need to be previously annotated. The available UML statechart execution flow functionality has been used. This is responsible for traversing the UML statechart during statechart simulation. The UML statechart will be traversed by the code generation algorithm and code is generated based on the mentioned statechart execution flow. The generated code will contain at least one bad path (contains a true positive, a real bug) and a good path (contains no bug, no real bug) per UML statechart. Notice that two paths appear in source code if those were previously explicitly modeled in the UML statechart with our UML statechart editor.
Listing 4.1 depicts the algorithm used to generate C source code from a previously annotated UML statechart.

Algorithm 4.1 ***C code generator algorithm*** ▷ *means code comment*

Input: Statechart
Output: .c and .h files

```
1: method GENERATETYPESH(sc)                                  ▷ Where sc = statechart
2:     def generateFile1(testModule.h, typesHAnnotationContent(sc))   ▷ decl. a func.
3:     def generateFile2(testModule.c, typesCAnnotationContent(sc))   ▷ generator
4: end method
5: method TYPESHANNOTATIONCONTENT(sc)          ▷ method for header file generator
6:     for s : getFileContent(sc).entrySet do                  ▷ iterating hashmap
7:         if s.value.contains('(') and s.key.contains('@') then
8:             println(s.key + '; void ' + s.value + ';')
```

```
 9:          end if
10:      end for
11: end method
12: method typesCAnnotationContent(sc)                    ▷ method for C file generator
13:      for s: getFunctionContent(sc).entrySet do
14:          if (!s.val.contains('authentication') and (!s.val.contains('declassification'))
15:              and (!s.value.contains('sanitization'))) then          ▷ in all cases except three
16:              println('void ' + s.value + ' {}')
17:          end if
18:      end for
19:      for region : sc.regions do
20:          if region.name.equalsIgnoreCase('bad_path()') then        ▷ get bad path contents
21:              print('void ' + region.name + '{')
22:              for s: getBadPathContent(sc).entrySet do
23:                  if s.key.contains('//@ @variable') then    ▷ checking variable annotations
24:                      println(s.key + ' ' + s.value + ';')
25:                  end if
26:                  if s.value.contains('(') then              ▷ checking function declarations
27:                      println(s.value + ';')
28:                  end if
29:              end for
30:              println('}')
31:          end if
32:          if region.name.equalsIgnoreCase('good_path()') then       ▷ get good path content
33:              println('void ' + region.name + '{')
34:              for s: getGoodPathContent(sc).entrySet do      ▷ get statements and comments
35:                  if s.key.contains('//@ @variable') then      ▷ check var., get the comments
36:                      println(s.key + ';')
37:                  end if
38:                  println(s.value + ';')
39:              end for
40:              println('}')
41:          end if
42:      end for
43: end method
```

The input of the algorithm is an UML statechart. In xTend the *def* keyword defines a function. The Algorithm 4.1 starts with the method *generateTypesH*; its input *sc* is the UML statechart. The plug-in named *MyC* uses xTend to parse the UML statechart. This method calls another are two methods: *typesHAnnotationContent* to generate the header file, and *typesCAnnotationContent* to generates the source code. The method *typesHAnnotationContent* generates the required contents of the C header file. The contents is the function signature and the annotation of the function which was previously in the UML statechart diagram.
The method *typesCAnnotationContent* generates the required contents of the C source file. This file contains the annotation only for variable declarations (the function annotation is usually

located in the header file). In addition to what the generator Algorithm 4.1 creates, some other C code is added into the C source file (e.g., "#include"s, the "main" function, etc.).
The method *typesHAnnotationContent* is used to obtain the function annotation which was previously added with the help of the textual editor. This annotation will be placed by the code generator before the function signature contained in the header file. We declared a method named *getFileContent* which returns a hashmap containing annotations and statements e.g., variable declaration and function signatures. By iterating through the hashmap we make a check which detects whether the current statement is not a variable annotation. If it is the case, then we place the annotation and function signatures into the C header file.
The method *typesCAnnotationContent* is responsible for the generation of the C source code. Inside this function in order to get all function content there is a method called *getFunctionContent* which returns a hashmap with all function signatures and annotations. By iterating through this hashmap the required function signatures are placed inside the C code file. The functions whose signatures have annotations but whose bodies are empty are placed into the C source code file. The annotations of the functions are mainly placed into the header files.
The system was designed with two regions: *good_path()* and *bad_path()*. To get the content of *good_path()* and *bad_path()*, two methods have created inside the *.xtend* file, namely *getGoodPathContent()* and *getBadPathContent()*. Those two methods return two hashmaps, respectively. One hashmap contains the contents of *good_path()* and another hashmap contains the contents of *bad_path()*. Both hashmaps contain the function signatures, statements of C/C++ language and annotations. Usually the *good_path()* and *bad_path()* regions contain no function annotations, only the variable annotations and other statements such as function calls and variable declarations. The functions annotations are placed into the C/C++ header files during code generation.

4.7 Static Analysis Checkers

We developed three static code analysis checkers for the three types of bugs which we want to detect (*declassification*, *sanitization* and *authentication*). We used the static analysis engine, Smtcodan, and extended it with the functionality of the three new checkers.
We added the following files inside our engine: *AuthenticationFunctionChecker.java*, *DeclassificationFunctionChecker.java* and *SanitizationFunctionChecker.java*. Each of the classes in these files represents a checker containing the logic needed to detect the corresponding bugs. Next, Smtcodan was extended with three Java classes representing suitable models of the three function types: *Authentication_gen.java*, *Declassification_gen.java*, *Sanitization_gen.java*.
The generated C code with annotations represents the input for Smtcodan static analysis engine. The engine is separate from the code generator and thus involves parsing the code with annotations. Inside the engine, the models of the *authentication*, *declassification* and *sanitization* functions mark the H-typed secured variables or confidential variables as L; 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 have been defined and used inside our engine. If any of the program paths contained in a C program is not following the secure information flow policy enforced with our annotation language then a bug report should be triggered. This happens when for example either *authentication*, *declassification* or a *sanitization* function is missing.

4.8 Buggy Paths Display

We developed a UML sequence diagram generator and viewer. The goal is to represent a buggy path as a UML sequence diagram such that a user can easily trace the location (file name and line number) of the bug. Listing 4.2 depicts the algorithm used to generate a UML sequence diagram from a buggy path.

Algorithm 4.2 ***Sequence diagram generator*** ▷ *means code comment*

Input: List of statements and function calls belonging to the buggy path
Output: Sequence diagram represented inside a frame

```
1:  method drawSequenceDiagram(ArrayList<IASTNode> statementsList)
2:      …                                              ▷ Initialize the output frame:
3:      mcd := MyCanvasDraw(…);
4:      …
5:      for i from 0 to statementsList.size() − 1 do
6:          fnName := …                          ▷ get the file name of the i^th statement.
7:          if statementsList.get(i).getRawSignature().toString().contains("(") then
8:              …                                ▷ output function call avoiding repetitions.
9:          else
10:             allStatements := "";
11:             j := i;
12:             if j ≤ statementsList.size() − 2 then
13:                 repeat
14:                     lnNo := ….getStartingLineNumber();     ▷ obtain the j^th line number.
15:                                  ▷ Format the location of the j^th statement properly:
16:                     allStatements := allStatements + … + lnNo + … + " "
17:                                                  + … + fileName + … + "\n";
18:                     j := j+1
19:                 until statementsList.get(j).getRawSig().toString().contains("(");
20:                                          ▷ Add all statements formatted in the above loop:
21:                 mcd.buggyPathList.add(allStatements)
22:             end if
23:         end if
24:     end for;
25:                                      ▷ Format the output window properly and draw it:
26:     …
27:     mcd.paint(…)
28:     …
29: end method
```

The class *SequenceDiagramGenerator* is responsible for generating UML sequence diagrams. The method *drawSequenceDiagram* of this class actually creates the diagrams. The method input parameter is a list of *IASTNode*s. (An object typed as *IASTNode* is a node of the abstract

syntax tree.) We iterate through the list of *IASTNodes* and equip all statements except the function calls with line number and file name. After the list was filled a diagram will be drawn accordingly to contents of the list. For visualizing the diagram we employ a *JFrame* object with a *JScrollPane* object, which makes the frame scrollable (both not shown above for he sake of clarity). The default image format for saving the output is *.jpg*; other supported formats are *.png*, *.bmp* and *.gif*.
Figure 4.6 depicts a buggy program execution path as a UML sequence diagram.

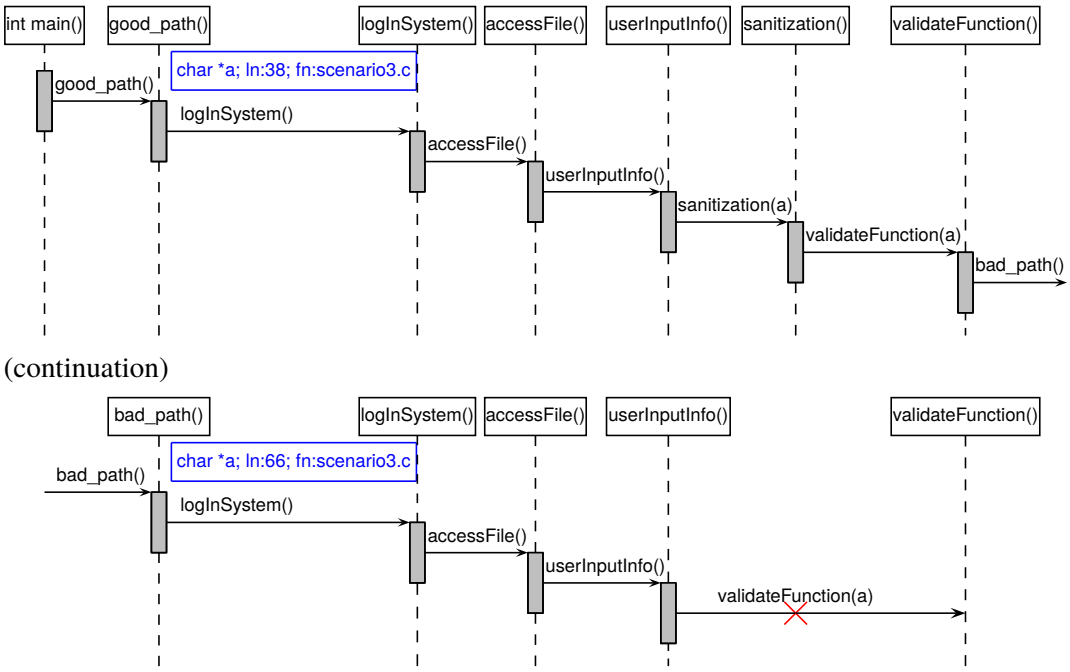

Figure 4.6: Buggy path trace displayed as UML sequence diagram.

Such a path is generated by Algorithm 4.2. The algorithm creates a list of elements which should be displayed in the UML sequence diagram. Such a list contains:

- function calls,
- if statements,
- switch-case statements,
- variable declaration and
- assignment of variables.

Inside the sequence diagram all function calls are included inside rectangular boxes attached to the head of each lifeline. The messages between lifelines represent function calls with arguments. A function call moves from one lifeline to the next one. The statements of a function body which precede the function calls are typeset as blue boxes attached to the corresponding lifelines. For example, in the Figure 4.6 from the function call *int main()* there is a transition to *good_path()*. Each transition inside the UML statechart is reflected in code as a function call. This will help the user to trace the buggy path more easily.
Inside the UML sequence diagram each statement which is located before a function (on a program path) call is attached to a lifeline and depicted with blue color. For example the text depicted with blue color in Figure 4.6, "*char *c; ln:38; fn:scenario3.c*", attached to the lifeline

of *good_path()* function means that the declaration *char *a;* is located at line number (ln) 38 in the file named (fn) *scenario3.c*. The red cross shows the error location.

5 Experiments

This chapter presents examples of how our approach can be used to detect bugs, including an authentication scenario in § 5.1, a declassification scenario in § 5.2, and a sanitization scenario §,5.3. We show how our checkers run on the generated code for the sanitization scenario in § 5.4 and present a UML sequence diagram for the sanitization scenario in § 5.5. Next, for each of the *authentication*, *declassification* and *sanitization* scenarios a textual description will be first given and afterwards the scenario will be remodeled with our UML statechart editor. A state in our UML statechart can be any C language declaration or function; a transition represents a function call or passing a variable. After finishing editing with our UML statechart editor, C code files will be generated. The previously added annotations will reside inside the generated files as textual annotations. Next, the code is analyzed using our static analysis checkers, and bug reports will be generated if needed.

5.1 Authentication Scenario

We illustrate how our approach deals with an authentication scenario on a real-life example. We selected an open-source Java-code example for accessing a database in order to highlight what kinds of vulnerabilities this code may exhibit. In case that no authentication of a given occurs, no database access should be granted for the user.
Listing 5.1 includes a Boolean variable *isUserAuthentic* and a method for authenticating an user.

Listing 5.1: Java code example for authentication scenario.

```
private boolean isUserAuthentic = false; // authenticate the user;
// if the user gets authenticated, set to true, otherwise to false
public boolean authenticateUser(String username, String password) {}
public DBAccess createUserAccess(String userName, String userType,
    String userPassword) {
   ...
   if (isUserAuthentic) {
     access.setUserName(userName);
     access.setUserType(userType);
     access.setUserPassword(userPassword);}
   return access;
}
```

If the user has not been first authenticated, *createUserAccess()* should not create the database access object. Such checks are often easily forgotten in production code. This kind of errors can be avoided by modeling the authentication scenario during the design phase using UML state charts.
Figure 5.1 depicts an authentication scenario were we remodel the previous given code from Listing 5.1 as a UML statechart.

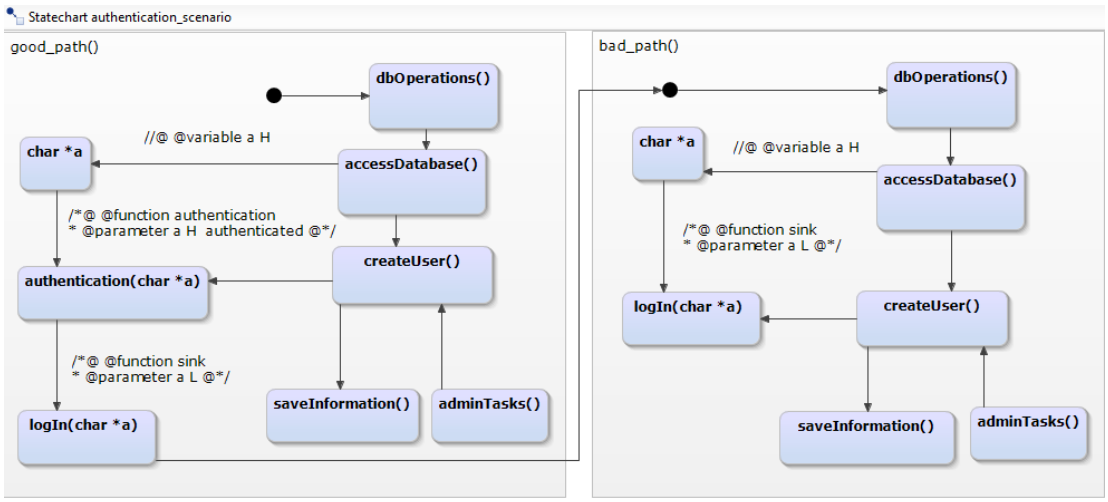

Figure 5.1: UML statechart modeling for the authentication scenario.

Notice an arrow from *logIn(char *a)* to the beginning of *bad_path()*: in this example (as in all our examples) we combine the traversals of *good_path()* and *bad_path()*. This way we check that the analysis detects errors where they exist but does not report errors where they do not exist.

In this scenario a user *A* wants to access a database. First, the user has to provide his/her id, password and account number. Then, the user sends a request to access the database. The database administrator creates a new access token using his id, name and password based on his policy. According to the policy, a user can get either a view-only access or a full access or no access to the database.

Figure 5.1 depicts a high-labeled variable (the left-hand side of Figure 5.1) *char *a* which is initially annotated as H. The annotation is attached to the box *char *a*. A transition from the box *accessDatabase()* to the box *char *a* is added; the transition gets labeled by *//@ @variable a H*. The variable *char *a* passes through the function *authentication(char *a)*. This authentication function is represented as a box inside the UML statechart as *void authentication(char *a)*. The annotation

/@ @function authentication**

* * @parameter a H authenticated @*/.*

is attached to the transition from the box *char *a* to the box *void authentication(char *a)*. This function makes the high-labeled variable (*a*) low by applying the policy rules. After passing this function the variable *a* is annotated with L and the tag *authenticated*.

In this scenario inside *good_path()* there is no bug because the data flow passes through an authentication function. After authentication, the data can be securely passed to other systems or released to other users.

While switching from *void authentication(char *a)* to the *void logIn(char *a)* function there is another annotation

/@ @function sink*

* * @parameter a L @*/.*

This *void logIn(char *a)* function is a sink function, which expects that its parameter was previ-

ously set to low. If the parameter does not pass through the authentication function, it remains H (High). In this case a bug report should be triggered.
Inside the execution path *bad_path()* (the right-hand side of Figure 5.1) a bug should be triggered because there is no authentication function. The role of the authentication function would be to set the type of the variable previously annotated by H to L. Notice that this does not happen on this execution path. Thus, a bug report should be generated.

5.2 Declassification Scenario

We have chosen a real-life example to demonstrate how our approach deals with a declassification scenario. Consider a user A who wants to access his/her bank account. After A has provided his/her password and account number, A sends the request to the bank server to view the account information. At the bank server A's access policy is stored. The request will be checked against the policy. The server will check next the following declassification goals according to: (i) which information is released, (ii) who releases the information, (iii) where in the system the information is released, and (iv) when is the information released.
Figure 5.2 depicts the previously described declassification scenario in a simplified manner and remodeled with our UML statechart editor.

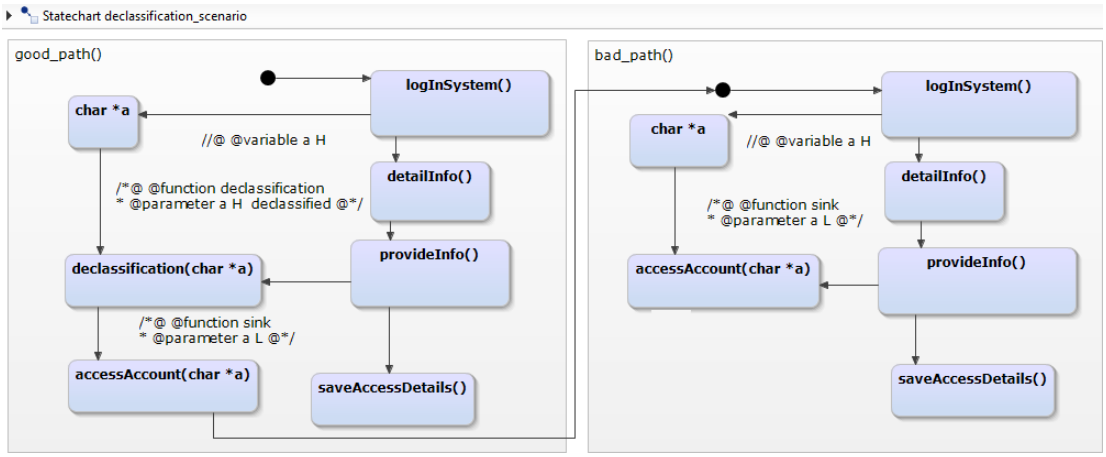

Figure 5.2: UML statechart model of the declassification scenario.

Again, there is an arrow from *accessAccount(char *a)* to the beginning of *bad_path()* because we want test the prototype against a good and a bad scenario in the same run.
Consider the same user A as before; A wants to access his/her bank account. Through A's request the method *logInSystem()* will be called, and a variable *a* will be passed to a declassification function. Figure 5.2 depicts a high-labeled variable *char *a* (the left-hand side of Figure 5.2) which is initially annotated as H. The box *char *a* has an incoming transition from state *logInSystem()*. That transition is annotated by "*//@ @ variable a H*". The variable *a* passes through a declassification function. This declassification function is represented as a box *void declassification(char *a)* in the UML statechart.
The declassification function is annotated by
/@ @function declassification*

* *@parameter a H declassified @*/.*

The declassification function marks the high, secured variable as low according to the policy rules. After passing this function the variable *a* is annotated with L and the tag *declassified*. In the *good_path()* there is no bug because the variable *a* passes through a declassification function. After the declassification function, the type of the variable *a* is lowered from H (High) to L (Low).

Next, the function *accessAccount* can be executed securely. The function *accessAccount(char *a)* expects a parameter which is L (Low). If the parameter is indeed low, there is no bug.

The *bad_path()* (depicted on the right-hand side of Figure 5.2) contains no declassification function. Thus, the variable *a* remains H (High). In *bad_path()* the box *accessAccount(char *a)* gets the parameter as labeled H (High). As a result, a bug report will be triggered.

5.3 Sanitization Scenario

We demonstrate our approach of dealing with sanitization on a real-life scenario.

Most of the web applications are developed by programmers which are not security-aware. Thus, it is easy to discover vulnerabilities in this situation. A major source for vulnerabilities is based on the missing input-validation. Web applications use input which can be malicious for sensitive procedures. In many real-life applications this input is not previously sanitized.

Consider a user *A* who wants to access some file from a server. First, he needs to provide the file name: e.g., he wants to access an *.exe* file. He could insert an OS command injection. Thus, user input should pass first through a sanitization function such that the input parameter(s) will be sanitized (such that their types could be soundly lowered from H to L).

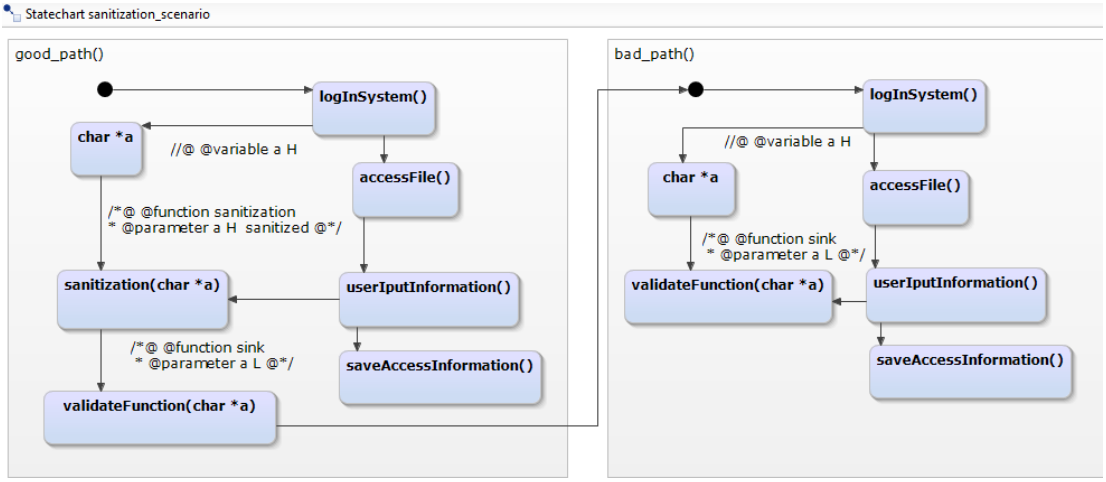

Figure 5.3: UML statechart model of the sanitization scenario.

Figure 5.3 depicts a similar sanitization scenario as before. We remodel the aforementioned textual description in a simplified manner using an UML statechart diagram inside our UML statechart editor. The used input has to go first through a sanitization function in order to become properly cleansed.

Notice that there is an arrow from *validateFunction(char *a)* to the beginning of *bad_path* because we want to ensure that the analysis traverses a properly typed and an ill-typed case in the

same run, thereby checking the quality of our analysis.
Consider the user A who wants to access a file located on a remote server. First, A needs to provide the name of the file he/she wants to access. Then, A sends a request to the server in order to access the desired file. Next, the *logInSystem()* function labels variable *char *a* as H

- and passes *a* to a sanitization function on the left-hand side of Figure 5.3
- but does not do that on the right-hand side of Figure 5.3.

After that, *validateFunction(char *a)* is called.
Figure 5.3 depicts a high labeled variable, *char *a*, which is initially annotated as H. It passes through a sanitization function, *void sanitization(char *a)* on the left-hand side of Figure 5.3. This function marks the H-labeled variable as L when following the previously enforced policy. After passing this function the variable *char *a* is annotated with L and the tag *sanitized*. Now it is safe to pass the information to another data-processing function or to release it outside the system.
In the *good_path()* on the left-hand side of Figure 5.3 a sanitization function exist. During symbolic execution, this function changes the security-type of the variable *a* from H to L. The function *validateFunction* expects and gets the parameter *a* marked as L. There is no bug in the *good_path()*.
In the *bad_path()* on the right-hand side of Figure 5.3 no sanitization function is used. Thus, *validateFunction* gets the parameter *a* as H, although it expects L. In this case a bug should be triggered on this path; the cause is the absence of a sanitization function.

5.4 Static Analysis Checkers

In this section we briefly describe implementation details of our checkers by running them on the previously created UML statecharts.

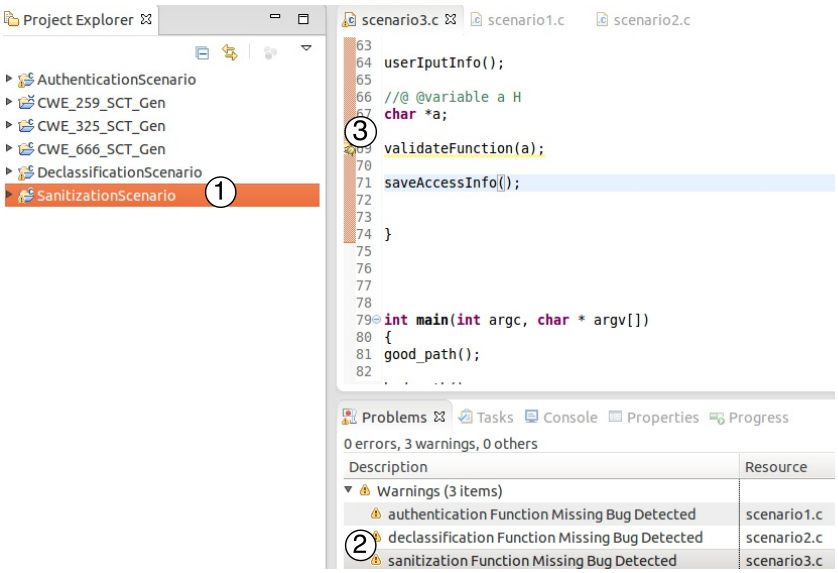

Figure 5.4: Checker's view of bug reports.

Figure 5.4 depicts the three C programs (which were generated by our C source code generator)

added into Eclipse CDT projects. These projects were imported into an Eclipse CDT workspace. The available checkers are started on each of the projects by clicking right and selecting *Run as C/C++ code analysis*. In case a bug is detected, it will be shown in the Problems view. Figure 5.4 depicts the bug reports obtained by running the checkers in parallel on the generated programs. The circled numbers depicted in Figure 5.4 indicate the following: ① indicates the analyzed programs (generated programs), ② indicates the bug reports associated to each of the found bugs, ③ indicates the location (line number 69) where the bug was located.

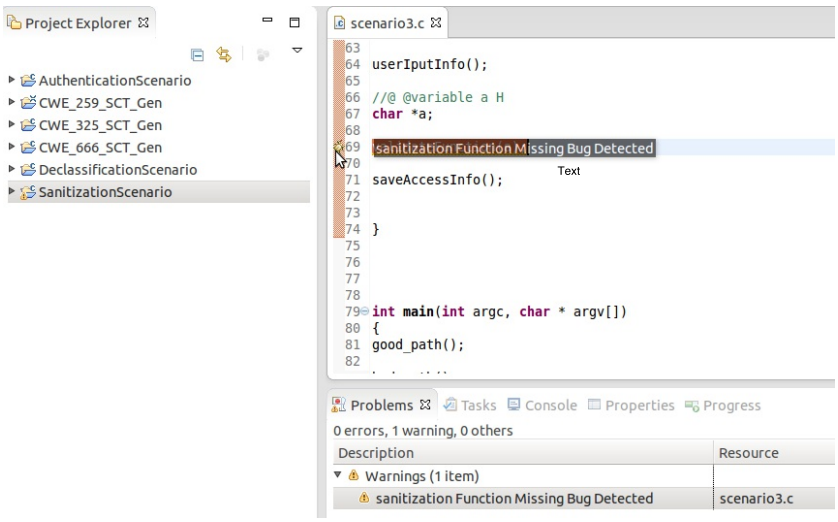

Figure 5.5: Checker producing a sanitization bug report with message.

Figure 5.5 depicts a sanitization bug with the associated message. If the user clicks the bug icon associated to the bug location, a bug message will be displayed. By clicking the bug symbol in line 69 the message *sanitization Function Missing Bug Detected* will be displayed. This helps the user to better visually link the bug location to its type.

5.5 Error Tracing

Figure 4.6 represents an error trace as a UML sequence diagram. In the source code, *good_path()* is traversed before *bad_path()*.
In this experiment we ran our sanitization checker on the generated code for the sanitization scenario. Each lifeline represents entering a new function. Each arrow connecting a lifeline to the next one is part of the *chain* of function calls belonging to the buggy program path.
The error trace is displayed only after the bug detection phase is over and at least one buggy path was found. Function calls, variable declarations and other types of statements will be included inside the UML sequence diagram. The goal of displaying the path as an UML sequence diagram is to represent the error path graphically in order to help the user to find the bug cause faster. The error location is marked with a red cross, ×.

6 Related Work

We overview

- sanitization approaches in § 6.1,
- declassification techniques in § 6.2,
- authentication tools in § 6.3,
- source code annotation languages in § 6.4,
- approaches to detect vulnerabilities based on taint-style detection in § 6.5,
- static analysis tools in § 6.6,
- dynamic analysis techniques in § 6.7 and
- hybrid analysis tools in § 6.8.

The detection of information flow vulnerabilities [60] can be addressed with

- dynamic analysis techniques [5, 25–27, 75],
- static analysis techniques [32, 65, 83, 92, 101] (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 [8, 38, 61].

Additionally, extended static checking [20] (ESC) is a promising research area which tries to cope with the shortage of not having certain program runtime information.

6.1 Sanitization

Weinberger et al. [94] empirically studied sanitization approaches against XSS in web application frameworks. 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.
Samuel et al. [77] proposed a template based framework which uses type qualifiers to automate context-sensitive XSS sanitization.
Saner [8] is a an approach used for the evaluation of the sanitization process in web applications. The approach relies on two complementary analysis techniques to identify faulty sanitization procedures. The dynamic technique used by Saner can reconstruct source code which is used for sanitization of input. Faulty sanitization routines are detected by running the reconstructed source code with malicious inputs. By applying it to real-world applications it identified novel vulnerabilities that stem from incorrect or incomplete sanitization.

6.2 Declassification

Hicks et al. [36] present an approach in which declassifier functions are part of an global policy used for trusted declassification. Individual principals are declaratively specifying 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 the Java programming

language and give proofs for a version of noninterference called noninterference modulo trusted methods. Their approach is based on extending Jif. They use this approach to build an e-mail client which they claim to be secure.

6.3 Authentication

DIFC (decentralized information flow control) [6] is integrated into a Platform as a Service (PaaS) cloud model which can be tested by augmenting existing open-source implementations such as VMware *Cloud Foundry* and Red Hat *OpenShift*. DIFC has can be used to protect user-data integrity and secrecy.

6.4 Annotation Languages

Many annotation languages have been proposed for making applications more secure by

- extending the C type system to be used during running time [17, 22, 57, 58, 90],
- augmenting a programming language such as PHP and Python and the runtime of this language [102],
- annotating function interfaces [22, 57, 90],
- annotating models in order to detect information flow bugs [47],
- annotating source code files [71, 72, 88] and
- annotating control flows [22, 24, 57].

The following annotation languages have made a significant impact. Microsoft SAL annotations [57] helped to detect more than 1000 exploitable vulnerabilities in Windows code [7]; Jif [14], AURA [41], FlowCaml [82, 83], FINE [86], and Fable [87] can be used to express information flow related concerns and have proved to be highly usable for helping to detect information flow bugs.
Recently taint modes have been integrated in programming languages such as Caml-based FlowCaml [84] and Ada-based SPARK Examiner [11]. However, none of these annotation and programming languages have support for introducing information flow restrictions in both models and the source code such as our approach has.

6.5 Taint-style Detection of Vulnerabilities

Huang et al. [38] adapted parts of the techniques used in `cqual` to develop an intra-procedural analysis for PHP programs. In [39], the same authors presented an alternative approach that is based on bounded model checking.
Whaley et al. [96, 97] present a context-sensitive pointer analysis used for inter-procedural analysis of Java programs. Binary decision diagrams are used during the analysis similar to Livshits et al. [52] which have used it to detect taint-style vulnerabilities.
Balzarotti et al. [8] propose an approach based on taint-style vulnerabilities detection using the tool Pixy [44, 45]. Pixy is an open source static PHP analyzer that uses taint analysis for detecting XSS vulnerabilities.

6.6 Static Analysis

The static code analysis techniques need to *know* which parts of the code are sinks, sources, and which variables should be tagged. A practical solution for tagging these elements in source code is based on a pre-annotated libraries which contain annotations attached to function declarations. Leino [49] reports about the annotation burden as being very time consuming and disliked by some programming teams. This applies to the annotation languages [17, 22, 24, 47, 57, 58, 71, 72, 88, 90, 102] mentioned in § 6.4.
These studies rely on manually written annotations, while our annotation language is integrated into two editors which are used to annotate UML statecharts and C code by selecting annotations from a list and without the need to memorize the tags of the annotation language.
Chess et al. [13] present an overview of static analysis tools. Tools such as ITS4 and RATS [100] use predefined patterns to detect potentially dangerous areas of a program. These tools are superior to grep-based tools but still do not have any knowledge how data is propagated inside a program; thus these tools can not be used in combination with taint-style techniques. There are several approaches [10, 33, 53] which use path-sensitive analysis in order to find errors in C/C++ programs. These tools are usable for taint-style problems but are unsound w.r.t. pointer analysis and thus miss certain errors.

6.6.1 Source-Code–Based Analysis

Evans et al. [23] propose the Splint tool, Wheeler et al. [98] propose the Flawfinder tool, and Shankar et al. [79] propose the `cqual` tool. These tools are used to detect information flow bugs in source code and have a comprehensive user manual which describes how the provided annotation language can be used in order to annotate source code.
Darvas et al. [18] used a theorem-proving approach to prove or disprove secure information flow properties for Java CARD programs. They employed an interactive theorem prover KeY (instead of an automatic one).
Barthe et al. [9] present mostly theoretical results on characterizing various secure information flow problems, including non-deterministic and termination-sensitive cases in a self-compositional framework. They showed that their self-compositional framework can handle delimited information release as originally proposed by Sabelfeld et al. [73].
Minamide et al. [59] propose an algorithm for string-based analysis. The goal is to syntactically isolate tainted substrings from untainted substrings in PHP programs. They label non-terminals in a Context-Free Grammar (CFG) with annotations used for taintedness and untaintedness analysis. Their approach is used for detecting XSS and has a relatively high performance overhead. This approach is further extended and applied by Wassermann [93].
Chaudhuri et al. [12] developed Rubyx used for performing symbolic execution of Ruby-on-Rails applications based on an assume/assert language. The authors analyzed small applications (up to 20k LOC and running time up to 3 minutes) and claim that symbolic execution is a promising avenue for further analyzing web applications.
Li et al. [50] proposed relaxed non-interference. This 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.

Liu et al. [51] propose a new general-purpose static analysis used for the inference of explicit information flows. Their analysis is light-weight, works on Java programs and requires no user annotations. Their approach can be easily integrated in verification tools to verify integrity and confidentiality of sensitive program data.
Zheng et al. [105] introduced path-sensitive static analysis used for PHP applications based on the Z3-str. They leveraged a modified version of the Z3 SMT solver that is also capable of analyzing strings.
Shar et al. [80, 81] proposed static code attributes for predicting SQLi and XSS vulnerabilities.
Yu et al. [104] built an automata-based string analysis tool called Stranger based on the static code analysis tool Pixy [43]. 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 [103].

6.6.2 Model-Based Analysis

UMLSec [46] is a model-driven approach that allows the development of secure applications with UML. Compared with our approach, neither does UMLSec include automatic code generation, nor can the annotations be used for automated constraints checking.
Heldal et al. [34, 35] introduced an UML profile that incorporates a decentralized label model [67] into the UML. It allows the annotation of UML artifacts with Jif [66] labels in order to generate Jif code from the UML model automatically. However, the Jif-style annotation already proved to be non-trivial on the code level [69], while [35] 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 tools for automated checking of previously imposed constraints.
IFlow [47] 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 [48]. 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.

6.7 Dynamic Analysis

McCamant et al. [56] propose an quantitative approach w.r.t. data information flow. They view information-flow analysis as a network flow problem with capacities. They also present a dynamic technique used to measure the data which public observers could obtain through leaks.
TAJ [91] is a taint analysis suitable for industrial applications. An experimental evaluation indicates that the hybrid thin-slicing algorithm inside TAJ is a good 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.
RESIN [102] can be used for constructing web applications to prevent a range of problems, such as XSS and SQL injection, inadvertent password disclosure and missing access control checks. RESIN is based on assertions which are added to the code. Three previously unknown missing

access control vulnerabilities in phpBB could be prevented through the usage of RESIN. The main disadvantage of RESIN is its overhead (33% slowdown when running HotCRP).

6.8 Hybrid Analysis

WebSSARI [38] uses unsound concolic analysis in order to analyze PHP applications. WebSSARI proved to be useful in finding XSS and SQL injections vulnerabilities in several open-source PHP applications.
Moore et al. [61] present two methods to use static analysis to increase the efficiency of hybrid information-flow monitors. First, they reduce the running-time overhead by statically determining a possibly small subset of variables whose security level has to be tracked by a sound monitor during program executions. The second method is based on deriving sufficient conditions for soundly incorporating a variety of memory abstractions into a monitor for languages with dynamically allocated memory.
Balzarotti et al. [8] use a hybrid analysis technique in order to find faulty routines which are used for input sanitization. The static analysis component of their tool, Saner, extends Pixy and analyzes string modification with automata. The dynamic component is used to check the analysis results in order to reduce the number of false positives.

7 Conclusion

Our goal is further automation of the detection of information-flow bugs. We extended a keyword-based annotation language that can be used out of the box for annotating UML statecharts and C code in two software development phases by providing two editors for inserting security annotations. We evaluated our approach on real-life programs and showed that our approach is applicable to real scenarios.
To the best of our knowledge our annotation language is the first light-weight annotation language usable for specifying information-flow security constraints which can be employed in the design and coding phase for the detection of information-flow bugs.

8 Publication List

1. *Automated Detection of Information Flow Vulnerabilities in UML State Charts and C Code*, P. Muntean, A. Rabbi, A. Ibing, and C. Eckert. In IEEE International Workshop on Model-Based Verification & Validation (MVV), 2015, IEEE, [64].

2. *Semi-Automated Detection of Sanitization, Authentication and Declassification Errors in UML State Charts*, A. Rabbi, Master Thesis, Technische Universität München, 2015, [70]

Bibliography

[1] itemis AG. *Yakindu Open-Source Statechart Tools*. URL: https://www.itemis.com/en/yakindu/statechart-tools/ (visited on 08/08/2016).

[2] Iván Arce, Kathleen Clark-Fisher, Neil Daswani, Jim DelGrosso, Danny Dhillon, Christoph Kern, Tadayoshi Kohno, Carl Landwehr, Gary McGraw, Brook Schoenfield, Margo Seltzer, Diomidis Spinellis, Izar Tarandach and Jacob West. *Avoiding the top 10 software security design flaws*. Tech. rep. IEEE Center for Secure Design, Aug. 2014. URL: https://www.computer.org/cms/CYBSI/docs/Top-10-Flaws.pdf.

[3] *Authentication*. URL: https://en.wikipedia.org/wiki/Authentication (visited on 08/03/2016).

[4] *Authorization*. URL: https://en.wikipedia.org/wiki/Authorization (visited on 08/03/2016).

[5] Thanassis Avgerinos, Sang Kil Cha, Brent Lim Tze Hao and David Brumley. "AEG: Automatic Exploit Generation." In: *Proceedings of the Network and Distributed System Security Symposium (NDSS)* (Feb. 2011).

[6] Jean Bacon, David Eyers, Thomas F. J.-M Pasquier, Jatinder Singh, Ioannis Papagiannis and Peter Pietzuch. "Information Flow Control for Secure Cloud Computing." In: *IEEE Transactions on Network and Service Management* 11.1 (2014), pp. 76–89.

[7] Thomas Ball, Brian Hackett, Shuvendu Lahiri and Shaz Qadeer. *Annotation-based Property Checking for Systems Software*. Tech. rep. Microsoft, May 2008.

[8] Davide Balzarotti, Marco Cova, Vika Felmetsger, Nenad Jovanovic, Engin Kirda, Christopher Kruegel and Giovani Vigna. "Saner: Composing Static and Dynamic Analysis to Validate Sanitization in Web Applications." In: *IEEE Symposium on Security and Privacy (S&P)*. IEEE. 2008, pp. 387–401.

[9] Gilles Barthe, Pedro Ruben D'Argenio and Tamara Rezk. "Secure information flow by self-composition." In: *Proceedings of the Computer Security Foundations Workshop (CSFW)*. IEEE. June 2004, pp. 100–114. ISBN: 0-7695-2169-X.

[10] William R. Bush, Jonathan D. Pincus and David J. Sielaff. "A Static Analyzer for Finding Dynamic Programming Errors." In: *Software—Practice & Experience* 30.7 (2000), pp. 775–802.

[11] Roderick Chapman and Adrian Hilton. "Enforcing Security and Safety Models with an Information Flow Analysis Tool." In: *ACM SIGAda* 24.4 (2004).

[12] Avik Chaudhuri and Jeffrey Foster. "Symbolic Security Analysis of Ruby-on-Rails Web Applications." In: *Proceedings of the 17th ACM Conference on Computer and Communications Security (CCS)*. ACM, 2010, pp. 585–594.

[13] Brian Chess and Gary McGraw. "Static Analysis for Security." In: *IEEE Security & Privacy (S&P)* 6 (2004), pp. 76–79.

[14] Stephen Chong, Andrew Clifford Myers, Nate Nystrom, Lantian Zheng and Steve Zdancewic. *Jif: Java + Information Flow*. Software release. July 2006. URL: http://www.cs.cornell.edu/jif.

[15] Ellis Saul Cohen. "Information Transmission in Computational Systems." In: *Proceedings of the sixth ACM symposium on operating systems principles (SOSP)*. West Lafayette, Indiana, USA: ACM, 1977, pp. 133–139. DOI: 10.1145/800214.806556.

[16] Ellis Saul Cohen. "Information Transmission in Sequential Programs." In: (Dec. 1978). Ed. by Richard A. DeMillo, David P. Dobkin, Anita K. Jones and Richard J. Lipton, pp. 301–339. URL: https://smartech.gatech.edu/bitstream/handle/1853/40598/g-36-619_142482.pdf?sequence=1.

[17] Jeremey Condit, Matthew Harren, Zachary Anderson, David Gay and George Ciprian Necula. "Dependent types for low-level programming." In: *ESOP*. 2007.

[18] Ádám Darvas, Reiner Hähnle and David Sands. "A Theorem Proving Approach to Analysis of Secure Information Flow." In: *Security in Pervasive Computing*. Springer, 2005, pp. 193–209.

[19] Dorothy Elizabeth Robling Denning. "A Lattice Model of Secure Information Flow." In: *Communications of the ACM* 19.5 (1976), pp. 236–243.

[20] David L. Detlefs, K. Rustan M. Leino, Greg Nelson and James B. Saxe. "Extended Static Checking." In: *Compaq SRC Research Report 159* (1998).

[21] Eric Elliot. *Programming JavaScript Applications*. O'Reilly, June 2014. ISBN: 978-1-4919-5029-6.

[22] David Evans. "Static Detection of Dynamic Memory Errors." In: *SIGPLAN Conference on Programming Language Design and Implementation (PLDI)* (1996).

[23] David Evans and David Larochelle. "Improving Security Using Extensible Lightweight Static Analysis." In: *IEEE Software* (Jan. 2002).

[24] David Evans and David Larochelle. *Splint - Manual*. Online available at: http://www.splint.org/manual/html/sec8.html.

[25] Jeffrey S. Fenton. *An abstract computer model demonstrating directional information flow*. 1974.

[26] Jeffrey S. Fenton. "Information protection systems." PhD thesis. University of Cambridge, 1973.

[27] Jeffrey S. Fenton. "Memoryless subsystems." In: *Computer Journal* 17.2 (Jan. 1974), pp. 143–147.

[28] The Eclipse Foundation. *xTend Documentation*. Online available at: http://www.eclipse.org/xtend/documentation/.

[29] The Eclipse Foundation. *xText Documentation*. vhttp://www.eclipse.org/Xtext/documentation.html.

[30] Roberto Giacobazzi and Isabella Mastroeni. "Adjoining Declassification and Attack Models by Abstract Interpretation." In: *Programming Languages and Systems*. Vol. 3444. Lecture Notes in Computer Science. Springer, 2005, pp. 295–310.

[31] Pablo Giambiagi and Mads Dam. “On the Secure Implementation of Security Protocols.” In: *Programming Languages and Systems (ESOP)*. Vol. 2618. Lecture Notes in Computer Science. Springer, Feb. 2003, pp. 144–158.

[32] Marco Guarnieri, Paul El-Khoury and Gabriel Serme. “Security Vulnerabilities Detection and Protection Using Eclipse.” In: *ECLIPSE-IT, 6th Workshop of the Italian Eclipse Community* (Sept. 2011).

[33] Seth Hallem, Benjamin Chelf, Yichen Xie and Dawson Engler. “A system and language for building system-specific, static analyses.” In: *Proceedings of the ACM SIGPLAN 2002 conference on Programming language design and implementation (PLDI)*. Vol. 37. 5. ACM, 2002.

[34] Rogardt Heldal and Fredrik Hultin. “Bridging Model-based and Language-based Security.” In: *(ESORICS)*. Lecture Notes in Computer Science 2808 (2003). Ed. by E. Snekkenes and D. Gollmann. Online available at: http://dx.doi.org/10.1007/978-3-540-39650-514, pp. 235–252.

[35] Rogardt Heldal, Steffen Schlager and Jakob Bende. *Supporting Confidentiality in UML: A Profile for the Decentralized Label Model*. Tech. rep. TUM-I0415. Technische Universität München, 2004, pp. 56–70.

[36] Boniface Hicks, Dave King, Patrick McDaniel and Michael Hicks. “Trusted Declassification: High-level Policy for a Security-typed Language.” In: *Programming Languages and Analysis for Security (PLAS)*. ACM, 2006, pp. 65–74. DOI: 10.1145/1134744.1134757.

[37] Boniface Hicks, David King and Patrick McDaniel. *Declassification with Cryptographic Functions in a Security-typed Language*. Tech. rep. NAS-TR-0004-2005. Network and Security Center, Department of Computer Science, Pennsylvania State University, Jan. 2005.

[38] Yao-Wen Huang, Fang Yu, Christian Hang, Chung-Hung Tsai, Der-Tsai Lee and Sy-Yen Kuo. “Securing Web Application Code by Static Analysis and Runtime Protection.” In: *Proceedings of the 13th International Conference on World Wide Web (WWW)*. ACM. 2004, pp. 40–52.

[39] Yao-Wen Huang, Fang Yu, Christian Hang, Chung-Hung Tsai, Der-Tsai Lee and Sy-Yen Kuo. “Verifying Web Applications using Bounded Model Checking.” In: *International Conference on Dependable Systems and Networks (DSN)*. IEEE. 2004, pp. 199–208.

[40] William Jackson. *Static vs. Dynamic Code Analysis: Advantages and Disadvantages*. URL: http://gcn.com/articles/2009/02/09/static-vs-dynamic-code-analysis.aspx.

[41] Limin Jia, Jeffrey Vaughan, Karl Mazurak, Jianzhou Zhao, Luke Zarko, Joseph Schorr and Steve Zdancewic. “Aura: A Programming Language for Authorization and Audit.” In: *The 13th ACM SIGPLAN International Conference on Functional Programming (ICFP)* (2008).

[42] Rajeev Joshi and K. Rustan M. Leino. “A Semantic Approach to Secure Information Flow.” In: *Proceedings of 4th International Conference of Mathematics of Program Construction (MPC)*. Vol. 37. 1. Elsevier, 2000, pp. 113–138.

[43] Nenad Jovanovic, Christopher Kruegel and Engin Kirda. “Pixy: A Static Analysis Tool for Detecting Web Application Vulnerabilities.” In: *Symposium on Security and Privacy (S&P)*. IEEE, 2006, pp. 258–263.

[44] Nenad Jovanovic, Cristopher Kruegel and Engin Kirda. “Pixy: a static analysis tool for detecting web application vulnerabilities (short paper).” In: *Proceedings of the 2006 IEEE symposium on Security and Privacy (S&P)*. 2010, pp. 258–263.

[45] Nenad Jovanovic, Cristopher Kruegel and Engin Kirda. “Precise alias analysis for static detection of web application vulnerabilities.” In: *Proceedings of the 2006 workshop on Programming languages and analysis for security (PLAS)*. ACM. 2006, pp. 27–36.

[46] Jan Jürjens. *Secure Systems Development with UML*. Springer Verlag, 2005.

[47] Kuzman Katkalov, Kurt Stenzel, Marian Borek and Wolfgang Reif. “Model-Driven Development of Information Flow-Secure Systems with IFlow.” In: *ASE Science Journal* 2.2 (2013).

[48] KIT. “JOANA (Java Object-sensitive ANAlysis) - Information Flow Control Framework for Java.” In: *KIT* (2014). Online available at: `http://pp.ipd.kit.edu/projects/joana/`.

[49] K. Rustan M. Leino. “Extended Static Checking: a Ten-Year Pesrsective.” In: *Proceeding Informatics - 10 Years Back. 10 Years Ahead*. Lecture Notes in Computer Science 2000 (Mar. 2001). Ed. by Reinhard Wilhelm, pp. 157–175. DOI: `10.1007/3-540-44577-3_11`.

[50] Peng Li and Steve Zdancewic. “Downgrading Policies and Relaxed Noninterference.” In: *ACM SIGPLAN Notices*. Vol. 40. 1. ACM. 2005, pp. 158–170.

[51] Yin Liu and Ana Milanova. “Static Analysis for Inference of Explicit Information Flow.” In: *Proceedings of the 8th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*. ACM. 2008, pp. 50–56.

[52] V. Benjamin Livshits and Monica S. Lam. “Finding Security Vulnerabilities in Java Applications with Static Analysis.” In: *Proceedings of the 14th conference on USENIX Security Symposium (USENIX SEC)*. 2005, pp. 18–18.

[53] V. Benjamin Livshits and Monica S. Lam. “Tracking Pointers with Path and Context Sensitivity for Bug Detection in C Programs.” In: *ACM SIGSOFT Software Engineering Notes*. Vol. 28. 5. ACM. 2003, pp. 317–326.

[54] V. Benjamin Livshits, Aditya V. Nori, Sriram K. Rajamani and Anindya Banerjee. “Merlin: specification inference for explicit information flow problems.” In: *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. Ed. by Michael Hind and Amer Diwan. ACM, June 2009, pp. 75–86. DOI: `10.1145/1542476.1542485`.

[55] Fred Long, Dhruv Mohindra, Robert C. Seacord, Dean F. Sutherland and David Svoboda. *The CERT® Oracle® secure coding standard for Java™*. Addison-Wesley, Sept. 2011. ISBN: 978-0-321-80395-5.

[56] Stephen McCamant and Michael D. Ernst. “Quantitative Information Flow as Network Flow Capacity.” In: *ACM SIGPLAN Notices* 43.6 (2008), pp. 193–205.

[57] Microsoft. *MSDN Run-time Library Reference - SAL annotations*. Online available at: http://msdn.microsoft.com/en--us/library/ms235402.aspx. 2014.

[58] Sun Microsystems. *Lock_Lint - Static Data Race and Deadlock Detection Tool for C*. vhttp://developers.sun.com/sunstudio/articles/locklint.html.

[59] Yasuhiko Minamide. "Static Approximation of Dynamically Generated Web Pages." In: *Proceedings of the 14th international Conference on World Wide Web (WWW)*. ACM. 2005, pp. 432–441.

[60] *CWE-200: Information Exposure*. Online available at: http://cwe.mitre.org/data/definitions/200.html.

[61] Scott Moore and Stephen Chong. "Static analysis for efficient hybrid information-flow control." In: *Proceedings of the IEEE 24th Computer Security Foundations Symposium (CSF)* (2011), pp. 146–160.

[62] Paul Muntean. *SIBASE Report TP 5.1 – AP 5.1.2: Modeling of information-flow restrictions*. Sept. 2014. URL: https://mediatum.ub.tum.de/doc/1244626/1244626.pdf.

[63] Paul Muntean, Andreas Ibing and Claudia Eckert. "Context-Sensitive Detection of Information Exposure Bugs with Symbolic Execution." In: *Proceedings of the International Workshop on Innovative Software Development Methodologies and Practices (InnoSWDev)* (Nov. 2014), pp. 84–93.

[64] Paul Muntean, Adnan Rabbi, Andreas Ibing and Claudia Eckert. "Automated Detection of Information Flow Vulnerabilities in UML State Charts and C Code." In: *Model-Driven Verification and Validation (MVV)* (2015).

[65] Andrew Clifford Myers. "JFlow: Practical Mostly-Static Information Flow Control." In: *Proceedings of the 26th ACM Symposium on Principles of Programming Languages (POPL)* (Jan. 1999).

[66] Andrew Clifford Myers and Barbara Liskov. "A Decentralized Model for Information Flow Control." In: *Proceedings of the sixteenth ACM symposium on Operating Ssystems Principles (SOSP)* (1997), pp. 129–142.

[67] Andrew Clifford Myers and Barbara Liskov. "Protecting Privacy Using the Decentralized Label Model." In: *ACM Transactions on Software Engineering and Methodology (TOSEM)* 9 Issue 4 (Oct. 2000), pp. 410–442.

[68] *National Vulnerability Database (NVD)*. Information leaks only: https://web.nvd.nist.gov/view/vuln/search-results?adv_search=true&cves=on&cwe_id=CWE-200&pub_date_start_month=7&pub_date_start_year=2015&pub_date_end_month=6&pub_date_end_year=2016&cvss_version=3. URL: https://web.nvd.nist.gov/view/vuln/search-results?adv_search=true&cves=on&pub_date_start_month=7&pub_date_start_year=2015&pub_date_end_month=6&pub_date_end_year=2016 (visited on 08/09/2016).

[69] Soren Preibusch. “Information Flow Control for Static Enforcement of User-defined Privacy Policies.” In: *IEEE International Symposium on Policies for Distributed Systems and Networks (POLICY)* (June 2011).

[70] Adnan Rabbi. “Semi-Automated Detection of Sanitization, Authentication and Declassification Errors in UML State Charts.” MA thesis. Technische Universität München, 2015.

[71] David S. Rosemblum. “Towards a Method of Programming with Assertions.” In: *ACM* 1 (Jan. 1992).

[72] David S. Rosenblum. “A Practical Approach to Programming with Assertions.” In: *IEEE Transactions on Software Engineering (TSE)* 21 (Jan. 1995).

[73] Andrei Sabelfeld and Andrew Clifford Myers. “A Model for Delimited Information Release.” In: *Software Security-Theories and Systems*. Springer, 2004, pp. 174–191.

[74] Andrei Sabelfeld and Andrew Clifford Myers. “Language-based information-flow security.” In: *IEEE Journal on Selected Areas in Communications* 21.1 (2003), pp. 5–19. DOI: 10.1109/JSAC.2002.806121.

[75] Andrei Sabelfeld and Alejandro Russo. “From Dynamic to Static and Back: Riding the Roller Coaster of Information-flow Control Research.” In: *International Conference on Perspectives of System Informatics* (2009).

[76] Andrei Sabelfeld and David Sands. “Declassification: Dimensions and Principles.” In: *Proceedings of 18th IEEE Computer Security Foundations Symposium (CSF)*. Vol. 17. 5. Oct. 2009, pp. 517–548.

[77] Mike Samuel, Prateek Saxena and Dawn Song. “Context-Sensitive Auto-Sanitization in Web Templating Languages using Type Qualifiers.” In: *Proceedings of the 18th ACM Conference on Computer and Communications Security (CCS)*. ACM. 2011, pp. 587–600.

[78] *Sanitization*. URL: https://en.wikipedia.org/wiki/Sanitization_(classified_information) (visited on 07/26/2016).

[79] Umesh Shankar, Kunal Talwar, Jeffrey S. Foster and David Wagner. “Detecting Format-String Vulnerabilities with Type Qualifiers.” In: *USENIX Security Symposium (USENIX SEC)* (Aug. 2001).

[80] Lwin Khin Shar and Hee Beng Kuan Tan. “Predicting Common Web Application Vulnerabilities from Input Validation and Sanitization Code Patterns.” In: *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE. 2012, pp. 310–313.

[81] Lwin Khin Shar, Hee Beng Kuan Tan and Lionel C. Briand. “Mining SQL Injection and Cross Site Scripting Vulnerabilities using Hybrid Program Analysis.” In: *Proceedings of the International Conference on Software Engineering (ICSE)*. IEEE Press. 2013, pp. 642–651.

[82] Vincent Simonet. *FlowCaml in a Nutshell*. In G. Hutton, ed. APPSEM-II, 2003, pp. 152–165.

[83] Vincent Simonet. *The Flow Caml System: Documentation and User’s Manual*. Tech. rep. INRIA, July 2003.

[84] Vincent Simonet. “The Flow Caml system. Software release.” In: *Software Release* (July 2013). Online available at: http://cristal.intia.fr/~simonet/soft/flowcaml.

[85] *Static Code Analysis*. URL: http://searchwindevelopment.techtarget.com/definition/static-analysis. Search Win Development Tech target.

[86] Nikhil Swamy, Juan Chen and Ravi Chugh. “Enforcing Stateful Authorization and Information Flow Policies in FINE.” In: *In proceedings of the 19th European Symposium on Programming (ESOP)* (Mar. 2010).

[87] Nikhil Swamy, Brian J. Corcoran and Michael Hicks. “Fable: A language for Enforcing User-defined Security policies.” In: *Symposium on Security and Privacy (S&P)* (2008).

[88] Lin Tan, Yuanyuan Zhou and Yoann Padioleau. “aComment: mining annotations from comments and code to detect interrupt-related concurency bugs.” In: *Proceedings of the 33rd International Conference on Software Engineering (ICSE)* (May 2011).

[89] TechTarget. *Authentication, Authorization, and Accounting (AAA)*. URL: http://searchsecurity.techtarget.com/definition/authentication-authorization-and-accounting (visited on 08/03/2016).

[90] Linus Torvalds. *Sparse - A sematic parser for C*. Online available at: http://www.kernel.org/pub/software/devel/sparse.

[91] Omer Tripp, Marco Pistoia, Stephen J. Fink, Manu Sridharan and Omri Weisman. “TAJ: Effective Taint Analysis of Web Applications.” In: *ACM Sigplan Notices* 44.6 (2009), pp. 87–97.

[92] Dennis Volpano, Geoffrey Smith and Cynthia Irvine. “A Sound Type System for Secure Flow Analysis.” In: *Journal of Computer Security* 4.3 (1996), pp. 167–187.

[93] Gary Michael Wassermann. “Techniques and Tools for Engineering Secure Web Applications.” PhD thesis. 1 Shields Ave, Davis, CA 95616, United States: Stanford University, Mar. 2007.

[94] Joel Weinberger, Prateek Saxena, Devdatta Akhawe, Matthew Finifter, Richard Shin and Dawn Song. “A Systematic Analysis of XSS Sanitization in Web Application Frameworks.” In: *Proceedings of the 16th European conference on Research in computer security (ESORICS)*. Springer, 2011, pp. 150–171.

[95] Aaron Weiss. *Prevent Web Attacks Using Input Sanitization*. Oct. 2012. URL: http://www.esecurityplanet.com/browser-security/prevent-web-attacks-using-input-sanitization.html (visited on 07/27/2016).

[96] John Whaley. “Context-Sensitive Pointer Analysis Using Binary Decision Diagrams.” PhD thesis. 450 Serra Mall, Stanford, CA 94305, United States: University of California, Davis, Sept. 2008.

[97] John Whaley and Monica S. Lam. “Cloning-Based Context-Sensitive Pointer Alias Analysis Using Binary Decision Diagrams.” In: *ACM SIGPLAN Notices*. Vol. 39. 6. ACM. 2004, pp. 131–144.

[98] David A. Wheeler. *Flawfinder*. Online available at: http://www.dwheeler.com/flawfinder/.

[99] Wikipedia. *Information flow (information theory)*. URL: https://en.wikipedia.org/wiki/Information_flow_(information_theory) (visited on 08/04/2016).

[100] John Wilander and Mariam Kamkar. "A Comparison of Publicly Available Tools for Static Intrusion Prevention." In: *Proceedings of the Network and Distributed System Security Symposium (NDSS)* (2002).

[101] Xusheng Xiao, Nikolai Tillmann, Manuel Fähndrich, Peli de Halleux and xMichał Moskal. *Transparent Privacy Control via Static Information Flow Analysis*. Tech. rep. MSR-TR-2011-93. Microsoft Research, Aug. 2011.

[102] Alexander Yip, Xi Wang, Nickolai Zeldovich and M. Frans Kaashoek. "Improving Application Security with Data Flow Assertions." In: *Proceedings of the sixteenth ACM symposium on Operating systems principles (SOSP)* (Oct. 2009).

[103] Fang Yu, Muath Alkhalaf and Tevfik Bultan. "Patching Vulnerabilities With Sanitization Synthesis." In: *Proceedings of the 33rd International Conference on Software Engineering (ICSE)*. ACM. 2011, pp. 251–260.

[104] Fang Yu, Muath Alkhalaf and Tevfik Bultan. "Stranger: An Automata-based String Analysis Tool for PHP." In: *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Springer, 2010, pp. 154–157.

[105] Yunhui Zheng and Xiangyu Zhang. "Path Sensitive Static Analysis of Web Applications for Remote Code Execution Vulnerability Detection." In: *Proceedings of the International Conference on Software Engineering (ICSE)*. IEEE Press. 2013, pp. 652–661.