2015 IEEE International Conference on Software Quality, Reliability and Security - Companion

Automated Detection of Information Flow Vulnerabilities in UML State Charts and C Code

Paul Muntean, Adnan Rabbi, Andreas Ibing, and Claudia Eckert
Department of Computer Science
Technical University Munich
Email: {paul, rabbi, ibing, eckert}@sec.in.tum.de

***Abstract*—Information flow vulnerabilities in UML state charts and C code are detrimental as they can cause data leakages or unexpected program behavior. Detecting such vulnerabilities with static code analysis techniques is challenging because code is usually not available during the software design phase and previous knowledge about what should be annotated and tracked is needed. In this paper we propose textual annotations used to introduce information flow constraints in UML state charts and code which are afterwards automatically loaded by information flow checkers that check if imposed constraints hold or not. We evaluated our approach on 6 open source test cases available in the National Institute of Standards and Technology (NIST) Juliet test suite for C/C++. Our results show that our approach is effective and can be further applied to other types of UML models and programming languages as well, in order to detect different types of vulnerabilities.**

***Keywords*—model-based verification, information flow vulnerability, static code analysis**

I. Introduction

The US National Vulnerability Database (NVD) [44] lists 8454 common vulnerabilities and exposures in the last 12 months from which 403 (4.77%) 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 as, information flow leakages/disclosure, weird program behavior and weaker cryptographic algorithm encryption.

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). This is why it is important to develop techniques and tools which can detect this type of vulnerabilities before they materialize in production code.

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 [6] can be used to provide additional knowledge regarding security issues. On the other hand code annotations can increase the number of source code lines by 10% [27]. In order to detect information flow vulnerabilities software artifacts have to be annotated with annotations attached to public data, private data and to system trust boundaries. Next, annotated artifacts have to be made tractable by tools which can use the annotations and check if information flow constraints hold or not based on information propagation techniques.

The detection of information flow vulnerabilities in code and UML state charts is not well addressed and is particularly challenging. Foremost, there is no common annotation language for annotating UML state charts 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 the software development process in order to avoid later costly repairs or exploitable vulnerabilities.

In this paper we address this open problem by providing: a light-weight security annotation language, two editors used to edit source code files and UML state charts and four information flow checkers. The checkers can automatically load and use code annotations in order to detect explicit and implicit information flow [53] vulnerabilities based on Extended Static Checking (ESC) [10] in UML state charts and C code.

In summary, our contributions are:

- A novel light-weight security annotation language used to define information flow constraints in UML state charts and source code, §III.
- Definition of information flow inference rules which were implemented inside our information flow checkers, §IV-2.
- Two annotation language editors designed as Eclipse plug-ins which are used to edit UML state charts and source code files, §V-B and §V-D.
- Experiments are presented in section §V based on automatic loading and usage of textual annotations inside 3 new checkers, §V-C and a checker, §V-E which is an extended version of [39].

The remainder of the paper is organized as follows: §II introduces two motivating examples of information flow vulnerabilities. §III presents addressed challenges, the annotation language tag set and the used language design process. §IV gives a brief overview about the implementation. §V presents experimental results. §VI addresses related work. Finally, §VII contains the conclusion and future steps.

978-1-4673-9598-4/15 $31.00 © 2015 IEEE
DOI 10.1109/QRS-C.2015.30

II. Motivation

The scenarios II-1 and II-2 highlight that: first, in order to detect implicit information flow vulnerabilities during the design phase a design tool is needed where for instance a cryptographic algorithm can be modeled and annotated so that the model is tractable afterwards for information flow propagation checkers; second, for detecting explicit information flow vulnerabilities in source code an information flow checker is needed that can track confidential (private) information on all satisfiable program execution paths. Next, we present two scenarios which usually arise in a software company specialized in producing software that has to satisfy imposed security requirements during design and coding phase. Lets assume, Bob is a software engineer in this security software company. Due to his years of experience Bob is frequently involved in both the design and coding phase.

1) Detecting Vulnerabilities During Design: During the design phase of a new software product Bob gets the following requirement from the stakeholder Alice.

Make sure that the cryptographic algorithms used in the "security packages" contain all algorithm steps in the right order as specified in the algorithm API.

Note, that if a step of an cryptographic algorithm was forgotten inside a program then this can lead to software vulnerabilities [36].

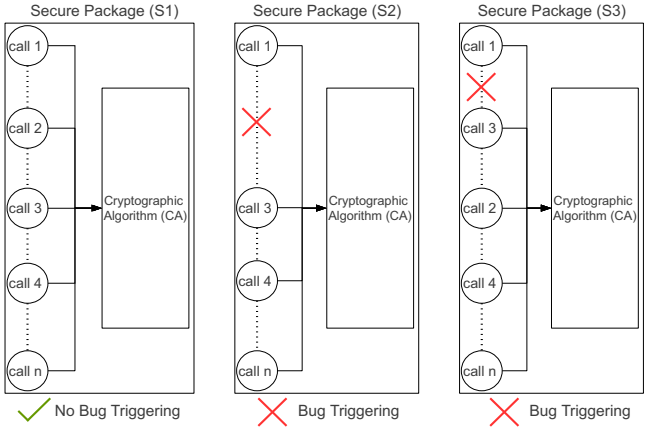

Fig. 1: Implicit information flows sketch (✓ no bug, ✗ bug)

Figure 1 depicts three distinct implicit information flows (Denning [53]) of a cryptographic algorithm contained in three packages, (S1), (S2) and (S3), where each flow starts with "call 1" and ends with "call n". The "call 1" up to "call n" depicted in figure 1 represent C function calls. The dotted lines represent implicit information flows between "call 1" up to "call n" and indicate indirect interdependency between the presence and ordering of the function calls. This indirect interdependency we call implicit information flow. The solid arrows represent function calls to the cryptographic algorithm (CA) which is depicted as a solid rectangle in figure 1. The first information flow depicted in figure 1 with (S1) is a correct flow (✓) since the function calling ordering is respected and no algorithm call is missing. The information flows contained in S2 and S3 marked with (✗) in figure 1 contains a bug respectively. The information flow contained in S2 does not contain the required "call 2" whereas the flow contained in S3 has "call 2" switched with "call 3". This scenario highlights the situation that in case a function call is missing and/or the ordering between one or more function calls is switched then a bug report should be created.

2) Detecting Vulnerabilities During Coding: Alice remembers during the coding phase that the software product contains multiple software modules which are responsible for handling confidential information inside secure software packages and issues Bob the following requirement.

Make sure that no confidential/private information is leaked out of the "security packages" contained in our system.

After a short period of brainstorming Bob comes with the requirement remodeled in figure 2.

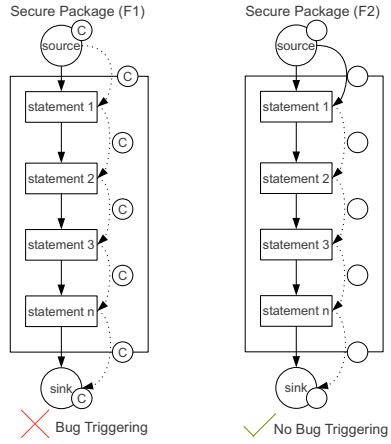

Fig. 2: Explicit information flows sketch (✓ no bug, ✗ bug)

Figure 2 depicts two explicit information flows (Denning [53]) contained in two secure packages (F1) and (F2) where each of the flows starts with "statement 1" and ends with "statement n". The "statement 1" up to "statement n" represent C language statements. F1 is depicted with a solid line containing the flow from the "source" to the "sink" indicated with circles at the top and bottom of each of the two information flows. A "source" is any function or programming language statement which provides private information through a system boundary. A "sink" can be a function call or programming language statement which exposes private information to the outside of the system through a system boundary. Note that a system boundary can be a statement, function call, class, package or module. In figure 2 the "source" and "sink" represent language statements where information enters and respectively leaves F1 or F2. The solid arrows represent an explicit information flow between the "statement 1" up to "statement n". The "source" was tagged with label "c" (confidential) as it inserts confidential information into F1. The dotted arrows represent the passing of the confidential label "c" between the program statements. When a variable labeled with "c" is about to leave F1 then a bug report should be created. This is indicated in figure 2, (✗), at the bottom of F1. Figure 2 depicts inside F2 the same information flow as before but the "source" in this information flow is not tagged. This represents a non-restricted information flow and no bug report should be issued—depicted in figure 2 at the bottom of F2 with ✓.

III. Challenges and Language Implementation

A. Challenges and our Idea

Our goal is to overcome the challenge of not being able to detect implicit and explicit information flow bugs in UML state charts and C code. Thus, we need an annotation language which can be used to annotate UML state charts (during risk analysis [6]) and code by inserting information flow restrictions during two software development phases (design and coding). Our insight is that the same annotation language can be used to add information flow constraints to UML state charts and code in order to detect information flow vulnerabilities.

The challenge was addressed by designing a light-weight annotation language containing textual annotations which can be used to annotate source code and UML state charts which are backward compatible. The textual annotations (single-line and multi-line annotations) are similar to [46], [47] from annotation format perspective. The single-line annotations have the start tag `"//@"` and the multi-line annotations have the start tag `"/*@"` and the end tag `"@*/"`.

For the sake of brevity we will briefly list other addressed challenges throughout our approach: converting textual comments into annotations objects, introducing syntactically correct annotations into files, how to use the same annotation language in order to annotate UML state charts and source code, dealing with scattered annotations and attaching annotations to the right function declaration or variable.

B. Annotation Language Tags

Ⓐ

Target Type	Description
`@function`	function name tagging

Ⓑ

Target Type	Annotation Tag	Description
`@function`	`sink`	uses info.
	`source`	provides info.
	`declassification`	declassifies info.
	`sanitization`	sanitizes info.
	`trust_boundary`	is a trust-boundary

Ⓒ

Target Type	Description
`@preStep`	previous function call name
`@postStep`	next function call name

Ⓓ

Target Type	Annotation Tag	Description
`@parameter`	`confidential H/L`	confidential High/Low tags
	`source H/L`	source High/Low tags

Ⓔ

Target Type	Annotation Tag	Description
`@variable`	`confidential H/L`	confidential High/Low tags
	`source H/L`	source High/Low tags

Ⓕ

Target Type	Description
`Parameter Name`	the tagged parameter name
`Comment`	optional textual comment

TABLE I: Security language annotation tags

Table I contains in section: Ⓐ the annotation language target types, Ⓑ the annotation tags which can be used in combination with the target tag `@function`, Ⓒ the tags `@preStep` and `@postStep` which are used to specify information about previous and post function calls, Ⓓ the tag `@parameter` and the labels confidential and source which can be accompanied by labels L and H which are used to tag public and private variables. Source can be used as file descriptor (e.g., in Posix: 0 stdin, 1: stdout, 2: stderror, 3...), Ⓔ the tag `@variable` which can be used only inside single line annotations whereas `@parameter` is used only in multi line annotations and Ⓕ the tags which are used to specify the name of a parameter or a string comment. The tags were defined and implemented iteratively based on the work flow presented in figure 3 and by using the xText [12] language definition grammar.

C. Language Implementation Process

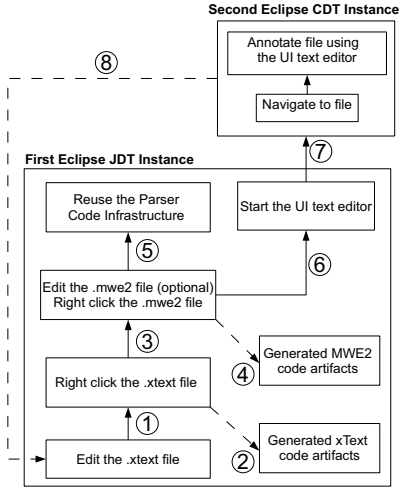

Fig. 3: Language design process

The process depicted in figure 3 was used in order to implement our annotation language. Figure 3 depicts the annotation language implementation process. The process is comprised of the following steps: Foremost, the `.xtext` file containing the language grammar indicated in figure 3 with ① was edited. Next the grammar file is "compiled" and software artifacts are generated, indicated in figure 3 with ②. After editing the `.mwe2` file we "compile" it, depicted with number ④ in figure 3. The result of compiling is: a parser, a lexer and class bindings between these two (lexer and parser) and the grammar `ECore` model indicated in figure 3 with the two boxes located on the right of ② and ④. The generated parser, lexer and the bindings were reused inside our engine, ⑤ and in the UI source file editor indicated with ⑥ in figure 3. After opening and editing a source file with the editor, ⑦, the file can be parsed and the annotations can be automatically loaded and used inside our checkers. This process can be repeated as many times as needed, ⑧.

IV. Implementation

1) The Grammar of Our Annotation Language: Figure 4 contains in Extended Backus–Naur Form (EBNF) notation the grammar of our annotation language. The following type face conventions were used: Italic font for non-terminals, **bold typewriter font** for literal terminals including keywords. Our annotation language grammar has two grammar rules `S_L_Anno` and `M_L_Anno` used for defining security annotations. The `Func_Decl` and `Attr_Definition` rules are used to recognize C/C++ function declarations and variable

Ann_Lang ::= HeaderModel*;

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

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

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

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

Fig. 4: Light-weight annotation language grammar excerpt

declarations. Without the previous two rules the source file editor could not parse existing library files and make annotation language suggestions which are presented in figure V-D. Due to paper space limitations we do not present the details of `Func_Decl` and `Attr_Def` grammar rules.

2) Inference Rules for Secure Information Flows: The goal is to prevent the information flowing from H (high security level, private) variables to L (low security level, public) variables through trust boundaries. The inference rules are implemented inside our static analysis engine which can handle pointers. Lets consider the following C `if` statement, `if a(L) ≤ b(H) then{} else{}`, where variable `a` has attached the label L and variable `b` has attached the label H. There could be implicit (the variables inside the `then` or `else` branch do not depend on the values of `a` or `b`) and explicit (the variables inside the `then` or `else` branch depend on the values of `a` or `b`) flows between variables contained in the `then` or `else` as follows: L to L, H to H, L to H and H to L. If a variable labeled H is used afterwards inside a trust boundary then a information flow leakages should be reported and a bug report should be created.

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

Fig. 5: Secure typing system specialized on trust boundaries

Figure 5 presents the typing system on which our information flow inference rules, depicted in figure 6, are based on. In the first row of figure 5,Ⓐ, we define the following data types: `H` and `L` used to attach private and public labels to program variables (High/private and Low/public) and `PreStep` and `PostStep` used to attach function call ordering labels to previous and post function calls. Figure 5, Ⓑ, presents three types of phrases on which our inference rules are based.

✓⓪ (INT) $\gamma \vdash n : \text{L}$

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

② (ARITH) $\dfrac{\gamma \vdash e : \text{L} \quad \gamma \vdash e' : \text{L}}{\gamma \vdash e + e' : \text{L}}$

✓③ (R-VAL) $\dfrac{\gamma \vdash e : \text{H}\ var}{\gamma \vdash e : \text{H}}$

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

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

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

⑦ (WHILE) $\dfrac{\gamma \vdash e : \text{H} \quad \gamma \vdash c : \text{H}\ cmd}{\gamma \vdash while\ e\ do\ c : \text{H}\ cmd}$

✓⑧ (F-CALL-P) $\dfrac{\gamma \vdash e : \tau\ (\text{PreStep, Poststep})\ \text{H, L}}{\gamma \vdash e : \tau_r\ (\text{PreStep, PostStep})\ \text{H, H}}$

✓⑨ (F-ORDER) $\dfrac{\gamma \vdash e : \tau_1(\text{PreStep}_1, \text{PostStep}_1) * \tau_2(\text{PreStep}_2, \text{PostStep}_2) \ldots}{\gamma \vdash e : \tau_{r1}(\text{PreStep}_1, \text{PostStep}_1) * \tau_{r2}(\text{PreStep}_2, \text{PostStep}_2) \ldots}$

Fig. 6: Typing rules specialized to L, H, PreStep, PostStep for secure explicit and implicit information flow (✓implemented)

Figure 6 depicts our secure information flow inference rules which are based on the Denning [53] lattice model and Volpano et al. [58]. 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 $\gamma \vdash e :$ L, ③, means that expression e has security level L. $\gamma \vdash e : \tau\ \ (PreStep,\ Poststep)$ H, L, ⑧, means that if a function call was tagged with the labels $PreStep$ (H) and $PostStep$ (L) then the label of $PostStep$ is replaced with H after execution. (e.g., strcpy(var1(L), var2(H)) after execution we have strcpy(var1(H), var2(H))).

Note that for inference rules, ①, ③, ④, ⑤, ⑥ and ⑦ we instantiate them with security level H. We decided to do this in order to give concrete inference rules instances. The rules are intended to work the same if instantiated with label L. The rules ⓪, ②, ⑧ are instantiated with L and

work the same when instantiated with security level H. The inference rules presented in figure 6 describe how the label(s): ⓪ `L` is attached to an integer value, ① `H` is attached to a variable ② `L` is passed to the result of an arithmetic addition operation ③ `H` is passed during a `return` statement 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, ⑦ `H` is passed during a `while` statement, ⑧ `PreStep` and `PostStep` are used to pass the label `H` between the parameters of a function call, ⑨ `PreStep_i` and `PostStep_i` are used to define a fixed ordering and presence of function calls (implicit information flow, functions are not calling each other), no security level was needed (see listing 2).

3) Information Flow Checkers: We developed 4 C code checkers based on the Codan API [26] for 4 types of bugs ((a) one checker for the programs contained in CWE-526, CWE-534 and CWE-535 used to detect information exposure bugs) and ((b) three checkers for each of the programs generated from: CWE-259, CWE-325 and CWE-666 (three new C programs were obtained by remodeling the programs contained in CWE-259, CWE-325 and CWE-666 with UML state charts and then C code was generated). The available or generated source code was parsed and a Control Flow Graph (CFG) was constructed for each program. The Juliet test suite contain many ”good” and bad paths. We decided to model only one ”good” path and ”bad” path in each UML state chart for the sake of brevity. Thus, the user can model and generate as many CFG paths as he wants. The checker (a) uses the inference rules: ③ and ⑧ and the other three checkers (b) use the rules: ⓪, ①, ④, ⑧ and ⑨, depicted in figure 6 in order to detect the bugs. The plug-in checker (a) was developed in 3 person-days and the other three checkers (b) were developed in 1 person-day for each of them. Thus, the most time consuming tasks was implementation of inference rules.

4) UML State Chart Editor: We developed an UML state chart editor based on the open source Yakindu SCT [21] framework. We extended the existing language grammar with our annotation language grammar in order to support our set of tags. Furthermore, we implemented an annotation proposal filter which was used to filter out (not present to the user) the annotation language tags of the Yakindu SCT language grammar.

5) Source Code Editor: We implemented a source code editor which offers annotation language proposals which are context sensitive with respect to the position of the currently edited syntax line. If for example, a C expression is not properly parsed then the proposal mechanism would not work from that line on until the end of the file. Thus, the editor suggestions work only if the whole file is parsed without errors.

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

V. Experiments

A. Methodology

We selected 6 C test cases contained in the open source Juliet test suite [43], ((a) CWE-259 [32](the programs contain a hard-coded password, which it uses for its own inbound authentication or for outbound communication to external components), CWE-325 [36](the software does not implement a required step in a cryptographic algorithm, resulting in weaker encryption than advertised by that algorithm), CWE-666 [33](the software performs an operation on a resource at the wrong phase of the resource’s lifecycle, which can lead to unexpected behaviors) and (b) CWE-526 [37](environmental variables may contain sensitive information about a remote server), CWE-534 [34](the application does not sufficiently restrict access to a log file that is used for debugging), CWE-535 [35](a command shell error message indicates that there exists an unhandled exception in the web application code. An attacker can leverage the conditions that cause these vulnerabilities in order to gain unauthorized access to the system)).

Vulnerabilities (a) were selected: first, because these contain information flows which can be modeled with UML state charts and second, in order to present how our approach can address the first scenario presented in section II-1. The vulnerabilities (b) were selected since they contain information flow bugs inside C code and they represent a good fit with regard to how our approach addresses the second scenario depicted in section II-2. The first three programs contained in (a) were remodeled with our UML state charts editor in order to detect implicit and explicit information flow bugs in the generated code.

First, we remodeled programs (a) with our UML state chart editor V-B by attaching to the UML state chart states information flow propagation restrictions. Next, C code was generated from the UML state charts using our C code generator. After code generation, we ran our three (one explicit and two implicit) checkers for CWE-259 (explicit information flow propagation checker), CWE-325 (implicit information flow propagation checker) and CWE-666 (implicit information flow propagation checker)) on the generated code. Second, we annotated programs (b) with information flow restriction annotations using our source code language editor V-D. Next, we ran our explicit information flow checker on the annotated code (for CWE-526, CWE-534 and CWE-535 (explicit information flow propagation checker)).

The goal of the experiments presented in sections V-B, V-C and V-D, V-E are to find out the followings: (1) if all vulnerabilities could be successfully detected, (2) if false positives were generated, (3) the vulnerabilities detection runtime overhead and (4) if our approach is usable for detecting explicit and implicit information flow bugs in UML state charts

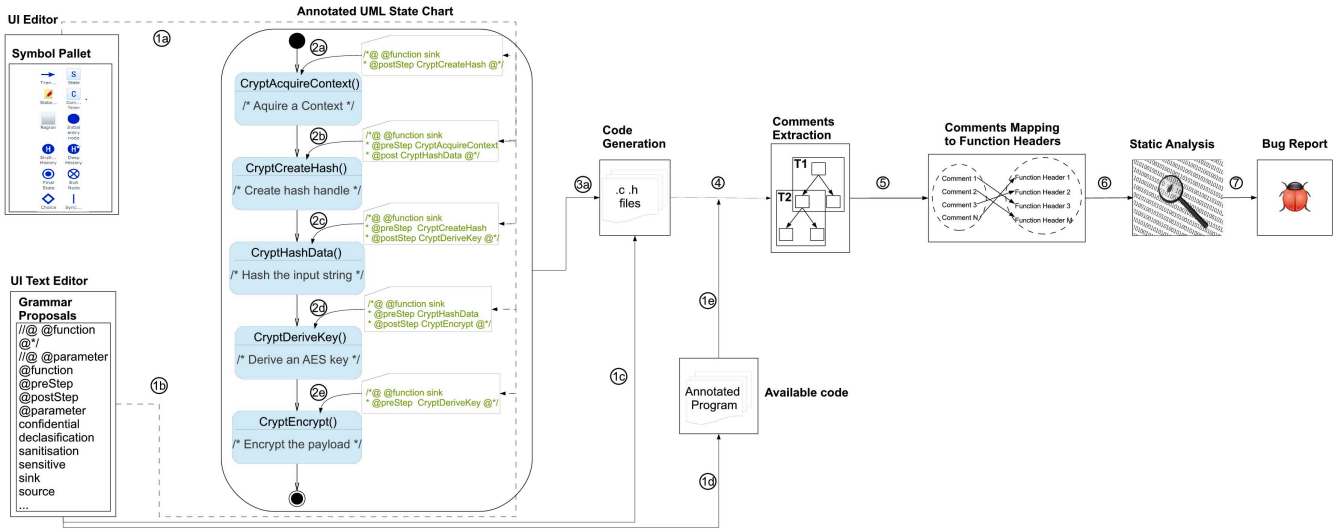

Fig. 7: The process of annotating UML state charts and/or source code and bug detection

and source code. Furthermore, we have given a brief overview of bug reports in figures 9 and 11 in order to describe how bugs can be traced back to the location where they were detected. As testing platform we used the Ubuntu OS running kernel 3.8.0-35-generic, 64-bit, Intel i7-4770 CPU @ 3.40GHz × 8, 16 GB RAM. Finally, we addressed threats to validity.

B. Annotating UML State Charts

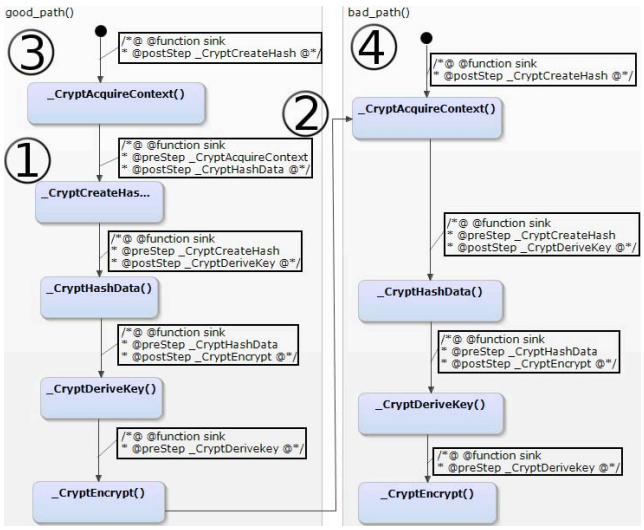

Fig. 8: First C program contained in the CWE-325 test case [43] remodeled with an UML state chart

Figure 8 depicts the first program contained in CWE-325 [36] (”Missing Required Cryptographic Step” and contains implicit information flows) remodeled with our UML state chart editor. The user can edit the UML state chart using the tool palette containing the following: → transition, S state, state annotation, C composite states, region, ● initial entry node, Ⓗ shallow history, Ⓗ* deep history, ◉ final state, ⊗ exit node, ◇ choice and | synchronization. For the sake of brevity we highlight the remodeling of only CWE-325 and exclude showing remodeling details for CWE-259 and CWE-666.

Figure 8 depicts with boxes having rounded corners (e.g., figure 8 circled number ①) UML state chart states which were added for each of the function calls of the first program contained in CWE-325 [43] (depicts ”good” and ”wrong” usage of a cryptographic algorithm). The good path (not buggy path) and bad path (buggy path) (”good_path()” and ”bad_path()” strings) are contained inside two large boxes depicted in figure 8 with circled numbers ③ and ④. In those two boxes we remodeled the ”good” and ”bad” program execution paths contained in Juliet test case CWE-325 [43]. Next, we attached to each UML state chart state transition representing a program function call an annotation box (e.g., figure 8 circled number ②). The annotation boxes contain annotation tags marking what is the correct, previous and next (post) function call for the used cryptographic algorithm inside the first program contained in CWE-325 [43]. Concretely, we used the tag `sink` (was used since each processing step uses data in order to perform some internal operations) and the tags `@preStep` and `@postStep` in order to mark the previous expected function call and the next expected function call with respect to CWE-325. These annotations were used to specify allowed information flows. The modeling of the UML state chart (each algorithm function call was successfully modeled) was performed by one user which needed 10 minutes for modeling the UML state chart and adding the annotations. Each consecutive algorithm function call can be regarded as a forbidden information flow if the order of the call does not match the information stored in his `@preStep` and `@postStep` tags.

The annotations depicted in listings 1, 2 and 3 were obtained from the first three programs contained in CWE-259, CWE-325 and CWE-666 [43] which were previously remodeled with UML state charts and then code was generated.

Listing 1: cwe_259.h

```
//@ @variable b H
char *b;
//@ @variable a L
char *a;
/*@ @function sink
* @parameter b H
* @parameter a L @*/
void _strcpy(char *a, char *b);
/*@ @function sink
* @parameter a L @*/
void _LogonUserA(char *a);
/*@ @function source
* @parameter a L @*/
void _fgets(char *a);
```

Listing 2: cwe_325.h

```
/*@ @function sink
* @postStep _CryptCreateHash @*/
void _CryptAcquireContext();
/*@ @function sink
* @preStep _CryptAcquireContext
* @postStep _CryptHashData @*/
void _CryptCreateHash();
/*@ @function sink
* @preStep _CryptCreateHash
* @postStep _CryptDeriveKey @*/
void _CryptHashData();
/*@ @function sink
* @preStep _CryptHashData
* @postStep _CryptEncrypt @*/
void _CryptDeriveKey();
/*@ @function sink
* @preStep _CryptDerivekey @*/
void _CryptEncrypt();
```

Listing 3: cwe_666.h

```
/*@ @function sink
* @postStep _listen @*/
void _bind();
/*@ @function sink
* @preStep _bind
* @postStep _accept @*/
void _listen();
/*@ @function sink
* @preStep _listen @*/
void _accept();
```

The programs were remodeled with UML state charts by following the steps presented in figure 7 sequentially: (1a) (select tool from palette), (1b) (select annotations) (2a), (2b), (2c), (2d), (2e) (insert annotations) and (3a) which represents the C code generation step based on the C code generator presented in section IV-6. On the generated code the editor V-D can be applied as many times as needed in order to refine the annotations, indicated in figure 7 with (1c). Next, the C language comments (annotations) depicted in listings 1, 2 and 3 are used to detect bugs in the generated programs. Steps ④, ⑤, ⑥ and ⑦ depicted in figure 7 are performed sequentially and automatically by our three checkers on each of the three generated programs in order to detect the bugs.

C. UML State Chart Information Flow Checkers Results

File	SLoC	LoCo	FP	TP	TTP	T [s]	DB
cwe_259.c	26	-	0	1	1	0.005	✓
cwe_259.h	-	24	-	-	-	0.008	
cwe_325.c	34	-	0	1	1	0.003	✓
cwe_325.h	-	13	-	-	-	0.005	
cwe_666.c	24	-	0	1	1	0.002	✓
cwe_666.h	-	11	-	-	-	0.003	
Total	84	48	0	3	3	0.026	✓

TABLE II: Bug detection results for generated programs

Table II contains the following abbreviations: Source Lines of Code (SLoC), Lines of Comments (LoCo), FP (False-Positives), True Positives (TP), Total True-Positives (TTP) per Test Case (TC) with all programs included, Time in seconds (T [s]) and Detected Bug (DB). Table II presents the results of running our three checkers (2 implicit information flow checkers for CWE-325 and CWE-666 and one explicit information flow checker for CWE-259) on each of the generated programs. The three generated programs contained respectively three true positives which were successfully detected as indicated in table II column 8. In general if a UML state chart is properly modeled and annotated then every kind of information flow related bug can be addressed since our approach is based on program execution paths which assure full program path coverage. The trustworthiness of our approach (true/false positive) is based on correct information flow inference rules. Proving correctness of the used inference rules is beyond the scope of this paper.

Fig. 9: Bug reports for the three generated programs

Figure 9 depicts the bug reports obtained by running the checkers in parallel on the generated programs. The circled numbers in figure 9 indicate the following: number ① indicates the analyzed programs (generated programs), number ② presents three bug reports (each generated by one of our checkers for the analyzed programs) and number ③ shows the bug location (line number 18) in source code file cwe_325.c by clicking on the second bug report (Missing Crypto Step Bug Detected). The bug reports depicted in figure 9 with number ②, confirm that all bugs were successfully detected and no false positives and false negatives were generated.

D. Annotating Source Code

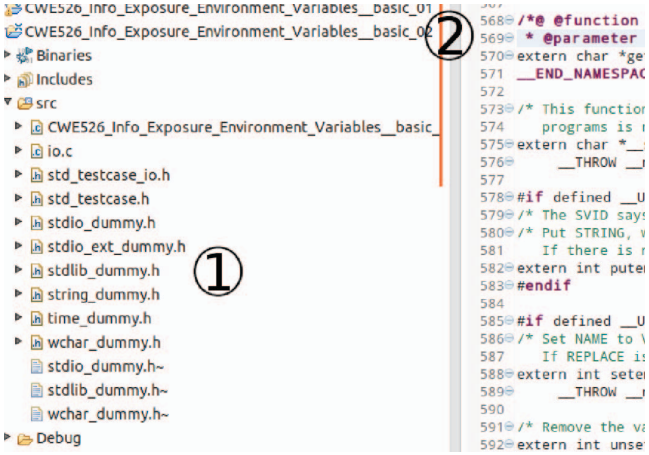

Fig. 10: Annotating source code files

Figure 10, number ①, indicates the annotated file "stdlib_dummy.h" contained in the library (AnnoLib) which is used inside the second Eclipse C project. The library was edited once and then re-used 90 times in each of the 90 programs contained in the workspace. Number ② indicates the place (lines 568–569) inside the current opened file where one annotation was added.

Listing 4: Source code annotations (→ depicts belonging)

```
0. /*@ @function source
1. * @parameter __name confidential @*/
2. char *getenv(__const char *__name);      → stdlib_dummy.h
3. /*@ @function sink
4. * @parameter __format confidential @*/
5. int printf(__const char *__restrict __format, ...); →
       stdio_dummy.h
6. /*@ @function source
7. * @parameter password confidential @*/
8. int LogonUserA(char *username, char *domain, char password
       ,...); → windows_dummy.h
9. /*@ @function source
10. * @parameter password confidential @*/
11. int LogonUserW(char *username, char *domain, char password
       ,...); → windows_dummy.h
12. //@ @function sink
13. int fprintf(...); → stdio_dummy.h
14. //@ @function sink
15. int fwprintf(...); → wchar_dummy.h
```

Listing 4 depicts all generated annotations as C language comments for CWE-526, CWE-534 and CWE-535. In listing 4 "→" indicates in which file belonging to AnnoLib each annotated function declaration is contained. Furthermore, we used in listing 4 "..." to denote that not all parameters were listed due to paper space limitations. We attached in total 6 annotations to the programs trust boundaries of the selected test programs indicated in listing 4 as code comments. Each code comment is attached to the next function declaration contained in listing 4 on the following line. The AnnoLib contains 3902 LOC, 6 annotations and 6 files which are copies of the C standard library files. The AnnoLib was annotated by following the steps indicated in figure 7 sequentially: (1d) and (1e) using our source code editor V-D. Steps ④, ⑤, ⑥ and ⑦ depicted in figure 7 are performed sequentially and automatically by our information flow checker after the annotations were added in order to detect the bugs.

E. Source Code Information Flow Checker Results

TC	TPr	SLoC	FP	TP	TTP	T [s]	DB
CWE-526	18	1172	0	17	18	37.432	✓
CWE-534	36	5745	0	34	36	88.426	✓
CWE-535	36	5218	0	34	36	86.191	✓
AnnoLib	-	2194	-	-	-	1.400	-
Total	90	14329	0	85	90	213.399	✓

TABLE III: Bug detection results with annotated library

Table III contains the same abbreviations as table II and uses additionally the abbreviations: Test Case (TC) and Test Program (TPr). Table III presents the results of running our explicit information flow checker together with the annotated library, AnnoLib on 90 C programs. This table is similar to the table depicted in [39] except the fact that this time we used an annotated library in order to annotate program sinks and sources. The average AnnoLib loading time (annotations parsing and conversion to EObjects) per C program (90 TPr) was 1.4 [s]. If the AnnoLib would be loaded only once (this was not possible due to our current engine Eclipse plug-in implementation) for all the 90 TPr we could save 126 [s] (90 TPr × 1.4 [s]), thus achieving an overhead of 1.1%. We think that 1.1% is a low overhead value regarding the fact that the 6 annotations were distributed in 6 header files contained in AnnoLib (3902 LOC).

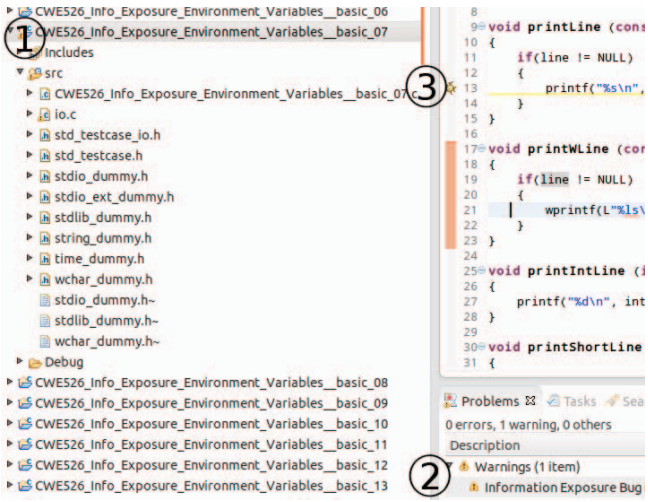

Fig. 11: Information flow bug report

Figure 11 depicts a bug report obtained after running our checker on the 7th program contained in CWE-526 [43]. Number ① contained in figure 11 indicates the current analyzed program, number ② presents the bug report of our information flow checker and number ③ depicts the bug location (line number 13) in the program where the information exposure bug was detected. Next, we tested our checker on the 90 TPr and detected 85 true positives out of 90 true positives (5 programs were not analyzed due to the current limitation of our engine to deal with C "goto" statements) without generating any false positives and false negatives.

F. Threats to Validity

Internal Validity: Our experimental results presented in table III may not support our findings for several reasons. If our annotation gathering analysis incorrectly skips some annotations or misses some constraints, then it might report false positives or false negatives. For these reason we built an annotation checking mechanism inside our tool which checks if the generated annotation objects are valid models of the textual annotation. If we misinterpreted the timing results, then the potential total overhead of 1.1% could not be feasible. We addressed these factors by testing our tool extensively on open source programs with a known number of information flow bugs and repeating each experiment for three times for each of the analyzed programs.

External Validity: Our results cannot potentially be generalized for other scenarios since our light-weight annotation language has a reduced expressiveness and only 5 inference rules were implemented. Furthermore, we did not test our UML state chart editor on very large UML state charts but as far as we noticed the used UML state chart framework [21] can deal with a high number of created states (greater than 100 states) inside a single UML state chart. Our two editors and four information flow checkers are Eclipse plug-ins and can run only in the Eclipse environment. We mitigate this issue by arguing that developing a GUI for an Eclipse plug-in is not a time consuming task for an experienced developer.

VI. Related Work

The detection of information flow vulnerabilities [31] can be addressed with dynamic analysis techniques [2], [16], [48], static analysis techniques [17], [41], [51], [58], [60] (similar to our approach with respect to static analysis of code and tracking of data information flow) and hybrid techniques which combine static and dynamic approaches [38]. Also, extended static checking [10] (ESC) is a promising research area which tries to cope with the shortage of not having certain program run-time information.

The static code analysis techniques need to *know* which parts of the code are: sinks, sources and which variables should be tagged. A solution for tagging these elements in source code is based on a pre-annotated library which contains all the needed annotations attached to function declarations. Leino [27] reports about the annotation burden as being very time consuming and disliked by some programming teams. There are many annotation languages proposed until now for extending the C type system [9], [13], [29], [30], [57] to be used during run-time as a new language run-time for PHP and Python [61] to annotate function interfaces [13], [29], [57], to annotate models in order to detect information flow bugs [24] to annotate source code files [46], [47], [56] or to annotate control flows [13], [15], [29]. The studies rely on manually written annotations while our annotation language is integrated into two editors which are be used to annotate UML state charts and C code by selecting annotations from a list and without the need to memorize a new annotation language.

The following annotation languages have made significant impact: Microsoft's SAL annotations [29] helped to detect more than 1000 potential security vulnerabilities in Windows code [3]. In addition, several other annotation languages including FlowCaml [50], Jif [7], Fable [55], AURA [22] and FINE [54] express information flow related concerns. Recently taint modes integrated in programming languages as Caml-based FlowCaml [52], Ada-based SPARK Examiner [5] and the scripting. However, none of these annotation and programming languages have support for introducing information flow restrictions in both models and the source code.

Splint [14], Flawfinder [59] and Cqual [49] are used to detect information flow bugs in source code and come with comprehensive user manuals describing how the annotation language can be used in order to annotate source code. iFlow [24] is used for detecting information flow bugs in models and is based on modeling dynamic behavior of the application using UML sequence diagrams and translating them into code by analyzing it with JOANA [25]. In comparison with our approach these tools do not use the same annotation language for annotating UML models and code. Thus, a user has to learn to use two annotation languages which can be perceived to be a high burden in some scenarios. UMLSec [23] is a model-driven approach that allows the development of secure applications with UML. Compared with our approach, UMLSec does neither include automatic code generation nor the annotations can be used for automated constraints checking. Heldal et al. [18], [19] introduced an UML profile that incorporates a decentralized label model [40] into the UML. It allows the annotation of UML artifacts with Jif [42] labels in order to generate Jif code from the UML model automatically. However, the Jif-style annotation already proved to be non-trivial on the code level [45], while [19] notes that the actual automatic Jif code generation is still future work. These approaches can not be used to annotate both UML models and code. Moreover, these approaches lack of tools for automated checking of previously imposed constraints.

VII. Conclusion and Future Work

We developed a keyword-based annotation language that can be used out of the box for annotating UML state charts and C code in two software development phases by providing two editors for inserting security annotations in order to detect information flow bugs automatically. We evaluated our approach on open source programs and showed that our approach is applicable to real scenarios.

To the best of our knowledge our annotation language is the only light-weight annotation language usable for specifying information flow security constraints which can be used in the design and coding phase in order to detect information flow bugs.

In future we want to extend our source code editor as a pop-up window based proposal editor used to add/retrieve annotation to/from a library. The definition of new language annotation tags should be possible from the same window by providing two running modes (language extension mode and annotation mode). The envisaged result is to reduce the gap between annotations insertion/retrieval and the definition of new language tags.

Acknowledgments

This research is funded by the German Ministry for Education and Research (BMBF) under grant number 01IS13020.

REFERENCES

[1] S. Arzt and et al. SuSi: A Tool for the Fully Automated Classification and Categorization of Android Sources and Sinks. Technical report, TUD-CS-2013-0114, EC SPRIDE, 2013.

[2] T. Avgerinos and et al. AEG: Automatic Exploit Generation. *Proceedings of the Network and Distributed System Security Symposium (NDSS 11)*, February 2011.

[3] T. Ball and et al. Annotation-based property checking for systems software. Technical report, Microsoft, May 2008.

[4] L. Cavallaro and et al. On the Limits of Information Flow Techniques for Malware Analysis and Containment. *International Conference, DIMVA 2008*, pages 143–163, July 10-11 2008.

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

[6] B. Chess and G. McGraw. Static Analysis for Security . *IEEE Security & Privacy*, November/December 2004.

[7] S. Chong and et al. Jif: Java + information flow, July 2006. Software release.

[8] A. Cimatti, A. Griggio, B. Schaafsma, and R. Sebastiani. The MathSAT 5 SMT solver. *TACAS 2013.*, 2013.

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

[10] D. L. Detlefs and et al. Extended Static Checking . *Compaq SRC Research Report 159*, 1998.

[11] M. Dietz and et al. Quire: Lightweight Provenance for Smart Phone Operating Systems. *Proc. of the 20th USENIX conference on Security (SEC'11)*, 2011.

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

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

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

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

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

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

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

[19] R. Heldal and et al. Supporting Confidentiality in UML: A Profile for the Decentralized Label Model. Technical Report TUM-I0415, 2004.

[20] N. Husted and et al. Android Provenance: Diagnosing Device Disorders. *5th USENIX Workshop on the Theory and Practice of Provenance*, 2013.

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

[22] L. Jia and et al. Aura: A programming language for authorization and audit . *ICFP*, 2008.

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

[24] K. Katkalov and et al. Model-Driven Development of Information Flow-Secure Systems with IFlow. *ASE Science Journal*, 2(2), 2013.

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

[26] A. Laskavaia. Codan- C/C++ static analysis framework for CDT, *EclipseCon, 2011*, http://www.eclipsecon.org/2011/sessions/index0a55.html?id=2088

[27] K. Rustan M. Leino. Extended Static Checking: a Ten-Year Pesrsective. *Proceeding Informatics - 10 Years Back. 10 Years Ahead*, Jan. 2001.

[28] NASDAQ Stock Market. Facebook NASDAQ. http://www.nasdaq.com/symbol/fb, September 2014.

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

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

[31] Mitre. CWE-200: Information Exposure, http://cwe.mitre.org/data/definitions/200.html

[32] Mitre. CWE-259: Use of Hard-coded Password, http://cwe.mitre.org/data/definitions/259.html

[33] Mitre. CWE-666: Operation on Resource in Wrong Phase of Lifetime, http://cwe.mitre.org/data/definitions/666.html

[34] Mitre. CWE-534: Information Exposure Through Debug Log Files, http://cwe.mitre.org/data/definitions/534.html

[35] Mitre. CWE-535: Information Exposure Through Shell Error Message, http://cwe.mitre.org/data/definitions/535.html

[36] Mitre. CWE-325: Missing Required Cryptographic Step, http://cwe.mitre.org/data/definitions/325.html

[37] Mitre. CWE-526: Information Exposure Through Environmental Variables, http://cwe.mitre.org/data/definitions/526.html

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

[39] P. Muntean and et al. Context-Sensitive Detection of Information Exposure Bugs with Symbolic Execution. *Proc. of the InnoSWDev'14*, 2014.

[40] A. Myers and B. Liskov. Protecting privacy using the decentralized label model. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 9 Issue 4:410–442, Oct. 2000.

[41] A. C. Myers. JFlow: Practical Mostly-Static Information Flow Control. *Proc. of the 26th ACM POPL'99*, Jan. 1999.

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

[43] NIST. Juliet Test Suite v1.2 for C/C++. http://samate.nist.gov/SRD/testsuites/juliet/Juliet_Test_Suite_v1.2_for_C_Cpp.zip

[44] National Vulnerability Database (NVD), https://web.nvd.nist.gov/view/vuln/search-results?query=information+exposure&search_type=all&cves=on

[45] S. Preibusch. Information ow control for static enforcement of user-de ned privacy policies. *POLICY 2011, IEEE International Symposium on Policies for Distributed Systems and Networks*, June 2011.

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

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

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

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

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

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

[52] V. Simonet. The Flow Caml system. Software release. http://cristal.inria.fr/~simonet/soft/flowcaml, July 2003.

[53] Dorothy E. Denning. A lattice model of secure information flow. *Communications of the ACM*, pages 236–243, 1976.

[54] N. Swamy and et al. Enforcing Stateful Authorization and Information Flow Policies in FINE . *In proc. of ESOP'10*, March 2010.

[55] N. Swamy and et al. Fable: A language for enforcing user-defined security policies . *In S&P*, 2008.

[56] L. Tan and et al. aComment:Mining Annotations from Comments and Code to Detect Interrupt Related Concurency Bugs. *ACM*, May 2011.

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

[58] D. Volpano and et al. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(3):167–187, 1996.

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

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

[61] A. Yip and et al. Improving Application Security with Data Flow Assertions. *SOSP'09*, Oct. 2009.