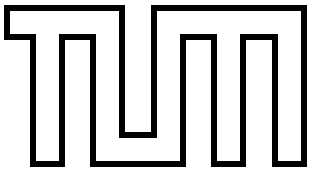


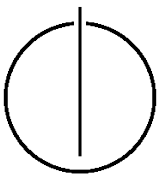
FAKULTÄT FÜR INFORMATIK

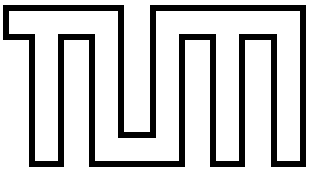
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Automated Detection, Localization and Removal of Information Exposure Errors

Kommanapalli Vasantha





FAKULTÄT FÜR INFORMATIK

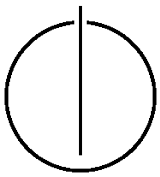
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Automated Detection, Localization and Removal of Information Exposure Errors

Automatisches Erkennen, Lokalisieren und Entfernen von ungewollter Informationsfreigabe

Author:	Kommanapalli Vasantha
Supervisor:	Prof. Dr. Claudia Eckert
Advisor:	M. Sc. Paul Muntean
Date:	November 15, 2015



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

München, den 8. Oktober 2015 Kommanapalli Vasantha

Acknowledgments

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

Abstract

Automatically detecting, localizing and fixing information errors is very much needed in current generation as people are not ready to spend much on their time in debugging and fixing the errors. In general debugging requires a lot of time and effort. Even if the bug's root cause is known, finding the bug and fixing it is a tedious task. The information that is exposed may be very valuable information like passwords or any information that is used for launching many deadly attacks. In order to fix the information exposure bug we should refactor the code in such a way that the attacks or information exposure can be restricted.

The main objective of this master thesis is to develop a quick fix generation tool for information exposure bugs. Based on the available information exposure checker which detects errors in the open source Juliet test cases: CWE-526, CWE-534 and CWE-535 a new quick fix tool for the removal of these bugs should be developed. The bug location and the bug fix location can be different (code lines) in a buggy program. Thus, there is a need for developing a bug quick fix localization algorithm based on software bug fix localization techniques. It should help to determine the code location where the quick fix should be inserted. Based on the bug location the developed algorithm should indicate where the quick fix should be inserted in the program. We can consider a bug fix to be a valid fix if it is able to remove the confidential parameter inside a function call to a system trust boundary. After the quick fix location was determined and the format of the quick fix was chosen then the quick fix will be inserted in the program with the help of the Eclipse CDT/LTK API.

The effectiveness of the implemented code refactoring is checked by re-running the information flow checker on the above mentioned open source Juliet test cases. If the checker detects no bug then the code patches are considered to be valid. The generated patches are syntactically correct, can be semi-automatically inserted into code and do not need additional human refinement. The generated patches should be correct and sound.

Contents

Outline of the Thesis

Part I: Introduction and Theory

CHAPTER 1: INTRODUCTION

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

CHAPTER 2: TECHNICAL AND SCIENTIFIC FUNDAMENTALS

This chapter presents all the techincal and the scientific fundamentals that are required in order understand the work done in this thesis

CHAPTER 3: RELATED WORK

This chapter presents all the related work similar to the work done in this thesis, some comparisions of approaches of different people and also comparisions of similar tools

Part II: Design and Development

CHAPTER 4: OVERVIEW

This chapter presents the whole developed system's overview and all the modules involved.

CHAPTER 4: LOCALIZATION ALGORITHM

This chapter presents the localization algorithm that was developed in order to detect the bugs which are present in the files other than the source files. It also helps in reducing the time required for detecting and localizing the bugs.

CHAPTER 4: APPROACH

This chapter presents the appraoch that has be owned in order to achieve the goal.

CHAPTER 4: IMPLEMENTATION

This chapter presents the whole implementation details about the developed tool.

Part III: Evaluation and Discussion

CHAPTER 5: EVALUATION

This chapter presents the requirements for the process.

CHAPTER 6: DISCUSSION

This chapter presents the requirements for the process.

CHAPTER 7: CONCLUSION

This chapter presents the requirements for the process.

Part I.

Introduction and Theory

1. Introduction

*"The only truly secure system is one that is powered off, cast in a block of concrete and sealed in a lead-lined room with armed guards." — **Gene Spafford***

According to the ultimate security vulnerabiltiy datasource CVE details [6], there are 2082 vulnerabilities related to information exposure. Among many Information Flow(IF) weaknesses, Information exposure is one of the serious issue. These kind of weaknesses may exist in the code without breaking anything in the code but it offers some confidential information to any attacker who wishes to exploit the system. Information exposure bugs could be introduced in different stages of software development. During design, architecture or coding phase and can lead to exposure of critical information or can also lead to strange program behavior. Therefore care should be taken to test the software thoroughly before releasing in order to detect such weaknesses. The testing process in general accounts for half of the whole effort in the software development cycle according to [17] and [19].

In 2013, sensitive data exposure ranked 6th in the AOWASP top ten list [4] and in 2011, information exposure through an error message ranked 17th among CWE/SANS top 25 [3]. Recently, Forbes 2015 [1] published in one of the articles that among the topmost data breaches occurred in the previous years Neiman Marcus hack is famous. In January 2013, many debit and credit card information of almost 350,000 customers have been hacked. It is believed that the breach happened because of a malicious software that was installed onto the Neiman Marcus system. This software collected all the payment card information from customers who purchased. This proves that the software has helped the attackers to leak all the sensitive information through which they could get access to the system without leaving any trace of hacking. Sensitive information can be leaked in many ways [2]. It can be through the environmental variables which contain the sensitive information about any remote server, through a log file that was used while debugging the application where access to that file was not restricted or through a command shell error message which indicates that the web application code has some unhandled exception. In the last case, the attacker can take advantage of that error causing condition in order to gain access to the system without authorisation.

There are many static analysis approaches which can solve this problem by building a control flow graphs (CFG) which, can give us an idea about the execution paths and provides high path coverage. There are many tools that have been built based on this like ESP [11], SLAM [8]. There are many dynamic analysis approaches also but it doesnot guarantee if all the paths have been analysed and also the computational overhead is high. Whereas static analysis provides us with all the execution paths but it requires some heuristics in order to select the paths of interest like the satisfiable paths. This can be done using the SMT solver. SMT solver solves the mathematical expressions provided to it and sometimes this expressions can get very complex [10]. These IF errors can be addressed using dynamic analysis [23], static [26], [21] and hybrid anaylsis(combination of both static

and dynamic analysis).

The goal of this thesis is to develop an algorithm for fault localization(localizer) and also to repair the information exposure buggy programs. Repairing is done using the precise information like failure detection, bug diagnosis, buggy variables which are nothing but the program variables that directly influence the appearance of a bug in the program. For example the buggy variable ”line” reported by the information exposure checker [ref]. In order to repair the buggy program, failure detection and bug diagnosis data has been used to generate the quick fixes for information exposure bugs with the help of a refactoring wizard. We have developed a localization algorithm in order to localize the bug. We need a localizer since the cause for the bug will not be the place where the bug was detected but earlier in the program code where the information flows into the buggy variable. Therefore a novel algorithm is developed in order to detect possible insertion locations for the generated code patches. Here the code patches can be inserted at two different locations: (i) place where the bug was found —”in-place-fix”, (ii) place where the information flows in the buggy variable —”not-in-place-fix”. Approach for generating the program repair depends on the code patch patterns, SMT solving and searching possible quick fix locations searching in the program execution paths that will not affect the program behavior by inserting the patch at ”not-in-place” location. Generated patches do not change the program behavior for the input that doesnot trigger the bug, therefore the generated code patch is sound. It doesnot need further human refinement(final) , no alien code(human readable), syntactically correct and compilable. The defect class that have been addressed here is information exposure through log files , error reports and environmental variables exposure through which sensitive information like the password or the path to remote server could be exposed and the hacker could exploit the system. The fix defect class consists of the removal of the confidential data at the point where the information flows from the trust boundary based on semi-defined patch patterns.The aim of the quick-fix code patch is information exposure error mitigation (e.g., to prevent that an attacker exploits the error in order to gain system access or display of sensitive information). Program repair depends on two dimensions: (i) an oracle which decides upon what is incorrect in order to detect the bug, (ii) another oracle to decide what should be kept correct in order to attain software correctness [ref]. Patches are generated automatically and inserted semi-automatically offline with the possibility to insert them also online. 90 C programs of Juliet test suite CWE-526, CWE-534, CWE-535 [2] have been used to evaluate the developed approach. CWE-526 contains information exposure through environmental variables related bugs and the potential mitigation would be to protect the information stored in the environmental from being exposed to the user. CWE-534 contains information exposure through debug log files related bugs and the potential mitigation would be to remove the debug files before deploying the application into produciton. CWE-535 contains information exposure through shell error message related bugs . In all the three CWE’s the common consequence of the bug is loss of confidentiality.

1.1. Information Exposure Bug

Information Exposure Bug is an error in the software code which intentionally or unintentionally discloses sensitive information to an user who is not explicitly authorized to have

access rights to that information. This information could be sensitive within the developed product's own functionality like a private message, which provides information about the product itself or which exposes the environment that is not available for the attacker and that could be very useful for the attacker like the installation path of that product which can be accessed remotely. Many of the information exposures is due to many of the program errors like the PHP script error that may reveal the path of the program or could be some timing discrepancies in encrypting the data. There are in general many kinds of problems that involve information exposures and the severity of those problems can range vastly based on the type of the sensitive information that has been revealed by the errors. Information exposures are also named as information leak or information disclosure.

1.2. Motivation with Example

In general detecting a information exposure bug depends on accurately finding the source code location where some sensitive information leaves the defined trust- boundary. This is the point where an attacker can exploit this IE vulnerability where sensitive information is leaked out. As already mentioned this information could be any confidential data. So in order to make sure that no sensitive information leaks out , a tool is developed in order to restrict the flow of information outside of the system's trust boundary at the location where the bug was found. In this section a real-world bug fix is presented as an example in order to depict that generating a code patch is not that easy. One needs an insight into the program's functionality and also the kind of bug they are dealing with. Normally, a bug can be fixed in many ways with functionally correct patches. Some of the generated patches may change the program behavior. So care must be take that the program behavior is not changed after the patch insertion.

Listing 1.1: CWE-534 test programs source

```
1
2  if(STATIC_CONST_TRUE)
3     {
4        {
5           char password[100] = "";
6           size_t passwordLen = 0;
7           HANDLE pHandle;
8           char * username = "User";
9           char * domain = "Domain";
10          FILE * pFile = fopen("debug.txt", "a+");
11          if (fgets(password, 100, stdin) == NULL)
12          {
13             printLine("fgets() failed");
14             /* Restore NUL terminator if fgets fails */
15             password[0] = '\0';
16          }
17          /* Remove the carriage return from the string that is inserted
               by fgets() */
18          passwordLen = strlen(password);
19          if (passwordLen > 0)
```

```
20          {
21             password[passwordLen-1] = '\0';
22          }
23          /* Use the password in LogonUser() to establish that it is
               "sensitive" */
24          if (LogonUserA(
25                   username,
26                   domain,
27                   password,
28                   LOGON32_LOGON_NETWORK,
29                   LOGON32_PROVIDER_DEFAULT,
30                   &pHandle) != 0)
31          {
32             printLine("User logged in successfully.");
33             CloseHandle(pHandle);
34          }
35          else
36          {
37             printLine("Unable to login.");
38          }
39          /* FLAW: Write sensitive data to the log */
40       - fprintf(pFile, "User attempted access with password: %s\n",
             password);
41        /*Source code patch(Quick fix)*/
42       + fprintf(pFile,"User attempted access with password:\n");
43          {
44             fclose(pFile);
45          }
46       }
47    }
```

Listing 1.2: CWE-526 test programs source

```
1
2  void CWE526_bad(){
3      if (staticFive == 5){
4         /*FLAW:environment variable exposed*/
5         - printLine(getenv("PATH"));
6          /*Source code patch(Quick fix)*/
7           + printLine(getenv(" "));
8          }
9              }
```

Listing 1.3: CWE-526 test programs sink

```
1  void printLine (const char *line){
2            if(line != NULL){
3                 printf("%s\n", line);
4                         }
5                       }
```

Listing 1.2 and 1.3 shows one of the information exposure scenario between the source 1.2 at line 5 and the sink 1.3 at line 3. The above code snippet is taken from CWE-526 test case [ref]. As one can see the system "$PATH$" variable in line 5 of source 1.2 is sent to the sink at line 3 of sink 1.3 $printf()$. Here $printLine()$ is a wrapper class for the function printf() in C which is located in another C source file. Therefore printf() is the trust-boundary for the system. As define by the getenv() function the return value of that function is set to be confidential. The return value of this funciotn will be propagated with the help of static execution and explicit IF. Whenever a symbolic variable which is confidential is about leave the trust-boundary or the sink $printf()$ a notification is sent to the interpreter. This is the condition for the bug tiggering based on which the IE checker detects a information exposure bug. In order restrict the confidential data to leave the sink.

The source code patch that has been introduced into the original code is represented with "+" and by using an italic font in listing 1.2 at line 7. In the code patch care has taken that the "PATH" variable is removed from the $getenv()$ function so that the return value with not be any confidential data. In this way we can restrict the confidential data to be sent to the sink 1.3 $printf()$. In general the code patch can be inserted at the place where the bug was found as shown in listing1.1 since the source and sink are the same source file we can insert the code patch at the place where the bug was found. But one can observe that in the lisitngs 1.2 and 1.3, the source and sink are two different files and there can be scenarios in which the sink can be used by different sources. So if we try to restrict the sink where the bug was detected, the program behavior of other programs will be changed. In order not to change the program behavior a pre-fix location has to found where the information flows from the source to sink and create a patch at that location as represented with "+" in listing 1.1. "-" represents that the statement has to removed from the source code at that location and "+" indicates that the statement has to be inserted at that location.

It is very hard task to find the right program variables in order to impose a condition with a code patch because sometimes we cannot impose the constraint of the variable which the checker reports but we also need to check the earlier location where the information flows. This is the "not-in-place" bug fix location. We need to use the localizer algorithm to find the pre-fix location. The insertion location and the source code patch pattern may also influence the overall program behavior. Therefore care must be taken that the patches are syntactically correct, compilable.

1.3. Security

Degree of protection or the resistance towards any harm form occurring is called security and it can be applied to any vulnerable and valuable things like people, community, nation or any organization. As given by ISECOM security provides [16]:

"a form of protection where a separation is created between the assets and the threat." These separations are generically called "controls"

Different types of securities related to IT realm:

- Computer security
- Internet security

- Application security
- Data security
- Information security
- Network security

Among all the above mentioned securities this thesis revolves arounf information security. Information security in general is the practice of protecting the information from unauthorized accesses, use, disclosure, modification.

A system's security is often based on some of the security properties like:

- Confidentiality : Availability of information to the authorized parties only. In this first one should know the information that should be protected and also define "authorized". We use authentication, authorization and access control in order to keep the information confidential. Authentication comes first, in which we verify if the person or the agent is the same they claim to be. This can be done for example through picture ID, password etc. Next comes authorization in which one verify the role of an agent. Finally, access control in order to see if the agent has the access rights on the information based on his/her role.

- Integrity : Unauthorized parties should not be able to modify any of the data. There are some computational techniques for preserving data integrity like: message digests(MD5 or SHA-1) , comparisons, message authentication and integrity codes(MAC/MIC) and checksums.

- Availability : The system should be made available all the time for the users. It should be in a operational state. This can be achieved by planning , determining the optimized computing and memory capacity and also prediciting the peak usage requirements. While designing, load balancing and fail-over solutions should also included while designing. System can be made unavailable by the attacker in many ways like the denial of service attacks which can bring down the network servers, applications. Attacker can delete some important information.

- Authenticity : It involves indentity proof which assures that the message or any transaction or any exchange of information is from the source it claims to be from. This authentication can be taken care in many ways like providing username and password for the agent. Cracking the password may be easy for the hacker therfore stronger passwords should be made.

  For secure information flow we deal mostly with confidentiality and integrity.

1.4. Basic Terminologies

- In-place fix location: It is the location where the bug was found and code patch has to be inserted at that location in order to remove the bug.

- Not-in-place fix location: It is not the location where the bug was found but it is the location prior to the bug location and the code patch has to be inserted at that location in order to remove the bug earlier.
- Code refactoring: Code refactoring is the process of changing the code in order to remove the bugs without changing the program behavior.
- Code patches: Code patches are code snippets or statements that are to be inserted into the code.
- Coverage: It is the extent to which any fix could handle all the bug-triggering inputs accurately. In general many inputs can trigger the bug [13].
- Disruption: Sometimes a fix can change the normal behavior of the program, therefore disruption measures the deviation of the program's behavior from its original intended behavior [13].

1.5. Contribution

Problem statement: Providing source code patches which can ”in-place” or ”not-in-place” quick fixes which can remove the information exposure bugs and can also be used independently. Later on the correctness of the patches is verified by running the bug checker again on the refactored code. In total the contribution of this thesis:

- A localizer algorithm to find the ”not-in-place” bug locations or the earliest quick fix locations.[ref]
- An algorithm for generation of ”in-place” and ”not-in-place” bug fixes.[ref]
- Source files differential views which shows the semi-automated patch insertion.[ref]
- Verifying the behavior of the patched program automatically. [ref]

2. Technical and Scientific Fundamentals

2.1. Sources of information flows with examples

Principal sources of information flows are of two types: [17] [24]

- Implicit flow
- Explicit flow

Implicit Flow: Implicit flow is one where secret data implicitly flows due to some control flow that is affected by secret values
Explicit Flow: Explicit flow is one where secret data explicitly flows to public contexts that is direct copying of the secrets.

Listing 2.1: Explicit Information Flow

```
1 L:=H;
```

Listing 2.2: Implicit Information Flow

```
1 if(H)then
2 L:=true;
3 else
4 L:=false;
```

In the above listings 2.1 and 2.2 both are semantically equivalent. In lisitng 2.1 information H explicitly flows into L via an assignment where as in lisitng 2.2 information doesnot explicitly flow but at the end H and L have the same information. Therefore we can say that H implicitly flows into L.

2.2. Different Attacks

There are some attack patterns which can exploit the illegitimate information flows [15] In order to define these attack patterns lets define P= $\{p_0, ..., p_{n-1}\}$ which represents a different partitions of an application executing on the top of the system kernel and H be the hardware that the system uses.Let A be the attacker who is trying to modify the data without any proper authorization. Different attack patterns are:

- $p_i \iff A$: This pattern is use insider attack pattern in which A has been granted access directly in order to extract the information from p_i [15], [9], [28].

- $p_i \Longleftrightarrow p_j$:This pattern exploits the flaws in the system policy. In this attack pattern information may be exchanged between p_i and p_j by exploiting the unauthorized access to the component [20].

- $p_i \Longleftrightarrow H$: This pattern exploits the covert channnels and storage channels. In this attack pattern A has physical access to the system under computation and extracts all the information directly that has been stored at H, from p_i [25], [14], [7].

- $p_i \Longleftrightarrow K \Longleftrightarrow A$: This pattern exploits the flaws in the system policy. In this attack pattern A uses the flaws in the implementation of p_i, K or H in order to extract the information from p_i.

- $p_i \Longleftrightarrow K \Longleftrightarrow p_j$: This pattern exploits the covert channels and the flaws in the system policy. In this attack pattern A's covert channel between p_i,p_j is established in different ways like encoding the message throught a hidden message storage in the shared resources, through timing behavior of the shared resources [20].

- $p_i \Longleftrightarrow K \Longleftrightarrow H \Longleftrightarrow p_j$: This pattern is use physical attacks and exploits covert channels. In this attack pattern A's covert channel between p_i,p_j is established through encoding a message and sending it throught H into the physical environment and then receiving at some other interface of H [18, 22].

2.3. Analysis Techniques

The process of analyzing the program behavior with respect to the properties of a computer program like safety, liveness, robustness and correctness is called program anaylsis. It's major focus is on program correctness which focuses on making sure that the program behavior is not changes(it does only what it is supposed to do) program optimization which focuses on reducing the resource usage in order to increase the program's performance [5]. In general the program analysis can be done in three differnt ways:

- Static analysis
- Dynamic analysis
- Hybrid analysis

Static Analysis: Analyzing the computer program without executing it, is called static program analysis. This analysis can be done on the source code or any other form of it like the object code. We use static analysis in building automated tools which involves human analysis like understanding the program,code reviews. Formal static analysis include model checking, data-flow analysis, abstract interpretation symbolic execution. [12] In general the static technique in order enforce secure information flow is by the usage of type systems. In many of the security-typed languages, the program variables and expressions types are augumented by using annotations which defines some policies on their usage. There are many enforcement schemes which prevents sensitive information

exposure, one among them is Denning-style enforcement which prohibits the implicit information flows through keeping track of the security level of the program counter and public side effects in secret contexts [24].

Dynamic Analysis:

Analyzing the computer program after executing , is called Dynamic program analysis. It can analyze the program in a single execution of the program and sometime may lead to the degradation of the performance of a program because of the runtime checks. Dynamic program analysis include testing, monitoring, program slicing. In this approach the security checks are done in a similar way as of static analysis. In dynmaic analysis it keeps a no assignment simple invariant to the low variables in high context.

Hybrid Analysis: It is a fusion of both static and dynamic analysis which offers the advantage of more information to be available at runtime on the actual execution trace, while it also keeps the overhead of runtime moderate because some static information has been already gathered.

In this thesis, the main concern is about the symbolic variables tainting and the propagation taint variables which helped in defining the location to insert the fix and to decide upon the fix.

The propagation of the taint variable values can be done using hybrid, static or dynamic taint-analysis.

Static Taint-Analysis

Static Taint Analysis (STA) is done taking into account all the possible execution paths of a program. In general it doesnot provide the interation of the environment and the information about the runtime. Therefore the interaction has to be simulated. STA tools are user input dependent[42]. Therefore we have models already developed which simulates their execution during the static execution. Developed tool is similar to PREfix [43] i,e. both trace the distinct execution paths in order to simulate the function calls and also of each operator in the paths.

We have developed the function models in which the sinks and sources are directly defined and also simulated the real execution using the function models.

Dynamic Taint-Analysis Dynamic Taint-Analysis(DTA) is another approach in which the taintness of a variable is computed during the program execution. In DTA there is the possibility to gather the data flow information but only for one path at a time. Therefore, it leads to less path coverage and the dependencies of the program variables can be calculated.

In our approach we cannot make use of DTA because we need high path coverage and also all the satisfiable possible execution paths. We used SMT solver in order to achieve this.

Hybrid Taint-Analysis Hybrid Taint Analysis(HTA) includes both the above mentioned taint analyses. In this approach we can have the information of static analysis during dynamic analyses and vice-verse which can overcome some of the disadvantages of both the approaches.

In the first approach all the executable paths are explored similar to the static execution and finally interleaves concrete execution along with the symbolic execution. These concrete values are then used in order to allow the algorithm to proceed when complex constrains are encountered.

In the second approach the execution of the program is monitored and the information available after static analysis is used in order to decide upon when it is safe to halt tracking the confidential variables for example. In general we statically taint the variables in our function models and use symbolic execution in order to find all the candidate paths for the IE bugs and also the confidential variables based on explicit IF's.

2.4. Program Transformation

Any program is an object which is structured with semantics. This structure can help us to transform a program and semantics gives us the opportunity to compare programs and also to verify the validity of the transformations. Because of this structure and similar semantics, all the programming languages can be clustered into different classes. The main goal of this program transformation is to define the program transformations in such a way that they can be reused across wide range of languages. It is used for compiler construcion, software visualization, document generation and in many other areas of software engineering. This program transformation can be of two types: translating source language to different target language, rephrasing the source language to the same target language. Apart from monitoring the execution of the code, static or dynamic analysis

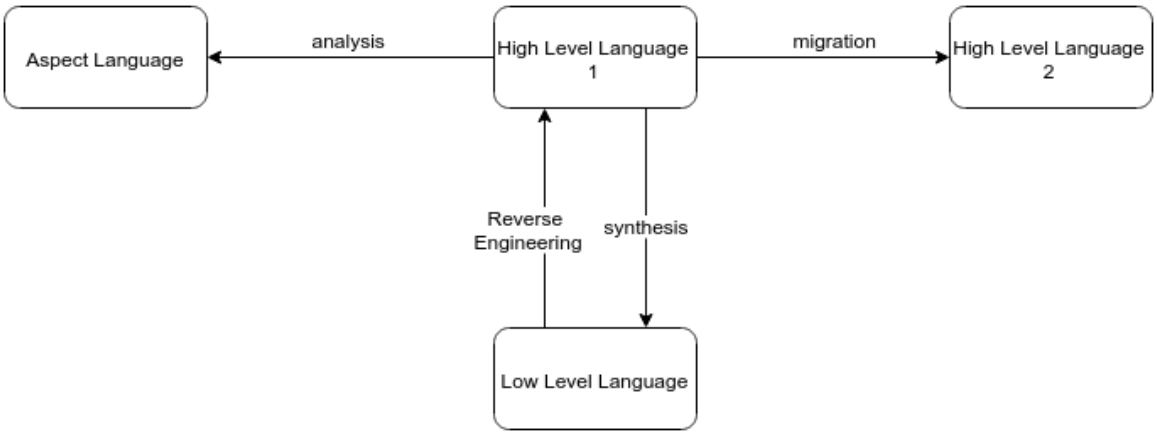


Figure 2.1.: scenarios of translation

can filter, rewrite or wraps the code inorder to enforce some integrity constraints on the code. The main idea behind the code transformation is to compute publicly by replacing all the secret values with dummy variable. Then run the secret computation under the restrictions. Code transformation is an operation that generates another program which is semantically equivalent to the original code with respect to the formal semantics. But sometimes the transformed code can result to be different in many ways [27]. This transformation can be done manually or using transformation systems. This can be specified as procedures that are automated which can modify the data structures like abstract syntax tree in which each statement in the source code is represented as a node in the abstract syntax tree. Requirement of this program transformation is to effectively process the programs written in any of the existing programming languages.

3. Related Work

Part II.

Design and Development

4. System Overview

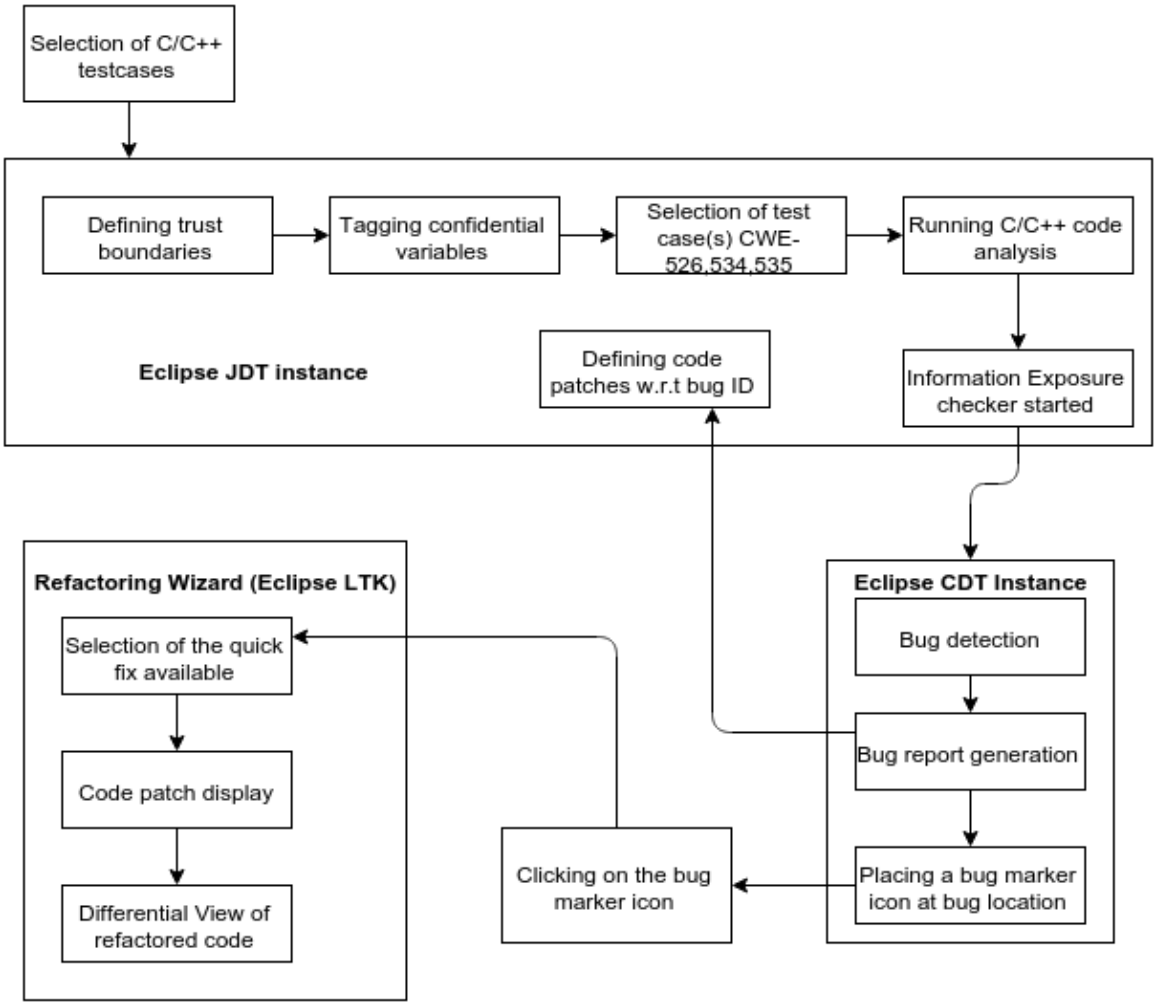


Figure 4.1.: System Workflow

Exisiting information-flow checker is based on the static analysis engine which can be scaled to many other types of checkers and also for large test cases. The system workflow is depicted in figure 4.1. First, all the C/C++ test programs have to be selected which are to be imported in the workspace. Second, all the confidential variables in the test programs have to tainted and the trust boundaries need to be defined. Third, after running the eclipse java application, a new eclipse CDT instance is launched in which we can find already imported test programs. Then a test program is selected and a sub menu "Run C/C++ Code Analysis" option has to selected after right clicking on it. Fourth, a refac-

toring wizard appears after clicking on the bug marker that is placed after running the information exposure checker. Basically the IE checker is run as a Eclipse plug-in project. Once this project is started a new eclipse CDT instance is launched as shown in the figure 6.1 containing 90 Test Programs(TP) namely CWE-526,534,535. A Graphical User Interface(GUI) is provided by the codan API. After running the checker the results can be seen in the view represented by figure 6.2. We can also use one of the Codan API feature in order to configure the bug report's representation to be Warnings, errors or infos. The end result of the IF checker is the bug report containing the description of the bug, file and the path which can lead to the location of the bug on clicking on the warning as represented in figure 6.2 marked with number 3. After reaching the bug location in the file, upon clickin on the bug marker icon a refactoring wizard is launched containing the quick fix options as shown in figure 6.4. After selection any one option, a window with the selected code patch is displaye and upon clicking next a differential view with both the buggy program and refactored code is displayed as shown in figure 6.6. After clicking on finish the code is refactored.

5. Localization Algorithm

Algorithm 1: Decision on starting the localization algorithm for searching quick fix locations and for generating the code patches

Input: $location_{node}$, $filename_{node}$,$sourcefile_{node}$
Output: Refactorings set $R_{set} := \{r_j | \ 0 \leq \text{j} < 2 \ \}$
$N_{set} := \{n_t | \ 0 \leq \text{t} \leq \text{n}, \forall \ \text{n} \geq 0 \ \}$;//set of nodes
R_{set} :=$\emptyset$;N_{set} :=$\emptyset$;
//initialising nodes set and refactoring set to empty
$j = 0$;
function CheckerReport $(location_{node}, filename_{node}, sourcefile_{node})$
if $filename_{node} \neq sourcefile_{node}$ **then**
  $localizer.start()$;//Starting the localizer
else
  $N_{set} := N_{set} \bigcup node$;//adding node to node set
  $r_j := refact(node)$;//Refactoring the node
  $R_{set} := R_{set} \bigcup r_j$;//adding refactored node
  $j =: j + 1$;
end

Algorithm 2: Part-I: Localizer algorithm and generating code patches

nonumbering
Input: Set of satisfiable paths Sat_{paths} :={$spath_i$| 0 ≤ i ≤ n,∀ n ≥ 0 };
Output: Refactorings set R_{set} := {r_j| 0 ≤ j < 2 }
N_{set} := {n_t| 0 ≤ t ≤ n,∀ n ≥ 0 };//Set of Nodes
$pathloc_{list}$:= {$path_k$| 0 ≤ k ≤ n,∀ n ≥ 0 };//Set of paths list with filename and line number of every node in the path
$path_{location}$:=∅ //list of nodes in the path with filename and linenumber
R_{set} :=∅;// initialising refactorings set
N_{set} :=∅;// initialising nodes set
function Localizer ()
while *((Sat_{paths}.hasNext))* **do**
  for ***each*** $node_j$ ***in*** $spath_i$ **do**
    if $node_j.hasTranslationalUnit()$ **then**
      $filename := node_j.getfilename()$;
      $linenumber := node_j.getStartingLinenumber()$;
    else
      $filename := null$;
      $linenumber := -1$;
    end
    $path_{location} := path_{location} \bigcup getloc(filename, linenumber)$;//Add location of each node in the path
  end
  $pathloc_{list} := pathloc_{list} \bigcup path_{location}$;//Updating the paths list
  for ***each*** $buginfo$ ***in*** $spath_i$ **do**
    //verifying if the path has the bug info
    $bugloc_{new} := bugloc(path_{location})$;
    $loc_{bugloc} := loc_{bugloc} \bigcup bugloc_{new}$;//Updating the localizer bug locations
  end
  $path_{location}$:=∅;//emptyping the pathlocation
end
for ***each*** $bugloc$ ***in*** loc_{bugloc} **do**
  for $i \leftarrow 1$ ***to*** $bugloc.pathdepth$ **do**
    //we count the number of times the node appears in the buggy paths
    if $node_i$ ***in*** $path_location$ *of* loc_{bugloc} **then**
      $count := count + 1$;//Updating the count of the node
    else
      $count := 0$;//Updating count to 0 if it appears for the first time
    end
  end
  $bugmap := bugmap.buginfo.put(node_i, count)$;//maintaing a map between the node and its count
end

Algorithm 3: Part-II contd...

```
for each buginfo in bugmap do
    for each path_k in pathloc_list do
        for each path_l in loc_bugloc do
            if path_k ≠ path_l then
                // remove all definitely good nodes (from paths without
                the bug) and nodes without a useful location
                bugmap.remove(path_k);
            else
                continue;
            end
        end
    end
end
for each node in bugmap do
    // Get the node with the maximum count
    if node.getcount() > max then
        max := node.getcount()
    end
    if node.getcount() = max then
        suspicious_node := suspicious_node ⋃ node;// node with max count is the
        suspicious node
        node_map := node_map ⋃ getpath(node);// save the node and its
        corresponding path in a hashmap
    end
    if node_map.size ≤ 1 then
        N_set := N_set ⋃ node;//adding node to node set
        r_j := refact(node);//Refactoring the node
        R_set := R_set ⋃ r_j;//adding refactored node
        j =: j + 1;
    else
        //if nodes belong to same path then find first node in the path
        if nodeinnode_mapbelongtosamepath then
            N_set := N_set ⋃ firstnode;//adding node to node set
            r_j := refact(node);//Refactoring the node
            R_set := R_set ⋃ r_j;//adding refactored node
            j =: j + 1;
        else
            //if nodes belong to different paths create refactoring for
            each node in different paths
            N_set := N_set ⋃ node;//adding node to node set
            r_j := refact(node);//Refactoring the node
            R_set := R_set ⋃ r_j;//adding refactored node
            j =: j + 1;
        end
    end
end
```

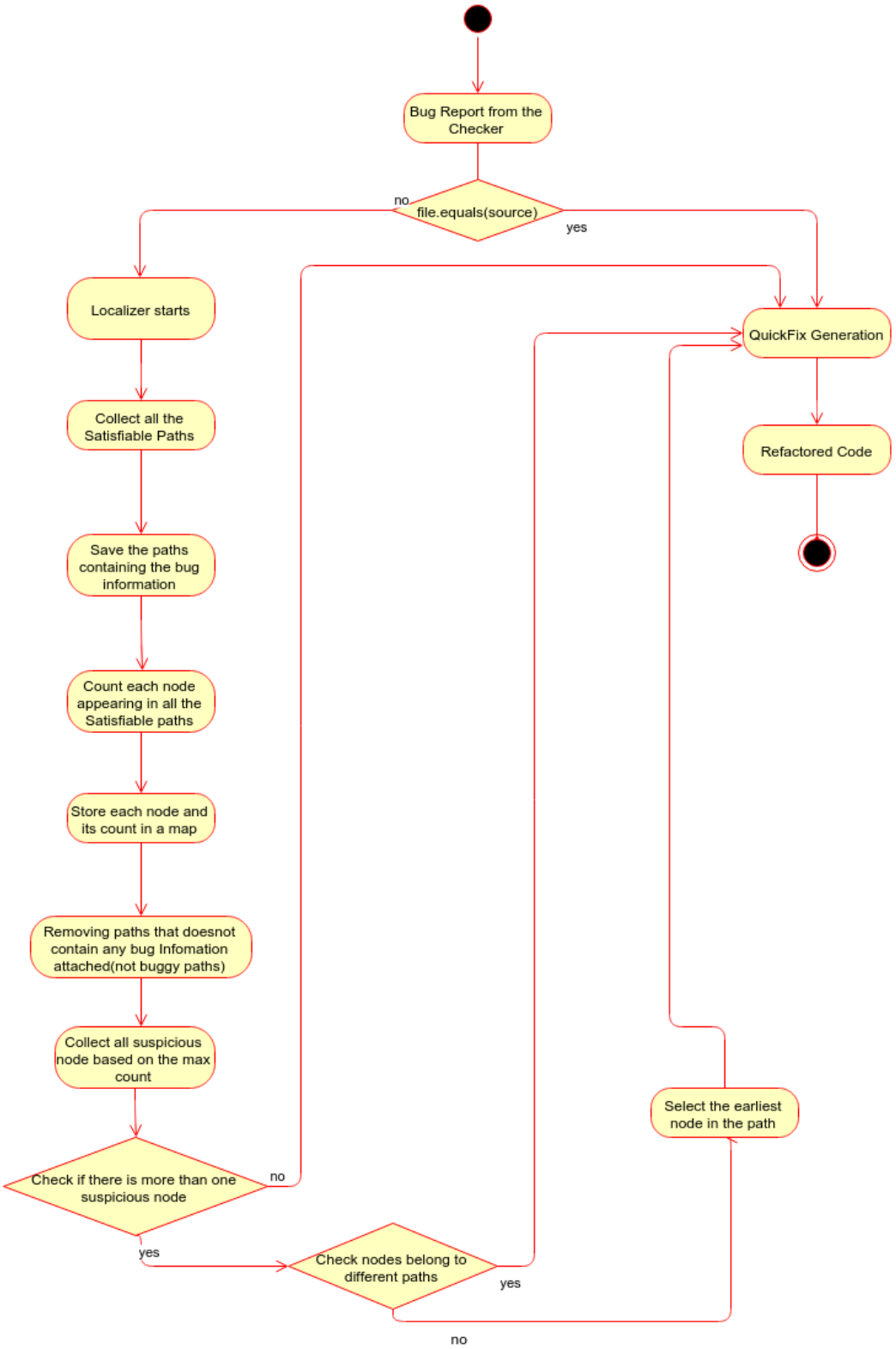


Figure 5.1.: Activity diagram of the localizer

6. Approach

6.1. About the Tool

Developed tool bridges the gap between the bug report that is been reported by the already exisitng information exposure bug checker and the automatically generated one or more bug fixes which helps in the removal of the information exposure bugs. Removal of the bug is tested again by re-running the information exposure checker on the patched program. Bug fixes includes structure of the fix, location where the fix has to be inserted and the values used in the fix patches. The bug localization for some files is done by the information exposure checker which is integrated in our static analysis engine and by the localizer for other files which contain the wrapper functions. The checker basically returns the file name, line number and also a bug ID which is unique for every kind of bug. Therefore along with the report, the checker also puts a marker at the bug location. After clicking the icon the refactoring wizard starts.

6.2. Bug Detection with tainting and triggering

Function models were used in order to describe the behaviour of of the C/C++ libraray function calls. Static analysis engine's implementation is dependant on these function models. Every function model class will contains five methods and also it implements the IFctModel. The methods are: the constructor, getName() used for returning the function name, getLibrarySignature() for returning the whole header of that function same as it is in the C standard library defined, exec(SymFunctionCall call) used for executing the function calls satically in which the variables can be tainted and also the trust boundaries are defined which could help in notifying the checker, getSignature() used for returning a SymFctSignature object which contains the data types of the parameters of that particular function call and also the return type of the function itself. Major difference between exec() method of the function models printf() and getenv is that the IF checker is notified that a trust-boundary is about to be passed in printf() function model and in getenv() the return type of the exec() method is set to be confidential. In the same way the source and sink which are contained in CWE-534/535 are implemented.

Listing 6.1: Function model of getenv()

```
1
2
3   public Mgetenv(Interpreter ps) {
4     this.ps = ps;
5     HashMap<String, Comment> commentMap = AnnotationExecution
6         .getCommentsMap();
7     if (commentMap == null) {
```

```
8         MyLogger.log_parser("Comment Map is Empty");
9       } else {
10        comment = commentMap.get(getLibrarySignature());
11        if (comment == null) {
12          MyLogger.log_parser("Mgetenv comment is null");
13        } else {
14
15          if (comment.getType().equals(Comment.singleLineComment)) {
16            annotationType = Comment.singleLineComment;
17            listSingleParamterAnnotation = AnnotationParserUtil
18                .getSingleLineParamterAnnotation(comment);
19            listSingleFunctionAnnotation = AnnotationParserUtil
20                .getSingleLineFunctionAnnotation(comment);
21            if (listSingleParamterAnnotation == null) {
22            } else {
23            }
24          } else if (comment.getType()
25              .equals(Comment.mutilineLineComment)) {
26            annotationType = Comment.mutilineLineComment;
27            listMultiParamterAnnotation = AnnotationParserUtil
28                .getMultilineParameterComment(comment);
29            listMultiFunctionAnnotation = AnnotationParserUtil
30                .getMultilineFunctionComment(comment);
31          }
32        }
33      }
34    }
35
36    public String getName() {
37      return "getenv";
38    }
39
40    private static String getLibrarySignature() {
41      return "extern char *getenv (__const char *__name) __THROW
            __nonnull ((1)) __wur;";
42    }
43
44    public SymFunctionReturn exec(SymFunctionCall call) {
45      ArrayList<IName> plist = call.getParams();
46      SymPointerOrig isp = ps.getLocalOrigSymPointer(plist.get(0));
47      IName nebn = new EnvVarName();
48      SymIntOrig sb_size = new SymIntOrig(new ImpVarName());
49      SymArrayOrig sb = new SymArrayOrig(nebn, sb_size);
50      try {
51        sb.setElemType(eSymType.SymPointer);
52        SymVarSSA ssa = (SymVarSSA) ps.declareLocal(eSymType.SymPointer,
53            null);
54        isp_ssa = (SymPointerSSA) ps.ssaCopy(isp);
55        isp_ssa.setTargetType(eSymType.SymPointer);
56
57        if (comment != null && annotationType != null
58            && annotationType.equals(Comment.mutilineLineComment)
```

```
59            && listMultiFunctionAnnotation != null
60            && listMultiParamterAnnotation != null) {
61          String function_property =
                AnnotationUtil.FUNCTION_PROPERTY_SOURCE;
62          if (listMultiFunctionAnnotation.get(0).getAtribute()
63               .equals(function_property.toString())) {
64
65            if (listMultiParamterAnnotation.get(0).getIndex() == 0) {
66              String security_level =
                    AnnotationUtil.PARAMTER_SECURITY_LEVEL_CONFIDENTIAL;
67              if (listMultiParamterAnnotation.get(0)
68                   .getSecurityType()
69                   .equals(security_level.toString())) {
70                isp_ssa.setConfidential(true);
71              } else {
72                isp_ssa.setConfidential(false);
73              }
74            }
75          } else {
76          }
77        }
78        isp_ssa.setTarget(sb);
79      } catch (Exception e) {
80        e.printStackTrace();
81      }
82      return new SymFunctionReturn(isp_ssa);
83    }
84
85    public SymFctSignature getSignature() {
86      SymFctSignature fsign = new SymFctSignature();
87      fsign.addParam(new SymPointerOrig(eSymType.SymArray, new
            Integer(1)));
88
89      fsign.setRType(new SymPointerOrig(eSymType.SymPointer, new
            Integer(1)));
90      return fsign;
91    }
92
93  }
```

Listing 6.2: Function model of printf()

```
1
2     public Mprintf(Interpreter ps) {
3       this.ps = ps;
4
5       commentMap = AnnotationExecution.getCommentsMap();
6       if (commentMap == null) {
7         MyLogger.log_parser("Comment Map is Empty");
8       } else {
9         comment = commentMap.get(getLibrarySignature());
```

```
10        if (comment == null) {
11          MyLogger.log_parser("Mgetenv comment is null");
12        } else {
13
14          if (comment.getType().equals(Comment.singleLineComment)) {
15            annotationType = Comment.singleLineComment;
16            listSingleParamterAnnotation = AnnotationParserUtil
17                .getSingleLineParamterAnnotation(comment);
18            listSingleFunctionAnnotation = AnnotationParserUtil
19                .getSingleLineFunctionAnnotation(comment);
20            if (listSingleParamterAnnotation == null) {
21            } else {
22            }
23          } else if (comment.getType()
24              .equals(Comment.mutilineLineComment)) {
25            annotationType = Comment.mutilineLineComment;
26            listMultiParamterAnnotation = AnnotationParserUtil
27                .getMultilineParameterComment(comment);
28            listMultiFunctionAnnotation = AnnotationParserUtil
29                .getMultilineFunctionComment(comment);
30          }
31        }
32      }
33    }
34
35    public String getName() {
36      return "printf";
37    }
38
39    public static String getLibrarySignature() {
40      // contained in stdio.h
41      return "extern int printf (__const char *__restrict __format,
            ...);";
42    }
43
44    public SymFunctionReturn exec(SymFunctionCall call) {
45      ArrayList<IName> plist = call.getParams();
46      SymArraySSA newdestarr = null;
47      SymArraySSA sourcearr = null;
48      try {
49        int t = 0;
50        t = plist.size() - 1;
51
52        SymVarSSA sourceSSA = (SymVarSSA) ps
53            .resolveOrigSymVar(plist.get(t)).getCurrentSSACopy();
54
55        if (sourceSSA != null && comment != null && annotationType !=
              null) {
56
57          if (annotationType.equals(Comment.mutilineLineComment)
58              && listMultiFunctionAnnotation != null
59              && listMultiParamterAnnotation != null) {
```

```
60              String function_property =
                    AnnotationUtil.FUNCTION_PROPERTY_SINK;
61              for (FunctionComment element : listMultiFunctionAnnotation)
62                if (element.getAtribute().equals(
63                      function_property.toString())) {
64                  ps.notifyTrustBoundary(sourceSSA);
65                } else {
66                }
67            } else {
68            }
69        }
70      } catch (Exception e) {
71        e.printStackTrace();
72      }
73      IName retname = new ImpVarName();
74      SymPointerOrig retvalorig = new SymPointerOrig(retname);
75      retvalorig.setTargetType(eSymType.SymPointer);
76      SymVarSSA ssa = (SymVarSSA) ps.declareLocal(eSymType.SymInt, null);
77      return new SymFunctionReturn(ssa);
78    }
79    public SymFctSignature getSignature() {
80      SymFctSignature fsign = new SymFctSignature();
81      fsign.addParam(new SymPointerOrig(eSymType.SymArray, new
            Integer(1)));
82      fsign.addParam(new SymPointerOrig(eSymType.SymArray, new
            Integer(1)));
83
84      fsign.setRType(new SymPointerOrig(eSymType.SymPointer, new
            Integer(1)));
85      return fsign;
86    }
87
88  }
```

Function model that currently exists in SAE are some of the C functions: atoi(), fclose(), fgets(), fwgets(), fgetws(), fopen(), gets(), memcpy(), mod(), puts(), rand(), srand, strcpy(), strlen(), time(), wcscpy(), wcslen(). Funtion models which were used in this thesis are getenv() as the source and printf() as sink from the test programs CWE-526. LogonUserA(), LogonUserW() as source and fprintf(), fwprintf() as sink from the test programs CWE-534 and CWE-535. These function models were used for both tainting and also triggering.

6.3. Semi Automated Patch insertion Wizard

The information exposure checker after detecting a bug, it puts a bug marker which is a yellow color bug icon as shown in the figure 6.3. It is placed to the left side of the C/C++ statement at the place where it contains a information exposure bug. Therefore, after clicking on the bug marker, one can start the wizard for the code refactoring. This wizard contains three user pages. In the first page, the user can make a selection of either ”in-place” (latest) or ”not-in-place” (earliest) fixes as shown in figure 6.4. In the second page

one can have a look at the code patch that is going to be inserted in the code for refactoring as shown in figure 6.5. In the third page one can have a differential view depicting the difference between the refactored code and the original source code files as shown in figure 6.6. One can navigate through all the three pages using the buttons ”Back”, ”Next”, ”Cancel” in order to see the changes inserting ”in-place” and ”not-in-place” fixes. Finally, once the decision has been made on the selection user has to click on ”finish” button in order to make sure that the selected patch is written to the file and then the wizard exits.

6.4. Tool Demo

The refactoring process is carried out stepwise. Following windows shows all the steps involved:
1. After running the Eclipse Java application, a CDT instance is displayed. From this instance one can select a testcase that are imported from the Juilet testsuite as shown in figure 6.1. After the selection of testcase, right click on the testcase. Once clicked, a window with many option appear. Among the options click on ”Run C/C++” Code Analysis. Once the analysis is run then the Information Exposure checker is started and checks for the bugs.

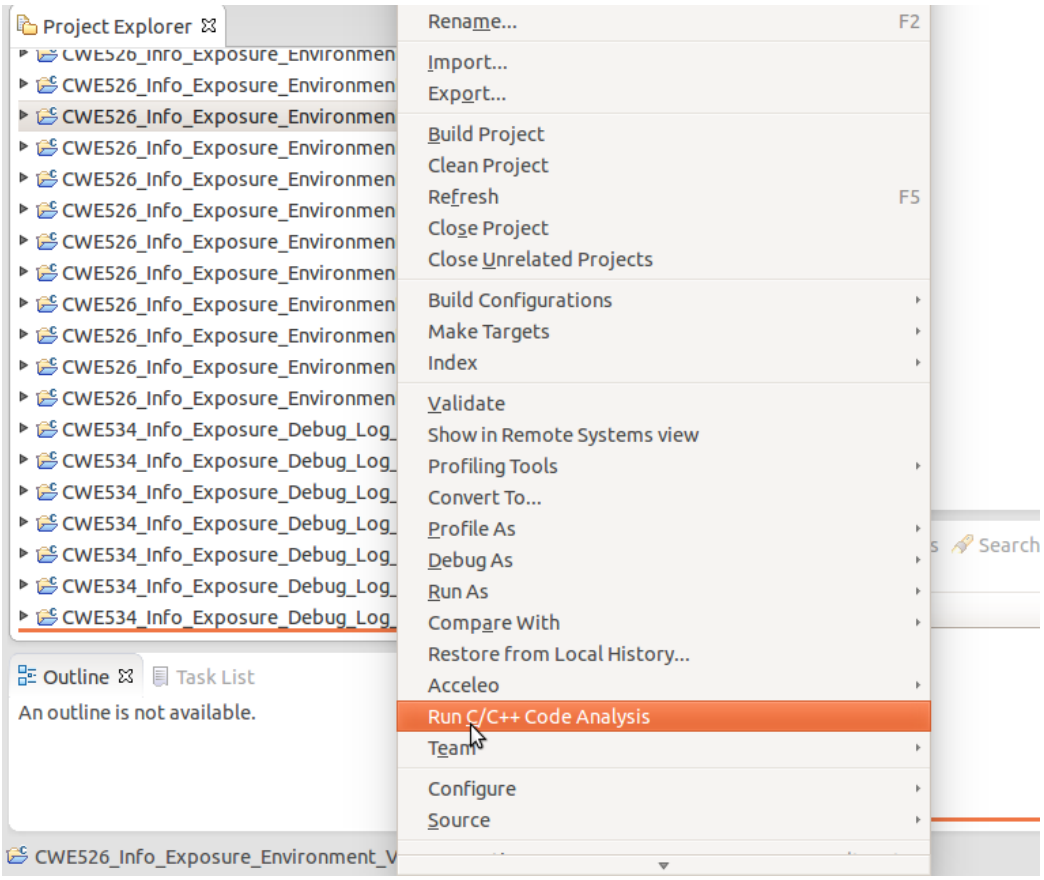


Figure 6.1.: C/C++ testcase selection and running analysis

2. After the checker checks for the information exposure bugs. The next window consists of three views: A view with yellow color marker placed at the location where the bug was detected as shown in the figure 6.2 annotated with **2**. A view displaying all the bugs that are found, annotated with **3** and by clicking on any of the bug details in the window, it takes us to the location where the bug is located. A view with the the list of all the test-cases annotated with **1**

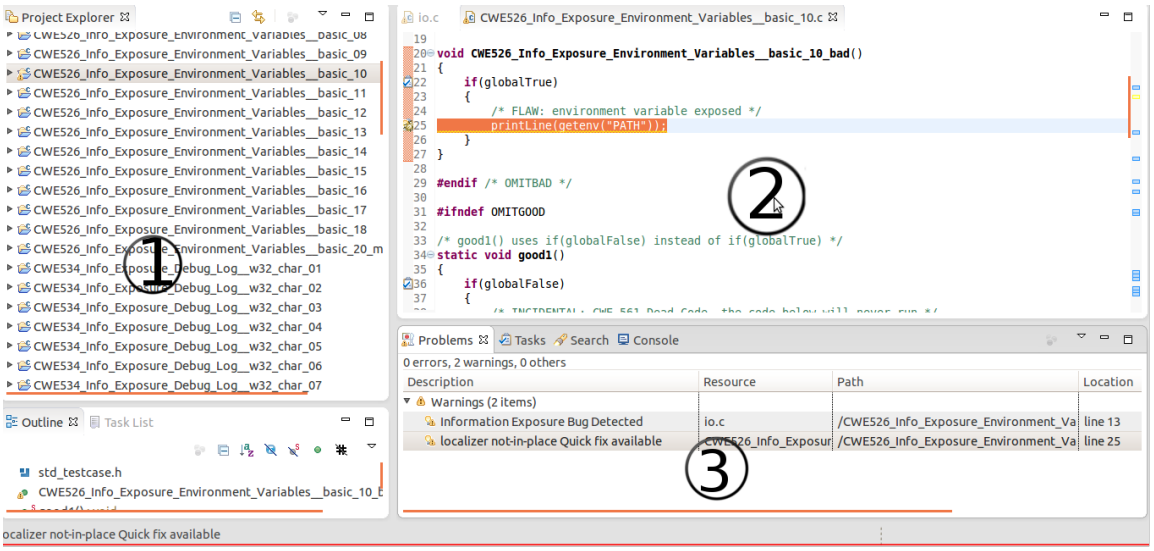


Figure 6.2.: Marker at the bug location

3. When a user places the mouse cursor on the yellow icon which is the bug marker. A small window with a message the problem message and description is displayed as shown in the figure 6.3 e.g "Quick fix is available" and "Problem description: localizer not-in-place Quick fix available"

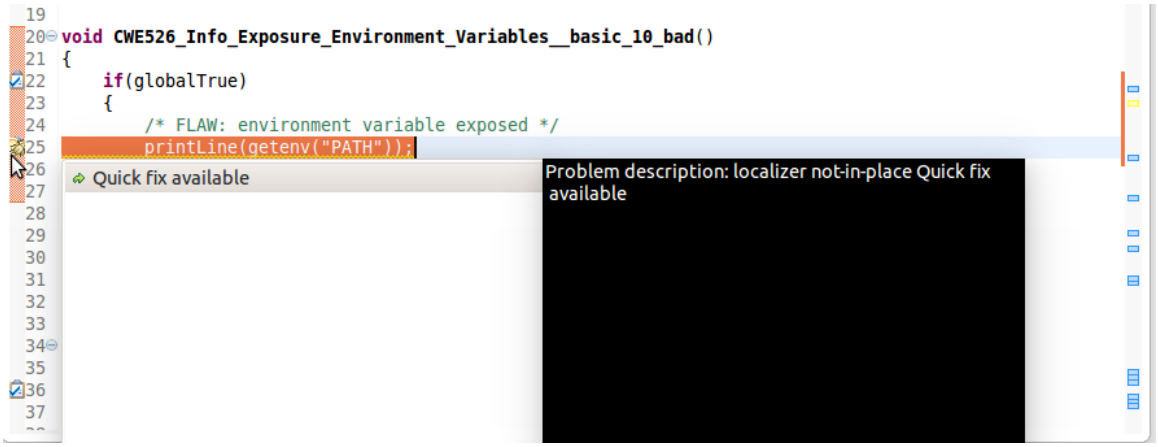


Figure 6.3.: Click on the marker icon

4. After clicking on this bug marker icon a window showing the following option is displayed with earliest quick fix(not-in-place) and latest quick fix(in-place). After selecting any one of the option and clicking on ”Next” button will take the user to the next window. By clicking on the ”Cancel” button will stop the refactoring process as shown in figure 6.4.

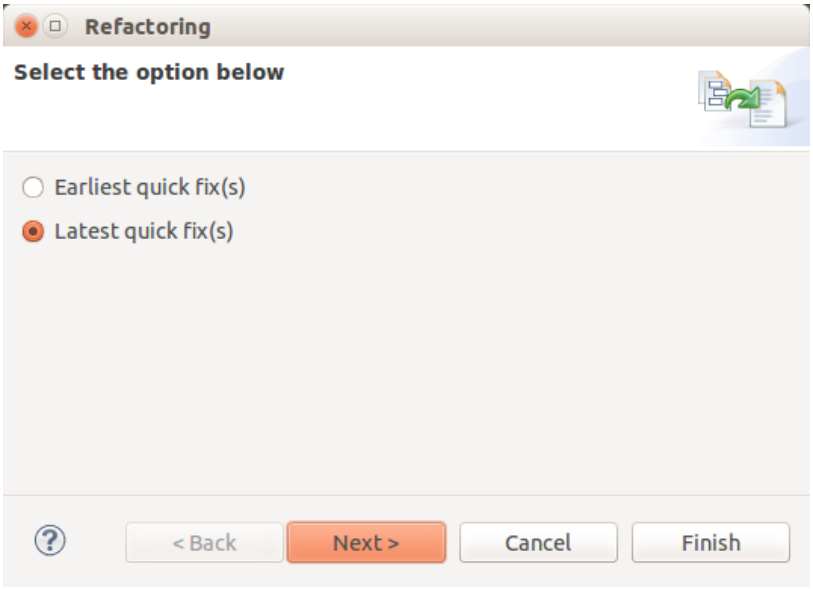


Figure 6.4.: Refactoring Wizard with two options

5. User can then see a window which displays the code patch based on the selection in the previous window . Here ”printLine(”Confidential data has been restricted”)” is the code patch. User can go to the previous window by clicking on ”Back” button, ”Finish” to proceed to the next window, ‘ ”Cancel” to cancel the refactoring process as shown in figure 6.5.

6. Once the user clicks on ”Next” from the previous window figure 6.5, then a window with differential view is displayed which shows the difference between the original code and the refactored code by highlighting the place where the code was refactored as shown in figure 6.6.

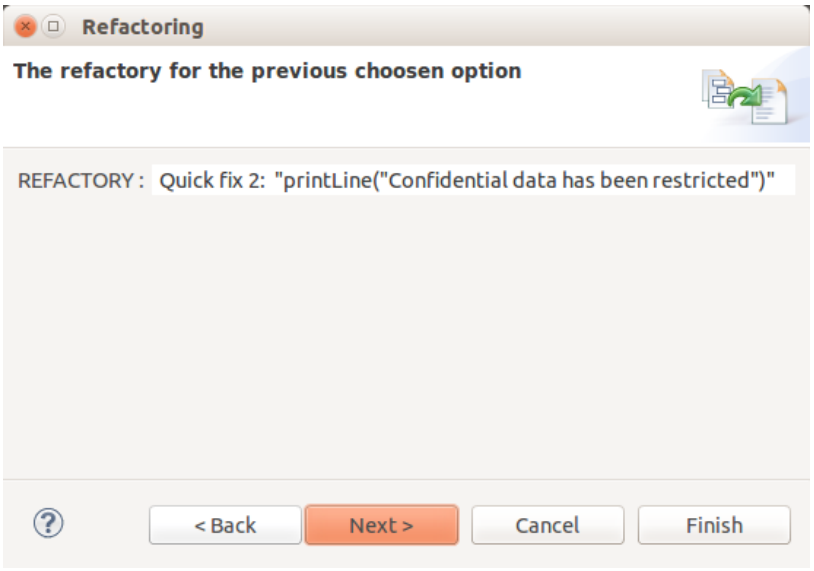


Figure 6.5.: Patch according to the previous selection

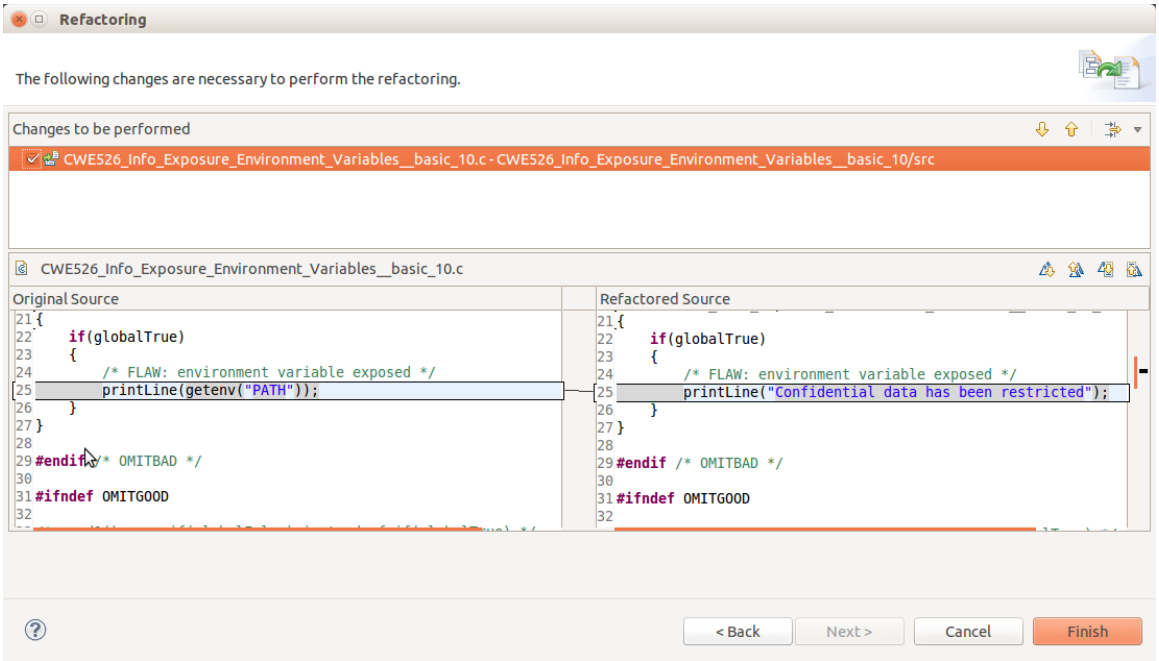


Figure 6.6.: Differential view of the patched and unpatched programs

7. Implementation

Developed information exposure bugs fixing tool was integrates into already existing Static Analysis Engine(SAE) which is a plugin of Eclipse IDE. A Refactoring wizard was developed in order to semi-automatically insert the generated code patch into the buggy program in order to fix the errors. This was developed using the Eclipse Language Tool Kit (LTK), JFace and CDT. Refactoring of code is done in two steps: First, performed bug detection analysis where if the bug was found a marker icon is placed at the bug location. This is the ”in-place” location. If the marker is put in a file other than the original source code file then we start the localization algorithm. This is because, other files may contain wrapper functions where the bug was detected but we cannot fix the bug at that location since this wrapper function may be used by other functions where there is no confidential data being sent. Therefore, we have fix the bug at a place in the source code file from where the confidential data leaves. This is the ”not-in-place” location. So after running the localizer algorithm we get some suspicious nodes. From these suspicious node again we propagate through the node in order to verify if some confidential information flows through it by checking it is tagged with ”confidential”. Once we find the exact node from where the information started flowing, again a marker is placed at that location. Once the bug fix locations are found, then we start the refactoring wizard in order to insert the code patches into the buggy code.

Part III.

Evaluation and Discussion

8. Empirical Evaluation

Main goal of this chapter is to assess how much is our IE checker is efficient in terms of the count that it is able to detect true-positives, false-negatives, false-positives, execution times and also its ability to fix the detected bugs. Along with this, highlighting the research work that has been carried out in this thesis. For the purpose of evaluation, we have used the IE checker and the Juliet test cases CWE-524/535/536. All the results from this evaluation will be represented in tables and also in graphs.

8.1. Test Setup

In order to fix the detected IE bugs, developed refactoring tool was used for generating the two types of patches and then inserting in the code program. Exisiting IE checker[ref] was used in order to detect the IE bugs in the code programs and also to classify the bug depending on the bug description and its unique ID. Used 64-bit Linux Kernel 3.13.0-32.57, Intel i5- 3230 CPU @ 2.60GHz Ã 4 was used as a test system. Calculated the times needed to generated all the patches and also the total time for detecting the bug. Measured the times in milliseconds.

8.2. Methodology

CWE- 526/534/535 were used as the test cases for running our already existing IE checker because these test cases contain the bugs for which the IE checker was built. All the test cases were also publicly online available in the Juliet test suite [ref]. The number of test programs present in CWE-526 are 18 Test Programs(TPr), CWE-534 has 36 TPr and CWE-536 has 36 TPr. All these test programs CWE-526/534/535 were exported in to the eclipse workspace by already created Eclipse CDT project instance.

After all the test programs were imported into the Eclipse CDT workspace, the existing IE checker which is an eclipse JDT plugin was made to run automatically for each test program available in the workspace. This can be done by selecting the submenu ”Run C/C++ Code analysis” after right clicking on the selected test program. The following time were measured and represented in the respective tables: time for running the checker for detecting the bug was measured [ref], time for refactoring[], execution time for the test programs which belong to the test case, these were measured for all the analyzed test programs. Also the number of true positives, false positives, true negatives and false negatives were are also calculated and reperesented in the table[].

[// here write how all the values in the table were calculated]

It was already know the number of test programs that should have the bug marker icon in the test program after running the IE checker using static analysis and also the count on how many should not have the bug marker icon (5 out of 90 test programs will not contain

the bug icon since it was not possible for running the static analysis). Usually a bug icon is similar to the image of a bug and the description of the bug is shown in the console view with a triangle symbol with an exclamation mark inside it.

Different types of messages that can be seen are:

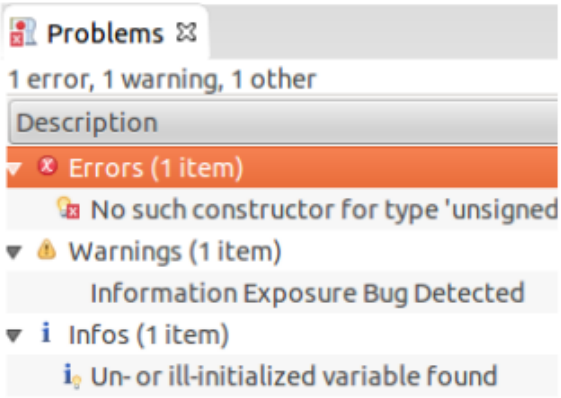


Figure 8.1.: Types of possible messages

1. Errors
2. Warnings
3. Infos

Errors are represented with red circle with a cross inside icon, Warnings are represented with a traingle and exclamation mark inside icon, infos is represented with i symbol icon. In this IE checker we do not use errors and infos but only the warnings where the bug descripted will be displayed in the console view of the Eclipse CDT instance after running the IE checker and detecting the bugs as shown in figure 8.1. Therefore once the bugs are detected and the bug mark icon is placed, one can see the traingle icon with exclamation mark inside in the console view. Once we click on the message that has been generated by the Information exposure bug report represented in figure 8.1 then the cursor will be pointing to the line number where the bug was found. There are different modes in which the IE checker can be configured in order to launch the checker as shown in figure 8.2. This bug triggering modes can be very useful during the software development by giving a chance to the developer on how to control and when should eclipse trigger the bug detection analysis. This will help the developer to avoid insertion of bugs during the software development.

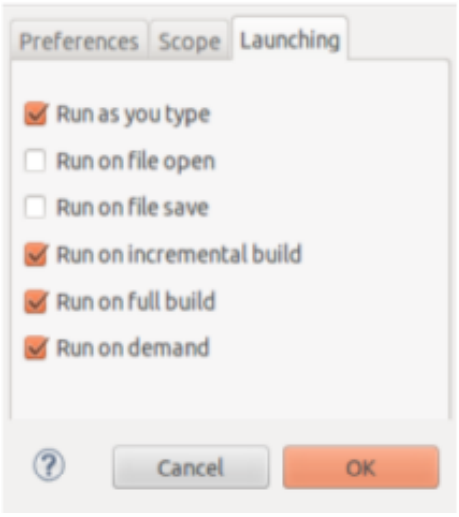


Figure 8.2.: Different modes

Test Program	PGT	TT	No; of Paths	IP	Total no; of nodes	BDT
CWE526-env-01	0.014	7.442	8	0	250	7.428
CWE526-env-02	0.02	4.082	12	0	498	4.062
CWE526-env-03	0.013	3.603	12	0	498	3.59
CWE526-env-04	0.017	3.306	12	0	498	3.289
CWE526-env-05	0.017	3.456	12	0	498	3.439
CWE526-env-06	0.01	3.455	12	0	498	3.445
CWE526-env-07	0.011	3.386	12	0	500	3.375
CWE526-env-08	0.01	3.169	12	0	564	3.159
CWE526-env-09	0.008	3.241	12	0	498	3.233
CWE526-env-10	0.012	3.373	12	0	498	3.361
CWE526-env-11	0.014	3.207	12	0	564	3.193
CWE526-env-12	0.017	3.116	20	0	1001	3.099
CWE526-env-13	0.013	3.236	12	0	498	3.223
CWE526-env-14	0.014	3.239	12	0	498	3.225
CWE526-env-15	0.014	3.272	12	0	507	3.258
CWE526-env-16	0.005	3.182	10	0	356	3.177
CWE526-env-17	0.005	3.303	12	0	533	3.298
CWE526-env-18	0.0	0	0	0	0	0

Table 8.1.: Measured values of CWE-526

Testcase	RT	TT	No; of Paths	IP	Total no; of nodes
CWE534-debug-char-01	0.004	5.644	87	0	7560
CWE534-debug-char-02	0.043	9.688	372	0	48645
CWE534-debug-char-03	0.01	8.649	372	0	48645
CWE534-debug-char-04	0.013	8.62	370	0	48552
CWE534-debug-char-05	0.016	8.487	372	0	48645
CWE534-debug-char-06	0.015	8.264	371	0	48609
CWE534-debug-char-07	0.014	8.352	372	0	48645
CWE534-debug-char-08	0.018	6.421	198	0	30141
CWE534-debug-char-09	0.051	10.177	372	0	48645
CWE534-debug-char-10	0.022	7.894	278	0	36495
CWE534-debug-char-11	0.017	8.891	372	0	51699
CWE534-debug-char-12	0.007	7.363	308	0	32341
CWE534-debug-char-13	0.015	48.654	372	0	48645
CWE534-debug-char-14	0.02	8.517	372	0	48645
CWE534-debug-char-15	0.011	8.555	372	0	49014
CWE534-debug-char-16	0.003	4.266	92	0	8551
CWE534-debug-char-17	0.002	3.989	57	0	5905
CWE534-debug-char-18	0.0	0	0	0	0
CWE534-debug-wchar-01	0.003	3.846	61	0	5816
CWE534-debug-wchar-02	0.005	5.726	204	0	29470
CWE534-debug-wchar-03	0.005	5.662	204	0	29470
CWE534-debug-wchar-04	0.019	5.73	204	0	29470
CWE534-debug-wchar-05	0.004	5.806	204	0	29470
CWE534-debug-wchar-06	0.005	5.711	204	0	29470
CWE534-debug-wchar-07	0.006	5.666	204	0	29470
CWE534-debug-wchar-08	0.008	5.619	204	0	31084
CWE534-debug-wchar-09	0.007	5.674	204	0	29470
CWE534-debug-wchar-10	0.003	5.732	202	0	29276
CWE534-debug-wchar-11	0.005	5.718	204	0	31084
CWE534-debug-wchar-12	0.003	6.337	211	0	24400
CWE534-debug-wchar-13	0.004	5.903	203	0	29404
CWE534-debug-wchar-14	0.005	5.548	204	0	29470
CWE534-debug-wchar-15	0.003	5.779	199	0	29323
CWE534-debug-wchar-16	0.001	3.847	66	0	6608
CWE534-debug-wchar-17	0.002	4.027	78	0	8752
CWE534-debug-wchar-18	0.0	0	0	0	0

Table 8.2.: Measured values of CWE-534

8.3. Results and Constraints

8.4. Correctness validation

8.5. Correctness validation

all the validations

Testcase	RT	TT	No; of Paths	IP	Total no; of nodes
CWE535-shell-char-01	0.001	3.946	67	0	5239
CWE535-shell-char-02	0.014	7.316	288	0	33700
CWE535-shell-char-03	0.013	7.379	288	0	33700
CWE535-shell-char-04	0.015	7.472	288	0	33700
CWE535-shell-char-05	0.025	7.336	288	0	33700
CWE535-shell-char-06	0.014	7.41	288	0	33700
CWE535-shell-char-07	0.017	7.257	288	0	33700
CWE535-shell-char-08	0.014	7.392	288	0	36070
CWE535-shell-char-09	0.018	7.367	288	0	33700
CWE535-shell-char-10	0.005	4.797	112	0	12794
CWE535-shell-char-11	0.019	7.25	288	0	36070
CWE535-shell-char-12	0.007	6.509	236	0	22707
CWE535-shell-char-13	0.012	7.285	288	0	33700
CWE535-shell-char-14	0.018	7.547	288	0	33700
CWE535-shell-char-15	0.015	6.921	254	0	29867
CWE535-shell-char-16	0.004	4.223	71	0	6007
CWE535-shell-char-17	0.003	4.491	92	0	8910
CWE535-shell-char-18	0.0	0.0	0	0	0
CWE535-shell-wchar-01	0.001	3.645	48	0	4198
CWE535-shell-wchar-02	0.005	5.223	165	0	21568
CWE535-shell-wchar-03	0.006	5.208	165	0	21568
CWE535-shell-wchar-04	0.005	5.28	165	0	21568
CWE535-shell-wchar-05	0.005	5.248	165	0	21568
CWE535-shell-wchar-06	0.008	5.47	165	0	21568
CWE535-shell-wchar-07	0.006	5.216	165	0	21568
CWE535-shell-wchar-08	0.007	5.543	165	0	22876
CWE535-shell-wchar-09	0.005	5.24	165	0	21568
CWE535-shell-wchar-10	0.008	5.036	165	0	21568
CWE535-shell-wchar-11	0.007	5.326	165	0	22876
CWE535-shell-wchar-12	0.004	5.245	170	0	18037
CWE535-shell-wchar-13	0.01	5.218	165	0	21568
CWE535-shell-wchar-14	0.006	5.521	170	0	21568
CWE535-shell-wchar-15	0.005	5.401	165	0	21730
CWE535-shell-wchar-16	0.001	3.914	165	0	4922
CWE535-shell-wchar-17	0.006	5.561	165	0	21568
CWE535-shell-wchar-18	0.0	0	0	0	0

Table 8.3.: Measured values of CWE-535

8.6. Efficiency and Overhead

about the tool Efficiency.

8.7. Program Behaviour

Program behaviour tells us if the newly inserted code patch changes the behaviour of the program or not. Here program behaviour means that the inserted code patches should not be able to influence the already existing program paths. The abbrevation form the table IPa(Influencible paths), IP(Influencible Programs), % Ratio represents the ratio between the total number of programs to the total number of influencible programs which contains atleast one influencible path. Based on the decision made at the time for running the localizer, which means there exits a not-in-place fix also. So after finding the not-in-place fix and inserting the code patch into the program form refactoring then the program paths are then verified in order to check if the inserted code patch would change the program behaviour or not. After verifying for the influencible paths, if it finds any such influencible paths then the refactorings are not at all generated and the decision is left to the programmer to refactor the code or not. This way the change of program behaviour can be avoided by not proposing the refactorings at all.

8.8. Usefulness of the generated Patches

Usefulness of the generated patches.

9. Discussion

9.1. Limitations

limitations of the tool and approach.

9.2. Applications

Where can this be applied.

9.3. Future Work

about the Future work.

10. Summary and Conclusion

Appendix

A. Definitions and Abbrevations

1. IF : Information Flow
2. IFE : Information Flow Exposure

Bibliography

[1] *http://www.forbes.com/sites/moneybuilder/2015/01/13/the-big-data-breaches-of-2014/*.

[2] CWE. *https://cwe.mitre.org/*.

[3] Top 25 vulnerabilities. *http://www.veracode.com/directory/cwe-sans-top-25*.

[4] Top ten vulnerablities. *http://www.veracode.com/directory/owasp-top-10*.

[5] Wikipedia. *https://en.wikipedia.org/wiki/Program_analysis*.

[6] *http://www.cvedetails.com/cwe-definitions/1/orderbyvulnerabilities.html?order=2&trc=668&sha=0427874c* jan 2015.

[7] M. Backes, M. Dürmuth, S. Gerling, M. Pinkal, and C. Sporleder. Acoustic Side-channel Attacks on Printers. In *Proceedings of the 19th USENIX Conference on Security*, USENIX Security'10, pages 20–20, Berkeley, CA, USA, 2010. USENIX Association.

[8] T. Ball and Sriram K. Rajamani. Automatically Validating Temporal Safety Properties of Interfaces. In *Proceedings of the 8th International SPIN Workshop on Model Checking of Software*, SPIN '01, pages 103–122, New York, NY, USA, 2001. Springer-Verlag New York, Inc.

[9] N. Baracaldo and J. Joshi. Beyond Accountability: Using Obligations to Reduce Risk Exposure and Deter Insider Attacks. In *Proceedings of the 18th ACM Symposium on Access Control Models and Technologies*, SACMAT '13.

[10] C. Cadar and K. Sen. Symbolic Execution for Software Testing: Three Decades Later. *Commun. ACM*, 56(2):82–90, February 2013.

[11] M. Das, S. Lerner, and M. Seigle. ESP: Path-sensitive Program Verification in Polynomial Time. *SIGPLAN Not.*, 37(5):57–68, May 2002.

[12] V. D'silva, D. Kroening, and G. Weissenbacher. A Survey of Automated Techniques for Formal Software Verification. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 27(7):1165–1178, July 2008.

[13] Z. Gu, Earl T. Barr, David J. Hamilton, and Z. Su. Has the Bug Really Been Fixed? In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pages 55–64, New York, NY, USA, 2010. ACM.

[14] T. Halevi and N. Saxena. A Closer Look at Keyboard Acoustic Emanations: Random Passwords, Typing Styles and Decoding Techniques. In *Proceedings of the 7th ACM Symposium on Information, Computer and Communications Security*, ASIACCS '12, pages 89–90, New York, NY, USA, 2012. ACM.

[15] M. Hanspach and J. Keller. A Taxonomy for Attack Patterns on Information Flows in Component-Based Operating Systems. *CoRR*, abs/1403.1165, 2014.

[16] P. Herzog. Open Source Security Testing Methodology Manual (OSSTMM). *http://www.isecom.org/research/osstmm.html*.

[17] D. King, B. Hicks, M. Hicks, and T. Jaeger. Implicit Flows: Can'T Live with 'Em, Can'T Live Without 'Em. In *Proceedings of the 4th International Conference on Information Systems Security*, ICISS '08, pages 56–70, Berlin, Heidelberg, 2008. Springer-Verlag.

[18] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. seL4: Formal Verification of an OS Kernel. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, pages 207–220, New York, NY, USA, 2009. ACM.

[19] D.S. Kushwaha and A. K. Misra. Software Test Effort Estimation. *SIGSOFT Softw. Eng. Notes*, 33(3):6:1–6:5, May 2008.

[20] Butler W. Lampson. A Note on the Confinement Problem. *Commun. ACM*, 16(10):613–615, October 1973.

[21] Andrew C. Myers. JFlow: Practical Mostly-static Information Flow Control. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '99, pages 228–241, New York, NY, USA, 1999. ACM.

[22] J. Robinson, W. S. Harrison, N. Hanebutte, P. Oman, and J. Alves-Foss. Implementing Middleware for Content Filtering and Information Flow Control. In *Proceedings of the 2007 ACM Workshop on Computer Security Architecture*, CSAW '07, pages 47–53, New York, NY, USA, 2007. ACM.

[23] A. Sabelfeld and A. Russo. From Dynamic to Static and Back: Riding the Roller Coaster of Information-flow Control Research. In *Proceedings of the 7th International Andrei Ershov Memorial Conference on Perspectives of Systems Informatics*, PSI'09, pages 352–365, Berlin, Heidelberg, 2010. Springer-Verlag.

[24] N. Tobias, H. Benedikt, and G. Orna. *https://books.google.de/books?id=ckaVltDIatwC&pg=PA340&lpg=PA Nfex3s27N22Zv-98qY&hl=en&sa=X&ved=0CCYQ6AEwAWoVChMIjqyisYnYxwIVRBcsCh3PGAE1#v=o*

[25] W. van Eck. Electromagnetic Radiation from Video Display Units: An Eavesdropping Risk? *Comput. Secur.*, 4(4):269–286, December 1985.

[26] D. Volpano, C. Irvine, and G. Smith. A Sound Type System for Secure Flow Analysis. *J. Comput. Secur.*, 4(2-3):167–187, January 1996.

[27] M. Ward. *Proving Program Refinements and Transformations*. PhD thesis, 1986. AAID-90357.

[28] Y. Yu and T. Chiueh. Display-only File Server: A Solution Against Information Theft Due to Insider Attack. In *Proceedings of the 4th ACM Workshop on Digital Rights Management*, DRM '04, pages 31–39, New York, NY, USA, 2004. ACM.