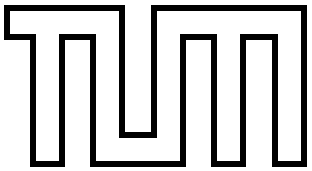


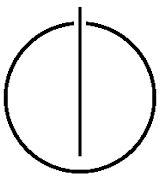
FAKULTÄT FÜR INFORMATIK

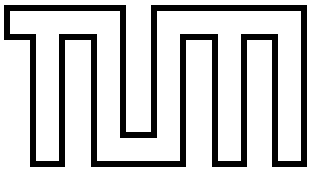
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Automated Detection, Localization and Removal of Information Exposure Errors

Kommanapalli Vasantha





FAKULTÄT FÜR INFORMATIK

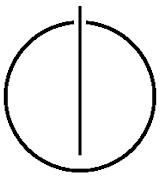
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

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 21. 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 developers cannot waste their productive 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, tracking and fixing the bug 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 minimized.

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. The algorithm should be able 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 [7], there are 2082 vulnerabilities related to information exposure. Among many Information Flow(IF) weaknesses, Information exposure is one of the serious issues. This kind of weaknesses can exist in the code, without breaking anything, yet providing some confidential information to the attacker who wishes to exploit the system. Information exposure bugs could be introduced at different stages of software development. During design, architecture or coding phase can expose critical information and 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 [20] and [22].

In 2013, sensitive data exposure ranked 6th in the AOWASP top ten list [5] and in 2011, information exposure through an error message ranked 17th among CWE/SANS top 25 [4]. 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, a lot of debit and credit card information concerning 350,000 customers have been hacked. It is believed that the breach was made possible just by installing malicious software 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 accessed to the system without leaving any trace of hacking. Sensitive information can be leaked in many ways [2]. Some of them are:

- through the environmental variables which contain the sensitive information about any remote server.
- through a log file that was used while debugging the application and late on, the access to that file was not restricted.
- 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 control flow graphs (CFG) which, can give us an idea about the execution paths and provides high path coverage. There are many tools like ESP [12], SLAM [9] that have been built based on static analysis approach. There are also many dynamic analysis approaches

but these do not 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 as an input, how much ever complex they are [11]. These IF errors can be addressed using dynamic analysis [26], static [29], [24] and hybrid anaylsis(combination of both static and dynamic analysis).

The goal of this thesis is to develop an algorithm which localizes the faults and repairs the information exposure in 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 a 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 because, the cause for the bug will not only be at the place where the bug was detected but also at a location 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”. In the process of generating the program repairs, the following information is necessary: The process for generating the program repair involves:

- code patch patterns.
- SMT solver.
- searching the possible quick fix locations that will not affect the program behaviour after inserting the patch at the ”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 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 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. In general,there are 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 finding the source code location accurately 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 mentioned earlier, 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 the type 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 from C/C++ Juilet test suite [3]. 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. Therefore, in order to restrict the confidential data to leave the sink we need to refactor the buggy program by introducing small source code patches.

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 been 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 listing 1.1 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 are 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 behavioral aspects of other programs can be changed . In order to retain the program behavior, a pre-fix location has to be 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 called 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 and compilable.

1.3. Security

Degree of protection or the resistance towards any harm from 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 [17]:

"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 is focussed on 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 firstly 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 verifies 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 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 to the users at all times. 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. During the process of designing, load balancing and fail-over solutions should also be taken into consideration. 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 identity 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 of, in many ways like providing username and password for the agent. Cracking the password may be easy for the hacker therefore stronger passwords should be made.

  For secure information flow, confidentiality and integrity are ensured.

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 refers to 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 [14].
- 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 expected behavior [14].

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 5
- An algorithm for generation of ”in-place” and ”not-in-place” bug fixes 1, 2
- Source files differential views which shows the semi-automated patch insertion 6.3
- Verifying the behavior of the patched program automatically 6.4

2. Technical and Scientific Fundamentals

2.1. Sources of information flows with examples

Principal sources of information flows are of two types: [20] [27]

- Implicit flow
- Explicit flow

Implicit Flow: Implicit flow is the one where secret data implicitly flows due to some control flow that is affected by secret values
Explicit Flow: Explicit flow is the 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 [16] 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 [10], [16], [31].

- $p_i \Longleftrightarrow p_j$: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 [23].

- $p_i \Longleftrightarrow H$: This pattern exploits the covert channels 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 [8], [15], [28].

- $p_i \Longleftrightarrow K \Longleftrightarrow A$: 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 [23].

- $p_i \Longleftrightarrow K \Longleftrightarrow H \Longleftrightarrow p_j$: 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 through H into the physical environment and then receiving at some other interface of H [21, 25].

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. Its major focus is on program correctness which ensures that the program 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 [6]. In general the program analysis can be done in three different 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. [13] In general the static technique in order to enforce secure information flow is by the usage of type systems. In many 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 [27].

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 interaction 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 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 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. The structure can help us to transform a program. 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 can filter, rewrite and 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 from the original code [30]. 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. This program transformation requires effective processing of programs written in any of the existing programming language.

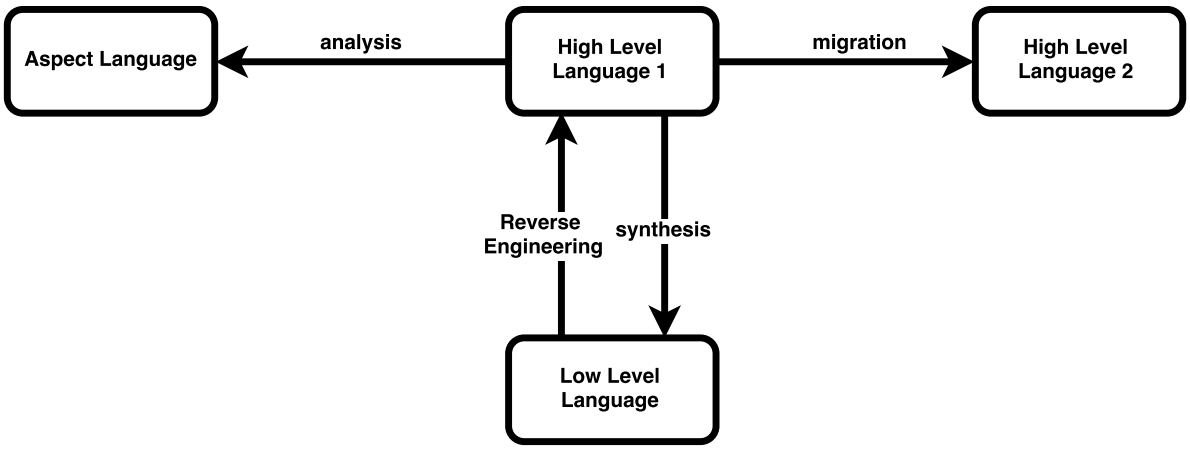


Figure 2.1.: Scenarios of translation

3. Related Work

Part II.

Design and Development

4. System Overview

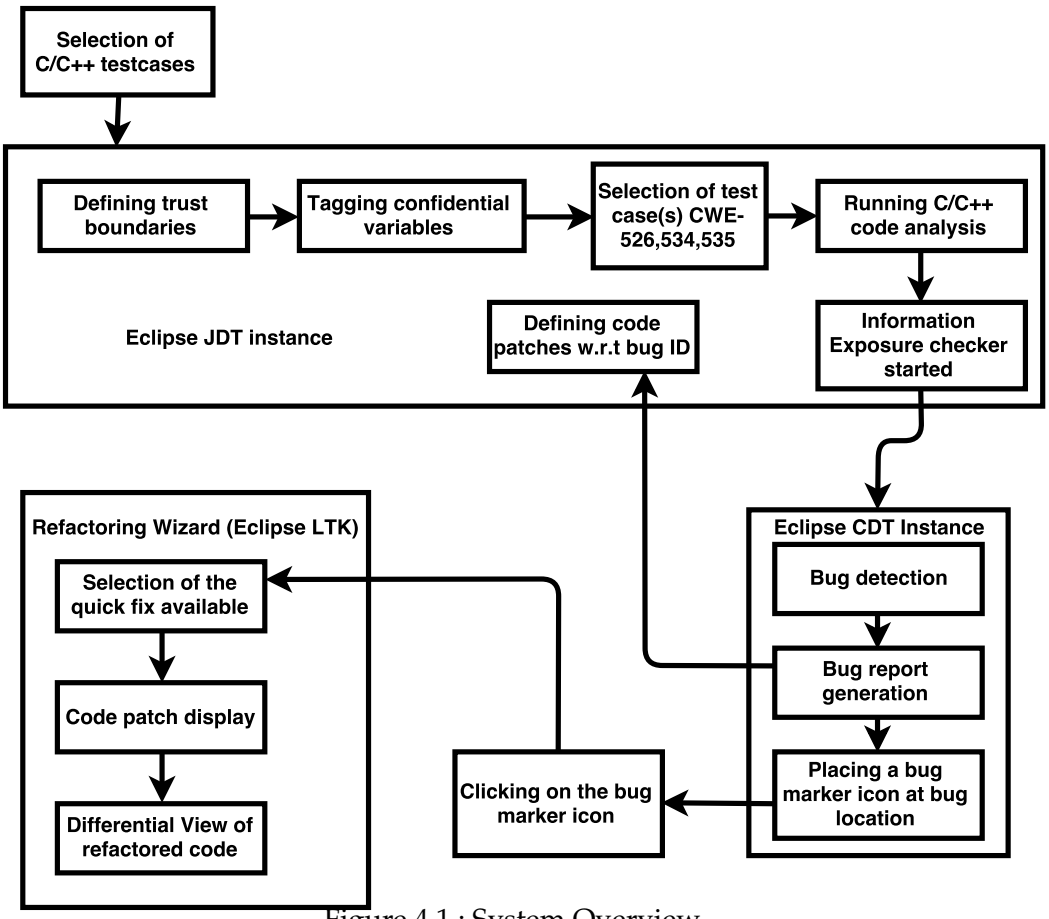


Figure 4.1.: System Overview

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. Step by step procedure involved in the system's workflow:

- all the C/C++ test programs have to be selected which are to be imported in the workspace.
- all the confidential variables in the test programs have to be tainted and the trust boundaries need to be defined.
- 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.
- a refactoring wizard appears after clicking on the bug marker that is placed after running the information exposure checker.

Basically the IE checker is run as an 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, CWE-534, CWE-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, the file and the path which can lead to the location of the bug on clicking on the warning message as represented in figure 6.2 marked with number 3. After reaching the bug location in the file, upon clicking on the bug marker icon a refactoring wizard is launched containing the quick fix options as shown in figure 6.4. After selecting any one option, a window with the selected code patch is displayed and upon clicking next,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

Alogrithm 1 explains how the decision is made for running the localizer algortihm. The input for this algorithm is the node containing the bug location, filename and also the source code file name. Output is a set of refactorings. For the current test programs we find a set of two refactorings only. First we initialise the nodes set $node_{set}$, refactorings set R_{set} to empty set. Then, after the IE checker reports a bug a check is done in order to find if the file in which the bug was detected is a source file or not. If both the file name and the source files are not the same then the localizer is started else refactorings for the detected buggy nodes is done. In the process of refactoring the nodes, the buggy nodes are collected and then manual code patches for the coressponding bugs are made. Then the manually made code patches are then translated into nodes and with the help of eclipse LTK the code patches are then inserted into the buggy program.

Algorithm 2 shows how a localizer works. Input for this algorithm is a set of satisfiable paths Sat_{paths} and output is a set of refactorings R_{set}. N_{set} is the set of nodes. $pathloc_{list}$ is a set of paths list which contains the nodes which inturn contains the file name and the line number of every node in a particular path. Initially $pathloc_{list}$, R_{set}, N_{set} are all set to empty. Below are the steps followed in the localizer algorithm:

- First the localizer checks for the satisfiable paths.
- For every node in the path it then checks if the node has a translation unit using the the function *hasTranslationalUnit()*.

- If the node has a translational unit then the filename and the linenumber of that particular node is stored else the value of the filename is set to null and the line number to be -1.
- Particular node values are then added to the $path_{location}$ set.
- Once all the nodes in a particular path has been traversed and its filename and linenumber are stored then the particular path details are added to the $pathloc_{list}$ set. This is done until all the satisfiable paths exist.
- Then the frequency of the nodes occuring in all the satisfiable paths is counted.
- Once all the nodes frequency is counted, then all the nodes with their respective frequency is maintained in a bugmap data structure.
- Now all the not buggy paths are removed form the list of all satisfiable paths in a way that all the nodes which appeared in both the buggy and not buggy paths are also removed. This makes sure that the nodes that appear in both buggy and not buggy path are not the cause for the bug. In this way one can filter out the not suspicious nodes.
- Filtering helps us to save time when we start back traversing the paths in order to find the ”not-in-place” quick fix locations.
- Once all the not buggy nodes are removed from the bugmap then we check for the node which contains the maximum frequency. The node which contains the maximum frequency is supposed to be the most suspicious node. If we have nodes with the same frequency then all the nodes are collected and added to the set of $suspicious_{nodes}$ and also the path of that coressponding node in $node_{map}$ hashmap.
- We then check the size of the $node_{map}$, if the $node_{map}$ size is 1 then the refactoring is created for that particular node and added to the refactorings list.
- If the size of $node_{map}$ is greater than one then we check if the nodes in the hashmap belong to different program paths.
- If the nodes belong to different program paths then the refactoring for every suspicious node is created and added to the set of refactorings node.
- If they doesnot belong to different nodes then the earliest node that appears in the program execution path is considered to be the earliest quick fix location and a refactoring for only that node is created and added to the refactorings list.

  After all the refactorings are created then the generated code patches are then inserted into the buggy program with the help of eclipse LTK API, in order to remove the bugs. This localizer is very efficient and faster in localising the bugs. Detecting the ”not -in-place” fix can also ne done by backward propagation of all the nodes in the buggy path butit takes more time then the localizer because in localizer we donot parse all the nodes. We filter the not suspicious nodes and parse only the suspicious nodes. Therefore we happen to parse very less number of nodes using the localizer

then backward propagation mechanism. Figure 5.1 depicts the activity diagram of the localizer and how the decisions are made. In the eclipse LTK API that has been used with show two types of options at the time of refactorting i.e, ”Earliest Fix” and ”Latest Fix” which means the ”not-in-place” fix and the ”in-place” fix respectively. In the test program CWE-526 we use the localizer because the bug was detected initially at one of the library files. Therefore one cannot insert or change any code in the library files because there can be many other function which make use of those library files. So in order to avoid this we made use of the localizer to find a location where the bug can be fixed in the source file itself but not in the library files.

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

Input: Set of satisfiable paths Sat_{paths} :=$\{spath_i | \ 0 \leq \text{i} \leq \text{n}, \forall \ \text{n} \geq 0 \}$;
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`
$pathloc_{list} := \{path_k | \ 0 \leq \text{k} \leq \text{n}, \forall \ \text{n} \geq 0 \}$; `//Set of paths list with filename and line number of every node in the path`
$path_{location}$:=$\emptyset$; `//list of nodes in the path with filename and linenumber`
R_{set} :=$\emptyset$; `// initialising refactorings set`
N_{set} :=$\emptyset$; `// 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
  $path_{location}$:=$\emptyset$; `//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 Localizer algorithm and generating code patches

```
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
```

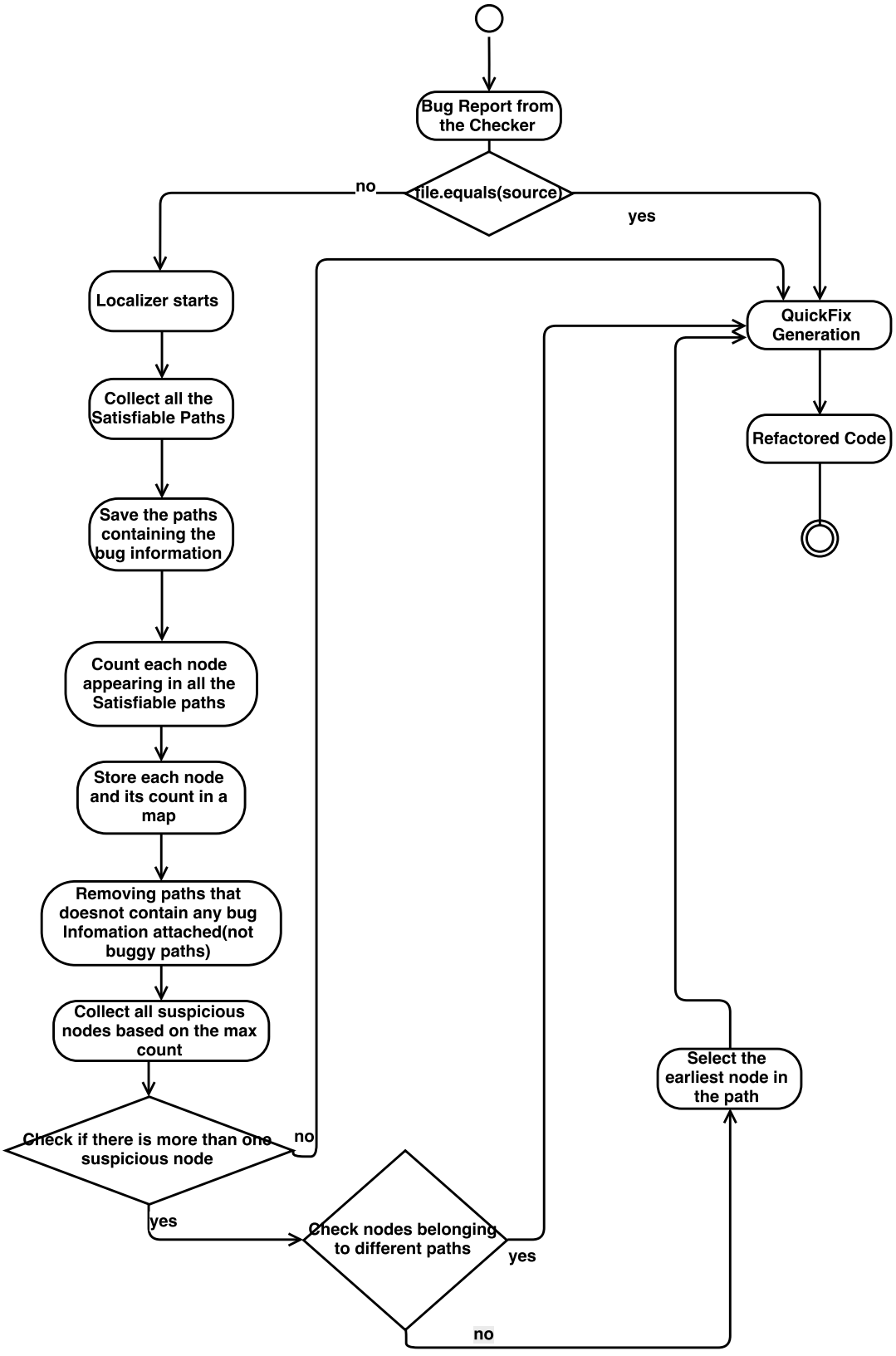


Figure 5.1.: Localizer activity diagram

6. Approach

General Steps involved in detecting, localizing and fixing bugs are:

- Identifying the error
- Finding the error
- Analyzing the error
- Proving the analysis
- Covering all the lateral damages
- Fixing the error
- Validating the solution

Identifying the error: This is a general step in which we identify the error. There are times when some errors can waste a lot of our developing time. Sometimes the errors reported by the users are also hard to interprete and some of the information could also be misleading. Therefore one can do the following steps:

- See the error if it is understandable or ask the user to send some screenshots.
- Try to reproduce the error. One should not claim that there is no error without reproducing it.
- Understanding the expected behaviour. In some of the complex applications ,it is difficult to understand the expected behaviour of an error. So, it is better to ask the product owner or check the documentation in order to find the expected behaviour of the error.

Finding the error: Once we have indentified the error, we then have to find where the error is located. One has to go through the code in order to find the error location. Imagine if there thousand lines of code it is very diffcult to go through the code. Some of the techniques that could be helpful are:

- Logging
- Debugging
- Removing code

Analyzing the error: This step is a bottom-up approach in which we try to analyze the code from the place we found the error and try to figure out the cause for the error.

Proving the Analysis: After analyzing the error, there might be some other errors influenced by the original error. so one has to come up with some automated test cases.

Covering lateral damage: At this step the developer would be ready with the code patch that he wanted to insert into the code in order to remove the error but he should be careful enough that the original behaviour of the program is not at all disturbed.

Fixing the error: Finally after all the verfication for program behavior to be preserved is done, we then insert the code patch in order to fix the bug.

Validating the solution: Once again we run all the test cases in order to make sure that the error is removed.

Therefore all these steps can be handled manually for softwares which contain very less number of lines of code. Imagine a large project where there are thousands of lines of code. It is really a hetic task to identify and fix the bugs. In order to make this task simpler we need to automate the process. The aim of this thesis is to provide a tool for this process of fixing the errors automatically.

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 the C/C++ library 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 statically 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 trustboundary 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/CWE-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 the 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 places 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 options 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.

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 testcases annotated with **1**

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”

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.

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

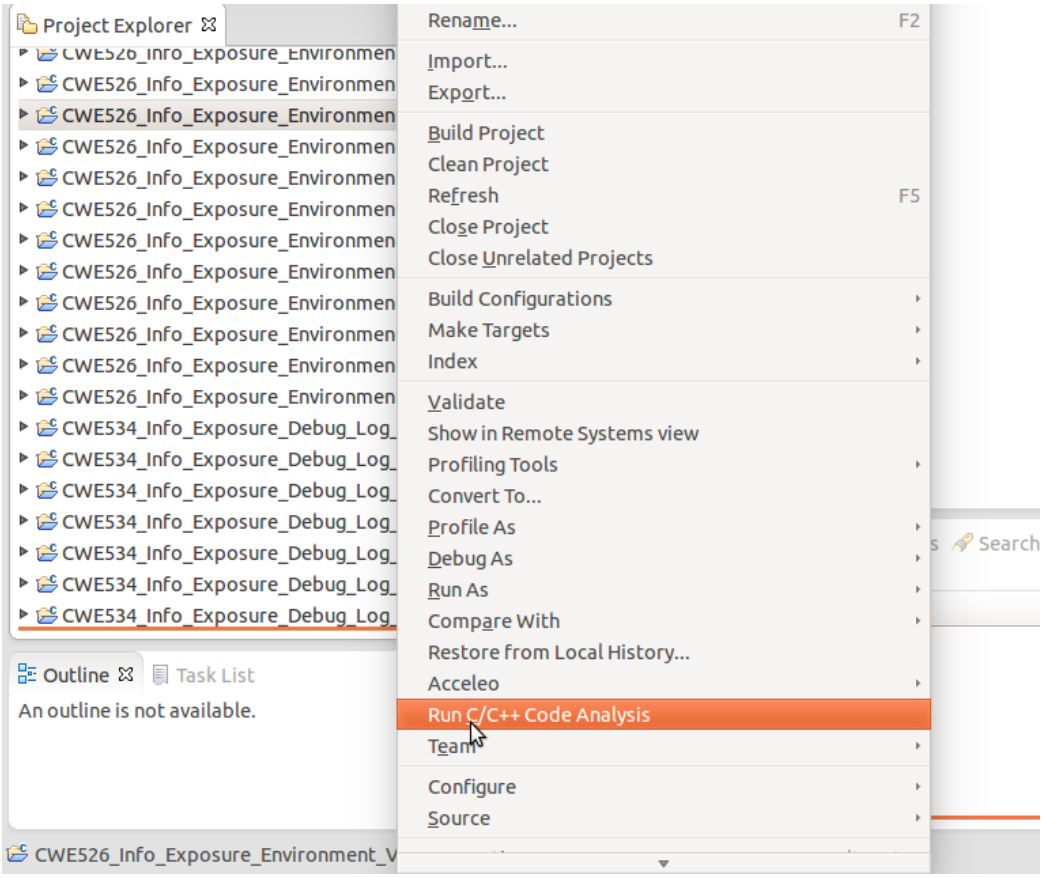


Figure 6.1.: C/C++ testcase selection and running analysis

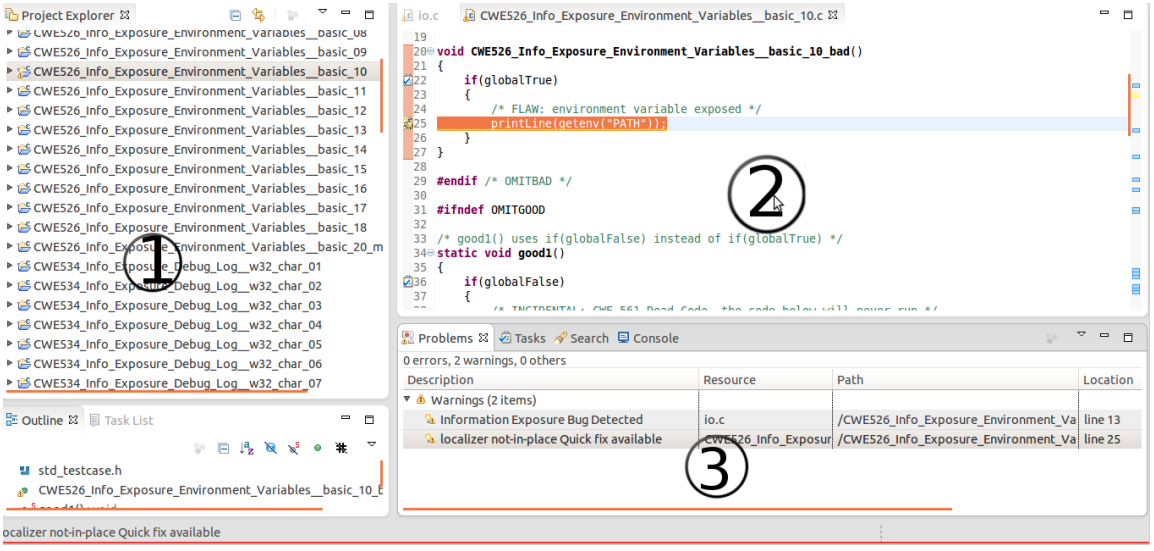


Figure 6.2.: Marker at the bug location

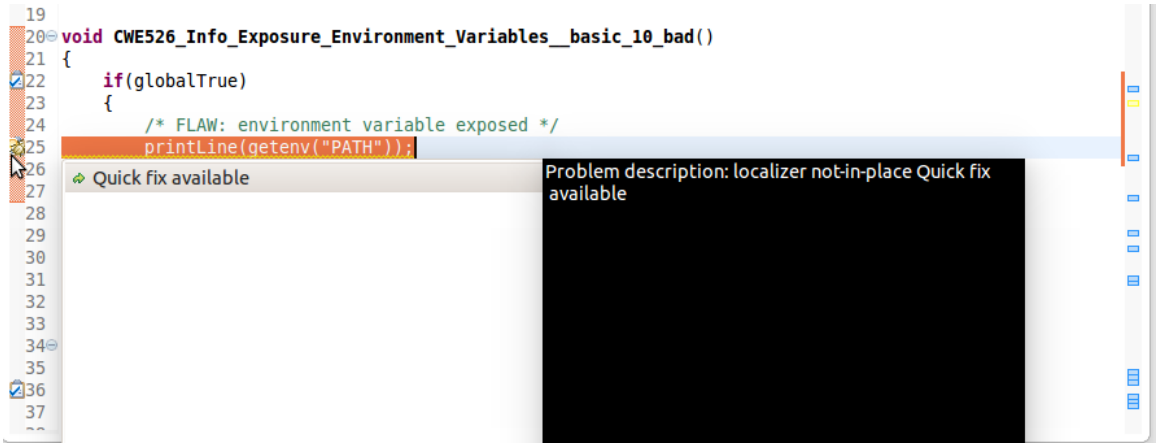


Figure 6.3.: Click on the marker icon

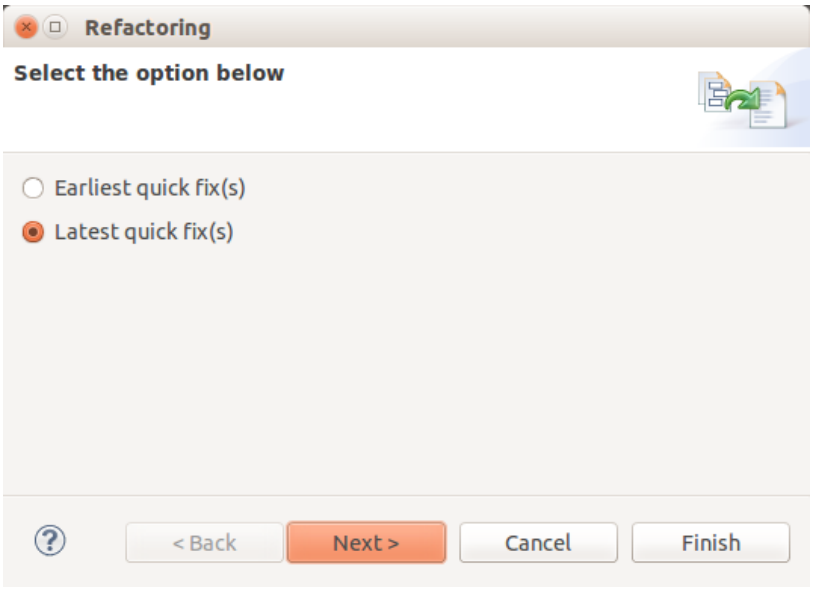


Figure 6.4.: Refactoring Wizard with two options

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.

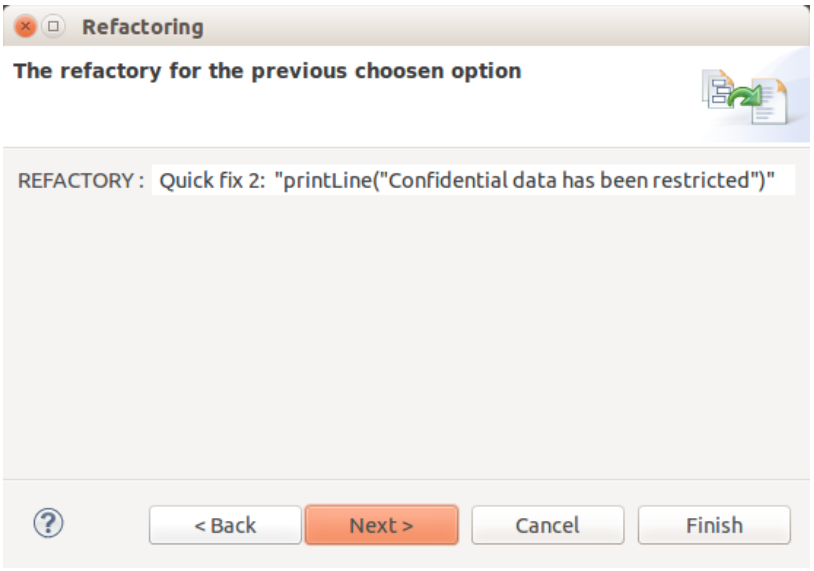


Figure 6.5.: Patch according to the previous selection

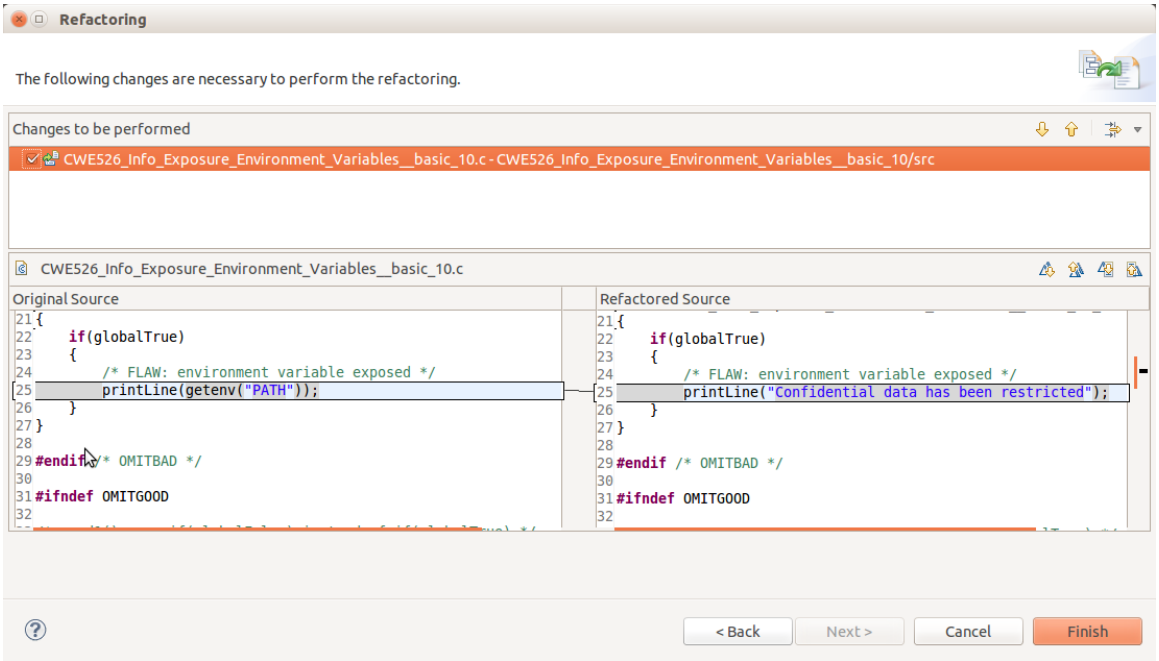


Figure 6.6.: Differential view of the patched and unpatched programs

7. Implementation

Developed information exposure bugs fixing tool was integrated 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, performing 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 to fix the bug at a place in the source code file from where the confidential data leaves. This is the ”not-in-place” fix location. So after running the localizer algorithm we get some suspicious nodes. From these suspicious nodes, we then again propagate through all the nodes in order to verify if some confidential information flows through it. This can be done by checking it the variables are tagged with the tag ”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. This refactoring LTK API is found in org.eclipse.ltk.core.refactoring and org.eclipse.ltk.ui.refactoring plug-ins.

In general the life cycle of a refactoring is as follows:

- First the user starts refactoring.
- Verfication is done to see if the refactoring is applicable at that context as desired by the user using the function *checkInitialCondtions()*
- If necessary, the user is asked if he has additional information.
- Once all the information is available then a final check is performed using *checkFinalConditions()* and all the changes in their source text are calculated using *createChange()*.
- After this the differential view with both the source code and the refactored code appears and once the user clicks on OK then the refactoring is done.

org.eclipse.ltk.core.refactoring.Refactoring subclass should always be created which is the main class for the refactorting. For the refactoring participants to be supported *ProcessorBasedRefactoring* should be extended. And also a subclass of org.eclipse.ltk.ui.refactoring.RefactoringWiz is necessary for the UI side which is used for handling every individual pages of the refactoring wizard through which the user will be able to scroll with the help of ”next” and ”back” and also ”finish” in order to invoke the final operation of the refactoring process.

Steps followed for refactoring:

- In the first step when all the information required for triggering the refactoring is collected. For example consider that the plug-in has a text selection, therefore the information about the selected text and also its position in the document is saved in an info object. So this info object is then evaluated with the help of RenamePropertyRefactoring.

Listing 7.1: Class RenameProperty

```
RefactoringProcessor processor = new RenamePropertyProcessor( info );
RenamePropertyRefactoring ref = new RenamePropertyRefactoring( processor );
RenamePropertyWizard wizard = new RenamePropertyWizard( ref, info );
RefactoringWizardOpenOperation op = new RefactoringWizardOpenOperation(wizard );
try {
  String titleForFailedChecks = ""; //$NON-NLS-1$
  op.run(getShell(), titleForFailedChecks );
} catch( InterruptedException irex ) {
  // operation was cancelled
}
```

- Now LTK controls the next process. Initially the *checkInitialCondtions()* method is called on the refactoring object. This checks if the basic conditions are enough for the refactoring.

Listing 7.2: Class RenamePropertyDelegate

```

RefactoringStatus checkInitialConditions() {
  RefactoringStatus result = new RefactoringStatus();
  IFile sourceFile = info.getSourceFile();
  if( sourceFile == null || !sourceFile.exists() ) {
    result.addFatalError(
        CoreTexts.renamePropertyDelegate_noSourceFile );
  } else if( info.getSourceFile().isReadOnly() ) {
    result.addFatalError( CoreTexts.renamePropertyDelegate_roFile );
  } else if( isEmpty( info.getOldName() ) || !isPropertyKey(
      info.getSourceFile(), info.getOldName() ) ) {
    result.addFatalError(
        CoreTexts.renamePropertyDelegate_noPropertyKey );
  }
  return result;
}
```

- Once the result from the above step is satisfied then the wizard is displayed and asks the user for additional information if necessary. This UI is implemented as a

subclass of *UserInputWizardPage* Whichever the data has been entered at this point is then made available to the *RenamePropertyRefactoring* via the Info object.

- *checkFinalConditions()* method is called before the last page is displayed via the refactoring wizard. All the changes are then handled by the createChange() method.

Listing 7.3: Class RenamePropertyDelegate(core)

```
private Change createRenameChange() {
  // create a change object for the file that contains the property
  // which the user has selected to rename
  IFile file = info.getSourceFile();
  TextFileChange result = new TextFileChange( file.getName(), file
      );
  // a file change contains a tree of edits, first add the root of
      them
  MultiTextEdit fileChangeRootEdit = new MultiTextEdit();
  result.setEdit( fileChangeRootEdit );

  // edit object for the text replacement in the file, this is the
      only child
  ReplaceEdit edit = new ReplaceEdit( info.getOffset(),
  info.getOldName().length(),
  info.getNewName() );
  fileChangeRootEdit.addChild( edit );
  return result;
}
```

- Once the user reaches the final page then if the user doesnot click ”finish” button immediately then the differential view will be displayed with the requested changes.

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, CWE-534, CWE-535. 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 x 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

The refactoring tool was run on 3 kinds of test programs with many flow variants and generated two different types of code patches in order to automatically fix the bugs that were detected by the IE checker. Previously developed IE checker and classification mechanism in order to differentiate different types of bugs. Measured all the times required for running the checker, for detecting the bug, for refactoring and also the overall overhead that occured because of refactoring. Times were measured in milliseconds and then again converted into seconds[s]. Program behaviour was preserved for all the program inputs which did not trigger the bug. Some of the research questions that have been addressed are:

- Overall computational overhead of the tool.
- Usefulness of the generated code patches.
- About the program behavior.
- Correctness

CWE- 526, CWE-534, CWE-535 were used as the test programs 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 patch generation[], execution time for the test programs which belong to the test case. Also the number of true positives, false positives, true negatives and false negatives were are also calculated and reperesented in the table 8.1. Following are the abbrevations in the table: True postives (TP), Lines of Code (LOC), and these lines are excluding the code comments, False Positives (FP), False Negatives (FN), Total True Positives (TTP). Developed tool was able to find 85 TPs out of 90 TPs which are the available test programs. Tool was able to detect all the IE bugs. But the programs containing the goto statements were not able to be tested since the Codan API doesnot support building the Control Flow Graphs (CFGs) for the C source code containing goto statements. For example, goto A; . Among all the test programs available in Juilet test suite CWE-526 has 1 program which contians goto statement, 2 in CWE-534 and 2 in CWE-535. As a whole 5 programs out of 90 test programs were able to be analyzed by building the CFGs. Therefore ,the overall test coverage reached to 94.44%. It can be concluded that the tool would reach a test coverage of 100% if this limitation was removed.

Test Programs	**TP**	**LOC**	**FP**	**FN**	**TTP**	**TTP**
CWE-526	18	960	0	0	17	18
CWE-534	36	5783	0	0	34	36
CWE-535	36	5273	0	0	34	36
Total	90	12016	0	0	85	90

Table 8.1.: Measured values of CWE-526

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.

We can see three different types of messages:

1. Errors
2. Warnings
3. Infos

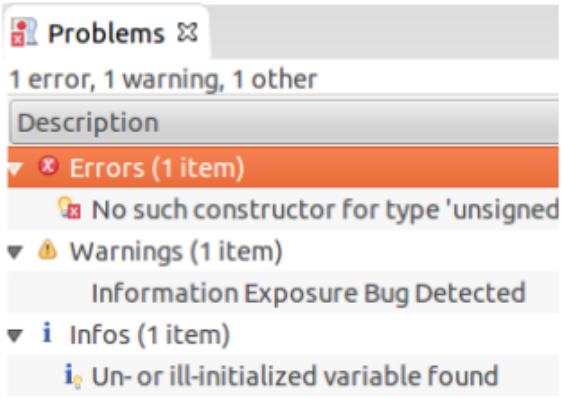


Figure 8.1.: Types of possible messages

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.

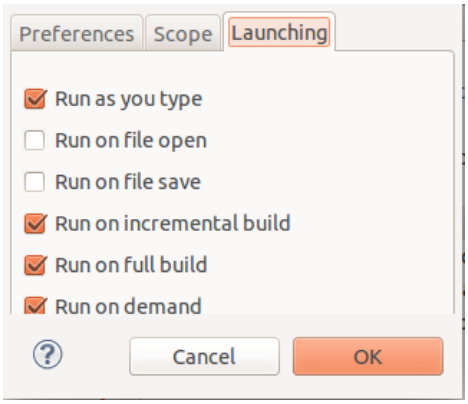


Figure 8.2.: Different running modes

8.3. Results and Constraints

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

Table 8.2.: Measured values of CWE-526

Table8.2 depicts all the measured values for the test case CWE-526,Table8.3 depicts all the measured values for the test case CWE-534 and Table8.4 depicts all the measured values for the test case CWE-535.The column abbrevations are Path Generation Time (PGT), Total Time (TT), Influencible Paths (IP), Bug Detection Time (BDT).

8.4. Efficiency and Overhead

Figure 8.3 shows the results when 18 C/C++ programs of the test case CWE-526 from juilet test suite were run. The black bars above the yellow bars indicate the overhead introduced due to the patch generation and the yellow bars indicate the time taken for the bug detection. The total overhead introduced for running these 18 test programs is 0.35% which was obtained by comparing the patch generation time (0.214[s]) with the bug detection time (60.854[s]) 61.068[s]-0.214[s] which can be observed from the table 8.5.

Figure 8.4 shows the results when 34 C/C++ programs of the test case CWE-534 from juilet test suite were run out of 36 test cases because the SMT solver used could build the CFG for the test cases containing goto statements. The black bars above the yellow bars indicate the overhead introduced due to the patch generation and the yellow bars indicate the time taken for the bug detection. The total overhead introduced for running these 34 test programs is 0.13% which was obtained by comparing the patch generation

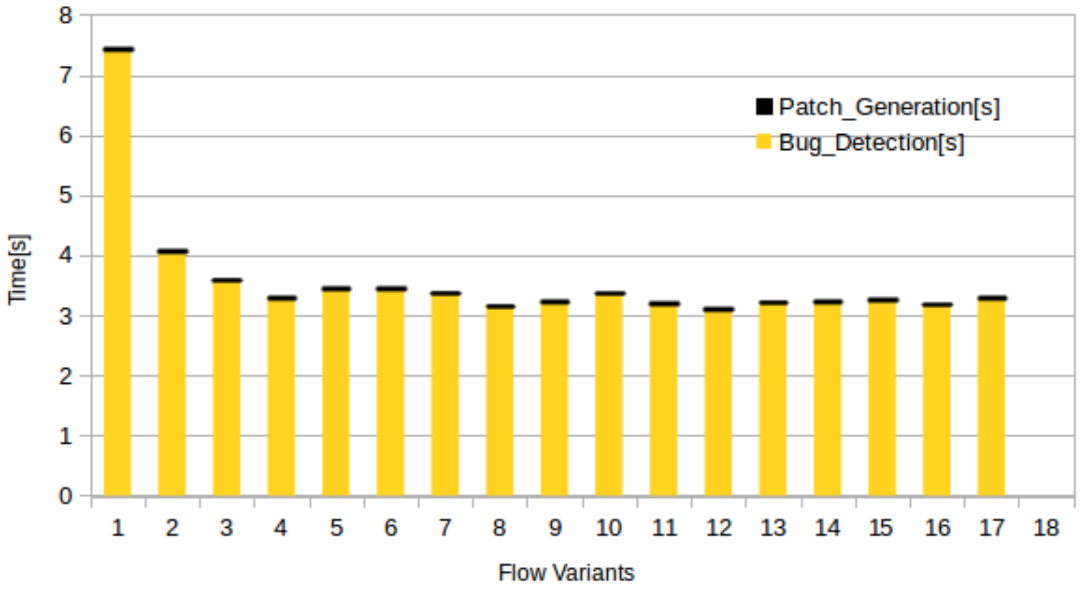


Figure 8.3.: Comparision between the patch generation and bug detection times of CWE-526

time (0.369[s]) with the bug detection time (264.384[s]) 264.753[s]-0.369[s] which can be observed from the table 8.5

Figure 8.5 shows the results when 34 C/C++ programs of the test case CWE-535 from juilet test suite were run out of 36 test cases because the SMT solver used could build the CFG for the test cases containing goto statements. The black bars above the yellow bars indicate the overhead introduced due to the patch generation and the yellow bars indicate the time taken for the bug detection. The total overhead introduced for running these 34 test programs is 0.15% which was obtained by comparing the patch generation time (0.309[s]) with the bug detection time (198.884[s]) 199.193[s]-0.309[s] which can be observed from the table 8.5

8.5. 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 8.6 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

Test Program	RT	TT	No; of Paths	IP	Total no; of nodes	BDT[ms]
CWE534-debug-char-01	0.004	5.644	87	0	7560	5.64
CWE534-debug-char-02	0.043	9.688	372	0	48645	9.645
CWE534-debug-char-03	0.01	8.649	372	0	48645	8.639
CWE534-debug-char-04	0.013	8.62	370	0	48552	8.607
CWE534-debug-char-05	0.016	8.487	372	0	48645	8.471
CWE534-debug-char-06	0.015	8.264	371	0	48609	8.249
CWE534-debug-char-07	0.014	8.352	372	0	48645	8.338
CWE534-debug-char-08	0.018	6.421	198	0	30141	6.403
CWE534-debug-char-09	0.051	10.177	372	0	48645	10.126
CWE534-debug-char-10	0.022	7.894	278	0	36495	7.872
CWE534-debug-char-11	0.017	8.891	372	0	51699	8.874
CWE534-debug-char-12	0.007	7.363	308	0	32341	7.356
CWE534-debug-char-13	0.015	48.654	372	0	48645	48.63
CWE534-debug-char-14	0.02	8.517	372	0	48645	8.497
CWE534-debug-char-15	0.011	8.555	372	0	49014	8.544
CWE534-debug-char-16	0.003	4.266	92	0	8551	4.263
CWE534-debug-char-17	0.002	3.989	57	0	5905	3.987
CWE534-debug-char-18	0	0	0	0	0	0
CWE534-debug-wchar-01	0.003	3.846	61	0	5816	3.843
CWE534-debug-wchar-02	0.005	5.726	204	0	29470	5.721
CWE534-debug-wchar-03	0.005	5.662	204	0	29470	5.657
CWE534-debug-wchar-04	0.019	5.73	204	0	29470	5.711
CWE534-debug-wchar-05	0.004	5.806	204	0	29470	5.802
CWE534-debug-wchar-06	0.005	5.711	204	0	29470	5.706
CWE534-debug-wchar-07	0.006	5.666	204	0	29470	5.66
CWE534-debug-wchar-08	0.008	5.619	204	0	31084	5.611
CWE534-debug-wchar-09	0.007	5.674	204	0	29470	5.667
CWE534-debug-wchar-10	0.003	5.732	202	0	29276	5.729
CWE534-debug-wchar-11	0.005	5.718	204	0	31084	5.713
CWE534-debug-wchar-12	0.003	6.337	211	0	24400	6.334
CWE534-debug-wchar-13	0.004	5.903	203	0	29404	5.899
CWE534-debug-wchar-14	0.005	5.548	204	0	29470	5.543
CWE534-debug-wchar-15	0.003	5.779	199	0	29323	5.776
CWE534-debug-wchar-16	0.001	3.847	66	0	6608	3.846
CWE534-debug-wchar-17	0.002	4.027	78	0	8752	4.025
CWE534-debug-wchar-18	0	0	0	0	0	0

Table 8.3.: Measured values of CWE-534

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.

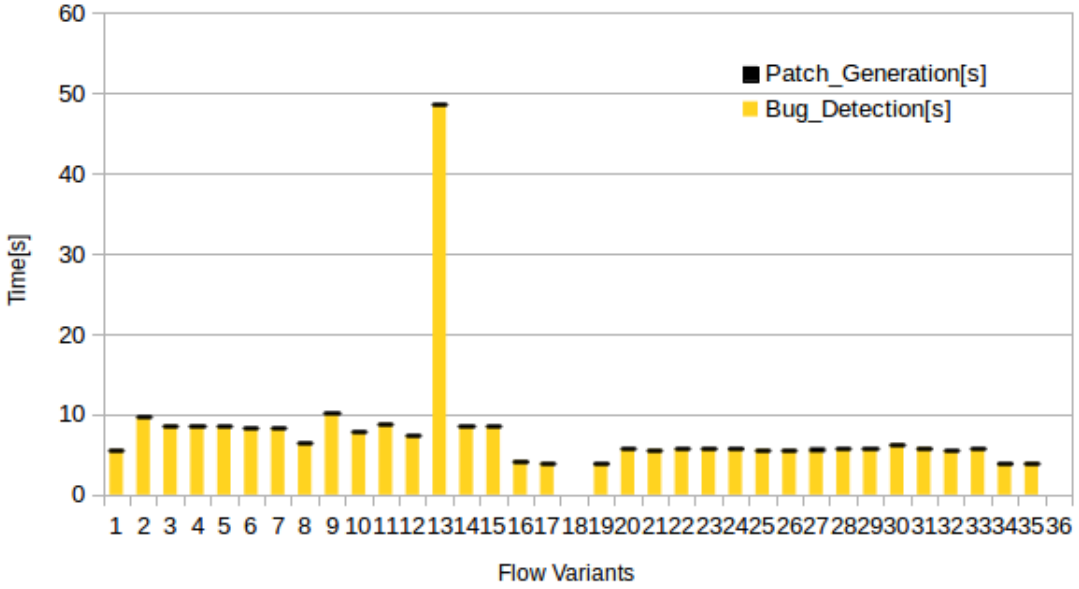


Figure 8.4.: Comparision between the patch generation and bug detection times of CWE-534

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. Change in program behavior after the patch insertion was completely avoided since the buggy variables found in the test programs are all independent variables. So, if any dependant variables are found later in the test programs then the choice of patch insertion is left to the user. From table 8.6, one can observe the column 3 tells us that there were no programs with influencible paths and column 4 tells us that there were no infulencible paths at all. Some of the test cases in the test programs CWE-526, CWE-534 and CWE-535 like CWE526-Env-Var-18, CWE534-Debug-Log-char-18 , CWE534-Debug-Log-wchar-t-18, CWE535-Shell-Error-char-18 , CWE535-Shell-wchar-t-18 , a total of 5 programs couldnot be verified. These programs contain the goto statements for which the SMT solver couldnot generate the executable paths and as a results no program patches were generated. Thus, it can be concluded that the program behavior is not changed at all.

8.6. Usefulness of the generated Patches

The usefulness of the generated patches is addressed by the following scenarios: First scenario is by verifying if the generated code patches are syntactically correct or not and

Test Program	RT	TT	No; of Paths	IP	Total no; of nodes	BDT[ms]
CWE535-shell-char-01	0.001	3.946	67	0	5239	3.945
CWE535-shell-char-02	0.014	7.316	288	0	33700	7.302
CWE535-shell-char-03	0.013	7.379	288	0	33700	7.366
CWE535-shell-char-04	0.015	7.472	288	0	33700	7.457
CWE535-shell-char-05	0.025	7.336	288	0	33700	7.311
CWE535-shell-char-06	0.014	7.41	288	0	33700	7.396
CWE535-shell-char-07	0.017	7.257	288	0	33700	7.24
CWE535-shell-char-08	0.014	7.392	288	0	36070	7.378
CWE535-shell-char-09	0.018	7.367	288	0	33700	7.349
CWE535-shell-char-10	0.005	4.797	112	0	12794	4.792
CWE535-shell-char-11	0.019	7.25	288	0	36070	7.231
CWE535-shell-char-12	0.007	6.509	236	0	22707	6.502
CWE535-shell-char-13	0.012	7.285	288	0	33700	7.273
CWE535-shell-char-14	0.018	7.547	288	0	33700	7.529
CWE535-shell-char-15	0.015	6.921	254	0	29867	6.906
CWE535-shell-char-16	0.004	4.223	71	0	6007	4.219
CWE535-shell-char-17	0.003	4.491	92	0	8910	4.488
CWE535-shell-char-18	0	0	0	0	0	0
CWE535-shell-wchar-01	0.001	3.645	48	0	4198	3.644
CWE535-shell-wchar-02	0.005	5.223	165	0	21568	5.218
CWE535-shell-wchar-03	0.006	5.208	165	0	21568	5.202
CWE535-shell-wchar-04	0.005	5.28	165	0	21568	5.275
CWE535-shell-wchar-05	0.005	5.248	165	0	21568	5.243
CWE535-shell-wchar-06	0.008	5.47	165	0	21568	5.462
CWE535-shell-wchar-07	0.006	5.216	165	0	21568	5.21
CWE535-shell-wchar-08	0.007	5.543	165	0	22876	5.536
CWE535-shell-wchar-09	0.005	5.24	165	0	21568	5.235
CWE535-shell-wchar-10	0.008	5.036	165	0	21568	5.028
CWE535-shell-wchar-11	0.007	5.326	165	0	22876	5.319
CWE535-shell-wchar-12	0.004	5.245	170	0	18037	5.241
CWE535-shell-wchar-13	0.01	5.218	165	0	21568	5.208
CWE535-shell-wchar-14	0.006	5.521	170	0	21568	5.515
CWE535-shell-wchar-15	0.005	5.401	165	0	21730	5.396
CWE535-shell-wchar-16	0.001	3.914	165	0	4922	3.913
CWE535-shell-wchar-17	0.006	5.561	165	0	21568	5.555
CWE535-shell-wchar-18	0	0	0	0	0	0

Table 8.4.: Measured values of CWE-535

also if the refactored code after the patch insertion can be recompiled. Second, verifying if the inserted code patch was useful in removing the detected bugs or not. Table 8.7 column 4 shows the usefulness of the ”not-in-place-fix”, column 1 of table 8.7 shows the name of the test programs, column 3 of table 8.7 shows the usefulness of the ”in-place-fix”. The

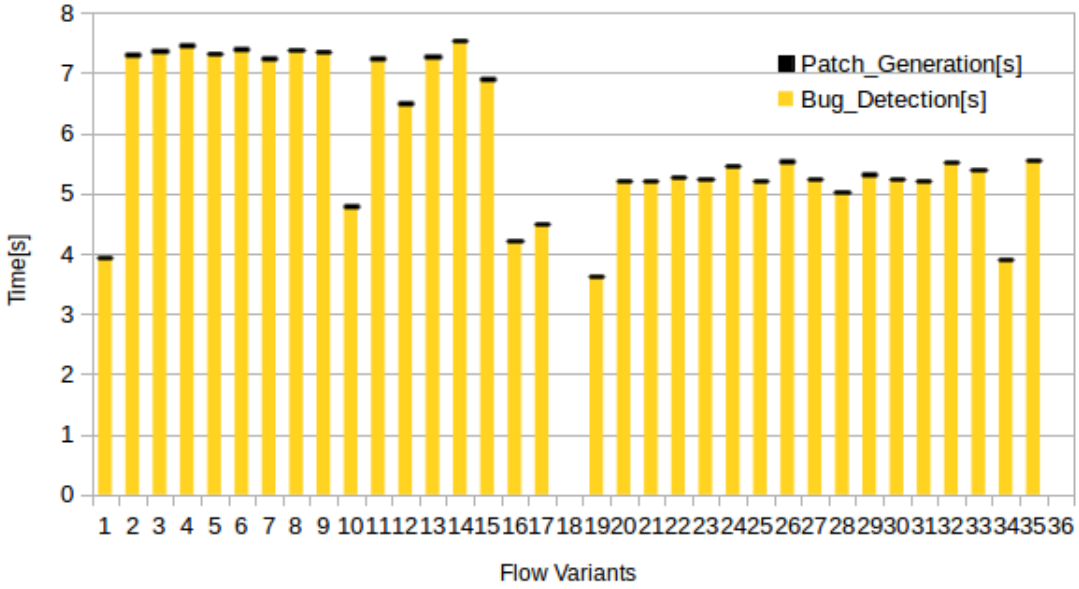


Figure 8.5.: Comparision between the patch generation and bug detection times of CWE-535

Test Programs	TPGT	OTT	TBDT[ms]
CWE-526	0.214	61.068	60.854
CWE-534	0.369	264.753	264.384
CWE-535	0.309	199.193	198.884
Total	0.892	525.014	524.122

Table 8.5.: Overall Overhead

Test Programs	TTP	IP	IPa	% Ratio
CWE-526	18	0	0	0
CWE-534	36	0	0	0
CWE-535	36	0	0	0
Total	90	0	0	0

Table 8.6.: Results of bug fixing

results of the recompilation after patch insertions of ”in-place-fix” and ”not-in-place-fix” is shown in the column 2 of table 8.7. As a result column 3 and column 4 of table 8.7 tells

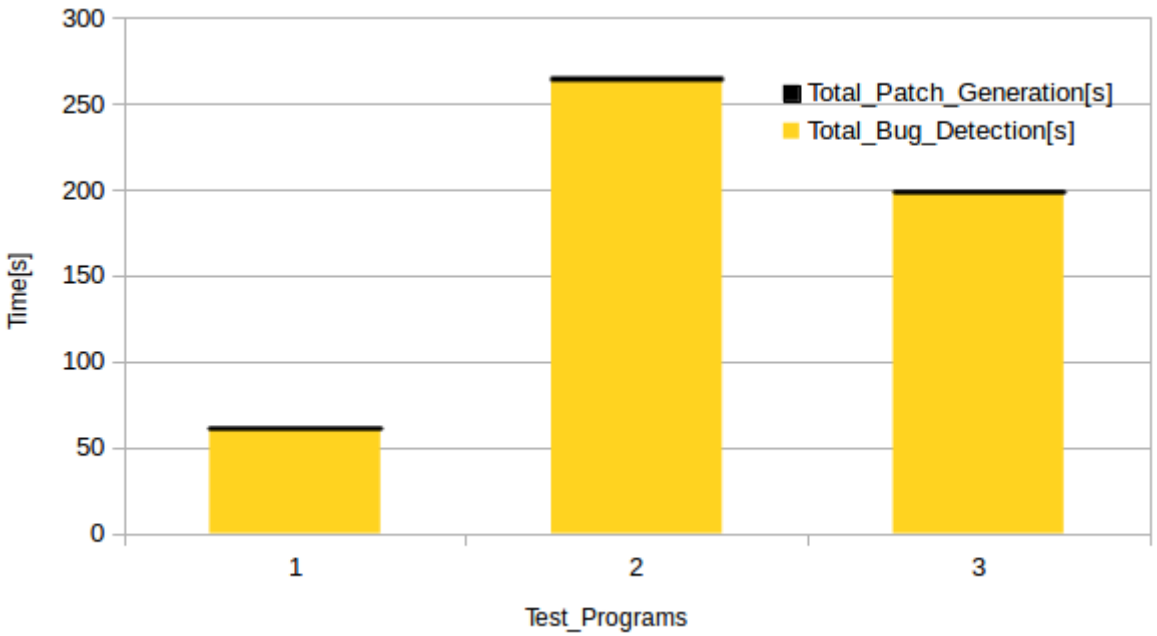


Figure 8.6.: Overall overhead

us that the ”in-place-fix” and the ”not-in-place-fix” were useful in removing the detected bugs.

Here, in the test programs CWE-526, CWE-534, CWE-535 we can observe that there are no influencible paths since there are not dependent variables in the detected bugs which can influence other variables further on in the program. Some of the test cases in the test programs CWE-526, CWE-534 and CWE-535 like CWE526-Env-Var-18, CWE534-Debug-Log-char-18 , CWE534-Debug-Log-wchar-t-18, CWE535-Shell-Error-char-18 , CWE535-Shell-wchar-t-18 , a total of 5 programs couldnot be verified. These programs contain the goto statements for which the SMT solver couldnot generate the executable paths and as a results no program patches were generated. Therefore, it can be concluded that the program behavior will not be changed due to the insertion of this code patches in order to fix the bugs.

Test Programs	Recompile	in-place-fix	not-in-place-fix
CWE-526	✓	✓	✓
CWE-534	✓		✓
CWE-535	✓		✓

Table 8.7.: Results of bug fixing

8.7. Threats to Validity

8.7.1. External Validity:

In external validity we deal with the ability to generalize the empirically evaluated results. In the empirical evaluation that had been carried out 3 test cases, a total of 90 test programs were used. But there are even more other factors related to the testing platform and also the evaluation which could impact the results obtained.

In general the developed IE checker and the bug fixing can be used to other test cases also. If other test cases doesnot contain any function models in SAE then, First all the function models have to be defined, second all the variables which are confidential have to be tainted. Later on the IE checker has to be run and then based on the bug report the refactoring patch has to be decided. Once the patch is generated, it then has to inserted in order to get the refactored bug free code. The patch generation approach is also generalized which includes the steps like detecting the failure, diagnosing the bug, localizing the bug cause, and the repair inference). The overall overhead that has been acquired is 0.17% which is negligble when compared with the execution time.

Presently the available function models are less but in order to defined new function models, one has to follow the same design pattern. Every function model has a number of 5 methods. And all the code used for defining a function model is almost the same for other function models,so one can reuse the same code. This leads to high level of resuse of code.

8.7.2. Internal Validity:

Internal validity deals with the ability to draw some conclusions about the experimental conditions which could make any difference if something is altered in those conditions because the experimental conditions could be either dependent or independent experimental conditions. The definitions of sink and source may sometimes lead to errors or biased for the test programs contianing many lines of code. For the test cases used in thesis are not error prone.

All the test programs which we have used are all publicly available in the Juliet test suite. The test programs were not at all modified. If the execution times which have been calculated are not accurate then the calculated overhead may mislead. The decisions made for the patch generation and selection of patch according to the bug are static. Therefore this will not lead to any overhead introduced by the developed tool. If the IE checker fails to detect the type of the bug then the approach may not work at all or can also suffer from imprecision.

Therefore all these constraints does not pose any internal, external threats to validity.

9. Conclusion and Future Work

It has been successfully proved that the existing IE checker is capable of detecting information exposure bugs. Codan API was used in order to parse the C/C++ source files. Marking the bug locations was easily implemented using the Codan API's markup capabilities. For the test programs containing C goto statements, building CFG was not possible since the existing Codan API doesnot support. In this thesis a novel approach for automatically detecting, localizing and fixing the IE errors have been proposed. The proposed tool can automatically generate code patches using the static analysis along with the defined code patches for removing the errors. The generated code patches are compilable, semi-automatically inserted into the programs containing bugs and these patch do not need any further human refinement. These code patches can be inserted into the buggy program with the help of the developed refactoring tool. The developed localizer algorithm helps in reducing the overall overhead of back tracking since it has the ability to filter out the non-suspicious node. The experimental results show that the developed tool is efficient and can successfully remove all the IE bugs. This approach can also be applied for larger projects since the generated code patche could remove all the bugs and also the program behavior is preserved. This approach can also be applied in future with other bug checkers [18], [19].

Appendix

A. Definitions and Abbrevations

1. **CWE** : Common Weakness Enumeration
2. **CDT** : C/ C++ Development Tools
3. **LTK** : Language Toolkit
4. **CVE** : Common Vulnerabilities and Exposures
5. **IF** : Information Flow
6. **IFE** : Information Flow Exposure
7. **AOWASP** : Open Web Application Security Project
8. **CFG** : Control Flow Graphs
9. **SMT** : Satisfiability Modulo Theories
10. **ISECOM** : Institute for Security and Open Methodologies
11. **IT** : Information Technology

B. Localizer

Listing B.1: Localizer

```
1
2  public class Localizer {
3
4    private static Set<PathLoc> setofPathlocs;
5
6    boolean found = false;
7
8    ArrayList<IASTName> fcnames = new ArrayList<IASTName>();
9
10   private class PathLoc implements Comparable<PathLoc> {
11     IFile filename;
12     int startinglinenumber;
13     IBasicBlock node;
14
15     private PathLoc(IFile file, int starting_line_no, IBasicBlock n) {
16       filename = file;
17       startinglinenumber = starting_line_no;
18       node = n;
19     }
20
21     @Override
22     public int compareTo(PathLoc o) {
23       return (null != o
24            && ((null == this.filename && null == o.filename) || (null
                 != this.filename
25                && null != o.filename && this.filename
26                   .equals(o.filename))) && this.startinglinenumber ==
                      o.startinglinenumber) ? 0
27            : -1;
28     }
29   }
30
31   private PathLoc getloc(IFile file, int starting_line_no, IBasicBlock
         node) {
32     PathLoc ret = new PathLoc(file, starting_line_no, node);
33     if (null == setofPathlocs)
34       setofPathlocs = new HashSet<>();
35     // it checks if the file and line number is already in the
36     // pathloc else it adds it to the pathloc and then returns
37     for (PathLoc p : setofPathlocs)
38       if (p.compareTo(ret) == 0)
39         return p;
```

```
40      setofPathlocs.add(ret);
41      return ret;
42    }
43
44    private class BugLoc {
45      List<PathLoc> path;
46      int index;
47      PathLoc fail;
48      BugInfo info;
49
50      public BugLoc(List<PathLoc> path, int index, BugInfo info) {
51        this.path = path;
52        this.index = index;
53        this.info = info;
54        this.fail = path.get(index);
55      }
56    }
57
58    SymFunctionCall nextCall;
59    SymFunctionReturn nextRVal;
60    WorkPoolManager wpm;
61    Interpreter ps;
62    // list<program path>
63    List<List<PathLoc>> listofpathLocslist;
64    // One list of bug <pathdepth, bugtype>
65    List<BugLoc> localizer_bug_locations;
66
67    private void reset() {
68      listofpathLocslist = new ArrayList<>();
69      localizer_bug_locations = new ArrayList<>();
70      setofPathlocs = null;
71    }
72
73    public Localizer(WorkPoolManager wpm) {
74      reset();
75      this.wpm = wpm;
76    }
77
78    public void reportPath(ProgramPathIBB path) {
79      List<IBasicBlock> nodeslist = new ArrayList<IBasicBlock>();
80      List<PathLoc> pathlocation = new ArrayList<PathLoc>();
81
82      for (IBasicBlock node : path.toProgramPath()) {
83        IASTNode in = null;
84        if (node instanceof CxxPlainNode) {
85          in = ((CxxPlainNode) node).getNode();
86        } else if (node instanceof CxxDecisionNode) {
87          in = ((CxxDecisionNode) node).getNode();
88        } else if (node instanceof IExitNode) {
89          CxxExitNode en = (CxxExitNode) node;
90          in = en.getNode();
91        }
```

```
92
93        IFile filename = null;
94        int line_number = -1;
95        if (null != in
96            && null != in.getTranslationUnit()
97            && null != in.getTranslationUnit()
98                .getOriginatingTranslationUnit()
99            && null != in.getFileLocation()) {
100         filename = (IFile) in.getTranslationUnit()
101             .getOriginatingTranslationUnit().getResource();
102         line_number = in.getFileLocation().getStartingLineNumber();
103         // TODO: dirty hack, remove when cfg does not contain a
                function
104         // call node twice (once before the call, once after)
105         if (pathlocation.contains(getloc(filename, line_number, node))
106             &&
107             /* maybe remove */in instanceof CASTExpressionStatement
108             &&
109             /* these lines */((CASTExpressionStatement) in)
110                 .getExpression() instanceof
                        CASTFunctionCallExpression) {
111           filename = null;
112           line_number = -1;
113         }
114         nodeslist.add(node);
115
116       }
117       pathlocation.add(getloc(filename, line_number, node));
118
119     }
120     // here we have the SatPaths
121     listofpathLocslist.add(pathlocation);
122     for (Map.Entry<Integer, BugInfo> loc : path.bug_locations) {
123       // System.out.println(loc.getKey() + "it is the path ");
124       // adding buginfo ,the depth and list of pathlocs
125       localizer_bug_locations.add(new BugLoc(pathlocation,
              loc.getKey(),
126             loc.getValue()));
127     }
128   }
129
130   public void report(Interpreter ps) throws CoreException {
131     // <buginfo, <location, bad_path_count>>
132     Map<BugInfo, Map<PathLoc, Integer>> bugmap = new HashMap<>();
133     List<PathLoc> badblocks = new ArrayList<>();
134     int maxbad = 0;
135
136     System.out.println("Found " + localizer_bug_locations.size()
137         + " bugs in " + listofpathLocslist.size() + " valid paths.");
138     int j = 0;
139
140     // mark all possibly bad nodes
```

```
141     // if the bugmap initially doesnot contain the same buginfo as
            that of
142     // Bugloc then add it to the bugmap
143     for (BugLoc buglocation : localizer_bug_locations) {
144       if (!bugmap.containsKey(buglocation.info))
145         bugmap.put(buglocation.info, new HashMap<PathLoc, Integer>());
146       // traverse untill the end of the path depth
147       for (int i = 0; i <= buglocation.index; i++) {
148         // if the bugmap has the key loc.path pathloc(file,line number)
149         // then count is that key which is the index else increement
                the
150         // counter
151         // if the bug info is not present then increement the value
152         int count = bugmap.get(buglocation.info).containsKey(
153             buglocation.path.get(i)) ? bugmap.get(buglocation.info)
154             .get(buglocation.path.get(i)) : 0;
155         ;
156         // now add the bug pathloc and the count
157         bugmap.get(buglocation.info).put(buglocation.path.get(i),
158             count + 1);
159
160       }
161
162     }
163
164     for (Map.Entry<BugInfo, Map<PathLoc, Integer>> problem : bugmap
165         .entrySet()) {
166       // remove all definitely good nodes (from paths without the bug)
              and
167       // nodes without a useful location
168       // TODO: maybe calculate the set of good PathLocs before marking
              bad
169       // nodes and only add them to bugmap, if they are not in the good
170       // set.
171       // here we remove the not buggy path from the list of paths which
172       // doesnot contain the bug_locations and the pathloc which
              doesnot
173       // have a
174       // file location and starting line
175       outer: for (List<PathLoc> listofpathloc : listofpathLocslist) {
176         for (BugLoc buglocationsbylocalizer : localizer_bug_locations)
177           if (buglocationsbylocalizer.path == listofpathloc)
178             continue outer;
179         for (PathLoc l : listofpathloc)
180           problem.getValue().remove(l);
181       }
182       problem.getValue().remove(getloc(null, -1, null));
183       // get the locations with the highest count
184       Map<PathLoc, Integer> badcount = problem.getValue();
185       for (PathLoc node : badcount.keySet()) {
186
187         if (badcount.get(node) > maxbad) {
```

```
188
189           badblocks = new ArrayList<>();
190           maxbad = badcount.get(node);
191         }
192         // here we collect the nodes with the same keyvalue
193         if (badcount.get(node) == maxbad) {
194           badblocks.add(node);
195
196         }
197
198       }
199       // report problematic locations
200       ArrayList<PathLoc> badnode = new ArrayList<PathLoc>();
201       ArrayList<PathLoc> same_path = new ArrayList<PathLoc>();
202       HashMap<PathLoc, List<PathLoc>> node_path = new HashMap<PathLoc,
              List<PathLoc>>();
203       for (List<PathLoc> listofpathloc : listofpathLocslist) {
204         for (PathLoc node : badblocks) {
205           if (listofpathloc.contains(node)) {
206             // maintain hashmap of the node and its corresponding
207             // path
208             node_path.put(node, listofpathloc);
209           }
210
211         }
212       }
213
214       for (PathLoc node : node_path.keySet()) {
215         IASTNode in = null;
216         LocalizerVisitor lvisitor = new LocalizerVisitor(ps);
217         // check for the confidential data
218         if (node.node instanceof CxxPlainNode) {
219           CxxPlainNode pn = (CxxPlainNode) node.node;
220           in = pn.getNode();
221
222           if (in != null) {
223             synchronized (in) {
224               in.accept(lvisitor);
225
226             }
227           }
228         } else if (node.node instanceof CxxDecisionNode) {
229           CxxDecisionNode dn = (CxxDecisionNode) node.node;
230           in = dn.getNode();
231           if (in != null) {
232             synchronized (in) {
233               in.accept(lvisitor);
234
235             }
236           }
237         } else if (node.node instanceof CxxExitNode) {
238           CxxExitNode en = (CxxExitNode) node.node;
```

```
239             in = en.getNode();
240             if (in != null) {
241               synchronized (in) {
242                 in.accept(lvisitor);
243
244               }
245             }
246           }
247           // add all the buggy nodes
248           if (null != node.filename && -1 != node.startinglinenumber
249               && lvisitor.isStatus() == true) {
250             // add all buggy nodes
251             badnode.add(node);
252           }
253         }
254         // check if all the nodes belong to different paths
255
256         IASTNode in = null;
257         // if the buggy nodes are more than 1
258         if (badnode.size() > 1) {
259
260           for (int l = 0; l < badnode.size(); l++) {
261             for (int k = l + 1; k < badnode.size(); k++) {
262               // check if the nodes belong to same path
263               if (node_path.get(badnode.get(l)).equals(
264                   node_path.get(badnode.get(k)))) {
265                 // maintain nodes belonging to same path
266                 same_path.add(badnode.get(l));
267
268               } else {
269                 // if does not belong to same path report all
270                 wpm.psReportProblem(problem.getKey().id, in,
271                     badnode.get(l).filename,
272                     badnode.get(l).startinglinenumber,
273                     problem.getKey().problemMessage,
274                     problem.getKey().problemDescription);
275
276               }
277
278             }
279           }
280           // now traverse through the paths of those nodes in the same
281           // path
282
283           List<PathLoc> newlist = node_path.get(same_path.get(0));
284           for (int p = 0; p < newlist.size(); p++) {
285             for (int q = 0; q < same_path.size(); q++) {
286               // here we get to know the earliest node to be fixed
287               if (newlist.get(p).equals(same_path.get(q))) {
288                 if (newlist.get(p).node instanceof CxxPlainNode) {
289                   CxxPlainNode pn = (CxxPlainNode) newlist.get(p).node;
290                   in = pn.getNode();
```

```
291
292                 } else if (newlist.get(p).node instanceof
                        CxxDecisionNode) {
293                   CxxDecisionNode dn = (CxxDecisionNode) newlist
294                       .get(p).node;
295                   in = dn.getNode();
296
297                 } else if (newlist.get(p).node instanceof CxxExitNode) {
298                   CxxExitNode en = (CxxExitNode) newlist.get(p).node;
299                   in = en.getNode();
300                 }
301                 wpm.psReportProblem(problem.getKey().id, in,
302                     badnode.get(p).filename,
303                     badnode.get(p).startinglinenumber,
304                     problem.getKey().problemMessage,
305                     problem.getKey().problemDescription);
306                 break;
307               }
308
309             }
310           }
311         } else {
312           if (badnode.size() != 0) {
313             if (badnode.get(0).node instanceof CxxPlainNode) {
314               CxxPlainNode pn = (CxxPlainNode) badnode.get(0).node;
315               in = pn.getNode();
316
317             } else if (badnode.get(0).node instanceof CxxDecisionNode) {
318               CxxDecisionNode dn = (CxxDecisionNode)
                      badnode.get(0).node;
319               in = dn.getNode();
320
321             } else if (badnode.get(0).node instanceof CxxExitNode) {
322               CxxExitNode en = (CxxExitNode) badnode.get(0).node;
323               in = en.getNode();
324             }
325             wpm.psReportProblem(problem.getKey().id, in,
326                 badnode.get(0).filename,
327                 badnode.get(0).startinglinenumber,
328                 problem.getKey().problemMessage,
329                 problem.getKey().problemDescription);
330           }
331         }
332
333       }
334       reset();
335     }
336 }
```

```
1 package smtcodan.quickfixes;
2
```

```
3  import java.util.ArrayList;
4  import java.util.StringTokenizer;
5  import java.util.regex.Matcher;
6  import java.util.regex.Pattern;
7
8  import smtcodan.Solver;
9
10 public class AssertInformationExposure implements IAssertLogicOperation
       {
11
12   /** The sv. */
13   private Solver sv;
14
15   /** The buggy variable. */
16   public static String buggyVariable;
17
18   /** The race condition quick fix message. */
19   public static String INFORMATION_EXPOSURE_QUICK_FIX_MESSAGE =
         "Quick-Fix Location";
20
21   /** The race condition bug location message. */
22   public static String INFORMATION_EXPOSURE_BUG_LOCATION_MESSAGE =
         "Information Exposure Bug Location";
23
24   /**
25    * Gets the buggy variable.
26    *
27    * @return the buggy variable
28    */
29   public static String getBuggyVariable() {
30     return buggyVariable;
31   }
32
33   /**
34    * Sets the buggy variable.
35    *
36    * @param buggyVariable
37    *          the new buggy variable
38    */
39   public void setBuggyVariable(String buggyVariable) {
40     AssertInformationExposure.buggyVariable = buggyVariable;
41   }
42
43   /**
44    * Gets the information overflow quick fix string.
45    *
46    * @return the information overflow quick fix string
47    */
48
49   public static String getInformationExposureQuickFixString1() {
50     StringTokenizer str = null;
51     ArrayList<String> arguments_fix = new ArrayList<String>();
```

```
52
53      String old = QuickFixInformationExposure.getReplaceStringOld();
54      String var = AssertInformationExposure.getBuggyVariable();
55
56      str = new StringTokenizer(old, ",\";;\\( )", true);
57
58      if (str != null)
59        while (str.hasMoreTokens()) {
60          arguments_fix.add(str.nextToken());
61        }
62      int length1 = arguments_fix.size();
63      int i = 0;
64      int j = 0;
65      int wordcount = 0;
66      int formatcount = 0;
67      for (i = length1 - 1; i > 0; i--) {
68        if ((arguments_fix.get(i).toString()).equals(")")) {
69          for (j = i; j > 0; j--) {
70            if (!arguments_fix.get(j).equals("\"")
71                && arguments_fix.get(j).equals(var)) {
72              wordcount++;
73              arguments_fix.remove(j);
74              arguments_fix.remove(j - 2);
75
76              System.out.println(arguments_fix.get(j - 2));
77
78              break;
79            }
80          }
81
82          break;
83        }
84      }
85      int length2 = arguments_fix.size();
86      for (i = length2 - 1; i > 0; i--) {
87        if ((arguments_fix.get(i).toString()).equals("\"")) {
88          for (j = i - 1; j > 0; j--) {
89            Pattern pattern = Pattern.compile("%.?");
90            Matcher matcher = pattern.matcher(arguments_fix.get(j));
91            if (!arguments_fix.get(j).equals("\"") && matcher.find()) {
92              formatcount++;
93              if (wordcount == formatcount) {
94                arguments_fix.remove(j);
95                System.out.println(arguments_fix);
96
97                break;
98              }
99            }
100         }
101
102         break;
103       }
```

```
104       }
105
106       String fix = "";
107       System.out.println(arguments_fix);
108       for (String s : arguments_fix) {
109         fix += s;
110       }
111       System.out.println(fix);
112       String fix_final = removeLastChar(fix);
113
114       return fix_final;
115     }
116
117     public static String getInformationExposureQuickFixString2() {
118       StringTokenizer str = null;
119       ArrayList<String> arguments_fix = new ArrayList<String>();
120
121       String old = QuickFixInformationExposure.getReplaceStringOld();
122       // String var=AssertInformationExposure.getBuggyVariable();
123
124       String fix = old.replaceAll("\\(.*\\;", "(" + "\""
125           + "Confidential data has been restricted" + "\"" + ")");
126       return fix;
127     }
128
129     /**
130      * Instantiates a new assert buffer overflow.
131      */
132     public AssertInformationExposure() {
133       super();
134       this.sv = new Solver();
135     }
136
137     @Override
138     public String createLinearSystem(SMTConstraintObject ob) {
139       // TODO Auto-generated method stub
140       return null;
141     }
142
143     @Override
144     public void setConstraintValue(String quickFix) {
145       // TODO Auto-generated method stub
146
147     }
148
149     private static String removeLastChar(String str) {
150       return str.substring(0, str.length() - 1);
151     }
152
153 }
```

Bibliography

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

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

[3] Source for Juliet Test Suite. *http://samate.nist.gov/SRD/testsuite.php*.

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

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

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

[7] *http://www.cvedetails.com/cwe-definitions/1/orderbyvulnerabilities.html?order=2&trc=668&sha=0427874cc4* jan 2015.

[8] 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.

[9] 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.

[10] 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.

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

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

[13] 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.

[14] 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.

[15] 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.

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

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

[18] A. Ibing. Path-Sensitive Race Detection with Partial Order Reduced Symbolic Execution. In Carlos Canal and Akram Idani, editors, *Software Engineering and Formal Methods*, volume 8938 of *Lecture Notes in Computer Science*, pages 311–322. Springer International Publishing.

[19] Andreas Ibing and Alexandra Mai. A Fixed-Point Algorithm for Automated Static Detection of Infinite Loops. In *Proceedings of the 2015 IEEE 16th International Symposium on High Assurance Systems Engineering*, HASE '15, pages 44–51, Washington, DC, USA, 2015. IEEE Computer Society.

[20] 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.

[21] 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.

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

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

[24] 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.

[25] 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.

[26] 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.

[27] 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*

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

[29] 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.

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

[31] 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.