Draft Version Wednesday 29th October, 2014

Program Repairing using Exception Types, Constraint Automata and Typestate

Student Name: Aritra Dhar

IIIT-D-MTech-CS-IS-12-004
Dec 08, 2014

Indraprastha Institute of Information Technology
New Delhi

Thesis Committee
Dr. Rahul Purandare (Advisor)

Submitted in partial fulfillment of the requirements
for the Degree of M.Tech. in Computer Science,
with specialization in Information Security

©2014 Aritra Dhar
All rights reserved

Keywords: Software Engineering, Program Repairing, Availability, Program Analysis, Static Analysis, Exception

Certificate

This is to certify that the thesis titled **"Program Repairing using Exception Types, Constraint Automata and Typestatee"** submitted by **Aritra Dhar** for the partial fulfillment of the requirements for the degree of *Master of Technology* in *Computer Science & Engineering* is a record of the bona fide work carried out by him under my guidance and supervision in the Program Analysis Research group at Indraprastha Institute of Information Technology, Delhi. This work has not been submitted anywhere else for the reward of any other degree.

Dr. Rahul Purandare
Indraprastha Institute of Information Technology, New Delhi

Abstract

Runtime Exceptions are common types of exceptions which may lead to system crash which results in shutdown or restart. For may critical application such scenario is unacceptable due to their nature which requires availability of the service.

Program bugs which causes runtime exceptions often go unnoticed at the time of development as these exceptions are unchecked exceptions. The key issue is to guide the program through some exception suppression procedure which will leads the program to a consistent state hence improve the chance of surviving a fatal crash. Here we consider such programs for which restart is not an option.

In our work, we present a novel technique to recover from unexpected runtime exceptions. We have used hybrid of two techniques for efficient detection of potential point of failure and patch it closest to that to minimize the damage. One technique uses type of runtime exception to apply appropriate patch. The other technique will provides typestate analysis technique which will detect typestate violations to apply the right patch. We also performed static analysis to taint variables and object to make sure no patching is applied to those which leaves the system. We only patched those statements for which the associated variables and objects stay and die inside the program. The taint analysis phase is follwed by the repairing phase where we tried to apply the patch close to the point of exception thus minimizing the effect of exception.

Acknowledgments

Revision

This work would not have been possible without support from a number of people. Foremost, I would like to extend my deepest gratitude to Dr. Rahul Purandare for his expert guidance and for the extremely productive brainstorming sessions I had with him. I am grateful to my friends, seniors and juniors who offered fresh perspectives on my research. I am also thankful to IIIT-Delhi for providing excellent infrastructure and support. Last but never the least, I am immensely grateful to my parents, family members and close friends, for their invaluable support and unconditional love.

Aritra Dhar

New Delhi

Wednesday 29th October, 2014

Dedicated to,

Ma, Baba and Dida

Contents

List of Figures

List of Tables

List of Code Snippets

Chapter 1

Introduction

Exception handling [13] attributes to the response of program during runtime to some exceptional condition encounter. Most of the time it changes normal flow of program. In many cases exception handling is natural part of software execution due to the nature of the software. An application which constantly accesses I/O which also includes share resources may throw exception if another application blocks it. Here in this paper we discuss and analyze java exceptions and produce repair patch based on that. Java supports two types of exceptions :

- **Checked exception** which requires explicit *throws* declaration at the method declaration or *try-catch* block by the developers. Such exceptions are handled carefully as they often involves accessing resources like network, database, file system, I/O etc. Code snippet 1.1 and 1.2 is an example of a checked exception type when one needs to read userinput from the console.

- **Unchecked exception** which does not enforce similar handing mechanism as the former one. *java.lang.RuntimeException* [12] and its subclasses and *java.lang.Error* [11] are types of unchecked exceptions. *NullPointerException*, *ArrayIndexOutOfBound*, *ArithmeticException* are examples of common java runtime exceptions. Code snippet 1.3 is an eample where the method returns an integer after performing division operation. If the denominator is zero then this operation will throw an *ArithmeticException divide-by-zero*. But as arithmetic exception is a runtime exception and which is a unchecked exception, we do not need to write a throw keyword or put try-catch block.

Code Snippets 1.1: Example 1 of java checked exception

```java
void foo() throws IOException
{
 InputStremReader is = new InputStremReader(System.in);
 BufferedReader br = new BufferedReader(is);
 String str = br.readline();
}
```

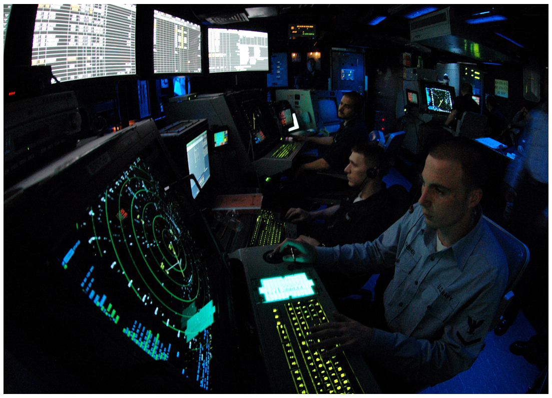
Code Snippets 1.2: Example 2 of java checked exception

```java
void foo()
{
 try
 {
  InputStremReader is = new InputStremReader(System.in);
  BufferedReader br = new BufferedReader(is);
  String str = br.readline();
 }
 catch(IOException ex)
 {
  System.err.println("IO Exception happend");
 }
}
```

Code Snippets 1.3: Example of java unchecked exception

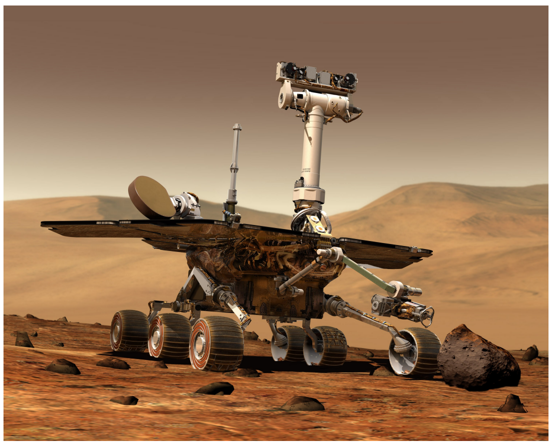
```java
int foo(int a, int b)
{
 return a/b;
}
```

Oracle official documentation says that "*Here's the bottom line guideline: If a client can reasonably be expected to recover from an exception, make it a checked exception. If a client cannot do anything to recover from the exception, make it an unchecked exception*". Unchecked exception, particularly runtime exceptions can be thrown from any point in the program making them quite unpredictable in nature. Due to this extensive testing phase is required to eliminate any bugs and solve corner cases. Yet many applications suffer unexpected runtime exception causing system crash which leads to shutdown or restart.



(a) Air Traffic Controller (ATC)

(b) Unmanned Aerial Vehicle (UAV)



(c) Mars Rover

(d) Life Support System

Figure 1.1: Several Realtime Critical Applications

We find out many applications where system shutdown/restart is expensive due to their nature. Notable examples are air traffic control, auto pilot, life support system, smart power grids, telephone networks, robots like UAV and rovers deployed for surveillance, reconnaissance and knowledge acquisition in remote locations etc. These applications are real-time sensitive and there is no room for exception handling in such system. Sudden crash involves risk of human life, expensive equipments and critical services. Other example includes web applications which uses scrips to dynamically generate websites and interfaces as per customer preferences. Many E-commerce websites handles queries, access and process customer and shopping items data and commits large amount of transactions. Sudden system crash may result in loss of precious time and data which eventually may result in a frustrated customers move to other websites. Many time bad or malicious code leads to some vulnerability to critical applications and website which can be exploited by attack to orchestrate system crash. Thought these examples cover a large variety of applications, all of them point to some concern of *availability*.

Usually, developers tests their code in series of verifications which involves code review, static and dynamic analysis of the code, generate test cases to cover as much potential input .Yet may corner cases can be left overlooked which can cause runtime exceptions.

Multi-threaded applications are also susceptible to erroneous thread interleaving. One such exception is *java.lang.IllegalMonitorStateException*, when a thread has attempted to wait on an object's monitor or to notify other threads waiting on an object's monitor without owning the specified monitor. Applications under adversarial situation should be considered where deliberate malicious input may cause it to fail. To recover from such situation, a mechanism is needed which can predict failure by doing invariant and symbolic analysis. Invariant analysis will detect particular variables outside legal/safe bound. Static analysis will indicate to the potential point of failure.

In this paper we proposed two solution to suppress runtime example and ensure system survivability. The approach consists of four primary phases

- **Taint Analysis** : We performed static taint analysis to mark all the program paths which lead to some tainted sink. The taint analysis goes like the following : we marked such variables and objects which are coming from outside world (like user input from web page or console), then we looked for taint propagating methods by which the tainting can be propagated to some no tainted variable or objects from some tainted ones. Then we also enlisted some taint sink such as console out put or database commit, which indicates that certain tainted variables or objects leaves the system. We marked such paths and such statement which involved tainted variables/objects as patching on such variables is not recommended.

- **Generate input data-set** : We index user input along with the global variables and method arguments of successful runs. The local variables are not indexed as they can be re-generated. These data-set is used as a reference to later executions which encounters runtime exceptions. Appropriate user input of previous successful run is chosen in terms of correlation coefficient.

- **Program slice for patching** : We perform static analysis prior to running the program to determine data dependencies of the variables. The analysis yields a dependency graph which is used to determine optimal slice to be used as patch. This patch is placed in catch block and executed with the values of previous successful run while the original code is wrapped in try block.

- **Determine type of exception and patching** : The characteristics of patching is dependent on the type of runtime exception encountered by the program. A piece of code may throws multiple types of exceptions and all of them are handled at the time of patching by instrumenting multiple catch blocks.

- **Use typestate for repairing** : Typestate analysis, sometimes called protocol analysis defines valid sequences of operations that can be typically modeled using Finite State Machine (FSM) where the states represent abstract state of the program and the symbols are certain method invocations to perform state transition. Typestates are capable of representing behavioral type refinements like Iterators, where *hasNext()* method should be called before the *next()* method call. Typestate analysis is widely used as a safety feature to ensure a certain sequence of operations maintains proper protocol or not. The documentations of the API used in the application will define the valid typestate for repairing.

The object of the patching is to repair the problem closest to it to minimize any collateral damage to other parts of the applications hence minimizing the chance of unintentional data loss/corruption.

Chapter 2

Motivation and Challenges

2.1 Historical Context

In recent past, we have seen couple of disastrous failure of critical military and civilian infrastructure system due to system failure/crash which is results of some very common runtime exceptions.

- In USS Yorktown, complete failure in propulsion and navigation system by a simple divide-by-zero exception in flight deck database (1998) [16].
- AT&T telephone network failure causing by one faulty switch causing ATC commutation blackout.
- Air-Traffic Control System in LA Airport lost communication with all 400 airplanes caused by a system crash triggered by integer (32bit) overflow [9].
- Mars rover curiosity B-side computer memory overflow causing OS suspend and multiple restart.
- Trans-Siberian Gas Pipeline Explosion in 1982 by deliberate bugs in software controlled valves.
- Near-blackout of the national grid in Austria caused by faulty function call.

All of these incidents have one thing common, all of them were critical system where availability is the major requirement. Most of the systems are such critical that in case of failure one can not simple shutdown and restart the system like general client applications as it may results in loss of human lives and massive amount of money.

2.2 Data from Stack Overflow Posts

Runtime Exception Type	**Occurrences**	**Percentage**
NullPointerException	34912	54.94%
ClassCastException	7504	11.81%
IndexOutOfBoundsException	6637	10.44%
SecurityException	5818	9.15%
NoSuchElementException	2392	3.76%
ArithmeticException	2338	3.67%
ConcurrentModificationExceptio	1889	2.97%
DOMException	1024	1.61%
ArrayStoreException	279	0.43%
MissingResourceException	277	0.43%
BufferOverFlowException	161	0.25%
NegativeArraySizeException	122	0.19%
BufferUnderFlowException	66	0.1%
LSException	64	0.1%
MalformedParameterizedTypeExce	38	0.05%
CMMException	8	0.01%
FileSystemNotFoundException	6	0.009%
NoSuchMechanismException	3	0.0045%
MirroredTypesException	1	0.0015%

Table 2.1: Most frequent java runtime exceptions from stack overflow data

We have analyzed data from stack overflow and we looked for java runtime exception which are discussed most frequently. In the table 2.1, the data we find is tabulated along with their occurrences and percentages.

From the table it is clear that null pointer exception in java is not only the most frequent but also the most dominant runtime exception having share of more than 50%. This data is highly motivational for us as there are number of cases where Java developers encounters software bugs which are mostly based on runtime exception.

2.3 Major Challenges

The challenges we faced during the research can be described in few points :

1. The major challenge was that the program we try to patch, in all the cases we actually don't know the internal logic of the program, we have patched it based on its behavior which can be damaging to the system itself. To prevent this we have tainted input variables which are coming from out side environment and see how they are interacting with other variables and object in the program. In case any of these variables go outside of the system, we marked the path to make sure patch won't be applied along that path.

2. Most of the patching are non-trivial in nature and adaptive based on the use case to take full advantages what resources available in the code rather than some deterministic patching technique.

Chapter 3

Related Works

3.1 Recent Works on Data Structure Repairing

Automated data-structure repairing techniques are there in the litarature for a while. In the papers [3], [2], [5], [4], [1] the authors mostly concentrated on specific data-structures like *FAT-32*, *ext2*, *CTAS* (a set of air-traffic control tools developed at the NASA Ames research center) and repairing them. The authors represented a specification language by which they able to see consistency property these data-structure. Given the specification, they able to detect the inconsistency of these data-structures and repair them. The repairing strategy involves detecting the consistency constraints for the particular data structure, for the violation, they replace the error condition with correct proposition. In the paper [5], the authors proposed repair strategy by goal-directed reasoning. This involves translating the data-structure to a abstract model by a set of model definition rules. The actual repair involves model reconstruction and statically mapped it to a data structure update. In their paper [6] authors Elkarablieh et al. proposed the idea to statically analyze the data structure to access the information like recurrent fields and local fields. They used their technique to some well known data structures like singly linked list, sorted list, doubly liked list, N-ary tree, AVL tree, binary search tree, disjoint set, red-black tree, Fibonacci heap etc.

3.2 Works on Software Patching

In their paper [14], authors Jeff H. Perkins et al. presented their *Clear view* system which works on windows x86 binaries without requiring any source code. They used invariants analysis for which they used Daikon [7]. They mostly patched security vulnerabilities by some candidate repair patches.

Fan Lon et al in their paper [10] presented their new system *RCV* which recovers applications from divide-by-zero and null-deference error. Their tool replaces *SIGFPE* and *SIGSEGV* signal handler with its own handler. The approach simply works by assigning zero at the time of divide-by-zero error, read zero and ignores write at the time of null-deference error. Their implementation was on $x86$ and $x86-64$ binaries and they also implemented a dynamic taint analysis to see the effect of their patching until the program stabilizes which they called as *error shepherding*.

3.3 Genetic Programming, Evolutionary Computation

Reserch works on program repair based on genetic programming and evolutionary computation can be found in the paper of Stephanie Forrest et al. [8] and Westley Weimer et al [15] respectively. In the papers, the authors used genetic programming to generate and evaluate test cases. They used their technique on the well known Microsoft Zune media player bug causing tme to freeze up.

Chapter 4

Problem Formulation

This part is incomplete, I am now writing the strategy part

We formulate the problem in following way

4.1 Runtime Exceptions

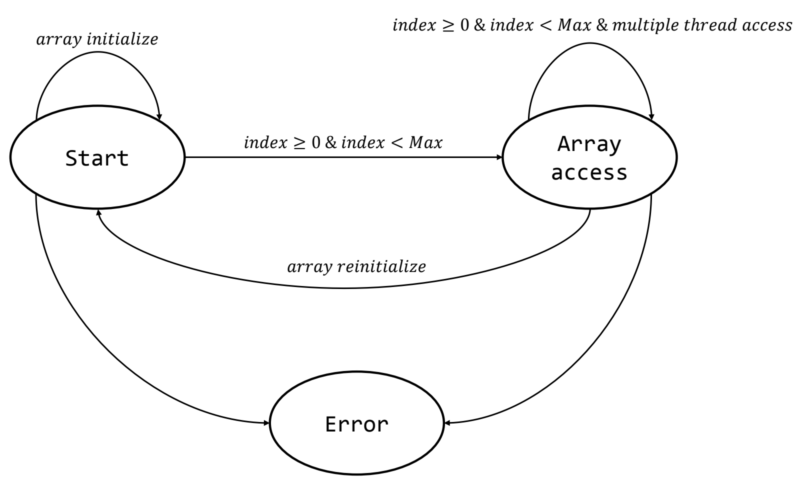


Figure 4.1: array index out of bound formulated as FSM

We can visualize all runtime exceptions as finite state machine (FSM). When a program violates such sequence, it throws runtime exception. In Figure 4.1, array index out of bound (java.lang. ArrayIndexOutOfBoundException) exception is described as a FSM. Here, a program will be in safe bound as long as the $array\_index \geq 0$ or $array\_index \leq max\_array\_size - 1$

Chapter 5

RepairingStrategy : Taint Analysis

We have used taint analysis to detect program paths between source-sink pair in the program to determine which variables and objects go to tainted sink like database, print stream, network stream etc. We have used InfoFlow framework and modify it for our usage. The detaild design of the taint analysis module is given in Chapter 9 Section 9.1.1.

5.1 Taint analysis : Definition

The term **taint** in the aspect of programming language is defined as below:

Definition 5.1.1. Set of variables which are associated with program input is the set of tainted variables.

Definition 5.1.2. Variables which are associated or referenced from tainted variables are also tainted.

So, the set of variables are called as **tainted variable set** which may trigger some undesirable events in the application.

5.2 Taint analysis : Taint Propagation

All tainted variables do not possess security threat. The tainted problem is defined at three points. They are:

1. Source descriptor $< m, n_s, p_s >$
2. Derivation descriptor $< m, n_d, p_d >$
3. Sink descriptor $< m, n_s, n_d, p_s, p_d >$

Where m is the method, n is the number of parameter(s), p is the access path.s and d denotes to source and sink(destination) respectively.

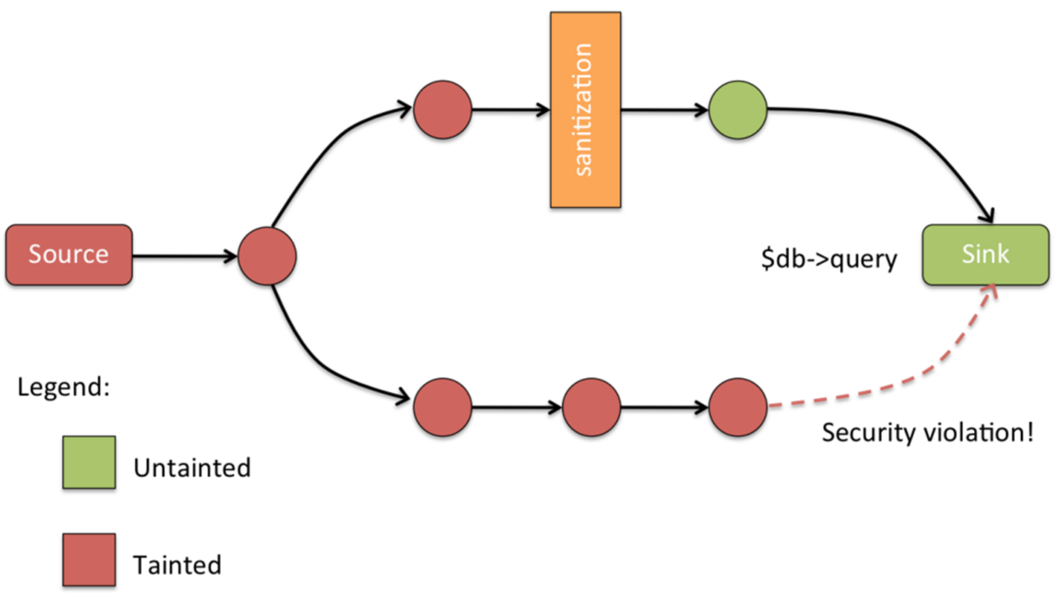


Figure 5.1: A simplified diagram indicating taint problem

5.3 Taint Analysis : Relevance with Repairing Effort

We have considered static taint analysis of the program (here we are analyzing only java byte code) to eliminate any possibility of patching on the statements which may go to some tainted sink like database, print stream or network stream. Doing such we can ensure that the variables and objects we are patching will be contained inside the system thus will not be leaked to outside. On such example can be a client application on which we have done patching. Assume that we patched a string object which was given as a input to the program. Due to some formatting problem, the program throws a runtime exception. Ins uch scenario we will regenerate the string object according to the constraint in the program to make sure it stays very close to a clean input string. in any case the generated string goes out from the system and used as a input to any external module it may causes problem as the patched sting was solely designed for that particular program.

To avoid such cases we analyze the statement which in in the path of potential tainted source and sink. In such cases we would not patch such statements.

Chapter 6

Repairing Strategy : Exception Type

Please review this section

Code Snippets 6.1: Java code which may throws runtime exceptions

```java

public class TestClass
{
	private int[] arr1;
	private int[] arr2;
	private int[] arr3;

	public TestClass(int[] arr1, int[] arr2, int[] arr3)
	{
		this.arr1 = arr1;
		this.arr2 = arr2;
		this.arr3 = arr3;
	}
	public int[] fun(int a, int b, int c, int d)
	{
		int temp0 = a + b;
		int temp1 = c * d;
```

```
            int temp2 = temp0 - temp1;
            //array index out of bound, negative index
            int temp3 = this.arr1[temp0];
            //array index out of bound, negative index
            int temp4 = this.arr2[temp1];
            //array index out of bound, negative index
            int temp5 = this.arr3[temp3];
            int temp6 = temp4 + temp5;
            int temp7 = temp6 - temp3;
            //array index out of bound, negative index, divide by zero
            this.arr1[temp6] = temp7/(d-a);
            //array index out of bound, negative index, divide by zero
            this.arr2[temp7] = temp7/temp4;
            if(arr2[temp1] ! = arr3[temp7])
                return arr1;
            else
                return null;
        }
}
public class MainClass
{
        public void main(String[] a)
        {
            int[] arr1 = {1,2,3,4};
            int[] arr2 = {1,2,3,4};
            int[] arr3 = {1,2,3,4};
            TestClass TC = new TestClass(arr1, arr2, arr3);
            int[] res = TC.fun(2,4,3,4);
            //Null pointer exception
            System.out.print("Result : "+res[2]);
        }
}
```

In the Example 6.1, we have given a piece of java code which shows multiple lines can throw several runtime exceptions. In this example we consider three very common runtime exceptions: NullPointerException, ArrayIndexOutOfBoundExcepltion, NegetiveIndexException, ArithmeticException (i.e. divide-by-zero). In rest of this section, this particular example will be used to demonstrate the repairing strategy.

6.1 Static Analysis

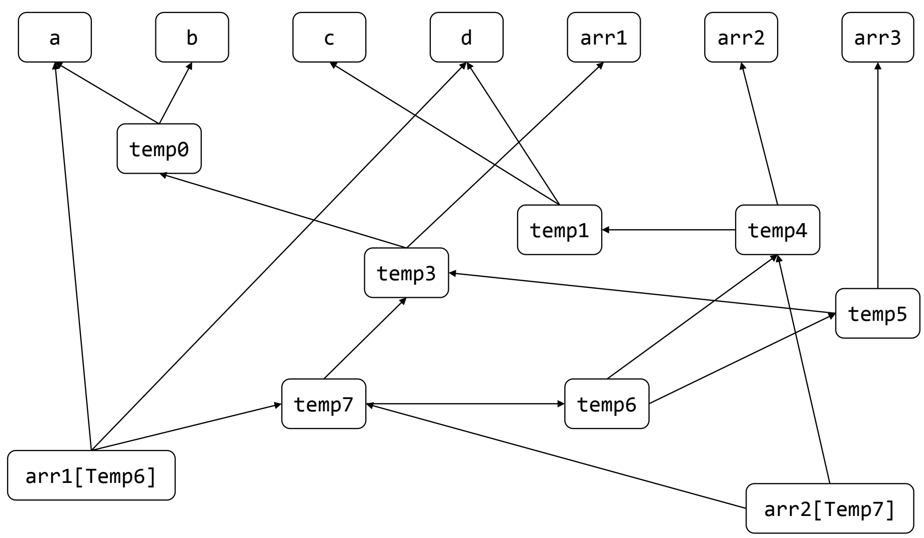


Figure 6.1: Data dependency graph of the variables in code snippet 6.1

We have done several static analysis a priori over the Java source code to discover :

1. Critical section of the code which are not eligible for patching. Eg. banking or any financial transaction which should be crashed in case of exception as suboptimal solution due to patching will led it to inconsistent state. This information will be available from the taint analysis module which will take place before the repairing module.

2. We also analyze all the methods for method specific shilding as they can be called from the paths leading to both tainted sink and non tainted sink. The detailed description is available in Section 9.2.1.

3. Static analysis of the program to discover potential points of failure and mark them.

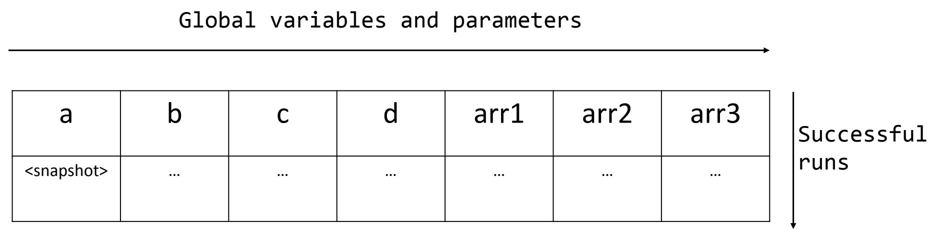


Figure 6.2: Indexed global variables and method arguments successful runs

4. Build data dependency graph which will be used to generate appropriate code slice to be used as patch. In Figure 6.1, the data dependency graph of the code snippet 6.1 is presented.

5. The static analysis will also reveal which kind of exception is likely to happened at the time of execution. This information is necessary at the time of instrumenting the patch as it will determine the catch block.

6.2 Data set for Successful Program Runs

Here we stored all the traces of successful program runs. Figure 6.2 shows such indexed traces of all the global variables and method arguments. We store the snapshots of these objects. We won't store local variables as they can always be regenerated. As it is required to capture the snapshot of all these variable, we made deep cone of all of these objects and variables.

6.3 Matrices

Please review this section.

6.4 Instrumenting Patching

We have used Soot framework which is a Java byte code manipulator to instrument patch. The patching technique is divided into two phases

6.4.1 Determine Exception Type

At the time of execution, the exception may happened due to some specific values of some variables. We will catch the exception. Here the type of runtime exception is $java.lang.ArrayIndexOutOfBound$. This will be used to produce the try-catch block.

6.4.2 Determine Optimal Code Slice

The optimal code slice will be determined from the data dependency graph which was rendered at the time of static analysis mentioned in Section 6.1. In the code snippet 6.2, the example code snippet shows such code slice inside the catch block. As the error occurred at the line $int\ temp5 = this.arr3[temp3]$; the statements which produces the temp3 and the statement which also involves $temp3$ or any other variables derived from $temp3$, would be included in the catch block for re-execution with the valued of the same from the data table of previous successful runs.

Code Snippets 6.2: Patching code slice based on exception type

```

public class TestClass
{
	private int[] arr1;
	private int[] arr2;
	private int[] arr3;

	public TestClass(int[] arr1, int[] arr2, int[] arr3)
	{
		this.arr1 = arr1;
		this.arr2 = arr2;
		this.arr3 = arr3;
	}
	public int[] fun(int a, int b, int c, int d)
	{
		try
		{
		  int temp0 = a + b;
		  int temp1 = c * d;
		  int temp2 = temp0 - temp1;
		  int temp3 = this.arr1[temp0];
```

```
                  int temp4 = this.arr2[temp1];
          //IndexOutOfBoundException as temp3 = 20
                  int temp5 = this.arr3[temp3];
                  int temp6 = temp4 + temp5;
                  int temp7 = temp6 - temp3;
                  this.arr1[temp6] = temp7/(d-a);
                  this.arr2[temp7] = temp7/temp4;
                }
                catch(IndexOutOfBoundsException indEx)
                {
                  int temp0 = a + b;
                  int temp1 = c * d;
                  int temp2 = temp0 - temp1;
                  int temp3 = this.arr1[temp0];
                //Bellow line is not part of the patch as
                //temp1 and temp3are not related to temp3
                //for which the exception occurred.
                  //int temp4 = this.arr2[temp1];
                  int temp5 = this.arr3[temp3];
                }
                if(arr2[temp1] ! = arr3[temp7])
                        return arr1;
                else
                        return null;
        }
}
public class MainClass
{
        public void main(String[] a)
        {
                int[] arr1 = {20,21,22,23};
                int[] arr2 = {1,2,3,4};
                int[] arr3 = {10,11,12,13};
                TestClass TC = new TestClass(arr1, arr2, arr3);
                int[] res = TC.fun(2,4,3,2);
                System.out.print("Result : "+res[2]);
        }
}
```

6.5 Variable Tracking and Monitoring

I have added standard taint analysis technique here as an example. We can change it later

Here we used taint analysis technique to tag variables and objects of our interest to monitor them. This steps are necessary as the values of the variables used during the instrumentation may cause further runtime exceptions. We used bit-vector which is an efficient technique to taint a object/variable. It requires maintain a single dimension byte array where each bit correspond to a single object/variable of our interest. The bit values will be flipped when it is required to taint (1) or untaint (1) an object/variable. We will only monitor these entities until all of them flushed from the program and the entire program reached to a stable state.

Chapter 7

Repairing Strategy : Bounded Forward and Backward Analysis

7.1 Example Scenario

We have performed dataflow analysis by extending Soot main class. The objectives of the dataflow analysis are the following:

- For a target statement analyze used and defined variables.
- Extracts other statements which are both above and bellow the target statement in the control flow graph on which the used and defined variables are dependent on.

In the code snippet 7.1, we gave an example code based on java *String* API to demonstrate the analysis.

Code Snippets 7.1: Dataflow analysis

```
void bar()
{
  foo("fname:lname");
}

String foo(String s)
{
```

```
  int a = s.indexof(":");
  int b = s.indexOf("&");
  int c = s.indexOf("#");
  int d = 0;
  if(c>0)
  {
    d = 1;
  }
  return s.substring(a,b);
}
```

Let us assume that our target is `s.substring(a,b)` which in this case may throw an array index out of bound exception. In this target statement, `a` and `b` are used variable which are dependent on another String API method i.e `indexOf()` which calculates index of starting of a sub-string or single character in the main string. In case the sub-string or the character does not exist in the main string, `indexOf()` method returns -1 which causes throwing a runtime exception in the `substring()` method call.
By using dataflow analysis we try to understand how these different variables are correlated and based on that how we can effectively apply patching technique so the patching code will have very less footprint in the instrumented bytecode. In the Section 7.2.1, we have given detailed explanations of such analysis.

7.2 Flow Functions

7.2.1 Bounded Forward Analysis

Let us define P_i as a program point/ node in the control flow graph. $in(P)$ and $out(P_i)$ respectively denotes in set and out set to and from the node P. We define set IG as the set of methods like `indexOf()`, `codePointAt()`, `CodePointBefore()` etc. which returns an integer which can be used as input to other String methods. We also define set IU which contains the methods which may use the integers produced by the methods in IG Then,

$$out(P_i) = in(P_i) \cup Def(P_i)$$

where statement in P is a invoke statement and method $m \in IG$ and

$$out(P_i) = in(P_i) \cap Used(P_i)$$

where statement in P is a invoke statement and method $m \in IU$. Initial entry set $= \phi$.

We have defined $Def(P_i)$ set as the set of variables and objects which are defined or redefined

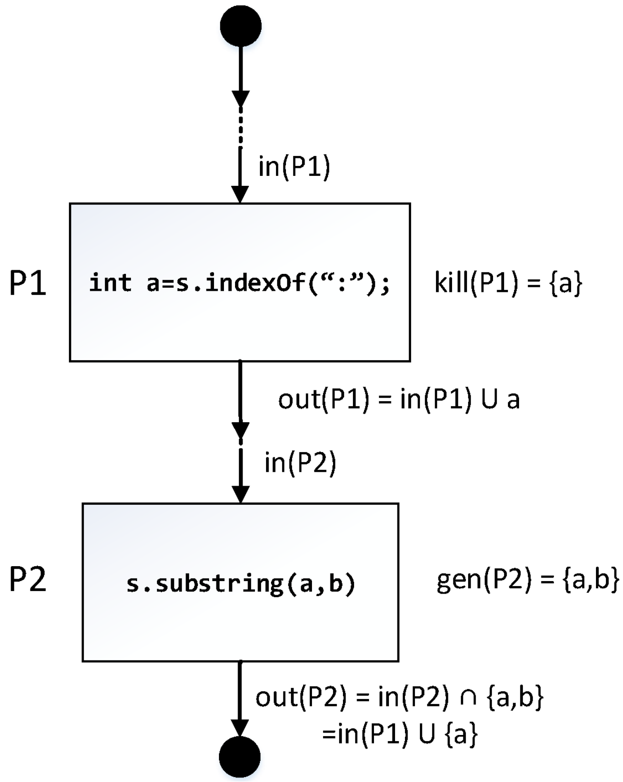


Figure 7.1: Dataflow diagram with in, out set in forward analysis

in the program point P_i. The set $Used(P_i)$ is also a set of variables and objects which are used in the program point P_i.

Example : Consider the program statement `Pi : int a = b.fun(c d)`. Here the variable `a` is initialized, so $Def(P_i) = \{$`a`$\}$ and as `b, c, d` are used, $Used(P_i) = \{$`b, c, d`$\}$

In the figure 7.1, we gave an example of a sample CFG with in set and out set.

7.3 Constraint Satisfaction

Dataflow analysis plays an important role in preparing the patching. One patching mechanism we have come up with `String` objects is tht by solving constraints which may come up in future will produce patch of better quality. More over, it is very easy to extend the solution to other objects type based on their API and characteristics of conditions. One such example is given in the following code snippet 7.2

Code Snippets 7.2: Better patching mechanism with constraint satisfaction

```

void foo(String s, int i, int j)
{
```

```
4         String str = s.substring(i,j);
5         //some operation
6         if(str.length() > 12){
7           //do something..
8         }
9         Integer in = 0;
10        try{
11          StreamReader isr = new InputStreamReader(System.in);
12          String sin = new BufferedReader(isr).readLine();
13          in = Integer.parseInt(sin);
14        }
15        catch(IOException ex){}
16        if(str.length() <= in){
17          //do something..
18        }
19        if(str.startsWith(SomeStringObject)){
20          //do something
21        }
22 }
```

In the code snippet 7.2, the statement at line no 4 is `s.substring(i,j)`, which can throw a `IndexOutOfBoundsException`. This statement requires patching whic involves generating a string for the object reference `str`. But in the progrm, in line numbers `7, 16` and `20`, there are threee conditional statements on `str` which involves constraint on the length and the prefix of the string. There may be some set of constraint which can be evaluated before hand, like the condition in in line numbers `7` which involve a constant integer. But there can be cases like the conditional statement in line numbers `16` which is also a lenght constraint like the former, but in involves another variable which is taken frrom console, i.e. the variable will be evaluated in run time. In such cases we can defer the constraint evaluation process for that paricular condition. We can evaluate all the conditions befor it, which can be safely evaluated. When we reach line number `16`, then the variable `tt` would be available and can be used to reevaluate the string `str`.

7.3.1 Constraint Storage

For each of the string object, we store in the way illustrated in the Figure 7.2.

Minimum length	Maximum length(L)	Prefix 1	Prefix 2	...	Prefix L-1	Contain 1	...	Contain L-1

Figure 7.2: String constraints storage format

Wen to evaluate a new string object we need bounds like the minimuma and maximum length, the prefixes and the candidate characters and their relative position. We keep minimum information to safely evaluate the string.

7.3.2 Constraint Evaluation Strategy

Algorithm 1: String object constraint evaluation

Data: String object Str and constraint set CS.
Result: String object Str such that $\forall i \in CS$, Str satisfies i

```
begin
    CS_Str ⟵ Get the constrint set for Str
    MinLength ⟵ CS_Str[0]
    MaxLength ⟵ CS_Str[1]
    PrefixSet_Str ⟵ CS_Str[2 → MaxLength + 1]
    ContainSet_Str ⟵ CS_Str[MaxLength + 2 → 2 * MaxLength + 1]
    for C ∈ PrefixSet_Str do
        if C is Empty then
            continue
        PrefixLength ⟵ LENGTH OF C
        if PrefixLength is Maximum ∈ PrefixSet_Str then
            Use C to construct Str
    for C ∈ ContainSet_Str do
        if C is Empty OR C ∈ Str then
            continue
        Str ← Str APPEND C
    return Str
```

7.3.3 Repairing Strategy using Constraint Evaluation

The patching is evaluated in two ways, static and dynamic. We evaluated those conditions which can be evaluated safely during compile time. Such constraints have constants like `if(s.length<10)`. We looked for particular constraints based on our storage specification

Chapter 8

Repairing Strategy : Constraint Automata

8.1 General Structure

Constraint automata is a formalism to describe the behavior and possible data flow in coordination models. Mostly used for model checking. We have used it for the purpose of program repairing technique. Here we define the finite state automata as follows :

$$(Q, \Sigma, \delta, q_0, F)$$

- Q: set of state where $|Q| = 2$, *legal state*(init) and *illegal state* (error).
- Σ: symbols, invariants based on exception type.
- δ: transition function. $init \rightarrow init$ is safe transition and $init \rightarrow error$ is the invariant violation.
- q_0: starting state, here $q_0 = init$.
- F: end state, here it same as q_0.

Figure 8.1: Constraint automata general model

According to the Figure 8.1, the repairing mechanism will only trigger when we have a transition from init state to error state due to invariant violation.

8.2 Patching Techniques

The patching technique is based on the exception type. We instrument the patching code in a catch block keeping the original statement encapsulated in try block.

8.2.1 Array index out of bound exception

Array index out of bound exception happen when one tries to access the array with a index which is more than the size of the array or less than zero i.e. with some negative value. We did the patching based on these two scenario.

- When the index is more than the array size, we patch it by assigning $array.length - 1$.
- When the index value is less than 0, then we patched it by assigning the ined to 0.

In the code snippet 8.1 we show such example.

Code Snippets 8.1: array index out of bound patching

```
void foo()
{
  int []arr = {1,2,3,4};
  int index = 10;
  int y = 0;
  try
  {
    //original code
```

```
    y = arr[index];
  }
  //patching instrumentation
  catch(IndexOutOfBoundException ex)
  {
    if(index > arr.length)
      y = arr[arr.length - 1];
    else
      y = a[0];
  }
}
```

8.2.2 Negative Array Size Exception

Negative array size exception occurs when one tries to create a array with a negative size. The patching is done based on data flow analysis. Suitable index size is determined by looking at the successive statement dependent on the array. To take a safe bound, we took maximum index size and set as the array size in the new array statement 8.2.

Code Snippets 8.2: arr index out of bound patching

```
void foo()
{
  int []arr = {1,2,3,4};
  int index = 10;
  int y = 0;
  try
  {
    //original code
    y = arr[index];
  }
  //patching instrumentation
  catch(IndexOutOfBoundException ex)
  {
    if(index > arr.length)
      y = arr[arr.length - 1];
    else
      y = a[0];
  }
}
```

8.2.3 Arithmetic Exception : Division-by-zero Exception

Division by zero causes arithmetic exception. There are two different cases which were considered here.

- **Case I :** The denominator is going to the taint sink but the left hand side is not going to any taint sink. Here we will not manipulate the denominator as we are not manipulating any variable which are going to any taint sink.
- **Case II :** The denominator and the left hand side, both are not going to any taint sink. So they are safe to patch.

In the code snippet 8.3, we demonstrate the patching technique with an example java code.

Code Snippets 8.3: arithmetic exception : division-by-zero patching

```
void foo()
{
  int a = 10;
  int b = 0;
        int y;
  try
  {
    //original code
    y = a/b;
  }
  //patching instrumentation
  catch(ArithmeticException ex)
  {
    //case I
    if(taintSink(b))
      y = 0;
    //case II
    else
    {
      b = 1;
      y = a/b;
    }
  }
}
```

8.2.4 Null Pointer Exception

Null pointer exception in Java is the most common runtime exception encountered. Thrown when an application attempts to use null in a case where an object is required. There exists various scenarios where null pointer exception can happen. These different scenario requires different patching techniques. Bellow we enlist all cases and corresponding patching techniques.

- **Case I** Calling the instance method of a null object.

  Patch : This is patched 8.4 by calling the constructor. In case there exists more than one constructor then we need to find most appropriate constructor. This is done by using data flow analysis in the successive statement to see which fields/methods been accessed and according to that most suitable constructor should be picked up, this will ensure safest way to deal with the later method calls/field accesses.

Code Snippets 8.4: appropriate constructor

```
class MyClass
{
 Integer field1;
 String field2;
 Double field3;

 public MyClass()
 {
  this.field1 = 1;
  this.field2 = null;
  this.field3 = null;
 }
 public MyClass(Integer field1, String field2)
 {
  this.field1 = field1;
  this.field2 = field2;
  this.field3 = null;
 }
 public MyClass(Integer field1, String field2, Double field3)
 {
  this.field1 = field1;
  this.field2 = field2;
  this.field3 = field3;
 }
```

```
 public Double getfield3()
 {
  return this.field3;
 }
}

class main
{
 Myclass mclass = null;
 Double a = null;
 try
 {
  //original code
  a = mclass.getfiled3() + 5.0;
 }
 //instrumentation
 catch(NullPointerException ex)
 {
  //choose appropriate constructor
  mlass = new MyClass(1, "a", 1.0);
  a = mclass.getfiled3();
 }
}
```

- **Case II** Possible Accessing or modifying the field of a null object.

  Patch : The patch is same as the previous one 8.4.

- **Case III** Taking the length of null as if it were an array.

  Patch : The patch 8.5 for this situation is very much similar to the negative array size exception. Here we will do a data-flow analysis to see all the successive statements where the array object has been used (read or write). For safety we will take the maximum index from those statements and reinitialize the array object with the size.

Code Snippets 8.5: array null pointer exception

```
int[] bar(int a)
{
 int []arr = new int[a];
 int []b = (a > 10) ? arr:null;
 return b;
}
```

```
void foo()
{
 int[] arr;
 int []arr = bar(5);
 try
 {
  //access or modify any field of arr
  //this will throw a null pointer exception
 }
 //instrumented code
 catch
 {
  int ARRAY_SIZE = 11;
  int []arr = new int[ARRAY_SIZE];
  //access or modify any field of arr
 }
}
```

- **Case IV** Accessing or modifying the slots of null as if it were an array. **Patch :** The patching mechanism is exactly same as before 8.5.

- **Case V** Throwing null as if it were a Throwable value.

Chapter 9

Design of the System

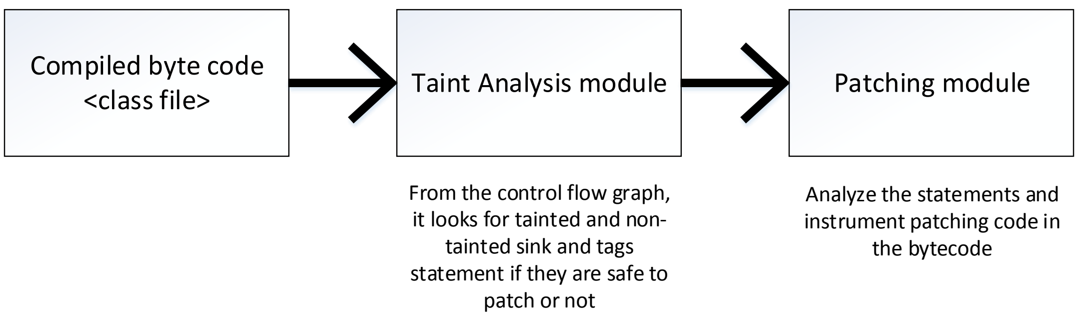


Figure 9.1: Overall Design

The overall design of the repairing framework is illustrated in Figure 9.1. The framework consists of two basic modules.

9.1 Taint analysis Module

The main purpose of the taint analysis module is to classify which of the statements are safe to patch or not. Based on the analysis result in this module, the tagged statement will be passed to the repairing module.

We have specify the list of source, sink and derivation methods in a configuration file before the analysis. The source methods includes methods which take input from user from console or web application forms like text box. The sink methods are sensitive data storage which are unsafe

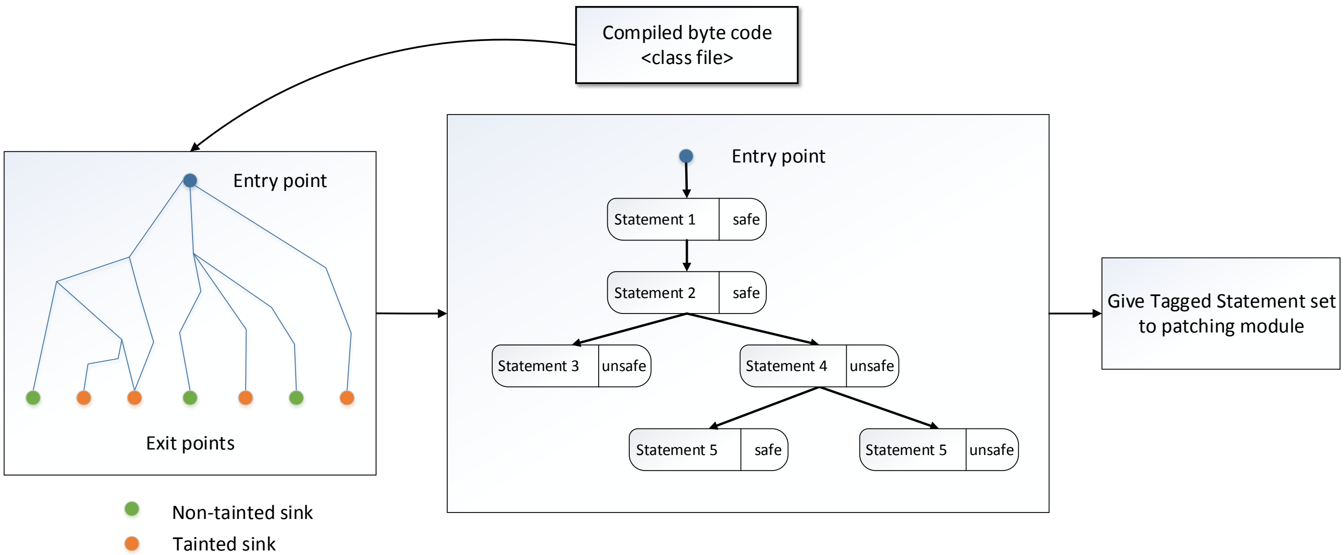


Figure 9.2: Design of the Taint Module

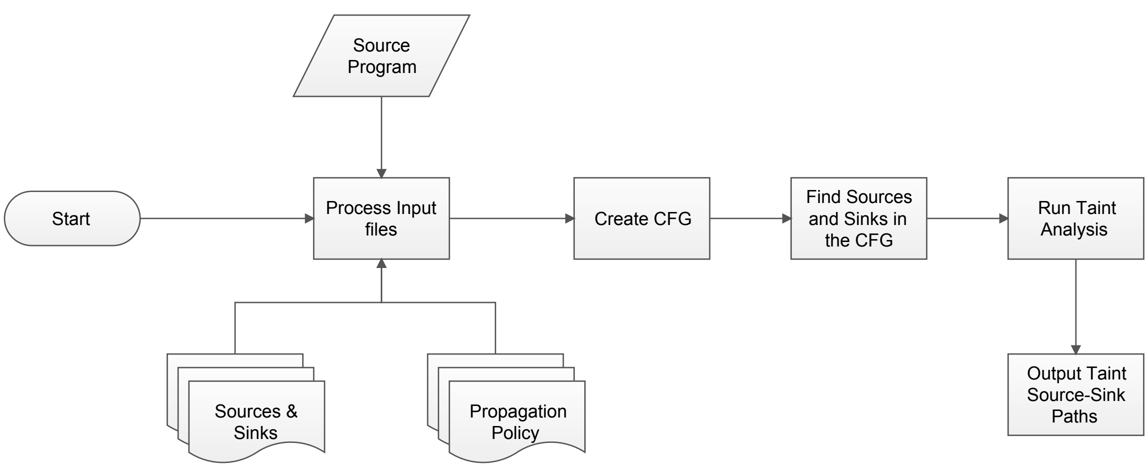


Figure 9.3: InfoFlow Framework design

to manipulate such as database, console print or methods to send a text file to printer etc. The overview of the taint analysis module is illustrated in the Figure 9.2. The input for the module is the compiled byte code intended to be repaired. Here we have generated a control flow graph (CFG) from the class file to get all the possible program paths. Here a point to be noted that any modification along the path going to the tainted sink is unsafe to patch.

9.1.1 Tainting Rules

Needs Revision

Table 9.1: Common Java libray taint source functions

Java Class	Source Method Name
java.io.InputStream	read()
java.io.BufferedReader	readLine()
java.net.URL	openConnection()
org.apache.http.HttpResponse	getEntity()
org.apache.http.util.EntityUtils	toString()
org.apache.http.util.EntityUtils	toByteArray()
org.apache.http.util.EntityUtils	getContentCharSet()
javax.servlet.http.HttpServletRequest	getParameter()
javax.servlet.ServletRequest	getParameter()
java.Util.Scanner	next()

Table 9.2: Common Java libray taint sink functions

Java Class	Sink Method Name
java.io.PrintStream	printf()
java.io.OutputStream	write()
java.io.FileOutputStream	write()
java.io.Writer	write()
java.net.Socket	connect()

We have used extended InFlow framework for the taint analysis module. The steps are

1. We defined list of source and sink tait methods listed in Table 9.1 and 9.2. We are only tainting the variables which are coming from the listed taint source methods.

2. We have also listed all taint propagation methods. The assignment (=) is the basic

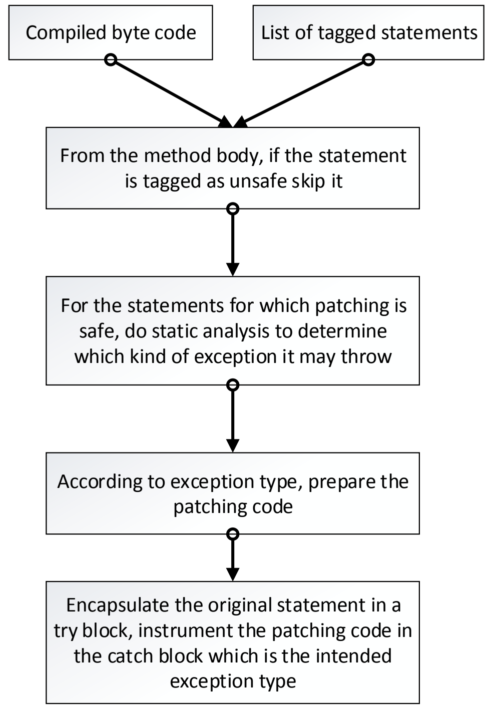


Figure 9.4: Design of the Patching Module

taint propagator. But there are other methods like *append* in *java.lang.StringBuffer* and *java.lang.StringBuilder* which are taint propagator.

3. All the variable which are referred to tainted variables/ objects or output of taint propagator over tainted variable/objects are also considered as tainted.

4. For all the program patch we see if such tainted variables are reaching the tainted sink or not. If they are reaching to some tainted sink then all the statements along that particular program path to which the tainted variables are assigned are marked as unsafe otherwise safe.

9.2 Repairing Module

The repairing module is consisted of three phases. All these three phases requires three sequential passes over the input bytecodes to produce the final patched result.

9.2.1 Method Shilding

When we are shielding a method, we also looked to the calling context of that particular method. The method can be called from a path which leads to some tainted sink and it can also be called from such path which does not contain any taint sink. In such cases, we have taken special care about the callee. The path to the tainted sink should not call a patched method as it can influence data which are leaving the system. So, we also maintained two different version of the method and instrument the calling site so that appropriate method is called.

Code Snippets 9.1: Same method calling in different scenario

```
int bar(int a, int b)
{
 return a/b;
}
void foo()
{
 int a = 10, b = 0, c = 15;
 int out = bar(a, b);
 TaintSink(out);
 int out1 = bar(c, b);
 NonTaintSink(out1);
}
```

Code Snippets 9.2: Mehod name modification for different calling context

```
int bar(int a, int b)
{
 return a/b;
}

int bar_untainted_fa844d57(int a, int b)
{
 int out;
 try
 {
  out = a/b;
 }
 catch(ArithmeticException ex)
 {
  b = 1;
  out = a/b;
```

```
 }
 return out;
}

void foo()
{
 int a = 10, b = 0, c = 15;

 //no modification in the call where the result can go to a tainted sink method
 int out = bar(a, b);
 TaintSink(out);

 //Modify the method call to the shielded method as the result is not going to
 //any tainted sink method
 int out1 = bar_untainted_fa844d57(c, b);
 NonTaintSink(out1);
}
```

In the Listing 9.1 and 9.2 we have defined an example code snippet of the original code and the patched code where we have renamed the method *bar* to *bar_untainted_fa844d57* before instrumenting any patching code in it. The variable *out* goes to a tainted sink while *out1* does not. So the we have done modification in the line where *out1* is defined. As *out* is going to a tainted sink method, we did not do any modification to it.

Chapter 10

BenchMark Result

Chapter 11

Conclusion and Future

Bibliography

[1] Demsky, B., Ernst, M. D., Guo, P. J., McCamant, S., Perkins, J. H., and Rinard, M. C. Inference and enforcement of data structure consistency specifications. In *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2006, Portland, Maine, USA, July 17-20, 2006* (2006), L. L. Pollock and M. Pezzè, Eds., ACM, pp. 233–244.

[2] Demsky, B., and Rinard, M. Automatic data structure repair for self-healing systems. In *In Proceedings of the 1 st Workshop on Algorithms and Architectures for Self-Managing Systems* (2003).

[3] Demsky, B., and Rinard, M. C. Automatic detection and repair of errors in data structures. In *Proceedings of the 2003 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications, OOPSLA 2003, October 26-30, 2003, Anaheim, CA, USA* (2003), R. Crocker and G. L. S. Jr., Eds., ACM, pp. 78–95.

[4] Demsky, B., and Rinard, M. C. Static specification analysis for termination of specification-based data structure repair. In *14th International Symposium on Software Reliability Engineering (ISSRE 2003), 17-20 November 2003, Denver, CO, USA* (2003), IEEE Computer Society, pp. 71–84.

[5] Demsky, B., and Rinard, M. C. Data structure repair using goal-directed reasoning. In *27th International Conference on Software Engineering (ICSE 2005), 15-21 May 2005, St. Louis, Missouri, USA* (2005), G. Roman, W. G. Griswold, and B. Nuseibeh, Eds., ACM, pp. 176–185.

[6] Elkarablieh, B., Khurshid, S., Vu, D., and McKinley, K. S. Starc: static analysis for efficient repair of complex data. In *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2007, October 21-25, 2007, Montreal, Quebec, Canada* [6], pp. 387–404.

[7] Ernst, M. D., Perkins, J. H., Guo, P. J., McCamant, S., Pacheco, C., Tschantz, M. S., and Xiao, C. The daikon system for dynamic detection of likely invariants. *Sci. Comput. Program. 69*, 1-3 (2007), 35–45.

[8] Forrest, S., Nguyen, T., Weimer, W., and Goues, C. L. A genetic programming approach to automated software repair. In *Genetic and Evolutionary Computation Conference, GECCO 2009, Proceedings, Montreal, Québec, Canada, July 8-12, 2009* [8], pp. 947–954.

[9] IEEE, S. IEEE Spectrum - lost radio contact leaves pilots on their own communications error wreaks havoc in the los angeles air control system. `http://spectrum.ieee.org/aerospace/aviation/lost-radio-contact-leaves-pilots-on-their-own`, 2004.

[10] Long, F., Sidiroglou-Douskos, S., and Rinard, M. C. Automatic runtime error repair and containment via recovery shepherding. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014* (2014), M. F. P. O'Boyle and K. Pingali, Eds., ACM, p. 26.

[11] Oracle. Oracle Java Documentation error (java platform se 6). `http://docs.oracle.com/javase/6/docs/api/java/lang/Error.html`.

[12] Oracle. Oracle Java Documentation run timeexception (java platform se 7). `http://docs.oracle.com/javase/7/docs/api/java/lang/RuntimeException.html`.

[13] Oracle. Oracle Java Documentation what is an exception? `http://docs.oracle.com/javase/tutorial/essential/exceptions/definition.html`.

[14] Perkins, J. H., Kim, S., Larsen, S., Amarasinghe, S. P., Bachrach, J., Carbin, M., Pacheco, C., Sherwood, F., Sidiroglou, S., Sullivan, G., Wong, W., Zibin, Y., Ernst, M. D., and Rinard, M. C. Automatically patching errors in deployed software. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles 2009, SOSP 2009, Big Sky, Montana, USA, October 11-14, 2009* (2009), J. N. Matthews and T. E. Anderson, Eds., ACM, pp. 87–102.

[15] Weimer, W., Forrest, S., Goues, C. L., and Nguyen, T. Automatic program repair with evolutionary computation. *Commun. ACM 53*, 5 (2010), 109–116.

[16] Wired. Sunk by windows nt. `http://archive.wired.com/science/discoveries/news/1998/07/13987`.

Appendix