Clotho: Saving Programs from Malformed Strings and Incorrect String-handling

Student Name: Aritra Dhar

IIIT-D-MTech-CS-IS-12-004
Nov 28, 2014

Indraprastha Institute of Information Technology
New Delhi

Thesis Committee
Dr. Rahul Purandare (Advisor)
Dr. Mohan Dhawan (External reviewer)
Dr. Sambuddho Chakravarty (Internal reviewer)

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

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

Programs are susceptible to malformed data coming from untrusted sources. Occasionally the programming logic or constructs used are inappropriate to handle all types of constraints that are imposed by legal and well-formed data. As a result programs produce unexpected results or even worse, they may crash. Program behavior in both of these cases would be highly undesirable.

In this thesis work, we present a novel hybrid approach that saves programs from crashing when the failures originate from malformed strings or inappropriate handling of strings. Our approach statically analyses a program to identify statements that are vulnerable to failures related to associated string data. It then generates patches that are likely to satisfy constraints on the data, and in case of failures produce program behavior which would be close to the expected. The precision of the patches is improved with the help of a dynamic analysis. The patches are activated only after a failure is detected, and the technique incurs no runtime overhead during normal course of execution, and negligible overhead in case of failures.

We have experimented with JAVA `String` API, and applied CLOTHO to several hugely popular open-source libraries to patch 30 bugs, several of them rated either critical or major. Our evaluation shows that CLOTHO is both practical and effective. The comparison of the patches generated by our technique with the actual patches developed by the programmers in the later versions shows that they are semantically similar.

Acknowledgments

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

Sunday 16th November, 2014

Dedicated to,

Ma, Baba and Dida

Contents

1 Introduction **2**

2 Motivation and Challenges **3**

2.1 Historical Context 3

2.2 Data from Stack Overflow Posts 4

2.3 Major Challenges 5

3 Related Works **6**

3.1 Recent Works on Data Structure Repairing 6

3.2 Works on Software Patching 7

3.3 Genetic Programming, Evolutionary Computation 7

4 Problem Formulation **8**

4.1 Runtime Exceptions 8

5 RepairingStrategy : Taint Analysis **9**

5.1 Taint analysis : Definition 9

5.2 Taint analysis : Taint Propagation 10

5.3 Taint Analysis : Relevance with Repairing Effort 10

6 Repairing Strategy : Exception Type **12**

6.1 Static Analysis . . . 14

6.2 Data set for Successful Program Runs . . . 15

6.3 Matrices . . . 15

6.4 Instrumenting Patching . . . 15

6.4.1 Determine Exception Type . . . 16

6.4.2 Determine Optimal Code Slice . . . 16

6.5 Variable Tracking and Monitoring . . . 18

7 Repairing Strategy : Bounded Forward and Backward Analysis **19**

7.1 Example Scenario . . . 19

7.2 Flow Functions . . . 20

7.2.1 Bounded Forward Analysis . . . 20

7.3 Constraint Satisfaction . . . 21

7.3.1 Constraint Storage . . . 22

7.3.2 Constraint Evaluation Strategy . . . 23

7.3.3 Repairing Strategy using Constraint Evaluation . . . 23

8 Repairing Strategy : Constraint Automata **24**

8.1 General Structure . . . 24

8.2 Patching Techniques . . . 25

8.2.1 Array index out of bound exception . . . 25

8.2.2 Negative Array Size Exception . . . 26

8.2.3 Arithmetic Exception : Division-by-zero Exception . . . 27

8.2.4 Null Pointer Exception . . . 28

9 Design of the System **31**

9.1 Taint analysis Module 31

9.1.1 Tainting Rules 33

9.2 Repairing Module 34

9.2.1 Method Shilding 35

10 Implementation Details **37**

10.1 Taint Analysis 37

10.2 Call graph analysis 38

10.3 Constraint Analysis 38

10.4 String Repairing Phase 38

10.4.1 Detecting Potential Point of Failure 39

10.4.2 Catch Block Instrumentation 39

10.5 Optimizations 39

11 Evaluation **41**

11.1 Accuracy 42

11.2 Overhead 45

11.3 Case studies 46

12 Conclusion and Future **49**

List of Figures

4.1 array index out of bound formulated as FSM 8

5.1 A simplified diagram indicating taint problem 10

6.1 Data dependency graph of the variables in code snippet 6.1 14

6.2 Indexed global variables and method arguments successful runs 15

7.1 Dataflow diagram with in, out set in forward analysis 21

7.2 String constraints storage format 22

8.1 Constraint automata general model 25

9.1 Overall Design 31

9.2 Design of the Taint Module 32

9.3 InfoFlow Framework design 32

9.4 Design of the Patching Module 34

11.1 CLOTHO evaluation. 45

List of Tables

2.1 Most frequent java runtime exceptions from stack overflow data 4

9.1 Common Java libray taint source functions 33

9.2 Common Java libray taint sink functions 33

11.1 CLOTHO's accuracy results when applied to 30 bugs in popular open-source libraries. 48

11.2 Precision results for taint analysis. 48

List of Code Snippets

6.1 Java code which may throws runtime exceptions 12

6.2 Patching code slice based on exception type 16

7.1 Dataflow analysis 19

7.2 Better patching mechanism with constraint satisfaction 21

8.1 array index out of bound patching 25

8.2 arr index out of bound patching 26

8.3 arithmetic exception : division-by-zero patching 27

8.4 appropriate constructor 28

8.5 array null pointer exception 29

9.1 Same method calling in different scenario 35

9.2 Mehod name modification for different calling context 35

Chapter 1

Introduction

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) [35].
- 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 [14].
- 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 [**?**], [5], [**?**], [**?**], [**?**] 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 [**?**], 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 [**?**] 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 [**?**], 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 [9]. They mostly patched security vulnerabilities by some candidate repair patches.

Fan Lon et al in their paper [**?**] 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. [**?**] and Westley Weimer et al [33] 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

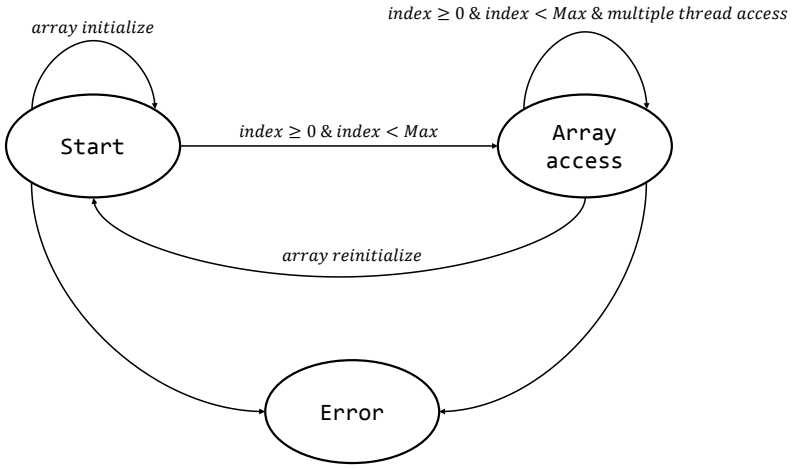


Figure 4.1: array index out of bound formulated as FSM

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.

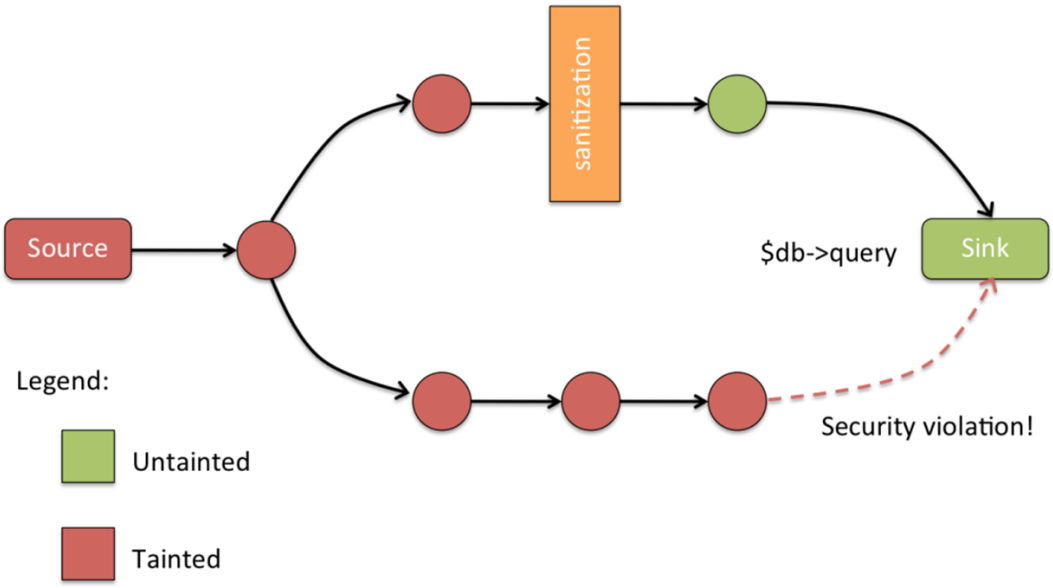


Figure 5.1: A simplified diagram indicating taint problem

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

```
1
2  public class TestClass
3  {
4          private int[] arr1;
5          private int[] arr2;
6          private int[] arr3;
7
8          public TestClass(int[] arr1, int[] arr2, int[] arr3)
9          {
10                 this.arr1 = arr1;
11                 this.arr2 = arr2;
12                 this.arr3 = arr3;
13         }
14         public int[] fun(int a, int b, int c, int d)
15         {
16                 int temp0 = a + b;
17                 int temp1 = c * d;
```

```
18                int temp2 = temp0 - temp1;
19                //array index out of bound, negative index
20                int temp3 = this.arr1[temp0];
21                //array index out of bound, negative index
22                int temp4 = this.arr2[temp1];
23                //array index out of bound, negative index
24                int temp5 = this.arr3[temp3];
25                int temp6 = temp4 + temp5;
26                int temp7 = temp6 - temp3;
27                //array index out of bound, negative index, divide by zero
28                this.arr1[temp6] = temp7/(d-a);
29                //array index out of bound, negative index, divide by zero
30                this.arr2[temp7] = temp7/temp4;
31                if(arr2[temp1] ! = arr3[temp7])
32                       return arr1;
33                else
34                       return null;
35        }
36 }
37 public class MainClass
38 {
39        public void main(String[] a)
40        {
41                int[] arr1 = {1,2,3,4};
42                int[] arr2 = {1,2,3,4};
43                int[] arr3 = {1,2,3,4};
44                TestClass TC = new TestClass(arr1, arr2, arr3);
45                int[] res = TC.fun(2,4,3,4);
46                //Null pointer exception
47                System.out.print("Result : "+res[2]);
48        }
49 }
```

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

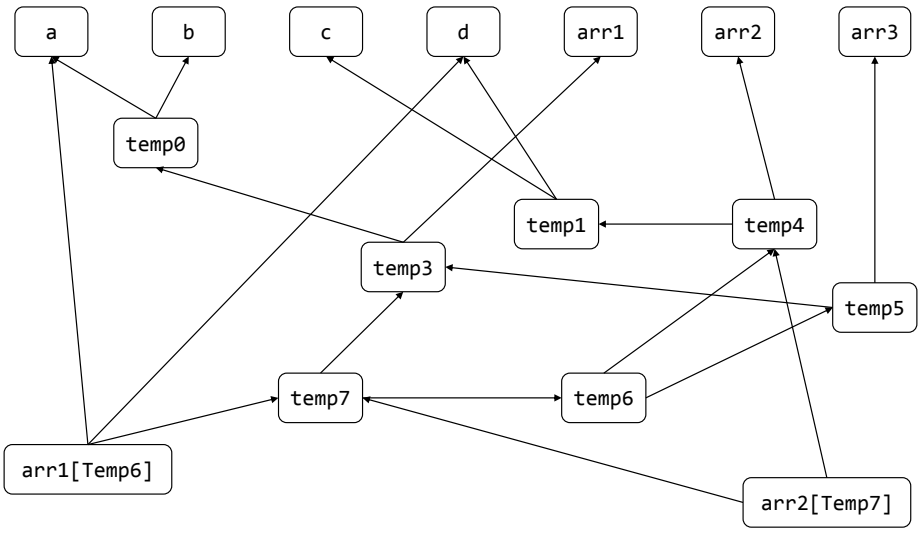


Figure 6.1: Data dependency graph of the variables in code snippet 6.1

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.

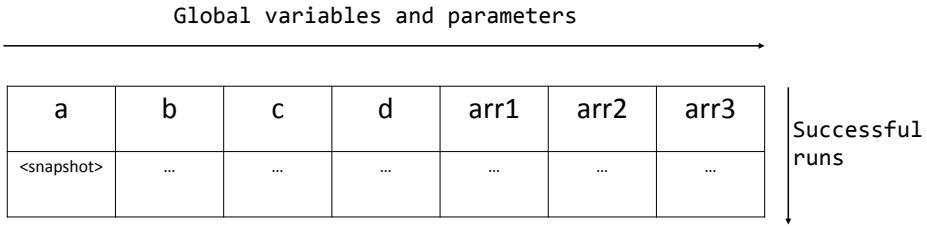


Figure 6.2: Indexed global variables and method arguments successful runs

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

```
1
2  public class TestClass
3  {
4          private int[] arr1;
5          private int[] arr2;
6          private int[] arr3;
7
8          public TestClass(int[] arr1, int[] arr2, int[] arr3)
9          {
10                 this.arr1 = arr1;
11                 this.arr2 = arr2;
12                 this.arr3 = arr3;
13         }
14         public int[] fun(int a, int b, int c, int d)
15         {
16                 try
17                 {
18                   int temp0 = a + b;
19                   int temp1 = c * d;
20                   int temp2 = temp0 - temp1;
21                   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

```
1 void bar()
2 {
3   foo("fname:lname");
4 }
5
6 String foo(String s)
7 {
```

```
8   int a = s.indexof(":");
9   int b = s.indexOf("&");
10  int c = s.indexOf("#");
11  int d = 0;
12  if(c>0)
13  {
14    d = 1;
15  }
16  return s.substring(a,b);
17 }
```

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

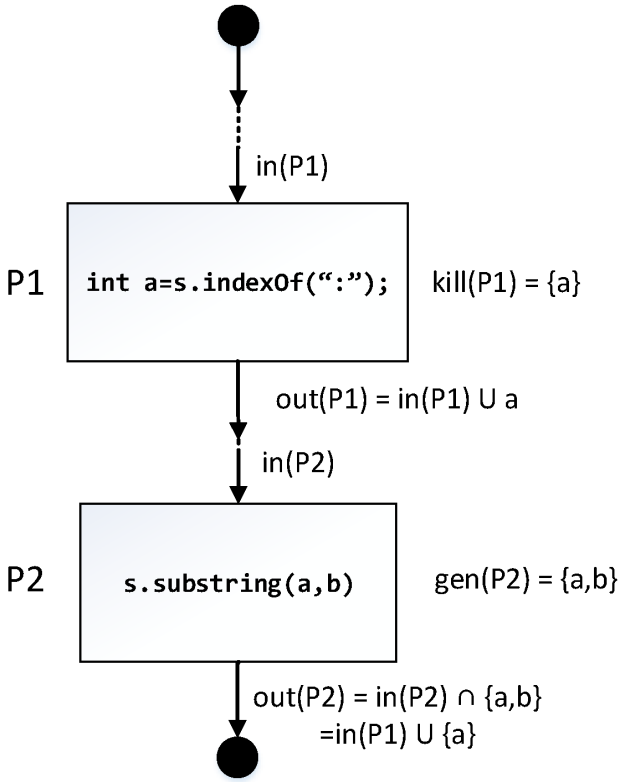


Figure 7.1: Dataflow diagram with in, out set in forward analysis

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 $Def(P_i) = \{\texttt{a}\}$ and as `b, c, d` are used, $Used(P_i) = \{\texttt{b, c, d}\}$

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

```
1
2 void foo(String s, int i, int j)
3 {
```

```
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
  $CS_{Str} \longleftarrow$ Get the constrint set for Str
  $MinLength \longleftarrow CS_{Str}[0]$
  $MaxLength \longleftarrow CS_{Str}[1]$
  $PrefixSet_{Str} \longleftarrow CS_{Str}[2 \rightarrow MaxLength + 1]$
  $ContainSet_{Str} \longleftarrow CS_{Str}[MaxLength + 2 \rightarrow 2 * MaxLength + 1]$
  for $C \in PrefixSet_{Str}$ **do**
    if C *is Empty* **then**
      continue
    $PrefixLength \longleftarrow$ **LENGTH OF** C
    if $PrefixLength$ *is Maximum* $\in PrefixSet_{Str}$ **then**
      Use C to construct Str
  for $C \in ContainSet_{Str}$ **do**
    if C *is Empty* **OR** $C \in Str$ **then**
      continue
    $Str \leftarrow Str$ **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

```
1 void foo()
2 {
3   int []arr = {1,2,3,4};
4   int index = 10;
5   int y = 0;
6   try
7   {
8     //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

```
1  void foo()
2  {
3    int a = 10;
4    int b = 0;
5         int y;
6    try
7    {
8      //original code
9      y = a/b;
10   }
11   //patching instrumentation
12   catch(ArithmeticException ex)
13   {
14     //case I
15     if(taintSink(b))
16       y = 0;
17     //case II
18     else
19     {
20       b = 1;
21       y = a/b;
22     }
23   }
24 }
```

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;
}
```

```
7  void foo()
8  {
9   int[] arr;
10  int []arr = bar(5);
11  try
12  {
13   //access or modify any field of arr
14   //this will throw a null pointer exception
15  }
16  //instrumented code
17  catch
18  {
19   int ARRAY_SIZE = 11;
20   int []arr = new int[ARRAY_SIZE];
21   //access or modify any field of arr
22  }
23 }
```

- **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

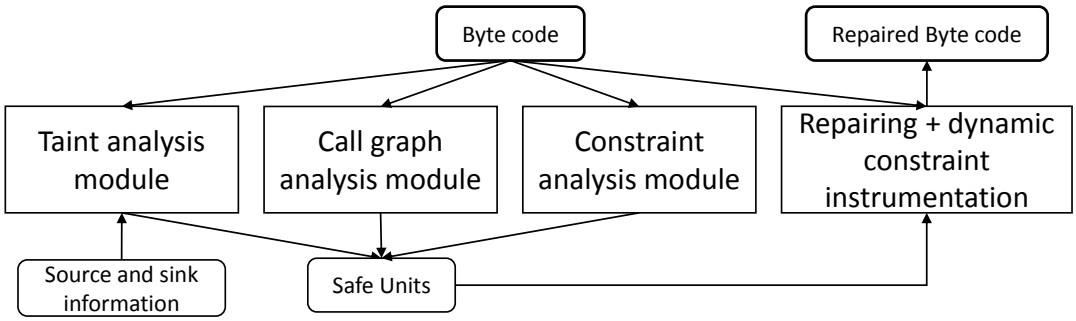


Figure 9.1: Overall Design

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

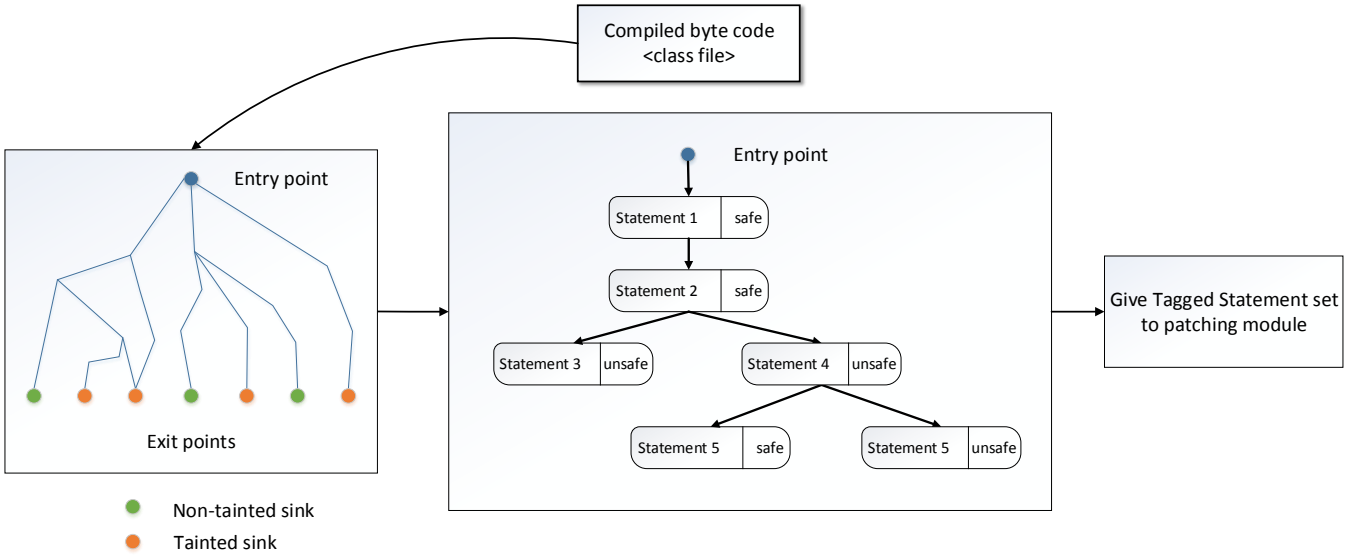


Figure 9.2: Design of the Taint Module

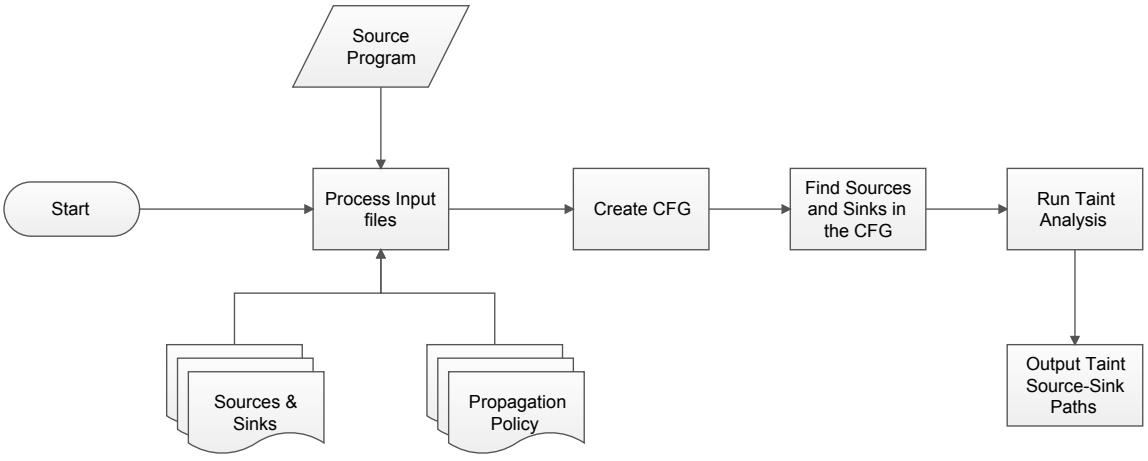


Figure 9.3: InfoFlow Framework design

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

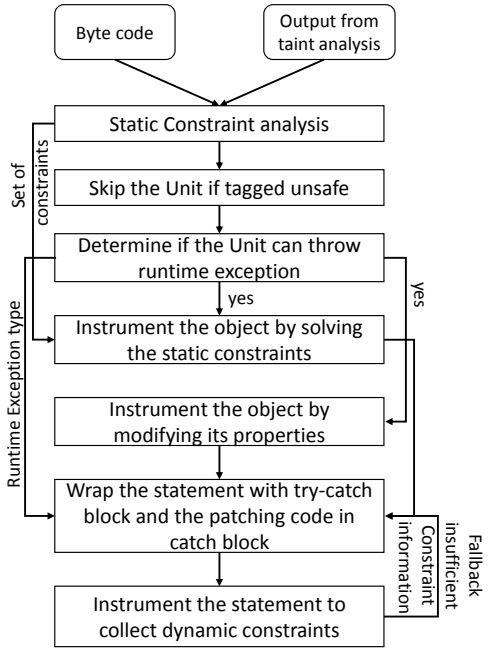


Figure 9.4: Design of the Patching Module

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

```
1  int bar(int a, int b)
2  {
3   return a/b;
4  }
5  void foo()
6  {
7   int a = 10, b = 0, c = 15;
8   int out = bar(a, b);
9   TaintSink(out);
10  int out1 = bar(c, b);
11  NonTaintSink(out1);
12 }
```

Code Snippets 9.2: Mehod name modification for different calling context

```
1  int bar(int a, int b)
2  {
3   return a/b;
4  }
5
6  int bar_untainted_fa844d57(int a, int b)
7  {
8   int out;
9   try
10  {
11   out = a/b;
12  }
13  catch(ArithmeticException ex)
14  {
15   b = 1;
16   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

Implementation Details

We implemented a prototype of CLOTHO as described in § **??** for repairing runtime exceptions originating from unhandled JAVA `String` APIs. Our end-to-end toolchain is completely automated and was written in $\sim$12.7K lines of JAVA. We leveraged the SOOT [29] framework for bytecode analysis and instrumentation, and INFOFLOW [30] for static taint analysis.

We now briefly describe a few salient features of our implementation, which is also available for download at `http://goo.gl/d1zcXD`.

10.1 Taint Analysis

INFOFLOW performs its taint propagation over `Units`, which are SOOT's intermediate representation of the JAVA source code. We extended the INFOFLOW framework to a) enable seamless coupling with SOOT, and b) determine whether it is safe to patch a given SOOT `Unit`. Specifically, we added a mapping that retrieves `Units` for statements to be patched given a specified method signature. This is relevant since the same statement, say `int x = 1;` has the exact same representation even if it appears more than once in a same method. We also added a utility method to determine if a `Unit` must be patched if it lies along the path between a source and sink in the call graph (as generated by SOOT).

10.2 Call graph analysis

CLOTHO leverages SOOT generated call graph to determine both inter- and intra-method checked runtime exceptions (recall § **??**). SOOT uses the `Trap` class to manage exception handling for both classes of exceptions discussed above. Each `Trap` object has start, end and handler unit. We tagged every `Unit` in a `HashMap` if it belonged to an existing `Trap`, so as to exclude it from instrumentation during the repairing phase.

10.3 Constraint Analysis

CLOTHO makes a forward pass over the `Units` identified by the taint analysis and other program analyses in the first phase to gather constrains over string literals of interest (recall § **??**), and builds a `HashMap` of `ConstraintDataType`, a custom data type to store and evaluate these constrains. Specifically, each `ConstraintDataType` entry stores four key parameters—the permissible prefixes, substrings, minimum and maximum length—that specify constraints corresponding to a `String` literal.

Constraint evaluation over these `ConstraintDataType` entries is done as discussed earlier in Algorithm 1. However, if the gathered constraints can not be satisfied statically, e.g., `if(str.contains(userInput()))`, CLOTHO instruments the bytecode before the conditional statement with a static invocation to i) populate the corresponding `ConstraintDataType` entry, and ii) recompute the permissible values of the string object with already existing constraints (see Code snippet **??**).

10.4 String Repairing Phase

The string repairing phase is divided into two sub-phases.

10.4.1 Detecting Potential Point of Failure

We have used specification from JAVA SE official documentation and list all the methods which throws runtime exception. We do forward pass to see if there is any invocation of such methods and if we find any we then cross check it with the results we got from the taint analysis. We also see if there is already some exception handling mechanism provided by the developer using the technique described in Section **??**. We detected such method calls and wrap them in try-catch block. In the catch block we place the appropriate exception type as provided by JAVA SE API documentation.

10.4.2 Catch Block Instrumentation

In this phase we instrument appropiate patching codes inside the catch block. We used the static constraint evaluation of Section **??** to statically evaluate the string. In cases there are more constraints which can't be solved statically, it would instrument necessary method call so that the constraints would populate and get solved in runtime **??**. In case there is no constraint, we repair the string in the method calls like `substrring`, `subSequence`, `charAt` etc. which are dependent on the index arguments. In those cases we used Algorithm 2 to repair them.

Algorithm 2: String patching based on parameters passed

Data: String object Str and index set IS which contains i or i, j.
Result: Repaired index set containing Ri or Ri, Rj based on input IS
begin
  $Length \longleftarrow$ length of Str
  if $Length == 0$ **then**
    $Ri, Rj \longleftarrow 0$
  else
    if i>j **then**
      $Ri \longleftarrow j - 1$;
    if i>$Lengrh$ **OR** j>$Lengrh$ **then**
      $Ri \longleftarrow Length - 1$ or $Rj \longleftarrow Length - 1$ based on condition
    if i<0 **OR** j<0h **then**
      $Ri \longleftarrow 0$ or $Rj \longleftarrow 0$ based on condition

10.5 Optimizations

CLOTHO performs few other optimizations to improve the precision and quality of the patches.

Minimize constraint analysis:

CLOTHO collects constraints only for those string literals that may be involved in a runtime exception. For example, if a string object does not involve API methods that can throw runtime exception, then it is not required to collect and evaluate constraints on them. This significantly reduces the number of statements analyzed for instrumentation.

Minimize patch instrumentation:

CLOTHO makes a forward pass over all bytecodes to determine if a specific string object is modified after it has been patched. If the object is not modified then no further patching of statements capable of throwing `NullPointerException` exceptions is required, since the constraint would have been satisfied in the beginning and it would be valid as long as the variable is not changed. Similarly, when the API usage is same and none of the method parameters are changed, no further patching would be required. This reduces the total number of possible instrumentations required.

Chapter 11

Evaluation

We now present an evaluation of CLOTHO. In § 11.1, we evaluate CLOTHO's effectiveness by measuring the quality of patches and related instrumentation required. We also measure how the several optimizations described in § **??** affect the patches generated by CLOTHO. In § 11.2, we measure the relative performance and resource penalties incurred with CLOTHO. In § 11.3, we describe our experiences with some of the major bugs from our data set.

Data Set. We mined bug repositories of several open-source JAVA-based applications and selected 30 bugs, majority of them being rated either major, critical or blocking. These bugs involved usage of 64+ different APIs from JAVA's `String`, `StringBuffer`, `StringBuilder`, and Apache `StringUtils` and Google Guava `StringUtils` classes.

Experimental Setup. All our experiments were performed on a laptop with 2.9 GHz dual core Intel i5 CPU, and 8 GB of RAM, and running Microsoft Windows 8.1. We used JDK v1.7 running with 2 GB of allocated heap space. All bug reproduction was done on Eclipse Juno IDE. We used SOOT v2.5.0 for bytecode analysis and instrumentation, and INFOFLOW snapshot from May'14 for static taint analysis.

11.1 Accuracy

We evaluate the precision of the patch and the effectiveness of CLOTHO based on several metrics as described below.

- **Effectiveness of the patch**: Precision and effectiveness of a patch is governed by the similarity between a CLOTHO generated patch and the developer's fix for the same bug. We define **Patch Quality Index (PQI)** as a measure of the effectiveness of the patch.

$$PQI = \frac{\#\ Constraints_{Similar}}{\#\ Constraints_{Developer}} * \frac{\#\ LOC_{Developer}}{\#\ LOC_{\text{CLOTHO}}} * \frac{Output_{\text{CLOTHO}}}{Output_{Developer}}$$

  Specifically, PQI compares the similarities in constraints and source line of code in CLOTHO's patch against the developer's version, as well as the actual output generated from both the patches, thereby considering both the logic and the technique to construct an effective patch. A higher value of PQI is preferred. Thus, if CLOTHO's patch has fewer constraints, or has more lines of code, or has fewer similarities in the output, the PQI will be lesser.

  Determining PQI is a three step process. First, we visually compared CLOTHO's patch with the developer's version and count the exact similar constraints observed in both patches to determine the closeness in terms of the set of constraints. Second, we disassembled the developer's patch and compared the count of bytecodes generated using CLOTHO's patch. Third, we observed the actual output of using the CLOTHO patched class files against the developer provided patch in a later version of the same library. In case the output is primitive or strings, an exact match is considered successful, else we iterate over the properties of the complex object to determine number of exact matches in the two outputs. In case of exception(s), we select the similarity ratio to be 0.5.

  Table 11.1 lists 30 real-world bugs mined from bug repositories of popular open-source libraries. We wrote a driver program to recreate the bug, and then applied CLOTHO to patch it. We observed that in all cases, CLOTHO successfully patched the offending class file in the concerned library.

• We observed that PQI for CLOTHO generated patches was high for most of the bugs, which indicates the effectiveness of the patches. Specifically, for 23 out of 30, i.e., for more than 75% cases, PQI for CLOTHO-generated patches was within 7% of the developer's fix. Note that there was one instance where the concerned bug [11] was still unpatched. In such cases, we looked for the potential or suggested patches in comments and discussion forums, and compared CLOTHO's patch with them.

• We observed that PQI for bug in Hama 0.2.0 [10] was as high as 1.38. We visually inspected both the CLOTHO-generated patch and the developer's fix and observed that CLOTHO's patch was considerably smaller. Further, we noticed that the developer's patch had i) multiple assertion blocks to make sure the error condition can be avoided, and ii) several additional conditional checks to avoid corner cases. Moreover, the developer's patch also had a completely different implementation of the method on which the bug was reported. Thus, the developer's fix was significantly larger than CLOTHO's patch.

• We noticed low PQI (between 0.5 and 0.79) for four of the patched libraries, which shows that the quality of the CLOTHO-generated patch was significantly lower than the developer's fix. A major reason for this drop in PQI is the presence of cascaded exceptions observed in the patched versions of these libraries upon execution. We note that CLOTHO in its present shape does not guarantee that the patch will never raise cascaded exceptions. Thus, in these four cases CLOTHO's patches were of lower quality then the original ones. However, we also note that all the cascaded exceptions were in fact checked exceptions, and it is expected that the application developers would handle them appropriately.

• Note that taint analysis only works when sources and sinks are defined. Since our library benchmarks have no notion of sources or sinks, CLOTHO's bytecode analysis of the libraries did not involve the taint analysis phase. However, even without taint analysis, CLOTHO's patches were of high quality, as demonstrated by the high PQI.

- **Precision of taint analysis**: CLOTHO leverages off-the-shelf tools (INFOFLOW) to perform the taint analysis. We measure precision of our choice of tool by measuring the number of statements in the analyzed code that are deemed unsafe to patch. Since

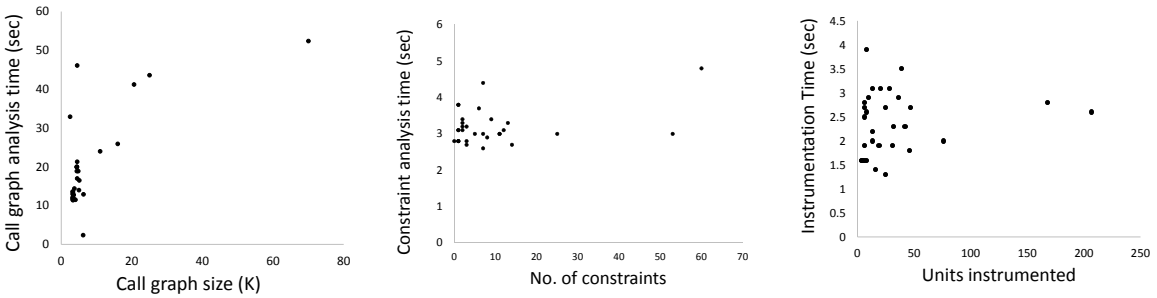
we could not measure the precision of our taint analysis on the library benchmarks (as discussed earlier), we select 3 diverse applications and apply CLOTHO in its entirety to obtain a measure of the precision of the taint analysis. Specifically, for each application we provided a set of sources, sinks and taint propagators to INFOFLOW, which listed the total number of tainted paths, i.e., paths from a sensitive source to a sink and thus must not be patched. Table 11.2 lists the results. We observe that the total number of tainted paths is less than 12% across the applications.

Threats to validity. Note that CLOTHO is dependent on INFOFLOW for achieving precision about the points of instrumentation. However, INFOFLOW currently has a major limitation—it does not support taint analysis for multi-threaded programs. Moreover, since it is still under active development, we observed that when applied to certain applications, INFOFLOW consumed inordinate amounts of memory and crashed. Thus, CLOTHO's precision is limited by the accuracy of its dependencies.

- **Already handled exceptions**: CLOTHO analyzes the call graph to determine if a potential runtime exception throwing statement is handled higher up in the call chain or in the same method. In such cases CLOTHO must abort the patching effort considering that the exception is caught with exact exception type or its base type. This is required else patching will disrupt the normal control flow of the program.

  We measure the extent of this optimization, which prevents disruption of the control flow using the **Flow Consistency Index (FCI)** that is calculated as $FCI = n$, where n is the number of exceptions in the application that must be ignored CLOTHO for forced patching of the bug. Note that $FCI \geq 0$, and a lower value of FCI is desirable. We observe that patching four bugs required CLOTHO to ignore at most one exception; rest required no changes.

- **Cascaded exceptions**: A cascaded exception arises if the CLOTHO-generated patch creates objects that when used as inputs to other JAVA APIs result in further exceptions. CLOTHO is prone to cascading exceptions because of the limitation of its intra-procedural analysis and a simple constraint evaluation mechanism. However, CLOTHO's constraint



(a) Variation in call graph analysis time with size of call graph.

(b) Variation in static constraint analysis time with number of constraints.

(c) Variation in instrumentation time with `Units` to be instrumented.

Figure 11.1: CLOTHO evaluation.

solver is pluggable and a more sophisticated third party solvers can easily be integrated. Specifically, cascaded exceptions may arise if the patch generates `String` objects that represent a malformed string. Further, if we keep the optimization in § 10.5, then cascaded failures may occur even for subsequent `String` APIs handling the malformed string following the point of patching. If the optimization is turned off, CLOTHO will automatically patch all relevant `String` APIs and thus handle all cascaded failures involving malformed `String` objects.

We observe that two benchmarks throw cascaded exceptions even after being repaired. The cascading was one level deep and triggered exception in another non-String code (and thus unpatched), which caused the application to crash.

Detailed evaluation for each of the bugs in our data set is available at `http://goo.gl/d1zcXD`.

11.2 Overhead

We measure the overhead of CLOTHO across different metrics identified below.

- **Execution overhead**: We randomly selected and patched 5 libraries (Apache Tapestry, Apache Wicket, Eclipse AspectJ Weaver, Hive and Nutch) from Table 11.1 to determine the execution overhead of the patched class files. We observed that CLOTHO reports an average overhead of $\sim$2.32μs per call across the 5 benchmarks for 50K runs of the patched

functionality in both the developer's version and CLOTHO's patched library. The maximum absolute overhead was observed for Hive at $\sim$3.96μs per call. The above overhead is imperceptible at human response time scales.

- **Call graph**: The size of the call graph directly governs the time and memory consumption for CLOTHO. Figure 11.1a shows the results for the benchmarks analyzed from our data set. The overall analysis time was under a minute for all the benchmarks. We observed that even for a call graph of $\sim 70K$ nodes (for `Wicket`), CLOTHO required just 52.4s and 210MB memory.

- **Constraint set**: CLOTHO performs an exhaustive multi-pass analysis to gather and evaluate the set of constraints for generating patches. A higher number of constraints and their complexity increases the duration of CLOTHO's analysis. Figure 11.1b compares the time required for static constraint collection and evaluation with an increasing number of constraints for the benchmarks used in our data set. We observe that across all the benchmarks used, CLOTHO required at most $\sim$5s for collecting and evaluating the constraints.

- **Instrumentation overhead**: CLOTHO performs bytecode instrumentation for actual patching. Figure 11.1c shows the variation in instrumentation time with increasing number of `Units` to be patched. We observe that even without optimization discussed in § **??**, CLOTHO takes under 4s to instrument all `Units` across all benchmarks. We believe that this time would be even less with the optimizations enabled, which significantly decrease the number of `Units` to be instrumented, and is evident in Table 11.1 where column $\mathcal{IC}_{WO}$ is much less than $\mathcal{IC}_{NO}$.

11.3 Case studies

We now report on experiences gained when using CLOTHO to patch several of the bugs reported in Table 11.1.

- The bug [1] as reported in the repository for Apache Aries cited String related issues.

However, our investigation showed that the bug was actually in the ASM framework that was invoked by Aries, and not in the was actually not in the Aries framework as originally reported. Thus, we patched the particular ASM methods containing the bugs, and retested it with the Aries framework to ensure conformance.

- The bug in Commons Math [19] had a bug related to incorrect formatting of the input string. However, it threw a completely irrelevant exception (`IndexOutOfBound`) instead of the `NumberFormatException`, which contains the information of the malformed string. The CLOTHO-generated patch fixes the undesirable behavior.

- The bug in OfBiz [23] throws a custom shutdown exception, when in fact it should throw a `StringIndexOutOfBoundsException` due to a `substring` invocation with incorrect bounds. This ultimately causes the library to throw some high priority exception and ultimately crash if not handed properly by the application. The patched version of the library catches the correct exception.

- The code to trigger bugs in some libraries, including Apache Commons Compress, Commons Lang, Commons Math and Ofbiz, each had string operations wrapped in `try-catch` block that were handled by `Exception` class, i.e., the base type of all exceptions. However, CLOTHO checks for already handled runtime exceptions during its call graph analysis, and thus did not patch the bugs. We turned off the call graph analysis module to force CLOTHO to generate the relevant patch for the bug.

- We also noticed several instances where the developer code does not follow proper programming practices regarding exception handling. For example, the SOAP bug [27] was reported for a faulty `substring` call that threw a `StringIndexOutOfBoundsException`. The entire method was wrapped in a `try-catch` that included the faulty substring call along with other servlet operations. However, the `catch` block handled the generic `Exception`, which is the base class of all exceptions. Thus, both servlet exceptions or `IndexOutOfBoundException` from the `substring` call were handled in a generic fashion. CLOTHO's patched library ensures that exceptions originating from the `substring` call are handled properly.

API	BugID	Priority	$\mathcal{N}_{CG}$	$\mathcal{N}_{Unit}$	PQI	FCI	$\mathcal{IC}_{NO}$	$\mathcal{IC}_{WO}$	$\mathcal{RS}_{CE}$
`Aries`	[1]	Major	$3.5K$	129	1.02	0	42	5	✓
`Commons CLI1.x`	[3]	Critical	$3.2K$	53	0.79	0	19	19	✓
`Commons CLI2.x`	[2]	Major	$3.2K$	21	0.50	1	13	2	×
`Commons Compress`	[4]	Blocker	$4.0K$	134	0.99	0	46	4	✓
`Commons IO`	[15]	Major	$3.3K$	125	1.01	0	76	1	✓
`Commons Lang`	[17]	Major	$5.1K$	240	0.98	0	168	8	✓
`Commons Math`	[19]	Major	$3.4K$	300	1.00	1	36	2	✓
`Commons Net`	[21]	Major	$3.3K$	14	1.07	0	6	1	✓
`Commons VFS`	[32]	Major	$4.5K$	37	1.00	0	20	2	✓
`Derby`	[6]	Major	$4.4K$	40	0.96	0	47	6	✓
`Eclipse AJ Weaver`	[8]	Major	$20.6K$	50	0.52	0	4	1	×
`Eclipse AJ`	[7]	Major	$25.0K$	39	1.03	0	6	1	✓
`FlexDK 3.4`	[25]	Minor	$6.3K$	600	0.96	0	207	25	✓
`Hama 0.2.0`	[10]	Critical	$3.7K$	35	1.38	0	28	5	✓
`HBase 0.92.0`	[11]	Critical	$4.8K$	61	1.01	0	13	2	✓
`Hive`	[12]	Trivial	$4.4K$	23	1.01	0	8	1	✓
`HttpClient`	[13]	Major	$3.3K$	14	1.13	0	6	1	✓
`jUDDI`	[16]	Major	$3.2K$	70	1.13	0	10	2	✓
`Log4j`	[18]	Major	$3.2K$	17	0.97	0	6	1	✓
`MyFaces Core`	[20]	Major	$4.5K$	50	1.00	0	4	2	✓
`Nutch`	[22]	Major	$4.5K$	90	0.98	0	8	1	✓
`Ofbiz`	[23]	Minor	$4.4K$	28	1.01	1	6	1	✓
`PDFBox`	[24]	Major	$4.4K$	23	1.14	0	8	1	✓
`Sling Eclipse IDE`	[26]	Major	$4.5K$	58	1.00	0	39	6	✓
`SOAP`	[27]	Major	$5.0K$	165	0.97	1	32	5	✓
`SOLR 1.2`	[28]	Major	$11.0K$	200	0.98	0	25	4	✓
`Struts2`	[36]	Major	$16.0K$	80	1.03	0	25	2	✓
`Tapestry 5`	[31]	Major	$6.2K$	71	0.98	0	31	5	✓
`Wicket`	[34]	Major	$70.0K$	68	0.96	0	16	1	✓
`XalanJ2`	[37]	Major	$3.3K$	33	1.03	0	13	2	✓

$\mathcal{N}_{CG}$ # nodes in call graph
$\mathcal{N}_{Unit}$ # `Units` analyzed
$\mathcal{RS}_{CE}$ Cascaded exception exists

PQI Patch Quality Index
FCI Flow Consistency Index

$\mathcal{IC}_{NO}$ Instrumentation w/o optimization (recall § **??**)
$\mathcal{IC}_{WO}$ Instrumentation w/ optimization (recall § **??**)

Table 11.1: CLOTHO's accuracy results when applied to 30 bugs in popular open-source libraries.

Application	KLOC	Total paths	Tainted paths
`Checkstyle`	$58.0K$	1977	88
`Jazzy Core`	$4.9K$	270	26
`JEdit`	$4.3K$	185	22

Table 11.2: Precision results for taint analysis.

Chapter 12

Conclusion and Future

Bibliography

[1] ARIES-1204. [aries-1204] - stringindexoutofbounds for blueprint apps that have constructors with multiple exceptions. https://issues.apache.org/jira/browse/ARIES-1204, 2014.

[2] CLI-46. [cli-46] - java.lang.stringindexoutofboundsexception. https://issues.apache.org/jira/browse/CLI-46, 2007.

[3] CLI193. [cli-193] - stringindexoutofboundsexception in helpformatter.findwrappos. https://issues.apache.org/jira/browse/CLI-193, 2010.

[4] COMPRESS-26. [compress26] - tararchiveentry(file) now crashes on file system roots. https://issues.apache.org/jira/browse/COMPRESS-26, 2009.

[5] 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).

[6] DERBY-4748. [derby-4748] - stringindexoutofboundsexception on syntax error (invalid commit). https://issues.apache.org/jira/browse/DERBY-4748, 2010.

[7] Eclipse Bug 333066. Bug 333066 - stringindexoutofboundsexception during compilation. https://bugs.eclipse.org/bugs/show_bug.cgi?id=333066, 2014.

[8] Eclipse Bug 432874. Bug 432874 - stringindexoutofboundsexception after adding project to inpath. https://bugs.eclipse.org/bugs/show_bug.cgi?id=432874, 2014.

[9] 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.

[10] HAMA-212. [hama-212] - when the index is zero, bytesutil.getrowindex will throws the indexoutofbound. https://issues.apache.org/jira/browse/HAMA-212, 2009.

[11] HBASE-4481. [hbase-4481] - testmergetool failed in 0.92 build 20. https://issues.apache.org/jira/browse/HBASE-4481, 2011.

[12] HIVE-6986. [hive-6986] - matchpath fails with small resultexprstring. https://issues.apache.org/jira/browse/HIVE-6986, 2014.

[13] HTTPCLIENT-150. [httpclient-150] - stringindexoutofbound exception in rfc2109 cookie validate when host name contains no domain information and is short in length than the cookie domain. https://issues.apache.org/jira/browse/HTTPCLIENT-150, 2003.

[14] 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.

[15] IO-179. [io-179] - stringindexoutofbounds exception on filenameutils.getpathnoendseparator. https://issues.apache.org/jira/browse/IO-179, 2008.

[16] JUDDI-292. [juddi-292] - ¡faultstring¿string index out of range: 35¡/faultstring¿. https://issues.apache.org/jira/browse/JUDDI-292, 2011.

[17] LANG-457. [lang-457] - numberutils createnumber thows a stringindexoutofboundsexception when only an "l" is passed in. https://issues.apache.org/jira/browse/LANG-457, 2008.

[18] LOG4J2-448. [log4j2-448] stringindexoutofbounds when using property substitution - asf jira. https://issues.apache.org/jira/browse/LOG4J2-448, 2013.

[19] MATH-198. [math-198] - java.lang.stringindexoutofboundsexception in complexformat.parse(string source, parseposition pos). https://issues.apache.org/jira/browse/MATH-198, 2008.

[20] MYFACES-416. [myfaces-416] - stringindexoutofboundsexception in addresource. https://issues.apache.org/jira/browse/MYFACES-416, 2005.

[21] NET-442. [net-442] - stringindexoutofboundsexception: String index out of range: -1 if server respond with root is current directory. https://issues.apache.org/jira/browse/NET-442, 2012.

[22] NUTCH-1547. [nutch-1547] - basicindexingfilter - problem to index full title. https://issues.apache.org/jira/browse/NUTCH-1547, 2013.

[23] OFBIZ-4237. [ofbiz-4237] - shutdown exception if invalid string entered. https://issues.apache.org/jira/browse/OFBIZ-4237, 2011.

[24] PDFBOX-467. [pdfbox-467] - index out of bounds exception. https://issues.apache.org/jira/browse/PDFBOX-467, 2009.

[25] SDK-14417. [sdk-14417] - stringindexoutofboundsexception when using a properties-file. http://bugs.adobe.com/jira/browse/SDK-14417,https://issues.apache.org/jira/browse/FLEX-13823, 2008.

[26] SLING-3095. [sling-3095] - stringindexoutofboundsexception within contentxmlhandler.java:210. https://issues.apache.org/jira/browse/SLING-3095, 2013.

[27] SOAP-130. [soap-130] - string indexoutofbounds in soapcontext. https://issues.apache.org/jira/browse/SOAP-130, 2004.

[28] SOLR-331. [solr-331] - stringindexoutofboundsexception when using synonyms and highlighting. https://issues.apache.org/jira/browse/SOLR-331, 2007.

[29] Soot. Soot: a java optimization framework. http://www.sable.mcgill.ca/soot/.

[30] Soot-infoflow. secure-software-engineering/soot-infoflow. https://github.com/secure-software-engineering/soot-infoflow.

[31] TAP5-1770. [tap5-1770] - pagetester causes stringindexoutofboundsexception for any page request path with query parameter. https://issues.apache.org/jira/browse/TAP5-1770, 2011.

[32] VFS-338. [vfs-338] - possible crash in extractwindowsrootprefix method. https://issues.apache.org/jira/browse/VFS-338, 2010.

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

[34] WICKET-4387. [wicket-4387] - stringindexoutofboundsexception when forwarding requests. https://issues.apache.org/jira/browse/WICKET-4387, 2012.

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

[36] WW-650. [ww-650] - cooluriservletdispatcher throws stringindexoutofboundsexception. https://issues.apache.org/jira/browse/WW-650, 2005.

[37] XALANJ-836. [xalanj-836] - exception in org.apache.xalan.xsltc.compiler.util.util.tojavaname(string). https://issues.apache.org/jira/browse/XALANJ-836, 2004.

Appendix