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 **6**

2.1 Historical Context 6

2.2 Data from Stack Overflow Posts 7

2.3 Major Challenges 8

3 Related Works **9**

3.1 Recent Works on Data Structure Repairing 9

3.2 Works on Software Patching 10

3.3 Genetic Programming, Evolutionary Computation 10

4 Problem Formulation **11**

4.1 Runtime Exceptions 11

5 RepairingStrategy : Taint Analysis **12**

5.1 Taint analysis : Definition 12

5.2 Taint analysis : Taint Propagation 13

5.3 Taint Analysis : Relevance with Repairing Effort 13

6 Repairing Strategy : Exception Type **15**

6.1 Static Analysis . . . 17

6.2 Data set for Successful Program Runs . . . 18

6.3 Matrices . . . 18

6.4 Instrumenting Patching . . . 18

6.4.1 Determine Exception Type . . . 19

6.4.2 Determine Optimal Code Slice . . . 19

6.5 Variable Tracking and Monitoring . . . 21

7 Repairing Strategy : Bounded Forward and Backward Analysis **22**

7.1 Example Scenario . . . 22

7.2 Flow Functions . . . 23

7.2.1 Bounded Forward Analysis . . . 23

7.3 Constraint Satisfaction . . . 24

7.3.1 Constraint Storage . . . 25

7.3.2 Constraint Evaluation Strategy . . . 26

7.3.3 Repairing Strategy using Constraint Evaluation . . . 26

8 Repairing Strategy : Constraint Automata **27**

8.1 General Structure . . . 27

8.2 Patching Techniques . . . 28

8.2.1 Array index out of bound exception . . . 28

8.2.2 Negative Array Size Exception . . . 29

8.2.3 Arithmetic Exception : Division-by-zero Exception . . . 30

8.2.4 Null Pointer Exception . . . 31

9 Clotho: Design of the System **34**

9.1 Goals . . . 34

9.2 Design . . . 34

9.2.1 Precise Identification of Instrumentation Points: . . . 35

9.2.2 Patch Generation: . . . 37

9.2.3 Code generation . . . 40

9.3 Limitations . . . 43

10 Implementation Details **45**

10.1 Taint Analysis . . . 45

10.2 Call graph analysis . . . 46

10.3 Constraint Analysis . . . 46

10.4 String Repairing Phase . . . 46

10.4.1 Detecting Potential Point of Failure . . . 47

10.4.2 Catch Block Instrumentation . . . 47

10.5 Optimizations . . . 47

11 Evaluation **49**

11.1 Accuracy . . . 50

11.2 Overhead . . . 53

11.3 Case studies . . . 54

12 Conclusion and Future Work **58**

List of Figures

4.1 array index out of bound formulated as FSM 11

5.1 A simplified diagram indicating taint problem 13

6.1 Data dependency graph of the variables in code snippet 6.1 17

6.2 Indexed global variables and method arguments successful runs 18

7.1 Dataflow diagram with in, out set in forward analysis 24

7.2 String constraints storage format 25

8.1 Constraint automata general model 28

9.1 CLOTHO workflow. 35

9.2 Constraints involving Strings. 37

9.3 Partial constraint representation model. 43

11.1 CLOTHO evaluation. 53

List of Tables

2.1 Most frequent java runtime exceptions from stack overflow data 7

9.1 Common sensitive sources in Java. 36

9.2 Common sensitive sinks in Java. 36

11.1 Clotho's accuracy results when applied to 30 bugs in popular open-source libraries. 56

11.2 Clotho's profiling results for time and memory footprint 57

11.3 Precision results for taint analysis. 57

List of Code Snippets

1.1 Apache Log4j bug example. 3

6.1 Java code which may throws runtime exceptions 15

6.2 Patching code slice based on exception type 19

7.1 Dataflow analysis 22

7.2 Better patching mechanism with constraint satisfaction 24

8.1 array index out of bound patching 28

8.2 arr index out of bound patching 29

8.3 arithmetic exception : division-by-zero patching 30

8.4 appropriate constructor 31

8.5 array null pointer exception 32

9.1 Code requiring dynamic string constraint evaluation. 39

9.2 Dynamic constraint collection and evaluation corresponding to code 9.1. 40

9.3 Example of parameter tweaking. 41

Chapter 1

Introduction

Modern software applications are large and complex. In addition, they run in diverse environments and get inputs from variety of data sources. As a result, predicting safe behavior for them at runtime can be difficult. Moreover, giving assurances about the quality of service becomes practically impossible when the applications are developed using third party libraries and components. Such applications often exhibit vulnerabilities that can be exploited by providing malicious inputs. In addition, diverse data sources and complex constraints on them make it challenging for programmers to ensure that all data elements are correctly validated and processed. Any deficiencies in this process makes applications prone to failures. Such defects can be hard to detect at the compilation-time, and irrespective of the validation techniques used, some of them may go undetected and exist in the applications even when the applications are in production.

The cost of failures can vary considerably depending on the mission-critical nature of applications. In particular, it would be extremely undesirable for an unmanned aerial vehicle on its mission to allow the control system to crash in case of a failure. Instead a suboptimal functioning for a short while might be acceptable until the system fully stabilizes. Generally speaking, expectations about the quality of service would largely depend on how a failure may impact the business. For example, a commercial online store may not afford a crash while it is listing its products to a customer. Such unpleasant experiences might result in customers

Code Snippets 1.1: Apache Log4j bug example.

```
1 private int substitute() {
2   if (priorVariables == null) {
3     priorVariables = new ArrayList<String>();
4     priorVariables.add(new String(chars, offset, length));
5   }
6 }
```

moving to other online stores making an adverse impact on the business. Similarly, a software company launching a new product would expect it to be stable while the product is undergoing beta-testing. Any crashes occurring at that time would result in negative feedback from the users and loss in the business. To make the matter worse, these failures may occur in software components that do not possess critical functionality, and hence, may even get less attention to their quality at the time of development. Nevertheless, irrespective of the criticality of these components, if the crash occurs it is equally undesirable.

As an example, consider Code 1.1 which depicts a bug that existed in Apache Log4j library version 2.0-beta9 [24] and crashed the logging framework. It was reported as a major bug in spite of the fact that it occurred in logging component. The object `priorVariables` is a `List` of String. On line 4, there is no check on the variables to ensure that invariants such as `offset + length <= chars.length`, `offset > 0`, and `length > 0` hold. In case of a failure, rather than allowing the application to crash, organizations would like to collect diagnostic information to identify the defects and allow the system to run suboptimal behavior for a while until it stabilizes. As long as such suboptimal behavior is within acceptable limits, the program survival would get higher preference. The bug can then be fixed in the later versions.

Several approaches have been proposed in the past to ensure that programs can recover from failures. Some of the approaches are based on static repairing where the patches are synthesized automatically based on the counter examples found in the field [42]. However, it is not always desirable to shut down the system for the post-mortem analysis and then relaunch it after fixing the defect. In order to overcome this weakness, dynamic approaches have been proposed to deal with problems that are related to memory, data, and incorrect programming constructs such as infinite loops [2, 22, 32]. Some of the approaches work either by identifying and isolating

damaged data or memory portions [6, 8, 9], or by delaying the execution until the program self-stabilizes [13], or by finding the alternative execution paths [33], or by disabling suppressing signals and hoping that the program can recover automatically from the errors [25]. Static approaches strive for correctness whereas dynamic approaches are typically optimistic and work on the assumption that some suboptimal behavior under certain conditions is acceptable.

In this work, we propose a novel approach, which is hybrid in the nature and deals with the failures originated from either malformed strings or incorrect handling of strings. The approach first identifies program statements statically that might be vulnerable to string-related failures, and then develops patches by trying to identify origins of errors and constraints on the strings. It uses dynamic analysis to improve the precision of the patches generated by the static analysis. The approach targets string variables for patching, firstly, because strings are used heavily in JAVA APIs and have been common sources of errors, and secondly, because by targeting a specific type of data the approach can develop patches that are more precise and result in the behavior that would be close to the intended behavior.

This work makes following contributions:

- We present the design and implementation of CLOTHO (§ 2, § 9 and § 10) that generates effective program patches to handle string-related errors. These patches get activated only in case of program failures during runtime, and save program from crashing ensuring its acceptable behavior.

- We use a finite state machine (FSM) as a formalism (§ **??**) to describe the behavior of JAVA `String` API, and apply it to drive the generation of exception-specific patches.

- We applied CLOTHO to several hugely popular open-source libraries to patch 30 bugs, several of them rated critical or major, resulting from unhandled runtime exceptions from JAVA `String` APIs. The results of our study (§ 11) indicate that CLOTHO can effectively produce patches that save programs from crashing due to failures originating from known bugs. The study also gives insights into the characteristics of the commonly occurring string problems.

- Manual inspection of the CLOTHO-generated patches reveals that in most cases they are semantically similar to the ones produced by the developers in the later versions. Thus, CLOTHO can also guide developers in the process of building patches for the future versions.

We have made our source code and data sets available to the open source community at `http://goo.gl/d1zcXD`.

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) [45].
- 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 [19].
- 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 [**?**], [7], [**?**], [**?**], [**?**] 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 [14]. 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 [43] 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

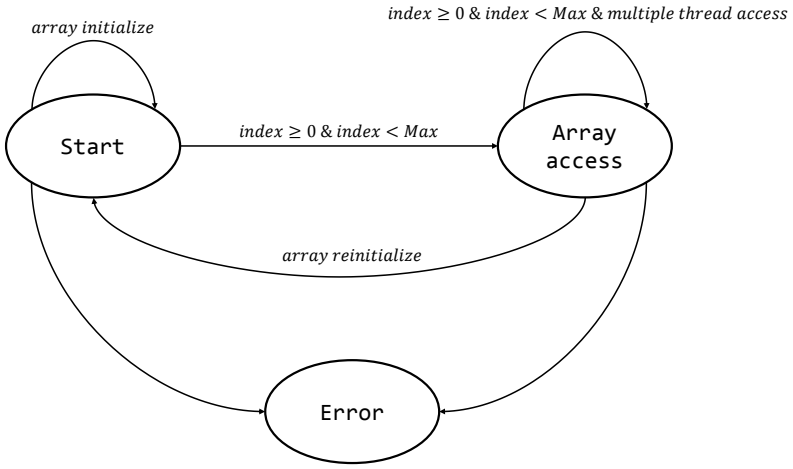


Figure 4.1: array index out of bound formulated as FSM

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 **??** Section **??**.

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.

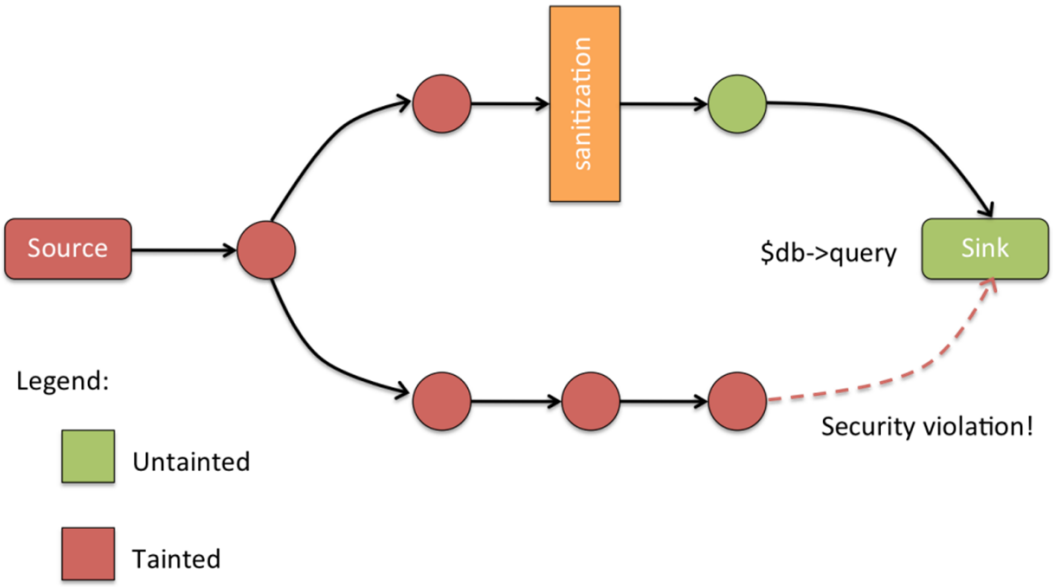


Figure 5.1: A simplified diagram indicating taint problem

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

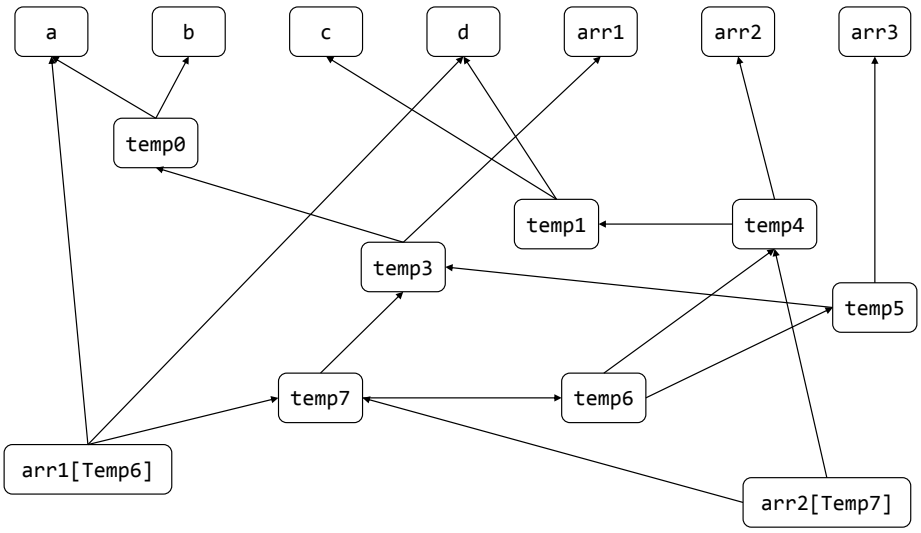


Figure 6.1: Data dependency graph of the variables in code snippet 6.1

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 **??**.

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

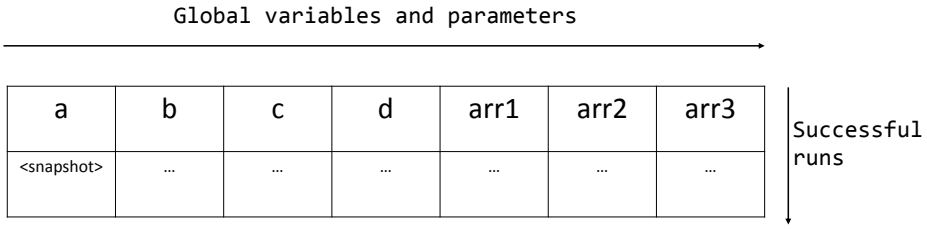


Figure 6.2: Indexed global variables and method arguments successful runs

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];
```

```
22                  int temp4 = this.arr2[temp1];
23            //IndexOutOfBoundException as temp3 = 20
24                  int temp5 = this.arr3[temp3];
25                  int temp6 = temp4 + temp5;
26                  int temp7 = temp6 - temp3;
27                  this.arr1[temp6] = temp7/(d-a);
28                  this.arr2[temp7] = temp7/temp4;
29                }
30                catch(IndexOutOfBoundsException indEx)
31                {
32                  int temp0 = a + b;
33                  int temp1 = c * d;
34                  int temp2 = temp0 - temp1;
35                  int temp3 = this.arr1[temp0];
36                //Bellow line is not part of the patch as
37                //temp1 and temp3are not related to temp3
38                //for which the exception occurred.
39                  //int temp4 = this.arr2[temp1];
40                  int temp5 = this.arr3[temp3];
41                }
42                if(arr2[temp1] ! = arr3[temp7])
43                       return arr1;
44                else
45                       return null;
46        }
47  }
48  public class MainClass
49  {
50        public void main(String[] a)
51        {
52                int[] arr1 = {20,21,22,23};
53                int[] arr2 = {1,2,3,4};
54                int[] arr3 = {10,11,12,13};
55                TestClass TC = new TestClass(arr1, arr2, arr3);
56                int[] res = TC.fun(2,4,3,2);
57                System.out.print("Result : "+res[2]);
58        }
59  }
```

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

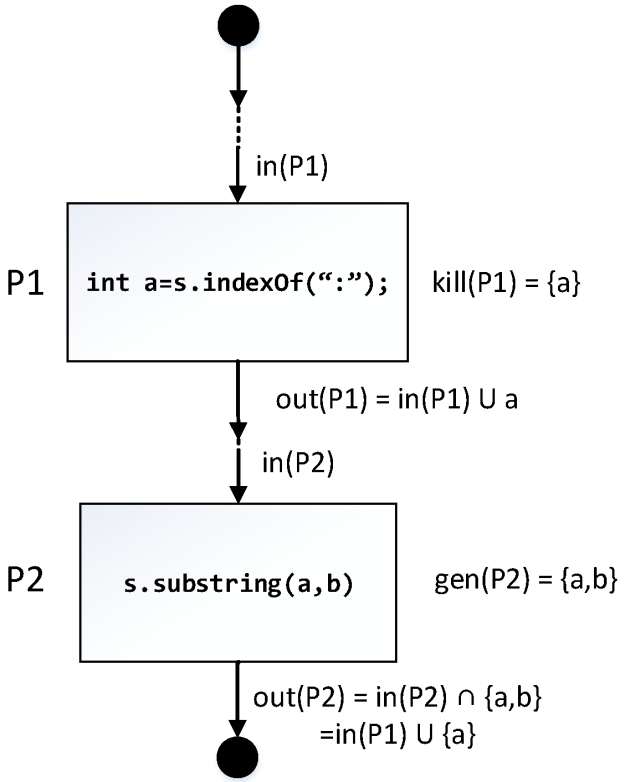


Figure 7.1: Dataflow diagram with in, out set in forward analysis

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 9.2.

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

Clotho: Design of the System

9.1 Goals

We identify the broad design goals for a technique to automatically repair malformed strings or incorrect handling of strings as follows:

(i) High patch fidelity. We require that the patched program must preserve the intended program behavior, i.e., the patch must be precise, and should not induce any undesirable control flows in the repaired program.

(ii) Non-invasive instrumentation. We require that the technique must ensure no side-effects (aside from optimally repairing objects) during normal program execution, and activate patches only when the program is guaranteed to crash.

(iii) Low system overhead. We desire that the patched program must incur no runtime overhead during normal program execution, and only negligible overhead in case of failures.

9.2 Design

Key Idea. CLOTHO leverages a combination of program analysis techniques to precisely identify program instrumentation points, and builds upon custom algorithms to generate targeted, high quality patches for repairing programs with potential runtime exceptions, while still satisfying

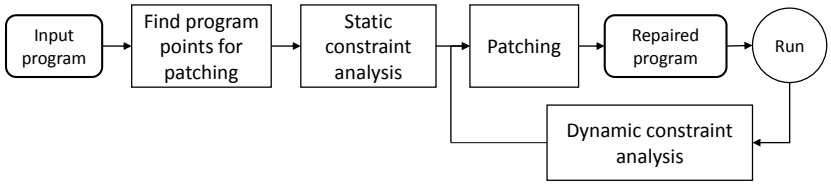


Figure 9.1: CLOTHO workflow.

goals mentioned in § 9.1.

Figure 9.1 shows CLOTHO's workflow, which involves three main stages. First, CLOTHO uses program analysis techniques to precisely identify points of interest, i.e., string objects or API arguments that must be repaired to prevent runtime exceptions. In the second stage, CLOTHO leverages custom algorithms to generate relevant patches. Specifically, CLOTHO performs intra-procedural static and dynamic analyses to identify and evaluate constraints on the string objects under consideration. Third, CLOTHO uses the constraints evaluated in the earlier stage to programatically generate and embed patches inside `catch` blocks to ensure that they do not get activated during normal program execution.

9.2.1 Precise Identification of Instrumentation Points:

In this stage, CLOTHO leverages a combination of program analyses to accurately determine the minimum set of points of interest where instrumentation is required to repair. We list several techniques below that help CLOTHO achieve precision.

(i) Taint analysis. The main purpose of taint analysis is to broadly identify which program statements can be patched (possibly even suboptimally) without affecting the program control flow, i.e., affect only objects that are generated and stay within the application throughout their lifetime. While this principle is not a binding constraint, it ensures that CLOTHO's repairing mechanism does not adversely affect critical program behavior. We specify a generic set of sensitive sources and sensitive sinks for each input program, to identify critical program paths where a repaired `String` objects (and thus possibly suboptimal) must not flow. For example, CLOTHO does not repair program statements that lie along a control flow path that leads to an I/O sink, like file system, console, network, GUI, etc.

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

Table 9.1: Common sensitive sources in JAVA.

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

Table 9.2: Common sensitive sinks in JAVA.

The taint analysis module take as input the compiled byte code intended to be repaired, and generates a control flow graph identifying program statements that lie along paths from sensitive sources to sensitive sinks. Since, CLOTHO targets strings in particular, it must support taint propagation for all JAVA APIs that support string manipulation, including `StringBuffer` and `StringBuilder`. All `String` objects (whether generated or assigned) that lie along the tainted path from a sensitive source to a sensitive sink are marked as *unsafe* to patch. Subsequently, CLOTHO does not repair such `String` objects. Tables 9.1 and 9.2 list some common sensitive sources and sinks for several classes in JAVA.

(ii) Call graph analysis. CLOTHO leverages call graph analysis to further improve the precision for finding instrumentation points. Although unlikely, it is possible that the developers may themselves handle code that raises runtime exceptions. Thus, CLOTHO must not instrument program points that are explicitly handled by the developers, since repairing such statements would definitely alter the intended control flow.

Checked runtime exceptions may be placed in the (i) same method, or (ii) upstream in the call chain. While handling the former scenario is trivial, CLOTHO handles the latter case by

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

Figure 9.2: Constraints involving Strings.

identifying all possible call chains (in the call graph) involving the concerned method using reverse Breadth First search (BFS), and determines ancestor methods where the call site was wrapped in `try-catch` block of compatible exception type or not.

(iii) Reaching definitions analysis. Taint and call graph analyses together provide a set of program points to be instrumented with the patch. However, this set can be further pruned. CLOTHO performs *reaching definitions* analysis to skip marked statements if (i) the string variables contained in such statements have already been patched upstream in the method, and (ii) the variables have not been redefined along any path that originates from the patched statement. This analysis further reduces instrumentation points in a program.

9.2.2 Patch Generation:

The output from the first stage is essentially a set of program points, typically bytecodes or some other intermediate representation, denoting `String` objects or APIs that are safe to repair. Once these instrumentation points have been identified, CLOTHO determines the possible patches that can be applied to each of them. Specifically, a program patch constitutes a set of constraints on either the `String` object or the parameters to the `String` API under consideration, such that the new repaired `String` object that is generated will satisfy all constraints and thus the patched program does not throw any runtime exceptions.

CLOTHO's patch generation mechanism involves two main parts (i) constraint collection and evaluation, and (ii) code generation. We now describe CLOTHO's patch generation mechanism in detail.

Constraint collection and evaluation. CLOTHO leverages a hybrid approach to collect all possible constraints that must be satisfied, and thus generates a high quality patch to repair the program. A constraint on a string object is defined as a set of permissible values that can uniquely define the string. CLOTHO uses a simple constraint set that includes minimum and

Algorithm 2: Patching strategy for `String` objects.

```
Data: Control flow graph CFG for program P
Result: Patched program P̂
begin
    for ∀ node N ∈ CFG do
        Statement S in node N
        if S contains String API call then
            str ⟵ String reference on S
        if S can throw RuntimeException then
            Exception class EC ⟵ RuntimeException of S
            CES_str ⟵ all conditional statement in P on str
            CS_str ⟵ output of Algorithm 3(CES_str)
            if str have sufficient constraints in CS_str then
                /* Static constraint analysis */
                str ⟵ output of Algorithm 4(CS_str)
            else if str encountered exception then
                /* Dynamic constraint analysis */
                CS_str ⟵ output of Algorithm 3(CES_str)
                if str have sufficient constraints in CS_str then
                    str ⟵ output of Algorithm 4(CS_str)
                else
                    str ⟵ output of Algorithm 6(S)
```

maximum length, along with set of permissible prefixes and substrings, as shown in Figure 9.2.

The hybrid approach has a static component that makes a forward pass over the program to collect declarative constraints on string objects, such as their length or prefix, etc. CLOTHO invokes the dynamic component if there are constraints such that the constraint set cannot be evaluated. In such scenarios CLOTHO (i) generates a patch that itself dynamically collects constraint information, (ii) augments it with the previously collected static constraint details, and (iii) evaluates these constraints on the fly to generate repaired `String` objects, which do not cause the program to throw runtime exceptions. Algorithm 2 gives an overview of this hybrid approach.

Static constraint collection: CLOTHO's static constraint collection phase identifies all declarative constraints. Algorithm 3 briefly describes the steps to populate the constraint store shown in Figure 9.2. Specifically, CLOTHO iterates over all program code and analyzes conditional statements involving string objects of the form `if (st.length() == 5)`. CLOTHO

Algorithm 3: Constraint collection for `String` objects.

Data: Set of conditional statement on string str
Result: Constraint set CS_{str}
begin
  for *Conditional statement*$\leftarrow i$, $\forall i \in CS_{str}$ **do**
    $i \Rightarrow str * OP$ `/*where * is the binary operator*/`
    if $* is ==$ **then**
      $maxlength_{str} \longleftarrow OP$
      $minlength_{str} \longleftarrow OP$
    else if $* is >$ **AND** $* is \geq$ **then** $minlength_{str} \longleftarrow OP$
    else if $* is <$ **AND** $* is \leq$ **then** $maxlength_{str} \longleftarrow OP$
    else if $* is$ *Prefix Check* **then** $PrefixSet_{str} \cup OP$
    else if $* is$ *Contains Check* **then** $ContainSet_{str} \cup OP$

Code Snippets 9.1: Code requiring dynamic string constraint evaluation.

```
1 void foo() {
2   String st = Input(); /* user input */
3   if (st.length() == 5) {/* do something */}
4   if (st.contains(Input())) {/* do something */}
5   st = st.substring(7, 10);
6 }
```

considers only those constraints that the object must satisfy to ensure that the control flows through the *preferred* branch of the conditional. We define the preferred branch as the one that does not throw exceptions or error conditions, like `System.err.print()`. In other words, CLOTHO only considers the conditional expressions in the branches that do not involve any exceptions or error paths. Note that CLOTHO also evaluates OP (in Algorithm 3) when collecting constraints, in case OP is a composite mathematical expression $f(x, y, z, ...)$, such as $x + y * z$, where all x, y and z are known to be numeric.

Dynamic constraint collection: The constraint set is populated at the end of the static phase, and CLOTHO leverages Algorithm 4 to evaluate these constraints and determine the potential safe values of the string object under consideration. However, there are scenarios, where there are potentially conflicting constraints or no permissible values of the constraints can be calculated statically.

Consider the example shown in Code 9.1, where the function `foo` performs a series of checks on

Algorithm 4: 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 constraint 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

Code Snippets 9.2: Dynamic constraint collection and evaluation corresponding to code 9.1.

```
1 String temp = Input();
2 ConstraintStore.updateSet("<foo()>", st, temp);
3 st = GenerateStringDynamic.init("<foo()>", st);
```

a user entered string before computing a substring on it. Since the constraints on the string `st` cannot be completely collected and evaluated statically. For such cases, CLOTHO instruments the code with statements to dynamically collect constraint information, augment them with previously known static constraints, and evaluate these constraints at runtime. Specifically, CLOTHO instruments the bytecode with constraint collection code just before the conditional statements under consideration. Thus, CLOTHO will insert Code 9.2 before line 4 in Code 9.1 to update and evaluate the set of constraints on string `st`.

9.2.3 Code generation

Code generation can be done either statically or dynamically depending on how the constraints are evaluated. In either scenario, a key component of code generation is *object repairing*.

Algorithm 5: Parameter tweaking based String patching.

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 $i > Length$ **OR** $j > Length$ **then**
    $Ri \longleftarrow Length - 1$ or $Rj \longleftarrow Length - 1$ based on condition
    `/* more conditions possible */`
  if $i < 0$ **OR** $j < 0$ **then**
    $Ri \longleftarrow 0$ or $Rj \longleftarrow 0$ based on condition
    `/* more conditions possible */`

Code Snippets 9.3: Example of parameter tweaking.

```
1 try{
2    c = s.chatAt(4);
3 } catch(IndexOutOfBoundException ex) {
4    c = s.failSafeCharAt(4, s.length());
5 }
```

Additionally, in certain cases where constraints cannot be satisfied, either statically or dynamically or both, Clotho resorts to *parameter tweaking*.

1. **Object repairing**: Clotho generates the code for the repaired object under consideration after all the constraints have been collected and evaluated. If the constraints are resolved statically, then Clotho updates its constraint data store and instruments the corresponding bytecodes appropriately. However, in case the patch requires dynamic constraint collection, Clotho embeds the code to dynamically collect constraints and generate the patch as well. Line 1 and 3 in Code 9.2 update the constraint set and generate the repaired object, respectively.

2. **Parameter tweaking**: It is possible that as a side-effect of object repairing, the newly patched object may throw runtime errors when invoked with certain string APIs. For example, `c = s.charAt(4);` may still throw runtime errors even if `s` has been repaired.

This is possible if the repaired `s` has a length less than 4. In such scenarios, CLOTHO patches the code with a `try-catch` block around the offending API call, and appropriately inserts the repaired code in the `catch` block but with tweaks to the API arguments, as shown in Code 9.3, to ensure that no further runtime exception is thrown. For example, if the length of the string is greater than 4, then the API works similar to default `charAt` API. However, if the length is 3, then line 4 is invoked with both arguments equal to string length, i.e., 3. Note that parameter tweaking is leveraged to counter a potentially suboptimal object repair that may throw cascading exceptions. Algorithm 6 briefly outlines the mechanism to correctly set the parameters for the offending string API.

Instrumentation:

CLOTHO embeds the repair in a `try-catch` ladder to ensure that the patches do not get activated during normal program execution, thereby minimizing any side-effects of repairing and preventing any inadvertent changes to the program's intended control flow.

An important task in the instrumentation stage is to determine the kind of exceptions that may be thrown, and appropriately construct the `catch` blocks. While most APIs throw only a single subclass of `RuntimeException`, it is possible that a statement may throw more than one subclasses, such as `NullPointerException` and `StringIndexOutOfBoundsException`. CLOTHO generates a `catch` ladder for each kind of exception, which also facilitates exception-specific repairing as well. In other words, a single patch may get distributed over multiple `catch` blocks. This mechanism is achieved with the help of a constraint representation model.

Constraint representation model. We use a finite state machine (FSM) as a formalism to describe the behavior of JAVA `String` API, and apply it to drive the generation of exception-specific `catch` blocks. The model is precomputed based on the API documentation of JAVA `Strings`. Formally, we define the constraint representation FSM model $(Q, \Sigma, \delta, q_0, F)$ as follows:

- Q: Set of states where $|Q| = 2$, *legal state*(safe) and *illegal state* (error).
- Σ: Set of symbols. Each symbol is defined as a tuple (ζ, η, Λ), where ζ is a `String` API

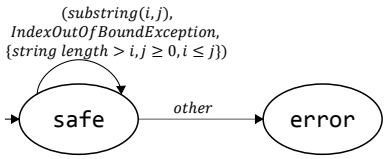


Figure 9.3: Partial constraint representation model.

operation, η is the type of an exception and $\Lambda = \{\lambda_1, \ldots, \lambda_n\}$ is the set of constraints. A constraint λ_i is defined as a constraint on a string that must be satisfied to allow successful execution of ζ.

- δ: Transition function. $safe \rightarrow safe$ is a safe transition and $safe \rightarrow error$ corresponds to the constraint violation.
- q_0: Starting state, here $q_0 = safe$.
- F: Singleton set of accept states which contains q_0.

A partial constraint representation model is depicted in Figure 9.3. It essentially specifies the constraints that are associated with `substring` method and `IndexOutofBoundException` exception that can be thrown by the method. A complete model would have several such self-looping transitions corresponding to other Java `String` API methods. The repairing mechanism gets triggered when an exception is thrown while performing a string operation after at least one of the constraints on the structure of the associated string is violated. This is represented by the transition labeled by `other`. The patches essentially support the same semantics identified by the transitions with the help of `catch` blocks.

9.3 Limitations

Clotho's major limitation arises from the fact that it is heavily directed towards repairing handling `String` objects and API exceptions. While this may seem to be a limitation, we believe that Clotho's strength lies in the fact that it mines contextual data about runtime exceptions related to `String` objects that helps development of intelligent program patches. Moreover, Clotho's technique is generic and can be ported to any other class of Java APIs.

Clotho generates precise patches considering the program context which avoids cascading

exceptions to a great extent producing the intended behavior in case of failures. However, it still cannot give guarantees about elimination of cascading exceptions, particularly when there are heavy object dependencies in the program.

The quality of CLOTHO's patches also depends on the nature of the constraint solver, which is pluggable. A more sophisticated solver may improve the quality of program repair, and we leave comparison of different solvers for future work.

Chapter 10

Implementation Details

We implemented a prototype of CLOTHO as described in § 9 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 [38] framework for bytecode analysis and instrumentation, and INFOFLOW [39] 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 § 9.2.1). 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 § 9.2.2), 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 4. 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 9.2).

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 6 to repair them.

Algorithm 6: 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 § 10.5 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 [16] 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 [15] 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

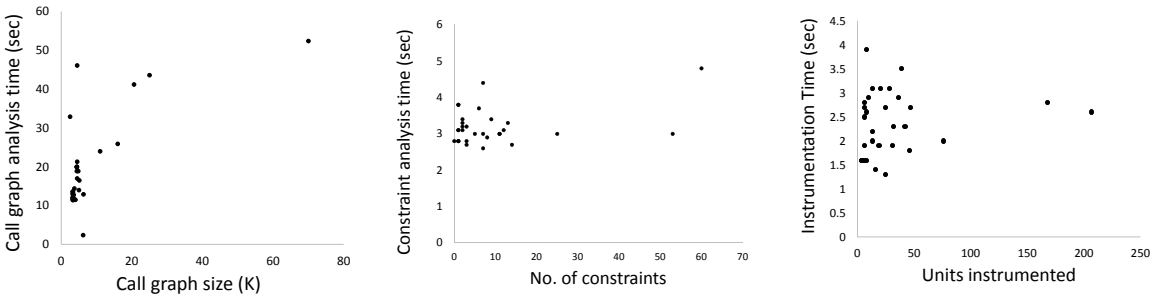
we could not measure the precision of our taint analysis on the library benchmarks (as discussed earlier), we select 3 diverse applications and apply CLOTHO in its entirety to obtain a measure of the precision of the taint analysis. Specifically, for each application we provided a set of sources, sinks 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 11.3 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 § 10.5, 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 [26] 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 [30] 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 [36] 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	Bug Ref	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`	[4]	Critical	$3.2K$	53	0.79	0	19	19	✓
`Commons CLI2.x`	[3]	Major	$3.2K$	21	0.50	1	13	2	×
`Commons Compress`	[5]	Blocker	$4.0K$	134	0.99	0	46	4	✓
`Commons IO`	[20]	Major	$3.3K$	125	1.01	0	76	1	✓
`Commons Lang`	[23]	Major	$5.1K$	240	0.98	0	168	8	✓
`Commons Math`	[26]	Major	$3.4K$	300	1.00	1	36	2	✓
`Commons Net`	[28]	Major	$3.3K$	14	1.07	0	6	1	✓
`Commons VFS`	[41]	Major	$4.5K$	37	1.00	0	20	2	✓
`Derby`	[10]	Major	$4.4K$	40	0.96	0	47	6	✓
`Eclipse AJ Weaver`	[12]	Major	$20.6K$	50	0.52	0	4	1	×
`Eclipse AJ`	[11]	Major	$25.0K$	39	1.03	0	6	1	✓
`FlexDK 3.4`	[34]	Minor	$6.3K$	600	0.96	0	207	25	✓
`Hama 0.2.0`	[15]	Critical	$3.7K$	35	1.38	0	28	5	✓
`HBase 0.92.0`	[16]	Critical	$4.8K$	61	1.01	0	13	2	✓
`Hive`	[17]	Trivial	$4.4K$	23	1.01	0	8	1	✓
`HttpClient`	[18]	Major	$3.3K$	14	1.13	0	6	1	✓
`jUDDI`	[21]	Major	$3.2K$	70	1.13	0	10	2	✓
`Log4j`	[24]	Major	$3.2K$	17	0.97	0	6	1	✓
`MyFaces Core`	[27]	Major	$4.5K$	50	1.00	0	4	2	✓
`Nutch`	[29]	Major	$4.5K$	90	0.98	0	8	1	✓
`Ofbiz`	[30]	Minor	$4.4K$	28	1.01	1	6	1	✓
`PDFBox`	[31]	Major	$4.4K$	23	1.14	0	8	1	✓
`Sling Eclipse IDE`	[35]	Major	$4.5K$	58	1.00	0	39	6	✓
`SOAP`	[36]	Major	$5.0K$	165	0.97	1	32	5	✓
`SOLR 1.2`	[37]	Major	$11.0K$	200	0.98	0	25	4	✓
`Struts2`	[46]	Major	$16.0K$	80	1.03	0	25	2	✓
`Tapestry 5`	[40]	Major	$6.2K$	71	0.98	0	31	5	✓
`Wicket`	[44]	Major	$70.0K$	68	0.96	0	16	1	✓
`XalanJ2`	[47]	Major	$3.3K$	33	1.03	0	13	2	✓

$\mathcal{N}_{CG}$ # nodes in call graph
$\mathcal{IC}_{NO}$ Instrumentation w/o optimization (recall § 10.5)
FCI Flow Consistency Index
$\mathcal{RS}_{CE}$ Cascaded exception exists

PQI Patch Quality Index
$\mathcal{N}_{Unit}$ # `Units` analyzed
$\mathcal{IC}_{WO}$ Instrumentation w/ optimization (recall § 10.5)

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

API	$\mathcal{PF}_{CA}$	$\mathcal{PF}_{TA}$	$\mathcal{PF}_{CG}$	$\mathcal{PF}_{IN}$
Aries	3.1/10	0.6/31	12.8/146	2.3/142
Commons CLI1.x	2.8/5	0.5/30	11.6/149	1.9/133
Commons CLI2.x	2.8/5	0.6/32	12/212	2/131
Commons Compress	2.7/5	0.5/30	11.5/209	1.8/130
Commons IO	3/11	0.5/33	12/209	2/141
Commons Lang	3/19	0.57/30	16.5/209	2.8/158
Commons Math	3/20	0.5/30	11.9/209	2.9/152
Commons Net	2.8/6	0.5/33	11.4/212	1.9/132
Commons VFS	3.7/13	1.4/7	46.1/151	3.1/143
Derby	3.3/15	0.5/31	19.9/208	2.7/146
Eclipse AJ Weaver	3.1/18	0.6/34	41.2/212	1.6/142
Eclipse AJ	3.8/24	0.5/34	43.6/214	1.6/156
FlexDK 3.4	4.8/40	0.4/31	12.9/209	2.6/189
Hama 0.2.0	2.6/5	0.5/33	14.4/210	3.1/134
HBase 0.92.0	3.2/15	1.4/11	18.9/212	3.1/144
Hive	3.4/12	1.5/11	20/121	1.6/143
HttpClient	2.8/6	0.5/33	12.3/212	2.7/131
jUDDI	3.1/11	0.5/34	13.6/209	2.9/138
Log4j	2.8/4	0.5/32	13/212	2.5/131
MyFaces Core	3.8/4	0.5/30	17/218	1.6/130
Nutch	3.2/15	1.2/30	32.9/215	3.9/147
Ofbiz	3.1/15	1.3/14	18.9/215	2.8/149
PDFBox	3.3/12	1.5/1	20/212	2.6/143
Sling Eclipse IDE	3/14	0.5/34	21.3/208	3.5/147
SOAP	3.4/19	0.6/30	14/209	2.3/148
SOLR 1.2	4.4/21	0.6/32	24/219	2.7/139
Struts2	3/14	0.5/32	25.9/210	1.3/140
Tapestry 5	2.9/10	0.5/34	2.4/211	1.9/136
Wicket	3/14	0.5/32	52.4/210	1.4/142
XalanJ2	2.7/8	0.5/33	13.5/213	2.2/131

$\mathcal{PF}_{CA}$ Profiling for constraint analysis phase | $\mathcal{PF}_{TA}$ Profiling for taint analysis phase
$\mathcal{PF}_{CG}$ Profiling for call graph phase | $\mathcal{PF}_{IN}$ Profiling for instrumentation phase

Table 11.2: CLOTHO's profiling results for time and memory footprint

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

Table 11.3: Precision results for taint analysis.

Chapter 12

Conclusion and Future Work

Running programs may crash unexpectedly due to vulnerabilities in the code and malformed data. The cost associated with such crashes can be vary with the criticality of the applications. Therefore, it is hardly a surprise that automatic program repairing has been an actively researched area over the past decade. In this work, we have presented a novel program repairing technique, and a tool CLOTHO based on it which employs hybrid program analysis to protect a running program from failures originating from string-handling errors leading to a program crash. Our choice of JAVA `String` APIs is driven mainly by the popular usage of string objects and bugs associated with them. By focusing on a specific data type, and taking the program context into account, CLOTHO can develop patches that are precise and semantically close to the ones developed by the developers. Hence, when the patches are activated, the program exhibits a behavior which is close to the intended program behaviour. Our study shows that CLOTHO can handle programs that are real, and can produce patches efficiently. Motivated by the results of our study as well as by the research conducted by other researchers, we intend to extend CLOTHO in the future by adding support for other JAVA APIs and also by adding more intelligence to the process of patch generation.

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] Carbin, M., Misailovic, S., Kling, M., and Rinard, M. C. Detecting and escaping infinite loops with jolt. In *Proceedings of the 25th European Conference on Object-oriented Programming* (Berlin, Heidelberg, 2011), ECOOP'11, Springer-Verlag, pp. 609–633.

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

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

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

[6] 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), pp. 233–244.

[7] 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).

[8] 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), pp. 71–84.

[9] 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), pp. 176–185.

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

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

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

[13] Eom, Y. h., and Demsky, B. Self-stabilizing java. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, NY, USA, 2012), PLDI '12, ACM, pp. 287–298.

[14] 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.

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

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

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

[18] 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.

[19] 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.

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

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

[22] Kling, M., Misailovic, S., Carbin, M., and Rinard, M. C. Bolt: on-demand infinite loop escape in unmodified binaries. In *OOPSLA* (2012), G. T. Leavens and M. B. Dwyer, Eds., ACM, pp. 431–450.

[23] 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.

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

[25] 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), p. 26.

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

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

[28] 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.

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

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

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

[32] 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), pp. 87–102.

[33] Pezzè, M., Rinard, M. C., Weimer, W., and Zeller, A. Self-repairing programs (dagstuhl seminar 11062). *Dagstuhl Reports 1*, 2 (2011), 16–29.

[34] 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.

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

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

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

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

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

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

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

[42] Wei, Y., Pei, Y., Furia, C. A., Silva, L. S., Buchholz, S., Meyer, B., and Zeller, A. Automated fixing of programs with contracts. In *ISSTA 2010: Proceedings of the 19th international symposium on Software testing and analysis* (New York, NY, July 2010), ACM, pp. 61–72.

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

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

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

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

[47] 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