CLOTHO: Saving Programs from Malformed Strings and Incorrect String-Handling

Aritra Dhar[*] Rahul Purandare[*] Mohan Dhawan[*,⁂] Suresh Rangaswamy[*]

[*]Xerox Research Centre India
Bangalore, India

[*]IIIT Delhi
New Delhi, India

[⁂]IBM Research
New Delhi, India

ABSTRACT

Software is susceptible to malformed data originating from untrusted sources. Occasionally the programming logic or constructs used are inappropriate to handle the varied constraints imposed by legal and well-formed data. Consequently, softwares may produce unexpected results or even crash.

In this paper, we present CLOTHO, a novel hybrid approach that saves such softwares from crashing when failures originate from malformed strings or inappropriate handling of strings. CLOTHO statically analyses a program to identify statements that are vulnerable to failures related to associated string data. CLOTHO then generates patches that are likely to satisfy constraints on the data, and in case of failures produces program behavior which would be close to the expected. The precision of the patches is improved with the help of a dynamic analysis.

We have implemented CLOTHO for the JAVA `String` API, and our evaluation based on several popular open-source libraries shows that CLOTHO generates patches that are semantically similar to the patches generated by the programmers in the later versions. Additionally, these patches are activated only when a failure is detected, and thus CLOTHO incurs no runtime overhead during normal execution, and negligible overhead in case of failures.

Categories and Subject Descriptors

D.2.5 [**SOFTWARE ENGINEERING**]: Testing and Debugging —*Error handling and recovery*

Keywords

Automatic Program Repair, Program Analysis, Strings

1. INTRODUCTION

Developers invest a significant amount of time and human involvement in testing and verification to make their software production ready. However, in spite of this effort and the tools used to ensure its safety and security, the software invariably carries subtle bugs, which are evident only when the software throws an exception and crashes. The cost of a crash varies depending on the criticality of the software, and whether it occurred during production or testing.

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

Code 1: Apache Log4j bug example.

A software bug in production systems may result in huge monetary losses to the tune of hundreds of millions of dollars for organizations running third-party software [1–4]. Further, these organizations must wait for the vendor to release a patch for the offending software, which may take days or even weeks. If a major software bug strikes during the internal acceptance testing, it may significantly hamper the testing progress itself, thereby affecting the entire software release cycle. Additionally, the software testers may have to wait for the newer patched version before they resume the testing process. Lastly, any such crash during a software's beta testing phase might frustrate the public resulting in rejection of the product itself. In all of the above scenarios, it would be extremely useful if a temporary program patch that not only saves the program from crashing, but also guarantees *acceptable* and close to the intended behavior can be applied to the software on-the-fly.

Software failures that result in crashes often originate from subtle program bugs that are related to unusual program inputs, unexpected environment changes, or specific thread schedules. While crashes are always undesirable, they are particularly annoying when they arise from *non-critical* modules that are not related to the core software functionality. For example, Code 1 depicts a bug in Apache Log4j library version 2.0-beta9 [34] that crashed the entire 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 such failures, rather than allowing the application to crash, organizations would prefer to collect diagnostic information to identify the defect, and proceed with a sub-optimal execution run hoping that it will eventually stabilize, or reveal a few more bugs.

Prior work [6, 36, 43, 56] proposes several mechanisms to automatically fix incorrect program behavior by generating program patches. These approaches either need a complete system shutdown to apply a patch, or isolate the faulty part of a data structure on the fly thereby limiting the functionality, or keep suppressing the exceptions with a hope that a suboptimal behavior would be acceptable until the application stabilizes.

ESEC/FSE'15, August 30 – September 4, 2015, Bergamo, Italy
© 2015 ACM. 978-1-4503-3675-8/15/08...$15.00
http://dx.doi.org/10.1145/2786805.2786877

In this work, we propose a novel hybrid approach that deals with failures originating due to malformed strings, or incorrect handling of strings in JAVA applications. We target string objects for patching, in particular, for the following two reasons. First, JAVA applications are typically built using libraries, and `String` APIs are commonly used in third party libraries [17, 31, 32]. In order to understand the usage and potential involvement of JAVA string objects in the application failures, we mined `stackoverflow` [52] for related posts. We observed that almost 33K out of 60K posts contained JAVA string related exceptions, indicating heavy string usage in programs. Second, we exploit extensive domain knowledge about strings to automatically synthesize high-quality patches.

CLOTHO performs precise static analysis to identify program locations that are vulnerable to string-related failures, and also the contexts under which they trigger a failure. This enables repairing the program close to the point of failure and generating precise patches that take into account the constraints on the string objects. CLOTHO further uses dynamic analysis to improve the precision of the patches generated by the static analysis.

We applied CLOTHO to patch 30 bugs, several of them rated critical or major, resulting from unhandled runtime exceptions from JAVA `String` APIs in various hugely popular open-source libraries. Our evaluation shows that CLOTHO develops precise patches that are semantically close to the ones developed by the developers.

This work makes following contributions:

(**1**) We present the design and implementation of CLOTHO (§ 2, § 4 and § 5) that automatically generates effective program patches to handle string-related errors.

(**2**) We use a finite state machine (FSM) as a formalism (§ 4) to describe the behavior of JAVA `String` API, and apply it to drive the generation of exception-specific patches.

(**3**) Our evaluation (§ 6) indicates that CLOTHO can effectively produce patches that save programs from crashing due to failures originating from known bugs. The results also gives insights into the characteristics of the commonly occurring string problems.

(**4**) Manual inspection of CLOTHO-generated patches reveal 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.

Our source code and data sets are available to the open source community at `https://github.com/aritradhar/CLOTHO`.

2. OVERVIEW

There is always a tradeoff between precision and scalability that static program analysis must balance. Static analysis achieves high scalability by making sound approximations, which typically leads to false positives. Complex programming logic and data coming from diverse sources make the already hard problem worse. As a result, successful execution of an application can never be guaranteed, and unexpected failures may happen. These failures often result in applications throwing runtime exceptions, which if not handled correctly may crash the application.

Code 2 shows a snippet from the `fileUtils` class in Apache Common IO library. The method `getPathNoEndSeparator` throws a `StringIndexOutOfBounds` exception, which originates from the `return` statement on line 13 when the method is called with parameter `"/foo.xml"`. Here, the value of `prefix` as returned by `getPrefixLength` is 1. It fails to satisfy the constraint implied by the program condition `prefix <= index + separatorAdd` for `substring` method, which ensures that `beginIndex` cannot be greater than `endIndex`. As a result, the exception is thrown.

```
1  public static String getPathNoEndSeparator
2        (String filename) {
3    return doGetPath(filename, 0);
4  }
5  private static String doGetPath
6        (String filename, int separatorAdd) {
7    if(filename == null) return null;
8    int prefix = getPrefixLength(filename);
9    if (prefix < 0) return null;
10   int index = indexOfLastSeparator(filename);
11   if ((prefix >= filename.length()) || (index < 0))
12       return "";
13   return filename.substring(prefix,
14       index + separatorAdd);
15 }
```

Code 2: Snippet from `fileUtils` class in Apache Commons library.

```
13 String temp = null;
14 try {
15   temp = filename.substring(prefix, index +
         separatorAdd);
16 } catch(IndexOutOfBoundsException ex) {
17    int length = filename.length;
18    int t = index + separatorAdd;
19    temp = filename.substring(
20    getStart(prefix,t,length),getEnd(prefix,t,length));
21 }
22 return temp;
```

Code 3: Patch for `fileUtils` bug in Apache Commons library.

A closer inspection of this code snippet shows that the string variable `filename` invokes two methods, namely `length` and `substring` on lines 11 and 13 respectively. JAVA `String` API documentation specifies that `length` does not throw any runtime exceptions. The only exception that this invoke statement can throw is when the receiver object referenced by `filename` is `null`. However, the check on line 7 indicates that this situation would not arise. The method `substring` may throw `IndexOutOfBoundsException` exception that can potentially crash the program. A good patch to handle this failure should take into account all of these observations.

Code 3 presents the patch automatically generated by CLOTHO. This patch replaces the invoke statement on line 13 in Code 2, which is now wrapped within a `try-catch` block. The `catch` corresponding to `IndexOutOfBoundsException` ensures that control passes to the catch block only when the exception is thrown. Line 20 shows two method calls namely `getStart` and `getEnd` that are inserted by CLOTHO. These methods, using the knowledge about the length of `filename` acquired with the help of the code on line 17, compute legally correct indexes required by `substring` method to satisfy the constraint related to `beginIndex` and `endIndex`. Method `substring` can now regenerate the substring ensuring that the method call will not fail. The actual patch provided by the developers is semantically similar to the one developed by CLOTHO, and both versions of the code generate the same output.

3. PROBLEM DEFINITION

Let the behavior $\mathcal{B}$ of program $\mathcal{P}$ for input $\mathcal{I}$ be a sequence of data values $< b_1, \ldots, b_n >$ shared with the environment, where these values may be used to print information on screen, access and manipulate files and databases, and exchange data with other processes or threads. For brevity, we assume that the program is sequential. However, our formalization and arguments can be extended to multi-threaded programs and their behaviors.

Consider the behavior $\mathcal{B}$ to be composed of $\mathcal{B}_n c = < b_{1_c}, \ldots, b_{n_c} >$ and $\mathcal{B}_n = < b_{1_n}, \ldots, b_{n_n} >$, where elements in $\mathcal{B}_c$ consist of critical

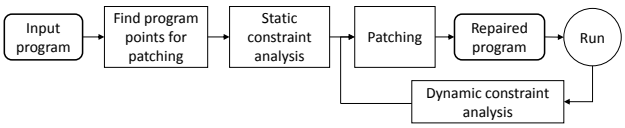


Figure 1: CLOTHO workflow.

values that form core functionality of the program, while elements in $\mathcal{B}_n$ are noncritical values with respect to the core functionality of the program. If $\mathcal{B}_\theta$ is the behavior for $\mathcal{P}$ under failure input θ, then our approach attempts to develop a program patch to convert $\mathcal{P}$ to $\mathcal{P}'$, such that $\mathcal{P}'$ does not violate the core behavior of $\mathcal{P}$ under failure, i.e., $\mathcal{B}'_{\theta_c}$ is same as the intended behavior $\mathcal{B}_{\theta_c}$. Note that resulting behavior $\mathcal{B}'_{\theta_n}$ may not be equivalent to $\mathcal{B}_{\theta_n}$.

In this work, we restrict our approach to `string` data values. 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. This goal naturally follows from the problem definition. However, we set two more goals associated with the security and performance of the technique.
(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.

4. CLOTHO

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 goals mentioned in § 3.

Figure 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.

4.1 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 precisely identify which program statements can be patched (possibly even suboptimally) without affecting the program's 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 sinks for each input program, to identify critical program paths where repaired `String` objects must not flow. For example, CLOTHO does not repair statements that lie along a control flow path leading to an I/O sink such as file system, console, network, and GUI.

Table 1: Common sensitive sources in JAVA.

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 2: Common sensitive sinks in JAVA.

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

The taint analysis module takes 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 `String` objects in particular, it must support taint propagation for all JAVA APIs that support string manipulation, including `StringBuffer` and `StringBuilder`. CLOTHO makes a default, conservative assumption that all objects leaving the system have potential to trigger unintended behavior. Thus, 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.

CLOTHO's configuration further enables developers to specify their own sets of sensitive sources and sinks, and exclude specific tainted paths. Tables 1 and 2 list 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. 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 identifying all possible call chains (in the *call graph*) involving the concerned method using reverse breadth first search, and determining ancestor methods where the call site was wrapped in `try-catch` block of compatible exception type.
(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.

4.2 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,

Table 3: Constraints involving `Strings`.

Min length(L)	Max length(L)	Prefix 1	...	Prefix$L$	Contain 1	...	Contain L

```
Data: Control flow graph CFG for program P
Result: Patched program P'
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
            /* Constraint collection for str */
            CS_str ⟵ output of Algorithm 3(CES_str)
            if str have sufficient constraints in CS_str then
            /* Constraint evaluation for str */
                str ⟵ output of Algorithm 4(CS_str)
            else
              /*Repair by parameter tweaking*/
              str ⟵ output of Algorithm 5(S)
            S' ⟵ Modify S with str
            Put S in try block
            Put S' with patched str in catch block
            with the exception class EC
    end
end
```

Algorithm 1: Static patching strategy for `String` objects.

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 satisfies all constraints. 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 uniquely define the string. CLOTHO uses a *constraint store* to maintain a set that includes constraints on minimum and maximum lengths, along with a set of permissible prefixes and substrings, as shown in Table 3.

The hybrid approach has a static component that makes a forward pass over the program to collect constraints on `String` objects, such as their length or prefix. 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. CLOTHO propagates constraints to ensure that the static constraint collection works correctly if the conditional statements involve any variables that are redefined during the collection and can be calculated statically. CLOTHO only collects constraints that are directly associated to `String` objects. Algorithms 1 and 2 give an overview of this hybrid approach.

(**1**) **Static constraint collection**: CLOTHO's static constraint collection phase identifies all constraints. Algorithm 3 briefly describes the steps to populate the constraint store. Specifically, CLOTHO identifies and analyzes conditional statements involving string objects such as `if (st.length() == 5)`. It considers only those constraints that the object must satisfy to ensure control flows through one of the *normal* branches of the conditional. A normal branch is one that does not represent an erroneous control flow. For

```
Data: Control flow graph CFG for program P
Result: Patched program P'
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
            CES_str ⟵ all conditional statement in P on str
            /* Constraint collection for str */
            CS_str ⟵ output of Algorithm 3(CES_str)
            if S encountered exception then
              if str have sufficient constraints in CS_str then
              /* Constraint solving for str */
                str ⟵ output of Algorithm 4(CS_str)
              else
                /* Parameter tweaking */
                str ⟵ output of Algorithm 5(S)
    end
end
```

Algorithm 2: Dynamic patching strategy for `String` objects.

```
Data: Set of conditional statement on string str
Result: Constraint set CS_str
begin
    for Conditional statement← i, ∀i ∈ CS_str do
        i ⇒ str * OP /*where * is the binary operator*/
        if * is == then
            maxlength_str ⟵ OP
            minlength_str ⟵ OP
        else if * is > OR * is ≥ then minlength_str ⟵ OP
        else if * is < OR * is ≤ then maxlength_str ⟵ OP
        else if * is Prefix Check then PrefixSet_str ∪ OP
        else if * is Contains Check then ContainSet_str ∪ OP
    end
end
```

Algorithm 3: Constraint collection for `String` objects.

example, a branch that encounters a statement `System.err.print()` would represent an erroneous control flow. Code 4 shows a string variable with several constraints as defined by the `if` statements on lines 2-5. The first three constraints are static as all of them can be evaluated at the compilation time. By analyzing these statements, CLOTHO populates the constraint store depicted in Figure 2.

(**2**) **Dynamic constraint collection**: The constraint set is populated at the end of the static phase. 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 exist potentially conflicting constraints or no permissible values of the constraints can be calculated statically.

For example, function `foo` in Code 4 performs a series of checks on a user-entered string `st` before computing a substring on it. Since the constraints on `st` cannot be completely collected and evaluated statically, CLOTHO instruments the code with statements to dynamically collect constraint information, augments them with previously known static constraints, and evaluates these constraints at runtime. Specifically, CLOTHO instruments the bytecode with constraint collection code just before the conditional statements under consideration. Code 5 depicts the example in Code 4 after CLOTHO's instrumentation. When the constraints in the source are complex, CLOTHO relies on its base framework to simplify them and then evaluates only the ones that are related to strings.

4.3 Code Generation

Code generation is done either statically or dynamically depending on how the constraints are evaluated. In either scenario, a key component of code generation is *object repairing*. Additionally, in certain cases where constraints cannot be satisfied, either statically or dynamically or both, CLOTHO resorts to *parameter tweaking*.

```
1 void foo(){
2   String st = "test String";
3   if(st.length == 5) {/*do something*/}
4   if(st.startsWith("ab")) {/*do something*/}
5   if(st.startsWith("abcd")) {/*do something*/}
6   /*userInput() accepts String input from console*/
7   if(st.contains(userInput())){/*do something*/}
8   st = st.substring(7, 10); /*Potential failure*/
9 }
```

Code 4: Static and dynamic constraint collection example

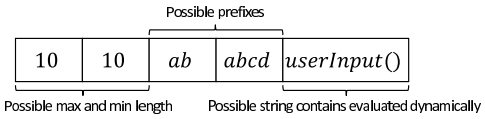


Figure 2: Constraint store for Code 4

```
1  String st = "test String";
2  ConstraintStore.updateLength("<foo()>", st, 5);
3  /*Executes if there is an exception over st*/
4  st = GenerateStringStatic.init("<foo()>", st);
5  if(st.length == 5) {}
6  if(st.startsWith("ab")) {/*do something*/}
7  ConstraintStore.updatePrefix("<foo()>", st, "ab");
8  st = GenerateStringStatic.init("<foo()>", st);
9  ConstraintStore.updatePrefix("<foo()>", st, "abcd");
10 st = GenerateStringStatic.init("<foo()>", st);
11 if(st.startsWith("abcd")) {/*do something*/}
12 String temp = Input();
13 ConstraintStore.updateSet("<foo()>", st, temp);
14 st = GenerateStringDynamic.init("<foo()>", st);
15 if(st.contains(temp){/*do something*/}
```

Code 5: Dynamic constraint collection and evaluation for code 4.

(**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 generates the patch as well. Lines 4, 8, 10 and 14 in Code 5 update the constraint set and generate the repaired object.

(**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. The snippet `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 to ensure that no further runtime exception is thrown (see Code 6). 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 5 briefly outlines the mechanism to correctly set the parameters for the offending string API.

4.4 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.

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
  end
  for $C \in ContainSet_{Str}$ **do**
    if C *is Empty* **OR** $C \in Str$ **then**
      continue
    $Str \leftarrow Str$ **APPEND** C
  end
  return Str
end

Algorithm 4: `String` object constraint evaluation.

Data: String object Str and index set IS which contains i or i, j.
Result: Repaired index set containing R_i or R_i, R_j based on input IS
begin
  $Length \longleftarrow$ length of Str
  if $Length == 0$ **then**
    $R_i, R_j \longleftarrow 0$
  else if $i > j$ **then**
    $R_i \longleftarrow j - 1$
  if $i > Length$ **OR** $j > Length$ **then**
    $R_i \longleftarrow Length - 1$ or $R_j \longleftarrow Length - 1$ based on condition
    `/* more conditions possible */`
  if $i < 0$ **OR** $j < 0$ **then**
    $R_i \longleftarrow 0$ or $R_j \longleftarrow 0$ based on condition
    `/* more conditions possible */`
end

Algorithm 5: Parameter tweaking based `String` patching.

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 facilitates exception-specific repairing as well. In other words, a single patch may get distributed over multiple `catch` blocks, which is achieved with the help of a constraint representation model.

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

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

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

Code 6: Example of parameter tweaking.

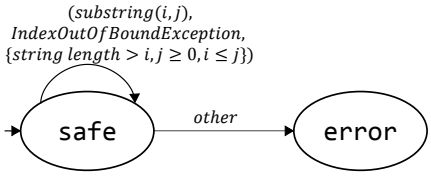


Figure 3: Partial constraint representation model.

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.

5. IMPLEMENTATION

We implemented a prototype of CLOTHO as described in § 4 for repairing runtime exceptions originating from unhandled JAVA `String` APIs. Our end-to-end toolchain is completely automated and was written in ~12.7K lines of JAVA. We leveraged the SOOT [50] framework for bytecode analysis and instrumentation, and INFOFLOW [51] for static taint analysis. We now briefly describe a few salient features of our implementation.

5.1 Taint Analysis

INFOFLOW performs 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 generated by SOOT.

5.2 Call Graph Analysis

CLOTHO leverages the SOOT generated call graph to determine both inter- and intra-method checked runtime exceptions. 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.

FORCED PATCHING. Some scenarios, like cascaded exceptions, may require CLOTHO to *force* patch the code, i.e., disable the exception analysis and patch the code irrespective of whether the exception was handled or not. A *cascaded exception* is an exception thrown in response to another exception originating from some other program point.

5.3 Constraint Analysis

CLOTHO makes a forward pass over the `Units` identified by the taint analysis and other program analysis in the first phase to gather constrains over string literals of interest, 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 described 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 5).

5.4 Optimizations

CLOTHO performs a few other optimizations to reduce total number of instrumentation which directly contributes to improved precision and quality of the patches.

(**1**) **Minimize constraint analysis** CLOTHO collects constraints only for string literals involved in a runtime exception. 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.

(**2**) **Minimize patch instrumentation** CLOTHO makes a forward pass over all program points to determine if the `String` object under consideration is modified after it has been patched. The rationale behind this is that all constraints remain valid as long as the object has not changed between program points. Similarly, when the API usage is same and none of the method parameters have changed, then no further patching would be required in later program point. Assume two statements S_1:`String x = st.subString(i,j)` and S_2:`String p = st.subString(i,j)` in two different program points and `st` gets repaired by CLOTHO before S_1. As long as `st` remains unchanged between S_1 and S_2 and the method invoked in these program points are identical along with their parameters, CLOTHO will skip instrumenting S_2.This further reduces the total number of possible instrumentation points.

6. EVALUATION

We now present an evaluation of CLOTHO. In § 6.1, we evaluate CLOTHO's effectiveness by measuring the quality of patches and related instrumentation required. We use the test suites bundled with the library itself to determine if CLOTHO generated patches violate any test case. We also measure how the several optimizations (as described in § 5.4) affect the patches generated by CLOTHO. In § 6.2, we measure the relative performance and resource penalties incurred with CLOTHO. In § 6.3, 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 libraries, and selected 30 most recent `String`-related bugs affecting the libraries, which are widely used across several products. All bugs except one were rated either major, critical or blocker. These bugs involved usage of over 64 APIs from JAVA's `String`, `StringBuffer`, `StringBuilder`, 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, 8 GB RAM and running Microsoft Windows 8.1. The JDK (v1.7) itself was provisioned with 2 GB heap space. We used INFOFLOW (snapshot from May 2014) for static taint analysis, and SOOT v2.5.0 for bytecode analysis and instrumentation.

Table 4: CLOTHO's accuracy results when applied to 30 bugs in popular open-source libraries.

$\mathcal{N}_{CG}$	# nodes in call graph	$\mathcal{F}_U$	# failed tests in unpatched	FCI	Flow Consistency Index
$\mathcal{N}_{Unit}$	# `Unit` analyzed	$\mathcal{F}_P$	# failed tests in patched version w/o forced patching	$\mathcal{IC}_{NO}$	Instrumentation w/o optimization (recall § 5.4)
$\mathcal{T}$	# total cases in test suite	$\mathcal{F}_P^*$	# failed tests in patched version w/ forced patching	$\mathcal{IC}_{WO}$	Instrumentation w/ optimization (recall § 5.4)
$\mathcal{S}_U$	# successful tests in unpatched	PPI	Patch Precision Index	$\mathcal{RS}_{CE}$	Cascaded exception exists

API	BugID	Priority	$\mathcal{N}_{CG}$	$\mathcal{N}_{Unit}$	$\mathcal{T}$	$\mathcal{S}_U$	$\mathcal{F}_U$	$\mathcal{F}_P$	$\mathcal{F}_P^*$	PPI	FCI	$\mathcal{IC}_{NO}$	$\mathcal{IC}_{WO}$	$\mathcal{RS}_{CE}$
`Aries`	[5]	Major	3.5K	129	20	18	2			0.83		42	5	
`Commons CLI1.x`	[9]	Critical	3.2K	53	16	14	2			0.74		19	19	
`Commons CLI2.x`	[8]	Major	3.2K	21	16	13	3	3	1	0.62	1	13	2	✓
`Commons Compress`	[10]	Blocker	4.0K	134	33	32	1			0.74		46	4	
`Commons IO`	[29]	Major	3.3K	125	28	27	1			0.77		76	1	
`Commons Lang`	[33]	Major	5.1K	240	18	16	2			0.59		168	8	
`Commons Math`	[37]	Major	3.4K	300	21	19	2	2		0.89	1	36	2	
`Commons Net`	[39]	Major	3.3K	14	23	22	1			0.84		6	1	
`Commons VFS`	[55]	Major	4.5K	37	19	18	1			0.65		20	2	
`Derby`	[16]	Major	4.4K	40	32	30	2			0.46		47	6	
`Eclipse AJ Weaver`	[19]	Major	20.6K	50	19	17	2	2	2	0.98		4	1	✓
`Eclipse AJ`	[18]	Major	25.0K	39	16	14	2			0.87		6	1	
`FlexDK 3.4`	[45]	Minor	6.3K	600	15	13	2			0.74		207	25	
`Hama 0.2.0`	[25]	Critical	3.7K	35	14	13	1			0.55		28	5	
`HBase 0.92.0`	[26]	Critical	4.8K	61	25	24	1			0.83		13	2	
`Hive`	[27]	Trivial	4.4K	23	19	18	1			0.75		8	1	
`HttpClient`	[28]	Major	3.3K	14	23	20	3			0.89		6	1	
`jUDDI`	[30]	Major	3.2K	70	29	28	1			0.85		10	2	
`Log4j`	[34]	Major	3.2K	17	11	8	3			0.74		6	1	
`MyFaces Core`	[38]	Major	4.5K	50	14	11	3			0.83		4	2	
`Nutch`	[40]	Major	4.5K	90	11	8	3			0.68		8	1	
`Ofbiz`	[41]	Minor	4.4K	28	23	20	3	3		0.45	1	6	1	
`PDFBox`	[42]	Major	4.4K	23	18	15	3			0.87		8	1	
`Sling Eclipse IDE`	[47]	Major	4.5K	58	6	5	1			0.59		39	6	
`SOAP`	[48]	Major	5.0K	165	21	18	3	3		0.84	1	32	5	
`SOLR 1.2`	[49]	Major	11.0K	200	14	12	2			0.89		25	4	
`Struts2`	[59]	Major	16.0K	80	13	11	2			0.76		25	2	
`Tapestry 5`	[53]	Major	6.2K	71	20	17	3			0.70		31	5	
`Wicket`	[58]	Major	70.0K	68	23	20	3			0.81		16	1	
`XalanJ2`	[60]	Major	3.3K	33	14	11	3			0.72		13	2	

6.1 Accuracy

We evaluate the precision and effectiveness of CLOTHO generated patch as described below.

(**1**) **Effectiveness of the patch**: A software patch is *effective* if it does not violate any existing test case from the software's test suite. Thus, we determined the effectiveness of CLOTHO generated patches by running them against the benchmark's existing test suites. Table 4 lists 30 real-world bugs mined from bug repositories of popular open-source libraries. Column $\mathcal{T}$ lists the total number of cases in the test suite, while $\mathcal{S}_U$ and $\mathcal{F}_U$ lists the number of successful and failed cases in unpatched version, respectively. Columns $\mathcal{F}_p$ and $\mathcal{F}_p^*$ represent the count of failed test cases without and with CLOTHO's forced patching, respectively.

We wrote a driver program to recreate the bug, and then applied CLOTHO on the library to patch it. We observed that CLOTHO without any optimizations patched 25 of the 30 offending bugs in our benchmarks, an effectiveness of over 83%. With forced patching enabled, CLOTHO successfully patches all but two benchmarks, thereby raising its effectiveness to over 93%. Note that even the force patched versions of `Commons CLI2.x` and `Eclipse AJ Weaver` fail test cases. We observed that in both benchmarks the failed test case throws a non-`String` related *cascaded* exception that CLOTHO could not patch.

(**2**) **Precision of the patch**: Precision 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 Precision Index (PPI)** as a measure of the precision of the patch.

$$PPI = \frac{\#\ Constraints_{\text{CLOTHO}}}{\#\ Constraints_{Developer}}$$

Specifically, PPI compares the similarities in constraints in CLOTHO's patch against the developer's version, thereby considering the core logic to construct an effective patch. CLOTHO analyzes and registers the constraints that are related to only `String` objects, thereby ensuring that PPI is also influenced by constraints on `String` objects alone.

If CLOTHO's patch has fewer constraints than the developer's fix, the PPI will be less than 1. In contrast, PPI greater than 1 indicates that CLOTHO generates many more constraints than those in the developer's fix. Thus, a PPI closer to 1 is desirable. However, for several benchmarks, we found the PPI value <1. This observation stems from the fact that the buggy version of the benchmark contains lesser constrains than the developer's fix version. Thus, CLOTHO analyzed fewer number of constrains in the buggy benchmark than developer's fix, thereby causing PPI < 1. PPI can also be computed automatically since CLOTHO already generates a list of constraints (in the form of bytecodes), and static analysis of the developer's patch can provide the same.

Table 4 lists the PPI for the benchmarks in our set. We note that PPI is > 0.7 for over 73% of the benchmarks. This high PPI across several benchmarks indicates the number of `String` constraints considered by CLOTHO is close to that of the developer's fix. This is a direct evidence that CLOTHO generates patch closer to program specification and hence comparable to an actual patch. PPI for the remainder of the benchmarks was observed to be lower, i.e., < 0.7. On manual investigation, we found that for some benchmarks the developers significantly changed the code structure and introduced several new sets of constraints in the patched version, which resulted in low PPI (< 0.7) for those benchmarks.

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

Table 5: Precision results for taint analysis.

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

(**3**) **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 taint analysis on the library benchmarks (for reasons 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 5 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.

(**4**) **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.

(**5**) **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 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 § 5.4, then cascaded failures may occur even for subsequent `String` APIs handling the malformed string following the point of patching.

If the above optimization is turned off, CLOTHO will automatically patch the relevant APIs and handle all cascaded failures involving malformed `String` objects. We observed that two benchmarks throw cascaded exceptions even after patching. The cascading was one level deep and triggered exception in non-`String` code (and thus unpatched), thereby causing the application to crash.

6.2 Overhead

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

(**1**) **Execution overhead**: We randomly selected and patched 5 libraries (Apache Tapestry, Apache Wicket, Eclipse AspectJ Weaver, Hive and Nutch) from Table 4 to determine the execution overhead of the patched class files. We observed that CLOTHO reports an average overhead of ~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 ~3.96μs per call. The above overhead is imperceptible at human response time scales.

(**2**) **Call graph**: The size of the call graph directly governs the time and memory consumption for CLOTHO. Figure 4a 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 ~70K nodes (for `Wicket`), CLOTHO required just 52.4s and 210MB memory.

(**3**) **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 4b 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 ~5s for collecting and evaluating the constraints.

(**4**) **Instrumentation overhead**: CLOTHO performs bytecode instrumentation for actual patching. Figure 4c shows the variation in instrumentation time with increasing number of `Units` to be patched. We observe that even without optimization discussed in § 5.4, 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 4 where column $\mathcal{IC}_{WO}$ is much less than $\mathcal{IC}_{NO}$.

6.3 Case Studies

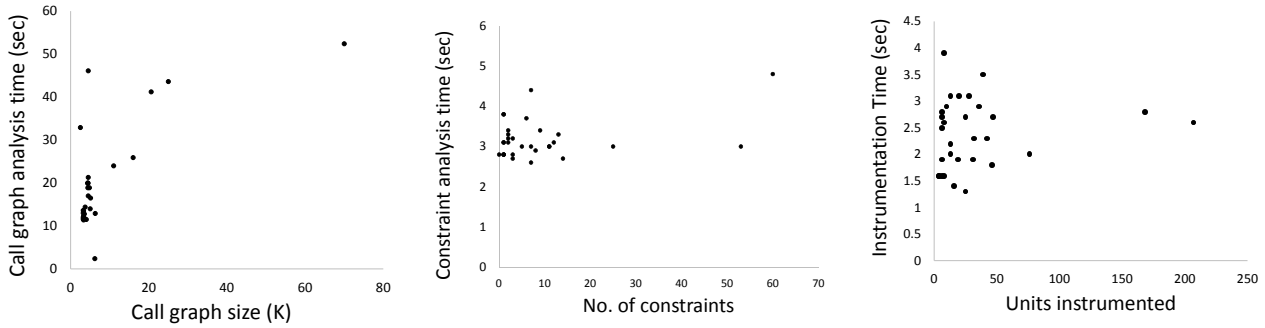
We now report on experiences gained when using CLOTHO to patch several of the bugs reported in Table 4.

(**1**) The bug [5] 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 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.

(**2**) The bug in Commons Math [37] 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.

(**3**) The bug in OfBiz [41] 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.

(**4**) 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.



(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 4: CLOTHO evaluation.

We turned off the call graph analysis module to force CLOTHO to generate the relevant patch for the bug.

(**5**) We also noticed several instances where the developer code does not follow proper programming practices regarding exception handling. For example, the SOAP bug [48] 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 for all exceptions. Thus, both the servlet exceptions and the `IndexOutOfBoundException` from the `substring` call were handled in a generic fashion. CLOTHO's patched library ensures that exceptions originating from `substring` are handled properly.

7. DISCUSSION AND FUTURE WORK

(**1**) **FOCUS ON STRING APIS.** In its present form, CLOTHO is primarily targeted towards repairing `String` objects and handling 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, which helps development of intelligent patches. Further, CLOTHO's technique is generic and can be ported to other classes of JAVA APIs. This requires extensive study of the characteristics and constraints of other object types. We leave this extension for the future.

(**2**) **PATCH CORRECTNESS.** CLOTHO attempts to generate precise patches considering the program context that avoids cascading exceptions to a great extent and producing the intended behavior in cases of failure. However, it sill cannot give guarantees about elimination of cascading exception, particularly when there are heavy object dependencies in the program. In the future, we plan to support CLOTHO with program invariants that would ensure acceptable behavior. The invariants can be specified by a programmer or can be automatically generated with the help of training runs and later used as assertion at the time of execution to ensure certain conditions remain true.

(**3**) **HANDLING OF LIMITED CONSTRAINTS.** CLOTHO's constraint data store is easy to build as it captures limited number of fairly simple `String` characteristics, which are subsequently used to generate patches for `String` objects. This approach may not be adequate particularly if the program contains a large number of complex constraints. The quality of CLOTHO's patches would generally depend on the nature of its constraint solver, which is pluggable. CLOTHO's current solver is simple and can efficiently handle a limited number of simple constraints. A more sophisticated off-the-shelf solver may improve the repair quality. However, the current evaluation of the tool on several library APIs described in § 6 indicates that the constraints that exist in practice are normally less complex and are limited in number.

8. RELATED WORK

Automated program repairing has been an active area over past decade. The approaches that have been proposed by the researchers broadly fall into two categories namely, static and dynamic.

8.1 Static Approaches

COUNTER-EXAMPLE DRIVEN. Some of the static approaches work based on a counter-example or a violated invariant that is reported from the field. These approaches then repair the program by automatically developing a patch and then ensuring its correctness using computationally intensive techniques such as model-checking [7,56]. Static techniques are effective in producing accurate patches. However, shutting down the system to produce and apply patches is not always a feasible or a desirable option. The motivation for our technique comes from the fact that for some applications fixing the bug after a program crash is not an option.

In concurrent work, Long and Rinard [35] propose a technique that takes as input a test suite consisting both, positive as well as negative test cases, and synthesizes program conditions that allow the program to pass all the test cases. The conditions are synthesized based on the values that related program variables hold during the execution of the test suite. In comparison, our technique deals with unexpected and undesirable situations that arise during a program execution, and hence, does not need a test suite.

STRING GENERATION. A key to effective program repairing is to reduce the problem by targeting a specific domain, or specific data structures such as arrays, or data types such as strings, and then use specialized transformation and solving techniques for repairing that can exploit the constrained environment.

Gulwani [24] considered Microsoft Excel programs as the use case scenario and identified string transformation as the major class of programming problem. The author designed an algorithm for learning a string expression that is consistent with input-output examples. An input-output example is generated from a mapping that maps a set of strings to a string defining a operation such as concatenation. Singh and Gulwani [46] deal with semantic transformation of a string that need to be interpreted as more than a sequence of characters. Their approach may complement our approach by improving our string generation process.

Samimi et al. present a tool *PHPQuickFix* based on the static analysis approach and that targets a special case of printing related bug in HTML [44]. Compared to their technique our approach addresses much wider range of problems related to Java strings and can be extended to other systems that have support for exceptions. Tatlock et al. focus on the JAVA database API and related query string [54]. They propose a solution of type errors that is caused due to a type mismatch in the database and the type assigned in the program. The authors also propose a solution for refactoring where changing a class name associated with some queries will reflect all the strings system wide. This approach is based on type-checking and addresses different class of repairing problems.

STATIC DATA REPAIR. Demsky et al. propose repair strategy by goal-directed reasoning [15]. This involves translating the data-structure to an abstract model by a set of model definition rules. The actual repair involves model reconstruction and statically mapping it to a data structure update. Elkarablieh et al. propose an idea to statically analyze a data structure to access the information like recurrent fields and local fields [20]. They use their technique to some well known data structures like singly linked list, sorted list, doubly linked list, N-ary tree, AVL tree, binary search tree, disjoint set, red-black tree, and Fibonacci heap. Overall, these approaches are complementary to our approach.

8.2 Dynamic Approaches

Static approaches cannot exactly predict when the failure can happen. However, dynamic approaches have a complete visibility to the program under execution and can have a precise knowledge of its state. In order to overcome some limitations of static approaches and exploit the execution knowledge, several promising dynamic approaches have been proposed.

DYNAMIC DATA REPAIR. Demsky and Rinard present techniques that mostly concentrate 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 [11–14]. The authors present a specification language using that they detect and check consistency properties in these data-structures and repair them. For a violation, they replace the error condition with correct proposition. These approaches develop either suboptimal patches or isolate the data structure that is damaged that allows at least part of the system to be functional. In contrast, our approach tries to provide support to the entire application in the context of string-related failures by trying to keep the system behavior as close as possible to the intended. The advantage of these approaches is that they are light-weight and like our approach, can fix a system on-the-fly potentially allowing some sub-optimal behavior for a finite time until the systems self-stabilizes.

Long et al. [36] present a technique that repairs a crashing system *RCV* on-the-fly. The technique deals with two types of system violations, namely, divide-by-zero and null-deference errors. Their tool replaces *SIGFPE* and *SIGSEGV* signal handler with its own handler. The approach 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 is on $x86$ and $x86-64$ binaries. They also implement a dynamic taint analysis to see the effect of their patching until the program stabilizes that they called as *error shepherding*. However, unlike CLOTHO, this technique requires a special runtime that is built into the operating system.

GENETIC PROGRAMMING. The researchers have also tried genetic programming approach for repairing. Goues et al. use genetic programming technique that is a stochastic search method inspired by biological evolution [23]. The technique generates program patch by using already existing test cases to deal with bugs such as infinite loop, null string, segmentation fault, and buffer overflow. Similar approach has been presented by Weimer et al. [57]. Goues et al. present study to understand the real life feasibility and the fraction of the bugs their genetic programming-based tool can repair as well as the cost associated with it [22]. These approaches are test-driven and rely on the availability and completeness of test-suites. In contrast, our approach is completely independent of a test-suite.

STRINGS RELATED. Samimi et al. present a dynamic repairing technique for PHP applications [44]. They present *PHPRepair*, a tool to repair auto-generated malformed HTML codes from the PHP scrips. Often the HTML codes do not have proper tags that are silently corrected by the browser but the result is different across browsers. The authors employ an efficient SAT solver using cost optimization to find an efficient repair.

OTHER DYNAMIC APPROACHES. Perkins et al. present a system named *Clear view* that works on windows x86 binaries without requiring any source code [43]. They use invariants analysis for that they use Daikon [21]. They mostly patched security vulnerabilities by some candidate repair patches. Unlike CLOTHO, *Clear view* requires a test-suite to develop the knowledge about the invariants.

8.3 CLOTHO's Main Contrasting Features

Our approach targets only string objects for repairing allowing it generate highly precise and intelligent program patches that generate very few or none cascading exceptional events and produces a program behavior that is very close to the expected behavior under the event of crashing. In addition, our approach is hybrid with a heavy static component that enables all the analysis including the side-effect analysis based on a taint analysis to perform dynamically. It incurs negligible overhead even in the event of crashing. Moreover, CLOTHO works at the application level, and hence is easily portable.

Many of the repairing approaches as described above have been found to be effective on various problems that are scoped appropriately. We believe that ours is the first generic approach that targets string objects and repairs program on-the-fly by generating precise patches using the properties of string objects and leveraging contextual information.

9. CONCLUSION

Running programs may crash unexpectedly due to vulnerabilities in the code and malformed data. The cost associated with such crashes can vary with the criticality of the applications. In this work, we present a novel program repairing technique, and a prototype 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 created by the developers. Hence, when the patches are activated, the program exhibits a behavior close to the intended program behavior. Our evaluation shows that CLOTHO can handle programs that are real, and can produce patches efficiently.

10. ACKNOWLEDGEMENTS

We would like to thank the anonymous reviewers for their valuable comments. We are also grateful to Steven Arzt for his help with SOOT related issues in CLOTHO.

11. REFERENCES

[1] Amazon sellers hit by nightmare before Christmas as glitch cuts prices to 1p. http://www.theguardian.com/money/2014/dec/14/amazon-glitch-prices-penny-repricerexpress.

[2] Nike Rebounds: How (and Why) Nike Recovered from Its Supply Chain Disaster. http://www.cio.com/article/2439601/supply-chain-management/nike-rebounds--how--and-why--nike-recovered-from-its-supply-chain-disaster.html.

[3] Supply Chain: Hershey's Bittersweet Lesson. http://www.cio.com/article/2440386/supply-chain-management/supply-chain---hershey-s-bittersweet-lesson.html.

[4] When Bad Things Happen to Good Projects. http://www.cio.com/article/2439385/project-management/when-bad-things-happen-to-good-projects.html.

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

[6] Michael Carbin, Sasa Misailovic, Michael Kling, and Martin C. Rinard. Detecting and escaping infinite loops with jolt. In *Proceedings of the 25th European Conference on Object-oriented Programming*, ECOOP'11, pages 609–633, Berlin, Heidelberg, 2011. Springer-Verlag.

[7] Pavol Cerny, ThomasA. Henzinger, Arjun Radhakrishna, Leonid Ryzhyk, and Thorsten Tarrach. Regression-free synthesis for concurrency. In Armin Biere and Roderick Bloem, editors, *Computer Aided Verification*, volume 8559 of *Lecture Notes in Computer Science*, pages 568–584. Springer International Publishing, 2014.

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

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

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

[11] Brian Demsky, Michael D. Ernst, Philip J. Guo, Stephen McCamant, Jeff H. Perkins, and Martin C. Rinard. 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*, pages 233–244, 2006.

[12] Brian Demsky and Martin Rinard. 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.

[13] Brian Demsky and Martin C. Rinard. Automatic detection and repair of errors in data structures. In *Proceedings of the 2003 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications, OOPSLA 2003, October 26-30, 2003, Anaheim, CA, USA*, pages 78–95, 2003.

[14] Brian Demsky and Martin C. Rinard. 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*, pages 71–84, 2003.

[15] Brian Demsky and Martin C. Rinard. Data structure repair using goal-directed reasoning. In *27th International Conference on Software Engineering (ICSE 2005), 15-21 May 2005, St. Louis, Missouri, USA*, pages 176–185, 2005.

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

[17] Robert Dyer, Hridesh Rajan, Hoan Anh Nguyen, and Tien N. Nguyen. A Large-scale Empirical Study of Java Language Feature Usage. http://lib.dr.iastate.edu/cgi/viewcontent.cgi?article=1284&context=cs_techreports.

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

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

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

[21] Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. The daikon system for dynamic detection of likely invariants. *Sci. Comput. Program.*, 69(1-3):35–45, 2007.

[22] Claire Le Goues, Michael Dewey-Vogt, Stephanie Forrest, and Westley Weimer. A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. In *34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland*, pages 3–13, 2012.

[23] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. Genprog: A generic method for automatic software repair. *IEEE Trans. Software Eng.*, 38(1):54–72, 2012.

[24] Sumit Gulwani. Automating string processing in spreadsheets using input-output examples. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '11, pages 317–330, New York, NY, USA, 2011. ACM.

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

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

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

[28] 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.

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

[30] JUDDI-292. <faultstring>string index out of range: 35</faultstring>. https://issues.apache.org/jira/browse/JUDDI-292, 2011.

[31] Kiyokuni Kawachiya, Kazunori Ogata, and Tamiya Onodera. A Quantitative Analysis of Space Waste from Java Strings and its Elimination at Garbage Collection Time. http://domino.watson.ibm.com/library/cyberdig.nsf/papers/F2BBE159220ADDF3852573990006DBF2/$File/RT0750.pdf.

[32] Kiyokuni Kawachiya, Kazunori Ogata, and Tamiya Onodera. Analysis and reduction of memory inefficiencies in java strings. In *Proceedings of the 23rd ACM SIGPLAN Conference on Object-oriented Programming Systems Languages and Applications*, OOPSLA '08, pages 385–402, New York, NY, USA, 2008. ACM.

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

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

[35] Fan Long and Martin Rinard. Staged Program Repair in SPR. http://dspace.mit.edu/handle/1721.1/95963.

[36] Fan Long, Stelios Sidiroglou-Douskos, and Martin C. Rinard. 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*, page 26, 2014.

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

[38] MYFACES-416. Stringindexoutofboundsexception in addresource. https://issues.apache.org/jira/browse/MYFACES-416, 2005.

[39] 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.

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

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

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

[43] Jeff H. Perkins, Sunghun Kim, Samuel Larsen, Saman P. Amarasinghe, Jonathan Bachrach, Michael Carbin, Carlos Pacheco, Frank Sherwood, Stelios Sidiroglou, Greg Sullivan, Weng-Fai Wong, Yoav Zibin, Michael D. Ernst, and Martin C. Rinard. 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*, pages 87–102, 2009.

[44] Hesam Samimi, Max Schäfer, Shay Artzi, Todd D. Millstein, Frank Tip, and Laurie J. Hendren. Automated repair of HTML generation errors in PHP applications using string constraint solving. In *34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland*, pages 277–287, 2012.

[45] 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.

[46] Rishabh Singh and Sumit Gulwani. Learning semantic string transformations from examples. *Proc. VLDB Endow.*, 5(8):740–751, April 2012.

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

[48] SOAP-130. String indexoutofbounds in soapcontext. https://issues.apache.org/jira/browse/SOAP-130, 2004.

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

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

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

[52] StackOverflow. Stack exchange data dump : Stack exchange, inc. : Free download & streaming : Internet archive. https://archive.org/details/stackexchange, 2013.

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

[54] Zachary Tatlock, Chris Tucker, David Shuffelton, Ranjit Jhala, and Sorin Lerner. Deep typechecking and refactoring. *SIGPLAN Not.*, 43(10):37–52, October 2008.

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

[56] Yi Wei, Yu Pei, Carlo A. Furia, Lucas S. Silva, Stefan Buchholz, Bertrand Meyer, and Andreas Zeller. Automated fixing of programs with contracts. In *ISSTA 2010: Proceedings of the 19th international symposium on Software testing and analysis*, pages 61–72, New York, NY, July 2010. ACM.

[57] Westley Weimer, Stephanie Forrest, Claire Le Goues, and ThanhVu Nguyen. Automatic program repair with evolutionary computation. *Commun. ACM*, 53(5):109–116, 2010.

[58] WICKET-4387. Stringindexoutofboundsexception when forwarding requests. https://issues.apache.org/jira/browse/WICKET-4387, 2012.

[59] WW-650. Cooluriservletdispatcher throws stringindexoutofboundsexception. https://issues.apache.org/jira/browse/WW-650, 2005.

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