TOOL: Program Repairing using Exception Types, Constraint Automata and Typestate

Abstract

Changes done

Runtime Exceptions are common types of exceptions which may lead to system crash which leads to shutdown or restart. For may critical application such scenario is unacceptable due to their nature which requires availability of the service. Program bugs which causes runtime exceptions often go unnoticed at the time of development as these exceptions are unchecked exceptions. The key issue is to guide the program through some exception suppression procedure which will leads the program to a consistent state hence improve the chance of surviving a fatal crash. Here we consider such programs for which restart is not an option.

In this paper, we present a novel technique to recover from unexpected runtime exceptions. We have used hybrid of two techniques for efficient detection of potential point of failure and patch it closest to that to minimize the damage. One technique uses type of runtime exception to apply appropriate patch. The other technique will provides typestate analysis technique which will detect typestate violations to apply the right patch.

1. Introduction

Changes done

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

(1) Checked exception which requires explicit `throws` declaration at the method declaration or `try-catch` block by the developers. Such exceptions are handled carefully as they often involves accessing resources like network, database, file system, I/O etc.

(2) Unchecked exception which does not enforce similar handing mechanism as the former one. `java.lang.RuntimeException` and its subclasses and *java.lang.Error* are types of unchecked exceptions. `NullPointerException`, `ArrayIndexOutOfBound`, `ArithmeticException` are examples of common JAVA runtime exceptions.

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

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

Many time bad or malicious code leads to some vulnerability to critical applications and website which can be exploited by attack to orchestrate system crash. Thought these examples cover a large variety of applications, all of them point to some concern of *availability*.

Usually, developers tests their code in series of verifications which involves code review, static and dynamic analysis of the code, generate test cases to cover as much potential input .Yet may corner cases can be left overlooked which can cause runtime exceptions. Multi-threaded applications are also susceptible to erroneous thread interleaving. One such exception is `java.lang.IllegalMonitorStateException`, when a thread has attempted to wait on an object's monitor or to notify other threads waiting on an object's monitor without owning the specified monitor. Applications under adversarial situation should be considered where deliberate malicious input may cause it to fail. To recover from such situation, a mechanism is needed which can predict failure by doing invariant and symbolic analysis. Invariant analysis will detect particular variables outside legal/safe bound. Symbolic analysis will indicate to the potential point of failure.

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

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

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

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

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

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

Runtime Exception Type	Frequency	%age
`NullPointerException`	34912	54.94
`ClassCastException`	7504	11.81
`IndexOutOfBoundsException`	6637	10.44
`SecurityException`	5818	9.15
`NoSuchElementException`	2392	3.76
`ArithmeticException`	2338	3.67
`ConcurrentModificationException`	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
`MalformedParameterizedTypeException`	38	0.05
`CMMException`	8	0.01
`FileSystemNotFoundException`	6	0.009
`NoSuchMechanismException`	3	0.0045
`MirroredTypesException`	1	0.0015

Table 1: Most frequent runtime exceptions from stack overflow

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.

(**1**) In USS Yorktown, complete failure in propulsion and navigation system by a simple divide-by-zero exception in flight deck database.

(**2**) AT&T telephone network failure causing by one faulty switch causing ATC commutation blackout.

(**3**) Air-Traffic Control System in LA Airport lost communication with all 400 airplanes caused by a system crash triggered by integer (32bit) overflow.

(**4**) Mars rover curiosity B-side computer memory overflow causing OS suspend and multiple restart.

(**5**) Trans-Siberian Gas Pipeline Explosion in 1982 by deliberate bugs in software controlled valves.

(**6**) Near-blackout of the national grid in Austria caused by faulty function call.

2.2 Data from Stack Overflow Posts

We have analyzed data from stack overflow and we looked for JAVA runtime exception which are discussed most fre-

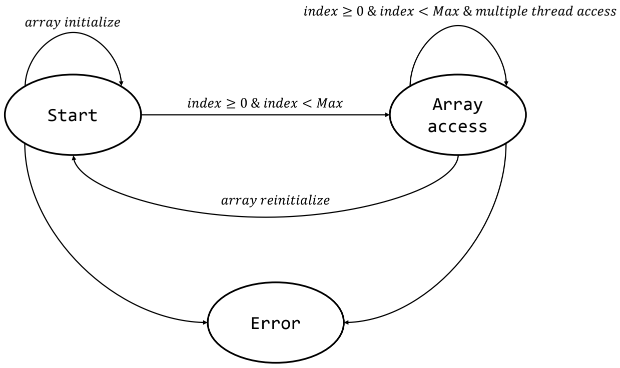

Figure 1: array index out of bound formulated as FSM

quently. In the table 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.

3. Problem Formulation

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

We formulate the problem in following way

3.1 Runtime Exceptions

We can visualize all runtime exceptions as finite state machine (FSM). When a program violates such sequence, it throws runtime exception. In Figure 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 $array\_index \leq max\_array\_size - 1$

4. Repairing Strategy: Exception Type

Please review this section

In the Example 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.

4.1 Symbolic Analysis

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.

(**2**) Symbolic analysis of the program to discover potential points of failure and mark them.

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

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

Listing 1: Java code which may throws runtime exceptions

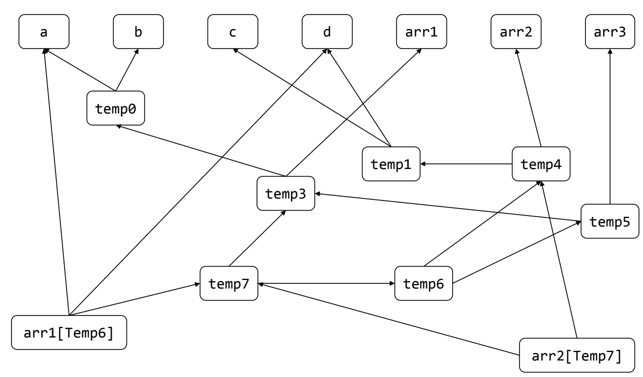

Figure 2: Data dependency graph of the variables in Example1

(**3**) Build data dependency graph which will be used to generate appropriate code slice to be used as patch. In Figure 2, the data dependency graph of the example code 1 is presented.

(**4**) The symbolic 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.

Global variables and parameters

a	b	c	d	arr1	arr2	arr3
<snapshot>	...	...	...	...	...	...

Successful runs

Figure 3: Indexed global variables and method arguments successful runs

4.2 Data set for Successful Program Runs

Here we will store all the traces of successful program runs.

Figure 3 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.

4.3 Matrices

Please review this section.

4.4 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

4.4.1 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.

4.4.2 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 4.1. In the Listing 2, the example code snippet shows such code slice inside the catch block. As the error occurred at the line *int temp*5 = *this.arr*3[*temp*3]; the statements which produces the temp3 and the statement which also involves *temp*3 or any other variables derived from *temp*3, would be included in the catch block for re-execution with the valued of the same from the data table of previous successful runs.

4.5 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.

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

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

Listing 2: patching code slice based on exception type

5. Repairing Strategy : Constraint Automata

5.1 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.

Figure 4: Constraint automata general model

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

Listing 3: array index out of bound patching

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

Listing 4: arr index out of bound patching

- q_0: starting state, here $q_0 = init$.
- F: end state, here it same as q_0.

According to the Figure 4, the repairing mechanism will only trigger when we have a transition from init state to error state due to invariant violation.

5.2 Patching Techniques

The patching technique is based on the exception type.

5.2.1 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 $array.length - 1$.

5.2.2 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.

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

Listing 5: arithmetic exception : division-by-zero patching

5.2.3 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.

5.2.4 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 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.
-
- **Case II** Possible Accessing or modifying the field of a null object.
  Patch : The patch is same as the previous one.
- **Case III** Taking the length of null as if it were an array.
  Patch : The patch for this situation is very much similar to the negative array size exception. Here we will do a dataflow 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.

```java
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;
   } catch(NullPointerException ex) {
      //instrumentation
      //choose appropriate constructor
      mlass = new MyClass(1, "a", 1.0);
      a = mclass.getfiled3();
   }
}
```

Listing 6: appropriate constructor

```java
int[] bar(int a) {
   int []arr = new int[a];
   int []b = (a > 10) ? arr:null;
   return b;
}
void foo() {
   int[] arr;
   int []arr = bar(5);
   try {
      //access or modify any field of arr
      //this will throw a null pointer exception
   } catch {
      //instrumented code
      int ARRAY_SIZE = 11;
      int []arr = new int[ARRAY_SIZE];
      //access or modify any field of arr
   }
}
```

Listing 7: array null pointer exception

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

6. Design of the System

The overall design of the repairing framework is illustrated in Figure 5. The framework consists of two basic modules.

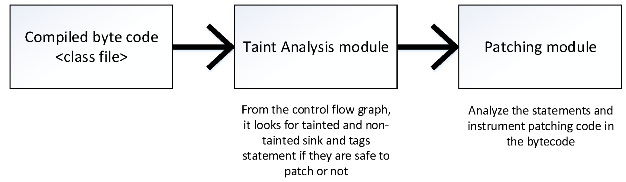

Figure 5: Overall Design

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

Table 2: Common JAVA library taint source functions

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

Table 3: Common JAVA library taint sink functions

6.1 Taint analysis Module

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

We have specify the list of source, sink and derivation methods in a configuration file before the analysis. The source methods includes methods which take input from user from console or web application forms like text box. The sink methods are sensitive data storage which are unsafe to manipulate such as database, console print or methods to send a text file to printer etc. The overview of the taint analysis module is illustrated in the Figure 6. The input for the module is the compiled byte code intended to be repaired. Here we have generated a control flow graph (CFG) from the class file to get all the possible program paths. Here a point to be noted that any modification along the path going to the tainted sink is unsafe to patch.

6.1.1 Tainting RulesNeeds Revision

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

(**1**) We defined list of source and sink taint methods listed in Table 2 and 3. We are only tainting the variables which are coming from the listed taint source methods.

(**2**) We have also listed all taint propagation methods. The assignment (=) is the basic taint propagator. But there are

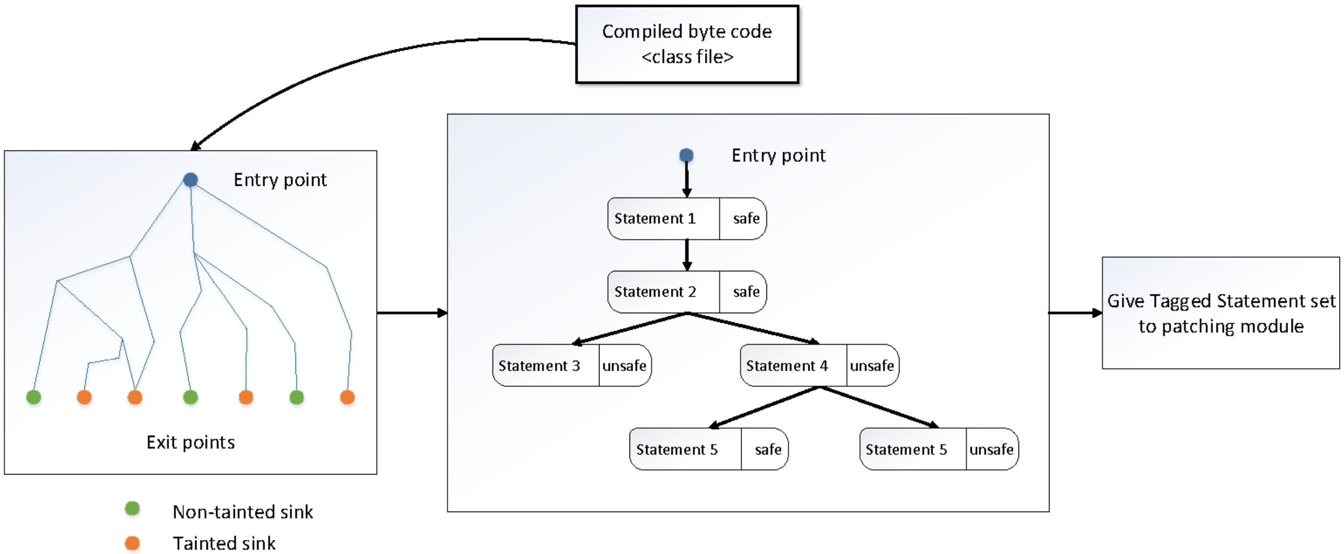

Figure 6: Design of the Taint Module

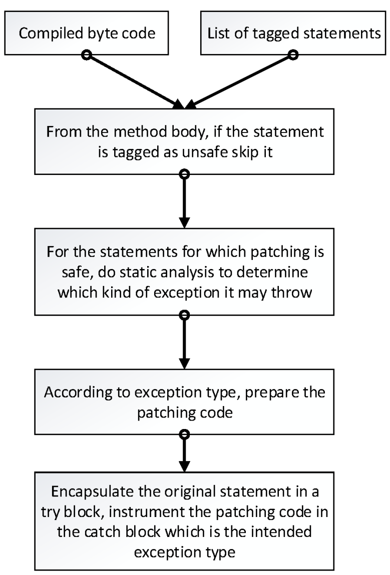

Figure 7: Design of the Patching Module

other methods like `append` in `java.lang.StringBuffer` and `java.lang.StringBuilder` which are taint propagator.

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

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

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

Listing 8: Same method calling in different scenario

6.2 Repairing Module

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

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

In the Listing 8 and 9 we have defined an example code snippet of the original code and the patched code where we have renamed the method *bar* to *bar_untainted_fa844d57* before instrumenting any patching code in it. The variable *out* goes to a tainted sink while *out1* does not. So the we have done modification in the line where *out1* is defined. As *out* is going to a tainted sink method, we did not do any modification to it.

```
int bar(int a, int b) {
    return a/b;
}
int bar_untainted_fa844d57(int a, int b) {
    int out;
    try {
        out = a/b;
    } catch(ArithmeticException ex) {
        b = 1;
        out = a/b;
    }
    return out;
}
void foo() {
    int a = 10, b = 0, c = 15;

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

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

Listing 9: Method name modification for different calling context

7. Benchmark Results

todo

8. Related Works

8.1 Recent Works on Data Structure Repairing

Automated data-structure repairing techniques are there in the literature for a while. In the papers [Demsky and Rinard 2003c,a, 2005, 2003b; Demsky et al. 2006] 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 [Demsky and Rinard 2005], 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 [Elkarablieh et al. 2007] 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.

8.2 Works on Software Patching

In their paper [Perkins et al. 2009], 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 [**?**]. They mostly patched security vulnerabilities by some candidate repair patches.

Fan Lon et al in their paper [Long et al. 2014] 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*.

8.3 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. [Forrest et al. 2009] and Westley Weimer et al [**?**] 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.

9. Conclusion and Future Works

References

B. Demsky and M. 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*, 2003a.

B. Demsky and M. 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. IEEE Computer Society, 2003b. ISBN 0-7695-2007-3. doi: 10.1109/ISSRE.2003.1251032. URL http://doi.ieeecomputersociety.org/10.1109/ISSRE.2003.1251032.

B. Demsky and M. C. Rinard. Automatic detection and repair of errors in data structures. In R. Crocker and G. L. S. Jr., editors, *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. ACM, 2003c. ISBN 1-58113-712-5. doi: 10.1145/949305.949314. URL http://doi.acm.org/10.1145/949305.949314.

B. Demsky and M. C. Rinard. Data structure repair using goal-directed reasoning. In G. Roman, W. G. Griswold, and B. Nuseibeh, editors, *27th International Conference on Software Engineering (ICSE 2005), 15-21 May 2005, St. Louis, Missouri, USA*, pages 176–185. ACM, 2005. doi: 10.1145/1062455.1062499. URL http://doi.acm.org/10.1145/1062455.1062499.

B. Demsky, M. D. Ernst, P. J. Guo, S. McCamant, J. H. Perkins, and M. C. Rinard. Inference and enforcement of data structure consistency specifications. In L. L. Pollock and M. Pezzè, editors, *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2006, Portland, Maine, USA, July 17-20, 2006*, pages 233–244. ACM, 2006. ISBN 1-59593-263-1. doi: 10.1145/1146238.1146266. URL http://doi.acm.org/10.1145/1146238.1146266.

B. Elkarablieh, S. Khurshid, D. Vu, and K. 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. doi: 10.1145/1297027.1297056. URL http://doi.acm.org/10.1145/1297027.1297056.

S. Forrest, T. Nguyen, W. Weimer, and C. L. Goues. A genetic programming approach to automated software repair. In *Genetic and Evolutionary Computation Conference, GECCO 2009, Proceedings, Montreal, Québec, Canada, July 8-12, 2009*, pages 947–954, 2009. doi: 10.1145/1569901.1570031. URL http://doi.acm.org/10.1145/1569901.1570031.

F. Long, S. Sidiroglou-Douskos, and M. C. Rinard. Automatic runtime error repair and containment via recovery shepherding. In M. F. P. O'Boyle and K. Pingali, editors, *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, page 26. ACM, 2014. ISBN 978-1-4503-2784-8. doi: 10.1145/2594291.2594337. URL http://doi.acm.org/10.1145/2594291.2594337.

J. H. Perkins, S. Kim, S. Larsen, S. P. Amarasinghe, J. Bachrach, M. Carbin, C. Pacheco, F. Sherwood, S. Sidiroglou, G. Sullivan, W. Wong, Y. Zibin, M. D. Ernst, and M. C. Rinard. Automatically patching errors in deployed software. In J. N. Matthews and T. E. Anderson, editors, *Proceedings of the 22nd ACM Symposium on Operating Systems Principles 2009, SOSP 2009, Big Sky, Montana, USA, October 11-14, 2009*, pages 87–102. ACM, 2009. ISBN 978-1-60558-752-3. doi: 10.1145/1629575.1629585. URL http://doi.acm.org/10.1145/1629575.1629585.