笔记模板2

1. 文章解决的问题

文章解决了整数溢出的问题，技术基于静态符号执行，融合检测以及修复生成与验证。

采用完全静态的方法（即不需要有触发整数溢出的输入，就可以检测出错误以及去修复。第一种不需要测试用例却可以修复整数溢出的方法），一个特点就是用SMT求解将故障定位和修复生成融合到单个算法中。

而且最后由程序员本人决定是否插入这个补丁

文章的贡献：

- 对C语言的整数溢出设计了一种新的修复技术以及技术的实现

文章采用SMT主要是为了三点:1. 检查可满足或不可满足的程序执行路径　2. 为了检查整数溢出是否存在，采用额外的SMT约束添加到SMT约束系统中，这个系统用来检查程序执行路径，在检查路径可满足性的同时检查整数溢出 3. 为了检查是否消除了整数溢出，使用额外的SMT约束。

2. 解决的思路

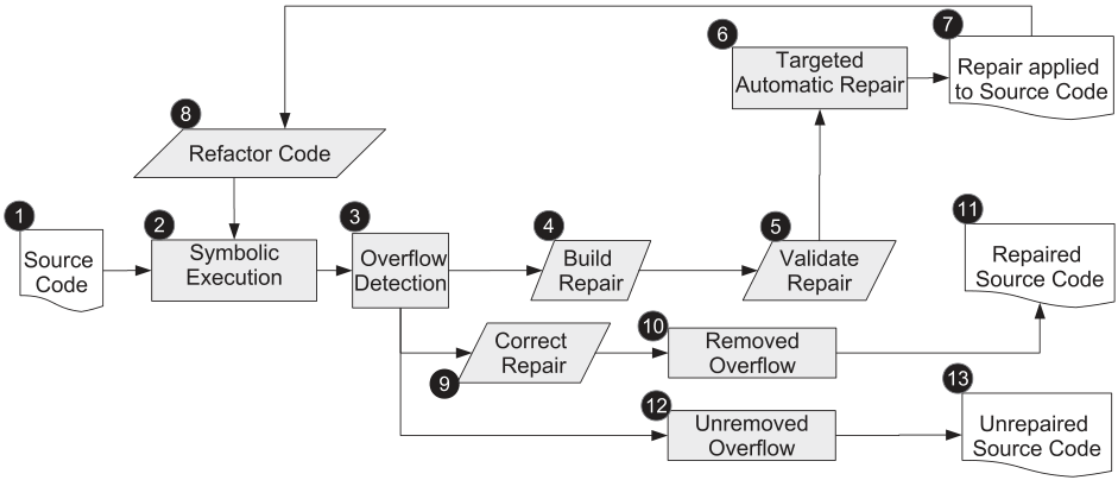
1. 溢出检查：
   - 第一条件：s_1为变量，s_2为正常量，当满足以下条件时，就被认为整数溢出。
     $s_1 > 0 \wedge （ s_1 > INT - MAX - s_2）$。
   - 第二条件：s_1为变量，s_2为负数，以下条件为真时，则溢出：
     $((s_1 > 0) \wedge (s1 > (INT - Min/S_2))) \vee ((s_1 < 0) \wedge (s_1 < (INT - Max/s_2)))$
   - 第三条件：两个相等的整数相乘：
     $((s_1 > 0) \wedge (s1 > sqrt(INT - MAX))) \vee ((s_1 < 0) \wedge (s_1 < (-sqrt(INT - MAX))))$

   整数的精度可以调整，就是说INT-MAX可以改变。而且s_1、s_2可以采用不同的类型，char、int、short等

3. 核心知识点或名词定义

1. INTREPAIR基于Codan静态符号执行引擎实现，这个引擎使用Z3 SMT解算器来解决约束。
2. Valid Repair 有效的修复r: r是一个补丁，v 是一个oracle，I是不触发错误的输入集合，B是程序的行为的集合，那么r 不能 改变I对应的B，但是r改变了I以外的B也就是说修复程序没有把正确改变，但是把非正确的行为改变了
3. Correct Repair：Q是程序可以到达的路径的集合，r修复了f，且没有引入其他错误，文中使用符号执行来区分有效和正确的输入
4. CFG：控制流图
5. SMT方程切片：对于一条语句会有对应的SMT方程，将其中的部分或一个符号变量成为切片

4.程序功能说明

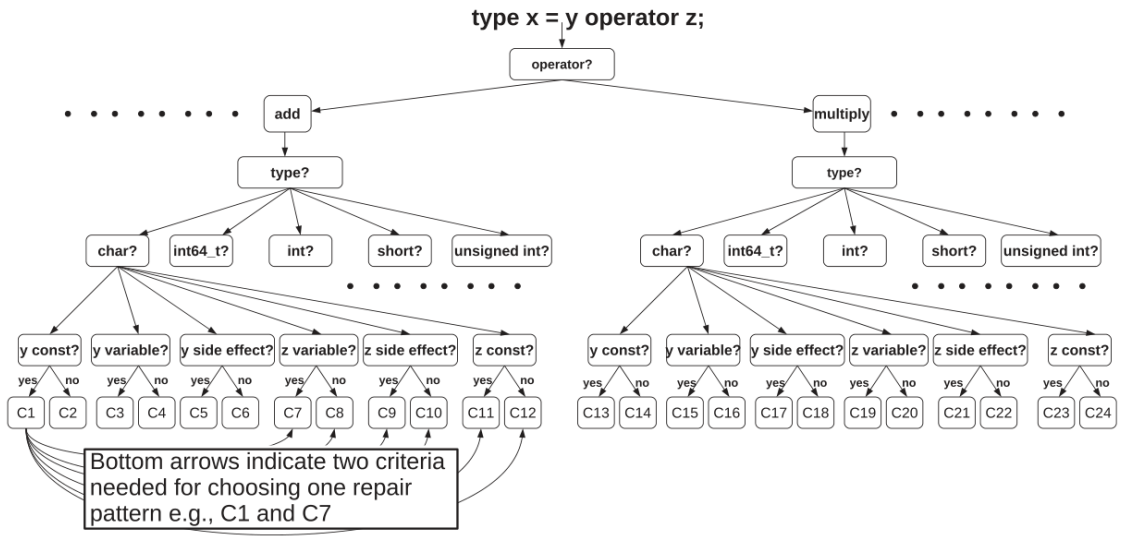


1. Overflow Detection: 首先会构建程序的CFG,然后执行以下步骤。
    - 提取CFG中的路径，在分支节点上检查路径的可满足性。
    - 当IntRepair在路径上遇到溢出的易发位置（如，赋值语句），就会用解释器执行溢出检查
    - 其实解释器会通知相应的检测器进行检查
    - 检查器检查可能溢出的符号变量以及相应的整数溢出可满足性检查。（如何检查出来的）就是说检查符号变量是否大于当前使用的整数上限值，这里的上限值都是从limits.h file（Linux的一个系统文件）提取出来的
    - 一旦条件为真，报告溢出。下限值是通过对当前使用的上限值求反将结果加1得到的
    - 用的是SMT解算器，生成一份问题报告（问题ID，ID指的是哪个检查器检测到的故障； 文件名词以及错误位置的行号）

    这里还有一个优化的步骤：IntRepair具有回溯功能，如果当前的路径不能满足即不能到达，则退回到前一个分支节点，对条件取反，执行新的路径

2. Repair Patterns

    修复模式存储在决策树中，决策树由作者手动构建的。



3. Decision tree used by IntRepair for storing and retrieving repair patterns.

最后的节点：1. y const:y是否是常量　　2. y variable:y是否是变量　　3 y side effect:y是否有副作用（比如int z = i++，或者int y = foo()）

如何选择修复模式？比如y对应C1~C6中的一种，z对应C7到C12中的一种，那么将它们进行组合，则Char下有36种。则这个图已经描绘了360个修复模式。

举例四个常见的修复模式

TABLE 1
Four Repair Patterns of IntRepair

Criteria	Statement	Repair Pattern Format	Description
C15 & C19	`char a=y*y;`	`if(y>sqrt(INT_MAX)|| y < -sqrt(INT_MAX)){ log_or_die();} else{...}`	multiply two equal variables
C15 & C17	`char a=y*3;`	`if(y > INT_MAX/3 || y < INT_MIN/3){ log_or_die();} else{...}`	multiply a variable with a constant
C3 & C7	`char a=y+z;`	`if(y > INT_MAX - z || y < INT_MIN - z){ log_or_die();} else{...}`	add two variables
C3 & C11	`char a=y+4;`	`if(y > INT_MAX - 4 || y < INT_MIN - 4){ log_or_die();} else{...}`	add a variable with a constant

当确定这条语句存在整数溢出，那么就用这个模板来代替这条语句

3. build repair：

    1. 确定整数的上限值：文章中说了将错误定位与修复是一起进行的，所以整数上限值不产生冲突
        - 首先确定每个类型的硬件溢出限制
        - 然后基于符号执行的路径遍历过程中，将当前使用的变量与支持的整数上限值（CHAR_MAX、INT_MAX、LLONG_MAX、SHORT_MAX、and UINT_MAX）进行比较。如果匹配成功，则将它设置为当前使用的整数上限值。
    2. 生成SMT约束系统

        接下去将整数溢出检查器中使用的符号变量和约束组合在一起，并且存储起来

        存储的信息为 举例：int result = a + b

        1. 存在整数溢出的语句（int result = a + b）
        2. 检测故障的SMT公式（(assert (= resSymbolic (+varAsymbolic varBsymbolic))））
        3. 用于检测这个对应故障的检查器的故障ID（ID-Integer_Overflow_Fault）
        4. 用于检测整数溢出的符号变量（resSymbolic）
        5. 整数溢出可能依赖的其他符号变量varAsymbolic
    3. 选择约束的变量

        根据检测到的语句类型，选择SMT约束变量（result、潜在的变量a，b），其中result会被进一步约束，这是为了之后可以检测生成的修复是否删除了整数溢出的故障

    4. 重新计算绑定检查约束：加强约束（怎么加强：通过程序路径执行能够得到第三步确定的变量的新的约束），将得到的约束添加到SMT约束系统，用Z3求解器来确定是否出现整数溢出
    5. 决定错误类型

        根据步骤2中保存的3来确定故障类型（故障ID）

    6. 选择修复模式

        使用决策树来选择合适的修复

        将一条语句中选择三个组件（操作、左操作数和右操作数），比如“ +” 、a 、b。

        然后会有一系列的规则：1. 左右操作数是否一致 2. 操作是什么等一系列规则

        基于上述规则，产生最大数量约束的修复模式，这些约束用来选择不同的修复模式，通过在决策树中进行组合以及分支遍历来实现的

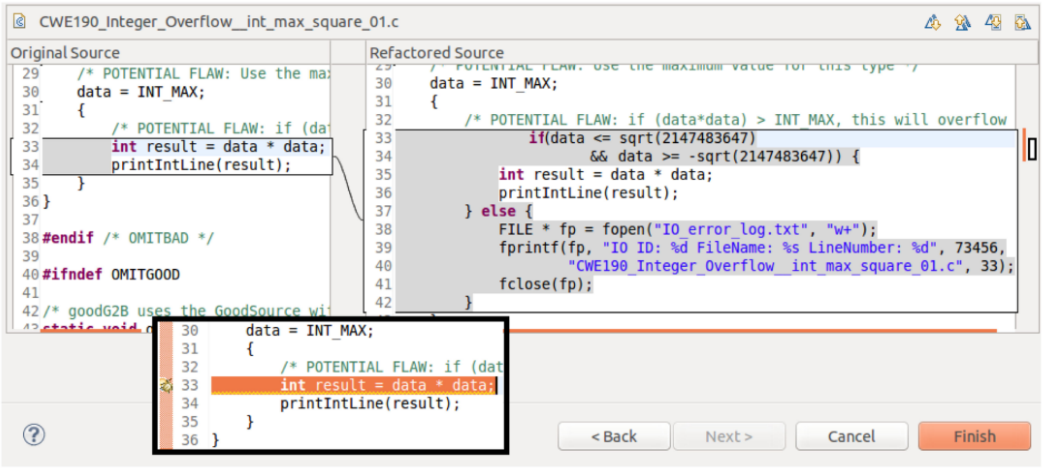
```
if(leftHandSide.equals(
rightHandSide)&&
operator.equals("*")) {
...
return "if(sqrt("+leftHandSide+") <= sqrt("+value4+") &&
       -sqrt("+ rightHandSide+") >= -sqrt("+value5 +"))
        "+"{"+"\n"+"\t"+"\t"+"\t"
          +faultyStm10 + "\n }else{ \n"
          +"FILE *fp=fopen (\"IO_"
          +"fault_log.txt\", + \"w+\");"
          +"\n fprintf(fp, \"IO_ID:%s
          +FileName:%s LineNumber:%d",
          +FileName+","+IO_fault+","
          +LineNumber+");"
          +"\n fclose(fp);" + " }\n";
}
```

Fig. 4. Repair pattern example.

value4为当前选择的整数上限值的平方根，所以sqrt(leftHandSide)<=sqrt(value4)跟leftHandSide<=value4的值是一致的。return语句后面的是要加到bug程序里的语句

修复模式基于之前的溢出检查一样的条件。

7. 创建新的SMT约束系统：这一步生成新的约束系统，用来检测修复后是否还会存在整数溢出
8. 实际的修复



5. 存在的问题

1. Codan只能支持C、C++，
2. 无法处理复杂的C语言结构
3. 面对循环与递归的问题时，IntRepair会产生错误的报告

6. 改进的思路

7. 想法来源