LeakPlug: 一个内存泄漏自动修复工具

杨至轩　　冯致远
1300012785　　Lost Connection

December 31, 2016

1　问题描述

学界中已有大量工作尝试自动检测内存泄漏，包括静态分析的方法，也包括动态分析的方法。但其中较少工作尝试对内存泄漏进行自动地修复。但实际上很多内存泄漏的修复对程序员来说都是非平凡的问题，因为正确的修复需要跟踪一个资源怎样在各函数中传递以及怎样被使用。但另一方面，软件分析技术非常适合用于解决这样的“跟踪、记录信息”的问题。故我们的项目也尝试用软件分析技术来解决内存泄漏自动修复的问题。

2　研究思路

自动修复可以是静态的修复或动态的修复。静态的修复指的是根据源码静态地对资源使用情况进行分析，并在源码上进行修复，即在恰当位置插入 `free()` 调用。动态的修复则是在运行时跟踪资源的使用情况并对泄漏的资源进行修复，类似于垃圾回收系统。**在这两种方案中我们选择静态修复的方案**，原因是我们认为 C 语言程序更多的是底层程序，运行时的依赖和开销越小越好。

静态的修复可以是保守的，也可是激进的。保守的修复指的是只进行能确保正确性的修复，激进的修复指的是提供修复的建议，并不保证建议一定是正确的，需要程序员进一步人工判断。保守的修复一定是不能处理所有的内存泄漏的情况的，而激进的修复则可以覆盖所有可能的泄漏情况。**在这两种方案中，我们选择保守的修复方案**，原因也是我们认为对 C 语言程序来说，往往也是宁愿有内存泄漏也要保证程序不崩溃出错。

为了进行保守的、静态的修复，我们采用的算法框架即是本门课教授的数据流分析框架。我们分析出每一个程序点一定已经分配的、一定还没有被释放的、以后也一定不会再用到的资源，然后在此程序点插入释放这些资源的语句。

以上三点研究思路都是和高庆等人 [1] 的研究一致。不过高庆等人的工作中没有对他们算法中的诸多非标准的设计选择进行解释，且有后续研究指出他们的实现中存在诸多 bug。故本研究的目的是重新探索各个设计选择对实际修复效果和性能的影响，以及给出更高质量的实现。

3 解法说明

因为 C 语言的各方面诸多的复杂性，想要直接给出一个支持了所有特性的解决方案是不实际的，故本研究采用的策略是**从一个最简实现开始，逐步经验性地评估哪一个特性是最需要支持的，然后再添加支持**。故研究的第一步是实现一个过程内的、不考虑结构体的、不处理循环的方案。

另一方面，因为 C 语言语法的复杂性，想要直接在 C 语言语句上进行数据流分析也是困难的。故我们选择借助 LLVM 框架，将 C 语言程序编译为 LLVM IR 后再进行数据流分析。

概览来说，我们的算法分为如下几个步骤：

1. 别名分析
2. 数据流分析
3. 在IR上进行修复，更新数据流分析结果
4. 将修复反馈到源码上

3.1 别名分析

为了跟踪资源的使用情况，我们的算法需要知道每个指针分别指向什么资源，即进行“别名分析”或“指向分析”。这一部分可以当作一个单独的模块。

因为别名分析在编译器的很多优化和静态分析算法中都会用到，LLVM 也提供了一系列已实现好的分析算法，并提供统一接口。Chris Lattner 等人还实现过 Data Structure Analysis (DSA)，应该仍然是 LLVM 上最强大的别名分析实现。不过 DSA 的实现已经被废弃很久，没有被继续维护。需要用特殊的方法编译、并使用它内部的接口。

不过 LLVM 提供的别名分析都没有提供很好的 MustAlias 的分析，但我们在进行修复时可能需要寻找一个指针“一定”指向某一个资源。故其实目前 LLVM 上的别名分析实现都不能完美满足我们的需求，尽管如此，我们目前仍然使用 DSA 作为我们的指针分析。

3.2 数据流分析

我们要进行的三项数据流分析分别“一定被分配的资源”(MustAllocated),“一定没有被释放的资源”(MustNotFreed),“之后一定不会被用到的资源”(NeverUsedLater)。本处我们详细说明 MustAllocated, 并给出正确性的证明，另外两项是相似的。

MustAllocated 的半格元素是资源的集合，我们对资源做的抽象是把一个 `malloc()` 语句当作一个资源，相当于一个 `malloc()` 永远返回同一份资源。

MustAllocated 的交汇操作是集合的交,因为我们想要进行的分析是 MustAlias，所以我们这三个分析的交汇操作都是集合交。相应的，MustAllocated 的顶元素是全集。

MustAllocated 在 `malloc()` 处的转移函数是将本条语句代表的资源加入到集合中，其他所有语句的转移函数都是恒等函数。

可以看出 MustAllocated 符合数据流分析的 GenKill 标准型，所以可以迭代求出最大不动点。

3.3 IR 上的检测及修复

进行了数据流分析后，我们知道了每个程序点资源的使用状况，于是可以尝试找出恰当的程序点进行资源的修复。在这里我们采用的原则也是“尽早修复”，我们对“尽早”的定义是距离函数入口点越近的语句越早。于是我们的算法是：

```
breadth-first-search every edge e = (h, t) on CFG {
    if(a resource
        m ∈ MustAllocated(h) ∩ MustNotFreed(h) ∩ NeverUsedLater(t) ) {
        insertFreeOnEdge(e, m)
        updateDataFlowAnalysis(e, m)
    }
}
```

我们在 `updateDataFlowAnalysis()` 时不需要完全从头重新计算三个数据流分析的结果，因为我们的修复是插入 `free()` 语句，在我们的三个数据流分析中它总是让格元素变小。而在我们插入 `free()` 语句之前，我们的数据流分析结果是一个不动点，即 $x = F(x)$，其中 x 表示整个数据流分析的结果，F 表示整个数据流分析的转移函数。插入修复语句后，整个数据流分析的转移函数变成了 G，但是注意到 G 满足 $G(x) \sqsubseteq x$，且 G 仍然是单调的。所以我们可以继续在 x 的基础上不断迭代应用 G 至不动点，而不需要重新再从 $\top$ 开始迭代。

3.4 反馈到 C 源码

我们在 IR 上进行了检测和修复之后，还想把修复结果反馈到源代码上。由于 LLVM IR 已经是非常底层的中间表示，一般来说在 IR 上的修改想要返还到源码中是困难的，文献中也几乎查不到相关工作。不过幸运的是，经过仔细观察：由于我们的修改比较简单，所以我们可以做到将 IR 上插入的 `free()` 语句反馈到源码中。

我们假设 LLVM 的 C 语言前端 clang 在将 C 源码的控制结构翻译为 IR 时进行的是一种朴素的翻译方式（不在这里赘述，可参见教材编译原理）。我们的修复是在 CFG 上的一条边上插入了一条 free 语句。如果这条边是在 CFG 中一个基本块内的一条边，那么这条边对应 C 语言源码中两条语句之间的转移或两个表达式之间的转移，我们可以在这两条语句（表达式）之间插入 free 语句。如果这条边是 CFG 中两个基本块之间的转移边，那么在 IR 的边上插入 free 语句相当于创建一个新的基本块，但是在 C 语言中没有“创建一个新的基本块”的对应物，这种情况下该怎么办呢？实际上，CFG 中基本块间的转移边都对应 C 语言中各种控制结构的转移边。经过对 C 语言翻译方式的细致检查，我们发现总可以用增加标志变量的方式在 C 语言中创建出等价的修改（也请见课堂报告幻灯片，不在次赘述，因为太繁琐了）。

4 解决程度

我们目前实现出了一个过程内分析的版本，且实现了在 IR 上的修复，对源码的修复仅实现了一个简单版本。总的来说，实际程序中的内存泄漏很少是过程内的泄漏，故我们的实现还需要进一步扩展成过程间的才会有使用价值。

5 小组分工

本工作的设计总是由杨至轩、冯致远在熊老师的指导下讨论得出。算法实现、文档则是由杨至轩实现和撰写的。

References

[1] Qing Gao, Yingfei Xiong, Yaqing Mi, Lu Zhang, Weikun Yang, Zhaoping Zhou, Bing Xie, and Hong Mei. Safe memory-leak fixing for c programs. In *Proceedings of the 37th International Conference on Software Engineering-Volume 1*, pages 459–470. IEEE Press, 2015.