# 显存泄露静态检测算法

## 研究基础及概念定义

抽象语法树（Abstract Syntax Tree，AST）：AST是程序抽象表达式的一种形式，属于程序中间表达式，这个格式可以充分的表示代码中不同的词法单元如何组织在一起。语法树可以通过语法分析过程得到，对于分析代码的语法结构起到重要作用。完全符合文法规格要求的语法树（分析树）对于进一步处理并不是最合适的，所以在实际使用中，一般会对语法树进行适当的简化和修改。由于抽象语法树的表示简单，且方便使用，所以被广泛的应用在编译器开发和静态分析领域。

针对源程序的每一行进行如下抽象：

<node>

<lineNum>1</lineNum>

<code>

<function>cudaMalloc</function>

<pointer>pointer1</pointer>

</code>

</node>

控制流图（Control Flow Graph，CFG）：控制流图是在代码解析过程中产生的一种中间表达式，是程序分析和静态检测中非常重要的一种代码表示方式。控制流图是一种反应程序逻辑控制流程的有向图。通常一个函数的控制流图可以表示为（N，E，Entry，Exit）。其中N代表节点的集合反映程序中的简单语句和复合语句的条件判断以及控制流汇合点等，E代表有向边的集合，反映程序中语句间的控制流关系。Entry为函数的固定唯一入口节点，Exit为函数唯一的推出节点。简单通俗的说：控制流图即时具有单一的、固定的入口节点和出口节点的有向图。对于非单入口和单出口的程序，可以通过人为添加一个统一入口和出口的方法解决。控制流图可以用于分析程序源代码的分支流向，用于确定代码块之间的调用关系。而且控制流图可以用于检测不可达路径，不可达路径是指无论任何的输入数据都无法使程序的执行到这个分支的程序代码。不可达路径的存在会使程序源代码混乱，也可能在后续的系统迭代中诱发安全漏洞。另一方面也对静态检测带来困难，容易引起误报。因为不可达路径永远不会被执行，但是对于静态检测器来说，很难判断这段代码的上下文逻辑。

int main(){

boolean flag = true;

int \*block = (int \*)malloc(sizeof(int) \* 1000);

if(flag){

free(block);

}

}

条件分支判断

由于控制流图存在不可达路径，需要对分支的条件进行判断；针对显存泄露采用静态分析方式，故不能动态的获取分支条件中变量的准确值。针对以上2点，条件分支判断中变量的取值范围采用集合表示，集合中的每一个值代表当前变量能够取到的所有情况。

集合定义如下。假设int变量X的初始值是1，此时初始化X的集合为{1}；boolean变量X的初始值为True，此时初始化X的集合为{True}

集合运算定义如下。（1）对于变量和常数的四则运算，需要把X的集合中每一个元素进行计算后的值形成一个新的集合作为运算后的集合。假设变量X的集合是{1,2}，X+1的集合即为{1+1,1+2}->{2,3}，减乘除类似。（2）对于变量X和变量Y的四则运算，则需要把每一个变量X集合的元素与变量Y集合的元素运算完成后形成一个新集合。假设变量X的集合是{1,2}，变量Y的集合是{4,5}，那么X+Y的集合即为{1+4,1+5,2+4,2+5}->{5,6,6,7}->{5,6,7}，因为要保证集合中的结果不重复，最后一步需要进行去重。

指针地址映射集合（P-M Set）

显存中的每一个地址都会有一个变量与之对应，显存的申请和释放都是基于指针变量进行的操作。

指针地址集合定义如下。令P表示程序中显存指针变量集合，M表示显存地址集合，对于Pi∈P，Mi∈M，{<{Pi},Mi>}表示指针地址映射集合。

本文将通过分析静态程序的抽象语法树，对指针地址映射集合进行增删改操作，来记录程序对显存的操作。最后通过分析指针地址映射集合里是否有空指针或者还未释放的显存地址。

设每一次操作的指针变量为pointer，显存地址为memory(由于是静态分析，所以不能获取真实的显存地址，本文用当前时间进行代替)。

针对指针地址映射集合的操作如下。（1）malloc(<pointer,memory>)：将指针pointer和显存地址插入到P-M Set中，主要针对cudaMalloc以及hipMalloc函数申请显存。

算法：向P-M Set中添加指针地址映射

输入：指针变量名pointer，显存地址memory

输出：无

Begin

获取当前时间time，将time赋值给memory

新建指针地址映射关系<{pointer},memory>

将<{pointer},memory>插入到P-M Set当中

End

（2）free(pointer)：将指针pointer所指向的显存地址释放，主要针对cudaFree以及hipFree函数释放显存。

算法：从P-M Set中删除pointer指针以及pointer指向的显存地址

输入：指针变量名pointer

输出：显存检测错误码

Begin

遍历每一组<{pointers},memory>，取出指针变量集合{pointers}

判断pointer是否在集合{pointers}当中

如果存在，将当前<{pointers},memory>映射删除，退出循环并返回SUCCESS

如果不存在，继续下一次遍历

遍历结束都没有找到当前pointer，记录一个重复删除异常并返回DUPLICATE\_ERR

（3）modify(<pointeri,memoryi>,<pointerj,memoryj>)：对指针变量进行赋值修改。针对i = j这样的赋值操作。

算法：修改P-M Set集合的元素

输入：指针地址映射关系<pointeri,memoryi>,<pointerj,memoryj>

输出：无

Begin

遍历每一组<{pointers},memory>，取出指针变量集合{pointers}

寻找pointeri所在的集合<{pointers}i,memoryi>

寻找pointerj所在的集合<{pointers}j,memoryj>

如果memoryi和memoryj是同一块显存地址

不做任何操作

如果memoryi和memoryj不是同一块显存地址

将pointeri从{pointers}i中移除，并添加到{pointers}j当中

（4）check()：在程序结束的时候对指针地址映射集合进行遍历，查看是否还有未释放的显存空间。

算法：遍历P-M Set集合

输入：无

输出：显存检测错误码

Begin

判断P-M Set是否为空

如果为空，返回SUCCESS

如果不为空，遍历每一组<{pointers},memory>

如果{pointers}为NULL，返回WILDPOINTER\_ERR

如果{pointers}不为NULL，返回REAMINPOINTER\_ERR

下面通过一段程序进行说明

1 int \*pointer1 = NULL;

2 int \*pointer2 = NULL;

3 size\_t size = 1024\*sizeof(int);

4 cudaMalloc((void\*\*)&pointer1, size);

5 cudaMalloc((void\*\*)&pointer2, size);

6 pointer1 = pointer2;

7 cudaError\_t err1 = cudaFree(pointer1);

8 cudaError\_t err2 = cudaFree(pointer2);

9 return;

指针地址映射集合状态转换表

|  |  |  |
| --- | --- | --- |
| 代码行号 | 注释及操作 | 指针地址映射集合状态 |
| 1、2、3 | 初始化指针以及显存大小 | {} |
| 4 | 为pointer1申请显存地址  malloc(<{pointer1}, 1576160766>) | {<{pointer1}, 1576160766>} |
| 5 | 为pointer2申请显存地址  malloc(<{pointer2}, 1576161000>) | {<{pointer1}, 1576160766>,<{pointer2}, 1576161000>} |
| 6 | 修改pointer1的值  modify(<{pointer1}, 1576160766>,<{pointer2}, 1576161000>) | {<{NULL}, 1576160766>,<{pointer1,pointer2}, 1576161000>} |
| 7 | 释放pointer1指向的显存  free(pointer1) | {<{NULL}, 1576160766>} |
| 8 | 释放pointer2指向的显存  free(pointer2) | {pointers}集合中不包含pointer2，报重复释放的错误 |
| 9 | 程序结束，遍历P-M Set  check() | 因为P-M Set不为空，并且包含NULL指针，所以返回WILDPOINTER\_ERR |

显存泄露故障检测基本思想

针对比较底层的显卡编程语言，比如CUDA以及HIP，应用程序需要cudaMalloc或hipMalloc等函数申请显存空间，当程序使用完相应的空间后，必须手动的调用cudaFree或hipFree等函数释放空间，否则随着程序不断的申请，显存总有被用完的时候，到时候会引起程序乃至系统的崩溃。

本文主要检测以下两个显存泄露类型：

1. 申请的显存空间忘记释放：针对只调用cudaMalloc、hipMalloc，而没有匹配的调用cudaFree、hipFree。
2. 申请的显存空间多次释放：针对申请的同一块显存空间被多次调用cudaFree、hipFree释放。

显存泄露故障检测流程如下：

1. 将源程序转化为抽象语法树的结构。
2. 逐行分析抽象语法树，分析的过程中建立并维护指针地址映射集合，针对分支判断变量的计算采用集合运算。
3. 如果变量集合符合判断条件，则进入该分支进行执行
4. 如果变量集合不符合判断条件，则跳过该分子往下执行
5. 循环（2）~（4）直到程序结束

相关工作对比