# 基于语法树的源码差异分析工具设计与显卡驱动分析

清华大学 武祥晋 2015年6月

【摘要】一款软件常常会依赖其他的软件，下层软件版本的更新可能导致该软件无法正常工作。为了进行针对性的更新，我们需要知道下层软件两版本之间具体发生了哪些修改。为了解决这一问题，人们开发出了各种代码差异分析工具。但是，它们并不能很好地适用于Linux 内核不同版本间的驱动移植。由于Linux内核代码对GCC 扩展语法和宏的广泛使用，其对编译器和基于抽象语法树的代码差异分析工具都是很大的挑战。在本文中，我们将一个抽象语法树差异分析工具前端替换为GCC，使其能够对Linux 内核进行语法差异分析，并以此分析两内核版本中的显卡驱动差异。

关键词：代码差异分析；抽象语法树；GCC；Linux 内核；驱动

**正文**

作者使用Linux内核的编译器GCC完成抽象语法树（AST）的构建，之后使用GumTree进行树差异分析，达到对Linux内核的不同版本进行语法差异分析，并根据这些差异信息为驱动移植工作提供辅助的目的。

作者的主要工作：

【

本文的主要工作包括以下三点：

1. 实现了一个GCC 插件，能在不影响正常编译的情况下，导出GCC 内部的抽象语法树，供GumTree 进一步处理。（第2章）

2. 修改GumTree 的前后端，使用导出的抽象语法树作为输入，并在差异展示时能够显示宏的展开情况。（第3章）

3. 使用此工具对Linux 内核中显卡驱动进行分析。（第4章）

】

作者利用GCC支持插件的机制，获取抽象语法树（具体操作，进行一次内核编译，在make命令后增加参数EXTRA\_CFLAGS=”-fplugin=gccdiff”，表示让GCC 在编译过程中加载自定义的插件），具体是借助GCC完成文本到语法树的解析，然后完成语法树差异分析后将差异对应回源代码。这就要求语法树不能经过任何中间代码优化。在GCC 中，这一时机对应的事件为PLUGIN\_PRE\_GENERICIZE。每当GCC 解析完成源代码中的一个顶层函数定义，就会触发一次该事件，并将构造出的语法树根节点作为参数传给作者在初始化时注册的回调函数。

在获取抽象语法树后，根据以下原则获取树的子节点：

1. 根据源代码结构选取需要的子节点

2. 尊重GCC 抽象语法树结构

3. 以上两条矛盾时，根据是否会影响差异分析结果裁决

【对于特殊情况，如表达式的处理，直接遍历操作数，而不获取其类型，对于va\_arg\_expr，其源码形如va\_arg(args, int)，但在抽象语法树中唯一操作数为args 的变量声明，需要额外获取其类型信息。同样，一些表达式带有冗余的影响差异分析的操作数，如component\_ref （即context.eax这样的成员访问）有一个操作数表示成员在结构体内偏移的字节数，只关心结构体名和成员名，因此这个操作数需要被舍弃。

对于某一个变量的声明和引用，GCC 使用相同的节点表示。例如，语句int a; 以及return a; 中的a 用同一个var\_decl 表示，其有一个标识符子节点（类型identifier\_node，值a）和一个类型子节点（类型integer\_type，值int）。然而，在第一种情况下应该输出两个子节点，而第二种情况下只应输出标识符子节点。

如下语句对应的节点为asm\_expr 类型：

asm (”btsl %2,%[pBase]\n\t”

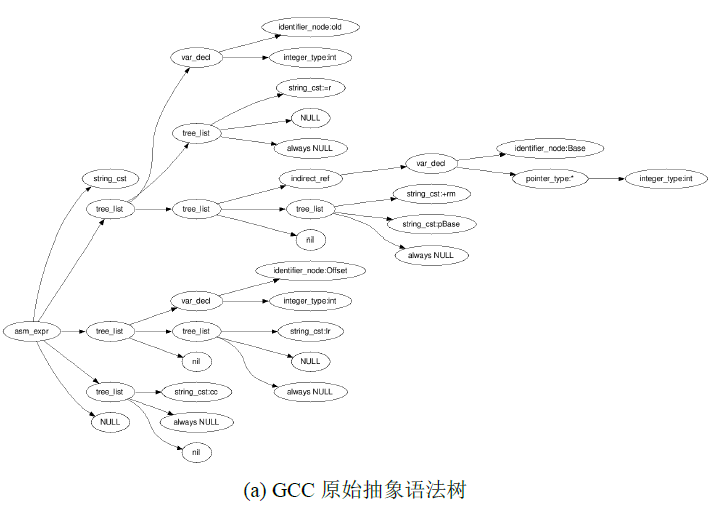
”sbb %0,%0”

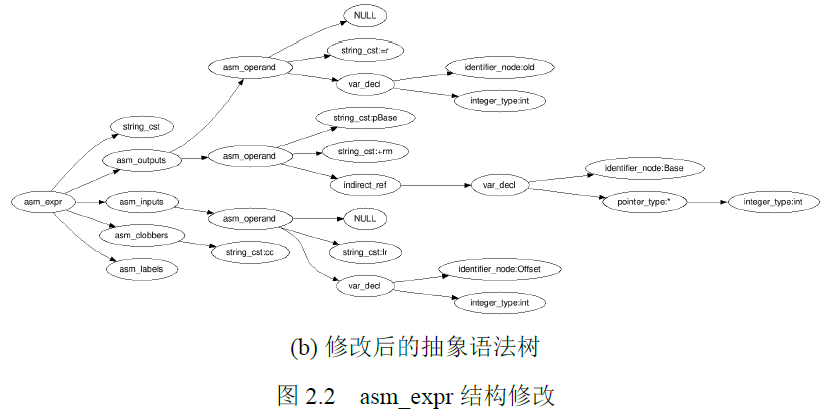
: ”=r” (old), [pBase]”+rm” (\*Base)

: ”Ir” (Offset)

: ”cc”);

GCC 原始的抽象语法树如图2.2(a)。图中NULL、nil 和always NULL 均表示空，后两者分别用于突出强调链表结束和恒空节点。asm\_expr 有5 个子节点，其中后4 个为操作数链表（例中第4 个操作数链表为空），每个操作数链表节点包含3 个子节点，分别为数据（如\*Base 对应的子树）、元数据和链表下一项。4 个链表表示的数据并不相同，实际为3 种类型。前2 个链表表示输入输出操作数，其元数据的3 个节点分别为约束（如+rm）、符号名（如pBase）和空。第3 个链表表示重写（clobber）操作数，仅有数据（如cc）而没有元数据。第4 个链表表示标号（label）操作数，元数据的信息实际可以从数据部分取得。这种在树结构中嵌入链表的做法会误导差异分析算法，使其将之作为一般的树结构处理。于是，我们将4 个链表转化成4 个节点，节点的每个子节点表】





作者在获取到语法树后，导出的节点分为8类。

• 字面量。额外信息为字面量的值。

• 类型。额外信息为类型名称。

• 声明。额外信息为声明位置。

• 表达式。额外信息为表达式位置，以及宏展开信息。（详见2.3）

• 标识符。额外信息为标识符名。

• 函数定义。额外信息为函数开始和结束位置。

• 引用（含义见2.2.2）。无额外信息。

• asm\_expr（含义见2.2.2）。无额外信息。

对于宏展开处理，因为宏展开是在预处理中完成的，本质是单词序列的替换。对于带参数的宏，称宏定义中的单词序列为替换（replacement）序列，其中某些单词为形参（parameter）。称宏展开后的单词序列为展开（expansion）序列，其与替换序列一一对应，形参被替换为实参（argument）。也即追溯展开序列中每一个的单词的来源时，有两种情况：部分直接来源于替换序列，部分来源于实参对形参的替换。这里共涉及到三种位置：宏展开位置、替换序列位置和可能的实参位置。宏的嵌套分为两种情况。当宏的定义中包含其他宏时，会进行递归的展开。而当宏的实参中包含宏时，该宏会先被完全展开作为实参，然后再进行上述的可能递归的展开。

因此，策略是：从最终展开向原始代码逐层回溯，若目标单词在某一层是形参，则不记录该层，目标单词变为对应实参；直到追溯到某一层目标单词是普通替换序列单词，此后没有目标单词，记录每一层的展开信息，直到回到原始代码。

获取完各差异信息，在导出格式上，论文方法是将导出的原始格式经过Python 转换成GumTree 接受的XML 格式，GumTree 直接在代码同目录下寻找增加了后缀的XML 文件，读入并完成树的构建。论文修改了GumTree内部的Tree结构，增加了一个标记域，在前端读入语法树时从XML元素的aux属性得到该值。

在可视化（HTML/CSS/JS 实现）展示阶段。首先调用GumTree 得到其编辑动作信息，然后使用Python 的difflib 得到文本差异信息。此后，根据编辑动作信息中包含的开始和结束位置，在文本差异结果上进行标注，即在对应位置增加HTML 开闭标签。这些差异标记并不直接对应高亮格式，而是在页面上方提供开关选项，用JS 控制使得不同的CSS生效，从而实现两种差异分析结果的显示开关。

论文在第四章对显卡驱动进行了分析，主要是函数的变化情况，增加或删除以及修改。此处对论文开发的工具进行了修改。

1. 树根节点由函数改为编译单元，降低了对GumTree 的调用次数，从而避免反复地创建和销毁进程，加快处理。

2. 对树节点上的附带信息进行精简，只保留对差异比较有影响的类型和字符串值，而删除无用的源码位置信息、宏展开来源信息。

3. 为了方便树比较之后的统计，在每个节点上标记其所属的函数名称。

4. 得到GumTree 的节点差异输出后，根据每个节点上标记的函数名称进行统计，而不再进行文本差异、差异合并标注、宏处理等操作。

作者选取Linux 版本3.5.4 和3.8.13，对VGA 控制台驱动进行差异分析。该驱动仅有一个编译单元，主源码文件为drivers/video/console/vgacon.c。差异分析的结果有函数的变化数据统计以及源代码差异查看。

**局限性**：

在GCC 中，所有对某变量的引用都指向其声明节点，所有值相同的类型节点或字面量节点共用一个节点。这导致作者导出的抽象语法树的叶节点缺少位置信息，从而无法应对#define MAX 0x7fffffff 这样的宏定义。

在具体差异分析中，作者的工具成功地捕获了隐含在宏展开中的差异，而文本差异工具是无法做到的。但是，当代码涉及到for 循环结构、尤其是改动较大时，作者的工具表现得不甚理想，会给出大量难以理解的节点移动操作。

**作者的展望：**

对于解析器（对获取的原始语法树操作的插件），可以讨论如何修复叶节点的位置问题，从而能够支持只含有字面量和类型的宏，以及更精确地标定差异位置。对于差异分析结果的利用，可以尝试生成语义补丁，进而利用Coccinelle 实现驱动自动移植。更进一步地，可以不局限于源代码的语法结构，向语义方向进化，能够识别一些语义等价的转化而不认为是差异。

**GCC相关知识**

每一个待编译的c 文件被称为一个编译单元（translation unit）。虽然GCC 名为编译器，但实际在对GCC 的一次调用中集成完成了对编译单元的预处理、编译和汇编操作。其中预处理首先进行词法分析得到单词（token）序列，然后在此之上根据指令进行单词替换，最终仍得到一个单词序列。编译过程读入上一阶段的单词序列，构建出对应的抽象语法树，然后在其上进行多遍的中间代码转化，最终生成结构性很弱的汇编指令流。汇编操作完成汇编代码到二进制目标文件的转化。

GCC 版本4.8.4已提供了插件支持，GCC 插件通过动态链接库的形式实现，并通过命令行参数-fplugin 指定。GCC 启动时会加载并调用插件的初始化函数，插件在初始化过程中对某些GCC 编译事件注册回调函数。

**GCC 抽象语法树相关知识**

GCC 内部用tree 抽象类型来表示一个抽象语法树节点，每个节点的具体类型（type）可以通过其TREE\_CODE 宏得到。由于GCC 最终的目标是编译，节点上并没有一个直接遍历所有子节点的接口，不同作用的子节点需要用专用的接口获得。也因此不同类型的节点接口不同，需要区别对待。在GCC 中，这些节点类型被分为11 个类别（class），相同类别的节点接口相似。

• tcc\_constant 表示字面量节点，如整数字面量（integer\_cst）。节点包含字面量值，没有子节点。

• tcc\_type 表示类型节点，如整数类型（integer\_type）。节点可能有类型名称，复合类型拥有子节点。

• tcc\_declaration 表示声明节点，如参数声明（parm\_decl）。可能的子节点包括名称、类型、函数声明的参数等。

• tcc\_reference, tcc\_comparison, tcc\_unary, tcc\_binary, tcc\_statement, tcc\_vl\_exp, tcc\_expression 都属于表达式，如加法（plus\_expr）。除了某些特殊节点，子节点即对应表达式的操作数（operand）。

• tcc\_exceptional 表示不能被分入其他类别的类型， 如语句列表（statement\_list）。本类别中节点接口无相似性，需要单独处理。

下面的代码可以得到如图2.1 的抽象语法树。

int main(int argc, char\* argv[]) {

if (argc >= 2)

printf(”Hello, %s!\n”, argv[1]);

return 0;

}



从图中可以看出，根节点是一个函数声明，其子节点为1 个标识符、1 个返回值声明（子节点含返回值类型）、2 个形参声明和1 个代码块，这些分别通过DECL\_NAME, DECL\_RESULT, DECL\_ARGUMENTS 和DECL\_SAVED\_TREE 获取。在树路径function\_decl / bind\_expr / statement\_list / cond\_expr /call\_expr / addr\_expr 下同样有一个函数声明节点，在GCC 中它的结构与根节点是一样的，也可以获取返回值声明、形参声明等子节点。