# 本 科 实 验 报 告

实验名称：动态测试

学　　员：马坚　　学　　号：201912509012

培养类型：指挥　　年　　级：2019 级

专　　业：软件工程　　所属学院：计算机学院

指导教员：陈振邦　　职　　称：

实 验 室：306-705　　实验日期：2022-6-7

国防科技大学训练部制

# 1. 实验目的和要求

## 1.1. 目的

实验 1：GCOV

基于 gtest 设计测试用例使用 GCOV 分析一个真实 C/C++开源程序，使同学们深入理解动态测试，加深对 Gtest 的理解。

实验 2：AFL-fuzzing

用 AFL 测试一个 C/C++开源程序，加深对模糊测试的理解。

## 1.2. 要求

实验 1：GCOV

设计测试用例配合 gtest 测试，并使用 GCOV 分析一个 1500 行以上的真实 C/C++开源程序（不能使用程序自带的测试用例!!）

提交实验报告，内容包括：

1）分析过程：使用过程、编译运行命令、测试用例构建等；

2）分析结果：覆盖率信息

附加提交文件包括：被分析代码+测试用例

实验 2：AFL-fuzzing

使用 AFL 测试一个 1500 行以上的真实 C/C++开源程序

行覆盖率和条件覆盖率越高越好

测试执行次数达到 500 万次

# 2. 实验内容和原理

## 2.1. 内容

通过不同的动态测试方法，统计 C/C++程序行覆盖率和条件覆盖率。

## 2.2. 原理

gcc 中指定-ftest-coverage 等覆盖率测试选项后，gcc 会：

1 在输出目标文件中留出一段存储区保存统计数据。

2 在源代码中每行可执行语句生成的代码之后附加一段更新覆盖率统计结果的代码,也就是前文说的插桩，

3 在最终可执行文件中进入用户代码 main 函数之前调用 gcov_init 内部函数初始化统计数据区，并将 gcov_exit 内部函数注册为 exit handlers 用户代码调用 exit 正常结束时，gcov_exit 函数得到调用，其继续调用 __gcov_flush 函数输出统计数据到 *.gcda 文件中。

## 3. 操作方法与实验步骤

### 3.1. Gcov & Lcov

操作方法：

1）寻找合适的开源项目；

2）用 lcov 分析代码行数，大于等于 1500 行；

3）对源代码中的条件语句进行分析；

4）新建 gtest.cpp 编写测试用例；

5）使用 genhtml 生成 html 页面查看覆盖率报告;

6）过程中重复步骤 3、4、5，直至行覆盖 100%，条件覆盖 95%。

实验步骤：

```
git clone https://bdgit.educoder.net/pvqf6mep3/SQA-Homework.git
cd SQA-Homework/
git checkout Dynamic-testing
cd gcov\&lcov/
g++ gtest.cpp -fprofile-arcs -ftest-coverage -o gtest -pthread -lgmock -lgtest
./gtest
lcov --rc lcov_branch_coverage=1 -c -d . -o test.info
genhtml --branch-coverage test.info -o ./output
cd output/ && firefox index.html
```

### 3.2. AFL-fuzzing

操作方法：

1）寻找合适的开源项目；

2）用 lcov 分析代码行数，大于等于 1500 行；

3）afl 编译源代码生成可执行文件；

4）新建文件夹 fuzz_in 和 fuzz_out；

5）fuzz_in 中创建 testcase 文件作为种子输入；

6）运行可执行文件，进行模糊测试。

实验步骤：

```
cd afl_fuzz/
afl-gcc -g -o afl_test allocate.c compile.c dictionary.c expression.c types.c
mkdir fuzz_in fuzz_out
echo "fuzzing" > fuzz_in/testcase
```

afl-fuzz -i fuzz_in -o fuzz_out ./afl_test -f

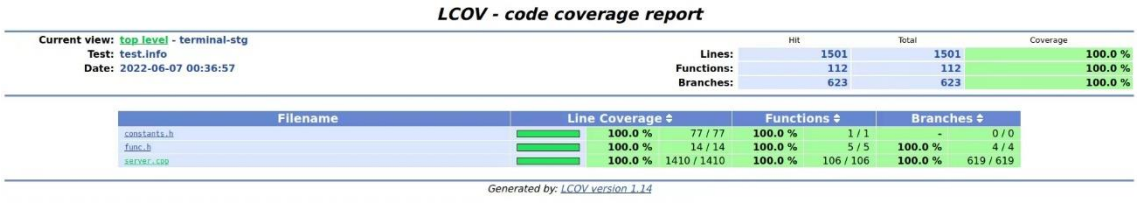
(如遇报错，请按日志内容进行修改）

# 4. 实验结果与分析

## 4.1. Gcov & Lcov

• 实验结果

行覆盖率：100%；条件覆盖率：100%；方法覆盖率：100%；



LCOV - code coverage report

Current view: top level - terminal-stg
Test: test.info
Date: 2022-06-07 00:36:57

| | Hit | Total | Coverage |
|---|---|---|---|
| Lines: | 1501 | 1501 | 100.0 % |
| Functions: | 112 | 112 | 100.0 % |
| Branches: | 623 | 623 | 100.0 % |

| Filename | Line Coverage | | Functions | | Branches | |
|---|---|---|---|---|---|---|
| constants.h | 100.0 % | 77 / 77 | 100.0 % | 1 / 1 | - | 0 / 0 |
| func.h | 100.0 % | 14 / 14 | 100.0 % | 5 / 5 | 100.0 % | 4 / 4 |
| server.cpp | 100.0 % | 1410 / 1410 | 100.0 % | 106 / 106 | 100.0 % | 619 / 619 |

Generated by: LCOV version 1.14

图 1 LCOV 代码覆盖率统计图

**• 实验分析**

测试用例设计：

1） 条件语句种类：

$if-else$：如果是$if(A)$，那么他的两个分支为$A$和$\overline{A}$，只是$\overline{A}$不会有执行语句；如果是$if(A)-else$，区别就是$\overline{A}$会有执行语句。

```
if (userlist == NULL) {
    log("can not find " REGISTERED_USER_FILE "");
    return;
}
```

图 2-1 $if(A)$类型语句

```
if (sessions[uid].state == USER_STATE_UNUSED
    || sessions[uid].state == USER_STATE_NOT_LOGIN) {
    return false;
} else {
    return true;
}
```

图 2-2 $if(A)-else$类型语句

$switch-case$：条件分支数 = $case$语句数 + 1，因为默认存在$default$分支，即使$default$分支没有显示。当然如果出现$switch-case$语句缺少$default$分支，那么可以认为这是一个$fault$，因为这种缺陷会带来安全上的隐患。

```
switch (cur.dir) {
    case DIR_UP: {
        if (cur.pos.y > 0) {(cur.pos.y)--;break;}
        else {cur.dir = DIR_DOWN;break;}
    }
```

图 2-3 *switch* — *case* 类型语句

```
    default: {
        cur = item_to_map[it.kind];
        map[y][x] = max( a: map[y][x],  b: cur);
    }
```

图 2-4 *switch* — *case* 语句中的*default* 语句

*for*：*for* 语句通常只有两个分支，例如*for*(*int* $i = 0; i < I; i++$)，它的真分支为：$i < I$，假分支为：$i \geq I$。通常情况下执行*for* 会将两个分支都走到，但当*for* 循环语句中存在*break* 语句时，会因提前退出循环而无法执行假分支。

```
for (int i = 0; i < USER_CNT; i++) {
    for (int j = i + 1; j < USER_CNT; j++) {
        if (sm.users[i].score < sm.users[j].score) {
            std::swap( &: sm.users[i],  &: sm.users[j]);
        }
    }
}
```

图 2-5 *for* 语句

*while*：*while* 语句的分支个数需要依据其判断条件而定，通常满足以下关系：分支数 = 单元条件数 × 2。如*while*($A$ && $B$)，其条件分支数为 4。

```
while (battles[bid].is_alloced) {
    battles[bid].global_time++;
    t[0] = myclock();
    move_bullets(bid);
    check_all_user_status(bid);
```

图 2-6 *while* 语句

其他：其他语句也可能带来分支，如*stdio* 库中对*FILE* 的操作，当成功打开文件即为执行真分支，反之执行假分支；*socket* 库中*bind* 操作，也分成功和失败，分别对应真假分支。

```
FILE* userlist = fopen( FileName: REGISTERED_USER_FILE,  Mode: "r");
```

图 2-7 *FILE* 操作

```
int sockfd = socket(AF_INET, SOCK_STREAM, 0);
```

图 2-8 *socket* 操作

2）条件分支测试：

只含有单元条件：如$if(A)$，设计测试用例$A$与$\overline{A}$；

含有两个及以上单元条件：如$if(A \ \&\& \ B)$，源代码层面，只要到设计两个测试用例$(A,B)$与$\left(\overline{A},\overline{B}\right)$即可。但编译过后，二进制码层面，$if(A \ \&\& \ B)$会改写成$if(A):if(B)$，因此上述两个测试用例是无法达到 100%条件覆盖的，因为当$A$为$false$时，就已经退出整个判断，并不会在意$B$到底是$true$还是$false$，因此需要增加测试用例$\left(A,\overline{B}\right)$。

测试用例结果：

1）不可达分支：

*assert* 语句：如果执行假分支，则会直接使程序崩溃。如下图所，当$uid \geq USER_CNT$，程序会直接退出。

```
int query_session_built(uint32_t uid) {
    assert(uid < USER_CNT);
    if (sessions[uid].state == USER_STATE_UNUSED
        || sessions[uid].state == USER_STATE_NOT_LOGIN) {
        return false;
    } else {
        return true;
    }
}
```

图 3-1 *assert* 语句假分支不可达

2）矛盾条件：

当判断条件前后矛盾时，只能执行其中一个分支，另一分支则无法被执行。如在$if(a<5):if(a>10)$，$a>10$条件的真分支是不可达的。

```
int uid = find_uid_by_user_name( user_name: argv[1]),
    status = atoi( String: argv[2]);
if (uid < 0 ||
    uid >= USER_CNT ||
    sessions[uid].conn < 0) {
    return -1;
}
```

图 3-2 矛盾条件

在上述代码中的*uid*来源于方法*find_uid_by_user_name*，其中关于*uid*返回值的定义如下：

```
int find_uid_by_user_name(const char* user_name) {
    int ret_uid = -1;
    log("find user %s", user_name);
    for (int i = 0; i < USER_CNT; i++) {
        if (query_session_built( uid: i)) {
            if (strncmp(user_name, sessions[i].user_name,
                        USERNAME_SIZE - 1) == 0) {
                ret_uid = i;
                break;
            }
        }
    }
```

图 3-3 方法*find_uid_by_user_name*的定义

通过代码可知，$uid < USER_CNT$是恒成立的，因此$uid \geq USER_CNT$的真分支是无法执行到的。

3）其他奇怪语句：

在测试中还发现一些奇怪语句，他的真假分支可谓是毫无逻辑，我对此深感疑惑。

单独测试时，未产生分支，增加额外方法后，产生分支。

```
:          1 : void test(int uid){
:          1 :     send_to_client(uid, SERVE
:            :     //server_message_t sm;
:            :     //wrap_send(-1, &sm);
:          1 : }
```

图 4-1 单独测试不产生分支

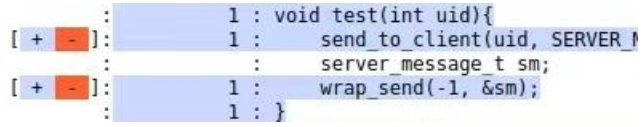



图 4-2 多函数测试产生分支

在不同的地方，*printf* 也可以产生分支。

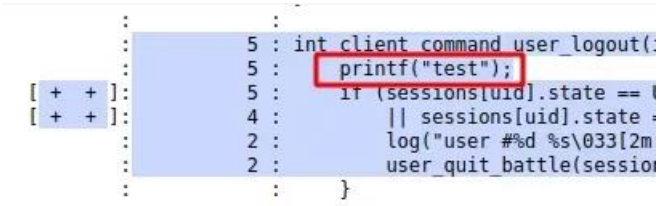



图 5-1 *printf* 不产生分支

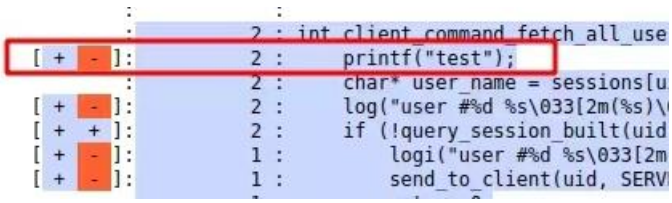



图 5-2 *printf* 产生分支

对于上述奇怪语句，在实验中我采取注释的方法，眼不见心不烦。

## 4.2. AFL-fuzzing

- 实验结果


图 6 测试达到 500 万次

## 5. 总结与致谢

通过本次实验，使我加深了对动态测试的理解，在设计测试用例时，由于没有区分源代码与二进制代码的区别，导致消耗较多时间。后来明白其中的差别后，进行设计便逐渐游刃有余。过程中还遇到了许多疑难困题，其中有部分是因为 lcov 自身的缺陷，这也说明了工具其本身也会有缺陷。

另外，在实验的过程中，因为我个人的无聊，特意去研究了 test.info 文件与 html 页面之间的关系，通过对比发现，html 中的每一行数据都可以在 test.info 文件中找到对应。

首先 test.info 文件中的首行为 TN：代表开始；接下来的一行开头为 SF：表示此代码的位置，并且以 end_of_record 结束。

```
1 TN:
2 SF:/home/jackyma/Study/软件质量保证/动态测试/terminal-stg/constants.h
3 FN:127,_Z14init_constantsv
4 FNDA:1,_Z14init_constantsv
```

图 7-1 test.info 文件开头内容

```
81 DA:207,1
82 end_of_record
83 SF:/home/jackyma/Stuc
```

图 7-2 end_of_record 结束

接下来是 FN：代表函数，其后紧跟着的为此函数在源代码中的行数，最后会有 i j 这样的后缀，这代表函数中传的参数，有一个就代表有一个参数。

紧挨着 FN 的是 FNDA：表示此函数执行的次数。

```
 96 FN:279,_Z9save_useri
 97 FNDA:2,_Z9save_useri
 98 FN:287,_Z28client_command_user_registeri
 99 FNDA:5,_Z28client_command_user_registeri
100 FN:323,_Z25client_command_user_logini
101 FNDA:9,_Z25client_command_user_logini
```

图 7-3 函数相关信息

```
286                :            :
287                :          5 : int client_command_user_register(int uid) {
288                :          5 :     int ul_index = -1;
289                :          5 :     char* user_name = sessions[uid].cm.user_name;
290                :          5 :     char* password = sessions[uid].cm.password;
291                :          5 :     log("user %s tries to register with password %s", user_name, password);
```

图 7-4 html 界面中信息

当所有函数信息记录完毕后，就是行信息。

开头为 DA，格式为 DA：行号，执行次数，如图 7-5 所示，表示行 231 执行了 6 次。如果该行产生了分支，那么紧随其后是 BRDA。其格式为：BRDA：行号，0（这个 0 我暂时还没搞明白，因为所有的都是 0），分支序号，执行次数.

```
DA:229,6
DA:231,6
BRDA:231,0,0,1
BRDA:231,0,1,5
DA:232,1
DA:233,1
```

图 7-5 test.info 中行信息

```
228                :          :
229                :          6 :    FILE* userlist = fopen(REGISTERED_USER_FILE, "r");
230                :          :
231        [ +  + ]:          6 :    if (userlist == NULL) {
232                :          1 :        log("can not find " REGISTERED_USER_FILE "");
233                :          1 :        return;
234                :          :    }
235                :          :
```

图 7-6 第 231 执行情况

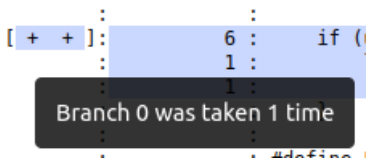



图 7-7 分支 0 执行情况

理论上在 test.info 中可以修改信息并在 html 中展示，于是我付诸实践，发现成功了！**但在此声明，本人的 100%覆盖率是完全用 gtest 测试出来的！（只不过把奇怪语句删除了）**按照此方法，可以通过脚本自动修改数据，达到“外挂”的效果，轻松实现 100%，甚至说，可以先扫描一遍所有的分支，然后让用户输入一个比例，修改条件分支覆盖率为用户想要的比例！

在实验的过程中有许多不懂的地方，感谢教员与教辅的耐心指导，使我在面对困难时有了更多的信心，最终成功完成实验。也感谢我身边的同学，当我有疑惑时，总是热情第帮助我分析问题并给予解决方案，没有同学的帮助也不会有现在的成果。