

目录

1	引言	1
1.1	实验目的	2
1.2	实验原理	2
1.2.1	白盒测试	2
1.2.2	语句覆盖	3
1.2.3	分支覆盖	3
1.2.4	实验团队信息	4
2	实验配置	4
2.1	硬件设备	4
2.2	软件配置	4
2.3	文件目录结构	5
3	实验	6
3.1	总体架构	6
3.2	脚本语言选择	6
3.3	实验预处理	8
3.3.1	测试用例命令修正	9
3.3.2	Bug 植入	9
3.3.3	预处理代码编写	11
3.4	对正确代码进行测试	14

3.4.1	实验原理	14
3.4.2	数据提取	14
3.4.3	数据可视化	16
3.5	对植入 Bug 的代码进行测试	17
3.5.1	实验原理	17
3.5.2	数据提取	18
3.5.3	数据可视化	19
3.6	对正确代码和植入 bug 代码的测试结果进行比较	20
3.6.1	实验原理	20
3.6.2	数据提取	21
3.6.3	数据可视化	23
4	实验总结	25
	参考文献	25
	附录	25

1 引言

当今的软件开发越来越复杂，需要应对不同的需求和场景，因此测试也越来越重要 [1]。测试不仅是为了发现软件中的错误和漏洞，也是为了保证软件的正确性、可靠性和稳定性。在测试过程中，测试用例的设计和编写是非常关键的，它们需要覆盖软件中的所有可能情况，以确保软件的正确性和可靠性。除了测试用例，还有一些测试方法和工具可以帮助开发者在软件开发和测试中提高效率和质量。

白盒测试是一种常用的测试方法，它需要开发者深入了解代码的内部实现，通过分析代码的执行路径来设计测试用例。与黑盒测试不同，白盒测试可以检测到软件中的逻辑错误和边界条件问题，因此在软件开发和测试中非常有用 [2]。然而，白盒测试需要开发者具备较强的编程和测试技能，需要大量的时间和精力来设计和编写测试用例。

自动化测试工具是另一个可以提高测试效率和质量的工具。自动化测试可以帮助开发者快速执行大量的测试用例，检测软件中的错误和漏洞，并生成详细的测试报告。自动化测试可以减少测试的时间和人力成本，提高测试的覆盖率和质量 [3]。

`gcov` 是一个常用的代码覆盖率分析工具，它可以帮助开发者分析代码的覆盖率，找到未经测试的代码，并提高代码的质量和可靠性。通过 `gcov` 生成的代码覆盖率报告可以告诉开发者哪些代码没有被测试到，以及哪些测试用例可以增加代码覆盖率。这些信息可以帮助开发者设计更好的测试用例，提高测试的覆盖率和质量。同时，`gcov` 也可以帮助开发者发现代码中的漏洞和错误，从而提高软件的质量和可靠性 [4]。

在本实验中，我们将学习如何使用 `gcov` 命令来生成代码覆盖率报告，并通过分析报告来发现代码中的漏洞和错误。通过实践和分析，我们将掌握一些软件测试的相关方法和经验，例如，如何设计和编写高质量的测试用例，如何进行白盒测试，以及如何使用自动化测试工具来提高测试效率等。

1.1 实验目的

- (1) 利用 gcov 对待测试项目进行自动化测试并分析可以帮助我们了解代码的覆盖率情况，发现未经测试的代码，从而提高代码的质量和可靠性。通过这个实验，我们可以学习如何使用 gcov 命令来生成代码覆盖率报告，并通过对分析报告来发现代码中的漏洞和错误。
- (2) 通过对测试结果的分析，我们可以掌握一些软件测试的相关方法和经验。例如，我们可以了解如何设计和编写高质量的测试用例，如何进行白盒测试，以及如何使用自动化测试工具来提高测试效率等。这些经验对于我们在软件开发和测试中都非常有用，可以帮助我们提高软件质量和可靠性，减少开发成本和时间。

1.2 实验原理

1.2.1 白盒测试

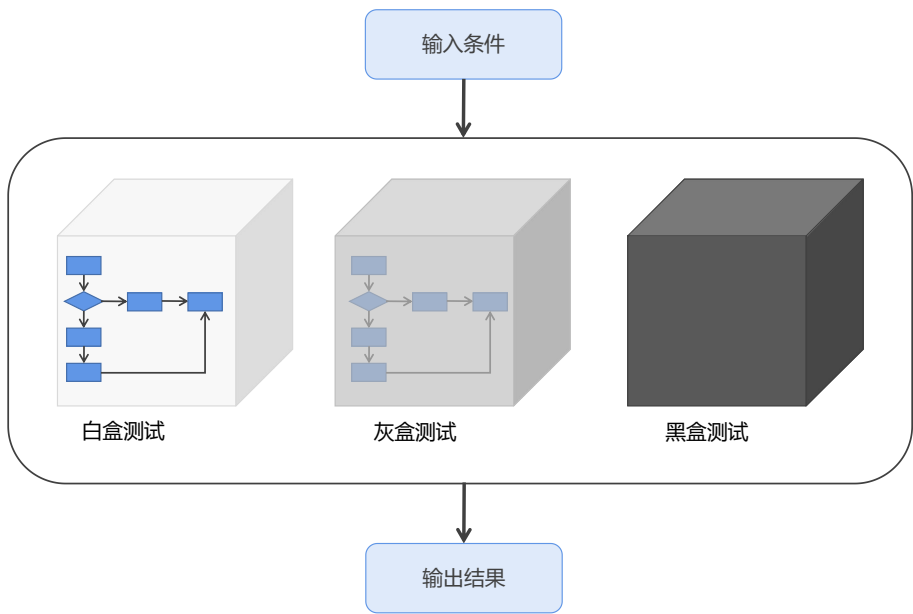


图 1: 白盒测试需要深入分析内部结构

白盒测试（White-box testing）是一种软件测试方法，也称为结构测试或基于

代码的测试。它基于深入了解软件系统的内部结构和工作原理来测试软件系统的正确性、可靠性和健壮性。在白盒测试中，测试人员通常需要访问和分析软件系统的源代码，以确定测试案例的范围和内容。测试人员可以使用各种技术和工具来检查代码，例如**代码覆盖率分析**、**路径分析**、**逻辑覆盖测试**等。

如图1所示，白盒测试可以帮助测试人员了解软件系统的内部结构和工作原理，以便更好地识别和修复软件系统中的缺陷。此外，白盒测试还可以帮助测试人员确定哪些代码需要更多的测试覆盖和哪些代码不需要测试覆盖。

1.2.2 语句覆盖

语句覆盖（Statement coverage）是一种软件测试技术，它的目的是测试程序中的每个语句是否至少被执行一次。在语句覆盖测试中，测试人员编写测试用例并执行这些测试用例来测试程序。在执行测试用例的过程中，测试人员记录下执行的语句，最终确定是否覆盖了所有的语句。如果所有的语句都被执行了至少一次，那么语句覆盖就达到了 100%。

语句覆盖测试可以帮助测试人员发现程序中未被执行的语句，从而减少程序中的漏洞和缺陷。但是，语句覆盖测试并不能保证程序的完全正确性，因为它并没有测试程序中语句之间的关系和逻辑正确性。需要注意的是，在进行语句覆盖测试时，测试人员应该注意测试用例的质量和覆盖率，尽可能地覆盖所有可能的情况，以便发现程序中的潜在问题。

1.2.3 分支覆盖

分支覆盖（Branch coverage）是一种软件测试技术，它的目的是测试程序中的每个分支是否都被执行至少一次。在分支覆盖测试中，测试人员编写测试用例并执行这些测试用例来测试程序。在执行测试用例的过程中，测试人员记录下执行的分支，最终确定是否覆盖了所有的分支。如果所有的分支都被执行了至少一次，那么分支覆盖就达到了 100%。

分支覆盖测试可以帮助测试人员发现程序中未被执行的分支，从而减少程序中的漏洞和缺陷。与语句覆盖测试不同的是，分支覆盖测试还可以测试程序中的

分支之间的关系和逻辑正确性。需要注意的是，在进行分支覆盖测试时，测试人员应该注意测试用例的质量和覆盖率，尽可能地覆盖所有可能的情况，以便发现程序中的潜在问题。

1.2.4 实验团队信息

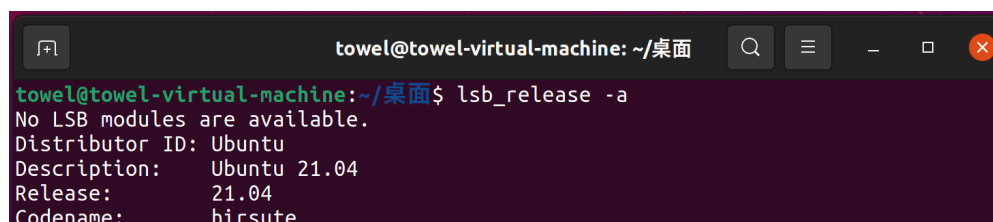
小组组长：XX

小组成员：XX, XX, XX, XX, XX

2 实验配置

2.1 硬件设备

该实验在提供的虚拟镜像下完成：如图2所示，当前系统的版本为 Ubuntu 21.04。其他硬件信息如内存分配等完全与镜像配置一致。

A terminal window titled 'towel@towel-virtual-machine: ~/桌面' with search, menu, and window control icons. The command 'lsb_release -a' has been executed, resulting in the following output:

```
towel@towel-virtual-machine:~/桌面$ lsb_release -a
No LSB modules are available.
Distributor ID: Ubuntu
Description:    Ubuntu 21.04
Release:        21.04
Codename:       hirsute
```

图 2: 实验运行操作系统配置

2.2 软件配置

1. 对于自动化运行代码的编辑工具，我们主要是在 VS Code (v1.76.1) 中完成。
2. 对于 Shell 类型，我们除了在 Linux 原生 Shell 终端中进行操作之外，还利用了 VSCode 的终端中进行操作。从图3(a)和图3(b)中可以看出，两种终端都是运行的 Bash 作为命令行解释器。

```
towel@towel-virtual-machine:~/桌面$ echo $0
bash
```

(a) 原生终端 Shell 类型

```
• towel@towel-virtual-machine:~/桌面/software-test$ echo $0
/usr/bin/bash
```

(b) VSCode 终端 Shell 类型

图 3: 两种 Shell 类型比较

3. 对于生成自动化脚本的工具，我们使用 Python 语言 (v3.9.4)。在第3.2章节中，会对该工具以及自动化生成脚本的过程进一步介绍。
4. 对于小组代码提交工具，我们采用 Git 来进行分布式版本控制。如图4所示，我们的核心代码在 Master 主分支上的提交记录呈现如图。为了方便小组合作，我们采用 Gitee 进行代码托管。这样所有成员就可以随时进行代码版本更新与资源同步。

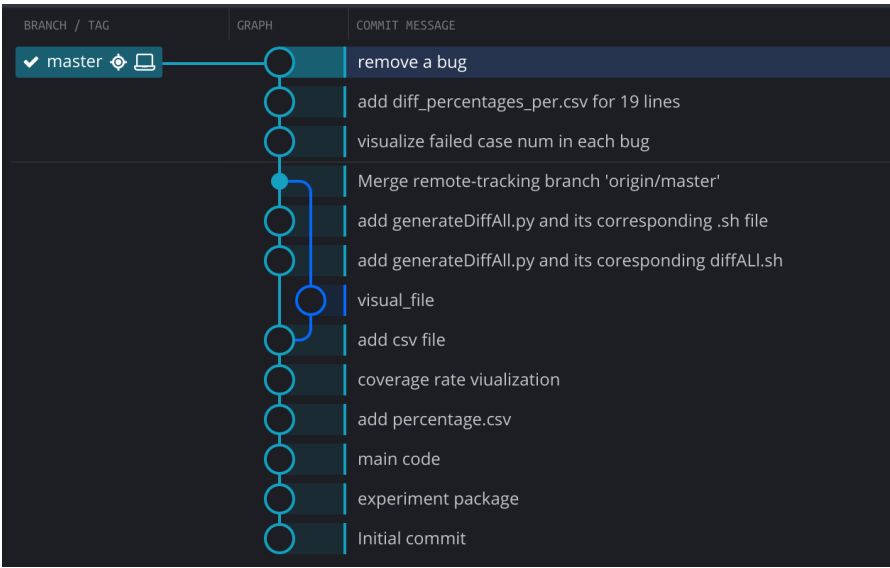
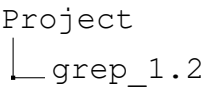
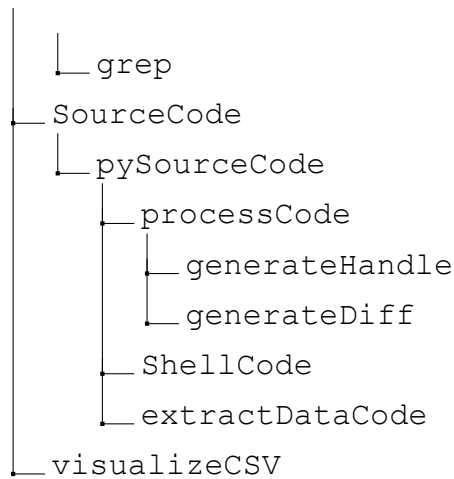


图 4: 实验代码版本管理

2.3 文件目录结构

对于整个项目目录 *Project*，总体采用如下的分布：





对于 `grep_1.2`，实际就是实验提供的源代码，包含了需要进行测试分析的程序。第二部分是我们编写的自动化测试代码的集合，其中生成类代码分为生成插桩覆盖的代码和生成输出结果对比的两类测试处理代码；还包括生成出 `Shell` 代码以及最后的数据提取代码。最后是提取出数据的 `csv` 文件的集合，便于在第3章节的后续进行数据分析。

3 实验

3.1 总体架构

如图5所示，本次实验总共可以从宏观上划分为 4 个板块：实验预处理、实验计划实施、自动化测试和可视化分析。从细节步骤上：首先我们了解了核心指令含义以及对输入用例进行预处理；然后通过从单次覆盖到多次覆盖，以及从语句覆盖到分支覆盖来层层递进，实施了实验计划；通过自动化编写脚本：完成了 18 个 Bug 的测试覆盖，并对 `Gcov` 插桩输出的内容与 `Test Oracle` 进行对比；最后，我们还编写了脚本提取总结出实验过程中的核心数据，并完成了可视化分析。

3.2 脚本语言选择

在本次实验中，我们需要自动化地执行大量的测试用例并分析测试结果。我们根据实验本身的特点，利用 `Shell` 脚本进行输入数据与测试。对于 18 个插入的 Bug，每一个 Bug 都要经过一轮 199 个用例的完整测试，并且输出到不同的文件

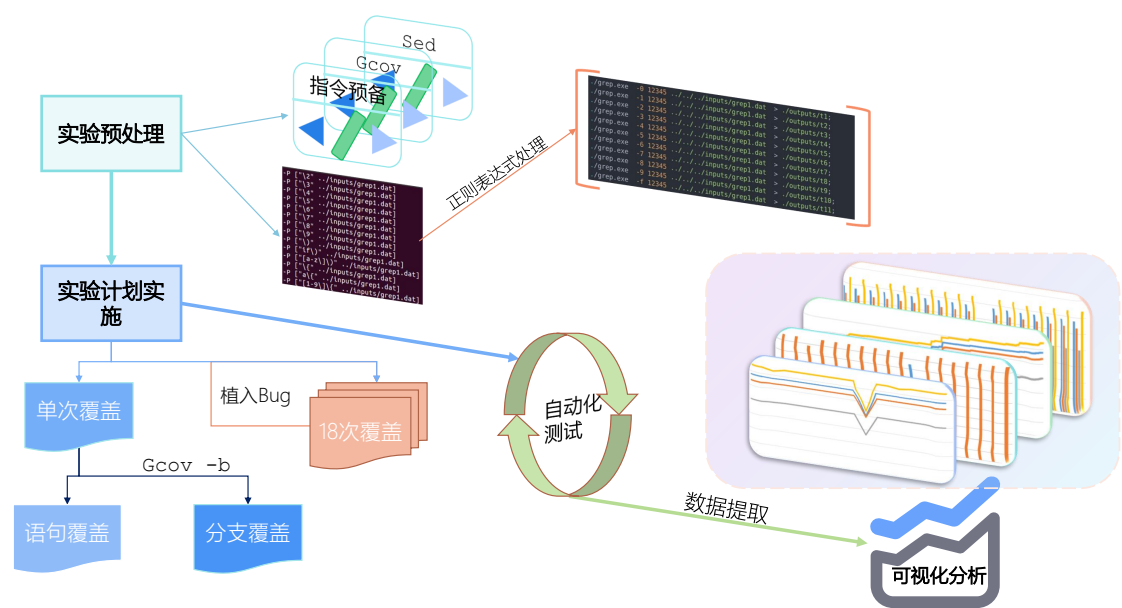


图 5: 实验整体流程

夹下。因此手动输入这 $199 \times 18 = 3582$ 个用例是极其费力的，也不符合自动化测试的思想。因此，为了实现自动化测试这一目标，我们选择了两种脚本语言：Shell 脚本和 Python 脚本。这两种脚本语言各自具有独特的优势，使我们能够更有效地完成实验任务。

1. Shell 脚本：

- Shell 脚本在处理文件操作和系统命令方面具有优势，使我们能够轻松地执行用例并将结果输出到不同的文件夹下。
- Shell 脚本可以直接在实验环境的 Ubuntu 操作系统的 Shell 中运行，方便快捷。
- Shell 脚本在执行简单任务时性能较好，适合我们实验中的大部分操作。

2. Python 脚本：

- Python 具有丰富的内置库和第三方库。这使得 Python 在处理字符串和文件读写方面非常强大。
- Python 具有简洁明了的语法，易于编写和阅读。

- Python 具有很好的跨平台兼容性，我们对 Python 代码的编写与测试既可以在本地电脑，也可以在虚拟机中进行。

在实验过程中，我们首先使用 Python 脚本生成 .sh 类型的 Shell 脚本文件，以便自动执行所有测试用例。之后，我们使用 Python 脚本对测试结果进行提取、处理和分析，生成 .csv 文件以便进行数据可视化。这种脚本语言选择策略的优势在于充分利用了 Shell 脚本和 Python 脚本各自的优势，提高了实验效率;同时，使实验过程更加自动化，减少了人为错误的可能性；并且，这还便于后期实验结果的分析 and 可视化，提高了实验的可读性和准确性。

3.3 实验预处理

在正式开始编写测试脚本和执行测试代码之前，我们需要对实验做一些预处理，包括测试用例命令的纠正、Bug 的植入并编写 Python 代码完成这些预处理。图6展示了实验预处理工作的组织结构。

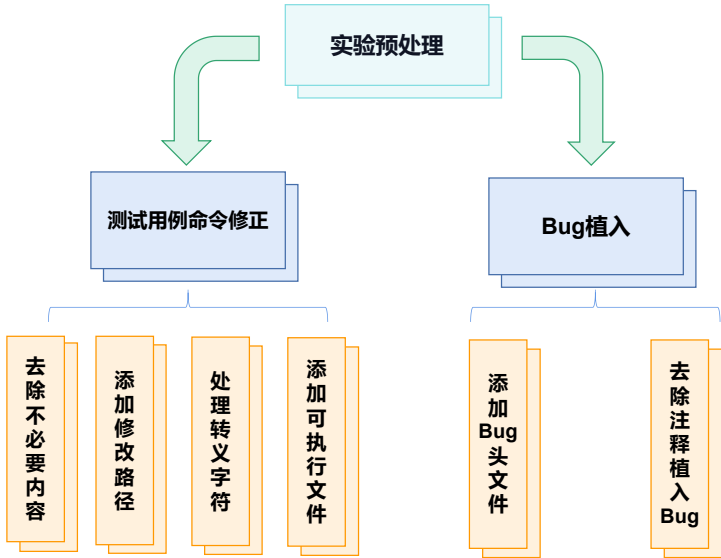


图 6: 实验预处理整体架构

3.3.1 测试用例命令修正

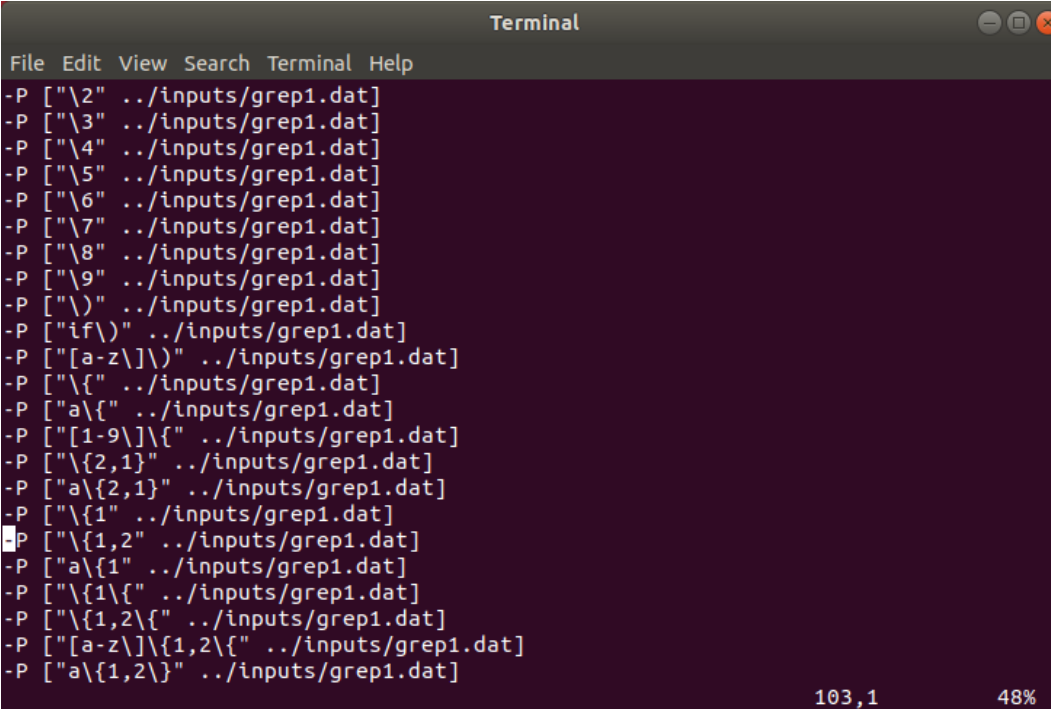
本次实验的测试用例命令存储在 *testplans.alt/v1* 目录下的 *v0.cov.universe* 中。如图7所示，我们需要对其做出如下修改以便测试用例可以正确执行：

- 删除命令中的“-P”参数。图7所示的每一行测试用例命令皆以-P 开头，这在实际执行测试代码时是不需要的，我们将其删除。
- 删除非输入内容中的中括号。同时，每行测试用例命令都由“[]”包围，这在实际执行测试代码时同样不需要，但其中的个别用例输入本身包含“[]”或“””，这些符号需要保留，否则就改变了原本的测试用例。因此，我们需要在删除不必要中括号的同时不改变测试输入里的中括号。
- 修改测试用例命令中的目录。在我们的实验目录结构中，*grep1.dat* 文件处在 *grep/inputs/* 目录下，与测试用例命令中的 *../inputs/* 不符，需要改正。
- 处理测试用例中的转义字符。由于部分测试用例输入中含有“\”符号，而这在 Shell 脚本语言中会被理解为转义字符，因此我们需要在“\”前添加转义字符使其能够被正确理解为符号本身。
- 添加可执行文件的命令。要正确执行测试代码，我们需要在每行测试用例命令前添加可执行文件名 *./grep.exe*。
- 添加测试结果输出路径。由于最终需要对测试代码的执行结果进行分析对比，因此我们为每次测试的执行结果指定各自的输出路径。

3.3.2 Bug 植入

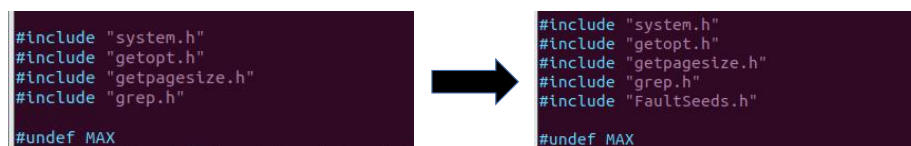
我们在 *grep.c* 文件添加 *FaultSeeds.h* 头文件以植入 Bug。

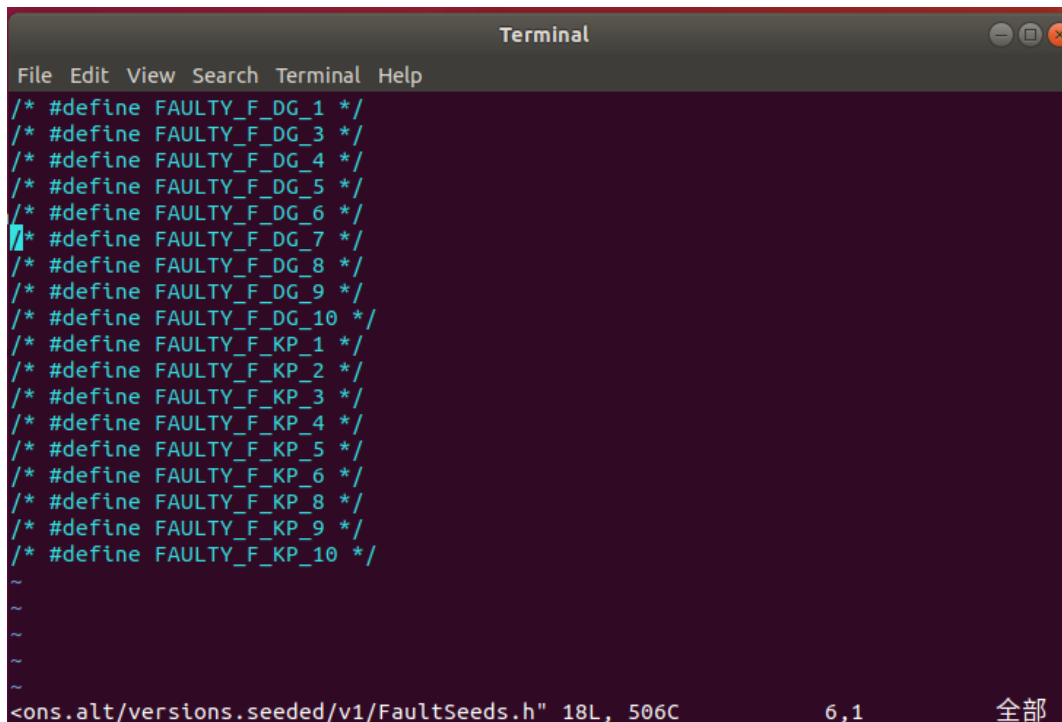
如图9所示，在 *FaultSeeds.h* 文件中，存储了 18 个被注释掉的 Bug。在正式的测试过程中，我们需要分别去掉这 18 行代码的注释来向被测试的程序中植入 18 个不同的 Bug，最后执行含有这些 Bug 的程序并统计测试用例的通过率。



```
Terminal
File Edit View Search Terminal Help
-P ["\2" ../inputs/grep1.dat]
-P ["\3" ../inputs/grep1.dat]
-P ["\4" ../inputs/grep1.dat]
-P ["\5" ../inputs/grep1.dat]
-P ["\6" ../inputs/grep1.dat]
-P ["\7" ../inputs/grep1.dat]
-P ["\8" ../inputs/grep1.dat]
-P ["\9" ../inputs/grep1.dat]
-P ["\" ../inputs/grep1.dat]
-P ["if\" ../inputs/grep1.dat]
-P ["[a-z\\\""] ../inputs/grep1.dat]
-P ["\" ../inputs/grep1.dat]
-P ["a\" ../inputs/grep1.dat]
-P ["[1-9\\\""] ../inputs/grep1.dat]
-P ["{2,1}" ../inputs/grep1.dat]
-P ["a{2,1}" ../inputs/grep1.dat]
-P ["{1}" ../inputs/grep1.dat]
-P ["{1,2}" ../inputs/grep1.dat]
-P ["a{1}" ../inputs/grep1.dat]
-P ["{1\\\""] ../inputs/grep1.dat]
-P ["{1,2\\\""] ../inputs/grep1.dat]
-P ["[a-z\\\"{1,2\\\""] ../inputs/grep1.dat]
-P ["a{1,2\\\""] ../inputs/grep1.dat]
103,1 48%
```

图 7: v0.cov.universe 中的测试用例

图 8: 添加 *FalutSeeds.h* 头文件



```
Terminal
File Edit View Search Terminal Help
/* #define FAULTY_F_DG_1 */
/* #define FAULTY_F_DG_3 */
/* #define FAULTY_F_DG_4 */
/* #define FAULTY_F_DG_5 */
/* #define FAULTY_F_DG_6 */
/* #define FAULTY_F_DG_7 */
/* #define FAULTY_F_DG_8 */
/* #define FAULTY_F_DG_9 */
/* #define FAULTY_F_DG_10 */
/* #define FAULTY_F_KP_1 */
/* #define FAULTY_F_KP_2 */
/* #define FAULTY_F_KP_3 */
/* #define FAULTY_F_KP_4 */
/* #define FAULTY_F_KP_5 */
/* #define FAULTY_F_KP_6 */
/* #define FAULTY_F_KP_8 */
/* #define FAULTY_F_KP_9 */
/* #define FAULTY_F_KP_10 */
~
~
~
~
<ons.alt/versions.seeded/v1/FaultSeeds.h" 18L, 506C 6,1 全部
```

图 9: 18 个被注释的 Bug

3.3.3 预处理代码编写

考虑到本实验共包含 199 个测试用例和 18 个 Bug，若手动修改测试用例命令和去除 *FaultSeeds.h* 中的注释，整个预处理工作将会十分繁琐，因此我们利用 Python 语言编写代码来自动化预处理过程。

首先，我们对测试用例命令进行修正，如下方代码所示，打开 *v0.cov.universe* 文件，对其逐行修改：

Code 1: handle.sh pt.2

```
1 allLines=[]
2 with open('./grep_1.2/grep/testplans.alt/v1/v0.cov.universe', '
  r') as f:
3     row=0
4
5     for line in f.readlines():
6         allLines.append(line)
7         row+=1
8         line=line.replace("../inputs", "../../../inputs")
```

```
9      line=line.replace("-P ", " ") # remove "-P "  
10     line=line.replace("]\n", " ")  
11     line=line.replace(" [", " ")  
12     line=line.replace("] ", " ")  
13  
14     line=line.replace("\\", "\\\\"") #shell: \ => \  
15  
16     line="./grep.exe "+line #add instruction ./grep.exe  
17     line=line+" > ./outputs/$i/t"+str(row)+";" # add  
        output path  
18     content+=line
```

代码解释:

- 修改目录。将 `../inputs` 替换为 `../../../inputs`;
- 删除不必要参数及符号。将 `-P` 参数和非测试输入中的中括号删除。注意到测试输入中的中括号前后一定为非空字符, 因此替换所有空格及换行符相邻的中括号为空格;
- 处理转义字符。由于 Python 语言中“\”同样表示转义, 因此 Python 代码中 2 个“\”对应 shell 命令中的 1 个“\”, 4 个“\”才能表示最终的“\”字符本身。
- 添加可执行文件。在每行命令的开头添加 `./grep.exe`;
- 添加输出路径。指定输出路径为 `./outputs/$i/t`+`str(row)`, 其中 `i` 为 Bug 序号, `row` 为测试用例序号, 即植入第 `i` 个 Bug 的情况下, 第 `row` 个测试用例的测试结果。

此外, 我们编写植入 Bug 的脚本命令, 同时将其和所有修正后的测试用例命令写入同一个脚本文件, 这样我们在完成预处理工作的同时生成了自动执行测试程序的脚本文件 `handle.sh`, 后续的正式测试过程可以直接运行此脚本得到测试结果。如下所示, `content` 初始内容为一些 shell 命令, 并逐行添加修正后的测试用例命令, 最终形成正式测试过程所运行的 `handle.sh` 脚本文件。

Code 2: handle.sh pt.1

```
1 content="\
2     echo 'Lines executed,Branches executed,Taken at least once,
3     Calls executed,' > cov_percentages.csv;\
4 mkdir outputs;\
5 for((i=1;i<=18;i++));do mkdir ./outputs/$i ;done;\
6 sed '' FaultSeeds.h > FaultSeeds;\
7 for((i=1;i<=18;i++));\
8     do sed $i's###g' FaultSeeds > tmp;sed $i's#/#g' \
        tmp > FaultSeeds.h;gcc -fprofile-arcs -ftest-coverage -
        I. -o grep.exe grep.c;"
```

代码解释：

- **创建表头。**我们将测试出来的代码覆盖率结果写入 *cov_percentages.csv* 以便进行统计和分析，此处为统计表格添加表头；
- **创建输出目录。**所有测试结果输出到 *outpus* 目录下，*outpus* 的每个子目录 *i* 为第 *i* 个 Bug 对应的输出目录。
- **去除注释植入 Bug。**我们利用 *sed* 命令将对应行的 * 和 / 替换为空字符。另外，我们要执行两次 *sed* 命令，且 *sed* 命令替换后的结果不会保存到原文件，因此我们将第一次 *sed* 后的结果输出到临时文件 *tmp*，第二次再输出回到 *FaultSeeds.h* 文件。每次植入 Bug 不仅需要去除当前行的代码注释，也需要将上一行的 Bug 重新加上注释，因此我们在最开始将 *FaultSeeds.h* 备份到 *FaultSeeds* 文件，每次去注释植入 Bug 则基于 *FaultSeeds* 进行。

sed 是一种在线编辑器，处理时，把当前处理的行存储在临时缓冲区中，称为“模式空间”（pattern space），接着用 *sed* 命令处理模式空间中的内容，处理完成后，把缓冲区的内容送往屏幕。文件内容并没有改变，因为这些都在模式空间处理^a。使用 *sed* 命令进行替换操作的基本语法格式为：*sed s/目标字符串/替换后的字符串/g* 文件名

其中 / 为分隔符，也可以使用 # 和 @，g 表示替换所有的目标字符串，若不添加 g 则表示只替换第一个匹配的目标字符串。

^a<https://www.gnu.org/software/sed/manual/sed.html>

- 重新编译植入 **Bug** 后的程序。每次植入 **Bug** 后需要重新编译 *grep.c* 代码，以便可执行文件在对应 **Bug** 下执行。

3.4 对正确代码进行测试

3.4.1 实验原理

gcov 是一个测试代码覆盖率的工具。与 **GCC** 一起使用来分析程序，以帮助创建更高效、更快的运行代码，并发现程序的未测试部分。使用 **gcov** 主要完成以下三项工作：

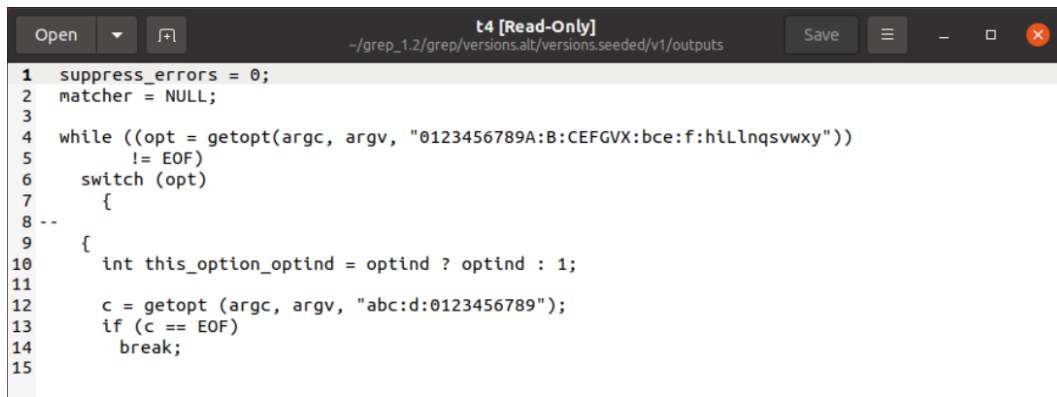
- 在编译阶段产生 **gcn** 和 **gcda** 文件。如果需要开启 **gcov** 功能，需要在测试开始之前，使用 `-fprofile-arcs -ftest-coverage` 命令对代码进行插桩。其中 `-fprofile-arcs` 命令表示在编译时产生 **gcn** 文件，它包含了重建基本块图和相应的块的源码的行号的信息。`-fprofile-arcs` 命令表示在编译时产生 **gcda** 文件，它包含了弧跳变的次数等信息。
- **gcov** 收集代码运行信息。首次运行时，会生成 **gcda** 文件，记录了代码基本块的执行次数。
- 生成 **gcov** 代码覆盖率报告。利用脚本 **handle.sh** 循环运行完测试用例输入并利用 **gcov** 命令生成 **gcov** 文件，包含了代码覆盖率报告。¹

3.4.2 数据提取

观察运行某一条正则表达式的输出结果，如图10所示：

我们可以看到该程序的输出为一串代码。猜测该代码串可能是该程序的输出结果，也可能为翻译某正则表达式的代码。同时也可以知道，该代码串与我们

¹<https://blog.csdn.net/yanxiangyfg/article/details/80989680>



```

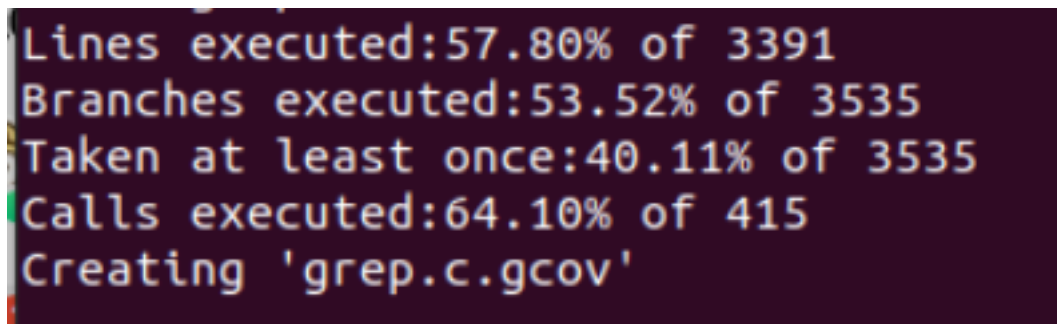
1 suppress_errors = 0;
2 matcher = NULL;
3
4 while ((opt = getopt(argc, argv, "0123456789A:B:CEFGVX:bce:f:hiLlnqsvwxy"))
5 != EOF)
6     switch (opt)
7     {
8 --
9     {
10         int this_option_optind = optind ? optind : 1;
11
12         c = getopt (argc, argv, "abc:d:0123456789");
13         if (c == EOF)
14             break;
15

```

图 10: 正则表达式的运行结果

的测试实验关系不大，最终只需使用 `diff` 指令判断输出正确与否。

对代码进行测试后，我们得到的结果如图11所示：



```

Lines executed:57.80% of 3391
Branches executed:53.52% of 3535
Taken at least once:40.11% of 3535
Calls executed:64.10% of 415
Creating 'grep.c.gcov'

```

图 11: 正确代码测试实验结果

最终代码的语句覆盖率为 57.80%，分支覆盖率为 53.52%，条件覆盖率为 40.11%，函数覆盖率为 64.10%。

我们希望通过 `shell` 命令和正则表达式自动化地提取这些数据。我们发现上述数据可以用正则表达式 `[0-9].[0-9]%` 表示，且这些数据均不为 0.00%（0.00% 同样能匹配上述正则表达式，且命令行会有这个输出，但它不被我们需要），因此我们使用 `shell` 命令 `grep` 提取以上数据。具体代码如下：

```

1 gcov -b grep.c | grep -oE '[0-9]+\.[0-9]+' | grep -v '0.00' |
  tr '\n' ',' >> cov_percentages.csv; echo ',' >>
  cov_percentages.csv;

```

代码解释：-o 表示只显示匹配 `pattern` 的部分，-E 表示将 `pattern` 当作正则表达式使用，-v 表示不匹配 0.00，后续部分表示数据与数据之间使用逗号分隔，语

句提取结束后换行。最终将这些数据存储在 csv 文件中。

3.4.3 数据可视化

查阅资料可知，如果在每条测试数据后都对其覆盖率输出，得到的覆盖率为累计覆盖率而不是单个语句的覆盖率。资料如下所示：

- 执行计数是累积的。如果在未删除.gcda 文件的情况下再次执行示例程序，则源中每行执行次数的计数将添加到上一次运行的结果中。这在几个方面可能很有用。例如，它可用于在测试验证套件的一部分期间累积多个程序运行的数据，或者在大量程序运行中提供更准确的长期信息。
- .gcda 文件中的数据在程序退出之前立即保存。对于使用-fprofile-arcs 编译的每个源文件，分析代码首先尝试读取现有的.gcda 文件；如果文件与可执行文件不匹配（基本块计数的数量不同），它将忽略文件的内容。然后，它添加新的执行计数，最后将数据写入文件。

我们在测试每条正则表达式后，都对其累计覆盖率进行输出。部分 shell 命令代码如下所示：

```
1 ./grep.exe -0 12345 ../../../../inputs/grep1.dat > ./outputs/t1;  
2 gcov -b grep.c | grep -oE '[0-9]+\.[0-9]+' | grep -v '0.00' |  
   tr '\n' ',' >> cov_percentages.csv;  
3 echo '' >> cov_percentages.csv;  
4 ./grep.exe -1 12345 ../../../../inputs/grep1.dat > ./outputs/t2;  
5 gcov -b grep.c | grep -oE '[0-9]+\.[0-9]+' | grep -v '0.00' |  
   tr '\n' ',' >> cov_percentages.csv;  
6 echo '' >> cov_percentages.csv;
```

最终得到覆盖率随样例变化的数据。我们用折线图来表示，如图12所示。

从图中可以看出，四种覆盖率的增长具有相同的趋势，且保持函数覆盖率 > 语句覆盖率 > 分支覆盖率 > 条件覆盖率。从增长的缓急方面来看，所有覆盖率在前 40 个样例时增长较为迅速（有 20% 到 25% 的提升），而在测试后 160 个样例时增长缓慢。

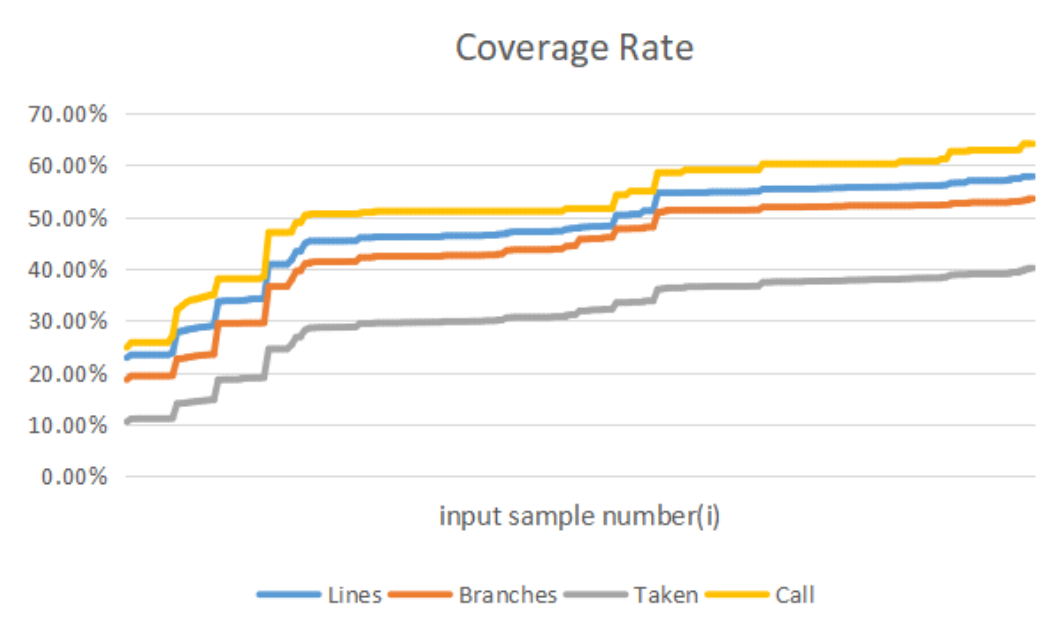


图 12: 累计覆盖率-样例数目折线图

3.5 对植入 Bug 的代码进行测试

3.5.1 实验原理

grep 项目为我们提供了 18 个 bug，它为我们存储在 FaultSeeds.h 文件中。将注释取消后便可以向代码中植入一个 bug。该部分主要是使用 gcov 命令对植入 bug 后的代码进行测试，得到其四种覆盖率并对其进行分析。部分实验的 shell 程序如下所示：

```

1  # 在根目录建立文件夹outputs存放结果
2  mkdir outputs;
3  # 为每一个bug建立一个文件夹存放结果
4  for((i=1;i<=18;i++));do mkdir ./outputs/$i ;done;
5  # 用变量FaultSeeds存放文件内容（用于实验结束还原文件）
6  sed ' ' FaultSeeds.h > FaultSeeds;
7  # 移除当前测试bug的 '\ ' 和 '*' （去除注释）
8  for((i=1;i<=18;i++));
9  do sed $i's#*##g' FaultSeeds > tmp;
10 sed $i's#/#g' tmp > FaultSeeds.h;
11 # 插桩，开始测试

```

```
12 gcc -fprofile-arcs -ftest-coverage -I. -o grep.exe grep.c;  
13 .....
```

3.5.2 数据提取

本部分数据提取同样采取 `grep` 语句和正则表达式匹配的方法，与 4.4 部分相同，因此在这里不再赘述。以下为数据提取的结果：

表 1: 植入 Bug 代码的覆盖率

Bug id	Lines executed	Branches executed	Taken at least once	Calls executed
1	57.82%	53.52%	40.11%	64.10%
2	57.82%	53.52%	40.11%	64.10%
3	57.44%	53.24%	39.75%	62.65%
4	57.82%	53.52%	40.11%	64.10%
5	57.82%	53.52%	40.11%	64.10%
6	57.82%	53.52%	40.11%	64.10%
7	57.82%	53.52%	40.11%	64.10%
8	57.84%	53.55%	40.14%	64.10%
9	57.82%	53.58%	40.11%	64.10%
10	57.82%	53.50%	40.08%	64.10%
11	39.85%	39.04%	26.45%	43.37%
12	57.82%	53.52%	40.11%	64.10%
13	57.82%	53.52%	40.11%	64.10%
14	57.68%	53.52%	40.08%	63.13%
15	57.82%	53.52%	40.11%	64.10%
16	57.82%	53.52%	40.11%	64.10%
17	57.82%	53.52%	40.11%	64.10%
18	57.77%	53.52%	40.11%	64.10%

从表中我们可以看出，除了 Bug11 之外，其余 17 个 bug 的四种覆盖率均与正确情况下相似。而 bug11 的覆盖率比其余 17 组数据低了约 15-20 个百分点。因此，可以猜想 bug11 错误比较严重，可能在后面 diff 时有着较高的错误率。

3.5.3 数据可视化

对上表中的数据进行可视化展示，可以更加直观的验证上述结论。可视化结果如图13所示。

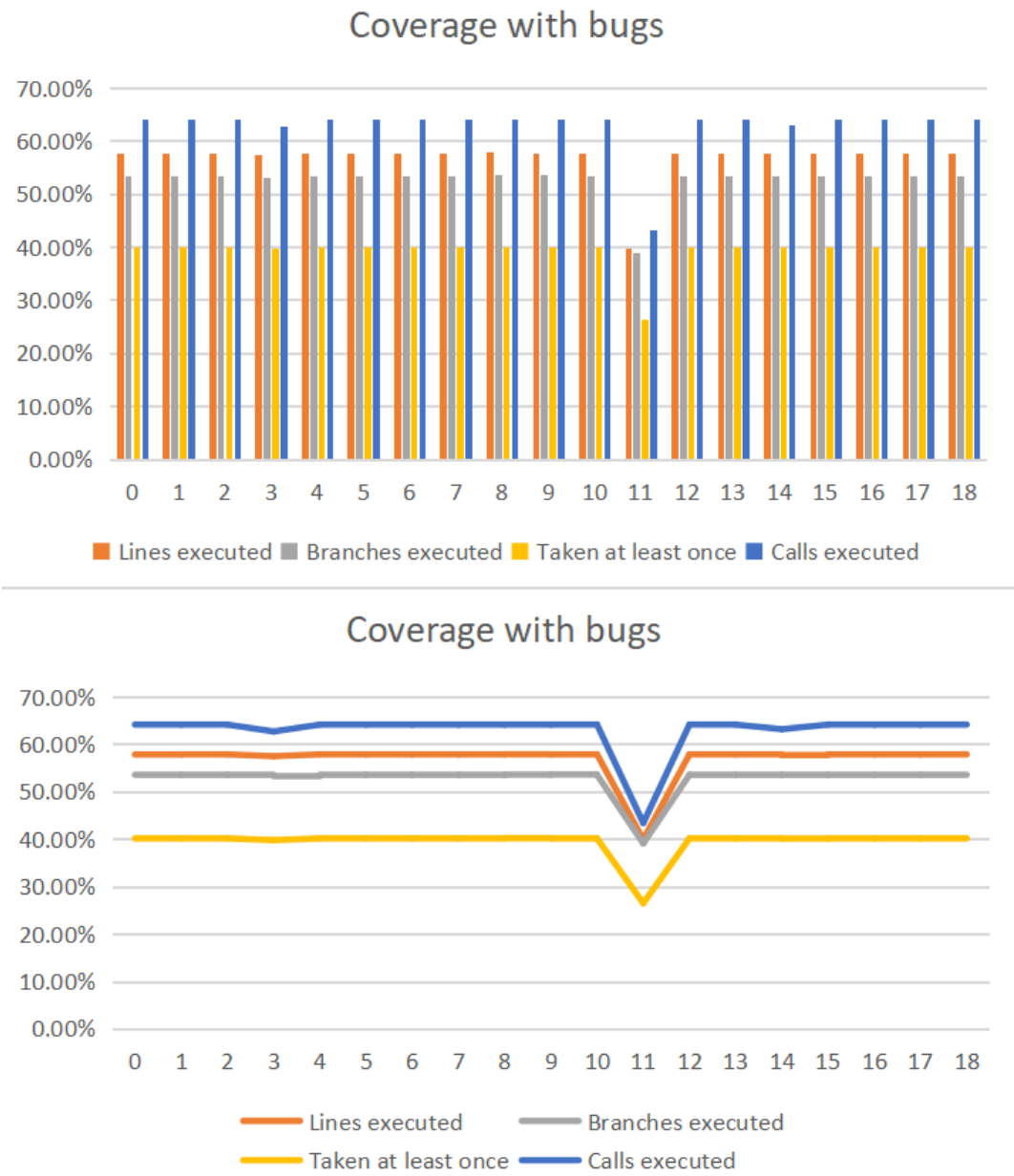


图 13: 植入 Bug 后的代码覆盖率

从表中也可以直观的看出以上结论。其中函数覆盖 > 语句覆盖 > 分支覆盖 > 条件覆盖的结论仍然成立。

3.6 对正确代码和植入 bug 代码的测试结果进行比较

3.6.1 实验原理

在生成了正确的结果文件以及分别植入了 18 个 bug 的结果文件之后，实验的最后一步是要将分别植入了 18 个 bug 的结果文件与正确的结果文件进行比较，从而得知每个 bug 针对每个测试用例是否通过，若结果一致则说明未检测到 bug。

首先我们需要生成一个比较正确版本结果和 bug 版本结果的脚本文件，使用 python 编写，具体代码如下：

```
1 # 新建脚本文件diffAll.sh
2 with open('diffAll.sh','w') as f:
3     #cover the history
4     # 写入新建result文件夹的命令
5     f.write('mkdir result;\n\
6 # 写入result文件夹下新建1~18的文件夹的命令
7 for((i=1;i<=18;i++));do mkdir ./result/$i ;done;\n')
8
9 with open('diffAll.sh', 'a') as f:
10     # 对于每一个bug版本
11     for bugCnt in range(1,19):
12         # 生成对应的199个测试结果的比较命令
13         f.write(f"echo '-----diff bug{bugCnt
14             }-----'\n")
15         for caseCnt in range(1,200):
16             # 比较该bug版本的该测试用例的结果的命令
17             f.write(f"echo 'diff case {caseCnt} in {bugCnt}' \n
18                 ")
19             # 将比较结果保存到对应的比较结果文件当中的命令
20             f.write(f"diff ./outputs/t{caseCnt} ../../versions.
21                 seeded/v1/outputs/{bugCnt}/t{caseCnt} \
22                 >./result/{bugCnt}/r{caseCnt} \n")
```

```
22 content="mkdir result; \n\  
23     for((i=1;i<=18;i++));do mkdir ./result/$i ;done; \n"
```

我们用 `diff` 命令完成这个比较，将比较后的结果保存到 `result` 目录下，然后分析结果是否为空，若为空，则两个文件一致，依次比较完 18 个版本的输出，写成脚本文件 `diffAll.sh`，生成的部分脚本展示如下：

```
1  # 建立文件夹result存放比较结果  
2  mkdir result;  
3  # 在result文件夹下为每一个bug建立一个文件夹存放199个结果文件的  
   比较结果  
4  for((i=1;i<=18;i++));do mkdir ./result/$i ;done;  
5  # 对于第一个bug  
6  echo '-----diff bug1-----'  
7  # 会与第一个bug的第一个测试用例（输入）  
8  echo 'diff case 1 in 1'  
9  # 比较第一个输入的正确的输出文件  
10 # 和植入了第一个bug的第一个输入的输出文件  
11 # 保存比较结果  
12 diff ./outputs/t1 ../../versions.seeded/v1/outputs/1/t1  
   >./result/1/r1  
13 # 与上一个类似  
14 echo 'diff case 2 in 1'  
15 diff ./outputs/t2 ../../versions.seeded/v1/outputs/1/t2  
   >./result/1/r2  
16 echo 'diff case 3 in 1'  
17 diff ./outputs/t3 ../../versions.seeded/v1/outputs/1/t3  
   >./result/1/r3  
18 .....
```

3.6.2 数据提取

在获得了 18 个 bug 的 199 个输入用例与正确结果的比较结果文件之后，需要将每个 bug 具体对某一个测试用例是否通过的情况提取出来，于是编写了 python

文件 `extractDiffData_per.py`，并将结果保存在 `csv` 文件中，具体程序如下所示：

```
1 # 每一列代表每一个bug的版本，每一行则为同一个测试用例输入分别到
   18个bug的版本
2 with open("dif_percentages_per.csv", "w") as f:
3     row=["epoch_"+str(bugCnt) for bugCnt in range(1,19)]
4     rowStr=",".join(row)
5     f.write(rowStr+"\n")
6
7 with open("dif_percentages_per.csv", "a") as f_csv:
8     # 对每一个测试用例
9     for caseCnt in range(1,200):
10         isFail=[] # 用来存储各个bug版本是否通过该测试用例
11         # 对于每个bug的版本
12         for bugCnt in range(1,19):
13             filename=f"./result/{bugCnt}/r{caseCnt}"
14             # 如果比较结果文件为空，则说明未检测到bug
15             if(os.path.getsize(filename)==0):
16                 isFail.append("0")
17             # 比较结果不为空，则说明检测到bug
18             else:
19                 isFail.append("1")
20         isFailStr=",".join(isFail)
21         # 将结果写入csv文件中
22         f_csv.write(f"{isFailStr}\n")
```

为了更好的对比较结果进行分析，我们需要统计出每个 `bug` 版本分页对用通过和未通过的测试用例个数，因此编写了 `python` 文件用于进一步提取上一步生成的 `csv` 文件当中的数据内容，具体程序如下所示：

```
1 import os
2 # 每一列代表每一个bug的版本
3 # 每一行则为同一个测试用例输入分别到18个bug的版本
4 with open("dif_percentages_per.csv", "w") as f:
5     row=["epoch_"+str(bugCnt) for bugCnt in range(1,19)]
6     rowStr=",".join(row)
```



```
7     f.write(rowStr+"\n")
8
9 with open("dif_percentages_per.csv","a") as f_csv:
10 import os
11 # 生成一个csv文件保存数据提取结果
12 with open("dif_percentages.csv","w") as f:
13     f.write("Bug epoch,Fail case num\n")
14
15 # 打开上一步生成的文件
16 with open("dif_percentages.csv","a") as f_csv:
17     # 对于每一个bug版本
18     for bugCnt in range(1,19):
19         failNum=0    # 统计该bug版本未通过的测试的个数
20         # 对每一个测试用例
21         for caseCnt in range(1,200):
22             # 若文件为空说明通过测试，否则未通过
23             filename=f"./result/{bugCnt}/r{caseCnt}"
24             if(os.path.getsize(filename)!=0):
25                 failNum+=1
26         # 记录数据
27         f_csv.write(f"{bugCnt},{failNum}\n")
```

3.6.3 数据可视化

根据上一步得到的数据进行可视化，如图 6 所示。

由图14可知，大部分 bug 版本的程序都通过了所有的测试用例，只有第 3 和 14 有极少量测试用例未通过，第 11 个 bug 版本有大量测试用例未通过。

再分别对第 3 个、第 11 个和第 14 个 bug 版本的测试用例进行分析，具体结果如图15所示，图中纵坐标为 0 表示测试通过，1 则表示测试未通过。

具体来说，bug3 的版本有 13 个输入未通过测试，bug11 的版本有 170 个输入未通过测试，bug14 的版本有 4 个未通过测试。



图 14: bug 版本-通过测试和未通过数目的柱状图

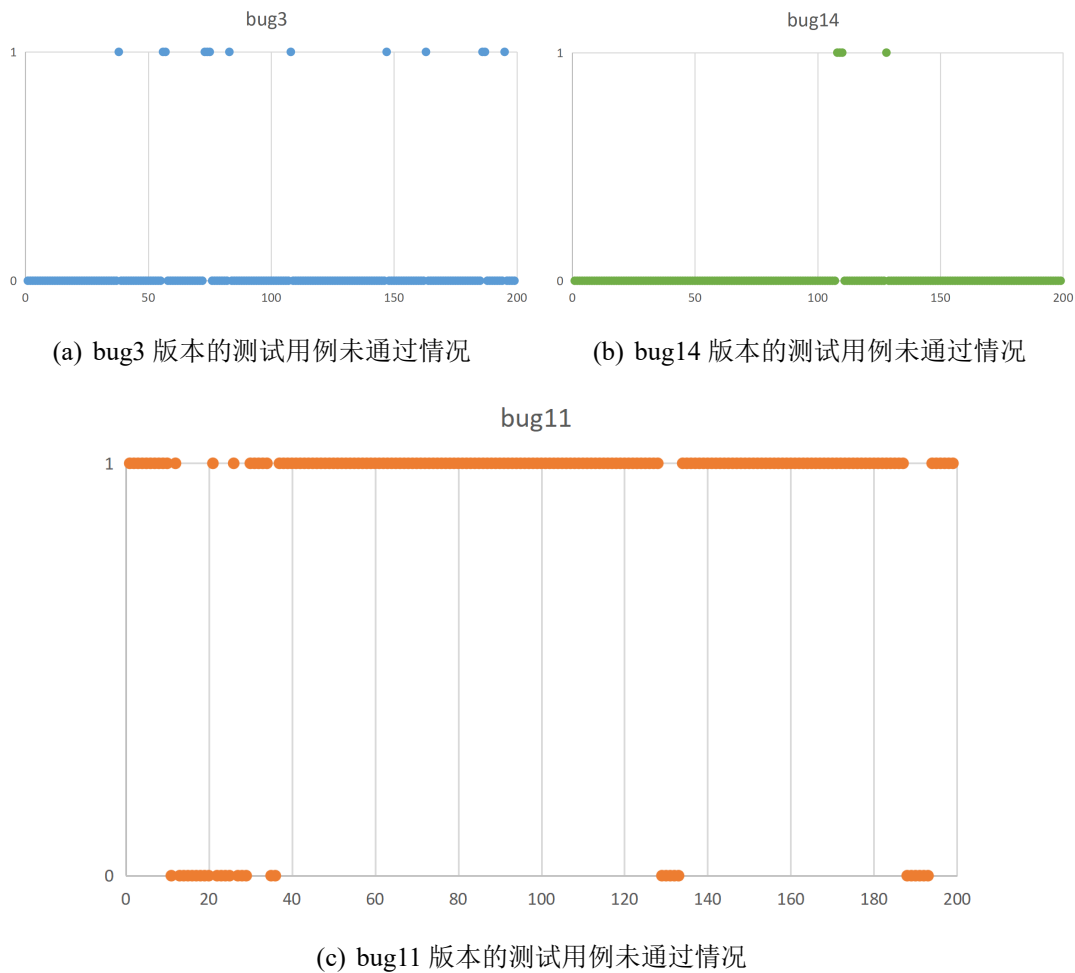


图 15: 三个有测试用例未通过的 bug 版本的散点图

4 实验总结

首先我们对实验进行了预处理，包括对测试用例的命令进行修正、将 bug 进行植入以及编写预处理代码，这里为后面的主要实验部分做好了基础准备，在主要的实验部分，总共有三个核心步骤，分别是对正确代码进行测试、对植入 bug 的代码进行测试以及对正确代码和植入 bug 代码的测试结果进行比较。

在对正确的的代码进行测试的部分，我们分别得到了测试的语句覆盖率、分支覆盖率和条件覆盖率。在对植入 Bug 的代码进行测试部分，我们通过分别对植入 bug 的头文件进行取消注释来植入每一个 bug，最终获得了累计覆盖率以及各个 bug 版本的三种覆盖率。最后一步就是对正确代码和植入 bug 代码的测试结果进行比较，通过 diff 命令我们得到了每个 bug 版本对于每一个测试用例是否通过的情况，并继续拧了各种角度的数据分析和可视化。

参考文献

- [1] Mohd Ehmer Khan and Farmeena Khan. Importance of software testing in software development life cycle. *International Journal of Computer Science Issues (IJCSI)*, 11(2):120, 2014.
- [2] Antonia Bertolino. Software testing research: Achievements, challenges, dreams. In *Future of Software Engineering (FOSE'07)*, pages 85–103. IEEE, 2007.
- [3] Glenford J Myers, Corey Sandler, and Tom Badgett. *The art of software testing*. John Wiley & Sons, 2011.
- [4] Mauro Pezzè and Michal Young. *Software testing and analysis: process, principles, and techniques*. John Wiley & Sons, 2008.

附录

填充