论文报告

雷博文

1. 论文题目: ReDeBug: Finding Unpatched Code Clones in Entire OS Distributions

2. 论文链接: https://ieeexplore.ieee.org/document/6234404

3. Redebug 开源链接: https://github.com/dbrumley/redebug

4. 论文概要:

ReDeBug，这是一个用于在 OS 发行规模代码库中快速查找未修补代码克隆的系统，它使用了一种快速、基于语法的方法，可以扩展到 OS 发行版大小的代码库，其中包括用多种不同语言编写的代码。ReDeBug 可以找到更少的代码克隆，但可以提高规模、速度、降低错误检测率，并且不受语言限制。我们通过检查 Debian Lenny/Squeeze、Ubuntu Maverick/Oneiric、所有 SourceForge C 和 C++项目以及 Linux 内核中所有包的所有代码来评估 ReDeBug，以查找未修补的代码克隆。

我们通过检查 Debian Lenny/Squeeze、Ubuntu Maverick/Oneiric、所有 SourceForge C 和 C++项目以及 Linux 内核中所有包的所有代码来评估 ReDeBug，以查找未修补的代码克隆。ReDeBug 以 700000 LoC/分钟的速度处理了超过 21 亿行代码，建立了一个源代码数据库，然后在 8 分钟内在一台商用台式机上检查了 376 个 Debian/Uuntu 安全相关补丁，在当前部署的代码中发现了 15546 个已知易受攻击代码的未修补副本。我们通过确认最新版本的 Debian Squeze 包中的 145 个真实 bug，展示了 ReDeBug 的真实影响。

5. 论文内容分析总结:

Ⅰ 引言

总的来说，我们的主要贡献是：

•我们分析了整个 OS 发行版，以了解未修补代码克隆的当前趋势。据我们所知，ReDeBug 是第一个探索超过 21 亿行完整 OS 发行版以了解未修补代码克隆问题的工具。

•我们描述了 ReDeBug，它在可伸缩性、速度和错误检测率方面为代码克隆检测提供了一个新的设计空间。特别是，该设计点使 ReDeBug 在日常环境中的典型开发人员能够使用，以便通过快速查询已知漏洞来提高代码的安全性。

•我们提供了 OS 发行版中复制代码总量的第一个经验测量。这表明，在未来，未修补的代码克隆将继续重要且相关。

Ⅱ ReDeBug

A 核心系统

ReDeBug 系统的核心通过以下步骤实现：

①ReDeBug 规范化每个文件。默认情况下，ReDeBug 删除典型的语言注释，删除所有非 ASCII 字符，删除多余的空格（新行除外），并将所有字符转换为小写。如果文件是 C、C++、Java 或 Perl（通过扩展名或 UNIX 文件命令标识），我们也会忽略大括号。

标准化是模块化的，因此可以很容易地更改精确的标准化步骤。

②规范化的文件基于新行和正则表达式子字符串进行标记。

③ReDeBug 在记号流上滑动长度为 n 的窗口。每 n 个记号被视为要比较的代码单位。

④对于记号流的计算：

4) Given two sets f_a and f_b of n-tokens, we compute the amount of code in common. When finding unpatched code clones, if f_a is the original buggy code snippet we calculate

$$\text{CONTAINTMENT}(f_a, f_b) = \frac{|f_a \cap f_b|}{|f_a|} \quad (1)$$

When we want to measure the total amount of similarity between files, we calculate the percentage of tokens in common (i.e., the Jaccard index):

$$\text{SIMILARITY}(f_a, f_b) = \frac{|f_a \cap f_b|}{|f_a \cup f_b|} \quad (2)$$

With either calculation it is common to only consider cases where the similarity or containment is greater than or equal to some pre-determined threshold θ. In our implementation, we also perform obvious optimizations such as when $\theta = 1$ only verifying $f_a \subseteq f_b$ instead of calculating an actual ratio for CONTAINMENT.

⑤ReDeBug 对识别出的未修补代码克隆执行精确匹配测试，以删除 Bloom 过滤器错误。ReDeBug 还使用编译器在可能的情况下识别代码克隆何时是死代码。

B 代码克隆检测

ReDeBug 用于检测错误代码克隆的总体步骤如下图所示：

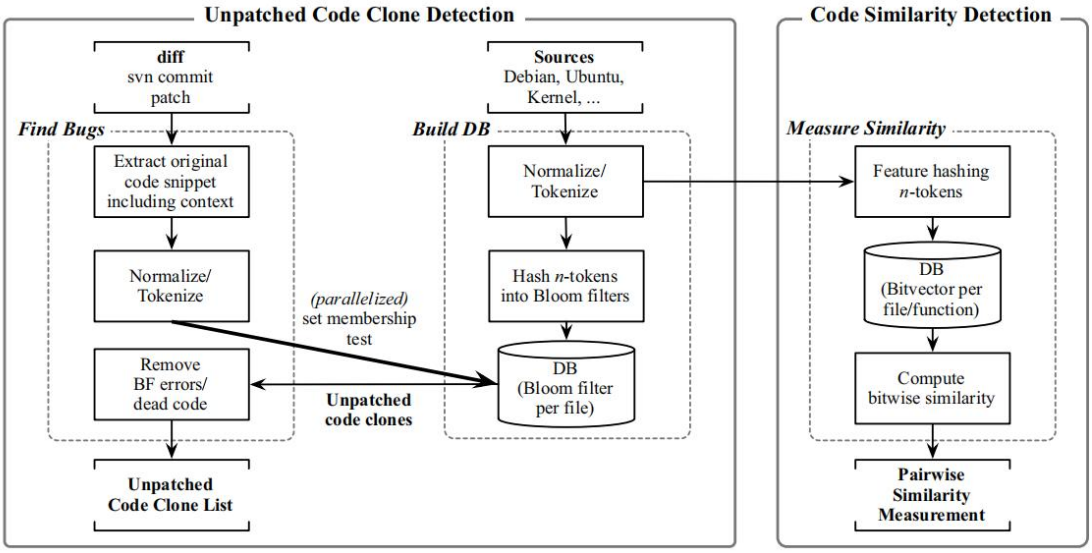


Figure 2: The ReDeBug workflow.

•步骤 1：预处理源。用户获取其分发中使用的所有源文件。对于 Debian，这是使用 apt 工具完成的。ReDeBug 然后自动：

1)按照 II-A 中的核心系统所述执行规范化和标记化。

2)在标记流上移动 n 长度窗口。对于每个窗口，生成的 n 标记被散列到 Bloom 过滤器中。

3)以原始数据格式存储每个源文件的 Bloom 筛选器。ReDeBug 在存储到磁盘之前压缩

Bloom 过滤器，以节省空间并减少查询时的磁盘访问量。

•步骤 2：检查未修补的代码副本。用户获得统一的 diff 软件补丁。ReDeBug 然后自动：

1)从预补丁源中提取原始代码片段和上下文信息的 c 标记。代码段的机制很简单：我们在补丁中提取以“-”符号为前缀的行（添加了以“+”符号为后缀的行，因此在原始 bug 代码中不存在）。如果补丁中提供了上下文信息，我们使用否则我们从原始源文件中获取。

2)对提取的原始错误代码片段进行规范化和标记化。标准化过程与步骤 1 中描述的相同。对于 C、C++、Java 和 Python，我们删除了 C 上下文行中的任何部分注释，因为这些语言支持多行注释，而 C 上下文行可能只有多行注释的头部或尾部。

3)将 n 记号窗口散列为一组散列 fp.

4)对每个哈希 n 记号窗口执行 Bloom 筛选器集成员资格测试。如果 CONTAIN-MENT（fp，f）≥θ，我们报告一个带有文件 f 的未修补代码克隆。

•步骤 3：对报告的克隆进行后处理。给定报告的未修补代码克隆，ReDeBug 会自动：

1)执行精确匹配测试以删除 Bloom 筛选器错误。

2)排除生成时未包含的死代码。对于 C，我们将 assert 语句添加到错误代码区域，并使用-g 选项进行编译，该选项允许我们使用 objdump-S 检查 assert 语句的存在。对于非编译语言，省略此步骤。

ReDeBug 只向用户报告非死代码。

C 代码相似性检测

ReDeBug 也可以用来衡量代码克隆的数量。在此模式下，ReDeBug 在代码对之间使用相似性度量。执行相似性测量时的主要问题是成对比较的成本。ReDeBug 使用位向量来加速逐对比较。（具体内容略）

D 相似性与错误查找

查找错误的算法与相似性检测的算法相似，只有少数例外。

Ⅲ 实施与评估及以后

一些对于 redebug 的实验、分析等等，对于我们实用性不大，就不在这里列出了。

6. Redebug 的使用

在 gitbub 下载它的代码如下：

这是在 Python 的版本，并且不同于在论文中中使用的，原始且更快的 C 版本。

用法：

Please refer to the help message for options:

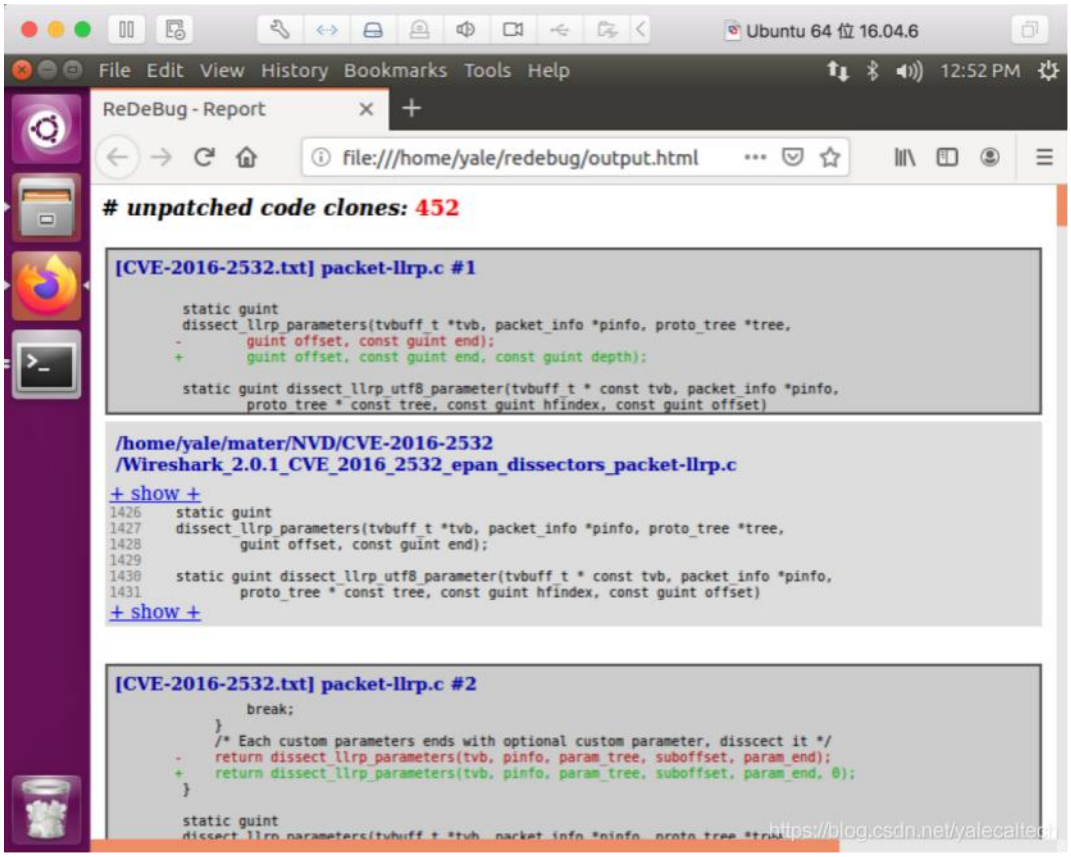
```
$ python redebug.py -h
usage: redebug.py [-h] [-n NUM] [-c NUM] [-v] patch_path source_path

positional arguments:
  patch_path            path to patch files (in unified diff format)
  source_path           path to source files

optional arguments:
  -h, --help            show this help message and exit
  -n NUM, --ngram NUM   use n-gram of NUM lines (default: 4)
  -c NUM, --context NUM
                        print NUM lines of context (default: 10)
  -v, --verbose         enable verbose mode (default: False)
```

需要补丁文件 + 源代码文件作为输入，输出为 output.html，打开该网站会显示有多少个没打补丁的克隆以及对应位置。

可参考的链接：https://blog.csdn.net/yalecaltech/article/details/107226303

7. （2.2.5）基于句法敏感的程序分析进行漏洞范围界定方法

ReDeBug 中需要输入：补丁文件和源代码文件，然后可以输出一个 output.html 文件（参考如下图）：



图片来自：

https://blog.csdn.net/yalecaltech/article/details/107226303

在网页中含有具体的报告，提示没打补丁的克隆有 452 处，以第一处为例，上方的 diff 文件可以看到是修改了出现漏洞的那一行，下方是源文件中可以看到没有针对漏洞处打补丁。相关的漏洞位置信息就在网站上面，如果显示有没打补丁的克隆出现那么说明这个版本受到该漏洞影响。

而为了获得这个版本的源代码，就需要结合版本信息中的版本号，可在代码仓库中切换到各版本发布对应的 commit，获取每个版本的源代码；也可以通过开源软件相应官网下载区下载各版本源代码。

所以，本方法以漏洞软件的代码仓库、版本信息以及漏洞补丁为输入，通过代码仓库和版本信息，应该可以实现自动化获取到每个版本的源代码，然后再用这个源代码和漏洞补丁作为 redebug 的输入就可以输出漏洞是否在该版本中存在了。

8. 一些个小问题惹：

①通过 2.2.4 方法得出来的一个 commit 中，修改过的地方可能不止一处，解决的漏洞也可能不止一个，那么对应这个方法中输入源代码+补丁，该补丁中的解决的漏洞也不止一处，那么在这个方法检测的时候怎么知道显示是哪个漏洞影响了这个版本的呢?(难道是这个补丁只解决了这一个漏洞？)

②redebug 是通过补丁来检测某是否存在另外克隆的代码片段还没有被修改过，而这个方法是通过补丁来检测该版本中这个片段是否被修改过，不清楚是否可以直接使用第一种方法来兼容第二种方法，也就是是否可以直接使用原有的 redebug 程序来做检测工作。