软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software, [doi: 10.13328/j.cnki.jos.000000]

E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel: +86-10-62562563

一种利用补丁的未知漏洞发现方法*

李 赞, 边 攀, 石文昌, 梁 彬

(中国人民大学 信息学院,北京 100872)

通讯作者: 梁彬, E-mail: liangb@ruc.edu.cn

摘 要: 近年来,利用含有已知漏洞的函数作为准则,通过查找相似代码实现来检测未知漏洞的方法已经被证明是有效的.但是,一个含有漏洞的函数通常也包含一些与已知漏洞无关的语句,严重影响相似度计算的结果,从而引发误报和漏报.本文提出了一种利用补丁来提高这种相似性检测准确性的漏洞发现新方法.结合漏洞的补丁信息,引入程序切片技术去除原来含有漏洞的函数中与漏洞无关的语句,利用获得的切片生成去噪的漏洞特征来进行潜在未知漏洞检测.该方法已经在一些真实的代码集中实施,并且实验结果证明该方法确实能够有效减弱漏洞无关语句的干扰,达到提高检测准确性的目的.该方法还成功检测到了 3 个新的未知漏洞且已经得到确认.

关键词: 漏洞;补丁;切片;相似性检测
中图法分类号: TP311

中文引用格式: 李赞,边攀,石文昌,梁彬.一种利用补丁的未知漏洞发现方法.软件学报. http://wtww.jos.org.cn/1000-9825/0000.htm

英文引用格式: Li Z, Bian P, Shi WC, Liang B. Leveraging patches to discover unknown vulnerabilities. Ruan Jian Xue Bao/Journal of Software, 2017 (in Chinese). http://www.jos.org.cn/1000-9825/0000.htm

Leveraging Patches to Discover Unknown Vulnerabilities

LI Zan, BIAN Pan, SHI Wen-Chang, LIANG Bin

(School of Information, Renmin University of China, Beijing 100872, China)

Abstract: In the recent years, taking the known vulnerable function as the criteria to retrieve the similar implementation has been proven to be an effective vulnerabilities detection method. However, a vulnerable function often contains some statements that are irrelevant to the vulnerability of interest, which may heavily interfere the similarity computation and lead to false positives and false negatives. In this paper, we present an approach to improve the precision of the retrieval-based vulnerabilities detection by leveraging the patch of the vulnerable function. The program slicing technique is adopted to exclude irrelevant statements from the original vulnerable function according to the patch. A denoised feature vector will be generated from the obtained slice and be used to search the potential unknown vulnerabilities in the code base. The approach has been applied to some real-world projects. The experiment result shows that our approach can effectively reducing the interference of irrelevant statements and improve the detection precision. Three confirmed unknown vulnerabilities are successfully detected from the projects.

Key words: vulnerability; patch; slicing; similarity detection

1 引言

软件代码中的漏洞(Vulnerability)是导致软件出现故障和错误的重要原因,因此漏洞检测一直是软件安全领域的一个研究热点.但是根据 Rice 定理[1],一般情况下一个程序是无法判定另一个程序是否含有漏洞代码的.因此,自动漏洞发现的方法基本都关注于特定漏洞的识别.静态检测技术作为主流的漏洞检测技术之一,已被证

* 基金项目: 国家自然科学基金(91418206)
Foundation item: National Natural Science Foundation of China (91418206)
收稿时间: 0000-00-00; 修改时间: 0000-00-00; 采用时间: 0000-00-00; jos 在线出版时间: 0000-00-00
CNKI 在线出版时间: 0000-00-00

明能够有效检测代码中的漏洞,并且有很多相关工作提出了静态检测特定漏洞的方法[2-10],如检测一些非安全的函数使用等.为了自动检测特定漏洞,这些静态分析方法需要依赖于先验知识也就是编码规则来检测与之违背的代码.而编码规则无论是基于经验人工给出还是利用程序自动提取都可能产生错误,从而导致误报(False Positive),因此传统的静态分析方法往往需要大量的时间进行人工审计.

为了不依赖于程序相关的先验知识,直接跳过传统静态分析方法提出规则的步骤,近年来,研究者们开始关注另一种利用相似性进行静态漏洞检测的思路.即从含有漏洞的代码段出发,检测待测代码中与已知漏洞在特征上相似的代码.该思路基于这样的背景:在软件开发过程中,某些代码的重复率很高,并且对于一个开发团队或开发者来说,其编码的结构和方式也存在一定的规律性,如果一处代码存在漏洞,那么其他结构或功能相似的代码段也可能存在漏洞.因此,从已知漏洞的相似代码中发现未知漏洞的思路有理可依.

目前在相似性检测方面,已经有一些研究成果[11-13],其中比较有代表性的是 Yamaguchi 等人在 2012 年提出的"异常外推"(Vulnerability Extrapolation)方法[11].该方法将函数的抽象语法树(Abstract Semantic Tree, AST)映射到一个特征向量空间,利用机器学习中的潜在语义分析(Latent Semantic Analysis)方法对特征向量进行主成分分析(Principle Component Analysis),提取出主要的 API(Application Programming Interface)使用模式,再利用已知漏洞函数的特征向量分别与其他的特征向量计算相似度,按照与已知漏洞函数相似度的大小对结果进行排序,最后只需对相似度高,次序靠前的部分候选函数进行审计即可.

但是,此类相似性检测的方法也存在一定的局限性.此类方法往往采用含有漏洞的函数整体进行特征提取并用于后续的相似性计算,而函数中与漏洞涉及的语句无关的其他信息就成为了影响漏洞相似度检测的噪声.尤其是当漏洞所在函数越长,包含的语句数目越多时,噪声相对于漏洞相关语句的比例越大,那么噪声对于相似度结果的影响就越大.因此,在进行相似度计算和排序时,既可能产生误报,又可能产生漏报(False Negative).当一个不含有漏洞的函数在噪声部分的特征与已知漏洞函数相似时,就可能因为噪声比例较大而得出较高的相似度从而产生误报;而当一个确实含有相似漏洞的函数,在漏洞相关语句外的噪声语句的特征与已知漏洞函数并不相似时,就可能会因计算的相似度值较低导致排序结果靠后,从而产生漏报.因此,如果不对已知漏洞所在函数中的噪声进行处理,抽取到的函数特征会掺杂有噪声特征,最终影响到检测方法的有效性,也给人工审计增加了难度.图 1 所示的例子就充分证明了含有漏洞函数中的噪声对于相似检测的影响.

```
libavformat/mov.c
static int mov_read_hdlr(MOVContext *c,
                         AVIOContext *pb, MOVAtom atom)
{ […]
  int64_t title_size; […]
  avio_r8(pb); /* version */
  avio_rb24(pb); /* flags */

  /* component type */
  ctype = avio_rl32(pb);
  type = avio_rl32(pb); /*component subtype */
  […]
  if    (type == MKTAG('v','i','d','e'))
  […]
  avio_rb32(pb); /* component  manufacture */
  avio_rb32(pb); /* component flags */
  avio_rb32(pb); /* component flags mask */

  title_size = atom.size - 24;
  if (title_size > 0) { […]
    title_str = av_malloc(title_size + 1);
    if (!title_str)
      return AVERROR(ENOMEM);

    ret = ffio_read_size(pb, title_str,
                                  title_size);
    if (ret < 0) {
      av_freep(&title_str);
      return ret;
    } […]
    av_freep(&title_str);
  }
  return 0;
}
```

```
libavformat/wavdec.c
static int w64_read_header(AVFormatContext *s)
{ […]
  AVIOContext *pb = s->pb; […]
  while (!avio_feof(pb)) { […]
    else if(!memcmp(guid,
                    ff_w64_guid_summarylist,16)){ […]
      uint32_t count, chunk_size, i; […]
      count = avio_rl32(pb);

      for (i = 0; i < count; i++) { […]
        if(avio_feof(pb)||(cur = avio_tell(pb))<0
                  || cur > end - 8 /* = tag + size */)
          break; […]
        avio_read(pb, chunk_key, 4);
        chunk_size = avio_rl32(pb);

        value = av_mallocz(chunk_size + 1);
        if (!value)
          return AVERROR(ENOMEM);

        ret = avio_get_str16le(pb, chunk_size,
                               value, chunk_size);
        avio_skip(pb, chunk_size - ret);

        av_dict_set(&s->metadata, chunk_key,
                    value, AV_DICT_DONT_STRDUP_VAL);
      } […]
    } else { […] }
  }
  avio_seek(pb, data_ofs, SEEK_SET); […]
  set_spdif(s, wav); […]
  return 0;
}
```

Fig.1 An original vulnerability (CVE-2017-5025) in FFmpeg (left) and its similar unknown one (right)

图 1 FFmpeg 中一个漏洞(CVE-2017-5025)的代码(左)及与之相似的未知漏洞的代码(右)

图 1(左)所示代码是 FFmpeg[14]的一个公开漏洞(CVE-2017-5025),能让攻击者利用特殊构造的视频文件引发堆崩溃.FFmpeg 是一个非常流行的开源音视频文件处理库,此类软件需要格外注意对外来文件或数据信息长度的检查.该漏洞的形成原因就是 FFmpeg 的函数 *mov_read_hdlr* 在解析 MP4 格式文件(由 Box 树形式组成)的 *hdlr* 子类型 Box 时,没有正确地对外部字符串大小 *title_size*(行 740)进行边界检查(行 741,只检查了下界而未检查上界),就直接加 1 后调用堆空间分配函数 *av_malloc*(行 744),而该分配函数对于参数大小为 0 的情况默认分配一个单位的堆空间,之后再执行写操作 *ffio_read_size* 时(行 748)就会导致堆溢出.该漏洞最直接相关的语句就是行 740,741,744 以及 748,其余接近 60 行代码都是与漏洞弱相关的,这些语句(如行 708-738 的读取操作)就构成了噪声.并且由于该函数某些操作和调用比较频繁,这种噪声在对整个函数进行特征抽取时一定会体现在最终的函数特征中.用这样的函数特征来计算相似性不会发现图 1(右)中的未知漏洞.图 1(右)是 FFmpeg 另一个文件中的函数 *w64_read_header*,它含有一个与该漏洞形成原理相同的未知漏洞.但其红色部分的漏洞相关语句与已知漏洞的并不完全相似,此外,该函数包含 100 多行代码,语句较多,在结构上与已知漏洞所在函数也极其不同,如果不对漏洞所在函数的噪声进行处理,这两个函数计算出的相似度值就非常低(低于 0.3).其他与该漏洞所在函数噪声特征比较相似的函数会被排序到前面从而导致误报,也增加了审计的工作量,而 *w64_read_header* 函数则会因为这些误报的出现而被排序在靠后的位置,因而该函数含有的未知漏洞无法被检测出,形成漏报.

本文针对目前漏洞相似性检测工作的这一局限,设计并实现了一种利用补丁进行未知漏洞发现的新方法.本方法与已有相似性检测方法主要有三点不同:首先,在已知漏洞的基础上综合考虑该漏洞的补丁信息,分析漏洞形成原因并准确定位漏洞相关语句,并且结合了程序切片技术,尽量去掉与漏洞无关的噪声语句,最后获得较为精确的漏洞特征,用于后续的相似度计算和检测.确保了最终匹配出的相似候选结果均是与漏洞特征相关的函数,减少了噪声带来的误报和漏报,提高了相似度检测的有效性.其次,由于切片能准确获取漏洞特征,可以直接用于相似度计算,因而天然地减少了对特征空间的主成分分析或者进一步提取主要编程模式的步骤,使得本方法具有较低的性能开销.最后,综合利用补丁的信息还有一大作用,就是可以对相似度计算后的排序结果进行二次筛选,去掉其中已经含有补丁语句而不会产生漏洞的误报.主要是通过加了补丁后的漏洞所在函数的特征与第一次计算出的候选结果中的函数进行第二次相似度计算,将相似度值不低于第一次计算结果的函数从候选中去掉,进一步降低了审计成本.最后,实验证明本方法在检测软件漏洞时确实可以准确高效地去除噪声影响及其相关误报和漏报,提高与漏洞相似的函数在最终结果中的排序,减少了审计的工作量,并且我们还在开源软件 FFmpeg 和 Ghostscript[15]中发现了一些新的未知漏洞.

本文的其余部分将按照下面的顺序进行组织:第 2 部分具体阐述本方法的设计思想与具体实现;第 3 部分描述整个实验的设计和实施,给出实验结果并进行分析;第 4 部分指出本方法的局限性和初步探讨未来的研究方向;第 5 部分详细介绍一些已有的研究工作并讨论与本工作的不同;最后,第 6 部分进行总结.

2 方法设计与实现

2.1 方法概述

本文设计并实现了一种利用补丁信息进行未知漏洞发现的方法,主要目标是解决现有工作中含有漏洞的函数的噪声语句影响及其可能导致的误报和漏报问题.事实上,已经有相关研究提出了一些利用已知漏洞进行漏洞发现的方法,但是在这些方法的实现中都忽视了已有漏洞的补丁所具有的潜在价值.补丁具有两种作用:第一种就是补丁明确了漏洞的位置和范围,根据补丁信息可以准确定位漏洞相关语句,再利用程序切片技术可以很容易地获得只与漏洞相关的特征语句,用这样的特征进行相似性检测可以提高检测的准确性,达到降低噪声引起的误报和漏报的目的.此外,既然切片保证了提取的特征只与漏洞相关,那么已经加过补丁或者本身含有补丁所需语句的函数也可能因为漏洞相关语句的相似性被检测出.此时,就需要利用到补丁的第二种作用,那就是利用含有补丁信息的函数特征再与检测出的候选函数进行一次相似度计算,将那些第二次计算结果比第一次的相似度更高的函数,也就是更倾向于已经含有补丁语句的函数从候选集中去掉.这样也减少了部分误报,从而进一步提高检测的准确性和降低人工审计的工作量.方法的整体框架和实施的流程如图 2 所示.

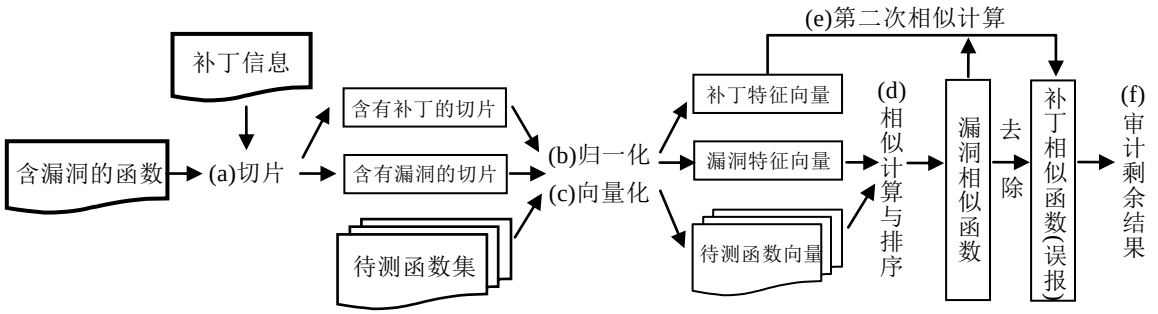

Fig.2 The overview framework of the approach using patches to discover unknown vulnerabilities

图 2 利用补丁的未知漏洞发现方法总体框架图

首先,对于已知漏洞函数,需要进行切片.切片的目的正是为了去除漏洞函数中的无关噪声,只保留漏洞相关特征.而要准确定位漏洞相关语句,需要补丁信息的指导.事实上,如果能够获得一个待测代码集的已知漏洞,其相应的补丁往往也能比较容易地获得.综合利用补丁语句的信息就能更好地定位漏洞形成的原因和漏洞所在的语句,再利用切片技术就能获得只与漏洞特征相关的切片.此外,切片过程除了需要对含有漏洞的函数进行切片,还需为第二次相似计算做准备,将补丁语句加入到含有漏洞的切片中形成含有补丁信息的切片.

其次,将切片和从待测代码集里的函数进行符号归一化和特征向量化.在获得了相应漏洞特征切片后,如何与待测函数进行相似度计算也是一个需要考虑的问题.为了简化特征的表示以及降低相似度计算的复杂性,我们采用了将切片和待测函数向量化以获得特征向量进行相似比较的方法.向量化是指将切片和所有待测函数都映射到一个向量空间,表示为其特征在向量空间中的对应向量的过程.本方法将函数中的一条语句作为一个特征,那么为了降低向量空间的维度,也使函数的特征具有一般性,需要在向量化之前进行一步符号的归一化处理.之后将每条特征语句采用哈希的方法映射到向量空间,所有函数的所有特征语句的哈希值就构成了整个向量空间.切片和每个待测函数的特征向量由其所包含的特征语句决定.

最后,进行相似度的计算和排序.计算与排序分成两次进行,第一次是利用漏洞特征向量与待测函数的向量分别进行相似度的计算,按照相似度的大小对结果排序,筛选出与漏洞特征相似的函数.而由于只包含漏洞的切片并不具有补丁的特征,其相似计算结果中有可能含有很多既有漏洞特征又已经加过补丁的函数,而这些函数是不具有漏洞的.为了避免这种原因造成的误报,我们进行了第二次相似计算.此时需要用补丁特征向量与初次计算结果中的函数的特征向量再次进行相似度计算和排序,将此次计算后相似度值仍然不低于初次计算结果的那些函数从候选结果中去掉,最终剩余的候选结果再进行人工审计就可以减少审计的工作量.

2.2 切片

切片对于本文提出的方法至关重要,是去掉漏洞函数中噪声语句影响并降低相关的误报和漏报的关键步骤.切片的对象是已知漏洞函数及其打过补丁后的补丁函数,生成的产物是含有漏洞的切片和含有补丁的切片.规则需要确定去掉哪些语句和保留哪些语句,在除了漏洞及其补丁之外没有更多先验知识的情况下,要精确且完整地界定代码中漏洞相关语句和噪声语句几乎是不可能的.因此,选择根据控制流和数据流来切片是一种可以接受的方案.

仍以图 1 提到的漏洞为例,原来含有漏洞的代码及其补丁如图 3 所示.由于行 741 未能准确地检查 *title_size* 的大小,只检查了是否小于 0,在行 742 和 743 加入了补丁语句,用于检查该变量是否超出限定的上界.根据补丁是对变量 *title_size* 的检查,可以得到切片的条件应该为数据依赖于该变量的语句,该例子中就是 *av_malloc* 的调用语句,从该句开始通过与之相关的控制依赖和数据依赖关系进行切片,我们就可以获得该函数含有漏洞语句的切片(红色部分).而把补丁语句也加入到含有漏洞的切片当中,就可以获得含有补丁的切片.

```
libavformat/mov.c
699 static int mov_read_hdlr(MOVContext *c,
---                        AVIOContext *pb, MOVAtom atom)
700 { […]
704   int64_t title_size; […]
708   avio_r8(pb); /* version */
709   avio_rb24(pb); /* flags */
710
711   /* component type */
712   ctype = avio_rl32(pb);
713   type = avio_rl32(pb); /*component subtype */
714   […]
736   avio_rb32(pb); /* component  manufacture */
737   avio_rb32(pb); /* component flags */
738   avio_rb32(pb); /* component flags mask */
739
740   title_size = atom.size - 24;
741   if (title_size > 0) {
742+    if (title_size > FFMIN(INT_MAX,SIZE_MAX-1));
743+      return AVERROR(ENOMEM);
744     title_str = av_malloc(title_size + 1);
745     if (!title_str)
746       return AVERROR(ENOMEM);
747
748     ret = ffio_read_size(pb, title_str, title_size);
749     if (ret < 0) {
750       av_freep(&title_str);
751       return ret;
752     }
753     title_str[title_size] = 0; […]
759   }
760   return 0;
761 }
```

Fig.3 The patch for the vulnerability of CVE-2017-5025 in FFmpeg

图 3 FFmpeg 对 CVE-2017-5025 漏洞的补丁

具体切片过程在代码编译后进行,切片算法参考了现有的研究工作[2].由于使用 GCC[16]作为编译工具,代码会在编译过程中生成与语言无关的 GIMPLE[16]树形式且未经过大量的优化,因此选择在 GIMPLE 语句层面上切片.并且,为了减少语句数目和向量空间维度,仅提取其中比较重要的三种类型语句,分别是条件 (Condition),赋值(Assignment)和函数调用(Call)语句,其中一条源代码语句可能对应多条 GIMPLE 语句操作.

切片时,考虑到如果仅保留漏洞所在语句进行计算匹配,切片条件过于严格,生成的切片特征向量维度太少,无法在最终匹配结果中将函数之间相似度的差别加以区分.因此,切片的条件要稍微放宽一点,即需要保留一些额外的语句,而这些语句必须与漏洞语句密切相关,或者是控制依赖关系或者是数据依赖关系.因此,切片时除了保留漏洞所在语句,对于函数调用类型的漏洞所在语句,保留直接数据依赖于调用结果的语句(对应于源码中的行 745 和行 749);对于赋值类型的漏洞所在语句,将额外保留其操作数的直接数据依赖语句以及一条直接数据依赖于被赋值变量的语句(本例中没有需要额外保留的语句).对于这些额外保留的的语句,其与漏洞模式的相关性有所减弱,为了减少其对相似度计算结果的影响,而只是起到适当放宽切片特征使相似度值有所区分的作用,需要给它们赋予一个较低的权重值,可以在[0.1,0.3]区间内确定,这是经过实验确定的值.

需要说明的是,待测函数集中的函数不需要进行切片,只需在编译后将每个函数中的条件,赋值和调用类型的 GIMPLE 语句抽取出来作为函数的特征即可.

2.3 符号归一化

使用 GIMPLE 语句作为函数的特征还需要进行符号归一化处理,这是为了降低向量化后向量空间的维度,也是为了进一步抽象语句特征使其具有一定程度的代表性.在此过程中,我们主要考虑三方面,变量名,变量类型,函数调用名.

首先,变量名不能直接出现在特征语句中.变量名一般具有很大的差异性,即使相似也会影响到向量化后的结果,所以变量名需要用其变量类型代替,常量表示为 *interger_cast*,函数调用参数列表直接用函数定义的参数类型表示,而对于结构体成员变量,枚举类型的变量以及宏变量等,各函数在使用时本来就是统一的,所以被保留下来.比如图 3 中行 741 的变量 *title_size* 就会用其类型 *int64_t* 表示,而 *atom.size* 则会用 *MOVAtom.size* 表示,行 741 就转化为 *int64_t = MOVAtom.size - integer_cast*.而行 745 转化为 *char* = av_malloc(size_t size)*.

其次,变量类型的归一.由于跨平台编程等原因,很多数据类型经过 typedef 的定义,如 size_t 就是从 unsigned

int 定义来的,如果不做处理就可能因为类型名不同使得向量化后不一致,从而漏掉某些相似的函数.根据变量类型表示的数据范围不同,将常见的一些数据类型加以总结并分类统一表示,具体分类见表 1.

Table 1 The clusters of common data types

表 1 常见数据类型分类说明表

数据类型	变量类型分类
bool	bool, _Bool
char	signed char, s8, __s8, int8_t
unsigned char	unsigned char, u_char, unchar, u8, __u8, uint8_t, __uint8_t, u_int8_t, Byte, Byte_t, byte_t
Short	short int, s16, __s16, int16_t
unsigned short	short unsigned int, umode_t, u_short, ushort, u16, __u16, uint16_t, __uint16_t, u_int16_t
int	int, s32, __s32, int32_t, int32
unsigned int	unsigned int, u_int, uint, u32, __u32, uint32_t, __uint32_t, u_int32_t, uint32
long	long int
unsigned long	long unsigned int, uintptr_t, u_long, ulong
long long	long long int, s64, __s64, int64_t
unsigned long long	Long long unsigned int, u64, __u64, uint64_t, __uint64_t, u_int64_t, u_quad_t
size_t	32bit: unsigned int, 64bit: unsigned long
ssize_t	32bit: int, 64bit: long

另外,函数调用名称也需要进行归一化.事实上,有很多函数实现的功能相似,只是根据上下文环境不同调用者选取了不同的函数进行调用.比如图 1 所举的实例中,已知漏洞函数调用的堆空间分配函数是 *av_malloc*,而未知漏洞中调用的是 *av_mallocz*.这两个函数同样用于在堆中分配一定大小的空间,后者是前者的一个包装,只在前者的基础上多加了一步初始化分配空间的操作.如果不进行函数名的归一,那么这两句调用对应到向量空间中就不一致,在进行相似匹配时也可能会将该未知漏洞漏掉.为了将这种情况也考虑在内,根据经验,开发者在开发类似功能代码时倾向于取相近的函数名,所以我们提取出待测代码集中所有被调用到的函数,利用字符串距离计算对它们进行了一个简单的聚类,将比较相似的字符串归并为同一个表示形式,比如 *av_malloc* 和 *av_mallocz* 都用 *av_malloc* 表示,达到函数名归一的目的.

2.4 特征向量映射

向量化是将函数和切片中提取的特征语句映射到向量空间的过程,是为了简化特征表示以及相似度的计算而采取的方法,相对于利用树和图等结构进行匹配和计算的方法减少了很多计算量.

待测代码集合 $C=\{F_1,...,F_n\}$表示其含有 n 个函数,V 代表 C 所对应的特征向量空间.我们可以将 C 到 V 的映射定义为$\emptyset$. $\emptyset$采用现有哈希算法 *hashpjw*[17]实现,每一条特征语句都会对应到 V 中的一个哈希值,则向量空间 V 的维度 $|V|$ 就是 C 中所有函数所包含的所有语句的总数.而对于每个函数 $F_i(1 \le i \le n)$,其向量的维度均为 $|V|$,每一个哈希值 h 表示的维度上的数值根据下面公式计算:

$$\emptyset_{Fi}(h) = I(s,h) \cdot \mathrm{TF-IDF}(s) \quad \text{其中} I(s,h) = \begin{cases} 1 & \text{如果哈希值 } h \text{ 对应的语句} s \in Fi \\ 0 & \text{如果哈希值 } h \text{ 对应的语句} s \notin Fi \end{cases}$$

其中,TF-IDF[18]是信息检索中常用的加权技术,其值等于词频 TF(Term Frequency)和逆向文件频率 IDF(Inverse Document Frequency)的乘积.引入该权重主要是考虑到语句的频繁程度和重要程度不同,比如赋值特征语句 *int = int + int* 会在代码集中大量存在,需要降低该句对计算结果的影响;而某些语句在某些函数中频繁出现,在其他函数中不太出现,所以为了评估每个哈希值对不同函数的重要程度,在向量化表示之后又添加了 TF-IDF 权重的计算.但是在本方法中我们将 TF 和 IDF 的计算从文档层面迁移到函数层面,而词的概念则对应于本方法中的哈希值,具体计算方法参考 Salton 和 McGill 的文献[18].

2.5 相似度计算与匹配

生成了特征向量之后就是利用相似度计算检测未知漏洞的过程.在本文提出的方法中,相似度计算与排序

分为两次,第一次是用漏洞特征向量与待测函数的特征向量分别进行相似度的计算,将结果按照相似度排序形成初步候选结果;第二次是对初步候选结果中的函数,利用补丁特征向量再与之进行一次相似度的计算,如果一个候选函数属于无漏洞的误报,那么该函数应该是含有补丁特征的函数,理论上其第二次匹配相似度应该高于或者至少是不低于初次计算时的相似度值.进行二次计算筛选的目的正是又一次利用补丁信息,除去初步候选集中的不含有漏洞的误报,进一步减轻审计的工作量.

而两个特征向量 $A(a_1,...,a_n)$和 $B(b_1,...,b_n)$的距离采用余弦相似度计算,公式如下.距离值应该属于[0,1]区间内,数值越大表明两个向量距离越近,其代表的相应切片和函数也就越相似.数值为 0 时表示两个向量完全不同,没有任何一维特征重合;数值为 1 时表示两个向量完全相同,在所有特征维度上都重合.

$$Simi(A,B)=\frac{A\times B}{|A|\cdot|B|}=\frac{a1\times b1+\ldots+an\times bn}{\sqrt{a_1^2+\ldots+a_n^2}\times\sqrt{b_1^2+\ldots+b_n^2}}$$

3 检测实验与分析

3.1 实验设置

我们将本方法部署在 64 位的 Linux 16.04 平台上,并利用 GCC 4.9 作为编译器.选取了多平台广泛支持的开源音视频处理库 FFmpeg 3.2.4 版本和开源图像浏览软件 Ghostscript 9.21 版本作为实验对象.其中,FFmpeg 包含 1583 个文件,15598 个函数,Ghostscript 含有 935 个文件,15875 个函数.漏洞方面则选取这两个软件在 2017 年最新公布的漏洞作为已知的目标漏洞,来进行检测实验.

3.2 性能分析

本方法涉及到三个主要的步骤,切片,归一化和向量化我们都直接部署在 GCC 中,在编译过程中同时实现切片和向量映射输出特征向量.相似度计算是在此之后单独进行的,而且相似度计算几乎都能够在 1 秒内完成,几乎不消耗时间,所以性能实验主要针对切片和向量映射所消耗的时间进行分析.我们首先对实验对象 FFmpeg 和 Ghostscript 进行单独编译记录下时间消耗,然后添加了本方法中切片,归一以及向量映射的处理过程后再次记录下运行时间,重复该过程 10 次取时间损耗的平均值,结果见表 2.

Table 2 Time consumed by the slicing process and signature extracting process

表 2 本方法切片和获得特征向量过程的时间消耗

实验对象	单独编译时间(秒)	加入切片和特征向量映射的时间(秒)	切片和向量映射的性能开销
FFmpeg	111.47	210.96	89%
Ghostscript	165.59	270.69	63%

可见,实际切片一直到特征向量映射步骤结束所需时间小于一次编译的时间,对于含 10000 个以上函数的代码分析仅需几分钟,性能开销完全在可以接受的范围内.

3.3 检测结果

本次对 FFmpeg 和 Ghostscript 进行未知漏洞发现实验的过程中共检测出了 3 个新的未知漏洞,其中对图 1 中的 FFmpeg 的 CVE-2017-5025 漏洞检测出 1 个未知漏洞[19],对 Ghostscript 的 CVE-2017-5951 漏洞检测出了 2 个相似的未知漏洞[20,21],我们已经提交给相应的开发团队并且已经得到确认.下面将用 FFmpeg 的漏洞 CVE-2017-5025 为例,说明结合了补丁信息进行切片的方法,在降低含有漏洞的函数中的无关语句噪声及其造成的相关误报和漏报方面的有效性,以及二次计算利用补丁特征向量过滤无漏洞函数的误报的有效性.

首先,如果不进行切片,单纯使用整个函数 *mov_read_hdlr* 的所有语句作为特征进行向量化后直接进行相似度计算的结果如表 3 所示(排序结果不包含目标漏洞所在函数).由于函数中漏洞无关的语句比例较高(56 行/61 行),造成最终的相似度计算结果受到了噪声的很大干扰,所有候选函数的相似度都偏低.同时从表 3 中可以发现,根本不含有漏洞相关语句的误报比例非常高,一共有 9 个（表 3 中×号所示）,比例占前 15 个的一半以上,

检测的准确性很低,误报较高.

Table 3 Top 15 most similar functions to CVE-2017-5025 without slicing process

表 3 不切片时前 15 个最相似于 CVE-2017-5025 的函数

排序	相似度	函数名	噪声误报(×)	排序	相似度	函数名	噪声误报(×)
1	0.67	read_header	×	9	0.53	rsd_read_header	×
2	0.63	get_aiff_header	×	10	0.53	mpc_read_header	
3	0.63	ape_read_header		11	0.49	mov_read_ftyp	
4	0.62	mov_read_frma	×	12	0.47	read_packet	×
5	0.60	mov_read_dref		13	0.47	read_header	×
6	0.59	qcp_read_packet	×	14	0.47	read_header	×
7	0.58	smacker_read_header		15	0.47	hls_write_header	
8	0.54	read_ints	×				

Table 4 Top 15 most similar functions to CVE-2017-5025 after slicing (only computing similarity once)

表 4 加入切片方法后的前 15 个最相似于 CVE-2017-5025 的函数(只进行一次相似计算)

排序	相似度	函数名	含补丁的误报(×)	排序	相似度	函数名	含补丁的误报(×)
1	0.89	mov_read_udta_string	×	9	0.72	avi_read_tag	×
2	0.80	**w64_read_header**		10	0.70	mov_read_wave	×
3	0.78	mpc8_parse_seektable	×	11	0.70	asf_read_stream_properties	×
4	0.78	mov_read_senc		12	0.68	**mov_read_keys(patch)**	×
5	0.76	wav_parse_bext_tag	×	13	0.68	mov_read_custom	×
6	0.75	mov_read_dref		14	0.67	ape_read_header	
7	0.75	ff_read_riff_info	×	15	0.65	vmd_read_header	×
8	0.73	read_header					

Table 5 Top 15 most similar functions to CVE-2017-5025 after the second time of sililarity computing

表 5 进行二次相似度计算后的前 15 个最相似于 CVE-2017-5025 的函数

排序	相似度	函数名	含补丁的误报(×)	排序	相似度	函数名	含补丁的误报(×)
1	**0.92**	**mov_read_udta_string**	×	9	0.68	w64_read_header	
2	**0.85**	**mpc8_parse_seektable**	×	10	0.66	mov_read_senc	
3	**0.81**	**wav_parse_bext_tag**	×	11	0.64	ff_read_riff_info	
4	**0.79**	**mov_read_wave**	×	12	0.63	mov_read_dref	
5	0.76	hls_append_segment		13	0.62	read_header	
6	0.74	open_input_file		14	0.61	avi_read_tag	
7	**0.73**	**asf_read_stream_properties**	×	15	0.61	avi_read_header	
8	0.72	ogg_read_page					

而利用补丁信息和程序切片技术获取只含有漏洞特征向量后,进行一次相似度计算后排序的结果见表 4. 实验结果表明引入切片技术后再次计算的相似度结果中没有一个是由于漏洞无关噪声导致的误报,并且相似度普遍提高了.此时再看表 3 中的几个噪声误报的相似度值,原来排序 12 的 *read_packet* 相似度 0.47,现在相似度 0.34,排序掉到 287 名,其他 8 个因为噪声误报的函数切片后的相似度值均低于 0.2,尤其原来排名为 2 和 4 的 *get_aiff_header* 和 *mov_read_frma* 两个函数相似度更是不到 0.1.以上结果充分证明了通过切片确实能有效去掉

噪声影响,减少因为噪声产生的无关误报.此外,我们还发现了表 4 候选结果中的第 12 个是一个已知漏洞,并且在我们发现之前刚打了补丁,而排序的第 2 个函数是一个未知漏洞,这两个函数在不切片的相似计算中并没有发现,说明切片降低了无关的噪声后,能更准确地匹配出相似的函数,提高与已知漏洞相似函数的排序,减少漏报.但是在不经过第二次相似计算之前,此时的结果中有 10 个函数(表 4 中×号所示)都是已经含有补丁语句的函数,也是不会导致漏洞的误报.

最后,在表 4 的基础上,利用补丁产生的含补丁特征的向量与其中的函数进行二次相似度计算,结果见表 5.从表 5 的实验结果可以看出有 5 个函数(表 5 中×号所示)的相似度值是高于了表 4 所示第一次相似度计算结果的,这说明这 5 个函数是已经包含有所添加的补丁特征的,那么它们就不会再含有该目标漏洞.可以从候选结果中去掉,不再审计.而前面表 4 的实验结果中另外 5 个含有补丁语句的误报没能被第二次相似度计算筛选出来的原因是它们使用的补丁的方式不同,因而本次实验使用的含补丁特征的向量未能提高这几个函数的相似度.

4 讨论与展望

本文提出的利用补丁的未知漏洞发现方法在检测未知漏洞时,也无可避免会有误报.但是实验已经证明,本方法确实能够有效减少因漏洞函数中的噪声引起的有关误报和漏报,同时在相似度结果中利用二次相似度计算也能减少含有补丁语句函数的误报,已经提高了相似性检测的有效性,减少了很多人工审计的工作.

需要指出的是本文提出的方法是一个比较一般性的方法,对漏洞的类型并不太敏感.当然,如果漏洞所在函数含有较多更为鲜明的漏洞特征,理论上来说本方法的检测精度也会更高.

其次,程序补丁的作用可能不只是修补程序漏洞,有些还可能是为了调整程序的功能.但是自动识别补丁中修补漏洞的代码和调整功能的代码是较为困难的,因此这个问题将放到未来工作中考虑.未来我们可能从更加准确地界定切片条件的角度出发,来考虑利用相似性从已知的敏感操作来检测未知的敏感操作,这样可能更有利于实现对补丁代码中漏洞相关语句的自动识别.此外,补丁也不只有增改代码的情况,有些是只减去一些错误的或过时的代码.但实验中发现这种情况相对较为少见,暂时不在本文考虑的范畴,本文更倾向于那些增改了代码的补丁,未来可能考虑这种情况的处理.

另外,本方法目前仅包含过程内的分析,相似度的计算以函数为对象.因为从经验的角度分析,一个函数是程序代码中的一个比较独立的功能单位,并且很多漏洞形成的原因及漏洞影响范围都局限在一个函数内部.另外,也是出于对切片和相似度计算性能的考虑,本方法仅实现了函数级别的相似度检测.所以对于需要进行过程间分析的相关漏洞,暂时不在本方法的适用范围之内.未来工作可能考虑引入过程间分析.

最后,其实有关克隆漏洞(Clone Bugs)检测的研究工作在某些思想上与本文的相似性检测方法具有一定程度的相关性,未来可能考虑研究克隆检测的相关方法是否能够被利用到本工作中.

5 相关工作

代码分析和软件漏洞检测工作一直都是软件安全领域的重要组成部分.尤其是静态分析在检测软件漏洞方面,已经有很多的方法和技术.有些工作是依赖于先验知识的,需要提前给出规则相关的一些知识,再静态分析代码生成规则,用以进行检测.如 Tan 等人提出的 iComment[3],是事先确定好要发现的规则类型并给出了模板,然后利用自然预言处理和机器学习等方法从源码中的注释提取规则填充模板内容,利用模板规则检查代码是否遵循注释的要求.另外 Tan 等人还提出了 AutoISES[4],关注的是 Linux 中的安全检查函数,也是静态分析代码抽取这些函数频繁保护的数据以生成规则,检测违例.另外有一些工作聚焦于研究如何自动提取代码规则[2,5-10].典型代表有 PR-Miner[2]以及 AntMiner[9],都是利用数据挖掘的思想,将代码潜在规则的抽取转化为频繁项集的挖掘.APISan[10]也是自动提取编程规则检测违例的工具,它无需人工模板和 API 使用文档,它使用一种放宽条件的符号执行方法提取 API 模式,检查 API 使用过程中返回值,参数条件,API 相关性以及 API 使用前后的检查条件等是否有违背.这些自动提取代码规则并用于检测的方法的有效性依赖于提取规则的正确性,但是自动提取很可能会出现错误,此外此类工作也非常依赖于人工审计,审计的成本较大.

近年来,已有一些研究者提出利用漏洞相似性检测未知漏洞的方法.如 Yamaguchi 等人则从异常外推[11]的角度给静态分析检测漏洞提供了新的思路,在提取特征时由仅仅提取 API 符号作为特征变为利用函数的抽象语法树映射到特征向量.而 discovRE[12]方法应用于在二进制代码上查找与已知漏洞函数相似的函数,首先利用量化特征如指令条数和基本块个数等组成的特征向量计算出一系列相近的候选函数,再从它们的结构特征也就是 CFG 图的相似度上进行比较筛选.但是该方法仍需要进行图匹配的计算,有一定的性能开销.同样是在二进制代码上进行漏洞检测,Feng[13]等人设计实现的漏洞搜索引擎 Genius 为了进行跨平台的漏洞检测,将控制流图抽象成平台无关的具有普遍适用性的特征,以实施一种可扩展的漏洞检测.此类方法大多采用漏洞所在函数整体作为特征提取的主体,会导致含有漏洞的函数中的无关语句形成相似计算时的噪声,从而产生相应的误报和漏报,这正是本方法被提出的原因.

另外,克隆检测的思想在某种程度上与本类方法具有相似性.如 Yamaguchi 等人[22]提出过利用源代码生成的代码属性图,用图查询方式进行克隆代码检测的方法,相对我们的方法来说利用图进行检测的开销较大.CP-Miner[23]则是基于符号特征的,生成符号序列检查其他代码中的复制序列,但需要挖掘频繁序列,可能出现很多误报.DECKARD[24]利用抽象语法树作为特征生成向量,计算向量相似度来进行克隆代码检测.这与 Yamaguchi 等人的异常外推方法在思路上很相似,只是相对于检测克隆代码,相似性漏洞检测需要研究如何得到抽象程度更高,更能应对代码变化的特征.Kim 等人 2017 年提出的 VUDDY[25]则根据已有漏洞所在函数提取特征指纹,检测是否有函数因为代码克隆而与该段代码含有相同漏洞.但这种方法中所有变量和函数全部用同一种类型和函数名称替换,粒度较粗.且每个函数体计算一个哈希值,只能检测精确克隆造成的相似漏洞,不能检测出代码结构有所改变的克隆函数,而本文提出的方法则考虑了这一点.

最后,值得一提的是有一些工作已经意识到补丁对于漏洞发现和利用的意义,其中比较有代表性的是 Brumley 等人提出的 APEG[26]方法.该方法从攻击者的角度考虑,如果提前得到漏洞补丁就可以在未及时打补丁的时间空隙进行漏洞利用.该方法主要关注未合理验证输入的漏洞类型,且注意到了补丁的意义所在并加以利用:一是通过对比软件含漏洞的版本和加了补丁的版本可以定位新加入的输入检查语句,二是通过求解到达新检查语句的路径条件获得使检查失败的非法输入以攻破已有漏洞.本文方法同样利用了补丁的价值,但是使用的目的和方法都与 APEG 有天壤之别.APEG 从攻击的角度着眼于漏洞本身的利用,使用补丁是为了获得非法输入.本文方法则是从已知漏洞出发去检测发现相似的未知漏洞,并且利用补丁帮助去掉可疑候选结果中的部分误报.另外本方法并未特地区分漏洞的类型.

6 总结

本文分析了当前一种由含有漏洞的代码段出发静态检测未知漏洞的相似性检测思路及其局限性,并提出了一种利用补丁的未知漏洞发现方法.该方法结合了漏洞的补丁信息以及程序切片的技术,准确去除已知漏洞函数中无关语句带来的噪声,减少了因这部分噪声带来的部分误报和漏报;并用两次相似度计算的方法分别用含有漏洞函数和打过补丁的函数的特征向量与待测函数的特征向量分别进行相似计算,将计算结果中已经含有补丁语句特征的函数过滤掉,减少了这部分误报,更加便于后续审计.通过实验,可以证明本方法对于前期噪声的减少是有效的,对 FFmpeg 和 Ghostscript 两个开源代码库的实验还检测出了 3 个新的未知漏洞,说明了本方法在未知漏洞发现方面是有效的.

References:

[1] Hopcroft J, Motwani J, Ullmann R. Introduction to automata theory, languages, and computation. Addison-Wesley, 2 edition, 2001.

[2] Li ZM, Zhou YY. PR-Miner: automatically extracting implicit programming rules and detecting violations in large software code. Proc. of the European Software Engineering Conference held jointly with ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2005. 306-315.

[3] Tan L, Yuan D, Krishna G, Zhou YY. iComment: bugs or bad comments. Proc. of ACM Symposium on Operating Systems Principles, 2007. 145-158.

[4] Tan L, Zhang XL, Ma X, Song WW, Zhou YY. AutoISES: automatically inferring security specification and detecting violations. Proc. of USENIX Security Symposium, 2008. 379-394.

[5] Tan L, Zhou YY, Padioleau Y. aComment: mining annotations from comments and code to detect interrupt related concurrency bugs. Proc. of the International Conference on Software Engineering, 2011. 11-20.

[6] Pradel M, Jaspan C, Aldrich J, Gross TR. Statically checking API protocol conformance with mined multi-object specifications. Proc. of the International Conference on Software Engineering, 2012. 521-530.

[7] Yamaguchi F, Wressnegger C, Gascon H, Rieck K. Chucky: exposing missing checks in source code for vulnerability discovery. Proc. of ACM SIGSAC conference on Computer & communications security, 2013. 499-510.

[8] Yamaguchi F, Golde N, Arp D, Rieck K. Modeling and discovering vulnerabilities with code property graphs. Proc. of IEEE Symposium on Security and Privacy, 2014. 590-604.

[9] Liang B, Bian P, Zhang Y, Shi WC, You W, Cai Y. AntMiner: mining more bugs by reducing noise interference. Proc. of International Conference on Software Engineering, 2016. 333-344.

[10] Yun I, Min C, Si X, Jang Y, Kim T, Naik M. APISan: sanitizing API usages through semantic cross-checking. USENIX Security Symposium, 2016. 363-378.

[11] Yamaguchi F, Lottmann M, Rieck K. Generalized vulnerability extrapolation using abstract syntax trees. Annual Computer Security Applications Conference, 2012. 359-368.

[12] Eschweiler S, Yakdan K, Padilla EG. discovRE: efficient cross-architecture identification of bugs in binary code. Annual Network and Distributed System Security Symposium. 2016.

[13] Feng Q, Zhou RD, Xu CC, Cheng Y, Testa B, Yin H. Scalable Graph-based Bug Search for Firmware Images. Proc. of ACM SIGSAC Conference on Computer and Communications Security, 2016. 480-491.

[14] FFmpeg. http://ffmpeg.org/.

[15] Ghostscript. https://www.ghostscript.com/.

[16] GNU Compiler Collections (GCC) Internals. https://GCC.gnu.org/onlinedocs/GCCint.

[17] Aho AV, Sethi R, Ullman JD. Compilers: principles, techniques, and tools, 1986.

[18] Salton G, McGill MJ. Introduction to Modern Information Retrieval. McGraw-Hill, 1986.

[19] The commit of FFmpeg. https://git.ffmpeg.org/gitweb/ffmpeg.git/commitdiff/a4fb44723dcaa56416173bc3d0ff41e9cda25067?hp=25a592e5d45672bdfdac35bf0119907cdcddd1b7.

[20] Bug 698066 of Ghostscript. https://bugs.ghostscript.com/show_bug.cgi?id=698066.

[21] Bug 698073 of Ghostscript. https://bugs.ghostscript.com/show_bug.cgi?id=698073.

[22] Yamaguchi F, Maier A, Gascon H, Rieck K. Automatic inference of search patterns for taint-style vulnerabilities. IEEE Symposium on Security and Privacy, 2015. 797-812.

[23] Li ZM, Lu S, Myagmar S, Zhou YY. CP-Miner: finding copy-paste and related bugs in large-scale software code. IEEE Trans. Software Engineering, 2006,32(3):176-192.

[24] David Y, Yahav E. Tracelet-based code search in executables. Proc. of ACM SIGPLAN Conference on Programming Language Design and Implementation, 2014.

[25] Kim S, Woo S, Lee H, Oh H. VUDDY: a scalable approach for vulnerable code clone discover. IEEE Symposium on Security and Privacy, 2017. 595-614.

[26] Brumley D, Poosankam P, Song D, Zheng J. Automatic patch-based exploit generation is possible: techniques and implications. IEEE Symposium on Security and Privacy, 2008. 143-157.