计算机系统应用 ISSN 1003-3254, CODEN CSAOBN
Computer Systems & Applications,2019,28(5):10−17 [doi: 10.15888/j.cnki.csa.006913]

E-mail: csa@iscas.ac.cn
http://www.c-s-a.org.cn
Tel: +86-10-62661041

基于文件格式信息的改进模糊测试方法①

刘天鹏, 程　亮, 张　阳, 佟思明

(中国科学院 软件研究所 可信计算与信息保障实验室, 北京 100190)
通讯作者: 刘天鹏, E-mail: liutianpeng@tca.iscas.ac.cn

摘　要: 本文针对盲目变异的模糊测试策略带来的效率低下的问题, 综合程序控制流图、输入种子样本特征、异常样本发现、模糊测试器路径反馈等信息, 提出一种更为有效的种子输入变异策略. 本文通过不断监控种子文件在目标程序中的执行路径, 并引入污点分析的方法, 以建立起新增执行路径的起始语句与种子文件中关键字节的一对多映射关系. 最终本文将根据这种映射关系对种子文件中的能够增加路径覆盖的字节进行变异, 以期得到更有效率的模糊测试结果. 我们的实验表明, 增加定向变异之后的模糊测试器在代码覆盖率, 以及模糊测试的效率上都有较为显著的提升.

关键词: 模糊测试; 污点追踪; 自动化漏洞检测

引用格式: 刘天鹏,程亮,张阳,佟思明.基于文件格式信息的改进模糊测试方法.计算机系统应用,2019,28(5):10–17. http://www.c-s-a.org.cn/1003-3254/6913.html

File-Type-Based Method to Improve Fuzz Testing

LIU Tian-Peng, CHENG Liang, ZHANG Yang, TONG Si-Ming

(Trusted Computing and Information Assurance Laboratory, Institute of Software, Chinese Academy of Sciences, Beijing 100190, China)

Abstract: To solve the problem of low efficiency caused by random mutation, a more effective mutation strategy is proposed in this study. The proposed approach synthesizes different kinds of information to help the Fuzzer mutate seed file, i.e., the CFG of program, the characteristics of input seed file, the information of abnormal input detection, and the branch courage of the Fuzzer. Based on this strategy, we design a new Fuzzer which continuously monitors the execution path of each seed file used as input of target program. Meanwhile, as most path constraints depend on only a few bytes in the input, periodical byte-level taint tracking will be necessary in the whole fuzzing process. After all this, we can infer a one-to-many mapping relation between the new execution path and key bytes in seed files, which can highlight the target start-end tuples of the seed file with more possibility to explore new branches in the target program to mutate. The result shows our design can improve the branch coverage of target program and the efficient of Fuzzing.

Key words: fuzzing; taint tracking; automated vulnerability detection

1　引言

软件漏洞挖掘一直是网络空间安全领域极为重要的分支之一, 发掘出来的软件漏洞经常被用来攻击用户电脑、服务器等, 从而获取用户隐私, 甚至危害国防安全. 目前常用的漏洞挖掘技术有静态分析 (如 IDAPro[1])、动态分析 (如 OLLYdbg[2]、Windbg[3])、模糊测试等, 然而除模糊测试之外的其他几种漏洞挖掘技术的效率都会随着软件规模的不断增长而不断降低.

① 基金项目: 国家自然科学基金 (61471344)
Foundation item: National Natural Science Foundation of China (61471344)
收稿时间: 2018-12-05; 修改时间: 2018-12-25; 采用时间: 2018-12-28; csa 在线出版时间: 2019-05-01

模糊测试自从被提出[4]以来, 一直作为自动化漏洞挖掘的利器被各大厂商 (如谷歌[5], 微软等) 广泛应用于软件质量控制环节.

模糊测试通常可以分为白盒模糊测试、黑盒模糊测试以及灰盒模糊测试三种, 其中灰盒模糊测试能够通过对程序轻量级分析获取程序执行信息, 并以此对执行结果进行评估, 从而制定测试策略. 这种方法相对于白盒模糊测试可以保证模糊测试的速度, 是目前运用最广泛的模糊测试方法. 例如 American Fuzzy Loop (AFL)[6]作为最广为使用的灰盒测试工具, 已经在上百种软件中发现漏洞, 近年来大量关于模糊测试的研究论文也选择该工具作为测试基准. 本文亦选择 AFL 作为实验基础.

进行灰盒模糊测试的分析人员往往需要根据围绕输入种子特征, 对输入样本进行变异、筛选, 并在此过程中及时监控程序的执行情况, 以发现并处理那些可以产生新的路径覆盖甚至是造成程序崩溃的异常样本, 从而尽可能保证模糊测试器不会错过任何一个可能触发漏洞的执行样本. 然而在这个过程中, 会不可避免的出现一些问题.

首先, 对于不同的目标软件来说, 用于测试的种子文件特征肯定是不相同的, 如果根据输入种子格式对模糊测试器进行定制, 那么就会带来难以拓展的问题; 而如果设计一种较为通用的模糊测试器, 就有可能忽略掉种子文件的格式对模糊测试过程所带来的信息增益[7], 造成模糊测试的效率低下. Peng H[8]等提出去掉目标程序中的完整性检测, 当出现输入完整性验证失败时, 将原程序中的完整性验证屏蔽之后继续执行. 但是由此带来很高的误报, 并且很大程度上无法复原相关漏洞在原始程序中的触发. 其次, 分析人员需要对使用的模糊测试器定制一个很好的变异策略, 以保证在反复执行目标程序的过程中得到尽可能多的路径覆盖[9]. 但是复杂的变异策略如基于符号执行的变异策略[10]可以产生高质量的变异, 但是执行效率很低, 且容易在实际求解过程中产生路径爆炸的问题; 而基于随机变异的策略虽然实现简单、执行速度优异, 但是很难产生高质量的变异[11]. Angora[12]是一种基于随机变异的模糊测试器, 设计者采用梯度下降优化算法解决分支的路径依赖问题, 但是这种方法需要对每一个新输入都进行污点分析, 这将极大的拖慢模糊测试的速率. 最后, 由于变异的过程将产生海量的中间样本[13,14], 程序的运行信息 (如路径覆盖、程序崩溃信息等) 有可能保存在这些样本之中[15], 如果将这些样本全部保存下来, 那么将会很大程度上降低程序执行的效率, 因此需要筛选并保留那些包含程序运行信息的样本文件[16].

针对上述问题, 我们设计了一种可以解析种子文件格式信息, 并且不通过符号执行策略来产生高质量变异的模糊测试器. 我们的设计有以下几个特点.

(1) 解析程序的控制流信息. 在每次模糊测试之前, 根据程序的执行信息获得程序中控制流图中的所有节点所对应的位置. 本文着重于对基于灰盒测试[17]进行研究, 因此需要在模糊测试之前对目标程序进行插桩编译, 以期在模糊测试的过程中全面掌握程序的运行信息, 并将目标程序的所有执行路径记录下来, 从而判读出新增的路径是否存在于控制流图中的边的集合, 由此可以得到每条新增路径所对应的内存起止位置. 此外, 还将利用控制流信息辅助对程序异常状态进行处理.

(2) 高效的利用文件格式所带来的信息增益. 我们通过污点分析的方法判断出输入的种子文件中能够影响控制流图节点信息的具体字节. 然而污点分析是一个十分必要但是执行速度又过于缓慢的进程, 因从我们利用格式解析器将污点分析得到的绝对位置转化成了相对于文件格式关键字的相对偏移量, 这样对于新输入, 只需要定位到关键字, 从上述信息中可以计算出污点的绝对位置, 从而减少了污点分析的执行次数.

(3) 定向变异与随机变异相结合的变异策略. 我们通过对污点分析得到了一些有可能对程序执行路径发生较大影响的重点字节, 我们将这些字节分组并对其按组进行编码, 最终在不同的执行过程中有选择地对一些字节采用定向变异的方式, 其余字节采用随机变异的策略.

我们在主流模糊测试工具 AFL 的设计思路上进行改进, 最终实现了模糊测试器 TaintFuzzy, 并得到了更为高效的模糊测试结果, 实验表明我们的模糊测试器与 AFL 有着相近的执行速度, 但是在单位时间内探索路径的数量增加了 12.26%, 并且产生了更深层次的路径覆盖.

2 背景

本节首先详细介绍了模糊测试器的设计流程, 然

后介绍了模糊测试工具 American Fuzzy Loop[6]的设计思路与变异手段.

2.1 模糊测试器工作流程

模糊测试器的设计方法有很多, 但是无论采用何种模糊测试策略, 其测试流程一般都包括识别测试目标及输入, 生成模糊测试数据, 执行模糊测试数据, 监视异常这几个阶段[18]. 大多数可利用的软件漏洞原因在于未对输入数据进行周密的合法性判断及处理, 因此模糊测试首先要确定目标的预期输入, 否则模糊测试的结果将有很大的局限性. 然后需要按照一定的策略根据预期输入不断生成测试用例, 常用的策略通常可分为 3 类: 基于变异的方法, 基于生成的方法和两者相结合的方法[18]. 而执行测试用例的过程一般需要与生成测试用例的过程交叉进行, 以验证生成的测试用例是否有效的增大目标程序的路径覆盖或是触发程序漏洞. 在这个过程中, 还需要异常监视程序来记录模糊测试过程中所有异常信息, 这部分往往通过插桩实现, 对于一些可以开启调试模式的目标程序, 也可以通过对应的调试接口实现.

2.2 AFL 设计策略

作为一款基于覆盖率的模糊测试器, AFL 能在在执行测试用例的同时通过遗传算法自动生成那些有可能在目标程序中产生更多路径覆盖的测试用例.

AFL 识别测试目标程序, 并对执行测试用例的过程进行异常监视的方法是对其进行插桩处理. 设计者使用 afl-gcc 作为编译工具实现了对目标程序的插桩, 并使用 afl-as 作为汇编器, 插入桩代码并执行会变操作. 桩代码中主要是一段汇编代码, 其中主要的操作包括保存 edi 等寄存器信息、生成一个[0, Map_size]范围内的随机数、保存生成的随机数、调用方法 afl_maybe_log、恢复寄存器信息. 其中值得一提的是, 在处理到某个分支, 需要对其插入桩代码时, AFL 会生成一个随机数, 用于标识这个代码块, 也就是说, 目标程序的每一个可执行分支都会在编译过程中赋予一个特殊的值作为这条分支的标识 l_{cur}.

为了高效执行测试用例, AFL 实现了一套 fork server 机制, 在模糊测试器启动的同时, 运行了一个 server 进程作为 fork 子进程的服务端, 模糊测试器本身并不 fork 子进程, 而是通过 server 实现与目标程序的通信. 通过这种设计, 将目标程序的子进程与模糊测试器的执行分离开来, 从而保证模糊测试器生成测试用例与目标程序执行测试用例能够同时进行, 且通过 server 完成载入目标文件、解析符号地址等重复性工作.

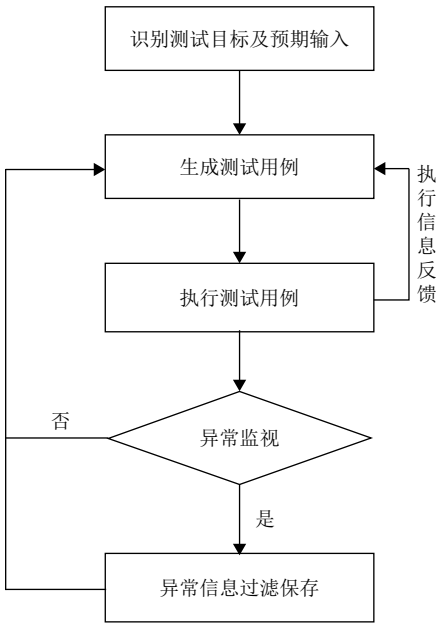

图 1 模糊测试基本流程

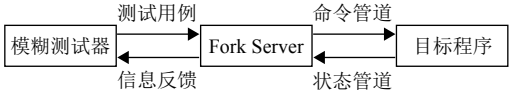

图 2 AFL 的 Fork Server 机制

在每次运行的时候, AFL 都会维护一个全局分支覆盖表, 这个表的索引即为编译过程中 AFL 对目标程序的分支所赋予的随机数标识 R_i, 而每个索引对应着一个长度为 8 个比特位的向量 b, 向量 b 中记录着该分支在目标程序在整个运行的过程中执行的次数. 这里的次数并不是一个准确值, 而是一个区间值, 向量 b 的第 i 位被置为 1 则代表着该分支的执行次数在区间$[2^i, 2^{(i+1)}]$内. AFL 通过每次检查分支覆盖是否发生变化来判断生成的测试用例是否需要保留, 从而避免保留所有测试用例对模糊测试速度造成的负面影响, 并且为复现程序执行异常状态提供了测试数据保障.

AFL 维护着一个队列, 每次从这个队列中取出一个文件, 对其使用一种基于遗传算法的变异策略进行变异[19]. 主要的变异类型[20]如下:

(1) FLIPBIT: 按位或是按字节翻转. 按照 1 个比特或是一个字节为步长从头开始对原始文件进行翻转操作, 整个翻转的过程不会增加或是减少输入文件的

bit 位, 因此不会改变原文件的格式信息

(2) ARITH: 对字节进行整数加减运算, 加减变异的上限默认为 35, 并且 AFL 会按照大端序以及小端序两种整数表示方式进行变异.

(3) INTEREST: 替换一些特殊内容到原文件中. 每次对 8、16 或是 32 个比特位及逆行替换, 替换之后不会影响输入文件长度.

(4) HAVOC: 完全随机的比特位翻转或对文件中某一块内容进行删除或是随即写操作.

(5) SPLICY: 在任意位置拼接两个完全不同的文件.

3 设计思路

本节介绍了我们的模糊测试器在设计过程中的三个关键部分, 第一部分是结合插桩代码直观的抽取程序的控制流信息, 第二部分是使用污点分析的方法获取测试文件中的关键字节, 最后一部分介绍了我们的变异策略.

3.1 获取程序的控制流信息

程序的控制流图可以描绘程序的控制流信息, 一个流图是一个有向图, 它由有限多个节点的结合 NG=$\{n_i\}$和节点间的有向边的集合$EG \subseteq NG \times NG$构成 $G = <NG, EG>$. 流图中的一个结点 n_i 表示程序的一列顺序计算, 而有向边则表示计算的控制转移. 我们的目的是通过控制流抽取的方法, 获取反汇编目标程序中的条件转移信息, 具体指的是判断语句的地址以及代码块的起止地址. 在我们的设计中, 实际上并不需要顺序执行的代码块中的所有信息. 此外, 在 2.2 小节中我们已经介绍了 AFL 中使用 afl-gcc 编译工具以及 afl-as 汇编工具对程序进行插桩的过程, 其中 AFL 为每一个选择分支都设置了一个随机数作为该分支的标识, 因此:

令目标程序中所有被 AFL 插桩过的节点集合为 N, 那么, 对于任意 $n_i \in$N, 我们这里可以通过读取插桩代码中设置的随机数 R_i 作为节点 n_i 的标识, 并通过反汇编工具得到节点中条件语句的位置 R_i_loc, 然后在反汇编代码中搜寻标识代码块起止位置, 并纪录为 $<Starti_loc, Endi_loc>$二元组, 最终存储在返回结果中. 因此我们可以将控制流图中的一个节点 n_i 抽象为一个包含节点标识, 条件语句位置, 节点起止位置四部分内容的数据结构:

$n_i = <Ri, Ri\_loc, Starti\_loc, Endi\_loc>$

从插桩代码中获取控制流中节点信息的流程如算法 1 所示.

算法 1. 获得条件语句节点信息

输入: 插桩后的二进制代码 (目标程序)
输出: 包含节点信息的数据结构 n_i

```
1 AF = Disassemble(F);
2 While not Eof (AF) do
3 {
4   Line = GetLine (AF);
5   if Line ⊆ CMP_Inst; {//判读是否为条件指令
6     n_i→R_i = GetKeyFromPin(Line); // 得到条件指令在插桩代码中的标识
7     (n_i→Starti_loc, n_i→ Endi_loc)=GetAddr(Lcur); //得到代码块的起止位置
8     n_i→Ri_loc = GetCmpLoc(Line); //得到条件指令位置
9     n_i = n_i →next;
10 }
11 else Line = Next (Line);
12 }
13 return n_i;
```

3.2 惰性污点分析

为了有效处理文件格式信息, 模糊测试器需要知道程序的执行路径受到输入文件中哪些字节的影响, 由此需要使用污点分析技术. 通常, 格式文件由两部分组成: 第一部分是关键字 (标识符), 用于标记文件格式或其后特定长度字节的语义, 如 PNG 图片文件由一系列数据包 (package) 组成, 每一个数据包以特定的关键字开头; 第二部分则为由其前驱关键字声明的数据类型的具体数据. 在进行污点分析之前, 我们利用 010Editor 的文件格式模版和 pfp 解析库对输入文件进行解析, 后者依赖前者得到输入文件中每个字节相对于其前驱关键字的偏移量. 关键字信息则可以由模版文件中直接获得. 我们选择以字节作为污点标记的最小单元, 并将输入文件中所有字节都标记为不同的污点源. 我们需要记录跟踪应用程序所在内存空间以及 8 个通用寄存器中每个字节的污点标签. 然而根据我们的需求, 这里只需要定位 3.1 节中得到的条件语句位置, 并通过内存污点跟踪系统提取每个条件语句位置上流通过的污点数据. 若两个污点同源, 则记录其源污点数据. 若两个污点数据属于同一关键字节组, 则记录当前字节的前驱关键字节信息, 以及当前字节相对于其前驱关键字节的字节偏移量. 由于每个条件语句可

能与多个污点字节有关, 我们最终建立的将是条件语句与字节位置信息之间的一对多映射关系. 污点分析的结果用十字链表结构 (如图 3 所示) 存储.

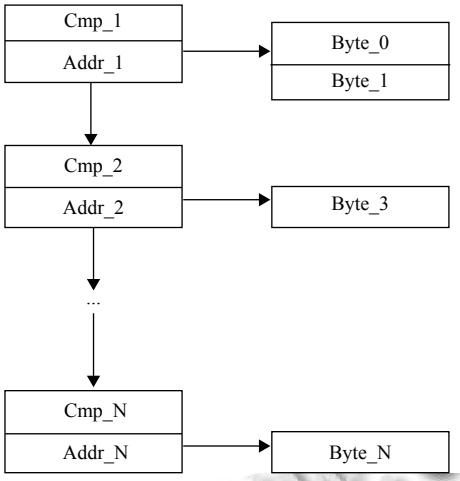

图 3 存储污点分析结果的十字链表

这里我们所做的工作主要是找到包含文件格式信息的字节, 后续的工作将会检查这些字节的变异是否会导致路径覆盖的增加, 如果增加了路径覆盖, 那么模糊测试器将会继续增加这些关键字节变异的概率. 相对于变异文件中的数据字节, 变异这些带有格式信息的字节往往能帮助我们拓展更多的程序执行路径.

然而污点分析是一个极其占用运行时间的方法, 尤其是我们需要以字节为单位进行污点分析的时候. 自动化模糊测试测试工具最大的优势在于极快的运行速度, 因此大部分的模糊测试器没有引入污点分析的方法对输入文件进行分析. 不过我们的研究发现, 对于基于遗传算法变异的模糊测试器来说, 并不需要对每一个输入文件进行污点分析. 以 AFL 维护的队列为例, 对于起始输入 *id_1* 来说, 我们需要对其进行污点分析, 以得到 *id_1* 中字节偏移信息对条件语句的影响, 并将最终的结果保存在一张映射表中. 对于经过遗传算法变异 *id_1* 得到的输入文件 *id_2_src_1*, 正如 2.2 小节中介绍的那样, AFL 的变异策略中 BITFLIP、ARITH 等部分并不会对文件的长度进行改变, 甚至不会改变输入文件中的字节信息, 因此没有必要对其进行字节级别的污点分析. 此外, 由于我们的污点信息记录的是各个污点字节相对于其前驱关键字的相对偏移, 我们需要做的只是将每一个变异文件与其父文件联系起来, 对其进行判定, 如果发现了相同的关键字, 则认为文件格式没有做出更改, 则可以直接使用其父文件的字节偏移信息映射表标记当前输入中个字节的语义. 于是, 即使变异后文件长度和父文件的长度不一致, 也无须对变异文件直接进行字节级别的污点分析, 从而很大程度上减少了对模糊测试整体速度的影响.

3.3 变异策略

如 2.2 小节中所提到的, AFL 的变异策略有很大的随机性, 因此往往会出现无法达到深层次路径覆盖的情况. 而我们的设计可以通过污点分析得到输入文件中的字节偏移对条件语句判决的影响, 并参考这个信息对输入文件进行更有针对性的变异, 这样做不仅可以减少很多无效的变异, 提高变异效率, 而且由于加入了文件格式信息对变异的影响, 因此对于深层次路径的探索也大有裨益. 另外, 我们已经得到条件语句在插桩代码中的标识 R_i, 因此我们可以获取插桩代码中的标识 R_i 与输入文件中的字节偏移之间的映射关系.

$$R_i \rightarrow (Start_i\_byte, End_i\_byte) \tag{1}$$

如果能够顺利得到输入文件的这种映射关系, 则对其采用特殊的定向变异策略, 否则采用随机变异的方法. 如果变异结果可以产生新增的路径覆盖, 那么监督程序将判断其为有价值的变异, 并将其保存到输入队列. 我们设计的模糊测试器对输入文件一次完整的处理过程如图 4 所示.

然而, 由于污点分析结果产出缓慢, 所以并不是所有输入文件在执行的过程中都能建立起这样的映射, 并且在实验过程中我们观察到, 原始的几个输入文件在输入进去的时候一定无法建立起这样的映射关系, 但是实际上, 原始的几个输入文件往往可以通过随机变异的方法就能产生大量新的路径覆盖. 因此我们设计了这样的变异策略: 对于原始的输入文件 *id_1*, 我们采取随机的方式对其进行变异, 如果产生了新的路径覆盖, 则将变异结果保存到输入队列命名为 *id_2_src_1*, 以纪录该变异结果新增的路径覆盖信息. 由 2.2 小节我们可以知道, 插桩代码会对每一个路径覆盖赋予一个标识 R_i. 当 *id_1* 的随机变异过程完成时, 继续对 *id_2_src_1* 进行随机变异, 并且此时很大概率已经得到了污点分析的结果, 因此可以根据污点分析结果与程序控制流信息的映射得到新增的路径覆盖的标识 R_i 受到输入文件中哪些字节的影响, 并对这些字节进行针对性的变异, 以提升模糊测试的效率. 算法 2 描

述了具体的变异策略.

算法 2: 对输入文件的变异策略

```
1 void Fuzz
2 while Not End of Pool
3 F = GetFileFromPool // 从文件池中取出一个待测试文件
4 if HasTaintTrackingResult(F) ==True // 判断是否已经得到当前文件 F 污点分析的结果, 如果其父文件存在, 那么返回其父文件污点分析结果.
5    F_mutate = RandomMutate(F); // 对 F 进行随机变异
6    if FindNewBranchCoverage(){
7       R_i = GetNewCoverageLabel; // 找到新增路径的标号
8       Save(F_mutate);
9       (start, end) = SearchtResult(R_i);// 在污点分析结果中找到公式 1 所示映射关系.
10      F_mutate = Mutate(star, end); //对关键字节进行变异
11         go to 7;}
12      else
13         Abondon(F_mutate); //如果没有产生路径覆盖则舍弃变异结果
14   else
15      F_mutate = RandomMutate(F);
16      go to 7;
```

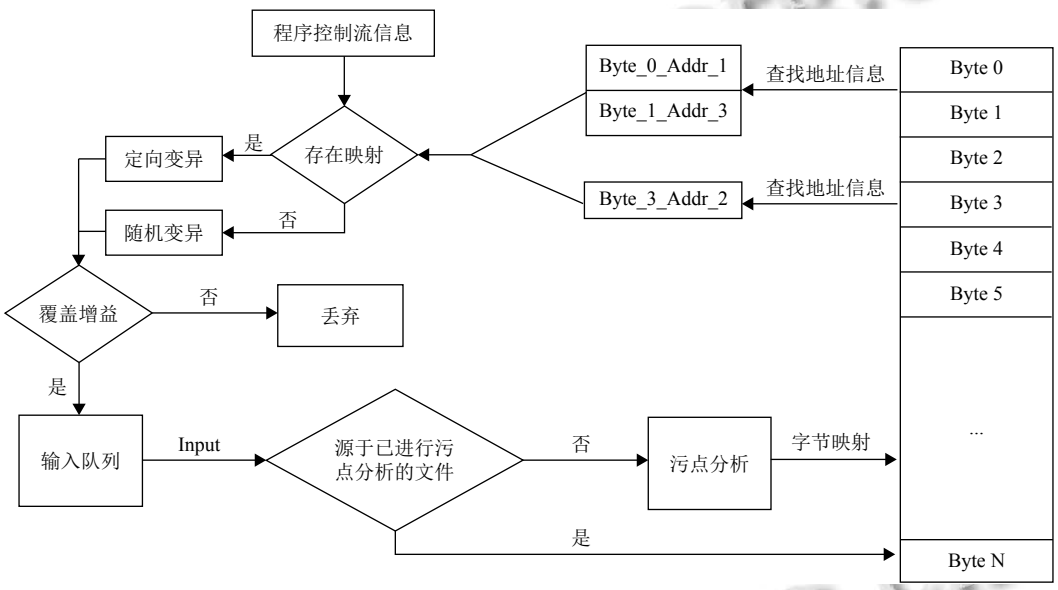

图 4 输入文件的处理过程

4 实验评估

我们在 AFL 的基础上实现了本文中所描述的设计 TaintFuzzy, 并将此工具与原版的 AFL_2.52 以及对每一个输入文件进行污点分析的早期版本 TaintFuzzy_pre 分别进行了 24 小时的对比试验. 实验目标为 libpng 库中的 readpng 程序, 起始的标准输入为 AFL 自带的 png 格式图片, 模糊测试的指令都为 readpng-name.

表 1 给出了对比实验的具体实验数据, 第一列为模糊测试器的名称, 第二列为 24 小时内模糊测试器在目标程序内探索的总路径数, 第三列为模糊测试器探索的路径深度, 第四列为覆盖率的变化, 第五列为平均执行速度的变化. 数据中第二行为 AFL 的测试数据, TaintFuzzy_pre 是对每一个输入文件进行污点分析时的测试数据, 第四行 TaintFuzzy 是对输入文件直接使用其父文件的字节偏移信息映射表的测试数据. 将 AFL 的实验数据以及 TaintFuzzy 的数据对比可以看出, TaintFuzzy 虽然牺牲了一部分执行速度, 但是在总路径数、路径深度以及分支覆盖率上都有一定的提升, 其中总路径数相对提升 12.26%, 分支覆盖率相对提升了 7.93%. 其中值得注意的是, 路径深度提升了 2 层, 相对提升为 33.33%. 路径深度的增加代表着模糊测试能够同时命中更多的选择分支.

表 1 模糊测试结果比较 (24 小时)

	总路径数	路径深度	分支覆盖率 (%)	平均执行速度
AFL	1093	6	3.28	303.46
TaintFuzzy_pre	775	6	2.84	193.17
TaintFuzzy	1227	8	3.54	290.43

我们的方法在执行过程中引入了动态污点分析技

术, 而频繁的污点分析必然会给平均执行速度带来一定幅度的下降. 结合表 1 中 TaintFuzzy_pre 的数据可以看出, 当我们对每一个输入文件都进行污点分析时, 平均执行速度以及 24 小时内执行路径覆盖都会大幅下降. 但是采用了对输入文件直接使用其父文件的字节偏移信息映射表的方法后平均执行速度为 290.43 sec/s, 相比 AFL 的执行速度只降低了约 4%. 由此可以看出, 只使用父文件的字节偏移信息映射表将为模糊测试器的效率带来大幅提升.

图 5 描述了 24 小时之内总路径数, 以及路径深度随时间变化的曲线, 虚线是 TaintFuzzy, 实线是 AFL 的曲线. 由图中可以看出, 除了能达到较高的探索路径总数之外, TaintFuzzy 总是能领先 AFL 达到更深的路径, 并且在第 4 小时以及第 10 小时的时候可以明显看出, 每回合路径深度的增加, 都可以较大幅度的提升探索路径的数量. 如算法 2 所述, 由于 TaintFuzzy 在每次模糊测试器发现新的执行路径的时候, 加入了结合文件格式信息的变异方法, 因此在拓展相同深度执行路径的过程中, TaintFuzzy 能够更好的对路径信息加以利用, 从而达到提升路径搜寻速度的效果, 因此图 5 中 TaintFuzzy 在增加总路径的方面表现出了更高的斜率.

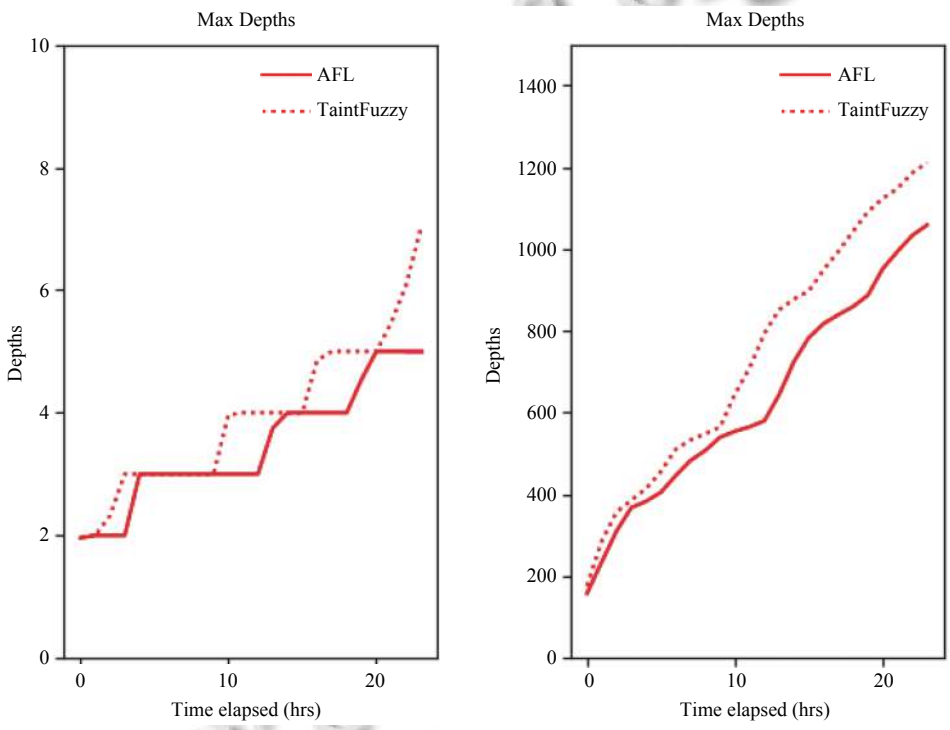

图 5 程序路径探索随时间变化的情况

5 结束语

模糊测试是一种能够高效地对软件脆弱性进行全面分析的方法. 本文设计了一种能够综合利用测试文件上下文信息以及程序执行路径信息的模糊测试器, 并且通过将污点分析得到的绝对位置转化成相对于文件格式关键字的相对位置, 减少了污点分析总工作量. 模糊测试在未来的一段时间内仍将是一种软件漏洞挖掘的利器, 模糊测试工具的改进也对实际软件开发有着重要意义.

参考文献

1 Eagle C. The IDA Pro Book. 2nd ed. San Francisco: No Starch Inc, 2011. 14–31.

2 Sharif M, Yegneswaran V, Saidi H, *et al*. Eureka: A framework for enabling static malware analysis. In: Jajodia S, Lopez J, eds. Computer Security – ESORICS 2008. Berlin, Heidelberg: Springer, 2008. 481–500.

3 Vostokov D. Windbg: A Reference Poster and Learning Cards. OpenTask, 2008. 126–141.

4 Miller BP, Fredriksen L, So B. An empirical study of the reliability of UNIX utilities. Communications of the ACM,

1990, 33(12): 32–44. [doi: 10.1145/96267.96279]
5 Sadowski C, Aftandilian E, Eagle A, *et al*. Lessons from building static analysis tools at Google. Communications of the ACM, 2018, 61(4): 58–66. [doi: 10.1145/3200906]
6 Technical "whitepaper" for afl-fuzz. http://lcamtuf.coredump.cx/afl/technical_details.txt.
7 Wüstholz V, Christakis M. Learning inputs in greybox fuzzing. arXiv: 1807.07875, 2018.
8 Peng H, Shoshitaishvili Y, Payer M. T-Fuzz: Fuzzing by program transformation. Proceedings of the 2018 IEEE Symposium on Security and Privacy. San Francisco, CA, USA. 2018. 697–710.
9 Rawat S, Jain V, Kumar A, *et al*. VUzzer: Application-aware evolutionary fuzzing. San Diego, CA, USA: Internet Society, 2017. [doi: 10.14722/ndss.2017.23404]
10 Schwartz EJ, Avgerinos T, Brumley D. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but Might Have Been Afraid to Ask). Proceedings of the 2010 IEEE Symposium on Security and Privacy. Berkeley/Oakland, CA, USA. 2010. 317–331.
11 Li YK, Chen BH, Chandramohan M, *et al*. Steelix: Program-state based binary fuzzing. Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering. Paderborn, Germany. 2017. 627–637.
12 Chen P, Chen H. Angora: Efficient fuzzing by principled search. Proceedings of the 2018 IEEE Symposium on Security and Privacy. San Francisco, CA, USA. 2018. 855–869.
13 Li Z, Zou DQ, Xu SH, *et al*. VulDeePecker: A deep learning-based system for vulnerability detection. arXiv: 1801.01681, 2018.
14 Wang JJ, Chen BH, Wei L, *et al*. Skyfire: Data-driven seed generation for fuzzing. Proceedings of the 2017 IEEE Symposium on Security and Privacy. San Jose, CA, USA. 2017.
15 Drozd W, Wagner MD. FuzzerGym: A competitive framework for fuzzing and learning. arXiv: 1807.07490, 2018.
16 Shoshitaishvili Y, Wang RY, Salls C, *et al*. SOK: (State of) the art of war: Offensive techniques in binary analysis. Proceedings of the 2016 IEEE Symposium on Security and Privacy. San Jose, CA, USA. 2016. 138–157.
17 Böhme M, Pham VT, Roychoudhury A. Coverage-based greybox fuzzing as Markov Chain. Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security. Vienna, Austria. 2016. 1032–1043.
18 Cohen MB, Snyder J, Rothermel G. Testing across configurations. ACM SIGSOFT Software Engineering Notes, 2006, 31(6): 1–9.
19 Binary fuzzing strategies: What works, what doesn't. https://lcamtuf.blogspot.sg/2014/08/binary-fuzzing-strategies-what-works.html.
20 Cha SK, Woo M, Brumley D. Program-adaptive mutational fuzzing. Proceedings of the 2015 IEEE Symposium on Security and Privacy. San Jose, CA, USA. 2015. 725–741.