基于模型约束的灰盒模糊测试技术

孙伯文，崔宝江
（北京邮电大学网络空间安全学院）

摘要：灰盒模糊测试技术是现在常用的并且行之有效的一种模糊测试技术。该模糊测试技术可以通过获取程序执行时的路径执行信息来指导模糊测试的方向。但是目前市面上常见的灰盒模糊测试系统都是基于随机变异的模糊测试系统，这样导致样本的穿透性十分的低，而且坏样本的数量也很多，会浪费很多不必要的算力。本文将提出利用模型约束技术来改进灰盒模糊测试系统的样本生成模块，通过对目标软件格式进行定义，从而有针对性的生成特定格式的样本，增加模糊测试中样本文件的穿透性，同时提高模糊测试效率，增加发现漏洞的可能性。

关键词：灰盒模糊测试；模型约束；样本生成；漏洞挖掘；
中图分类号：TP309.1

Grey box fuzzy technology based on model constraints

Sun Bowen, Cui Baojiang
(School of Cyberspace Security, Beijing University of Posts and Telecommunications)

Abstract: Grey box fuzzing technology is a kind of fuzzing technology that is commonly used now and effective. The fuzzing technology can guide the direction of fuzzing by acquiring the path execution information when the program is executed. However, the common gray box fuzzing test systems on the market are based on random mutation fuzzing test systems. This results in very low sample penetration and a large number of bad samples, which will waste a lot of unnecessary computing power. This article will propose the use of model constraint technology to improve the sample generation module of the gray box fuzzy test system. By defining the target software format, a specific format of the sample can be generated in a targeted manner to increase the penetration of the sample file in the fuzzy test. Improve fuzz testing efficiency and increase the possibility of finding vulnerabilities.

Key words: Grey-box fuzzing; Model constraints; Sample generation; Vulnerability mining;

0 引言

基于覆盖率的灰盒模糊测试技术（CGF）[1] [2] [3]是目前一种非常流行并且行之有效的模糊测试方法，被广泛运用于各种程序的模糊测试中。它相对于白盒模糊测试来说，基于覆盖率的灰盒模糊测试技术仅仅只需要收集程序运行时的部分信息就可以，但是目前常用的白盒模糊测试大多基于符号执行[4] [5] [6] [7]，需要大量的计算能力，而且还会遇到路径爆炸问题，而基于覆盖率的灰盒模糊测试可以有效地降低算力需求。同时它相比较于黑盒测试[8] [9]来说，可以通过对程序执行时信息的收取发现更多的优秀样本种子文件，从而可以有效地指导样本种子变异方向，达到一个更好的效果。

基于覆盖率的灰盒模糊测试系统是一种在牺牲大量复杂程序分析的情况，只收集程序的部分信息，通过分析这部分信息，从而使得模糊测试可以探索到更多程序的执行路径。CFG

作者简介：孙伯文（1995-），男，硕士研究生，主要研究网络空间安全方向
通信联系人：崔宝江（1972-），男，教授，博士生导师，主要研究网络空间安全. E-mail: cuibj@bupt.edu.cn

通过工具在程序执行的基本块中插入一条记录程序执行路径的插桩代码，而每条插桩代码中都记录了该基本块的唯一标识。然后模糊测试系统就可以在程序执行的时候，实时获取到程序的执行路径信息。这样模糊测试系统在生成一个新的样本的时候，会去分析该样本是否会产生一条新的路径，如果样本产生了一条新的路径，则会将该样本作为一个优秀的样本，并将其记录下来，作为新一轮模糊测试的种子文件。如果没有产生新的路径，则会认为该样本与之前同路径样本是同一个样本文件，所以会选择抛弃该样本文件。然后通过不断的循环和变异来产生更多的样本文件。

现在基于覆盖率的灰盒模糊测试系统的设计思路已经广泛运用在各种模糊测试系统中，在许多工具中以 AFL[10]最为有名，该工具通过挖掘出数百个开源项目的漏洞[11]来证明了基于覆盖率的灰盒模糊测试技术在实战中的有效性。但是 AFL 也存在一些问题。在 AFL 的变异模块中，运用了翻转、算数加减、边界值替换、块的删除、插入等操作来进行随机变异，而对于现在大部分的程序来说，这种随机化的变异策略很难达到程序的深层路径，如今的文件大多都具有特定的格式，例如文档文件（PDF、WORD）、可执行文件（ELF、PE）、音频文件（MP3、WAVE 等）、视频文件（MP4、RMVB 等）、压缩文件（RAR、ZIP）。上述这些文件都具有其特定的文件格式，而只靠随机变异，很难对深层路径进行模糊测试。而本文将基于模型约束的模糊测试技术与基于覆盖率的灰盒模糊测试相结合，将模型的概念应用到 AFL 中，从而提高 AFL 生成的样本针对特定格式的程序有很强的穿透性，可以对深层代码进行充分的模糊测试。

1 背景知识

1.1 基于覆盖率的模糊测试技术

基于覆盖率的模糊测试技术是目前一个非常火爆的技术，其代表作 AFL（American Fuzzing Lop）[10]是一个高水平的基于覆盖率的灰盒模糊测试工具，利用轻量级的编译期插桩和基于遗传算法的自动化测试用例生成策略，以生成能够触发程序中存在的缺陷的测试用例。AFL 在编译被测程序期间对被测程序进行插桩，使其能够在运行时获得每次用测试用例执行被测程序的过程中分支的执行情况，根据记录的运行时分支执行信息计算测试输入的分支覆盖能力，将能够触发新分支的测试用例视为感兴趣的测试用例，作为后续模糊测试过程的种子输入。

AFL 的插桩过程是通过在源代码编译过程产生的汇编文件中插入特定插桩代码，用于获取程序执行时候的路径信息[10]，具体代码逻辑如图 1 所示：

```
cur_BBL = <RANDOM>;
shm[cur_BBL ^ prev_BBL]++;
prev_BBL = cur_BBL >> 1;
```

图 1　AFL 插桩指令。

Fig. 1　AFL instrumentation instructions.

在插桩过程中，每一个基本块的插桩代码都会生成一个随机数，用于表示当前基本块的

唯一性，然后记录当前基本块的位置的时候通过对 cur_BBL 与 prev_BBL 进行异或，然后利用异或之后的结果在 shm 中去记录，并利用这条记录标记当前的路径（从 prev_BBL 跳转到 cur_BBL）。然后再交换 prev_BBL 与 cur_BBL，而之所以会有 prev_BBL = cur_BBL >> 1 主要原因是为了区分基本块 A 到基本块 B 和基本块 B 到基本块 A 的情况，使得在记录过程中，路径是具有方向性的。这样在程序的所有的基本块中插入插桩代码，就可以记录下程序在执行过程中的路径信息。并基于该路径信息来筛选样本种子的优劣性。

AFL 通过覆盖率信息来决定保留哪些生成的样本文件，并判断这些样本文件是不是优秀的，然后更具新的样本文件进行变异。整体的算法流程[10]如下：

算法 1 AFL 模糊测算法

```
Input: Seed Inputs S
    Tx = Ø
    T = S
    if T = Ø then
        add empty file to T
    end if
    repeat
        t = ChooseNext(T)
        p = AssignEnergy(t)
        for i from 1 to p do
            tmp = MutateInput(t)
            if tmp crashes then
                add tmp to Tx
            else if IsInteresting(tmp) then
                add tmp to T
            end if
        end for
    until timeout reached or abort-signal
    Output: Crashing Inpots Tx
```

如果在模糊测试的初始阶段，用户提供种子文件，则将用户提供的种子文件加入到样本队列 T 中。否则，将一个空文件放到样本队列 T 中用于作为初始种子文件。接下来，会有一个大循环，程序在这个循环中，会不停地对目标程序进行模糊测试，直到超时，或者被用户终止为止。而在执行的过程中，只有那些产生了新的路径的种子，才会被保留下来，然后并被不停地模糊测试。

对于每个种子，系统会通过 AssignEnergy 函数对该种子进行评分，然后给该种子赋予一个能量，即该种子变异的次数和程度，然后通过变异算法对样本文件 t 进行 p 次变异。在 MutateInput 中的变异策略包括：

Bitflip：按位翻转变异，该变异策略会对样本文件进行挨个 bit 的翻转，并且会进行连续 1bit 翻转，连续 2bit 翻转等等，一直到连续 32bit 翻转，进行模糊测试，同时还会将连续的路径不变的 bit 作为一个 token。用于后面变异做准备。

Arithmetic：整数加/减算术运算，该变异策略会对样本中的一些特殊位置进行加减运算，从而发现一些整数溢出上的问题。

Interest：替换原文内容，在计算机中，有一些比较特殊的值，例如：八字节的-128、-1、0、1、16、32、64、10、127；十六字节的-32768、-129、128、255、256…等等。而这些特殊的值，往往在处理上可能会存在处理不当的问题，所以这里对这些值进行替换，从而发现类似的问题。

Dictionary，字典变异，用户可以提供一些在模糊测试中可能有用的字符串加入到模糊测试系统的字典中，模糊测试系统会将用户提供的字符串随机插入、替换到样本文件中，从而增加发现新路径的可能性。

Havoc，破坏性变异，该变异策略会对样本文件进行大量的破坏其中包括：随机选取某个 bit 进行翻转、随机选取某个 byte，将其设置为随机的 interesting value、随机选取某个 byte，对其减去一个随机数、随机选取某个 byte，对其加上一个随机数、随机删除一段 byte 等等，会对文件造成大量修改，从而期望发现新的路径。

Splice，文件连接变异，在 seed 文件队列中随机选取一个，与当前的 seed 文件做对比。如果两者差别不大，就再重新随机选一个；如果两者相差比较明显，那么就随机选取一个位置，将两者都分割为头部和尾部。最后，将当前文件的头部与随机文件的尾部拼接起来，就得到了新的样本文件。在利用新的样本文件去对程序进行 Fuzz。

该系统通过 MutateInput 来生成新的样本文件之后，就会将该样本文件喂给待测程序，并且收集待测程序的执行信息。再通过对每个样本的执行信息进行分析，如果当前样本产生了一个新的路径，则认为生成的样本 tmp 是有趣的样本文件，并将该样本加入到循环队列 T 中。而在存储程序路径信息的时候，采取了存储桶的方法来避免路径爆炸的问题，然而这必然会带来部分路径碰撞的问题，不过相对于结果是一个可以接受的范围。

如果生成的样本 tmp 使得程序崩溃了，则将样本 tmp 放到崩溃队列 Tx 中，并且这也会成为新的变异种子。

1.2 基于模型约束的模糊测试技术

目前许多程序文件都都存在着基本格式的问题，而只有满足一定格式的情况下，程序才会继续执行下去。这些格式包含有魔术字、校验位、摘要、长度等等。例如下表：

表 1 ELF 文件结构表。

Tab. 1 ELF file structure table.

文件类型	字段名	字段含义
ELF	e_ident	其中包括 ELF 文件的标志（'\x7fELF'）文件类别、编码格式、文件版本等信息。
	e_type	文件类型
	e_machine	处理器体系架构
	e_version	目标文件版本
	e_entry	程序入口段
	e_phoff	程序头表的偏移
	e_shoff	节头表的偏移

	e_flags	处理器特定的标志位
	e_ehsize	ELF 文件头大小
	e_phentsize	程序头表中没一个表项的大小。
	e_phnum	程序头表中的表项个数
	e_shentsize	节头表中每一个表项的大小
	e_shnum	节表中的表项个数
	e_shstrndx	节头表中与节名字表相对应的表项的索引

这是一个 ELF 头里面的结构[12]，再这个结构中 e_ident 的魔术字、e_ehsize、e_phentsize、e_shentsize 本身就是一个固定的值，而如果对这种值也进行随机变异，就会让待测程序在检测阶段就直接终止，从而让后面的数据无法触碰到更深层次的逻辑。像 e_type、e_machine、e_version 等本身就有一个具体的范围，而超出该范围之后也会让程序早早的就终止，导致对其后的变异不具有任何意义。

```
static int
process_file_header (void)
{
  if (   elf_header.e_ident[EI_MAG0] != ELFMAG0
      || elf_header.e_ident[EI_MAG1] != ELFMAG1
      || elf_header.e_ident[EI_MAG2] != ELFMAG2
      || elf_header.e_ident[EI_MAG3] != ELFMAG3)
    {
      error
    (_("Not an ELF file - it has the wrong magic bytes at the start\n"));
      return 0;
    }

  ...

  return 1;
}
```

图 2　readelf 代码片段。

Fig. 2　readelf code snippet.

图 2 是从 readELF 中节选出来的部分代码，从代码中可以看出，当魔术字无法对应上的时候，程序会认为这不是一个 ELF 文件，则会强制退出。这样就会导致后续的变异无效后，同时也会导致模糊测试过程中做很多无用功，模糊测试程序并不知道这儿要填上固定的字符串。同样对于许多压缩文件，压缩文件在解压之前都会对待解压文件进行校验和的计算，而如果校验和不对，则会导致解压程序拒绝解压该文件。但是对于一般模糊测试程序来说，如果单纯靠随机变异使得校验和恰好和变异之后的文件相匹配其实是十分困难的，而且会浪费大量的算力来穷举这个过程。

而基于模型约束的模糊测试技术就是为了解决这个问题而提出的一种方案[8] [13]，其核心思想是在模糊测试前对目标文件格式进行描述，并且给出文件格式的一些约束信息，模糊测试程序在根据这些约束信息去针对性的生成符合约束的样本文件[14] [15]，从而达到可以深层次模糊测试的目的。例如可以通过模型约束开头必须是魔术字，某个结构中的关键字段必须是一个固定的值。或者通过模型指定某个结构是另外一个结构的校验和等。通过大量的对文件格式的描述，从而约束生成的样本，使模糊测试程序变异出的样本可以直接通过待测程序中的大量检测逻辑，从而达到深层，更容易出漏洞的模块，这样再通过对深层数据进行变异，

不断的对深层逻辑进行模糊测试，从而发现更多潜藏在大量逻辑之下的深层漏洞。

2 基于模型约束的灰盒模糊测试

基于覆盖率的模糊测试技术的优点是可以利用较低的程序分析来有效的指导样本的生成，这样可以避免没用意义的变异。而基于模型约束的模糊测试技术的有点也十分明显，其针对于哪些具有格式高度要去的程序有很强的样本穿透性，并且可以根据事先编写好的模型文件直接绕过程序内部很多检测，从而集中算力运用再易出错的部分进行变异。而如果将这两种方法相互结合，则可以提高任何一种模糊测试的效果，所以通过结合上述两种模糊测试方案，给出新的设计图如图 3 所示：

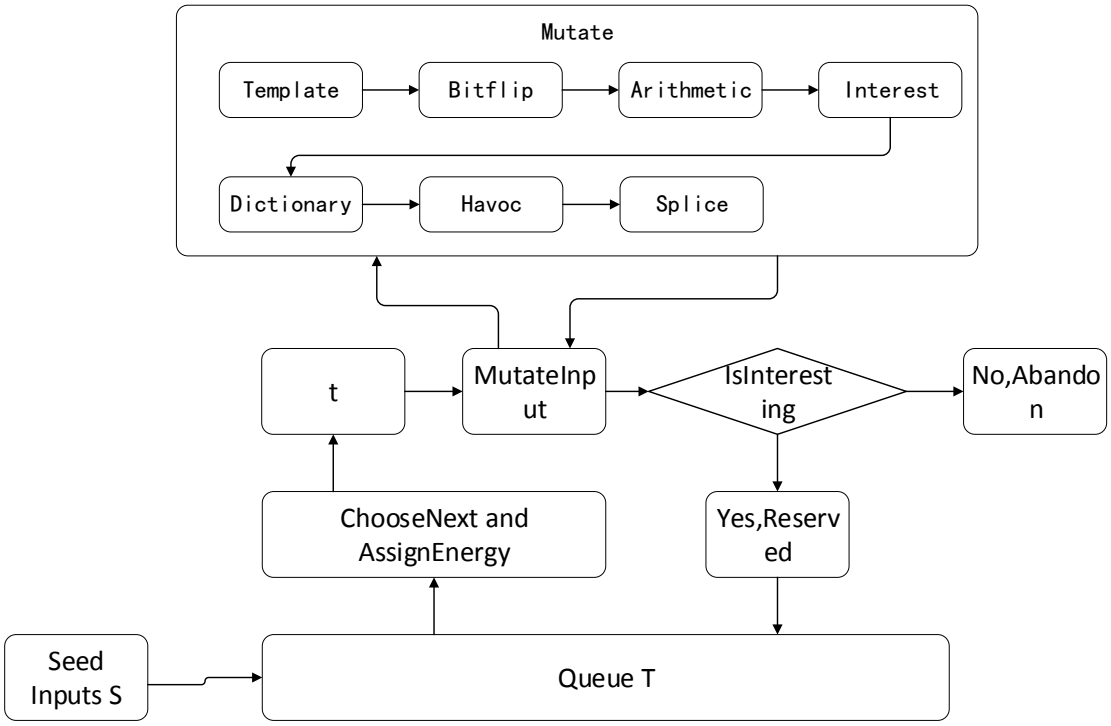

图 3 模糊测试设计图。

Fig. 3 Fuzz design.

整体流程同 AFL 类似，但是在变异模块进行修改，在原有变异模块的基础上加上 Template 变异模块，变异模块算法化如下：

算法 2 变异模块算法

```
function Mutate(seed s):
    Template(s)
    Bitflip(s)
    Arithmetic(s)
    Interest(s)
    Dictionary(s)
    Havoc(s)
    Splice(s)
end function
```

其中 Template 是基于模型变异的模块，先通过基于模型变异模块生成较为符合目标程序文件结构的样本种子文件，使得生成的样本种子具有极强的穿透性。然后再通过 Bitflip、Arithmetic、Interest、Dictionary、Havoc、Splice 等一系列随机变异，增强样本的随机性，提高样本种子发现新路径和发现新崩溃的概率，从而提高模糊测试的效率。

在 Template 变异规则里面主要利用模型约束技术，使样本数据能够按照模型文件约定的形式生成。模型文件采用 XML 格式来描述，按照输入数据的设计说明格式来描述各字段的约束属性，包括字段类型、长度、标识、约束关联等。其结构如下图所示：

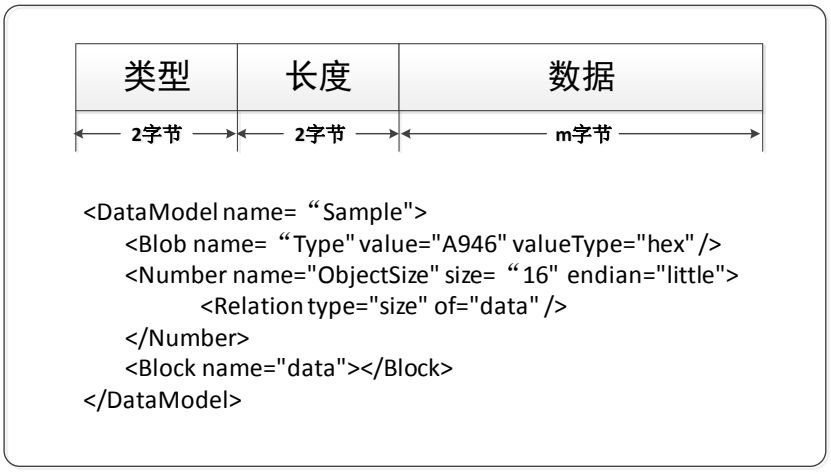

图 4　基于 XML 格式的样本数据格式描述。

Fig. 4　Description of sample data format based on XML format.

图 4 上为样本格式要求，图 4 下为 XML 格式的样本结构描述，在生成样本数据时，样本变异模块将基于该模型文件解析待变异的样本文件，如果能够成功解析，则按照指定的变异策略以模型中的字段为单位进行变异，如果不能成功解析，则按照指定的变异策略以样本文件整体为单位进行变异。

字段类型包括字符串型、数据型、无具体数据类型、分组类型。字符串型包含了名称，长度等常用属性；数据型可以是 8、16、32 或 64 字节的数据，并通过大小属性来设定，同时也包含名称属性作为名称；无具体数据类型用来定义不明类型的数据，包含名称、长度、值等属性；数据分组类型用来组合一个或者多个其他元素 ，如数字类型或者字符串类型。

字段中可以指定字段的默认值，可以指定某段数据是一个固定的值，用于绕过魔术字检测。在一个文件格式中也会具有偏移等相关变量，而都可以通过模型来进行描述，通过指定某个字段是地址，然后针对该地址处进行深度的变异，从而有针对性的对程序中部分功能进行深入检测。利用该 XML 还可以进行校验和的指定，在一些压缩包格式中存在很多校验和的检测，而绕过这些校验和的检测也是模型约束需要做的，通过指定某个变量是检验和，从而绕过程序内部的校验和的检测。

通过上述的变异策略，生成大量的优质样本文件，通过将这些样本文件喂给程序去执行，在执行完成之后，通过对样本的执行路径进行分析，在选取一批优秀的样本，参与到下一轮的循环中，从而不停的对目标程序进行模糊测试下去。

3 实验和结果

3.1 实验设计

为了验证算法的性能，通过对过往的软件进行模糊测试，并观察产生的路径和崩溃数来判断验证算法的性能。这里选择对 binutils 2.25 版本库中的 nm 这个小工具来进行模糊测试。之所以选择老版本的库，是为了更容易 fuzz 出崩溃。

为了实验具有对照性，在同一台机器上，分别利用改进的 AFL（下称 AFL-Template）对 nm 这款小工具进行 12 小时的模糊测试，并再利用原生的 AFL 对 nm 进行 12 小时的模糊测试。然后通过查看输出目录下的 queue 和 crashes 这两个目录下的文件创建时间来进行统计生成样本文件的速度。

3.2 实验结果

图 5 是 12 小时内，对 nm 这款小工具进行模糊测试的时候 AFL 和 AFL-Template 的样本生成数量的对比图：

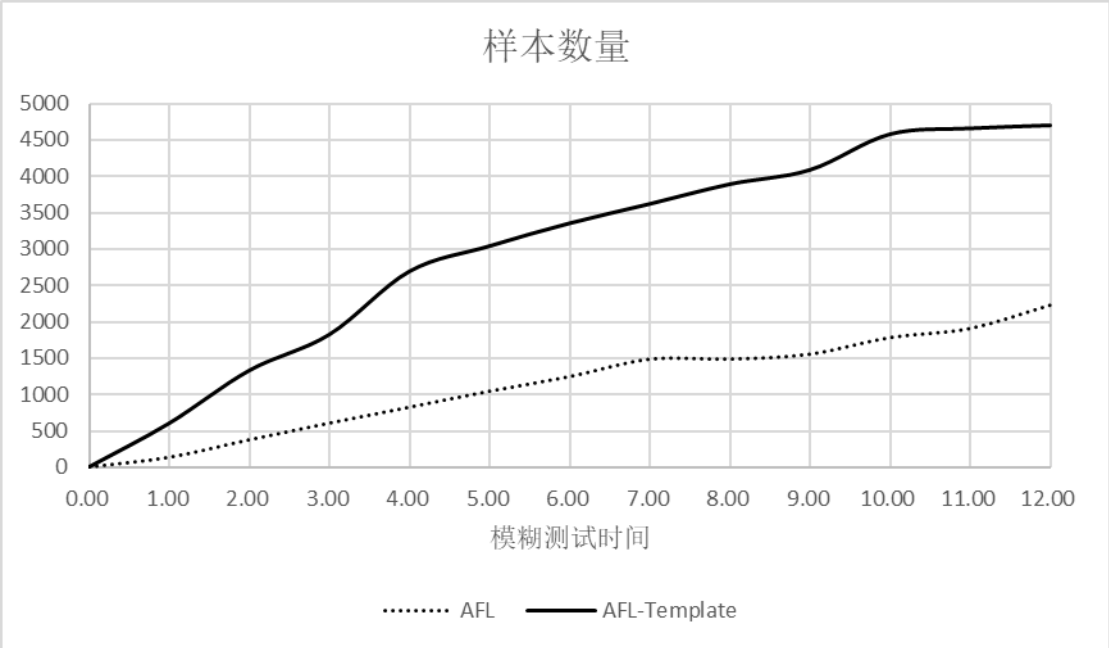

图 5　样本生成数量对比图。

Fig. 5　Comparison of the number of samples generated.

从图 5 中的数据可以看出前十个小时内，AFL-Template 的样本数量增长率明显高于 AFL，当运行时间达到十个小时之后，AFL-Template 的增长率逐渐降低，这是由于样本种子数量过多，导致模糊测试一轮循环时间较长，因此样本数量的增长在放缓，但是整体的样本个数却一直领先于 AFL。且 AFL-Template 的模糊测试效率也优于 AFL。

图 6 是 12 小时内，对 nm 这款小工具进行模糊测试的时候，AFL 和 AFL-Template 的崩溃样本数量的对比图：

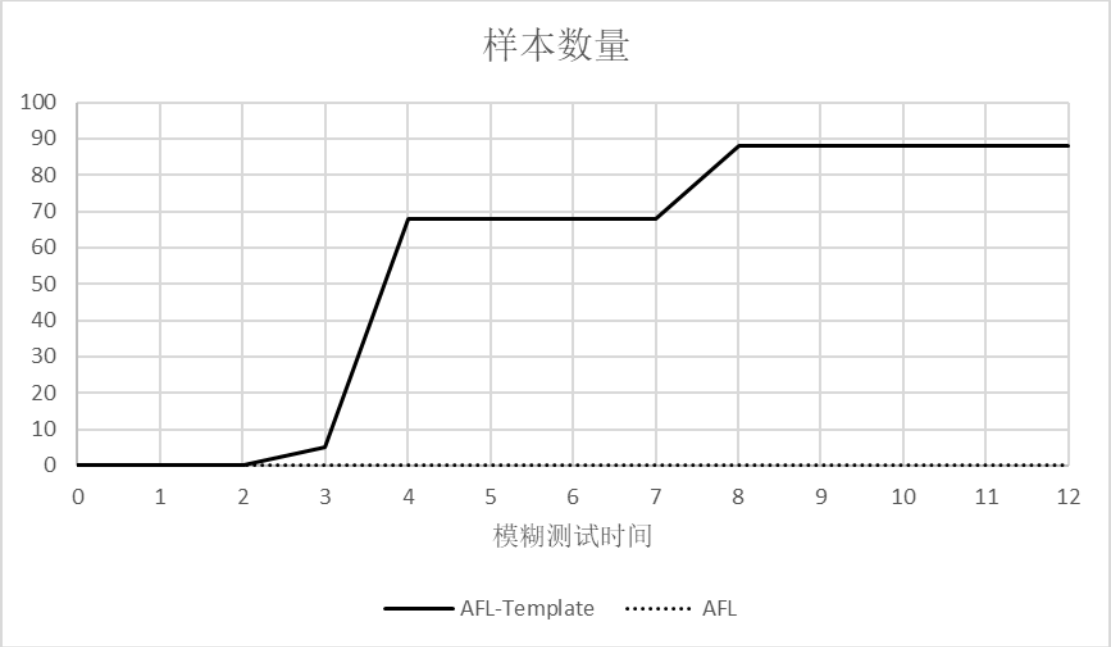

图 6　崩溃样本数量对比图。

Fig. 6　Comparison of the number of crash samples.

从图 6 中的数据可以看出前两个小时都没有产生崩溃样本，而当第三个小时开始的时候，AFL-Template 先产生了一个崩溃样本，之后 AFL-Template 通过该崩溃样本生成了更多的样本，然而 AFL 在 12 个小时内的模糊测试中，并没有产生崩溃样本。但是这里并不是说 AFL 不会产生崩溃样本，当随机变异次数足够多的时候，AFL 是可能也产生类似的崩溃样本的，但是因为不具有模型指导，所以无法像 AFL-Template 生成十分高效的样本文件去对目标进行模糊测试。所以导致了 AFL-Template 更快的产生了崩溃样本文件。

4　结论

本文中介绍了一种针对 AFL 的改进方案，通过将基于模型的模糊测试技术和基于覆盖率的模糊测试技术相结合，利用选取两种方案的优点，来提高样本文件的穿透性。通过增加模型约束，来控制变异出的样本文件，使变异出的样本文件具有更强的穿透性，可以再初期直接绕过一些魔术字检测，校验和检测等，从而让样本中其他变异部分可以充分的发挥作用。另一方面又通过路径反馈来检测每一个样本的优越性，摒弃只产生崩溃才是优秀样本的概率，从而提高样本整体的质量，使得模糊测试过程可以发现更多的崩溃，挺高测试程序的安全性。

[参考文献] (References)

[1] Chen C, Cui B, Ma J, et al. A systematic review of fuzzing techniques[J]. Computers & Security, 2018, 75: 118-137.
[2] Takanen A, Demott J D, Miller C, et al. Fuzzing for software security testing and quality assurance[M]. Artech House, 2018.
[3] Takanen A. Fuzzing: the Past, the Present and the Future[C]//Actes du 7ème symposium sur la sécurité des technologies de l'information et des communications (SSTIC). 2009: 202-212.
[4] Website, "Symbolic execution in vulnerability research," https://lcamtuf.blogspot.sg/2015/02/symbolic-execution-in-vuln-research.html, accessed: 2019-06-13.

[5] Pham V T, Böhme M, Roychoudhury A. Model-based whitebox fuzzing for program binaries[C]//2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE). IEEE, 2016: 543-553.
[6] 宋雪勤. 符号执行在软件安全领域中的研究与应用[D].西华大学,2018.
[7] 李政宇. 基于混合执行的二进制程序模糊测试关键技术研究[D].北京邮电大学,2017.
[8] Schieferdecker I, Großmann J, Schneider M. Model-Based Fuzzing for Security Testing[C]//Keynote talk at the 3rd International Workshop on Security Testing (SECTEST 2012), Montreal, Canada (April 2012). 2012.
[9] 曾淑娟,刘嘉勇,刘亮.基于Peach框架的测试样本优化方法的研究[J].网络安全技术与应用,2016(01):96-97.
[10] Website, "AFL technical details," http://lcamtuf.coredump.cx/afl/technical_details.txt, accessed: 2019-06-13.
[11] Website, "Afl vulnerability trophy case," http://lcamtuf.coredump.cx/afl/#bugs, accessed: 2019-06-13.
[12] Website, "Executable and Linkable Format (ELF)," http://www.cs.cmu.edu/afs/cs/academic/class/15213-f00/docs/elf.pdf, accessed: 2019-06-13.
[13] 梅国浚. 基于遗传算法和模型约束的漏洞挖掘技术研究与实现[D].北京邮电大学,2019.
[14] Website, "Peach Fuzzer Platform," http://www.peachfuzzer.com/products/peach-platform/, accessed: 2019-06-13.
[15] Schieferdecker I, Grossmann J, Schneider M. Model-based security testing[J]. arXiv preprint arXiv:1202.6118, 2012.