DOI: 10.3785/j.issn.1008-973X.2020.08.011

基于动态能量调控的导向式灰盒模糊测试技术

戴渭，陆余良，朱凯龙
(国防科技大学 电子对抗学院，安徽 合肥 230037)

摘 要：导向式灰盒模糊测试(DGF)是能够快速生成测试用例，达到给定的程序目标区域并且发现漏洞的模糊测试技术.针对当前DGF技术测试效率较低的问题，提出基于动态能量调控的DGF技术.通过静态分析技术构建程序的函数调用图(CG)和控制流图(CFGs)，定义并计算更准确的函数级别、基本块级别的目标距离；通过跟踪种子的执行轨迹，计算种子到目标区域的距离；基于动态能量调控函数对模糊测试中种子的变异数量进行更有效的调控，引导生成到达目标区域的测试用例.基于该方法，实现导向式模糊测试原型系统AFL-Ant，并与现有的导向式模糊测试方法进行对比实验.结果表明，本研究所提出的方法能够更加快速、有效地对目标区域进行测试，在补丁测试、漏洞复现方面具有较强的应用价值.

关键词：灰盒模糊测试；静态分析；距离计算；动态能量调控；导向式模糊测试
中图分类号：TP 309　　**文献标志码：**A　　**文章编号：**1008−973X(2020)08−1534−09

Directed grey-box fuzzing technology based on dynamic energy regulation

DAI Wei, LU Yu-liang, ZHU Kai-long
(*College of Electronic Engineering*, *National University of Defense Technology*, *Hefei* 230037, *China*)

Abstract: Directed gray-box fuzzing (DGF) is a kind of fuzzing technology which can quickly generate test cases to reach a given target area of the program and find vulnerabilities. A DGF technology based on dynamic energy regulation was proposed, aiming at the inefficiency of existing DGF technology. The function call graph (CG) and control flow graphs (CFGs) of the program are constructed by static analysis technology, and the more accurate target distance at function level and basic block level is defined and calculated. The distance from seed to the target area is calculated by tracking the execution trajectory of the seed. The dynamic energy regulation function is used to effectively control the mutation quantity of seeds in the process of fuzzing, and to guide the generation of test cases that can reach the target area. A prototype system AFL-Ant for DGF was implemented based on this method, and the comparison experiments with the existing DGF method were carried out. Results demonstrate that the proposed method can test the target area faster and more effectively, and it has strong application value in patch testing and vulnerability reproduction.

Key words: grey-box fuzzing; static analysis; distance calculation; dynamic energy regulation; directed fuzzing

模糊测试技术是重要的软件漏洞分析技术，其基本思想是向程序提供大量的特殊或者随机测试用例，然后监控程序运行过程中的异常情况，通过人工辅助分析异常测试用例数据，从而定位软件中的漏洞位置[1].模糊测试已经成为当前漏洞分析技术研究中的热门领域.

传统的模糊测试是黑盒测试，通过构造大量随机的测试用例运行程序，触发程序中的漏洞.虽然测试存在盲目性，但是由于模糊测试具有较好的可伸缩性，在应对大型应用程序时，相较于

收稿日期：2019−07−04.　　**网址：**www.zjujournals.com/eng/article/2020/1008-973X/202008011.shtml
基金项目：国家重点研发计划重点专项资助项目(2017YFB0802900).
作者简介：戴渭(1995—)，男，硕士生，从事漏洞挖掘与利用技术研究. orcid.org/0000-0002-4970-0169. E-mail: 1821007360@qq.com
通信联系人：陆余良，男，教授，博导. orcid.org/0000-0002-1712-4224. E-mail: 451762681@qq.com

其他漏洞分析技术具有更高的效率和更低的开销，而现实中的应用程序往往是大而复杂的，所以模糊测试技术是漏洞分析技术在现实应用中的主流，也是发现漏洞最多的技术. 灰盒模糊测试是通过轻量级的程序分析（主要用于监测）对模糊测试进行辅助的技术，它继承了黑盒测试器的高可拓展性的优点，同时能够较好地理解程序的关键结构，使模糊测试技术向“智能化”发展，是当前模糊测试技术的发展方向.

当前灰盒模糊测试采用符号执行、污点分析以及静态分析等技术帮助模糊测试更快获取程序的有效信息，使得测试覆盖更深、更广的路径，从而更快、更多地暴露程序中的漏洞. Sang 等[2] 开发的模糊测试器 Mayhem 使用混合符号执行技术以及基于索引的存储器建模技术，通过推理程序的行为来辅助生成测试用例. Stephens 等[3] 基于 AFL 开发的 Driller 结合混合符号执行的灰盒测试器，根据已有的输入快速到达检查语句，使用符号执行的求解器求解，得到能够通过检查语句的数值，生成测试用例达到新的路径，提高路径的覆盖率. Rawat 等[4] 开发的 VUzzer 结合静态分析和动态污点分析技术，在程序运行前使用静态分析根据控制流图中的基本块转换构建马尔科夫模型，计算出函数中达到每个基本块的概率，为程序的每个基本块分配不同的权重（如错误处理代码），计算一个种子在遗传变异中的权重，决定它在遗传变异中能够生成的测试用例数量，对更有可能暴露漏洞的路径进行优先级排序，将模糊测试集中在低频位置；结合动态污点分析技术设计种子的变异策略，在魔术字节、地址传送方面优化测试用例的生成，使测试用例能够达到更深的位置和难以通过的位置，增加它触发漏洞的可能. Johansson 等[5] 提出的 T-fuzz 使用二进制重写技术变异目标程序，通过动态追踪找到二进制程序中的检查节点，将检查语句删除得到新的程序库，通过对变异程序的测试产生 Crash，通过轻量级的符号执行对原程序进行 Crash 的复现. Böhme 等[6] 提出的 AFLFAST 使用基于马尔科夫链的模型来计算路径转换概率，将工作集中在对程序中的低频路径进行模糊测试. 灰盒模糊测试技术弥补了传统模糊测试技术的不足，使模糊测试变得更加高效.

当前大多数的模糊测试技术无法实现有效的导向性，当目标只集中在程序中的某一部分时，普通的模糊测试会将大量时间浪费在不相关区域，而无法集中在期望的区域. 导向式模糊测试（directed gray-box fuzzing, DGF）是能够快速生成测试用例达到给定的程序目标区域并且发现漏洞的模糊测试技术. DGF 将大量的时间预算用于达到这些特定的目标位置，对这些区域进行更集中的测试，从而更加高效地发现目标区域的漏洞，减少不必要的开销. 导向式模糊测试无须重量级的符号执行、程序分析和约束求解等. Böhme 等[7] 最新提出的导向式灰盒模糊测试器 AFLGO 在模糊测试框架 AFL[8] 的基础上，将目标区域的可达性作为一种优化，通过计算种子到目标区域的距离，使用启发式算法来提升距离更近的种子的生成测试用例的数量，使其生成到达程序目标区域的测试用例，实现模糊测试的导向性. AFLGO 通过与 AFL 以及导向式符号执行工具 KATCH[9] 的对比实验，证明了其导向性效果，但是它依然存在着以下问题：1）种子到目标区域距离的计算方法不准确；2）生成测试用例数量的调控方法不够有效；3）没有设计自适应的变异策略生成更高质量的测试用例.

为了解决问题 1）、2），本研究提出更加快速有效的 DGF 技术：基于动态能量调控的 DGF 技术. 通过静态分析技术构建程序的函数调用图（call graph, CG）和控制流图（control flow graphs, CFGs），定义更加准确的函数级别、基本块级别的目标距离计算方法；通过插桩技术跟踪种子的执行轨迹，执行基本块所包含的距离信息，计算种子到目标区域的距离；根据构建的动态能量调控函数对模糊测试中种子的变异数量进行更有效的动态调控，引导生成到达目标区域的测试用例，达到更加快速、有效地对目标区域进行测试的目的.

1 方法概述

在 DGF 中，目标区域为由若干个目标函数组成的函数集. 通过静态分析得到被测程序的 CFGs 以及 CG，以计算程序基本块到目标区域的距离. 通过跟踪种子在程序中所执行的基本块，定义种子到目标位置的距离，以距离的大小来分配种子在遗传变异中的能量，即生成测试用例的数量，从而实现模糊测试的导向性. 所提出的 DGF 整体架构如图 1 所示.

对一个程序进行导向式模糊测试，目标区域为程序中若干个函数组成的函数集，模糊测试的

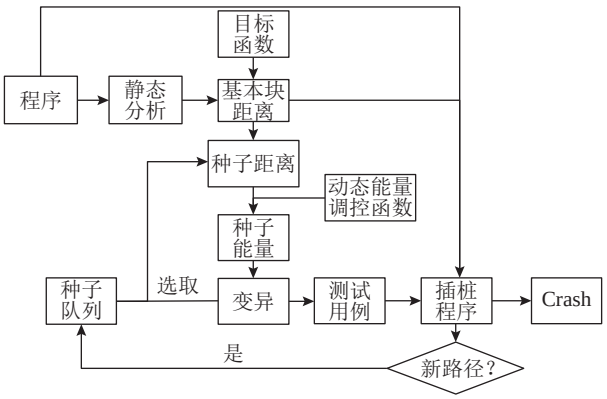

图 1 导向式灰盒模糊测试技术框架

Fig.1 Framework of DFG technology

流程如下：1）在目标程序运行之前，对程序进行静态分析，通过 CFGs 和 CG 计算程序所有基本块到目标区域的距离，将计算得到的距离信息插桩进入目标程序；2）从种子集中按顺序选取一个种子，执行程序，根据它执行的基本块和基本块的距离信息来计算种子到目标区域的距离；3）通过动态能量调控函数动态地调控种子的能量，调控变异生成测试用例的数量；4）如果变异生成的测试用例执行了新的路径，就将此测试用例加入种子集，如果产生 Crash，就加入 Crash 集合，直到该种子生成的测试用例数量达到预定数量；5）重复步骤 2）~ 4），当种子集中的所有种子都被选取为一次迭代，继续从头开始选取种子进行下一次迭代，直到时间结束.

DGF 的工作流程分为 2 个阶段：静态分析阶段（步骤 1））、模糊测试动态循环阶段（步骤 2）~ 5））. 静态分析阶段的输入为目标程序和给定的程序目标区域（若干个函数组成的函数集），得到的结果是每个基本块都插桩了基本块到目标区域距离信息的二进制程序. 模糊测试动态循环部分的目标为插桩后的二进制程序，输入为选定的种子集，输出为导致目标产生 Crash 的测试用例，或者达到预定时间.

所研究的基于动态能量调控的 DGF 技术分为 3 个部分：1）基本块到目标区域距离的计算；2）种子到目标区域距离的计算；3）种子的动态能量调控技术研究. 其中，部分 1）作用在静态分析阶段，部分 2）、3）作用在模糊测试动态循环阶段.

2 基本块到目标区域距离的计算

为了实现模糊测试的导向性，须使更加接近目标区域的种子生成更多的测试用例. 须计算一个种子到目标区域的距离，再根据距离来决定这个种子能够生成多少测试用例. 为了精确地计算种子到目标区域的距离，定义目标程序中所有基本块到目标区域距离的计算方法. 基本块距离计算分为 2 个部分：函数级别距离计算和基本块级别距离计算. 这部分是在程序的静态分析阶段完成的.

2.1 函数级别距离计算

函数级别距离计算用来计算目标程序中所有函数到目标区域的距离. 用静态分析获取程序的 CFGs 和 CG，以计算函数级别和基本块级别的距离. 使用 LLVM 编译器对源码进行编译即可获得程序的 CFGs 和 CG（当只有二进制程序可以使用的时候，可以使用比特码转换器[10]将二进制文件转换为 LLVM 的中间表示），使用 CFGs 和 CG 中的最短路径的边数来表示函数间和基本块间的距离，通过 Dijkstra's 算法[11]分别计算目标区域中每个目标函数到各个基本块的最短距离. 但是这种表示方法不够准确，因为 2 个函数间可能不止一种调用方法，被调用函数也可能不止一个进入点，这样相邻函数间的边应该被赋予不同的权值，因为它们的可达路径不止一种，不能直接简单地用边的条数来表示距离.

如图 2 所示为一个函数调用的示例图. 可以看出，函数 f_a 可以调用函数 f_b、f_c，但是有 2 条路径可以到达函数 f_b（x<10 和 x>20），同时函数 f_b 在 f_c 中出现次数为 2 次，而到达 f_c 的路径只有 1 条且出现的次数也只有 1 次，所以可以认为，f_a 到 f_b 的距离更近，因为它们之间有更多的路径，更容易到达. 为了更好地计算相邻函数间的距离，定义函数间的距离计算公式：

$$d'_1(f_1,f_2)=(2N_{\mathrm{cm}}+1)/(2N_{\mathrm{cm}}), \tag{1}$$

$$d'_2(f_1,f_2)=(2N_{\mathrm{noo}}+1)/(2N_{\mathrm{noo}}), \tag{2}$$

$$d'_{\mathrm{w}}(f_1,f_2)=d'_1d'_2. \tag{3}$$

```
fa (x){
    if (x < 10){
        fb (x);
    }
    else if (x > 20){
        fbc (x + 1);
    }
    else{
        fc (x + 2);
    }
}
```

图 2 函数调用示例

Fig.2 Example of function call

式中：N_{cm}为被调用函数能够被调用的方法个数，N_{noo}为被调用函数在调用函数中出现的次数，d'_1、d'_2为N_{cm}、N_{noo}的影响因子，d'_w为2个相邻函数之间边的权值.

例如，图2中$d'_1(f_a, f_b)$=1.25，$d'_2(f_a, f_b)$=1.25，$d'_w(f_a, f_b)$=1.25×1.25=1.562 5，$d'_1(f_a, f_c)$= 1.5，$d'_1(f_a, f_c)$=1.5，$d'_w(f_a, f_c)$=1.5×1.5=2.25. 通过程序和CG即可算出函数间边的权值，将其作为2个相邻函数之间的距离，再利用Dijkstra's算法分别计算每个目标函数到所有函数的最短距离，从而得到加权后每个函数在调用图中分别到各个目标函数的最短距离.

函数级别的距离计算是计算CG中的每个函数到所有目标函数的平均距离. 对于一个函数f到多个目标函数的距离计算，根据它到每个目标函数的最短距离，使用调和平均值来计算它到目标函数的平均距离作为到目标区域的距离，得到函数级别距离的计算公式：

$$d_f(f,T_f)=\begin{cases}\text{undefined}, & R(f,T_f)=\varnothing;\\ \left[\sum_{t_f\in R(f,T_f)} d_f(f,t_f)^{-1}\right]^{-1}, & \text{其他.}\end{cases} \tag{4}$$

式中：$d_f(f, t_f)$为函数f到目标函数t_f的最短距离，T_f为所有目标函数的集合，$R(f, T_f)$为函数f能够达到的所有目标函数的集合.

2.2 基本块级别距离计算

基本块级别距离计算用来计算程序中所有基本块到目标区域的距离. 虽然种子的距离是程序间的，但是本研究的计算只须对CG和每个程序间CFGs进行一次分析，将基本块级别的计算近似到函数级别的计算. 在同一个程序间CFGs中，将相邻基本块之间的边作为它们的距离，2个基本块之间距离作为它们之间最短路径的边数. 不在同一个CFGs的基本块目标距离通过基本块能够调用的函数来确定，以调用函数的函数级距离的常数倍近似计算，当一个基本块没有可以调用的函数时，通过它所能达到的所有可调用函数的基本块来确定. 以此得出基本块到目标区域的距离计算公式：

$$d_b(b,T_b)=\begin{cases}0, & b\in B_{T_f};\\ a\min\limits_{f\in N(b)}(d_f(f,T_f)), & b\in T;\\ \left[\sum(d_b(b,t)+d_b(t,T_b))^{-1}\right]^{-1}, & \text{其他.}\end{cases} \tag{5}$$

式中：b为基本块；T_b为目标区域；B_{T_f}为目标函数的基本块集合；$N(b)$为基本块b可以调用的函数集合，且满足$\forall b\in T$, $N(b)\neq\varnothing$；$d_b(b, t)$为2个基本块b、t之间最短路径的连接边数；a为常数.

3 种子到目标区域距离的计算

前2个级别的距离计算是在静态分析阶段进行的，而种子级别的距离计算，是在模糊测试循环开始后动态实现的. 通过跟踪种子执行了哪些基本块来计算种子到目标区域的距离. 在AFL和LibFuzzer[12]灰盒模糊测试器中，使用轻量级的插桩来获取程序的覆盖信息，这种插桩能够获取基本块的转换信息，以及粗略的分支命中计数. 在此基础上进行拓展，将得到的所有基本块距离向量加入程序的插桩，得到插桩后的二进制程序，这样就可以追踪并记录种子执行的基本块以及基本块所包含的距离信息，计算出种子到目标区域的平均距离：

$$d(s,T_f)=\frac{\sum_{b\in B(s)} d_b(b,T_b)}{|B(s)|}. \tag{6}$$

式中：$B(s)$为种子所执行的基本块.

这种将种子所执行的基本块到目标区域距离的平均数作为种子的距离的方法，存在过分聚焦短路径的问题. 如图3所示为种子的执行路径示例. 图中，T为目标区域. 可以看出，s_1、s_2执行的基本块分别为(1, 2, 3, 5, 7, 8)、(1, 4, 6, 9)，s_1和s_2都已经较接近目标区域，那么它们到目标区域的距离应该相似. 简单假设图中基本块到目标的距离为边的条数，按照式(6)计算2个种子的距离：$d(s_1, T)$=(1+2+3+4+5+6)/6=3.5；$d(s_2, T)$=(1+2+3+4)/4=2.5. 可以看出，路径更短的s_2到目标的距离较小，为了减小长路径种子与短路径种子距离大小差距，计算种子的调和因子$\varphi(s)$，

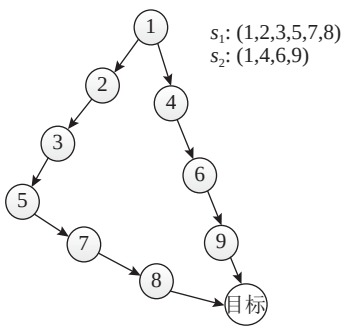

图3 种子的执行路径示例

Fig.3 Example of seed execution path

来缓解过分聚焦短路径的问题,得到调和后的种子距离:

$$\varphi(s)=\left(\sum_{b\in B(s)} d_b(b,T_f)^{-1}\right)^{-1}, \tag{7}$$

$$d'(s,T_b)=\varphi(s)d(s,T_b). \tag{8}$$

将得到的种子距离归一化,得到最终的距离向量:

$$d'(s,T_b)=\frac{d'(s,T_b)-d'_{\min}(s,T_b)}{d'_{\max}(s,T_b)-d'_{\min}(s,T_b)}. \tag{9}$$

式中:$d'_{\max}(s,T_b)$和$d'_{\min}(s,T_b)$分别为所有种子中离目标的最大距离和最小距离.

4 种子的动态能量调控

种子的能量调控用来调控所选中种子的变异次数,简单来说,如果当前的种子距离目标区域更加接近,给它分配更多的能量,增大变异次数,这样更有可能生成所期望的测试用例(到达目标区域).在AFLGO[7]中,开发了基于模拟退火算法的功率调度,随时间变化慢慢将更多能量分配给距离目标更加接近的种子,模拟退火算法在特定的时间内可以以一定的概率接受较差的种子,随着时间调节对较差种子的接受度.在一开始,它对所有的不同种子有着最大的接受度,在预定的退火时间之后,它只接受更好的种子,避免一开始就过度聚焦于距离更近的种子,使种子过早收敛,陷入局部最优.这种功率调度方法随时间对种子生成的测试用例数量进行调度,当种子集中存在过多的种子,或者被测程序运行一个测试用例的时间远远超出预期时,难以起到预期效果.在预定的退火时间后,种子只执行了其中一部分,那么功率调度便失去了所期望的效果,没有对所有的种子产生足够的影响,当作用在某些种子身上时,已经到了退火的末期甚至已经结束了.所以基于退火算法的功率调度不够稳定有效.为了解决这个问题,本研究提出能量动态调控方法:基于蚁群算法[13]的动态能量调控函数,动态地调控种子的能量,控制测试用例的生成.

4.1 蚁群算法

蚁群算法是用来寻找最优化路径的概率型算法,具有分布计算、信息正反馈和启发式搜索的特征,本质上是进化算法中的启发式全局优化算法.为了解决DGF中种子会过早收敛在较短执行路径,导致程序目标区域覆盖率不高的问题,可以将蚁群算法与种子的能量调控相结合.蚁群算法的基本思想如下:1)N只蚂蚁从一个出发点到四周搜索食物,N只都搜索完毕为一轮迭代;2)在行进的路上释放信息素,信息素的量和解的质量成正比.当每只蚂蚁找到食物时,回到出发点,在路径留下信息素,留下的量和距离成正相关,在第1次搜寻时,所有的路径具有相同的信息素,同时每次迭代后信息素按比例减少;3)蚂蚁选择一个路径的概率和信息素的量成正比.

质量较高的路径在每次迭代中信息素浓度越来越高,经过的蚂蚁也越来越多,而质量较差的路径的信息素逐渐减少,这样在迭代中,蚂蚁从刚开始的随机搜寻慢慢地往最优解收敛,从最开始的接受较差解到最后的只接受更优解,在一定程度上缓解过早收敛的问题.

在蚁群算法中,路径上的信息素是根据迭代动态变化的,在多次迭代之后信息素的差异会缓慢变大,通过一次次的迭代,蚂蚁最终收敛在最优路径.通过模拟蚂蚁觅食,用模糊测试中选取种子的迭代次数的增加来逐渐增大种子能量差异,动态地调控种子能量.

4.2 基于蚁群算法的动态能量调控函数

设定模糊测试中的种子为蚂蚁搜索的路径,种子与目标的距离远近为解的质量,即距离越近,质量越高,每次迭代留下来的信息素也就越多.将蚂蚁选择路径的概率用作能量调控,随着迭代次数的增加,信息素的量也在增加,种子的能量也逐渐增加,由此来实现动态能量调控函数.

$$\left.\begin{aligned} E_i(t)&=\frac{[\tau_i(t)]^a}{\sum_{s\in S}[\tau_i(t)]^a},\\ \tau_i(0)&=C,\\ \Delta\tau_i(t)&=QE_i(t)(1-d_i),\\ \tau_i(t+1)&=(1-\rho)\tau_i(t)+\Delta\tau_i(t),\\ e_i&=\frac{E_i(t)}{E(t)_{\max}},\\ P_i(t)&=(1-d_i)(1-e_i)+e_i. \end{aligned}\right\} \tag{10}$$

式中:Q、C为常数;S为种子集;t为迭代次数;s_i为种子i;d_i为种子i到目标区域的距离;τ_i为t_i路径上的信息素;ρ为信息素蒸发系数,$0<\rho<1$;$E_i(t)$为种子s_i在第t轮迭代时的启发度(即蚁群算法中选取一个路径的概率);$P_i(t)$为种子s_i在第t轮迭代时的能量.

启发度$E_i(t)$的大小和信息素的多少成正相关,而$E_i(t)$决定着能量$P_i(t)$的大小.在被测程序刚开始运行时,种子集中所有种子的信息素为相同数值,它们的路径也有相同的启发度和相同的

能量,在新一轮迭代中,上一轮的信息素会以蒸发系数的比率减小,同时会增加新得到的信息素 $\Delta\tau_i(t)$, $\Delta\tau_i(t)$根据种子到目标的距离远近以及上一轮信息素的多少来决定,这样在一轮轮的迭代下,距离目标更近的种子的路径上的信息素会越来越多,能量也会越来越大,不同种子的能量便在迭代之中产生差距.

如图 4 所示为在某次模糊测试中不同距离 d 的种子在迭代中的能量 $P(t)$的变化, $P(t)\in[0, 1.0]$. 刚开始,所有种子有着相同的能量,随着迭代次数的增加,不同种子的能量慢慢改变,最后稳定在一个值,这个值与种子距离目标的距离成负相关. 距离为 1.0 的种子在经过若干轮的迭代后,能量从一开始的 1.0 快速下降,最后稳定在 0;距离为 0 的种子在迭代中一直保持着高能量. 距离目标区域越近的种子,在迭代中能保持越高的能量. 如图 5 所示为在不同时刻,种子的距离对能量的影响. 可以看出,种子到目标的距离越大,种子的能量越小.

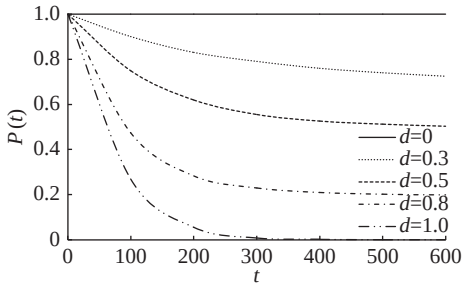

图 4 迭代次数对种子能量的影响

Fig.4 Effect of iteration number on seed energy

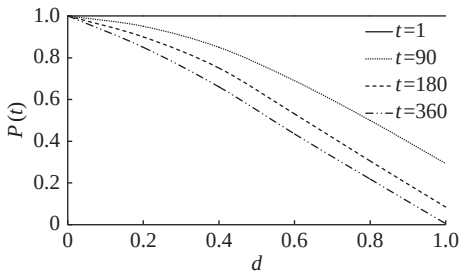

图 5 种子距离对种子能量的影响

Fig.5 Effect of seed distance on seed energy

5 系统实现及实验分析

5.1 原型系统的实现

为了实现和评估上节所提出的关键技术在 DGF 中的有效性与实用性,在模糊测试框架 AFL 的基础上设计原型系统 AFL-Ant. 设计以 python 脚本形式实现的距离计算器,输入为 CFGs、CG(CG)和目标函数集,使用 networkx 包解析图形,使用 Djikstra's 算法计算基本块到目标的距离,输出基本块距离信息 BB-distance. AFL-Ant 拓展了 AFL 的插桩,将 BB-distance 插桩到目标程序的基本块,在记录基本块转换的同时,记录所执行基本块的距离累积值,程序的拓展插桩使用 ASAN[14] 构建. AFL-Ant 的模糊测试器基于 2.40b 版本的 AFL,通过记录的基本块距离累计值计算当前的种子距离,使用本研究所设计的基于蚁群算法的动态能量调控方法来调控种子能量,将其与 AFL 集成,得到最终的能量调控公式,控制种子生成测试用例的数量:

$$\left.\begin{aligned} f &= 2^{8(P_i(t)-0.5)},\\ P'_i(t) &= P_{\text{afl}}(i)f. \end{aligned}\right\} \tag{11}$$

式中:f 为能量调控因子, $P_{\text{afl}}(i)$为 AFL 的策略中种子 s_i 的能量, $P'_i(t)$为最终种子在 DGF 中的真实能量.

在 AFL 能量调控的基础上控制种子的真实能量,根据 $P_i(t)$得到 f 的指数为 [−4, 4],作用在 f 上得到 $f\in[1/16, 16]$. 调控因子是根据 DGF 在实际应用中的表现得来的,要求 f 的范围使得种子能量差异能够拉大,又不至于完全忽略距离较远的种子,通过在原型系统中的实际测试,发现 [1/16, 16] 较合适.

在若干次迭代后,不同距离的种子能量差如图 6 所示. 其中,图 6(b)为图 6(a)的局部扩放. 可以看出,距离目标较近的种子调控因子能够保持较高,较远种子调控因子会快速下降到较低,最终会稳定在一定的数值上,这样便实现了根据迭代扩大种子能量差异的能量动态调控,例如:

$$P'_i(0)=16P_{\text{afl}}(i),\ \lim_{t\to\infty}P'_i(t)=P_{\text{afl}}(i)/16;\quad d=1.0, \tag{12}$$

$$P'_i(0)=16P_{\text{afl}}(i),\ \lim_{t\to\infty}P'_i(t)=P_{\text{afl}}(i);\quad d=0.5, \tag{13}$$

$$P'_i(0)=16P_{\text{afl}}(i),\ \lim_{t\to\infty}P'_i(t)=16P_{\text{afl}}(i);\quad d=0. \tag{14}$$

简单来说,距离目标区域较远的种子,能量将会在迭代之中变得越来越少. 这种动态调控函数根据迭代来变化,相较于根据时间来变化,表现更加稳定.

5.2 实验与评估

为了评估 AFL-Ant 在导向式模糊测试中的效果,选择几个常用目标程序,通过漏洞复现[15] 以及目标站点覆盖实验对其能力进行评估.

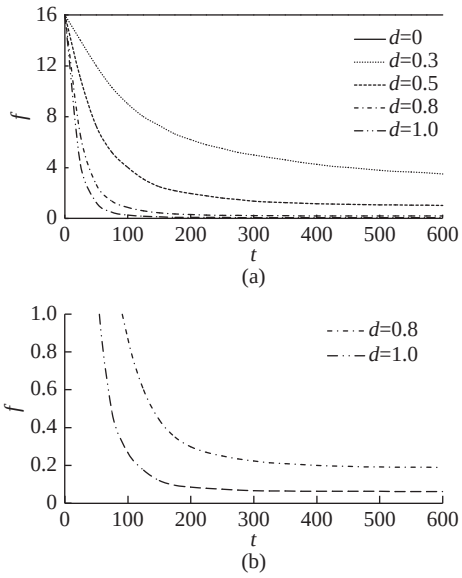

图 6 能量调控因子的变化

Fig.6 Change of energy regulatory factors

选取已经报告过的一些公开漏洞，通过比较复现这些漏洞所用的时间来评估 AFL-Ant 在导向性方面的能力. 结合 AFLGO 已经做过的工作，选取 2 个目标程序：GNU Binutils、Libpng[16]. GNU Binutils 是一组二进制工具的集合，包括连接器、汇编器和其他用于目标文件和档案的工具，它有将近 100 万行代码；Libpng 是读写 PNG 文件的跨平台的库，拥有将近 50 万行代码，它们都是被广泛使用的开源项目. 选取一些美国国家漏洞数据库（NCD）[17] 中报告的漏洞，这些漏洞用 CVE 编号来标识. 实验分别用 3 个工具对目标程序进行测试，每次测试时间上限设置为 8 h，对每个漏洞重复 20 次实验. 暴露时间 TTE 为给定的漏洞被生成的测试用例第 1 次触发所用的时间，使用特定版本的补丁修复来确定暴露的错误是否为给定的漏洞，当产生 Crash 的输入在固定版本修复后无法再次运行产生错误，则认为这个输入触发了给定的漏洞. 改善因子 F 为 AFLGO 和 AFL-Ant 的 TTE 与 AFL 的 TTE 的商，表示相较于 AFL 的效率提升了多少. 效应量 $\hat{A}_{12}$[18] 表示 AFLGO 和 AFL-Ant 产生的结果优于 AFL 的概率，即实验中 TTE 小于 AFL 的 TTE 的概率.

如表 1 所示，在 GNU Binutils 的漏洞复现结果中，除了 CVE-2016-4 490 之外，AFL-Ant 和 AFLGO 的速率都比 AFL 快 1~3 倍，其中有 4 个 CVE 的重现 AFL-Ant 比 AFLGO 快，另外 3 个 CVE 的重现 AFLGO 比 AFL-Ant 快. CVE-2016-4 491 和 CVE-2016-6 131 的复现较困难，耗时较长，在实验设置的时间内成功复现的概率较低，AFL-Ant 在

表 1 GNU Binutils 的漏洞复现结果

Tab.1 Results of GNU Binutils' vulnerability reproduction

漏洞编号	工具	次数	TTE / s	F	$\hat{A}_{12}$
2016-4 487	AFL-Ant	20	161	1.60	0.72
	AFLGO	20	181	1.42	0.60
	AFL	20	257	−	−
2016-4 488	AFL-Ant	20	531	1.94	0.81
	AFLGO	20	683	1.51	0.71
	AFL	20	1 031	−	−
2016-4 489	AFL-Ant	20	190	2.24	0.72
	AFLGO	20	175	2.43	0.68
	AFL	20	425	−	−
2016-4 490	AFL-Ant	20	91	0.66	0.31
	AFLGO	20	87	0.69	0.28
	AFL	20	60	−	−
2016-4 491	AFL-Ant	10	23 262	0.87	0.49
	AFLGO	5	24 121	0.83	0.41
	AFL	7	20 139	−	−
2016-4 492	AFL-Ant	20	531	1.78	0.89
	AFLGO	20	518	1.82	0.83
	AFL	20	943	−	−
2016-6 131	AFL-Ant	7	20 131	1.32	0.68
	AFLGO	5	21 230	1.25	0.63
	AFL	3	26 590	−	−

20 次实验中成功复现这 2 个的 CVE 的次数分别为 10、7 次，相较于 AFL 的 7、3 次，以及 AFLGO 的 5、5 次，AFL-Ant 有较好的表现.

在 GNU Binutils 中 CVE-2016-4 490 的复现实验中，AFL 所用的时间更短，原因是这个漏洞处在易暴露的位置，不需要目标区域的指导，模糊测试器能较快找到它，而 AFLGO 和 AFL-Ant 的种子的距离计算和拓展的执行跟踪会占用更多的资源，导致这个漏洞的复现花费更多的时间. CVE-2016-4 491 中 AFL 所用时间更短，分析漏洞位置发现原因是其位置涉及了间接调用，而导向式模糊测试的距离计算未考虑间接调用的情况，所以在实验中没有达到预想中的导向性.

如表 2 所示，在 Libpng 的漏洞复现结果中，AFL-Ant 和 AFLGO 的速率比 AFL 快 2~10 倍，相较于 GNU Binutils 中被测试的项目，Libpng 更加复杂，代码量也更多，所以可以较直观的看到，

DGF 在大型程序中能够有更好的效果. 在实验中, 有 4 个 CVE的复现 AFL-Ant 比 AFLGO 更快, 另外 1 个 CVE的复现 AFLGO 比 AFL-Ant 要快. 在 Libpng 的测试中, 每次实验均实现了漏洞的复现.

实验表明, AFL-Ant 的策略是有效的, 在对大型软件的测试中的效果尤其明显, 在漏洞复现中能够发挥有效的作用, 相较于 AFLGO 有较为明显的提高. 从实验整体的效应量$\hat{A}_{12}$上来看, AFL-Ant 在 2 个实验中的表现均比 AFLGO 好, 表明它能够保证更加稳定有效的导向性.

为了检测 AFL-Ant 中基于蚁群算法的动态能量调控函数是否比 AFLGO 的功率调度有着更稳定、更有效的表现, 以及 AFL-Ant 的距离计算方法是否比 AFLGO 的方法更加准确有效, 用 AFL-Ant、AFLGO 以及 Ant-GO, 针对程序中的特定目标站点进行覆盖实验. 其中 Ant-GO 使用 AFL-Ant 静态分析方法和 AFLGO 功率调度的模糊测试器.

在目标程序中, 某些位置难以被导向式模糊测试覆盖, 尽管这些位置不一定会触发崩溃, 但是能够快速覆盖这些位置, 提高导向式模糊测试的覆盖率, 也是衡量导向式模糊测试能力的一个标准, 选取目标程序 Libpng 和 Libtiff 中特定位置作为目标站点, 目标站点用文件名和行数来表示, 分别用 4 个工具对目标程序进行测试, 每次测试时间上限设置为 8 h, 对每个目标站点重复 10 次实验, 比较每次实验中到达目标站点的速度.

如表 3 所示为目标站点覆盖实验的结果, AFL-Ant 和 Ant-GO 都能在实验中表现出较好的效果, 相比于 AFL 在到达站点的时间上提升了约 2~6 倍. 在每组实验中, AFL-Ant 和 Ant-GO 都能比 AFLGO 和 AFL 使用更短的时间到达目标站点. 在站点 1、3 的实验中, 目标位置较容易到达, 3 个工具在 10 次实验中, 都能在指定时间内成功覆盖目标站点, 而在站点 2、4 的实验中, 目标站点的位置更加深入, 须花费更多的时间才能到达目标站点, AFL-Ant 成功覆盖目标站点的次数分别为 7、9 次, Ant-GO 为 6、7 次, AFLGO 和 AFL 分别为 5、5 次和 3、4 次.

从实验结果上来看, AFL-Ant 和 Ant-GO 在目标站点覆盖中有着比 AFLGO 更好的表现, 能够在更短的时间内到达目标站点, 而对 AFL-Ant 与 Ant-GO 的实验结果进行比较, AFL-Ant 能够更快地到达目标站点. 从站点 2、4 的覆盖成功次数和效应值$\hat{A}_{12}$上来看, AFL-Ant 相比于 Ant-GO 和 AFLGO, 在每组 10 次的重复实验中, 有更大概率能成功到达目标站点, 也有更大概率能产生更好的实验结果, 有更加稳定的表现. 证实本课题所

表 2 Libpng 的漏洞复现结果

Tab.2 Results of Libpng's vulnerability reproduction

漏洞编号	工具	次数	TTE / s	F	$\hat{A}_{12}$
2011-2501	AFL-Ant	20	341	3.23	0.83
	AFLGO	20	373	2.95	0.79
	AFL	20	1102	–	–
2011-3328	AFL-Ant	20	2315	4.67	0.97
	AFLGO	20	2508	4.31	0.93
	AFL	20	10800	–	–
2015-8472	AFL-Ant	20	31	8.68	0.84
	AFLGO	20	26	10.35	0.91
	AFL	20	269	–	–
2015-8540	AFL-Ant	20	221	2.91	0.76
	AFLGO	20	201	3.20	0.74
	AFL	20	643	–	–
2018-13785	AFL-Ant	20	881	2.76	0.75
	AFLGO	20	1002	2.43	0.71
	AFL	20	2431	–	–

表 3 目标站点覆盖结果

Tab.3 Results of target site coverage

目标站点	工具	次数	TTE / s	F	$\hat{A}_{12}$
pngread.c: 730	AFL-Ant	10	23	6.50	0.92
	Ant-GO	10	40	3.75	0.89
	AFLGO	10	61	2.46	0.83
	AFL	10	150	–	–
pngtrans.c: 686	AFL-Ant	7	6101	1.42	0.78
	Ant-GO	6	6940	1.25	0.64
	AFLGO	5	7521	1.16	0.51
	AFL	3	8710	–	–
tif_read.c: 447	AFL-Ant	10	69	3.33	0.96
	Ant-GO	10	84	2.74	0.94
	AFLGO	10	103	2.23	0.91
	AFL	10	230	–	–
tif_jbig.c: 211	AFL-Ant	9	2180	2.12	0.88
	Ant-GO	7	2317	2.00	0.85
	AFLGO	5	2912	1.59	0.79
	AFL	4	4620	–	–

研究的静态分析方法和基于蚁群算法的动态能量调控方法在导向式模糊测试中有着更好的实用性和有效性.

6 结 语

提出基于动态能量调控的导向式灰盒模糊测试技术. 对目标程序 GNU Binutils、Libpng 和 Libtiff 进行实验，分析触发时间和效应值，发现本研究所提方法在漏洞复现的若干次实验中触发指定漏洞的速度以及成功触发漏洞的次数相较于当前的 DGF 技术有所提高，在目标站点覆盖实验中成功覆盖目标站点的速度与次数也有所提高. 证明所研究的技术可以实现更加快速和有效的导向性，从而有着更大的概率去发现漏洞.

所提方法存在种子的变异质量不高，对程序路径的覆盖率不高的问题. 在未来工作中，可以结合污点分析[19]或者动态符号执行[20]技术，设计更加高效的种子遗传变异策略，提升 DGF 的能力. 另外，导向式模糊测试中使用 CG 和 CFGs 来进行种子的距离计算，没有考虑到间接调用关系，导致间接调用的函数间距离计算不够准确，未来可以改进静态分析方法，识别程序中的间接调用关系来完善种子的距离计算.

参考文献 (References):

[1] 吴世忠, 郭涛, 董国伟, 等. 软件漏洞分析技术 [M]. 北京: 科学出版社, 2014.

[2] SANG K C, AVGERINOS T, REBERT A, et al. Unleashing mayhem on binary code [C]// **IEEE Symposium on Security and Privacy**. Washington, DC: Institute of Electrical and Electronics Engineers, 2012: 380–394.

[3] STEPHENS N, GROSEN J, SALLS C, et al. Driller: augmenting fuzzing through selective symbolic execution [C]// **Network and Distributed System Security Symposium.** San Diego: Internet Society, 2016: 21–24.

[4] RAWAT S, JAIN V, KUMAR A, et al. VUzzer: application-aware evolutionary fuzzing [C]// **Network and Distributed System Security Symposium.** San Diego: Internet Society, 2017: 1–16.

[5] JOHANSSON W, SVENSSON M, LARSON U E, et al. T-Fuzz: model-based fuzzing for robustness testing of telecommunication protocols [C]// **IEEE International Conference on Software Testing.** Washington: IEEE Computer Society, 2014: 323–332.

[6] BÖHME M, PHAM V T, ROYCHOUDHURY A. Coverage-based greybox fuzzing as Markov chain [C]// **IEEE Transactions on Software Engineering**. Los Alamitos: Institute of Electrical and Electronics Engineers, 2016: 1032–1043.

[7] BÖHME M, PHAM V T, NGUYEN M D, et al. Directed greybox fuzzing [C]// **Acm Sigsac Conference on Computer and Communications Security**. New York: Association for Computing Machinery, 2017: 2329–2344.

[8] ZALEWSKI M. American fuzzy lop. [EB/OL]. [2014-11-01]. http://lcamtuf.coredump.cx/afl/.

[9] MARINESCU P D, CADAR C. KATCH: high-coverage testing of software patches [C]// **Joint Meeting on Foundations of Software Engineering**. New York: Association for Computing Machiner, 2013: 235–245.

[10] GANESH V, LEEK T, RINARD M. Taint-based directed whitebox fuzzing [C]// **IEEE 31st International Conference on Software Engineering**. Vancouver: Association for Computing Machinery, 2009: 474–484.

[11] MEHLHORN K. **Data structures and algorithms: Searching and sorting** [M]. Berlin: Springer, 1984: 90.

[12] LibFuzzer: a library for coverage-guided fuzz testing [EB/OL]. [2017-05-13]. http://llvm.org/docs/LibFuzzer.html.

[13] DORIGO M, GAMBARDELLA L M. A study of some properties of Ant-Q [C]// **International Conference on Parallel Problem Solving from Nature.** Berlin: Springer, 1996: 656–665.

[14] SEREBRYANY K, BRUENING D, POTAPENKO A, et al. AddressSanitizer: a fast address sanity checker [C]// **Usenix Conference on Technical Conference.** Berkeley: USENIX Association, 2012: 28–37.

[15] PHAM V T, NG W B, RUBINOV K, et al. Hercules: reproducing crashes in real-world application binaries [C]// **Proceedings of 37th International Conference. on Software Engineering (ICSE)**. Firenze: Institute of Electrical and Electronics Engineers, 2015: 891–901.

[16] LibPNG: a library for processing PNG files. [EB/OL]. [2017-05-13]. http://www.libpng.org/pub/png/libpng.html.

[17] US National Vulnerability Database. [DB/OL]. [2017-05-13]. https://nvd.nist.gov/vuln/search.

[18] VARGHA A, DELANEY H D. A Critique and improvement of the "CL" common language effect size statistics of McGraw and Wong[J]. **Journal of Educational and Behavioral Statistics**, Thousand oaks: BLANK, 2000, 25(2): 101–132.

[19] NEWSOME J. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software [J]. **Chinese Journal of Engineering Mathematics, Xian, China: China National Publishing Industry Trading Corporation**, 2005, 29(5): 720–724.

[20] GODEFROID P, K1ARLUND N, SEN K. DART: directed automated random testing [C]// **Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI05)**. New York: Association for Computing Machinery, 2005: 213–223.