2018-Hawkeye_Towards a Desired Directed Grey-box Fuzzer

笔记本: c++
创建时间: 2019/11/4 22:07 **更新时间:** 2019/11/5 10:43
作者: Mrs.Lee

书名	Hawkeye_Towards a Desired Directed Grey-box Fuzzer	作者	Hongxu Chen
出版社		阅读日期	2019.11.4

ABSTRACT

灰盒模糊测试是一种测试实际程序的实用有效方法。但是，大多数现有的灰箱模糊器缺乏针对性，即对程序中用户指定的目标站点执行的能力。为了强调定向模糊测试中的现有挑战，我们建议Hawkeye具有定向灰箱模糊测试器的四个所需属性。由于对被测程序和目标站点进行了新颖的静态分析，Hawkeye精确地收集了信息，例如调用图，函数和到目标的基本块级距离。在进行模糊测试时，Hawkeye会根据静态信息和执行轨迹来评估运动的种子，以生成动态指标，然后将其用于种子优先级划分，功率调度和自适应变异。
这些策略可帮助Hawkeye获得更好的定向性并吸引目标站点。我们将Hawkeye用作模糊测试框架，并在不同场景下的各种实际程序中对其进行了评估。实验结果表明，Hawkeye可以到达目标位置并比最先进的灰匣子模糊器（如AFL和AFLGo）更快地重现崩溃。特别是，Hawkeye可以将某些漏洞的暴露时间从3.5小时减少到0.5小时。到目前为止，Hawkeye已在Oniguruma，MJS等项目中检测到41起先前未知的崩溃，这些项目的漏洞预测工具提供了目标站点；所有这些崩溃均已确认，并且其中15个已分配了CVE ID。

INTRODUCTION

安全测试是现代软件最有效的漏洞检测技术之一。 在安全测试技术中，模糊测试[30]或模糊测试被认为是最有效和可扩展的，它为被测程序（PUT）提供各种输入并监视异常行为（例如，堆栈或缓冲区溢出，无效 读/写，断言失败或内存泄漏）[13]。 自从该提议以来，模糊测试已经在工业界和学术界变得越来越流行，并演变成用于不同测试场景的不同类型的模糊测试器。 根据对PUT内部结构的了解，可将Fuzzer分为黑盒，白盒或灰盒[10]。 最近，灰盒模糊器已被广泛使用并被证明是有效的[7]。 具体来说，AFL [48]及其派生词[6、7、12、15、24、43]引起了很多关注。

通常，现有的灰匣子模糊器（GF）旨在在有限的时间预算内覆盖尽可能多的程序状态。但是，存在几种测试方案，其中只涉及特定的程序状态，并且需要对其进行充分的测试。例如，如果MJS [39]（用于嵌入式设备的JavaScript引擎）在MSP432 ARM平台上发现了漏洞，则其他平台的相应代码中可能会出现类似

漏洞。在这种情况下，应该指示模糊器在这些位置重现错误。另一种情况是，在修补了错误后，程序员需要检查补丁是否完全修复了该错误。这要求模糊测试者将精力集中在那些修补的代码上。在这两种情况下，都要求将模糊器控制为到达PUT中某些用户指定的位置。为了清楚起见，我们将此类位置称为目标站点。按照[6]中的定义，我们将可以完成定向模糊任务的模糊器称为定向模糊器。

作为最先进的定向灰箱模糊器（简称DGF），AFLGo [6]将目标站点的可达性作为优化问题，并采用元启发式方法来促进更短距离的测试种子。 此处，距离是根据输入种子执行轨迹上的基本块到目标基本块的平均权重计算的，其中权重由程序的调用图和控制流图中的边缘确定，并且 元启发式是模拟退火[22]。 基于这些，AFLGo解决了定向模糊的功率调度问题-应该从当前的测试种子中生成多少个新输入（在AFLGo中称为“能量”）。 总而言之，以AFLGo为代表，DGF通过结合静态分析和动态分析实现了到达目标位置的目标。

纯粹的动态执行只能基于已经覆盖的跟踪来获得反馈，而无需了解预定义的目标站点。因此，需要进行静态分析以提取必要的信息，以指导朝DGF的目标部位执行。最广泛使用的方法是计算PUT的组件（例如，基本块，功能）到目标位置的距离（或权重），以便在执行时，DGF可以判断当前种子和目标位置之间的亲和力。从执行跟踪中的组件开始。主要挑战在于，需要在不损害某些所需功能的情况下有效地计算距离。特别是，它应该有助于保留种子多样性[4]。例如，AFLGo中使用的现有种子距离计算算法始终偏向于通向目标的最短路径（请参阅第2.1节），这可能会使输入变得饥饿，而这些输入可能更容易突变以到达目标位置并进一步触发崩溃。 libFuzzer的作者[26]认为，不考虑所有可能的痕迹可能无法揭示隐藏在较长路径中的错误[37]。这得出第一期望特性P1。另一个挑战是静态分析应该以可接受的开销提供精确的信息。这是因为粗略的静态分析不会给动态模糊带来太多好处，而重量级的静态分析本身可能需要花费相当长的时间才能开始动态模糊。该挑战得出第二期望特性P2。因此，第一个问题是要进行适当的静态分析，以收集DGF的必要信息。在通过静态分析提取信息之后，动态分析面临一些挑战-如何动态调整不同的策略以尽快到达目标站点。第一个挑战是如何正确分配能量给具有不同距离的输入，以及如何确定输入的优先级，使其更接近目标。这得出第三期望特性P3。第二个挑战是如何适应性地改变突变策略，因为GF在粗粒度（例如，批量缺失）和细粒度（例如，按位翻转）水平上可能具有各种突变算子。这得出第四期望特性P4。因此，第二个问题是对DGF中使用的动态策略进行适当的调整。

P1 DGF应该具有基于距离的鲁棒性机制，可以通过考虑到目标的所有迹线并避免对某些迹线的偏向来引导定向模糊。

P2在静态分析中，DGF应该在开销和实用程序之间取得平衡。

P3 DGF应该确定种子的优先级并安排其进度，以使其迅速到达目标位置。

P4当种子涵盖不同的程序状态时，DGF应采取自适应突变策略。

在本文中，我们提出了实现DGF四个所需特性的解决方案。对于P1，我们建议应用静态分析结果来扩大相邻函数的距离（第4.2节）；并且应基于增强的相邻功能距离来计算功能级别距离和基本块级别距离，以模拟功能之间的亲和力（第4.3节）。同时，在模糊测试过程中，我们通过将静态分析结果与运行时执行信息集成在一起，计算出基本块跟踪距离以及执行跟踪与目标函数的覆盖函数相似性（第4.4节）。对于P2，我们建议应用基于调用图（CG）和控制流图（CFG）的分析，即功能级别可达性分析，功能指针的指向分析（间接调用）和基本块指标（第4.1节）。对于P3，我们建议结合基本块跟踪距离和覆盖函数相似性来解决功率调度问题（第4.4节）和种子优先级问题（第4.6节）。对于P4，我们建议根据可达性分析和覆盖的功能相似性（第4.5节）应用自适应变异策略。考虑到这些属性，我们实施了DGF Hawkeye，并使用各种实际程序进行了全面评估。实验结果表明，在到达目标位置的时间和暴露事故的时间方面，鹰眼

在大多数情况下都优于最新的灰盒模糊器。特别是，Hawkeye可以比某些先进的AFLGo更快地曝光某些碰撞，其曝光时间从3.5小时减少到0.5小时。

DESIRED PROPERTIES OF DGF(总结很全面)

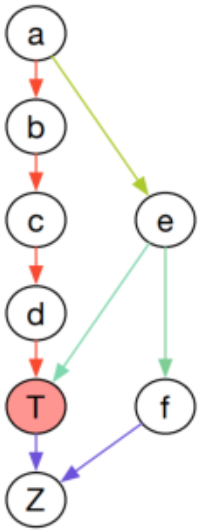

Figure 2: The fuzzing scenario modeled from Fig. 1: $\langle a, b, c, d, T, Z\rangle$ is a crashing trace passing T, $\langle a, e, T, Z\rangle$ is a normal trace passing T, and $\langle a, e, f, Z\rangle$ is a trace not passing T.

种子到目标的距离取决于执行轨迹中组件（目标块或功能）到目标的平均距离，其中组件到目标位置的距离基本上取决于目标之间的边缘数量。目标的组成部分。这种机制使AFLGo到达迹线⟨a，e，T，Z⟩的能量更多，因为它到达了目标T，并且感应的迹线距离小于⟨a，b，c，d，T，Z⟩。另一方面，对⟨a，b，c，d，T，Z less的关注较少，但是在某些情况下会导致碰撞。更糟糕的是，其他迹线（例如tracea，e，f，Z⟩）可能被错误地分配了更多的能量。结果，在我们进行的10次运行中，AFLGo也无法在24小时内重现崩溃。

现有DGF所面临的挑战来自以下几个方面：

1）目标功能可能会出现在PUT中的多个位置，并且可能会有多条不同的迹线通往目标。 2）由于调用图主要影响走线距离的计算（与目标位置的相异性），因此需要准确构建。特别是，函数之间的间接调用不应忽略。如果不能很好地解决上述两个问题，那么在这种情况下，基于距离的DGF指导机制将受到阻碍并失败。

P1 DGF应该定义一种可靠的基于距离的机制，该机制可以通过避免偏向某些迹线并考虑到目标的所有迹线来指导定向模糊测试。

P2 在静态分析中，DGF应该在开销和实用程序之间取得平衡。有效的静态分析可以从两个方面使动态模糊处理受益：1）在现实世界的C / C ++程序中，存在间接函数调用（例如，将函数指针作为C中的参数传递，或使用函数对象和成员函数指针）在C ++中）。在存在间接调用的情况下，无法直接从源代码或二进制指令中观察到调用站点。因此，需要在开销和公用事业之间进行权衡以进行分析。 2）并非所有呼叫关系都应得到平等对待。例如，某些函数在其调用函数中

多次出现，这意味着它们在运行时被调用的机会更高。从静态分析的角度来看，我们需要提供一种区分这些情况的方法。对于具有直接调用关系的功能之间的功能级别距离，很直观的是，在不同分支中多次调用的被调用方应“更接近”调用方。综上所述，以图2为例，DGF的理想设计是：1）如果函数a（间接）以间接方式调用T（即，链a→b→c→中的一个或多个调用） d→T是通过函数指针进行的），静态分析应捕获此类间接调用，否则从a到T的距离将不可用（即，视为不可到达）。 2）如果被呼叫者出现在更多不同的分支中并且在其呼叫者中出现次数更多，则应给出较小的距离，因为它可能有更多的机会被呼叫以达到目标。但是，由于静态分析的固有局限性，用静态短语对实际分支条件进行建模是不切实际的。例如，在给定非平凡的代码段的情况下，很难预测谓词的真实分支是否比运行时的错误分支执行得更多。另一方面，在灰盒模糊测试环境中，动态跟踪符号条件会花费太多时间。

P3 DGF应该选择种子并安排其播种时间，使其迅速到达目标位置。DGF应该选择种子并安排其播种时间，使其迅速到达目标位置。 AFL确定应从测试种子中生成多少新输入（即“能量”）以改善模糊测试的有效性（即增加覆盖范围）； 在[6，7]中将其称为“功率调度”。 在定向模糊测试中，模糊测试的目的不是尽可能快地达到覆盖范围的上限，而是尽可能快地达到特定的目标。 因此，DGF中的功率调度应确定应从测试种子生成多少个新输入，以便获得通向目标位点的新突变输入[6]。 同样，DGF中的种子优先顺序是确定测试种子的最佳fuzz，以尽可能快地到达目标位置。 两者均可通过基于距离的机制进行指导，该机制可测量当前种子与目标部位之间的亲和力。对于功率调度，理想的设计是，应为距离目标较小距离的种子迹线分配更多的能量以进行模糊测试，因为更靠近目标位置的迹线有更好的机会到达该位置。 因此，⟨a，e，T，Z⟩应该被分配与<a，b，c，d，T，Z⟩类似的能量<a，e，f，Z f>应该比前两个能量少。 。 先确定种子优先级，到目标的距离较小（“更近”）的种子应在随后的突变中更早地fuzz。 因此，应将⟨a，e，T，Z⟩和⟨a，b，c，d，T，Z⟩放在⟨a，e，f，Z⟩之前。

P4 当种子覆盖不同的程序状态时，DGF应该采用自适应突变策略。 GF通常应用不同的突变，例如按位翻转，字节重写，块替换，以从现有的种子生成新的测试种子。通常，这些突变体可分为两个级别：细粒突变（例如，按位翻转）和粗粒突变（例如，块替换）。尽管没有直接的证据表明细粒度的突变可能会保留执行轨迹，但广泛接受的是粗粒度随机突变很有可能极大地改变执行轨迹。因此，理想的设计是，当种子已经到达目标位点（包括目标品系，基本模块或功能）时，应给予较少机会进行粗粒突变。对于图2中的示例，考虑DGF已通过轨迹⟨a，b，c，d，T，Z⟩到达目标函数的情况，但尚未触发崩溃。现在，DGF应该为⟨a，b，c，d，T，Z⟩的输入分配较少的粗粒突变机会。同时，如果DGF刚刚启动而尚未达到⟨a，b，c，d，T，Z⟩，则DGF应该为粗粒突变提供更多机会。

for p3 AFLGo应用了基于模拟退火的功率调度器：它通过为目标分配更多的能量来偏向那些更接近目标的种子。 所应用的冷却时间表最初会根据“距离引导”的影响分配较小的权重，直到达到“利用”状态。 它解决了“探索与开发”问题[8]，并减轻了静态计算的基本块级距离带来的不精确性问题。 我们认为，这是一种有效的策略。 问题在于没有优先级排序程序，因此距离较小的新生成的种子可能需要等待很长时间才能被突变.

for p4 对于P4。 AFLGo的变异算子来自AFL的两种非确定性策略：1）破坏，它纯粹是随机的变异，例如位翻转，块替换等； 2）拼接，从两个现有种子的一些随机字节部分生成种子。值得注意的是，在运行时，AFLGo排除了所有确定性的突变过程，并且完全依赖于破坏/拼接策略的功率调度。这两种策略的随机性确实可以支持那些距目标距离较小的策略。但是，它也可能破坏接近目标的现有种子。实际上，只有通过一些特殊的先决条件才能实现一些细微的漏洞。实际上，不完整的修复程序可能仍会使某些关注案例容易受到攻击；

（1）对于P1，需要更精确的距离定义以保持迹线多样性，避免将注意力集中在短迹线上。
（2）对于P2，直接和间接呼叫都需要进行分析； 在静态距离计算期间，需要区分各种呼叫模式。
（3）对于P3，需要对当前功率调度进行调节。 还需要距离引导的种子优先级。
（4）对于P4，DGF需要一种自适应突变策略，当种子到目标之间的距离不同时，该策略可以最佳地应用细粒度和粗粒度突变。

APPROACH OVERVIEW

在本节中，我们简要介绍了我们提出的方法Hawkeye的工作流程。 图3给出了Hawkeye的概况，它由两个主要部分组成，即静态分析和模糊循环。

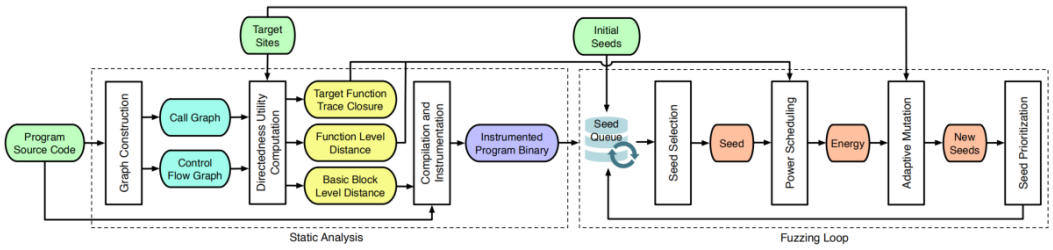

Figure 3: Approach Overview of Hawkeye

静态分析的输入是**程序源代码和目标站**点（即，模糊测试程序要到达的代码行）。 我们导出目标站点所在的基本块和功能，并将它们分别称为目标基本块和目标功能。 静态分析的主要输出是带有基本块级距离信息的已检测程序二进制文件。首先，我们基于基于包含的指针分析[3]来精确构造目标程序的调用图（CG）包含所有可能的调用。 此外，对于每个功能，我们都构造了控制流程图（CFG）（第4.1节）。 其次，我们基于CG和CFG（第4.3节）计算了一些实用程序，这些实用程序用于促进Hawkeye中的定向性。

通过函数级距离（第4.2节），基于CG计算功能级别距离。 该距离用于计算基本块级距离。 在模糊循环中也可以使用它来计算覆盖函数的相似性（第4.4节）。

基本块级距离是根据功能级距离以及CG和功能的CFG来计算的。 对于被认为能够到达目标位置之一的每个基本块，将静态测量此距离。 在模糊循环期间，它还用于计算基本块跟踪距离（第4.4节）。

根据CG为每个目标站点**计算目标功能跟踪闭合**，以获得可以到达目标站点的功能。 在模糊循环期间使用它来计算覆盖函数的相似性（第4.4节）。

Fuzzing Loop

模糊循环的输入是程序二进制文件，初始测试种子，目标位置以及功能级别距离和目标功能迹线闭合的信息。 模糊测试循环的输出是导致异常程序行为（例如崩溃或超时）的测试种子。

在模糊测试期间，模糊器从优先级种子队列中选择一个种子。模糊器对种子应用功率调度，目的是使那些被认为“更接近”目标种子的种子具有更多的突变机会，即能量（第4.4节）。具体而言，这是通过幂函数实现的，幂函数是覆盖函数相似性和基本块迹线距离的组合。对于突变期间每个新生成的测试种子，在捕获其执行跟踪之后，模糊器将根据实用程序（第3.1节）计算所覆盖的函数相似度和基本块跟踪距离。对于每个输入执行迹线，其基本块迹线距离计算为累积的基本块级距离除以已执行的基本块总数；根据当前执行功能与目标功能跟踪闭合的重叠以及功能级别距离，计算其覆盖功能相似性。确定能量后，模糊器会根

据种子上的变异者的粒度，对两种不同类别的变异自适应地分配变异预算（第4.5节）。然后，模糊器评估新生成的种子，以优先处理那些具有更多能量或已达到目标功能的种子（第4.6节）。

METHODOLOGY

为了识别调用图中的间接调用，我们建议对整个程序的函数指针应用基于包含的指针分析[3]。 该算法的核心思想是将输入程序（具有p：= q形式的语句）转换为以下形式的约束：“ q-points-set是p-points-set的子集”。本质上，这些点的传播 设置有四个规则，即地址，复制，分配，取消引用。 这种分析是上下文不敏感和流不敏感的，这意味着它既忽略了所分析函数的调用上下文，又忽略了函数内部的语句顺序，并且最终仅计算了一个包含所有程序点的指向点的解决方案。通常，在分析结束时将到达“要点集”中的一个固定点。

其中，计算了整个程序内部函数指针的指向集，从而形成了一个相对精确的调用图，其中包括所有可能的直接和间接调用。 该指针分析的复杂度为Θ（n3）。 我们之所以不应用上下文相关分析或流敏感分析，是因为它们计算量大且无法扩展到大型项目。 尽管如此，我们的调用图仍比LLVM内置API生成的调用图精确得多，后者不包含任何表示间接调用的显式节点。

每个功能的控制流程图是基于LLVM的IR生成的。 程序间流程图是通过收集整个程序的所有CFG和CG中的调用站点来构造的。 通过应用这些静态分析，我们获得了P2。

Adjacent-Function Distance Augmentation

```
void fa(int i) {
  if (i > 0) {
    fb(i);
  } else {
    fb(i*2);
    fc();
  }
}
```

(a)

```
void fa(int i) {
  if (i > 0) {
    fb(i);
    fb(i*2);
  } else {
    fc();
  }
}
```

(b)

图中afa到fb的距离应该小于到fc的距离并且小于b中fa到fc中的距离。

Cn表示一个主调函数的被调函数集合；被调用方的出现次数更多，则将有更多机会用更多不同的（实际）参数来动态执行被调用方，因此，调用方到被调用方之间的距离会更短。我们应用因子 $\Phi(C_N) = \frac{\phi \cdot C_N + 1}{\phi \cdot C_N}$ 来表示这种效应，其中ϕ是一个常数值（通常，ϕ= 2）。

包含至少一个被调函数调用点的主叫函数中基本块**CB**的数量。 这样做的理由是，随着更多具有调用点的分支，更多不同的执行将调用被调函数。我们应用因子 $\Psi(C_B) = \frac{\psi \cdot C_B + 1}{\psi \cdot C_B}$ 来表示这种效应，其中ϕ是一个常数值（通常，ψ= 2）。

给定一个（直接或间接）立即函数调用对（f1，f2），其中f1是调用方，f2是被调用方，f1和f2之间的原始距离为1（请参阅AFLGo [6]）。现在，利用上述两个指标，我们可以定义保持直接调用关系的函数对之间的增加距离。 最终的调整因子将是Φ和ψ的乘积，并且增加的相邻函数距离为：

$$d'_f(f_1, f_2) = \Psi(f_1, f_2) \cdot \Phi(f_1, f_2) \quad (1)$$

在上述示例中未示出的一种特殊情况是某些分支形成了循环（即循环）。 实际上，可以在运行时多次调用这些函数。 但是，不确定的是，如果喂入不同的种子，它们将在不同的运行中执行多少次。 幸运的是，在一个循环中被调用方的一个调用站点上的实际执行通常会产生相似的效果-循环探索相似的程序状态，并且在覆盖新路径方面的收益较少。 因此，循环内的函数调用不会带来很多执行跟踪多样性，就像在同一情况下，同一被调用者出现在参数明显不同的多个分支中的情况一样

函数级距离计算,与aflgo一样

$$d_f(n, T_f) = \begin{cases} \textbf{undefined.} & \text{if } R(n, T_f) = \emptyset \\ [\Sigma_{t_f \in R(n,T_f)} d_f(n, t_f)^{-1}]^{-1} & otherwise \end{cases} \quad (2)$$

基本块级距离，与aflgo一样

$$d_b(m, T_b) = \begin{cases} 0 & \textbf{if } m \in T_b \\ c \cdot \min_{n \in C_f^T(m)}(d_f(n, T_f)) & \textbf{if } m \in Trans_b \\ [\Sigma_{t \in Trans_b}(d_b^o(m, t) + d_b(t, T_b))^{-1}]^{-1} & otherwise \end{cases} \quad (3)$$

能量分配，在动态模糊测试期间，我们基于两个动态计算的指标对选定的种子应用功率调度：基本块跟踪距离和目标函数跟踪相似度。

Basic Block Trace Distance,与aflgo相同。

$$d_s(s, T_b) = \frac{\Sigma_{m \in \xi_b(s)} d_b(m, T_b)}{|\xi_b(s)|} \quad (4)$$

Covered Function Similarity.根据直觉来计算相似度，即在“expected traces”中覆盖更多功能的种子将有更多机会被突变以达到目标。通过跟踪当前覆盖的函数集（表示为ξf（s））并将其与目标函数迹线闭合ξf（Tf）相比较，可以计算出这种相似性 在图2的示例中，ξf（abcdTZ）= {a，b，c，d，T}，ξf（aeTZ）= {a，e，T}和ξf（ae f Z）= {a，e}。

$$c_s(s, T_f) = \frac{\Sigma_{f \in \xi_f(s) \cap \xi_f(T_f)} d_f(f, T_f)^{-1}}{|\xi_f(s) \cup \xi_f(T_f)|} \quad (5)$$

与ds相似，也应用了特征缩放归一化，最终相似度表示为c∽s。 请注意，这种相似性指标是我们方法中唯一提出的。

Scheduling。 调度处理将给给定种子多少突变机会的问题。 直觉是，如果当前种子执行的跟踪“接近”可以到达程序目标位置的任何预期跟踪，则该种子上的更多突变应更有利于生成预期种子。 纯粹基于迹线距离进行调度可能会偏向某些迹线模式。 对于AFLGo，如第2.2.2节所述，较短的路径将被分配更多的能量，这可能会使仍然到达目标位置的较长路径饿死。 为了减轻这种情况，我们提出了考虑跟踪距离（基于基本块级距离）和跟踪相似度（基于覆盖函数相似度）的幂函数：

$$p(s, T_b) = \tilde{c}_s(s, T_f) \cdot (1 - \tilde{d}_s(s, T_b)) \qquad (6)$$

Adaptive Mutation

在第4.4节中，对于每个种子，功率调度的输出是能量（也就是应用突变的时间），这将是我们自适应突变步骤的输入。 问题在于，给定种子可用的总能量，我们仍然需要为每种类型的突变子分配突变数。 通常，GF中使用两种类型的变异子。 有些是粗粒度的，因为它们会在突变期间更改大量字节。 其他的则很细粒度，因为它们仅涉及几个字节级的修改，插入或删除。 对于粗粒度突变，我们认为它们是：

（1）**Mixed havoc。** 这包括几个批量突变，即删除一个字节块，用缓冲区中的其他字节覆盖给定的块，删除某些行，多次复制某些行等。实际的突变涉及它们的组合。

（2）**Semantic mutation。** 当已知目标程序可处理语义相关的输入文件（例如javascript，xml，css等）时，使用此方法。详细地讲，这遵循Skyfire [43]，其中包括三个元突变，将另一个子树插入随机的AST位置， 删除给定的AST，并用另一个AST替换给定的位置。

（3）**Splice。** 这包括队列中的两个种子之间的交叉以及随后的混合破坏。

Algorithm 1: *adaptiveMutate*(): Adaptive Mutation

input : s, the seed to be fuzzed after power scheduling
output: $\mathcal{M}_s$, the map to store the new mutated seed, whose key is the seed and whole value is the energy of the seed
const. : γ, the constant ratio to do fine-grained mutation
const. : δ, the constant ratio to be adjusted

```
1  M_s = ∅;
2  p ← s.getScore();
3  if reachTarget(s) == false then
4      S' ← coarseMutate(s, p * (1 − γ));
5      for s' in S' do
6          M_s ← M_s ∪ {(s', s'.getScore())}
7      S'' ← fineMutate(s, p * γ);
8      for s'' in S'' do
9          M_s ← M_s ∪ {(s'', s''.getScore())}
10 else
11     S' ← coarseMutate(s, p * (1 − γ − δ));
12     for s' in S' do
13         M_s ← M_s ∪ {(s', s'.getScore())}
14     S'' ← fineMutate(s, p * (γ + δ));
15     for s'' in S'' do
16         M_s ← M_s ∪ {(s'', s''.getScore())}
```

byte-level modifications, insertions or deletions. For coarse-grained mutations, we consider them to be:

基本思想是在种子可以达到目标功能时减少粗粒突变的机会（10行）

Algorithm 2: *coarseMutate*(): Coarse-Grained Mutation

input : s, the seed to be fuzzed after power scheduling
input : i, the number of iterations to do mutation on the seed
output: $\mathcal{S}$, the set to store the *new* mutated seed
const. : σ, the constant ratio to do semantic mutations
const. : ζ, the constant ratio to do mixed havoc mutations

$\mathcal{S} = \emptyset$;
if $needSemMutation(s) == true$ **then**
  $\mathcal{S} \leftarrow \mathcal{S} \cup semMutate(s, i * \sigma)$;
  $\mathcal{S} \leftarrow \mathcal{S} \cup coarseHavoc(s, i * (1 - \sigma) * \zeta)$;
  $\mathcal{S} \leftarrow \mathcal{S} \cup splice(s, i * (1 - \sigma) * (1 - \zeta))$;
else
  $\mathcal{S} \leftarrow \mathcal{S} \cup coarseHavoc(s, i * \zeta)$;
  $\mathcal{S} \leftarrow \mathcal{S} \cup splice(s, i * (1 - \zeta))$;

Algorithm 3: *seedPrioritize*(): Seed Prioritization

input : s, the seed to be processed
output: Q_1, the tier 1 queue to store the most important seeds
output: Q_2, the tier 2 queue to store the important seeds
output: Q_3, the tier 3 queue to store the least important seeds
const. : η, the threshold of energy value for accepting important seeds

$Q_1 = Q_2 = Q_3 = \emptyset$;
if $seedIsNew(s) == true$ **then**
  if $seedWithNewEdge(s) == true$ **then**
    $Q_1 \leftarrow Q_1 \cup \{s\}$;
  else if $s.powerEnergy() > \eta$ **then**
    $Q_1 \leftarrow Q_1 \cup \{s\}$;
  else if $reachTarget(s) == true$ **then**
    $Q_1 \leftarrow Q_1 \cup \{s\}$;
  else
    $Q_2 \leftarrow Q_2 \cup \{s\}$;
else
  $Q_3 \leftarrow Q_3 \cup \{s\}$;

In practice, we assign the empirical values to the constants: γ =0.1, δ = 0.4, σ = 0.2, ζ = 0.8.

Seed Prioritization

RELATED WORK

我们的研究涉及以下研究领域：

定向灰箱模糊测试。 除了Hawkeye，还提出了其他一些DGF技术。AFLGo [6]是最先进的定向灰匣子模糊器，它利用基于模拟退火的功率调度表，

该功率表将逐渐增加的能量分配给使轨迹更靠近目标位置的输入。 在AFLGo中，作者提出了一种计算输入迹线与目标位置之间距离的新颖思路。 通过将目标距离计算与灰盒模糊测试相结合，这是一个很好的起点。 Hawkeye受到AFLGo的启发，但是在静态分析和动态模糊方面都进行了重大改进。 如§5所示，由于嵌入，Hawkeye在达到目标和重现崩溃方面通常胜过AFLGo。

定向灰箱模糊测试。除了Hawkeye，还提出了其他一些DGF技术。 AFLGo [6]是最先进的定向灰匣子模糊器，它利用基于模拟退火的功率调度表，该功率表将逐渐增加的能量分配给使轨迹更靠近目标位置的输入。在AFLGo中，作者提出了一种计算输入迹线与目标位置之间距离的新颖思路。通过将目标距离计算与灰盒模糊测试相结合，这是一个很好的起点。 Hawkeye受到AFLGo的启发，但是在静态分析和动态模糊方面都进行了重大改进。如§5所示，由于在设计中嵌入了对四个所需属性的深入考虑，在达到目标和重现崩溃方面，Hawkeye通常优于AFLGo。 SeededFuzz [45]使用各种程序分析技术来促进初始种子的生成和选择，这有助于实现定向模糊的目标。配备改进的种子选择和生成技术，SeededFuzz可以到达更多关键站点并发现更多漏洞。 SeededFuzz的核心技术与Hawkeye正交，因为SeededFuzz专注于初始种子输入的质量，而Hawkeye专注于四个理想的属性，而与初始种子无关。

定向符号执行。 定向符号执行（DSE）是与DGF最相关的技术之一，因为它还旨在执行PUT的目标位置。 已为DSE提出了几项工作[17，19，21，28，29]。 这些DSE技术依靠重量级的程序分析和约束解决方案来系统地到达目标站点。 DSE的一个典型示例是Katch [29]，它依赖于符号执行，并通过基于静态和动态程序分析的几种协同启发法得到了增强。 与手动测试套件相比，Katch可以有效地发现不完整补丁中的错误，并增加补丁覆盖范围。 但是，正如[6]中讨论的那样，由于DSE技术遭受了臭名昭著的路径爆炸问题的困扰，DGF在现实世界的程序上通常更为有效[40]。 与DSE相比，Hawkeye依赖轻量级程序分析，从而确保了其可伸缩性和执行效率。

污染分析辅助模糊测试。 污染分析也广泛用于促进定向白盒测试[12、16、24、34、44]。 在模糊测试中使用污点分析的主要直觉是确定应优先进行变异的输入的某些部分。以这样的方式，所述模糊器可以大大减少用于达到某些所需位置的搜索空间。 基于Taint的方法比DSE技术更具可伸缩性，并且可以帮助模糊器到达某些首选的位置，例如Fairfuzz [24]中的稀有分支或TaintScope [44]中与校验和相关的代码。 与Hawkeye不同，这些技术不是根据给定的目标站点（例如，我们的方案中的文件名和行号）提供的，而是基于源-接收器对的。 因此，此类技术在目标明确的情况下（例如补丁测试和崩溃再现）没有优势。

基于覆盖率的灰盒模糊测试。 基于覆盖的灰箱模糊测试（CGF）和DGF的目的是不同的。 但是，为提高CGF的性能而提出的一些技术也可以被Hawkeye采用。 例如，CollAFL [15]利用一种新颖的哈希算法来解决AFL的仪表碰撞问题。 Skyfire [43]学习概率上下文敏感语法（PGSG）来指定语法特征和语义规则，然后第二步利用学习到的PCSG生成新的测试种子。 徐等。 [46]提出了一组新的操作原语，以改善灰盒模糊器的性能。 CGF中的另一个重要主题是通过路径约束引导模糊器。 [12，25,32，34，40]旨在帮助CGF突破路径限制。 此外，Orthrus [38]对AST，CFG和CG进行了静态分析，以通过可定制的查询提取复杂的令牌。 鹰眼可以通过与上述技术结合而受益。