

Directed Greybox Fuzzing 定向灰箱模糊测试

笔记本：C++

创建时间：2019/11/4 9:55

更新时间：2019/11/4 22:02

作者：Mrs.Lee

书名	Directed Greybox Fuzzing	作者	Marcel Böhme
出版社		阅读日期	2019.11.4
摘要			
<p>在本文中，我们介绍了定向灰箱fuzz（DGF），它可以生成有效地到达给定的目标程序位置集的输入。我们开发并评估了基于模拟退火的功率计划，该计划将更多的能量逐渐分配给距离目标位置较近的种子，同时减少距离较远的种子的能量。用我们的实现AFLGo进行的实验表明，DGF的性能既有基于有符号执行的白盒模糊测试，也有无方向的灰盒模糊测试。</p>			
引言			
<p>现有的灰箱模糊器无法得到有效的定向。定向模糊器是安全研究人员组合中的重要工具。与无方向的模糊器不同，定向的模糊器将其大部分时间预算用于到达特定的目标位置，而不会浪费资源来强调不相关的程序组件。定向模糊器的典型应用可能包括：</p> <p>通过将更改的语句设置为目标来进行补丁测试[4，21]。当关键组件发生更改时，我们想检查这是否引入了任何漏洞。图1显示了介绍Heartbleed的提交[49]。专注于这些变化的模糊测试者更有可能exposing。</p> <p>通过将堆栈跟踪中的方法调用设置为目标来崩溃重现[18，29]。报告现场崩溃时，仅将堆栈跟踪和一些环境参数发送给内部开发团队。为了保护用户的隐私，通常无法使用特定的崩溃输入。定向模糊器可让内部团队迅速重现此类崩溃。</p> <p>静态分析报告验证[9]通过将语句设置为静态分析工具报告为潜在危险的目标。在图1中，工具可能会将行1480定位为潜在的缓冲区溢出。定向模糊器可以生成测试输入，以显示该漏洞是否确实存在。</p> <p>信息流检测[22]通过将敏感的源和汇设置为目标。为了揭露数据泄漏的漏洞，安全研究人员希望生成执行程序，以执行包含私人信息的敏感源和对外界可见的敏感接收器。定向模糊器可用于有效地生成此类执行。</p> <p>在本文中，我们介绍了定向灰箱模糊化（DGF），其重点是在程序中达到给定的目标位置集。在较高的级别上，我们将可及性视为优化问题，并采用特定的元启发式方法来最大程度地减少生成的种子到目标的距离。为了计算种子距</p>			

离，我们首先计算并测量每个基本块到目标的距离。尽管种子距离是过程间的，但我们的新方法仅需要对调用图进行一次分析，而对每个过程内CFG进行一次分析。在运行时，**模糊器汇总每个运动基本块的距离值，以计算种子距离作为其平均值**。DGF用于最小化种子距离的**元启发式算法称为“模拟退火”** [19]，并实现为功率调度。功率计划控制着所有种子的能量[6]。种子的能量指定了模糊种子所花费的时间。与所有灰盒模糊测试技术一样，**通过将分析移至编译时**，我们可以最大程度地减少运行时的开销。

DGF将目标位置的可达性转换为优化问题，而现有的定向（白盒）模糊处理将可达性转换为迭代约束满足问题。

对于补丁测试，我们将AFLGo与原始Katch基准上的最新技术Katch [21]定向符号执行引擎进行了比较。AFLGo发现了Katch无法发现的13个错误（七个CVE），并且AFLGo可以同时覆盖比Katch多13%的目标。但是，将这两种技术结合使用时，其目标覆盖率要比单独使用的目标高多达42%。两种定向的模糊方法都可以互补。对于崩溃重现，我们将AFLGo与原始BugRedux基准上最新的定向符号执行引擎BugRedux [18]进行了比较。当只有堆栈跟踪中的方法调用可用时，AFLGo可以产生比BugRedux多三倍的崩溃。

MOTIVATING EXAMPLE

我们使用Heartbleed漏洞作为案例研究和激励示例，以讨论两种用于定向模糊测试的不同方法。传统上，定向模糊测试基于符号执行。在这里，我们将基于符号执行引擎Klee [7]的补丁测试工具Katch [21]与本文介绍的有向灰箱模糊测试的实现AFLGo进行比较。

```
1455 + /* Read type and payload length first */
1456 + hbtype = *p++;
1457 + n2s(p, payload);
1458 + pl = p;
...
1465 + if (hbtype == TLS1_HB_REQUEST) {
1477 +     /* Enter response type, length and copy payload */
1478 +     *bp++ = TLS1_HB_RESPONSE;
1479 +     s2n(payload, bp);
1480 +     memcpy(bp, pl, payload);
```


CVE	Fuzzer	Runs	Mean TTE	Median TTE
	AFLGo	30	19m19s	17m04s
	KATCH	1	> 1 day	> 1 day

Figure 3: Time-to-Exposure (TTE), AFLGo versus KATCH.

首先，AFLGo编译OpenSSL。Clang的另一个编译器通道将经典AFL和我们的AFLGo工具添加到已编译的二进制文件中。AFL仪器向模糊器通知代码覆盖率的增加，而AFLGo仪器向模糊器通知执行的种子到给定目标集的距离。新颖的距离计算将在3.2节中讨论，并且比Katch更为复杂。它同时考虑所有目标，并在编译时完全建立，从而减少了运行时的开销。然后，AFLGo使用模拟退火对OpenSSL进行模糊处理[19]。最初，AFLGo进入探索阶段，就像AFL一样工作。

在探索阶段，AFLGo随机变异提供的种子以生成许多新输入。如果新的输入增加了代码覆盖率，则将其添加到要模糊化的种子集中；否则，将其丢弃。提供并生成的种子以连续循环进行模糊处理。例如，AFLGo从两个种子开始：s0行使图2中的分支**(b0, b')**，而s1行使**(b0, b1, b')**。假设存在从b'到t的直接路径，这是不可行的，即输入无法执行该路径。最初，两个种子将产生大致相同数量的新投入。探索的基本原理是探索其他路径，即使更长。即使s0“更接近”目标，s1的“子代”实际上更可能达到t。用户指定AFLGo进入漏洞利用的时间。对于我们的实验，我们将开发时间设置为20小时，超时设置为24小时。在开发阶段，AFLGo从更接近目标的种子中产生大量新的输入-基本上不会浪费宝贵的时间来模糊太远的种子。假设在这一点上，AFLGo生成了一个种子s2，它行使图2中的分支**(b0, b1, bi, bi + 1)**。在开发阶段，由于种子s2最接近种子s2，因此大部分时间都用于模糊测试目标t根据实现为功率计划的退火功能，AFLGo从勘探阶段缓慢过渡到开采阶段。

TECHNIQU (算法分析在笔记本

上)

我们开发了有针对性的灰箱模糊测试 (DGF)，这是一种漏洞检测技术，专注于到达用户定义的目标位置。DGF保留了灰箱模糊测试的效率，因为它在运行时不会进行任何程序分析，因为所有程序分析都是在编译时进行的。DGF易于并行化，因此可以在需要时分配更多的计算能力。DGF允许指定多个目标位置。我们定义了一种跨过程的距离度量（即种子到目标位置），该度量在检测时已完全确定并且可以在运行时有效地计算出来。尽管我们的度量是过程间的，但我们的程序分析实际上是基于调用图 (CG) 和过程内控制流图 (CFG) 进行的过程内分析。我们展示了与过程间分析相比，这如何产生二次节省。CG和CFG在LLVM编译器基础结构中很容易获得。使用这种新颖的距离量度，我们定义了一种新颖的功率计划[6]，该计划整合了最流行的退火功能，即指数冷却计划。根据我们的距离测度，基于退火的功率计划逐渐将更多的能量分配给更接近目标位置的种子，同时减少更远的种子的能量。

算法1显示了CGF如何工作的算法示意图。模糊器具有一组种子输入S，并以连续循环的方式从S中选择输入s，直到达到超时或中止模糊为止。该选择在choiceNext中实现。例如，AFL本质上按照添加顺序从循环队列中选择种子。对于选定的种子输入s，CGF确定在分配能量（第3行）中实现的通过模糊s生成的输入数量p。这也是实现 (annealing-based) 能量分配的地方。然后，模糊器根据mutate_input中实现的定义的变异算子，通过随机变异s来生成p个新输入（第5行）。AFL使用位翻转，简单的算法，边界值以及块删除和插入策略来生成新输入。如果生成的输入s'覆盖新分支，则将其添加到循环队列中（第9行）。如果生成的输入s'使程序崩溃，则将其添加到崩溃输入的集合SX中（第7行）。同样有趣的崩溃输入被标记为唯一崩溃。

Algorithm 1 Greybox Fuzzing

Input: Seed Inputs S

```
1: repeat
2:    $s = \text{CHOOSENEXT}(S)$ 
3:    $p = \text{ASSIGNENERGY}(s)$  // Our Modifications
4:   for  $i$  from 1 to  $p$  do
5:      $s' = \text{MUTATE\_INPUT}(s)$ 
6:     if  $t'$  crashes then
7:       add  $s'$  to  $S_x$ 
8:     else if  $\text{ISINTERESTING}(s')$  then
9:       add  $s'$  to  $S$ 
10:    end if
11:  end for
12: until timeout reached or abort-signal
Output: Crashing Inputs  $S_x$ 
```

Böhme等。[6]表明基于覆盖率的灰箱模糊可以建模为马尔可夫链。状态 i 是程序中的特定路径。从状态 i 到状态 j 的过渡概率 p_{ij} 由对执行路径 i 的种子进行模糊处理生成执行路径 j 的种子的概率给出。作者发现，CGF行使某些（高频）路径的频率明显高于其他路径。静态分布的密度正式描述了经过一定数量的迭代后，模糊器行使特定路径的可能性。Böhme等。开发了一种技术，通过根据AFLFast，实现的邻域密度来调整种子产生的模糊数量，从而将模糊器引向低频路径。种子 s 产生的模糊的数量也称为 s 的能量。种子的能量 s 由所谓的功率计划控制。注意，能量是在马尔可夫链中的状态局部的属性，这与在模拟退火中全局的温度不同。

种子输入与多个目标位置之间距离的度量

为了计算跨函数的距离，我们在函数级别的调用图（CG）和基本块级别的过程内控制流图（CFG）中为每个节点分配一个值。可以从给定的源代码参考（例如，d1_both.c: 1480）中迅速地识别目标功能 T_f 和目标基本块 T_b 。根据算术平均值与谐波平均值定义的节点距离之间的差异。节点距离显示在白色圆圈中。目标标记为灰色

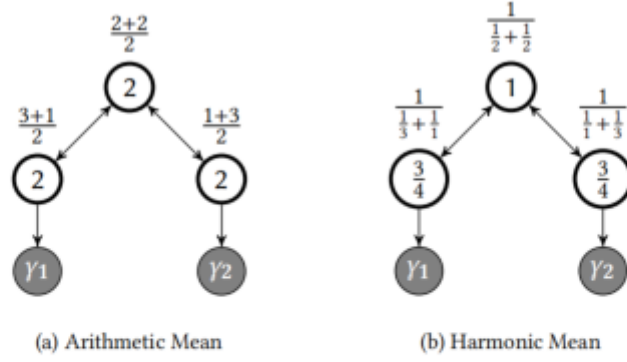


Figure 4: Difference between node distance defined in terms of arithmetic mean versus harmonic mean. Node distance is shown in the white circles. The targets are marked in gray.

function级别的目标距离确定了调用图中从一个功能到所有目标功能的距离，而function距离确定了调用图中的任何两个功能之间的距离。更正式地说，我们将函数距离 $df(n, n')$ 定义为调用图CG中沿着函数 n 和 n' 之间最短路径的边数。我们将函数 n 与目标函数 T_f 之间的函数级目标距离 $df(n, T_f)$ 定义为 n 与任何可达目标函数 $t_f \in T_f$ 之间的函数距离的谐波均值：

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

其中 $R(n, T_f)$ 是CG中从 n 可以到达的所有目标函数的集合。**谐波均值可以区分距离一个目标较近且距离另一个目标较远的节点与距离两个目标均等距的节点。**相反，算术平均值将为两个节点分配相同的目标距离。图4提供了一个示例。

基本块级目标距离决定了从基本块到调用函数的所有其他基本块的距离，以及被调用函数的功能级目标距离的倍数。直观地讲，我们根据基本块到调用链中某个函数的其他基本块的平均距离向目标位置分配目标距离，此外，如果该调用链越短，分配的目标距离就越小。BB距离确定CFG中任意两个基本块之间的距离。更正式地说，我们将BB距离 $db(m_1, m_2)$ 定义为函数 i 的控制流图 G_i 中沿着基本块 m_1 和 m_2 之间最短路径的边数。令 $N(m)$ 为基本block m 调用的函数集，使得 $\forall n \in N(m). R(n, T_f) \neq \emptyset$ 。令 T 为 G_i 中的基本块的集合，使得 $\forall m \in T. N(m) \neq \emptyset$ 。我们定义基本块 m 与目标基本块之间的基本块级目标距离 $db(m, T_b)$ ：

$$d_b(m, T_b) = \begin{cases} 0 & \text{if } m \in T_b \\ c \cdot \min_{n \in N(m)} (d_f(n, T_f)) & \text{if } m \in T \\ \left[\sum_{t \in T} (d_b(m, t) + d_b(t, T_b))^{-1} \right]^{-1} & \text{otherwise} \end{cases} \quad (2)$$

最后，我们具有定义标准化种子距离的所有要素，即种子 s 与目标位置集合 T_b 的距离。令 $\xi(s)$ 为种子 s 的执行轨迹。此跟踪包含执行的基本块。我们定义种子距离 $d(s, T_b)$ 为

$$d(s, T_b) = \frac{\sum_{m \in \xi(s)} d_b(m, T_b)}{|\xi(s)|} \quad (3)$$

对种子的距离进行归一化。

$$\tilde{d}(s, T_b) = \frac{d(s, T_b) - \min D}{\max D - \min D} \quad (4)$$

where

$$\min D = \min_{s' \in S} [d(s', T_b)] \quad (5)$$

$$\max D = \max_{s' \in S} [d(s', T_b)] \quad (6)$$

值得注意的是，在我们的实验中，将标准化种子距离定义为“标准化”基本块级目标距离（wrt.最小和最大目标距离）的算术平均值，导致概率密度集中在显着正峰度的值远小于0.5。这导致每个种子的能量大大减少。等式中标准化种子距离的定义（4）减少峰度并很好地在零和一之间分布。

注意，归一化种子距离 $\sim d \in [0, 1]$ 。还要注意，可以将距离计算的重量级程序分析转移到检测时，以使运行时的性能开销降至最低。首先，提取调用图和过程内控制流图。这可以使用编译器本身(<https://llvm.org/docs/Passes.html#dot-callgraph-print-call-graph-to-dot-file>)来实现，也可以使用位代码转换（或提升）（For instance, using mcsema: <https://github.com/trailofbits/mcsema>）来仅使用二进制文件来实现。给定目标位置，可以在检测时计算功能级别和基本块级别的目标距离。通过收集这些预先计算的距离值，在运行时仅计算归一化的种子距离。

基于退火算法的能量分配方案

我们开发了一种新颖的基于退火算法的能量分配方案（APS）。Böhme等。[6]显示灰盒模糊可以看作是可以使用功率调度表有效导航的马尔可夫链。12这为我们提供了使用马尔可夫链蒙特卡罗（MCMC）优化技术（例如模拟退火）的机会。我们基于退火的功率计划为“更接近”目标的种子分配了比“更远”的种子更多的能量，并且该能量差随温度降低（即随着时间的流逝）而增加。

Simulated Annealing(SA)[19]，SA算法渐渐收敛到全局最优解集。在我们的案例中，这组种子是行使最大目标位置数量的一组种子。SA是一种马尔可夫链蒙特卡罗方法（MCMC），用于在可接受的时间预算内在非常大的，通常是离散的搜索空间中近似全局最优值。SA的主要特征是，在随机游走期间，它始终接受更好的解决方案，但有时也可能接受更差的解决方案。

*temperature*是SA算法的参数，可调节较差情况的接受程度，并根据冷却时间表降低。开始时，当 $T = T_0 = 1$ 时，SA算法很可能接受较差的解决方案。最后，当 T 接近0时，它会退化为经典的梯度下降算法，并且只会接受更好的解决方案。

冷却时间表控制收敛速度，并且取决于初始温度 $T_0 = 1$ 和温度循环 $k \in \mathbb{N}$ 。请注意，尽管能量是种子的局部能量，但温度是所有种子的整体能量。最受欢迎的是指数冷却时间表[19]：

scneauue [19]:

$$T_{\text{exp}} = T_0 \cdot \alpha^k \quad (7)$$

where $\alpha < 1$ is a constant and typically $0.8 \leq \alpha \leq 0.99$.

基于退火算法的能量分配方案在自动漏洞检测中，我们通常只有有限的时间预算。因此，我们希望指定一个时间 t_x ，当退火过程经过足够的“探索”时间后，退火过程应进入“探索”状态。

直观地讲，在时间 t_x 处，模拟的退火过程与经典的梯度下降算法（又称贪婪搜索）相当。当 $T_k \leq 0.05$ 时，让冷却计划进入开发阶段。调整0.05以外的值以及不同的冷却时间表非常简单。因此，我们在时间 t 处计算温度 T_{exp} 如下。

$$0.05 = \alpha^{k_x} \quad \text{for } T_{exp} = 0.05; k = k_x \text{ in Eq. (7)} \quad (8)$$

$$k_x = \log(0.05) / \log(\alpha) \quad \text{solving for } k_x \text{ in Eq. (8)} \quad (9)$$

$$T_{exp} = \alpha^{\frac{t}{t_x} \frac{\log(0.05)}{\log(\alpha)}} \quad \text{for } k = \frac{t}{t_x} k_x \text{ in Eq. (7)} \quad (10)$$

$$= 20^{-\frac{t}{t_x}} \quad \text{simplifying Eq. (10)} \quad (11)$$

+

接下来，我们使用指数冷却时间表定义基于退火的功率时间表（APS）。给定种子 s 和目标位置 T_b ，APS将能量 p 分配为

$$p(s, T_b) = (1 - \tilde{d}(s, T_b)) \cdot (1 - T_{exp}) + 0.5T_{exp} \quad (12)$$

在图5中分别针对当前时间 t 和归一化种子距离 d 的三个值说明了APS的行为。注意能量 $p \in [0, 1]$ 。此外，当开始搜索（ $t = 0$ ）时，APS将相同能量分配给具有高种子距离的种子与具有低种子距离的种子。随着时间的流逝，仅锻炼目标（即， $d = 0$ ）的种子被分配越来越多的能量。

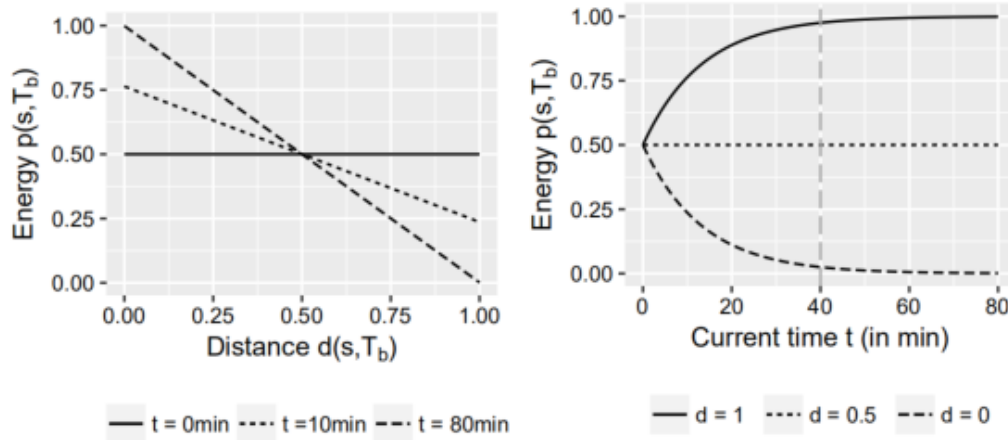
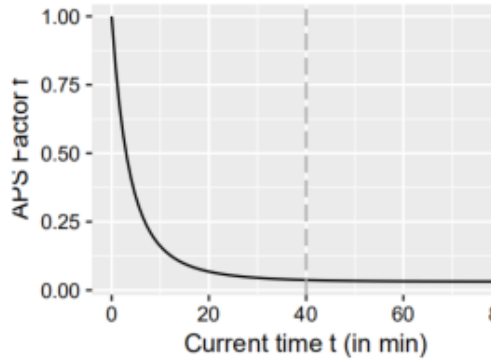


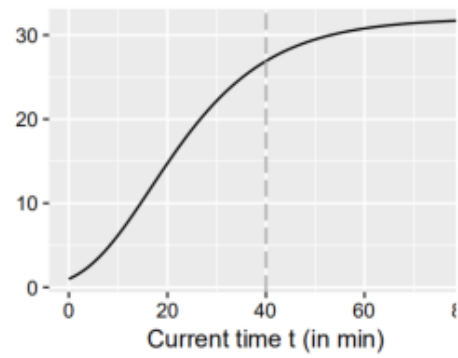
Figure 5: Impact of seed distance $\tilde{d}(s, T_b)$ and current time on the energy $p(s, T_b)$ of the seed s (for $t_x = 40$ min).

现在，AFL已经实施了功率计划表。13那么，我们如何集成APS？现有时间表根据 s 的执行时间和输入大小，何时发现 s 以及有多少祖先来分配能量。我们希望将AFL的现有电力计划与基于退火的电力计划进行整合，并定义最终的基于退火的综合电力计划。令 $p_{afl}(s)$ 是AFL通常分配给种子 s 的能量。给定基本块 T_b 作为目标，我们计算出种子 s 的积分APS $\hat{p}(s, T_b)$ 为

$$\hat{p}(s, T_b) = p_{\text{aff}}(s) \cdot 2^{10 \cdot p(s, T_b) - 5} \quad (13)$$



(a) Distance $\tilde{d}(s, T_b) = 1$



(b) Distance $\tilde{d}(s, T_b) = 0$

定向灰箱模糊化的可扩展性

EVALUATION SETUP

AFLGo由四个组件实现，分别是图形提取器，距离计算器，仪表和模糊器。通过与OSS-Fuzz的集成，我们证明了这些组件可以无缝整合到原始构建环境（例如make或ninja）中。总体架构如图7所示。在下文中，我们解释了如何实现这些组件。

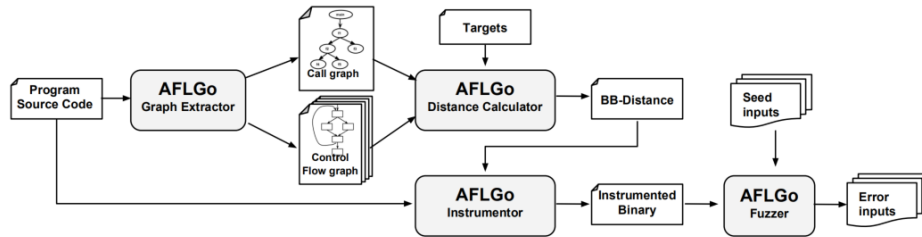


Figure 7: Architecture: After the Graph Extractor generates the call and control-flow graphs from the source code, the distance calculator computes the basic-block-level target distance for each basic block which is used by the Instrumentor during instrumentation. The instrumented binary informs the Fuzzer not only about coverage but also about the seed distance.

(1) AFLGo图形提取器（GE）生成调用图（CG）和相关的控制流图（CFG）。CG节点由功能签名标识，而CFG节点由源文件和相应基本块的第一条语句的行标识。GE被实现为AFL LLVM通过的扩展，由编译器afl-clang-fast激活。将编译器环境变量CC设置为afl-clang-fast并构建项目。

(2) AFLGo距离计算器（DC）会根据第3.2节获取调用图和每个过程内控制流，以计算每个基本块（BB）的过程间距离。DC被实现为Python脚本，该脚本使用networkx包来解析图形并根据Dijkstra的算法进行最短距离计算。DC生成BB距离文件，其中包含每个BB的基本块级目标距离。

(3) AFLGo将获取BB距离文件，并以目标二进制文件中的每个BB进行执行。具体来说，对于每个BB，它确定各自的BB级目标距离并注入扩展的蹦床。

蹦床是一段汇编代码，在每个跳转指令之后执行，以跟踪所覆盖的控制流边缘。边缘由共享的64kb内存中的一个字节标识。在64位体系结构上，我们的扩展使用了共享内存的16个附加字节：8个字节用于累加距离值，而8个字节用于记录执行的BB数。对于每个BB，AFLGo仪表会添加汇编代码i) 来加载当前BB的距离，并添加当前BB的目标距离，ii) 加载并增加已执行BB的数量，并且iii) 将这两个值存储到共享内存中。该工具被实现为AFL LLVM通过的扩展。编译器设置为afl-clang-fast，编译器标志引用BB-distance文件，并且该项目使用ASAN构建[35]。

(4) AFLGo Fuzzer已在AFL版本2.40b中实现（该版本已集成了AFLFast的探索时间表[6]）。它根据我们基于退火的功率调度表对检测的二进制文件进行模糊处理（请参见第3.3节）。共享内存中的其他16个字节将通知Fuzzer当前种子距离。当前种子距离是通过将累积的BB距离除以运动的BB数来计算的。

RELATED WORK

我们从对定向模糊和典型应用的现有方法的调查开始。接下来是对基于覆盖率的模糊测试，其目的是生成可以实现最大代码覆盖率的输入。最后，我们讨论了基于污点的定向模糊器，其目的是识别和模糊种子中的特定输入字节，以在给定位程序位置获得特定值。

据我们所知，定向模糊测试主要是在符号执行引擎中实现的，例如Klee [7]。定向符号执行（Directed Symbolic Execution, DSE）利用符号执行，程序分析和约束求解的重型机制来系统有效地探索可行路径的状态空间[20]。一旦确定了可以实际到达目标的可行路径，就将见证测试用例生成为后验，作为相应路径约束的解决方案。DSE已被用于到达危险程序位置，例如关键系统调用[15]，以覆盖补丁程序中的更改[3、21、34]，到达以前未发现的程序元素以增加覆盖率[66]，以验证静态分析报告[9]，用于突变测试[16]，以及内部重现现场故障[18、33]。与DSE相比，定向灰箱模糊测试（DGF）不需要繁琐的符号执行，程序分析和约束解决方案。DGF确实执行的轻量级程序分析在编译时完全进行。我们的实验表明DGF可以胜过DWF，并且两种技术在一起的效果甚至比每种技术都更有效。

Coverage-based fuzzing，旨在提高种子语料的代码覆盖。希望不执行程序要素e的种子语料库也将无法发现e中可观察到的漏洞。覆盖率指示的灰箱模糊器[6、31、36、43、53]使用轻量级工具在运行期间收集覆盖率信息。有几种增强技术。AFLFast [6]论点集中在低频路径中的模糊运动执行每单位时间更多的路径。Vuzzer [31]将权重分配给某些基本块，例如错误处理代码，以优先处理更可能揭示漏洞的路径。Sparks等。[36]使用遗传算法来发展一个固定大小的种子语料库和一个输入语法，以更深入地渗透到控制流逻辑中。基于覆盖率的白盒模糊测试器[7、12、13]使用符号执行来增加覆盖率。例如，Klee [7]有一个搜索策略来优先处理更接近未发现的基本块的路径。还研究了两种方法的组合和集成[26、38]。与有针对性的灰箱模糊测试相反，基于覆盖率的模糊器将所有程序元素都视为目标，以实现最大的代码覆盖率。但是，如果目标实际上仅是到达一组特定的目标位置，则不相关的位置会浪费资源。

基于污点的定向白盒模糊测试利用经典的污点分析[17、25]来识别种子输入的某些部分，这些部分应该优先进行模糊处理，以增加在目标位置观察漏洞所需的值的概率（例如，，除法运算符的分母中的零值）[11、31、40]。这样可以大大减少搜索空间。例如，Buzzfuzz [11]将种子文件的某些部分标记为可模糊的，从而控制所有已执行和关键系统调用的参数。

种子文件的大部分不需要进行模糊处理。基于覆盖率的模糊器Vuzzer [31]使用污点来练习原本很难触及的代码。识别了相关的条件语句后，Vuzzer使用污

点以更高的可能性实现不同的分支结果。与DSE不同，基于污点的定向白盒模糊测试不需要繁琐的符号执行和约束解决机制。但是，它要求用户提供已经可以到达目标位置的种子输入。相反，正如我们的实验所证明的那样，AFLGo甚至可以从空种子输入开始。在以后的工作中，研究DGF如何从Vuzzer中实现的类似基于tain的方法中受益是很有趣的：在运行时，分析将首先确定那些需要被消除以减少距离的条件语句，然后使用从而增加了在模糊测试期间实际上否定这些陈述的可能性。但是，我们的直觉是，灰盒模糊测试成为漏洞检测的最先进技术的主要因素是其效率。每秒生成和执行数千个输入的能力。秉承这一理念，我们将DGF设计为在编译时进行所有重量级分析，同时在运行时保持其效率。

CONCLUSION

像AFL这样基于覆盖率的灰盒模糊器会尝试覆盖更多程序路径，而不会引起程序分析的任何开销。像Klee或Katch这样的基于符号执行的白盒模糊测试工具，会使用符号程序分析和约束求解来在需要时准确地指导测试生成中的搜索。

符号执行一直是实现定向模糊器的首选技术[3、4、9、15、20、21、27、29、33、34、66]。达到给定目标仅是解决正确路径约束的问题。符号执行提供了一种分析，数学上严格的框架，可以专门探索到达目标位置的路径。相比之下，灰盒模糊测试本质上是一种随机方法，并且不支持开箱即用的定向性。从根本上讲，任何灰盒模糊测试器仅将随机突变应用于随机种子文件中的随机位置。

在本文中，我们试图将这种定向性引入灰盒模糊测试。为了在运行时保持灰箱模糊测试的效率，我们将大多数（轻量级）程序分析移至检测时间，并在测试生成过程中将模拟退火作为实用的全局元启发式方法。与定向符号执行不同，定向灰盒模糊测试不会由于重量级的程序分析或对执行指令的编码和求解作为路径约束而引起任何运行时性能开销。

定向灰盒模糊测试可以以多种方式使用：用于将搜索定向到有问题的更改或补丁，关键系统调用或危险位置，或指向我们希望重现的已报告漏洞的堆栈跟踪中的函数。我们展示了有针对性的灰盒模糊测试在补丁程序测试（需要到达补丁代码中的位置）和崩溃故障现场重现（其中需要重现堆栈跟踪）的应用。我们还将讨论将定向模糊器集成到连续模糊平台OSS-Fuzz中的问题。在将来的工作中，我们计划将基于符号执行的定向白盒模糊测试和定向灰盒模糊测试集成在一起。结合符号执行和搜索作为优化问题的集成定向模糊技术，将能够利用它们的综合优势来减轻其各自的弱点。正如我们的实验所证明的那样，这将导致更有效的补丁程序测试，以消除潜在的易受攻击的程序更改。我们还计划与静态分析工具集成以评估AFLGo的有效性，该工具可指出危险位置或对安全至关重要的组件。这将使我们能够将模糊测试的工作集中在更有可能包含漏洞的极端情况下。为了下载我们的工具AFLGo以及我们与OSS-Fuzz的集成，读者可以执行以下命令