

Directed Greybox Fuzzing

AFLGo

Marcel Böhme

National University of Singapore, Singapore
marcel.boehme@acm.org

Manh-Dung Nguyen

National University of Singapore, Singapore
dungnguy@comp.nus.edu.sg

Van-Thuan Pham*

National University of Singapore, Singapore
thuampv@comp.nus.edu.sg

Abhik Roychoudhury

National University of Singapore, Singapore
abhik@comp.nus.edu.sg

CCS2017

Greybox fuzzers

like **AFL** : cover more program paths

- use lightweight instrumentation to determine;
- a unique identifier for the exercised path;
- mutate a provided seed input;
- discover hundreds of high-impact vulnerabilities;
- so many security researchers involved in extending it.

However, existing greybox fuzzers cannot be effectively directed.

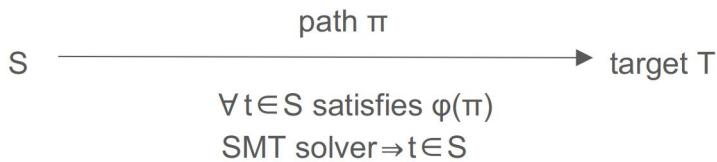
灰盒fuzz现在被认为是漏洞挖掘非常先进的一个手段，典型的工具如AFL，插装代码量非常小，具有对input跑过的分支进行记录的功能。比如用hash去表示分支，用bitmap去记录跑过的次数，把种子mutate，产生新的种子，如果执行了新的分支，这个input就加入测试队列，AFL发现很多高影响漏洞，并且还有很多安全研究人员在AFL基础上不断扩展。包括我们小组现在在做得enforce-afl。然而，目前现存的greybox fuzzer都不能有效定向，即不能控制产生的输入去执行指定的目标区域。

Directed fuzzers

Most existing directed fuzzers are based on symbolic execution

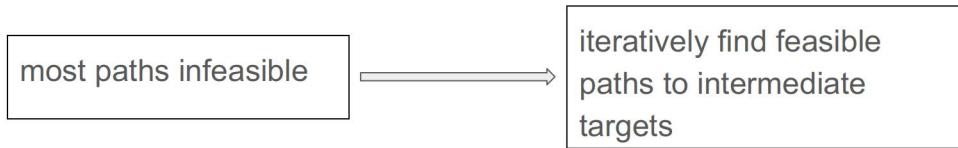
- systematic path exploration

e.g.



这篇文章作者认为现在大部分定向**fuzzers**基于符号执行，传统符号执行在不执行程序的前提下，通过模拟执行来进行程序分析得到相应的路径约束，然后通过约束求解器来得到可以触发目标代码的具体值。假设有一个路径 π 能到达目标位置， $\varphi(\pi)$ 是一个一阶逻辑公式，并且所有能执行目标T的输入都满足 $\varphi(\pi)$ ，那么根据布尔表达式满足理论SMT求解器就可以得到这样的一个具体t。（其实符号执行这块我也没具体去研究过）

Directed symbolic execution



KTACH uses the symbolic execution engine KLEE.

e.g.

```
φ(π₀) 1455 /* Read type and payload length first */  
1456 hbtpe = *p++;  
1457 n2s(p, payload);  
1458 p1 = p;  
...  
1465 + if (hbtpe == TLS1_HB_REQUEST) { intermediate target  
1477 + /* Enter response type, length and copy payload */  
1478 + *bp++ = TLS1_HB_RESPONSE;  
1479 + s2n(payload, bp);  
1480 + memcpy(bp, p1, payload); target
```

KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs, [OSDI 2008](#)

定向符号执行将target code可达到问题归结为反复求解约束满足问题。其实大部分路径是不直接到达，所以只好去找一个目标作为中介。上图就是有heartbleed漏洞的代码片段。文章距离用KATCH去测试，首先存π₀这样的路径到达第1465行，但是目标是第1480行。所以KATCH就要把 $\varphi(\pi_0) \wedge (\text{hbtpe} == \text{TLS1_HB_REQUEST})$ 传给约束求解器生成这样的一个input能执行target location第1480行。

Directed symbolic execution

KTACH uses the symbolic execution engine KLEE.

e.g.

```
φ(π0) |  
1455 + /* Read type and payload length first */  
1456 + hbtpe = *p++;  
1457 + n2s(p, payload);  
1458 + p1 = p;  
...  
1465 + if (hbtpe == TLS1_HB_REQUEST) { intermediate target  
1477 +     /* Enter response type, length and copy payload */  
1478 +     *bp++ = TLS1_HB_RESPONSE;  
1479 +     s2n(payload, bp);  
1480 +     memcpy(bp, p1, payload); target
```

$\phi(\pi_0) \wedge (\text{hbtpe} == \text{TLS1_HB_REQUEST})$

cost of time

- heavy-weight program analysis
- constraint solving

AFLGo	< 20 min
KATCH	> 24 hr

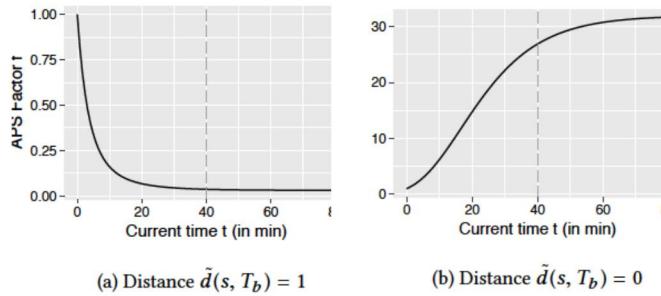
尽管这样效果非常的好，但是牺牲了效率，花费了太多的时间进行程序分析，约束求解。AFLGo不到20分钟就可以发现heartbleed，而KATCH 24小时内都发现不了

In brief, What they do?

Directed Greybox Fuzzing==DGF

- bring directedness to greybox fuzzing;
- use **simulated annealing**-based power schedule.

AFLGo



ok, 那么回归主题，这篇文章他们在AFL上究竟做了什么？DGF，在灰盒fuzzer中加入了定向性，在插桩的时候进行程序分析，维持了的灰盒fuzz的运行时高效性；同时利用了模拟退火作为一个全局的元启发策略，为离目标位置更近的种子分配更多的能量，同时如果种子离目标距离比较远，而且这种分配差异会随时间变大。和所有灰盒fuzz一样，程序分析是在编译时进行的。

MOTIVATING EXAMPLE

The Heartbleed bug CVE-2014-0160

was introduced on New Year's Eve 2011 by commit 4817504d which implemented a new feature called Heartbeat.

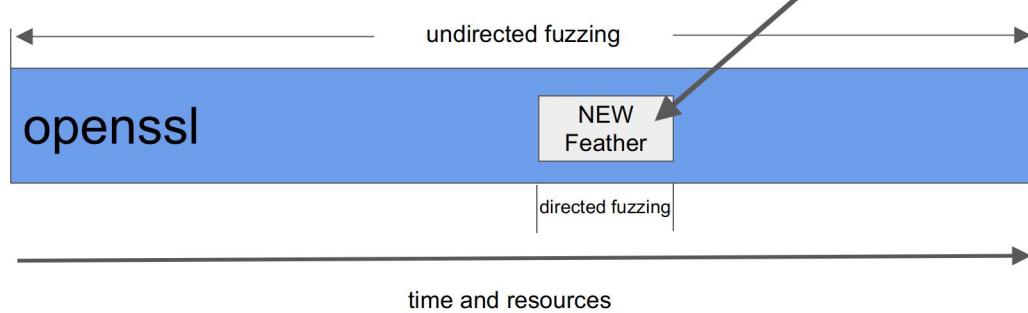


```
1455 + /* Read type and payload length first */
1456 + hbtpe = *p++;
1457 + n2s(p, payload);
1458 + p1 = p;
...
1465 + if (hbtpe == TLS1_HB_REQUEST) {
1477 +     /* Enter response type, length and copy payload */
1478 +     *bp++ = TLS1_HB_RESPONSE;
1479 +     s2n(payload, bp);
1480 +     memcpy(bp, p1, payload);
```

简单说一下heartbleed这个漏洞，首先我们知道心跳包用来检测对端是否还存活，防止大量已死的对端还在占用链接。这个漏洞是由于2011新年前夕commit 4817504d引进了一个新的特征，也就是心跳包所导致的。Hbyte是心跳包的类型，如果消息类型满足1465行，那么执行性1480行，将payload长度的数据拷贝到bp中。这个payload是长度可以是虚假的，比如我实际心跳包数据只有一个字节，我写一个65535的数字到payload中，OpenSSL这里是完全不检查实际长度的。那么将近64kb的内存会被泄漏出来，而且这64kb是p1的，在SSLv3记录的附近，可以使一些cookie，甚至username, password数据。

MOTIVATING EXAMPLE

The Heartbleed bug CVE-2014-0160



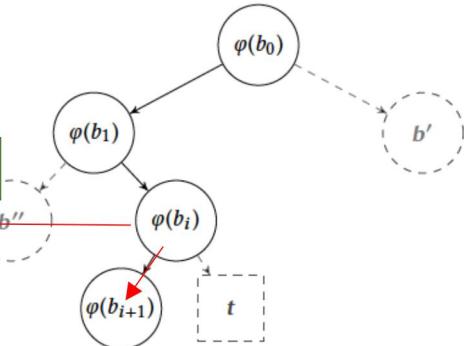
我们知道定向的fuzzer可以将最新版本新增或变化的部分作为测试目标，如果新添加的这部分有漏洞，那么定向fuzzer是可以检测出来的，像OpenSSL这样包含数百万行代码，用非定向的fuzzer去测，那肯定会浪费大量时间，相比定向的效率更高，因为它可以只测那些最新500多行代码部分。

MOTIVATING EXAMPLE

For each target t, KATCH executes the greedy search!

```
1455 /* Read type and payload length first */  
1456 hbtpe = *p++;  
1457 n2s(p, payload);  
1458 pl = p;  
...  
1465 if (hbtpe == 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);
```

hbtpe!=TLS1_HB_REQUEST



KATCH把openssl要编译成LLVM字节码。在测试过程中，KATCH对11个改变的基本块作为目标进行测试，这11个目标在之前的回归测试中没被测试过。对于每个目标，KATCH执行贪婪式搜索。KATCH会在之前回归测试中找到一个比较接近目标的种子，例如上图种子执行到 $\varphi(b_i)$ ，对应源码1465行，但是消息类型不正确，只能执行 b_{i+1} ，却不能执行1480行，却离t已经很近了，所以需要把判断条件否定。

MOTIVATING EXAMPLE

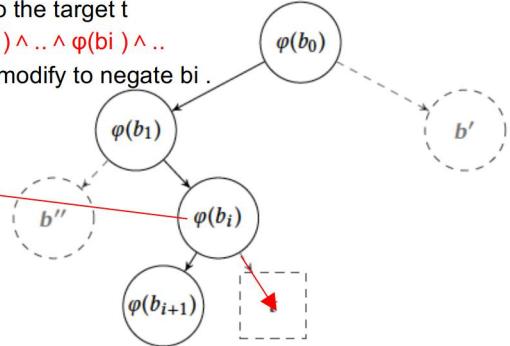
- to identify the executed branch b_i that is closest to the target t
- to construct a path constraint $\Pi(s) = \varphi(b_0) \wedge \varphi(b_1) \wedge \dots \wedge \varphi(b_i) \wedge \dots$
- to identify the specific input bytes in s it needs to modify to negate $\varphi(b_i)$.

PASS

$$\Pi' = \varphi(b_0) \wedge \varphi(b_1) \wedge \dots \wedge \boxed{\neg \varphi(b_i)}$$

to

Z3 SMT solver



这时KATCH会对程序进行分析，结合 s 所跑过的所有分支条件条件构建一个路径约束，识别出 s 中应该修改的字节能否定条件 $\varphi(b_i)$ 。这个新的约束路径是传给求解器（满足膜理论）然后计算出 s 中这个特殊字节需要修改的值。

Challenges

Due to the heavy-weight program analysis, KATCH takes a long time to generate an input.

- Distance is re-computed at runtime for every new path that is explored;
- The search might be incomplete since the interpreter might not support every bytecode;
- The constraint solver might not support every language feature, such as floating point arithmetic.

尽管这种DSE定向符号执行效果很好，但是代价很高，由于需要这种大量的程序分析，所以KATCH需要很长的时间才能产生一个输入。KATCH检查heartbleed时间超过一天。还有一些其他问题，比如在运行过程中探索新路径时，距离是要重新计算的。而且搜索也不是完整的，解释器也许不支持所有字节码。而且约束求解器也不是支持所有语言特征，比如浮点运算。

Opportunities

AFLGo generates and executes several thousand inputs per second.

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

- No program analysis at runtime and only light-weight program analysis at compile/instrumentation time;
- A global search based on **Simulated Annealing** allows AFLGo to approach a **global optimum**;
- AFLGo implements a parallel search.



KATCH
greedy search

再看看，AFLGo每秒可以产生上千input，程序分析是在编译/插装时候进行的，不影响运行时效率，而且它使用基于模拟退火的全局搜索达到一个全局最优，这里血药注意KATCH是greedy search是局部最优，可能会在某一个位置卡住，不能到达目标位置。此外，AFLGo还支持并行搜索，同时搜索所有目标。这里说明一下AFLGo实验本身是具有随机性的，所以需要考虑一些统计因素，因而他们做了30此实验，之后求了个平均值，可以看到AFLGo平均19分钟跑出结果，KATCH需要一天多时间。

How a coverage-based greybox fuzzer(CGF)works?

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
```

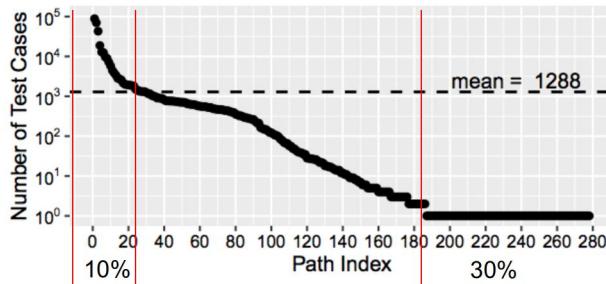
Output: Crashing Inputs S_x

AFLgo是在AFL基础上进行改进的，AFL是一个coverage-based的灰盒fuzzer，这是coverage-based greybox fuzzer工作流程图，将输入集合排成队列，例如AFL按照input加载的顺序排成一个循环队列，在第2步chooseNext实现了按顺序选取input喂给程序。AFLGo在第3行做了修改，使用模拟退火方式为种子分配能量值p，然后把s mutate p次，产生p个新输入s'，如果s'产生crash，然后就会被加入Sx这个集合，如果覆盖新路，s'就会被加入一开始的队列S。

Why the fuzzer generates p new inputs?

Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. 2016. [Coveragebased Greybox Fuzzing As Markov Chain](#). In Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS '16). 1032–1043.

Transition probability $P_{ij} = p\{s' \text{ [generated by } s \text{ (who exercises Path i)] exercises Path j}\}$



至于为什么要做 p 次，文章中只是简单说明了一下，在另一篇文章工作中他们搞了一个AFLfast，如上图，他们发现covered-based fuzzer总是更多的执行高频率路径，30%的路径只被单个输入执行过，而10%的路径被1k~10k的输入执行过。在AFLfast工作中他们将covered-based greybox fuzzing模型化成一个马尔科夫链，状态转换的概率 p_{ij} 由上面这个等式给出，执行路径i的种子 s mutate产生一个能执行路径j的种子 s' 的概率。他们发现通过稳定分布的密度可以描述经过多次迭代后高频率路径被执行的概率。Böhme等人开发了一个技术通过调整fuzz次数，可以使fuzzer引向低频率路径，这个次数称为能量，但是马尔科夫链中的能量只局限于其中一个状态，但是温度在模拟退火中具有全局性。

Measure of distance

An **inter-procedural** measure of distance is established at instrumentation-time and can be efficiently computed at runtime.

Inter-procedural

Inter-procedural Call graph (CG)

Intra-procedural control-flow graphs (CFGs)

下面介绍AFLGo技术细节，从距离测量方式开始说起，尽管他们定义的是一种过程间
的距离测量方式，实质上结合了过程间的函数调用图与过程内的控制流图，有过程内
分析也有过程间分析。距离测量方式在插桩时建立，提供运行时的计算。

Measure of Distance between a Seed Input and Multiple Target Locations

- Function-level target distance

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

$d_f(n, n')$ as the number of edges along the shortest path between functions n and n' ;

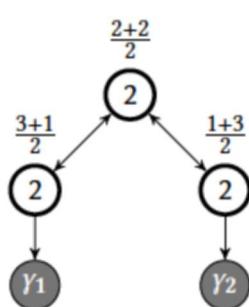
$d_f(n, T_f)$ as the harmonic mean of the function distance between n and any reachable target function;

$R(n, T_f)$ the set of all target functions that are reachable from n in CG.

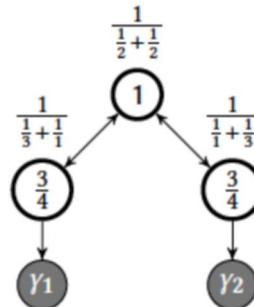
为了计算跨函数的基本块距离，他们在函数调用图 (CG) 中为每个函数节点赋值，为每个过程内的控制流程 (CF) 的每个基本块节点分赋值。那么函数层距离决定了函数调用图(CG)一个函数到目标函数的距离。定义这个距离是两个函数最短路径的边个数，距离 $d_f(n, T_f)$ 是函数 n 到所有可到达的目标函数的距离的一个调和平均数。

A Measure of Distance between a Seed Input and Multiple Target Locations

- Why they use harmonic mean?



(a) Arithmetic Mean



(b) Harmonic Mean

那么他们为什么用调和平均数，因为调和平均数可以区分不同点到目标的距离。上图白色的部分里面的数字是节点距离，灰色的是目标节点。左边三个节点到两个目标距离的算数平均数都是2，右边的则不一样。

A Measure of Distance between a Seed Input and Multiple Target Locations

- Basic-block-level target distance

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

$d_b(m_1, m_2)$ as the number of edges along the shortest path between BB between m_1 and m_2 in the control-flow graph G_i of function i ;

$N(m)$ be the set of functions called by basic blocks : $\forall n \in N(m). R(n, T_f) \neq \emptyset$;

T be the set of basic block m in G_i . $\forall m \in T. N(m) \neq \emptyset$

基本块距离不仅考虑函数层距离，还考虑同一控制流程图中基本块间的距离，
 $d_b(m_1, m_2)$ 定义同一函数 i 中控制流程图 G_i 两个基本块之间最短路径的边数， $N(m)$ 是被基本块调用且可达目标函数的函数集合， T 是相应的 $N(m)$ 非空的基本块集合， $c=10$ 是一个常量，放大函数层距离的作用。那么如果 m 属于 T ，直接取函数层最小距离，如果 m 不属于 T ，也就是 m 不调用函数，或者调用的函数不可达，那么就得算他到集合 T 中基本块的距离及相应函数层距离之和，最后求调和平均数。

A Measure of Distance between a Seed Input and Multiple Target Locations

- The seed distance

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

$\xi(s)$ be the execution trace of a seed s

最后定义种子距离 $d(s, Tb)$, ξ 是种子 s 的执行路径, 这个路径也可以看成一个被 s 执行过的基本块集合, 分母表示基本块数量, 分子表示, 每个基本块到目标的距离的累加,

A Measure of Distance between a Seed Input and Multiple Target Locations

- The normalized seed distance

$$\tilde{d}(s, T_b) = \frac{d(s, T_b) - \text{minD}}{\text{maxD} - \text{minD}}$$

$$\text{minD} = \min_{s' \in S} [d(s', T_b)]$$

$$\text{maxD} = \max_{s' \in S} [d(s', T_b)]$$

最后有定义了标准化种子距离，注意这里 $d\sim$ 的取值范围应该是0-1闭区间。基本块距离是在插桩时运算的，标准化种子 $d\sim$ 距离实在fuzz运行时，通过参考之前基本块距离运算的。

Annealing-based Power Schedules

- Simulated Annealing(SA)

SA is a Markov Chain Monte Carlo method (MCMC) for approximating the global optimum in a very large, often discrete search space within an acceptable time budget

The exponential cooling schedule:

$$T_{\text{exp}} = T_0 \cdot \alpha^k$$

where $\alpha < 1$ is a constant and typically $0.8 \leq \alpha \leq 0.99$. $k \in \mathbb{N}$.

为什么选取这个模拟退火，这里就涉及了马尔科夫蒙特卡洛方法以及一些统计学原理，这里我就不具体说明，但是这么做的目的就是为那些离目标近的种子分配更多能量，距离远的种子分配少的能量，随着时间推进，这种能量分配的差距就会越来越大。SA算法能够在可接受的时间预算中在非常大而离散的搜索空间中渐进收敛得到全局最优解（在这里，这个解集就是能跑最多目标的种子集合）SA在随机游走的过程中它总是接受better solutions当然有时也接受worse solution，温度是SA算法的一个参数，用来控制接收不好solutions，并且他也随降温公式降低。初始化温度为1,SA算法可能接受不好solution的概率很高，但是当温度为0时，他就只接受better solution，降温公式控制收敛速度，这里采用的是一个指数降温公式(7)

Annealing-based Power Schedules

- Annealing-based power schedule

At time t_x , the annealing process should enter “exploitation” after sufficient time of “exploration”.

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

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

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

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

在时间 t_x , 退火过程由“exploration”进入“exploitation”阶段, 对于时间 t_x , 那么 T_{exp} 在时间 t 的值应该是 (11)

Annealing-based Power Schedules

- Annealing-based power schedule

Given the seed s and the target locations T_b , the APS assigns energy p as:

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

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

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

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

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

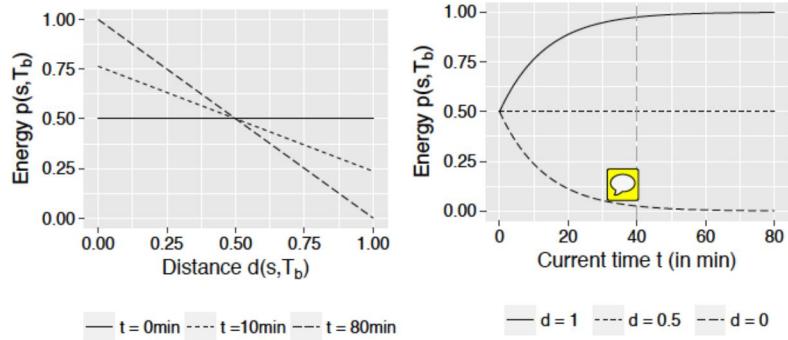
最后有这个能量公式，可以看出前面 α 的大小决定这个函数是关于 t 的增函数还是减函数

Annealing-based Power Schedules

- Annealing-based power schedule

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

$$\begin{aligned} T_{\text{exp}} &= \alpha^{\frac{t}{t_x} \frac{\log(0.05)}{\log(\alpha)}} \\ &= 20^{-\frac{t}{t_x}} \end{aligned}$$



这两幅图 $t_x=40$ ，左图是种子距离与能量的函数图，线性的，右图是时间与能量的函数图，在 $t=0$ 的时刻，所有种子能量一样，但是随着时间推进，近距离种子能量越来越多，远距离种子能量越来越少，但是值都保持在0到1的闭区间，我感觉右图画错了，

Annealing-based Power Schedules

- Practical Integration

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

The annealing-based power factor $f = 2^{10(p(s, T_b) - 0.5)}$ controls the increase or reduction of energy assigned by AFL's power schedule.

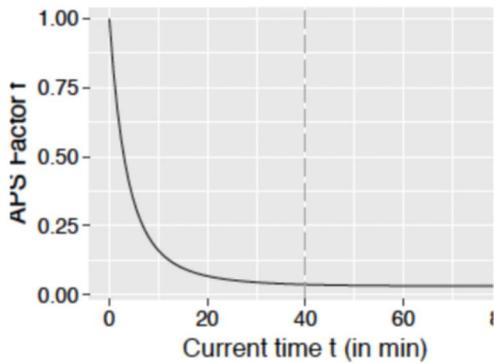
AFL也有一个能量公式，它基于执行时间和input 大小，所以将他们的退火能量公式与AFL现存的结合起来，结果如下

Annealing-based Power Schedules

- Practical Integration

$$\lim_{t \rightarrow \infty} \hat{p}(s, T_b) = \frac{p_{\text{afl}}}{32} \quad \text{if } \tilde{d}(s, T_b) = 1$$

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



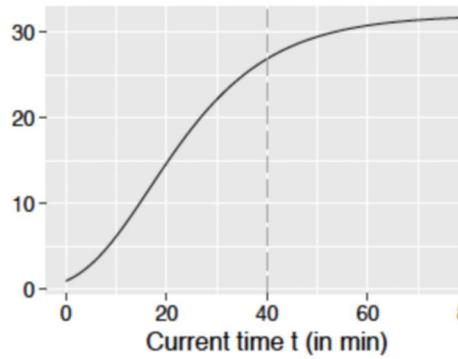
距离为一的种子在 $t=0$ 时能量为 1，这个与一开始AFL分配的是一样的，但是 10 分钟后，能量降到 15%，通过上面的公式，我们能看到，那些远离目标的种子分配的能量越来越少， t 趋于无穷时，大概是 afl 分配的 32 分之一，

Annealing-based Power Schedules

- Practical Integration

$$\lim_{t \rightarrow \infty} \hat{p}(s, T_b) = 32 \cdot p_{\text{afl}} \quad \text{if } \tilde{d}(s, T_b) = 0$$

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



当距离非常近的种子，能量越来越多，是afl的32倍。

Scalability of Directed Greybox Fuzzing

In a very early instantiation

AFLGo connects all the call-sites of one function with the first basic block of the called functions	several hours
AFLGo computes the target distance within the iCFG for every basic block as the average length of the shortest path to any target.	several hours

Today

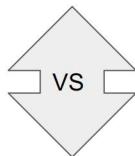
AFLGo compute function-level target distance in the call graph and compute the basic-block level target distance to the call sites within the same intra-procedural control-flow graph.

上面是他们设计AFLGo时的一个改进历程，起初他们构建进程间控制流程图(iCFG)，就把一个函数调用的所有函数的第一个基本块都连接起来。然后再计算过程间控制流程图(iCFG)中每个节点到目标距离。由于节点数目特别多，因而计算量大耗费时间多。现在他们控制流程图的节点距离只计算过程内的

Scalability of Directed Greybox Fuzzing

BB-level target distance relies on **Dijkstra's shortest-path algorithm** which has a worst-case complexity $O(V^2)$ where V is the number of nodes.

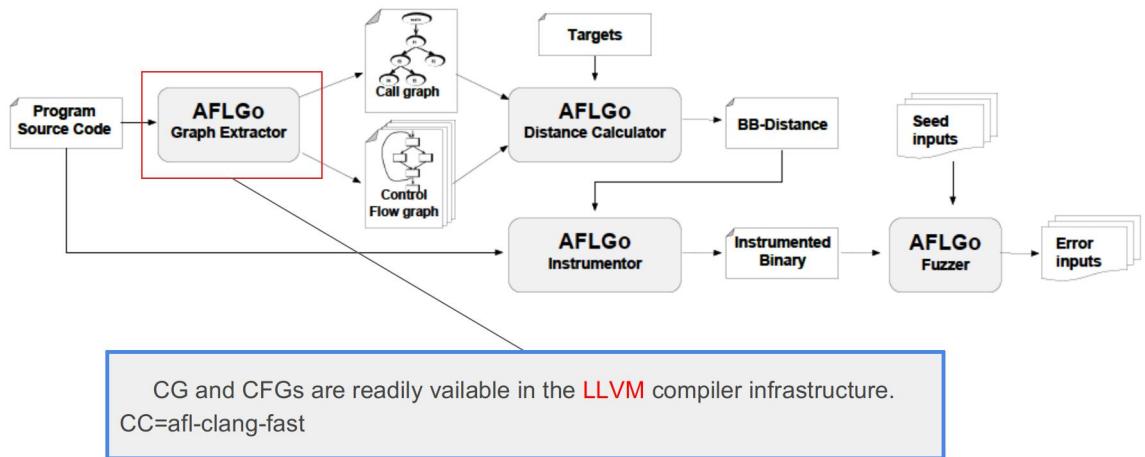
The complexity of a shortest-distance computation in the iCFG is $O(n^2 \cdot m^2)$



The complexity of our shortest distance computation $O(n^2 + nm^2)$

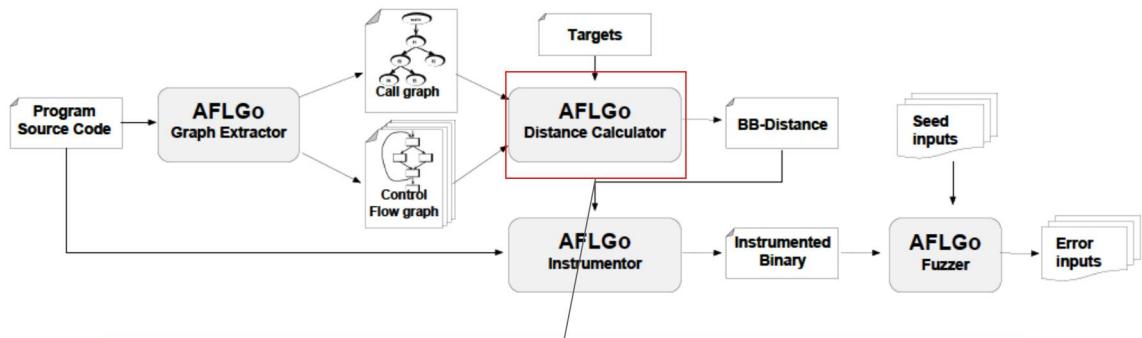
基本块层的距离计算复杂度如果按照Dijkstra's shortest-path algorithm算法，应该为为 V^2 ， V 是节点个数，那么对于过程间的距离计算，复杂度就应该是 $O(n^2*m^2)$ ， n 个 intra-procedural CFG，每个CFG m 个节点。现在AFLGo基本块层距离只计算过程内的 CFG 节点，时间复杂度降为 $O(n^2+m^2)$ 。不仅节省了大量的存储，而且把时间也从几小时缩短到几分钟。

Implementation



这是整个实现流程框架，AFLGo主要模块由灰色的标注，Graph extractor主要是生成CG与CFG，CG与CFG在LLVM架构中是可读的，所以编译时CC设为afl-clang-fast

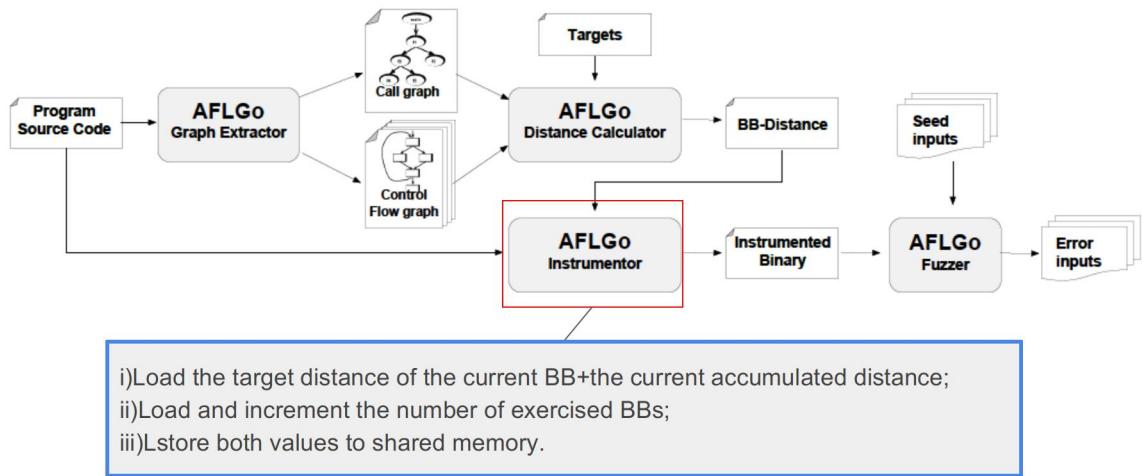
Implementation



The DC is implemented as a Python script that uses the [networkx](#) package for parsing the graphs and for shortest distance computation according to [Dijkstra's algorithm](#).

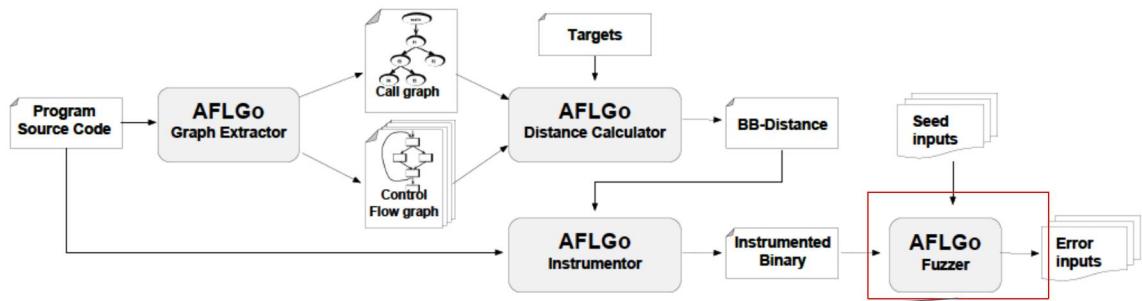
距离计算器DC是通过python脚本实现的，结合了之前的CG与CFG，利用networkx工具包解析图表，并且根据Djikstra's计算最短距离

Implementation



插桩的代码会记录跑过的分支，这些信息记录在64kb的共享存储区(bitmap)中，在一个64-bit结构中，他们扩展用了共享存储区的16字节，其中8字节进行距离的累加，8字节记录运行过得基本块数量。

Implementation



The additional 16 bytes in the shared memory inform the fuzzer about the current seed distance. The current seed distance is computed by dividing the accumulated BB-distance by the number of exercised BBs.

作者用AFL 2.40版本进行改进的，AFLGo根据我们的基于退火的能量公式进行测试。16字节的共享内存会通知fuzzer当前种子的距离，种子距离就是用基本块距离之和除以执行过的基本块数量。

APPLICATION 1: PATCH TESTING

AFLGo vs KATCH

Project Tools	diff, sdiff, diff3, cmp
Program Size	42,930 LoC
Chosen Commits	175 commits from Nov'09–May'12

GNU Diffutils

Project Tools	addr2line, ar, cxxfilt, elfedit, nm, objcopy, objdump, ranlib, readelf size, strings, strip
Program Size	68,830 LoC + 800kLoC from libraries
Chosen Commits	181 commits from Apr'11–Aug'12

GNU Binutils

下面部分来介绍一下实际的测试，作者也选择了一些其他的开源项目来测试。像有的开源项目已经被测试很久，也没再发现什么新的漏洞，但是现在这些项目又出现了新版本，新版本又出现了一些新特征，那么现在就需要对这些新特征部分进行测试。目前最先进的补丁测试工具是KATCH.我们就用它作为对比工具，和KATCH作者测试同样的程序，试验条件都一样。这些使他们测试的应用。

APPLICATION 1: PATCH TESTING

Table 1: Patch coverage results showing the number of previously uncovered targets that KATCH and AFLGo could cover in the stipulated time budget, respectively.

	#Changed Basic Blocks	#Uncovered Changed BBs	KATCH	AFLGo
<i>Binutils</i>	852	702	135	159
<i>Diffutils</i>	166	108	63	64
Sum	1018	810	198	223

这个表展示的是实验结果，显示的是被改变过的BB，但是没有被测试过的，然后KATCH与AFLGo在规定的时间cover的BB,AFLGo比KATCH多了13%，AFLGo 223，而KATCH是198.还有很多目标没被测试到是因为他们不可达，比如和操作系统有关等。再比如很多更改的基本块只能通过寄存器间接调用或跳转访问，但是如果这个边在分析的函数调用图或者控制流程图里，那么就不存在这样的情况。

APPLICATION 1: PATCH TESTING

Patch Coverage



Figure 9: Venn Diagram showing the number of changed BBs that KATCH and AFLGo cover individually and together.

当然他们也有各自的技术优势，符号执行可以进入难以访问的位置，而greybox fuzzer可以快速探索到目标的许多路径，而不会卡在某个位置，他们相互补充，比单独的效果好。

APPLICATION 1: PATCH TESTING

Vulnerability Detection

Table 2: Showing the number of previously unreported bugs found by AFLGo (reported Apr'2017) in addition to the number of bug reports for KATCH (reported Feb'2013).

	KATCH #Reports	AFLGo		
		#Reports ¹⁴	#New Reports	#CVEs
<i>Binutils</i>	7	4	12	7
<i>Diffutils</i>	0	N/A	1	0
Sum	7	4	13	7

AFLGo除了发现之前KATCH发现7个bugs中的4个，还额外发现了13个bugs，存在于现在的大部分版本里，bug类型就是缓冲区溢出，空指针之类的，其中有7个已经报CVE了。其中12个bug的发现借助于AFLGo的定向性，7个bug就在目标位置，5个是在目标位置附近发现的。

APPLICATION 1: PATCH TESTING

Vulnerability Detection

Table 3: Bug report and CVE-IDs for discoveries of AFLGo.

	Report-ID ¹⁶	CVE-ID	Report-ID	CVE-ID
Binutils	21408		21418	
	21409	CVE-2017-8392	21431	CVE-2017-8395
	21412	CVE-2017-8393	21432	CVE-2017-8396
	21414	CVE-2017-8394	21433	
	21415		21434	CVE-2017-8397
	21417		21438	CVE-2017-8398

Diffutils <http://lists.gnu.org/archive/html/bug-diffutils/2017-04/msg00002.html>

AFLGo在实验中表现得比KATCH出色，我们将其归结为定向灰盒fuzz的效率，AFLGo在运行时不需要程序分析，因此产生的input比KATCH多几个数量级。另一个高效的根源在于AFLGO使用运行时检测器ASAN，而KATCH需要基于约束的错误检查机制。

APPLICATION 2: CONTINUOUS FUZZING

We integrated AFLGo into Google's fully automated fuzzing platform OSS-Fuzz and discovered 26 distinct bugs in seven securitycritical open-source projects.

Table 4: The tested projects and AFLGo's discoveries.

Project	Description	#Reports	#CVEs
libxml2 [55]	XML Parser	4	4 req. & assigned
libming [56]	SWF Parser	1	1 req. & assigned
libdwarf [52]	DWARF Parser	7	4 req. & assigned
libc++abi [51]	LLVM Demangler	13	none requested
libav [50]	Video Processor	1	1 req. & assigned
expat [48]	XML Parser	0	none requested
boringssl [45]	Google's fork of OpenSSL	0	none requested
Sum		26	10 CVEs assigned

在实际应用的研究中，他们将AFLGo与Google's OSS-Fuzz整合起来了,OSS-Fuzz是一款持续测试平台，专门测试一些我们常用的一些关键的库以及其他开源项目，测试出bug后会自动报给项目维护部门，直到被打补丁为止。截止今年5月17日，OSS-Fuzz已经整合了47个开源项目。OSS-Fuzz发现了1000bug，其中264个是高危漏洞。但是它整合的AFL与LibFuzzer都不是定向的，不能直接对最新的更改部分进行测试。因而我们将AFLGo整合到OSS-Fuzz这个自动测试平台，并且从7个关键开源库中发现了26个不同的漏洞。其中一些漏洞可以被利用实施拒绝服务攻击，远程恶意代码攻击。接下来我们将重点从libxml2与libming两个库的测试说起。

APPLICATION 2: CONTINUOUS FUZZING

- LibXML2

CVE-2017-{9049,9050}:Incomplete fixes

By directing the greybox fuzzer towards the changed statements in the previous fixes of these bugs, AFLGo effectively generated other crashing inputs that would expose these bugs that were supposed to be fixed.

LibXML2是一个用C写得用来解析XML的库，是php的核心组件，测试了50个最新版本，AFLGo发现了四个不同的缓冲区溢出，这些crash在最新版本的PHP中的DOM验证器是可以重现的。其中两个是无效的写入，大小可达4kb，可能会被用于任意代码执行。报了4个CVE.其中前两组是由于之前不完整修复，AFLgo对之前版本报出已经修复的bug部分进行测试，AFLgo产生了一些其他的crashing输入导致bug重现

APPLICATION 2: CONTINUOUS FUZZING

- LibXML2

CVE-2017-{9047, 9048}

xmlAddID in valid.c

commit ef709ce2

```
6397 ret &= xmlValidateOneElement(ctxt, doc, elem);
6398 if (elem->type == XML_ELEMENT_NODE) {
6399     attr = elem->properties;
6400     while (attr != NULL) {
6401         value = xmlNodeListGetString(doc, attr->children, 0);
6402         ret &= xmlValidateOneAttribute(ctxt, elem, attr, value);
6403         if (value != NULL)
6404             xmlFree((char *)value);
6405         attr = attr->next;
6406     }
6407     ns = elem->nsDef;
6408     while (ns != NULL) {
6409         if (elem->ns == NULL)
6410             ret &= xmlValidateOneNamespace(ctxt, doc, elem, ns,
6411                                         NULL, ns->href);
6412         else
6413             ret &= xmlValidateOneNamespace(ctxt, doc, elem, ns,
6414                                         elem->ns->prefix, ns->href);
6415         ns = ns->next;
6416     }
6417 }
```

这两个CVE的在最上行`xmlValidateOneElement`中有，而灰色的部分之前被解决了开发者在`valic.c`文件中的`xmlAddID`函数中加入了边界检查,来修补空指针废弃的问题，这个函数被`xmlParseOneAttribute` and `xmlValidateOneNamespace`调用，但是要执行这两个函数，解析器首先要执行`xmlValidateOneElement`，当这个函数执行print不同指针`content`时发生了溢出。

APPLICATION 2: CONTINUOUS FUZZING

- LibMing

CVE-2016-9831

CVE-2017-7578

```
ERROR: AddressSanitizer: heap-buffer-overflow
WRITE of size 1 at 0x62e00000b298 thread T0
#0 0x5b1be7 in parseSWF_RGBAA parser.c:68:14
#1 0x5f004a in parseSWF_MORPHGRADIENTRECORD parser.c:771:3
#2 0x5f0c1f in parseSWF_MORPHGRADIENT parser.c:786:5
#3 0x5ee190 in parseSWF_MORPHFILLSTYLE parser.c:802:7
#4 0x5f1bbe in parseSWF_MORPHFILLSTYLES parser.c:829:7
#5 0x634ee5 in parseSWF_DEFINEMORPHSHAPE parser.c:2185:3
#6 0x543923 in blockParse blocktypes.c:145:14
#7 0x52b2a9 in readMovie main.c:265:11
#8 0x528f82 in main main.c:350:2
```

Figure 11: This previously reported bug had been fixed but incompletely. AFLGo could reproduce the exact same stack trace on the most recent (fixed) version.

Libming是一个用来读及生成swf文件的库，他们测试了最新的50个版本，AFLGo发现了不完整修复，之前已经报过CVE-2016-9831，而且也发出打补丁公告，但是AFLGFo生成了另一个crash输入产生了同样的stacktrace,他们做了详细的分析然后报了一个CVE-2017-7578

APPLICATION 3: CRASH REPRODUCTION

AFLGo vs AFL

Program	CVE-ID	Type of Vulnerability
LibPNG [54]	CVE-2011-2501	Buffer Overflow
LibPNG [54]	CVE-2011-3328	Division by Zero
LibPNG [54]	CVE-2015-8540	Buffer Overflow
Binutils [6]	CVE-2016-4487	Invalid Write
Binutils [6]	CVE-2016-4488	Invalid Write
Binutils [6]	CVE-2016-4489	Invalid Write
Binutils [6]	CVE-2016-4490	Write Access Violation
Binutils [6]	CVE-2016-4491	Stack Corruption
Binutils [6]	CVE-2016-4492	Write Access Violation
Binutils [6]	CVE-2016-6131	Write Access Violation

Figure 12: Subjects for Crash Reproduction.

对于一些知名的商业软件都有自动bug report机制，比如一个videolan client(VLC)CRASH了，由于libpng的一个缓冲区溢出导致，那么用户可以一键将这个bug报给开发者，这样做不仅确保软件的服务质量，也能保证个人隐私，免受更过用户被攻击，但是VLC不会把产生crash的文件也传回来，因为要考虑个人隐私，嫌烦，软件只是传一个stacktrace与一些环境变量。需要开发团队自己重现crash。在上面这些项目的测试中，他们用AFLGo与AFL对比做实验，找了一些之前报告的漏洞，他们看看AFLGo与AFL重现这些漏洞所用的平均时间。

APPLICATION 3: CRASH REPRODUCTION

	CVE-ID	Tool	Runs	μ TTE	Factor	\hat{A}_{12}	
LibPNG [54]	2011-2501	AFLGo	20	0h06m	2.81	0.79	
		AFL	20	0h18m	—	—	
	2011-3328	AFLGo	20	0h40m	4.48	0.94	
		AFL	18	3h00m	—	—	
	2015-8540	AFLGo	20	0m26s	10.66	0.87	
		AFL	20	4m34s	—	—	
	2016-4487	AFLGo	20	0h02m	1.64	0.59	
		AFL	20	0h04m	—	—	
Binutils [6]	2016-4488	AFLGo	20	0h11m	1.53	0.72	
		AFL	20	0h17m	—	—	
	2016-4489	AFLGo	20	0h03m	2.25	0.68	
		AFL	20	0h07m	—	—	
	2016-4490	AFLGo	20	1m33s	0.64	0.31	
		AFL	20	0m59s	—	—	
	2016-4491	AFLGo	5	6h38m	0.85	0.44	
		AFL	7	5h46m	—	—	
	2016-4492	AFLGo	20	0h09m	1.92	0.81	
		AFL	20	0h16m	—	—	
	2016-6131	AFLGo	6	5h53m	1.24	0.61	
		AFL	2	7h19m	—	—	
			<i>Mean</i> \hat{A}_{12}		<i>Median</i> \hat{A}_{12}		
			0.66		0.68		

这是对比的结果，TTE就是产生第一个导致error的输入的时间，Factor是AFL的TTE除以AFLGo的TTE，这个值大于1说明AFLgO的效果更好一些。
 A12代表AFLGo的TTE比AFL的TTE小的概率，统计效果明显的加粗表示。在重现libPNG的CVE， AFLgO是AFL的3-11倍快，在binutils中， AFLGo通常比AFL快1.5-2倍。

APPLICATION 3: CRASH REPRODUCTION

Does DGF Outperform the Symbolic Execution-based State of the Art?

AFLGo vs BugRedux

Subjects	BugRedux	AFLGo	Comments
sed.fault1	✗	✗	Takes two files as input
sed.fault2	✗	✓	
grep	✗	✓	
gzip.fault1	✗	✓	
gzip.fault2	✗	✓	
ncompress	✓	✓	
polymorph	✓	✓	

那么DGF要比最先进的符号执行好吗？那么这个实验用BugRedux最先进的crash重现工具（基于KLEE）与AFL对比，实验环境都是一样的，在对比中，为了使BugRedux也能产生非常好的效果，我们将stack trace中的method-call作为目标。最后可以看到AFLGo效果好，其中有一个AFLGo没成功是因为他没办法吧两个文件同时作为输入。有4个10分钟内重现，其余两个4小时，远低于我们要求的24小时时间预算。

CONCLUSION

Directed greybox fuzzing can be used in myriad ways:

- towards problematic changes or patches;
- towards critical system calls or dangerous locations;
- towards functions in the stacktrace of a reported vulnerability that we wish to reproduce.

Integrate AFLGo into OSS-Fuzz

```
$ git clone https://github.com/aflgo/aflgo.git  
$ git clone https://github.com/aflgo/oss-fuzz.git
```

这篇paper作者在插装时进行了程序分析，这样保持了灰盒的运行时效率，同时在测试时采用了模拟退火方法为seed进行能量分配，测试结果最先进的定向符号执行工具效果好。DFG也可以用在很多应用，比如补丁测试，或者测试有存在问题的升级代码，再比如说面向系统核心函数等等。说一下个人的几点思考，就算是定向的，但本质还是随机的，灰盒fuzz本质上是一种随机方法，input进行随机mutate，随机的路径等等。再比如说mutate大部分会产生一些非法输入，在有checksum的应用中遇到input格式检查时就无法通过，他这时的directness效果好坏没有具体说明。还有很多项目他们没有做过测试，这个效果好坏就不好说了。