辰 组

White paper

翻译

版本：V1.00

**文档修订记录**

|  |  |  |  |
| --- | --- | --- | --- |
| **版本编号** | **变更简要说明** | **变更人** | **日期** |
| Ver 0.05 | 新建文档 | 237534354 | 2017-6-14 |
| Ver 1.00 | 增加AFL-ios | 237534354 | 2017-7-03 |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

目录

1 前言 5

2 适用范围 5

3 术语和定义 5

4 缩略语 5

5 AFL 翻译 6

5.1 AFL FUZZ 技术白皮书 6

5.2 设计简介 6

5.3 覆盖度量 6

5.4 检测新行为 7

5.5 输入队列 8

5.6 剔除语料 9

5.7 修剪输入文件 9

5.7 模糊策略 10

5.7 字典 11

5.8 重复的崩溃 11

5.9 调查crash 11

5.10 fork 服务器 11

5.11 并行 12

5.12 二进制instumentation 12

5.13 afl 分析工具 12

6 AFL IOS移植 14

6.1 AFL IOS 14

6.1 AFL 应用到iOS 14

6.2 AFL build 14

6.3 AFL 移植修改 14

7 AFL 实验 15

7.1 AFL Unix 15

7.2 AFL 源码ArchiveALL 15

7.3 AFL 其他源码 与库 15

附录. 16

# 1 前言

本文档的目的是翻译AFL <http://lcamtuf.coredump.cx/afl/> 技术白皮书。

本文档由看雪 辰组起草

主要起草人：237534354

# 2 适用范围

本文档适用初学者或开发人员 了解AFL。

# 3 术语和定义

# 4 缩略语

AFL： American Fuzz loop；

# 5 AFL 翻译

## 5.1 AFL FUZZ 技术白皮书

==============================================================================

本文档简要介绍了American Fuzzy Lop的内容。有关一般使用说明，请参阅README。如果想知道  AFL背后的动机和设计目标，见historical\_notes.txt。

## 5.2 设计简介

AFL最优的地方不是专注于任何单一的操作原理，也不是任何具体理论的概念证明。而是该工具可以被认为是在实践中经过测试的黑客集合，被发现是令人惊讶的有效的，并且已经以当时可以想到的最简单，最强大的方式来实现。

由于可用的轻量级作为该工具的基础，使许多产生的功能成为可能，但是这种机制应该被认为仅仅是一种手段。唯一真正的原则是速度，可靠性和易用性。

## 5.3 覆盖度量

注入到已经编译程序中捕获分支（边缘）覆盖以及交叉分支命中计数。在分支点注入的代码基本上等同于：

  cur\_location = <COMPILE\_TIME\_RANDOM>;

  shared\_mem [cur\_location ^ prev\_location] ++;

  prev\_location = cur\_location >> 1;

cur\_location值随机生成以简化过程连接复杂项目，保持异或输出均匀分布。

shared\_mem []数组是一个64 kB的SHM区域，由调用者传递给被检测的二进制文件。输出映射中设置的每个字节都可以被认为是一个特定的（branch\_src，branch\_dst）元组在检测代码中的命中。

选择映射的大小，使得几乎所有的碰撞都是零星的目标，通常运动在2k和10k之间可分支的支点：

|  |  |  |
| --- | --- | --- |
| 分支数量 | 命中元组几率 | 示例目标 |
| 1000 | 0.75% | giflib，lzo |
| 2000 | 1.5% | zlib，tar，xz |
| 5000 | 3.5% | libpng，libwebp |
| 10000 | 7% | libxml |
| 20000 | 14% | 源码 |
| 50000 | 30% | - |

同时它的大小足够小，可以在接收端以微秒为单位分析映射，并轻松适应二级缓存在二级缓存中。

这种覆盖形式，相对于简单的块覆盖 程序的执行路径会提供更多的视野。特别地，它简单区分以下执行跟踪：

  A - > B - > C - > D - > E（元组：AB，BC，CD，DE）

  A - > B - > D - > C - > E（元组：AB，BD，DC，CE）

这有助于在底层代码中发现微妙的故障，因为安全漏洞常常与异常关联或不正确的状态转换，而不仅仅是到达新的基本程序块。

在前面显示的伪代码的最后一行中的移位操作的原因是保留元组的方向性（不存在，A ^ B与B ^ A无法区分）并保留紧密循环的标识（否则A ^ A明显等于B ^ B）。

英特尔CPU上没有简单的饱和算术运算符意味着命中计数器有时会绕到零。由于这是一个非常不可能和本地化的事件，所以被视为一种可以接受的表现权衡

## 5.4 检测新行为

fuzzer在以前的执行过程中保持了一个全局的元组映射。这些数据可以与单个跟踪快速比较，只需要几个双字或四字 范围的指令和一个简单的循环即可更新。

当突变输入产生包含新元组的执行跟踪时，相应的输入文件将被保留并进行路由以便稍后进行其他处理（请参见第5.5节）。即使它们的整体控制流序列是唯一的，也不会丢弃在执行跟踪中不触发新的局部尺度状态转换的输入（即，不产生新的元组）。

这种方法允许对程序状态进行非常细粒度和长期的探索，而不必对复杂执行跟踪进行任何计算密集和脆弱的全局比较，同时避免路径爆炸的祸害。

为了说明算法的属性，考虑到由于存在新的元组（CA，AE），下面显示的第二个跟踪将被认为是新的：

  ＃1：A - > B - > C - > D - > E

  ＃2：A - > B - > C - > A - > E

同时，＃2被处理，以下模式不会被视为唯一的，尽管总体执行路径显着不同：

  ＃3：A - > B - > C - > A - > B - > C - > A - > B - > C - > D - > E

除了检测新的元组外，模糊器还考虑粗略的元组命中数。这些分为几个桶：

1，2，3，4-7，8-15，16-31，32-127，128+

在某种程度上，桶的数量是一个实现方法：它允许由仪器产生的8位计数器的就地映射到由fuzzer可执行文件依赖的8位位图以跟踪已经看到的每个元组的执行次数。

单个桶的范围内的更改将被忽略;从一个桶到另一个桶的转换被标记为程序控制流程中的一个有趣的变化，并且被路由到下面所述的进化过程。

命中计数行为提供了一种区分潜在有趣的控制流程更改的方法，例如当通常只打一次时执行两次代码的代码块。同时，对于经验上不那么显着的变化，例如从47个周期到48个循环的循环来说，这是相当不敏感的。计数器还在密集的跟踪图中提供了一定程度的“意外”免疫来抵抗元组碰撞。

通过内存和执行时间限制，执行程序相当严格;默认情况下，超时设置为初始校准执行速度的5倍，向上舍入为20毫秒。积极的超时是为了防止戏剧性的Fuzzer性能下降，因为降低了速度，比如说，100％的速度提高了1％的覆盖率;我们务实地拒绝他们，并希望fuzzer会找到一种更便宜的方式来达到相同的代码。经验测试强烈地表明，更大的时间限制是不值得的。

## 5.5 输入队列

在程序中产生新的状态转换的变量测试用例被添加到输入队列中，并作为未来一轮模糊的起始点。他们补充，但不会自动替换现有的发现。

与更贪心的遗传算法相反，该方法允许该工具逐步探索底层数据格式的各种不相交和可能相互不兼容的特征，如图所示：

  http://lcamt​​uf.coredump.cx/afl/afl\_gzip.png

这里讨论这个算法的几个实例：

  http://lcamt​​uf.blogspot.com/2014/11/pulling-jpegs-out-of-thin-air.html

  http://lcamt​​uf.blogspot.com/2014/11/afl-fuzz-nobody-expects-cdata-sections.html

这个过程产生的合成语义库基本上是一个紧凑的集合，“嗯，这有新意！”输入文件，并可用于将任何其他测试流程放在一起（例如，手动压缩测试资源密集型桌面应用程序）。

使用这种方法，大多数目标的队列增长到1k到10k之间的条目;其中约10-30％是由于发现新的元组，其余与命中数的变化有关。

下表比较了使用几种不同方法引导模糊时发现文件语法和探索程序状态的相对能力。测试目标是使用-O3编译的GNU补丁2.7.3，并用虚拟文本文件播种;该会话包括使用afl-fuzz输入队列的单次传递：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Fuzzer guidance  Strategy used | Blocks  reached | Edges  reached | Edge hit  cnt var | Highest-coverage  Test case generated |
| (Initial file) | 156 | 163 | 1.0 | None |
| Blind fuzzing S | 182 | 205 | 2.23 | 前2字节 RCS差异 |
| Blind fuzzing S | 228 | 265 | 2.23 | 前4字节 c 模式差异 |
| Block Coverage | 855 | 1130 | 1.57 | 全 RCS 差异 |
| Edge Coverage | 1452 | 2070 | 2.18 | 单模块 c 模式差异 |
| AFL mode | 1765 | 2597 | 4.99 | 四模块 c 模式差异 |

盲模糊（“S”）的第一个条目对应于执行一轮测试; 第二组数字（“L”）显示了与循环运行相当的多个执行周期的循环中运行的模糊器，这需要更多时间来完全处理不断增长的队列。

在单独的实验中已经获得了大致相似的结果，其中模糊器被修改以编译所有随机的模糊阶段，并且仅留下一系列基本的顺序操作，例如依次翻转。 因为这种模式将无法改变输入文件的大小，所以会话被种下一个有效的统一差异：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Queue extension  strategy used | Block  reached | Edges  reached | Edge hit  Cnt var | Number of unique  Crashes found |
| (Initial file) | 624 | 717 | 1.0 | - |
| Blind fuzzing | 1101 | 1409 | 1.60 | 0 |
| Block coverage | 1255 | 1649 | 1.48 | 0 |
| Edge coverage | 1259 | 1734 | 1.72 | 0 |
| AFL model | 1452 | 1040 | 3.16 | 1 |

前面已经提到，遗传模糊的一些先前的工作依赖于维护一个单一的测试用例，并发展它以最大化覆盖。 至少在上述测试中，这种“贪心”的方法似乎对盲模糊策略没有实质性的好处。

## 5.6 剔除语料

以上概述的进步状态探索方法意味着在游戏中稍后合成的一些测试用例可能具有边缘覆盖，这是其祖先提供的覆盖范围的严格的超集。

为了优化模糊工作，AFL使用快速算法周期性地重新评估队列，该算法选择了仍然覆盖到目前为止所看到的每个元组的较小的测试用例子集，其特征使得它们对该工具特别有利。

该算法通过将每个队列条目分配成与其执行延迟和文件大小成比例的工作; 然后为每个元组选择最低得分候选。

然后依次对元组进行处理，使用如下的简单工作流程：

* + 1. 找到没有在临时工作集的下一个元组，
    2. 找到这个元组的获胜队列条目，
    3. 注册\*所有\*元组中存在的该条目的工作集的路径，
    4. 如果集合中没有元组，请转到1。

所产生的“受欢迎”条目的语料库通常比起始数据集小5-10倍。 “不受欢迎”条目不会被丢弃，但是在队列中遇到不同的概率时会跳过它们：

- 如果有新的，尚未要被模糊化受欢迎条目目前在队列中，不受欢迎条目的99％将被跳过去获得受欢迎条目。

- 如果没有新的受欢迎条目。

- 如果当前不受欢迎条目之前是模糊的，那么95％将被跳过。

- 如果还没有经历过任何模糊的回合，跳过的可能性下降到75％。

基于经验测试，这提供了队列循环速度和测试用例分集之间的合理平衡。

可以在具有afl-cmin的输入或输出语料库上执行稍微更复杂但更慢的剔除。 该工具永久丢弃冗余条目，并产生适合与ffl-fuzz或外部工具一起使用的较小的语料库。

## 5.7 修剪输入文件

文件大小对模糊性能具有显着的影响，因为大文件使目标二进制速度更慢，并且因为它们降低了突变将接触重要格式控制结构而不是冗余数据块的可能性。 这在perf\_tips.txt中有更详细的讨论。

用户将提供低质量起始语料的可能性，一些类型的突变可以产生迭代地增加生成的文件的大小的效果，所以重要的是抵制这种趋势。

幸运的是，instrumentation反馈提供了一种简单的方法来自动修剪输入文件，同时确保对文件所做的更改对执行路径没有影响。

afl-fuzz中的内置微调器尝试以可变长度和间隔顺序删除数据块; 任何不影响跟踪映射的校验和的删除都将提交到磁盘。 修剪的设计不是特别彻底的; 相反，它会尝试在精度和在进程上花费的execve（）调用数量之间取得平衡，选择块大小和逐步匹配。 平均每个文件收益约为5-20％

独立的afl-tmin工具使用更详尽的迭代算法，并尝试在修剪文件上执行字母表归一化。 afl-tmin的操作如下：

首先，该工具自动选择操作模式。 如果初始输入崩溃目标二进制文件，则afl-tmin将以非检测模式运行，只需保留产生更简单文件的任何调整，但仍会使目标崩溃。 如果目标没有崩溃，该工具将使用检测模式，并且只保留产生完全相同执行路径的调整。

实际的最小化算法是：

1. 尝试使用大的步进大小清除大块数据。 经验证明，这可以通过在以后抢占更细粒度的努力来减少执行人员的数量。
2. 使用二分法检索方法执行块大小与步进大小减小的块删除遍历。
3. 通过计数唯一字符并尝试批量替换每个值为零的值来执行字母表归一化。
4. 作为最后一个结果，在非零字节上执行逐字节归一化

而不是使用0x00字节置零，afl-tmin使用ASCII数字“0”。 这是因为这样的修改不太可能干扰文本解析，因此更有可能导致文本文件的成功最小化。

这里使用的算法比在学术工作中提出的一些其他测试用例最小化方法参与的要少，但在大多数现实世界的应用中，执行时间要少得多，并且往往会产生可比较的结果。

## 5.7 模糊策略

instrumentation提供的反馈使得您可以轻松了解各种模糊策略的价值，并优化其参数，使其在广泛的文件类型中工作得很好。 afl-fuzz使用的策略通常是格式不可知的，更详细地讨论在这：

<http://lcamtuf.blogspot.com/2014/08/binary-fuzzing-strategies-what-works.html>

特别是在早期，特别是afl-fuzz所做的大部分工作实际上是高度确定性的，并且仅在稍后阶段进行随机堆叠修改和测试案例拼接。 确定性策略包括：

- 具有不同长度和步进顺序位翻转，

   - 小整数的顺序加法和减法，

   - 已知有趣整数（0,1，INT\_MAX等）的顺序插入，

以确定性步骤开放的目的与其产生紧凑测试用例的趋势和非崩溃和崩溃输入之间的小差异有关。

以确定性模糊的方式，非确定性步骤包括堆叠的位翻转，插入，删除，算术和拼接不同的测试用例。

所有这些策略的相对收益率和execve（）成本已经被调查，并在上述博客文章中讨论。

由于在history\_notes.txt（主要是性能，简单性和可靠性）中讨论的原因，AFL通常不会试图解释特定突变和程序状态之间的关系; 模糊步骤名义上是盲目的，仅由输入队列的进化设计引导。这就是说，有一个（简单）例外：当一个新的队列条目经过最初设定的确定性的模糊步骤和调整，以文件中的某些区域中，观察到在执行路径的校验无影响 ，它们可能被排除在确定性模糊的剩余阶段之外，并且模糊器可以直接进行随机调整。 特别是对于详细的，人类可读的数据格式，这样可以将执行人员的数量减少10-40％左右，而不会明显降低覆盖率。 在极端情况下，如通常块对齐的焦油档案，收益可高达90％。

因为底层的“效应图”是本地的每个队列条目，并且只有在不改变底层文件的大小或一般布局的确定性阶段才保持有效，这种机制似乎非常可靠地工作，并被证明是简单易行的。

## 5.7 字典

instrumentation提供的反馈使得在某些类型的输入文件中自动识别语法令牌变得容易，并且检测预定义或自动检测的词典术语的某些组合构成了测试解析器的有效语法.

如何将这些功能都AFL-fuzz内实现的,讨论可以在这里找到:

<http://lcamtuf.blogspot.com/2015/01/afl-fuzz-making-up-grammar-with.html>

实质上，当基本的，通常容易获得的语法令牌以纯随机的方式组合在一起时，队列的instrumenataion和进化设计一起提供了一种反馈机制，以区分无意义的突变和触发仪器化代码中的新行为的突变 并且在这个发现的基础上增加构建更复杂的语法。

已经显示了这些词典，使得fuzzer能够快速重构高度冗长和复杂的语言（如JavaScript，SQL或XML）的语法; 在上述博客文章中给出了生成的SQL语句的几个示例。

有趣的是，AFL instrumentation还允许Fuzzer自动隔离输入文件中已经存在的语法标记。 它可以通过查找运行的字节来实现，当被翻转时，会对程序的执行路径产生一致的更改; 这暗示了与代码中嵌入的预定义值的底层原子比较。 fuzzer依靠这个信号来构建紧凑的“自动词典”，然后与其他模糊策略一起使用

## 5.8 重复的崩溃

崩溃的重复删除是任何合格的模糊工具更重要的问题之一。 许多幼稚的方法遇到的问题; 特别地，如果故障发生在通用库函数（例如，strcmp，strcpy）中，则仅查看故障地址可能导致完全不相关的问题聚集在一起; 而如果可以通过多个不同的，可能递归的代码路径达到故障，则校验和调用堆栈返回可能导致极端的崩溃计数膨胀。

在afl-fuzz中实现的解决方案考虑到如果满足两个条件中的任何一个，则崩溃是唯一的：

-crash trace包括在以前的任何一次崩溃中没有看到的元组，

   -crash trace缺少始终存在于早期故障中的元组。

这种方法很早就容易受到一些路径通胀的影响，但表现出非常强的自我限制作用，类似于作为afl-fuzz基石的执行路径分析逻辑.

## 5.9 调查crash

许多类型的crash的可利用性可能是模糊的; afl-fuzz尝试通过提供探测crash模式来解决这个问题，其中已知故障的测试用例以与fuzzer的正常操作非常相似的方式模糊，但是具有导致任何非崩溃突变被丢弃的约束.

这种方法的价值的详细讨论可以在这里找到：

<http://lcamtuf.blogspot.com/2014/11/afl-fuzz-crash-exploration-mode.html>

该方法使用仪器反馈来探索崩溃程序的状态，以超越模糊的故障条件，然后隔离新发现的人工检测输入。

关于崩溃的问题，值得注意的是，与正常的队列条目相比，崩溃的输入是\*非\*修剪; 它们与发现完全一致，以便将它们与队列中的父，非崩溃条目进行比较更容易。 也就是说，afl-tmin可以随意缩减。

## 5.10 fork 服务器

为了提高性能，afl-fuzz使用“fork服务器”，其中模糊进程通过execve（），链接和libc初始化一次，然后通过利用写时复制从已停止的进程映像克隆。 实现在这里更详细地描述：

<http://lcamtuf.blogspot.com/2014/10/fuzzing-binaries-without-execve.html>

fork服务器是注入instrumentation的一个组成部分，并且在第一个instrumentation功能上停止等待来自afl-fuzz的命令

使用快速目标，fork式服务器可以提供相当的性能提升，通常在1.5倍和2倍之间。 也可以：

- 以手动（“延迟”）模式使用fork式服务器，跳过较大的用户选择的初始化代码块。 它需要对目标程序进行非常适中的代码更改，并且有了一些目标，可以 产生10倍+性能提升。

- 启用“持久”模式，其中单个进程用于尝试多个输入，极大地限制了重复的fork（）调用的开销。 这通常需要对目标程序进行一些代码更改，但可以将快速目标的性能提高5倍或更多

- 近似于过程中模糊作业的优点，同时仍然保持了模糊程序与目标二进制之间的非常强大的隔离。

## 5.11 并行

并行化机制依赖于定期检查在其他CPU内核或远程机器上独立运行的实例产生的队列，然后选择性地拉入测试用例，当在本地尝试时，会产生手头尚未看到的行为。

这允许在fuzzer设置中具有极大的灵活性，包括对通常数据格式的不同解析器运行同步的实例，通常具有协同效应。

有关此设计的更多信息，请参阅parallel\_fuzzing.txt。

## 5.12 二进制instumentation

黑盒，仅二进制目标instrumentation用的QEMU的单独建造的版本“用户模拟”模式的帮助下完成的。 这也允许执行跨体系结构代码 - 例如x86上的ARM二进制文件。

QEMU使用基本块作为翻译单位; 仪器在其上执行，并使用大致类似于编译时钩子的模型：

if (block\_address > elf\_text\_start && block\_address < elf\_text\_end) {

cur\_location = (block\_address >> 4) ^ (block\_address << 8); shared\_mem[cur\_location ^ prev\_location]++;

prev\_location = cur\_location >> 1;

}

第二行中基于移位和XOR的加扰用于掩盖指令对齐的影响。

二进制翻译器如QEMU，DynamoRIO和PIN的启动相当缓慢; 为了应对这一点，QEMU模式利用与用于编译器设备代码类似的叉式服务器，有效地产生在\_start暂停的已初始化过程的副本.

新的基本块的首次翻译也导致大量的等待时间。 为了消除这个问题，通过在运行的仿真器和父进程之间提供一个通道来扩展AFL fork服务器。 该通道用于通知父进程关于任何新遇到的块的地址，并将其添加到将被复制以用于将来的子进程的转换高速缓存。

作为这两个优化的结果，QEMU模式的开销大约是2-5倍，与PIN的100x +相比。

## 5.13 afl 分析工具

文件格式分析器是前面讨论的最小化算法的简单扩展; 而不是尝试删除无操作块，该工具执行一系列步行字节翻转，然后注释输入文件中的字节运行。它使用以下分类方案：

- “No-op blocks” - 位翻转导致控制流没有明显变化的段。常见的例子可能是注释部分，位图文件中的像素数据等。

   - “Superficial content” - 一些段，但不是全部，位片会产生一些控制流程的变化。示例可以包括富文档中的字符串（例如，XML，RTF）。

   - “Critical stream” - 所有比特翻转以不同但相关的方式改变控制流的字节序列。这可能是压缩数据，

    非原子比较的关键字或魔数等

   - “Suspected length field” - 小的原子整数，当以任何方式触摸时，会导致对程序控制流的一致性更改，提示失败的长度检查。

   - “Suspected cksum or magic int” - 一个整数，其行为与长度字段类似，但具有使得长度解释不可能的数值。这是提示校验和或其他“魔数”整数。

   - “Suspected checksummed block” - 一个长的数据块，其中任何更改总是触发相同的新执行路径。可能是由于失败造成的 在进行任何后续解析之前进行校验和或类似的完整性检查。

   - “Magic value section” - 一个通用令牌，其中更改导致先前概述的二进制行为的类型，但不符合任何其他条件。可能是一个原子比较的关键字。

# 6 AFL IOS移植

## 6.1 AFL IOS

本文档参照

==============================================================================



==============================================================================

简要介绍了AFL在IOS的应用与移植以及实例.

应用场景:

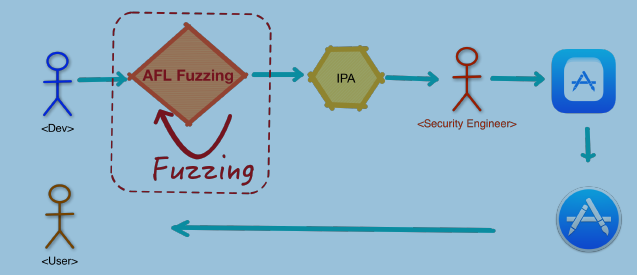
* 有源码的话,直接修改交叉编译链,用AFL替换编译器
* AFL FUZZ库 主要为以下4类常见:

1. Parser: JSON, XML,DLS Parser
2. Video& Audio Encoder and Decoder
3. Image Encoder and Decoder
4. Archive related libraries

## 6.1 AFL 应用到iOS

如下流程简易图

## 6.2 AFL build



按照README进行编译.

## 6.3 AFL 移植修改

修改对应的项目,将CC CXX改成afl-clang-fast afl-clang-fast++

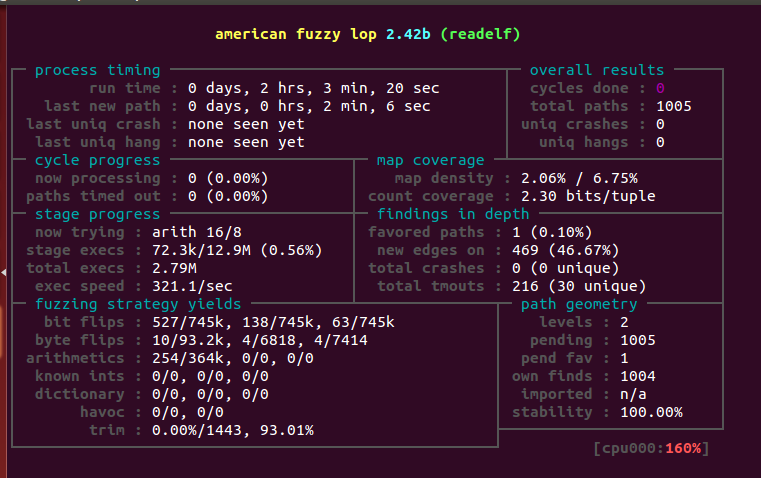
# 7 AFL 实验

## 7.1 AFL Unix

<http://lcamtuf.coredump.cx/afl/>

编译后对 命令行 使用afl-gcc afl-gcc++ 替换编译，

然后运行一个监控界面



## 7.2 AFL 源码ArchiveALL

Github: https://github.com/songfei/ArchiveALL

参照库中PPT

## 7.3 AFL 其他源码 与库

Github: https://github.com/Proteas/fuzzers\_based\_on\_afl

参照库中PPT

# 附录.