Reading Notes

cxl

2020/05/30

目录

1 SAVIOR: Towards Bug-Driven Hybrid Testing @S&P'20

1.1 Background/Problems

混合测试结合了模糊测试和混合执行。模糊测试可以测试容易到达的区域，混合执行可以探索复杂的区域。混合测试结合这两部分可以很好的探索程序的状态空间。但是现在很多的 fuzzer 都是使用代码覆盖率作为反馈，以代码覆盖率为反馈其实就是将程序空间的所有部分同等化对待，但是实际上有的代码是没有 bug 的，这种同等化对待会使得我们浪费很多时间于没有 bug 的代码部分，所以我们需要将更多的注意力放在有 bug 的代码部分。这样可以使得漏洞检测的速度大大提升。

1.2 Methods/Techniques

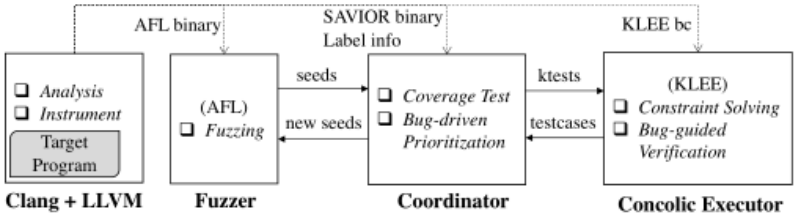

图 1: System architecture of SAVIOR.

本文作者提出了一种就 bug 驱动原则的混合模糊测试框架 SAVIOR，SAVIOR 的系统结构图见图1.SAVIOR 主要是通过 Bug-driven prioritization 和 Bug-guided verification 这两个部分来实现的。

Bug-driven prioritization:SAVIOR 初始时会对测试程序进行静态分析获得每个分支可到达每个基本块的集合，并且会利用 UBSan 工具为测试程序生成潜在 bug 标记，并对这些标记进行合理的筛减。在每个种子运行时，会根据种子的未探索的分支，以及这些分支所包含的潜在 bug 标记和这些边支约束求解的难度来计算这些种子的分数，从而给定种子的优先级。种子的优先级越高，即可以说明这个种子含有潜在 bug 越多，就会优先被混合执行来执行，这样可以使得时间更多的花费在有效的混合执行上，从而提高 bug 检测的

效率。

Bug-guided verification: SAVIOR 除了使用 bug 驱动的优先级来加速漏洞检测之外，还会沿着混合执行遍历的路径来验证标记的漏洞。特别在有些情况，程序的一个路径存在 bug，但是即使使用混合执行执行执行这条路径都无法触发这个 bug。因为这个 bug 可能存在潜在的约束，混合执行无法得知，例如本文中内存泄露的例子。SAVIOR 在为每个潜在 bug 添加标记之外还会添加相应的约束，如果 SAVIOR 可以解算约束，那么构造一个输入作为证明这个 bug，否则这个 bug 是不存在的。

1.3 Results/Evaluation

通过实验的评估，我们可以发现基于 bug 驱动的混合测试是优于传统的混合测试。SAVIOR 可以触发 481 次违规，其中 243 为真正的 bug。平均而言，SAVIOR 检测漏洞的速度快于 DRILLER 的 43.4% 和快于 QSYM 的 44.3%，结果分别的多发现了 88 个和 76 个 bug。

1.4 Limitations/Future work

1. Over-approximation in Vulnerability Labeling: SAVIOR 使用合理的算法来标记漏洞，但是可能由于 Over-approximation 导致许多假阳性标记被引入。解决这个问题可以采用更精确的静态分析来进行更细粒度的标记修剪。

2. Prediction in Vulnerability Detection：混合执行到达潜在 bug 的位置时，SAVIOR 会提取潜在 bug 标记的保护谓词，如果该保护谓词和当前路径约束条件相矛盾时，SAVIOR 会停止探索这个潜在的 bug，这样可以节约一定的时间。但是我们可以使用以前执行运行的信息来预测执行路径能否触发漏洞。本文作者为了实现这一点，使用了一种方法，该方法将标记点的路径约束向后总结到探索路径的前一个点，利用的核心技术是 the weakest precondition。

2 SLF: Fuzzing without V alid Seed Inputs @ICSE'19

2.1 Background/Problems

模糊测试是通过初始种子进行突变生成大量输入来测试程序，如果没有有效的种子，fuzzer 的性能会显著的下降。虽然符号执行可以从头开始生成种子的输入，但是它们有很多的限制，使得它们在现实中的复杂软件效率很低。所以生成有效的种子对于 fuzzing 是非常有必要的，本文提出一种不依赖源码和复杂程序分析的有效种子生成的 fuzzer。

2.2 Methods/Techniques

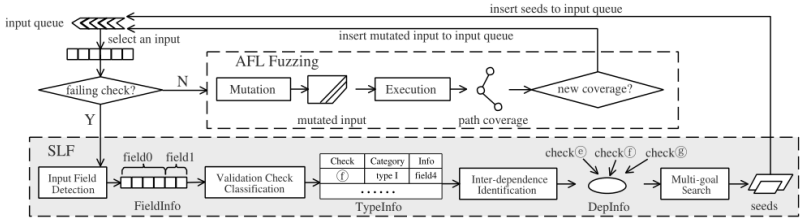

Fig. 3: Overview of SLF.

图 2: Overview of SLF.

SLF 的实现来自于两个重要的观察结果。第一，输入验证检查可以分为许多类别，每个类别对应特定的输入关系，并需要独特的变异策略。第二，可以通过预定义的格式中改变输入字节并观察位次的变化来有效检测类别。

SLF 的整体过程是首先随机生成 4 字节输入，通过执行输入来分析遇到的检查和错误信息来识别检查失败执行，如果没有找到验证失败，则当前种子良好，SLF 切换到 AFL，否则通过 SLF 生成种子方法来生成良好的种子。SLF 的整体过程图见图2。

SLF 生成良好种子主要由 4 个部分实现，第一个是 Input Field Detection，它将输入字节分组到字段，这允许在字段级别而不是字节级别上执行变异。第二个是 Input Validation Check Classification，它检测当前执行中的验证检查的类型。第三个是 inter- dependence identification，它标识前面的所有检查与当前执行失败的检查具有相

互依赖关系。第四个是 mutation algorithm ，它一个基于多目标梯度的搜索算法执行相应的突变，以满足所有相互依赖的检查。

Input Field Detection: 首先执行目标程序并收集检查信息，然后逐个字节的翻转并获得相应的检测信息，然后将相邻字节且检查信息相同的视为一组。

Input Validation Check Classification: 由于每种检查类型的不同特性，本文定义了三种检查策略来检查他们。具体是迭代各个输入字段，并尝试这三种检查策略，通过判断检查的谓词变量的更改是否预定类型的特征相匹配，如果匹配便将其设置为其类型。

inter-dependence identification: 根据每个检查的类型信息，我们关联那些谓词受到相同字段影响的检查。

mutation algorithm: 它会利用失败检查和失败检查的相依赖的检查的梯度来进行梯度搜索突变算法，并且该突变会使用两种方式针对两种不同的情况。

2.3 Results/Evaluation

我们对 SLF 评估了现实中 20 个真实项目，这些项目分别来自一些其他 fuzzer 的基准和谷歌的 fuzzer 测试套件，实验证明 SLF 比现有 fuzzer（AFL 和 AFLFast），符号执行引擎（KLEE 和 S2E）和混合模糊测试（Driller）效率都高。并且它可以生成比其他技术多 7.3 倍的有效种子输入，平均覆盖比其他技术多 3.5 倍的路径。

2.4 Limitations/Future work

1. SLF 不能处理与输入字段有复杂控制依赖关系的检查，这些复杂依赖关系的检查只能用常规的模糊测试方法来解决（例如突变）。

2. SLF 在字节级标识输入字段，因此无法推断包含位级字段检查。

3 Angora: Efficient Fuzzing by Principled Search @S&P'19

Xianglin Cheng, 2020-06-06

3.1 Background/Problems

fuzzing 是一种寻找软件 bug 的流行技术。但是最先进的 fuzzers 性能仍有很多需要改进的地方。基于符号执行的模糊测试可以产生高质量的输入，但是速度较慢；基于突变的模糊测试难以生产高质量输入。本文为了解决以上问题，实现了一种新型的基于突变的 fuzzer，它可以在不使用符号执行的情况下解决路径约束来增加分支覆盖率，从而实现 fuzzer 的更有效。

3.2 Methods/Techniques

Angora 的实现主要分为 5 个小部分，分别是用上文敏感方式进行分支覆盖，字节级别的污点分析，基于梯度下降的搜索策略，数据类型长度和是否带符号的判断和输入长度的探索。

上文敏感方式进行分支覆盖：AFL 使用的是上下文不敏感的分支覆盖方式，但是这种方式不能识别相同分支潜在的不同内部状态，所以在 AFL 的基础上添加了上下文特征。并且为了避免产生过多的独立分支，Angora 使用了异或型的 hash 函数对栈进行 hash。

字节级别的污点分析：单独跟踪每个字节的开销是巨大的。故 Angora 一旦在一个输入上运行污染跟踪，就可以记录哪些字节偏移量流到了每个条件语句中，在以后突变输入的执行便不需要进行污点分析，从而大大降低污点分析的开销。Angora 将每个变量在输入中的字节偏移作为污染标签，并且 Angora 提出了一种新型的数据结构来解决以往保存污染标签数据结构开销大的问题。这个数据结构是一个索引为污染标签且值为二叉树节点的表（二叉树是位向量到污染标签的映射），并且提出了三条剪枝策略使得性能开销再次下降。

基于梯度下降的搜索策略：Angora 把 fuzz 中的变异问题视作一个搜索问题，并应用机器学习中的搜索算法。我们将执行一个分支的条件语句视作一个黑盒函数 f(x) 的约束，x 是流入这个约束的输入

值 f ()。并且将所有的条件表达式转化为三种约束。为了满足这些约束，我们需要求 f(x) 的最小值，这里使用梯度下降来达到这个目的。

数据类型长度和是否带符号的判断：为了节约梯度下降的求解时间。我们需要判断数据类型长度和是否带符号，数据类型的长度在污点分析阶段已经判断过了，而类型（是否带符号）可以通过语义判断。

输入长度的探索：Angora 只有当增加长度能得到新分支时，才增加长度，也就是使得读取长度尽可能满足程序的需要。具体做法是，对于类似 read 的函数，根据它的读入长度是否会影响条件语句的结果来决定扩大长度与否。

3.3 Results/Evaluation

实验评估表明，Angora 的表现原因超过了最先进的 fuzzer。在 LAVA-M 的数据集下，Angora 可以发现更多的 bug，特别是在 who 中，Angora 发现 bug 的数量是第二好的 fuzzer Steelix 发现 bug 的 8 倍。并且 Angora 还发现了 103 个 LAVA 作者注射但不能触发的 bug。我们还在 8 个流行的、成熟的开源程序上测试了 Angora，Angora 在 file、jhead、nm、objdump 和 size 中分别发现了 6,52,29,40 和 48 个新 bug。

4 Different is Good: Detecting the Use of Uninitialized Variables through Differential Replay @CCS'19

Xianglin Cheng, 2020-06-08

4.1 Background/Problems

使用未初始化变量会带来一系列的问题，内核中使用未初始化变量可能导致内核信息泄露。目前检测这样的问题通常会对程序的堆栈区进行污点分析，但是此分析过程的开销会很大。目前最先进的内存泄露漏洞检测工具 Bochspwn Reloaded 检测该类问题使用的是对一些特定的指令进行污点分析，虽然这样解决了一定的开销问题，但是这样会产生一定的误报。本文提出了一种差异重放和符号污点分析相结合的方法来解决这个问题。

4.2 Methods/Techniques

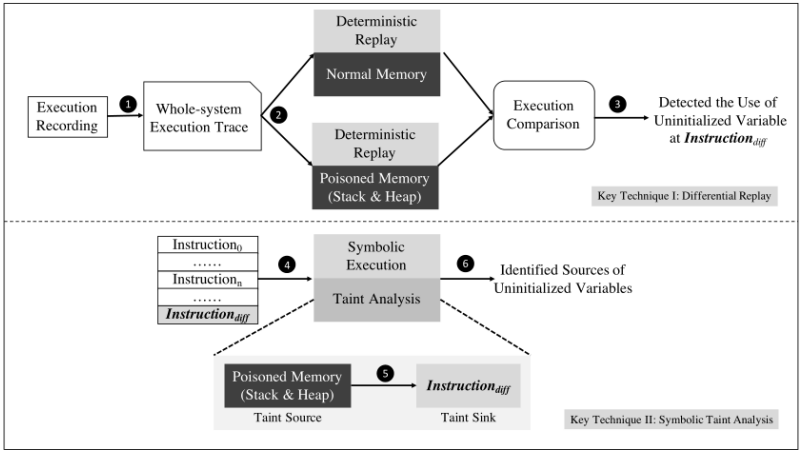

图 3: Overview of TimePlayer.

本文提出了一种结合差异重放和符号污点分析以来检测因使用未初始化变量而导致核信息泄漏的方法。其中，差异重放用来快速发现未初始化变量的使用，而不会执行耗时的全系统动态污染分析。符

号污点分析用来确认未初始化变量分配的位置。本文将该方法实现为工具 TimePlayer，TimePlayer 的整体过程图见图3.

Differential replay：首先使用一个全系统模拟器来记录操作系统内核和用户程序执行。然后启动两个实例，一个实例不改变任何内存，而另一个实例用特殊的值初始化内存。（具体来说，我们更改从内核堆栈和堆分配的内存的初始值）如果两个实例具有相同的程序状态，则设定的变量已经初始化，否则该变量没有被初始化。

Symbolic taint analysis：首先在检测到差异的指令回退 N 个栈帧，利用得到的 trace 进行符号污点分析，然后进行程序执行的跟踪，最后确定差异重放分配变量的确切位置，从而找到可能已经泄露的未初始化的内存区域。

4.3 Results/Evaluation

TimePlayer 成功检测出 Windows 7 和 Windows 10 内核 (x86/x64) 的 34 个新问题和 85 个已经修补过的问题 (其中一些是公开未知的)。34 个新问题中，有 17 个被微软确认为零日漏洞。并且 TimePlayer 在 47 小时左右可以检测到 34 个新问题，而目前最好的工具 Bochspwn Reloaded 在 66 小时左右只能检测到 7 个新问题，这可以体现 TimePlayer 的检测效率远远优于之前的工具。

4.4 Limitations/Future work

1. TimePlayer 可能会有误报（有些漏洞被遗漏），这是由于动态系统的性质导致的，TimePlayer 的有效性取决于检测过程的代码覆盖率。所以可以与模糊测试相结合使用，从而带来更高的代码覆盖率。

2. 差异重放的速率比普通重放的速率降低 22-24 倍，但是可以通过并行化重放优化来解决这个问题。

5 HyDiff: Hybrid Differential Software Analysis @ICSE'20

Xianglin Cheng, 2020-06-10

5.1 Background/Problems

差分软件分析有两种情况，一种情况是分析同一程序上执行不同输入的差异，另一种情况是分析多个程序或变体上执行相同输入的差异。差分软件分析具有很大的研究意义。例如回归错误检测，side-channels 分析和 DNN 鲁棒性的测试都可以作为差分软件分析的实例。但是由于差分软件分析需要同时对多个程序进行执行并且程序的差异表现在不同的方面上等问题使得目前的差分软件分析效果一般。本文首次提出将模糊测试和符号执行相结合应用到差分软件分析上，使得查分软件分析的效果大大提升。

5.2 Methods/Techniques

图4是 HyDiff 的整体过程图，该图可以分为三个部分，上部分为待测试的程序和初始种子作为输入，下部分是产生的测试输入按照不同的差异进行的分类。中间部分为 HyDiff 的核心部分，左部分为差分 Fuzzing 的工作流程，右部分为差分符号执行的工作流程。

Differential Fuzzing：本文使用的 fuzzing 与正常的 fuzzing 大致类似。与正常 fuzzing 最大的区别也是本文使用的差分 fuzzing 的核心便是对输入的评估，与正常输入的评估不同，本文的差分 fuzzing 使用除代码覆盖率之外的一些微分指标，分别为 output difference,the decision history difference, the cost difference, 和 the patch distance。这些指标只要其中一个揭示该输入可以触发新的行为，就会被保存下来。

Differential Symbolic Execution：HyDiff 的差分符号执行以两种模式运行，可以利用差分符号执行得到的输入进行混合符号执行，也可以使用传统的纯符号执行。这样可以使得差分模糊测试和差分符号执行并行的运行从而提高 HyDiff 的效率。Hydiff 的差分符号执行具体分为三步，首先是树扩展，再次是深入探索最后是生成合适的输入。

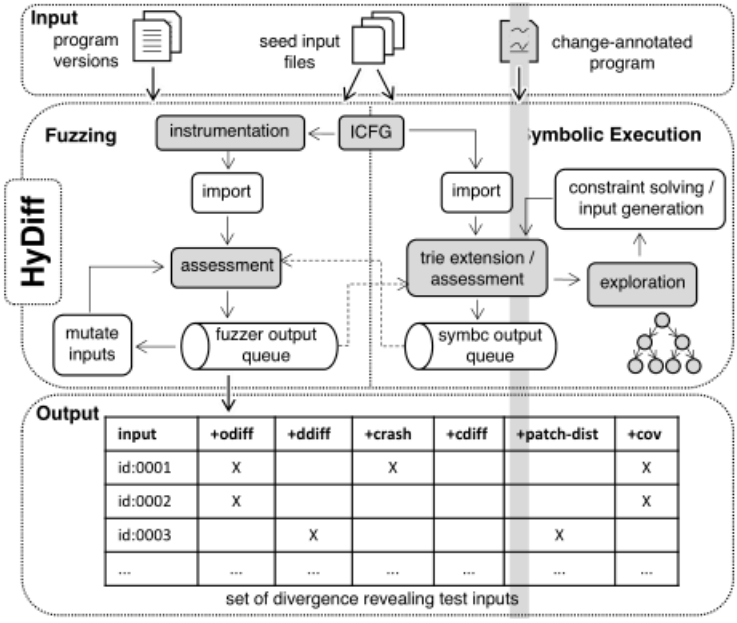

图 4: Overview of HyDiff.

5.3 Results/Evaluation

我们评估了 HyDiff 的三个应用:(1) 回归错误检测，(2) 侧通道漏洞检测，和 (3) DNNs 的鲁棒性分析。对于回归错误检测，我们利用 HyDiff 良好的检测了 TCAS，DEfect4J 基准和 Apache CLI 的回归错误。对于边通道漏洞检测，我们评估了 HyDiff 在以前关于边通道漏洞的研究以及模幂运算的实现。对于 DNNs 的鲁棒性分析，我们利用 HyDiff 评估了 MNIST 数据集。

5.4 Limitations/Future work

1. 需要人工来准备差异分析程序

2.HyDiff 需要更多的计算资源，因为它同时使用了符号执行和模糊测试。

3. 作者将来计划用其他的启发式方法扩展案例研究和实验，并且使用更多类型的 change annotations。

6 MOpt: Optimized Mutation Scheduling for Fuzzers @USEN Security'19

Xianglin Cheng, 2020-06-13

6.1 Background/Problems

基于变异的模糊测试是目前最流行的漏洞发现方法之一。其生成有趣测试用例的性能很大程度上取决于变异调度策略。然而现有的模糊测试通常使用固定的概率分布来选择变异算子，但是这在寻找程序上的漏洞效率很低。这往往是因为这种固定的变异策略有几点不足（以 AFL 的固定变异策略说明），分别是没考虑不同变异操作在一个目标程序上的有效性不同，其次是没考虑变异操作在不同目标程序上的有效性不同，再次是 AFL 花费大量时间在确定性变异阶段，而 havoc 阶段往往对于生成有趣种子更有效，最后是 AFL 花费大量时间在无效的变异操作上。所以本文提出了一种采用定制的粒子群优化算法（PSO）的新型突变调度算，使得基于突变的模糊测试效率更高。

6.2 Methods/Techniques

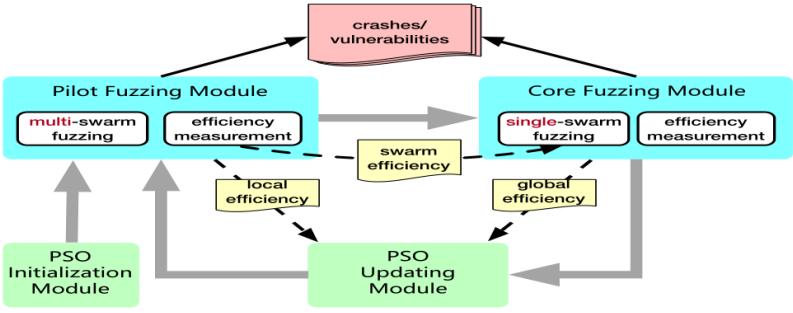

图 5: The workflow of MOPT.

本文通过粒子群算法（PSO）进行调整，从而可以动态的评估变异操作的有效性，调整变异操作的概率分布。实际上是根据本地的最优概率以及全局的最优概率更新每个粒子的概率，以及最后选择最

优的群来得到最优的变异操作的概率分布，使得 fuzzing 可以发现更多的高质量的测试用例。

图5是 MOPT 的工作流程图。MOPT 是由四个核心模块组成，即 PSO 初始化和更新模块，以及 pliot 模糊测试和 core 模糊测试模块。PSO 初始化模块的运行用于设置 PSO 算法的初始参数。其他三个模块组成一个迭代循环，共同工作，不断模糊目标程序。每迭代一次，都会使用粒子群算法更新变异操作的概率。pilot 模糊测试模块采用多个群来选择变异算子并进行模糊测试，从而评估每个粒子群的局部效率。并且 pilot 模糊测试模块还会对每个群的效率进行评估从而选择最有效的群。core 模糊测试模块利用其所探索的概率分布来调度变异算子，从而可以得到评估每个粒子的全局效率。利用该迭代回路，模糊器可以利用粒子群算法寻找最优的概率分布来选择变异算子，并逐步提高模糊效率。其中粒子 p（变异操作）的运动计算公式如图6。

$$
\begin{aligned}
v_{now}[S_i][P_j] \leftarrow & w \times v_{now}[S_i][P_j] \\
& + r \times (L_{best}[S_i][P_j] - x_{now}[S_i][P_j]) \\
& + r \times (G_{best}[P_j] \quad - x_{now}[S_i][P_j]).
\end{aligned}
$$

$$
x_{now}[S_i][P_j] \leftarrow x_{now}[S_i][P_j] + v_{now}[S_i][P_j].
$$

图 6: 粒子 p 的运动计算公式.

6.3 Results/Evaluation

本文将 MOPT 应用于几种最先进的模糊测试，包括 AFL、AFLFAST 和 Vuzzer，并在 13 个现实项目中进行了评估。在这 13 个现实的项目实验下，MOPT-AFL 共发现 112 个安全漏洞，其中 97 个以前未知的漏洞（其中 66 个经 CVE 确认）和 15 个已知的 CVE 漏洞。与 AFL 相比，MOPT-AFL 发现的漏洞多 170%，崩溃多 350%，程序路径多 100% 。MOPT AFLFAST 和 MOPT Vuzzer 在我们的数据集上也超过了它们的对手。从实验的结果可以证明 MOPT 具有合理

性、稳定性和低成本性。

6.4 Limitations/Future work

1. 本文作者计划研究更好的变异算子以进一步提高 MOPT 的有效性具有重要意义。

2. 本文作者计划构建一个更全面和更具代表性的基准数据集来系统地评估模糊器的性能。

7 TOFU: Target-Oriented FUzzers @arxiv'20

Xianglin Cheng, 2020-06-16

7.1 Background/Problems

模糊测试已经被证明是一种发现漏洞、发现安全漏洞和生成增加代码覆盖率的测试输入的强大方法。但是有时人们对面向目标的方法更感兴趣，即导向型模糊测试。以往的导向型模糊测试都是利用随机突变的方式对种子进行突变，这种突变方式的 fuzzer 在对程序的输入具有高度结构化的情况下效果是非常不好的。本文首先提出使用结构化突变应用到导向型模糊测试。不仅针对本身输入进行结构化突变，本文还结合使用了命令行结构化突变器，使得本文的模糊测试方法取得良好的效果。

7.2 Methods/Techniques

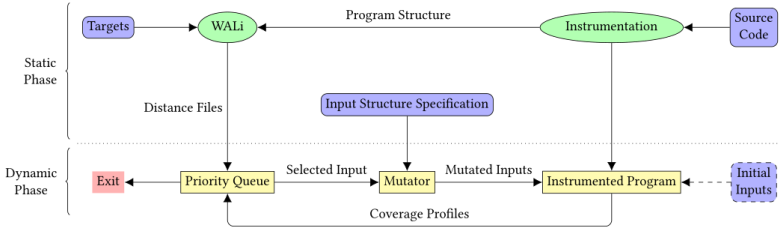

图 7: The workflow of TOFU.

图7是 TOFU 的总体工作流程。TOFU 分为静态阶段和动态阶段。TOFU 对程序进行插桩以便在程序运行后获得基本块的代码率。并且在这一阶段生成距离文件（记录程序中每个基本块到目标基本块的距离），同时在这一阶段还可以接收输入的结构规范，并生成动态阶段使用的结构化突变器。动态阶段通过计算种子的距离获得相应的分数添加到优先级，然后选择分数高的进行突变，进而进行循环过程。TOFU 的核心为 Gray-Box Fuzzing with Prioritization 和 Structured Mutation。

Gray-Box Fuzzing with Prioritization：TOFU 除了使用距离来作为优先级之外（距离越小优先级越高），还是用了两种机制来改进优先级。首先是利用一种机制来抑制一些“看起来很像”但已经考虑过的输入。通过使用一个字典来记录程序执行期间这些“看起来很像的输入”突变的次数，将该次数的反相关也作为分数的一部分来一直该输入。第二种机制是突变一个种子之前会将该种子的分数乘以 1.2 添加到优先队列中，从而保证以后的突变还能改变该输入。

Structured Mutation：Structured Mutation 可以分为 Protobuf-Based Structured Mutation 和 Command-Line-Language Structured Mutator 两部分。Protobuf-Based Structured Mutation 部分是使用谷歌的 libprotobuf-mutator 和能提供的能描述程序输入结构的标准文件来生成输入结构突变器。Command-Line-Language Structured Mutator 部分是利用谷歌的 libprotobuf-mutator 和命令行标志和选项的规范来生成命令行突变器。

7.3 Results/Evaluation

TOFU 距离计算的速度要远远快于 AFLGo 的距离计算速度，例如，对于 libxml2, TOFU 的距离计算阶段只需要 2 分钟左右，而 AFLGo 的距离计算阶段则需要 80 分钟左右。本文的实验证明 TOFU 到达目标的速率和到达目标的数量要大于 AFLGo，例如，在 xmllint 上，TOFU 比 AFLGo 快 28%，同时多达到 45% 的目标。并且本文将 TOFU 与不使用距离引导搜索的 TOFU，不使用输入结构知识的 TOFU 进行比较，实验证明距离引导搜索和输入结构知识对 TOFU 的性能有显著影响。

7.4 Limitations/Future work

1. 本文使用的距离仅用于计算优先级，本文的种子能量是由用户自定义的，自定义的种子能量不具有自适应性，应该适当考虑距离或其他的条件来获得自适应的种子能量。

2. 本文的机构化突变器是基于 protobuf 规范，所以他不能捕获有效输入的所有约束。虽然本文提出了使用附加语义来增强这些粗略规范，但是依然很难捕获有效输入的所有约束。