评论（7）

ConfigFuzz

1、背景

维护内核安全至关重要。每天不断有新的patch加入到内核中，这些patch可能隐藏着错误；另一方面，每天也不断有新的错误被用户发现并提交到社区中。如何发掘patch中的错误，并对已有的错误进行复现以供分析，成为了一项labor-intensive的工作。在这一背景下，DGF for OS Kernel就十分适合这一任务。

DGF(Directed Greybox Fuzzing) for OS Kernel以内核中某一目标点位（一个基本块）为驱动，通过对内核做轻量级分析，得到触发目标点位所需的部分依赖（约束），然后每次测试前利用它们筛选种子（seed selection），之后根据种子的性能对其进行排序(seed scheduling），从而引导测试向目标点位前进。

2、问题

由此可见，如何解决依赖问题，即及时有效找到足够多的依赖，并有效利用其引导测试，是影响DGF for OS Kernel性能的重要因素。

依赖分为显式依赖和隐式依赖，然而我们在实际使用中发现，隐式依赖问题仍然没有得到良好解决，从而造成DGF for OS Kernel仍然浪费大量的时间与计算资源，进行无效探索。

为什么解决隐式依赖困难?

- 找到所有隐式依赖是不可能的。理论上找到所有隐式依赖需要在内核上执行准确的数据流与控制流分析，①内核存在大量的间接调用，变量重名等复杂的调用方式，而现有工具的精度不够，没有办法精准地处理以上所有的情况；②内核源码越2400万行，内核镜像规模十分庞大，现有分析方法对其做general analysis的开销太大难。这影响了seed selection，没用的种子筛不掉。
- 迅速有效地正确执行构成隐式依赖的语句是困难的。种子正确执行目标点位（basic block）的约束（隐式依赖）太多，而搜索空间又太大。这影响了seed scheduling，让无效的种子做更多无效的mutate。

3、现有工作的问题

- 因此，为了提高seed selection的效率，现有工作通常做2方面的工作获取隐式依赖：
  - 先验搜索：在生成seed前，使用静态分析得到一个seed正确执行目标点位**可能**要满足什么依赖，然后在每轮mutate前，将不满足依赖的种子筛掉[SyzLego]。由于内核空间太大，隐式依赖类型多且复杂（如间接调用，共享变量，nested structures等），因此现有工作只聚焦一种类型，如SyzLego只关注间接调用这种类型，有很多其它类型的依赖被忽略。
  - 后验搜索：在生成seed后，如果发现某一seed恰好覆盖了正确的路径，则分析该seed之所以成功覆盖路径，是因为它满足了什么依赖，然后依此筛除种子[MoonShine, HFL]。这种方法会利用符号执行(concolic execution)、内存读写地址检查等方式，验证一个读语句和一个写语句确实构成依赖。然而这类方法在内核这种规模庞大的软件上执行时，会有巨大的开销。

  与先验搜索相比，后验搜索方式由于是在测试过程中进行，因此会影响测试的时间。虽然先验搜索由于精度问题会造成一定的假阳性，但是即便有种子应用了这些错误的规则，在后面的seed scheduling中也可以在一定程度上将它们筛除。所以我们选择用先验搜索的方式得到隐式依赖规则。

- 另一方面，为了有效引导seed scheduling，现有工作主要以2种指标衡量seed的性能以做排序（这两个指标是并列的）：
  - 覆盖新路径：如果seed覆盖了之前没有覆盖的路径，那么fuzzer会认为这个种子有探索的价值，因此将这个seed放入mutate队列中，基于Syzkaller的一系列DGF工作[SyzDirect, SyzLego]都采用这一指标来衡量seed性能。但是，因为这种方式没有对seed的探索方向做出限制，导致fuzzer浪费过多计算资源探索无用的路径，这些路径既没有离目标点位更近，也没有离与目标点位相依赖的点位更近；DAFL则提出利用DUG(Def-Use Graph)来限制DGF只探索与目标点位相关的路径，但是内核的数据流形式十分复杂，目前还无法精准地构建内核的完整DUG，且有着较大开销，因此这种方式难以在内核上应用。
  - seed到目标点位的距离：DGF不同于其它非引导的模糊测试，AFLGo首次提出以seed执行产生的trace是否距离目标点位更近（其计算方式为：计算CFG上，一个seed执行路径中的每一个基本块到目标点位的最小距离的调和平均值。），来衡量seed的性能。之后的DGF工作从计算方式[WindRanger, Hawkeye]对AFLGo做出改进，但其本质都是计算seed到目标点位的距离。但是，这些以到目标点位为导向的计算方式仍然使DGF经常get stuck in a rut（陷入死循环中，不论怎么探索，距离值都不再变化）。

  因为以上两种指标是正交的，因此，ConfigFuzz同时基于这两种类型的指标来设置种子性能的评判标准。新路径能确保种子会尝试探索内核空间，而不会只沿固定的路径前进；计算种子的距离则会保证种子不会漫无目的地探索，而是不断朝着目标点位前进。

4、Insight

由于内核的规模和复杂性，我们无法在内核上实现高效精准的数据流和控制流分析，所以完全依赖数据流/控制流的分析方式不可行，我们需要其它信息对搜索进行指导，在保证搜索精度的同时提高效率。

实际上，Linux遵循模块化的编程思想，会将功能相关的代码以某种形式组织在一起，而我们发现**构成隐式依赖的代码，在功能上大概率相关，**所以我们应该在相关功能的范围内搜索。

此外，我们发现配置项给内核源码在功能上做了准确的划分，因此能对隐式依赖的搜索提供指导作用，可以提高搜索的效率。具体来讲，内核配置项的特征体现在如下几个方面：

配置项的特征

- Define And Use（自己创建，自己使用，DAU）特性：如果配置项为内核添加了新的全局变量，或是在已有的结构体内添加了新成员，那么内核**一定会**对这些新添加的全局变量（或结构体成员）进行读写操作，且这些代码**一定**落在这个配置项（或相关配置项）的管辖范围内。
- 高占比：在最新版本的内核中，被配置项覆盖的代码占到66%左右，这说明在配置项覆盖的代码范围内寻找隐式依赖的成功率较高；

因此，我们认为配置项能为隐式依赖的搜索提供指导作用，主要包括以下两方面：

- 目标代码块被某配置项包裹时，依赖代码块会被其相关配置项（这个配置项，以及其父/子/兄弟节点）包裹；
- 目标代码块读写的内核对象（kernel object，即结构体，如task_struct, page, swap_info_struct等）里有成员被某配置项包裹时，依赖代码块在其相关配置项包裹的代码块**附近（lexically near）**。

5、解决方案

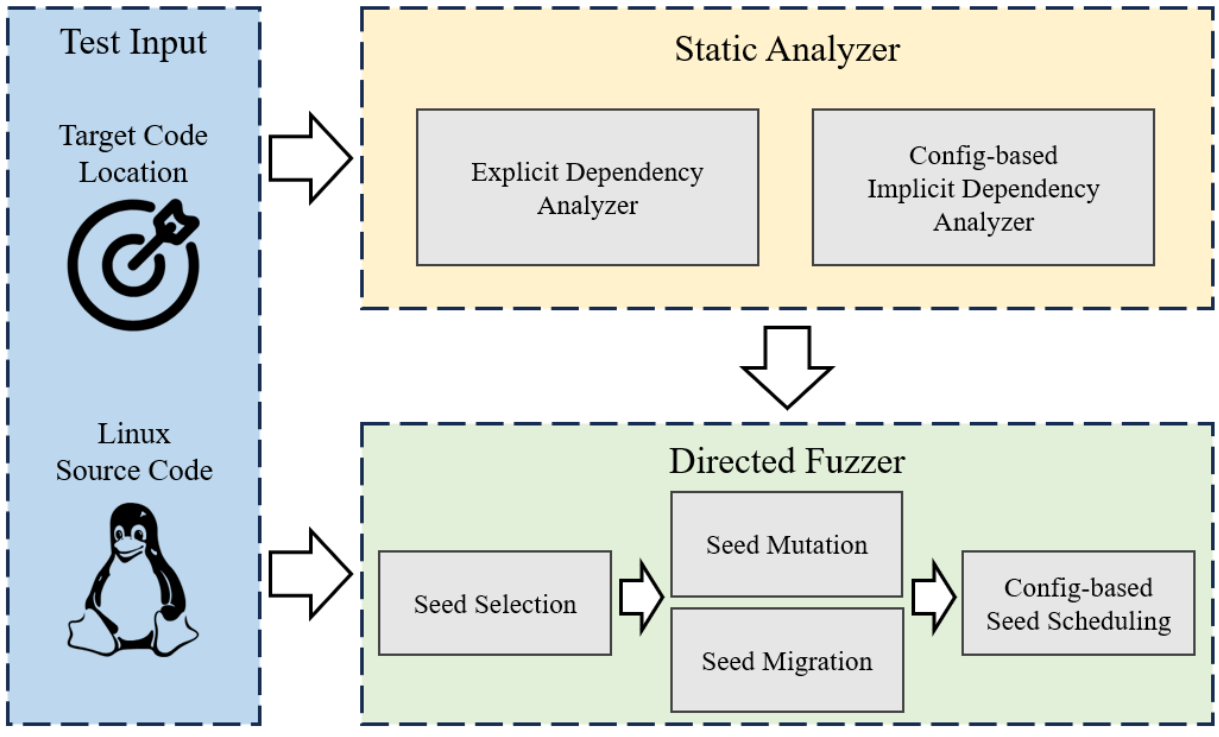
基于以上的Insight，我们提出了ConfigFuzz(Config-based Directed Gerybox Fuzzing)，一种利用配置项的全新DGF架构，其通过使用配置项提供的信息，来提高搜索隐式依赖的准确性与数量，从而提高测试抵达目标点位的速度，增加测试的成功率。

首先，它尝试获取目标点位所隶属的配置项，或其操作的内核对象成员所隶属的配置项，然后在这些配置项及其相关配置项（在配置项树中的父节点/子节点/兄弟节点）管辖的代码里或附近，搜索并筛选出可能满足目标点位隐式依赖的写语句。在每一轮测试前，ConfigFuzz会利用得到的隐式依赖规则对种子进行筛选；

其次，ConfigFuzz设置了一种全新的计算方式，通过计算种子到被相关配置项覆盖的代码块的距离，和覆盖代码块的比例来衡量种子的性能，从而在每一轮测试中对种子进行排序，使性能更好的种子获得更多的能量进行变异，从而提高目标点位的抵达速度。

6、Overview

ConfigFuzz的工作流如下图所示。ConfigFuzz接收一个包含用户指定待测试目标点位的Linux源码为输入，首先会调用静态分析器（Static Analyzer），分析正确抵达目标点位所需的显式依赖，和隐式依赖（Config-based Implicit Dependency Analyzer），生成一系列规则，包括种子生成规则，以及种子迁移规则。在测试阶段（Directed Fuzzer），测试器会将内核源码编译成内核镜像，利用静态分析器得到的规则生成模板，模板则会生成种子。在每一轮测试前，测试器利用规则对上一轮的种子进行筛选（Seed Selection），对剩下的种子进行变异（Seed Mutation），而需要跨架构执行的种子则利用规则进行迁移（Seed Migration）。种子执行后，测试器根据种子的性能对种子进行优先级排序（Config-based Seed Scheduling），优先级更高的种子获得更多的能量进行变异。测试将一轮一轮地不断进行，直到成功测试目标点位或超时为止。



7、挑战

然而，ConfigFuzz仍然面临以下三个挑战：

- **不能有效进行非相关配置项的搜索：**某对构成依赖的读代码和写代码所隶属（或在其附近）的两个配置项，虽然管辖的功能有联系（因为有隐式依赖了），但由于开发者设计存在的缺陷，它们在配置树上并不相关（即不为父子/兄弟关系）（例：CVE-2024-26796）。在这种情况下，因为我们无法直接通过其中一个配置项搜索到另一个配置项，从而导致搜索提前终止。

同功能种子在不同架构间的迁移：在架构…

赵瑞霖 1月16日 10:18

- **同功能种子在不同架构间的迁移：**在架构相关代码中，同一功能通常在多个架构下都存在，且有各自的实现方式，所以在当前架构下存在的隐式依赖，在另一架构下也很可能存在。因此我们应当通过轻量的方式对种子做修改，以实现不同架构间的迁移，同时保证种子功能不变，从而免去再次搜索隐式依赖带来的开销。然而，两个架构下的写语句未必都被配置项管辖，或所管辖的配置项不相同/不相关，从而导致完全基于配置项的搜索方式失效，也就不能有效实现架构迁移。
- **难以筛选有效的写语句：**尽管找到写某一变量的代码块集不难，但它们并不能都可以正确地写变量。这是由于许多写操作实际上是动态写值，无法在执行前确认变量会被写成什么。假设平均一条读语句对应的写语句有n条，每一条写语句的执行路径上又有一条读语句，与其它写语句构成隐式依赖，那么，假设一个种子里有m个隐式依赖，则我们要生成m^n条规则，假阳性过高，且容易造成指数爆炸。目前如何筛选写语句，除了逐一进行人工判断之外别无它法，这一点在Demy中得到了证实。

8、Protocol

洋葱搜索法

- 从目标点位开始，使用类似SyzDirect的方法，通过向上回溯CFG识别入口系统调用，和执行路径。每发现一个路径分支时，识别该分支判断代码块（是一个读代码块）是否可能构成隐式依赖：
  - 如果所需的变量值**是**当前入口系统调用直接设置的参数，则其不是隐式依赖。
  - 如果所需的变量值**不是**当前入口系统调用syscall使用的参数，则可能是隐式依赖，需要进一步分析。

  在路径上所有可能的隐式依赖点位都识别完成后，首先使用Healer的方法识别该入口系统调用的显式依赖。然后对于每个可能存在隐式依赖的代码块，进行**基于配置项的**依赖写语句识别。因为这个“基于配置项”的搜索方式，是依次从代码块本身隶属的配置项，到代码块读取的变量例的配置项，再到代码块附近的配置项，是逐层向外的过程，因此我们也称呼这个算法为“洋葱搜索法”。
- 为了在短时间内找到隐式依赖，需要对搜索空间做剪枝。因为配置项为代码做了功能上的划分。首先查看当前目标代码块是否在某配置项内部。

  如果目标代码**在**某配置项内部，则优先在该配置项及其相关配置项管辖的代码块中搜索，检查每个代码块是否将目标变量写成正确的值，如果是则二者可能构成隐式依赖。此外，如果一个代码块同时被多个配置项所包裹，则我们优先从最内层配置项开始搜索，然后逐渐向外搜索。
  - 如果目标代码块读的变量定义在当前配置项下，则只在当前配置项和子配置项内搜索；
  - 如果目标代码块读的变量定义在父配置项下，则在当前配置项，父配置项，子配置项和兄弟配置项中搜索。

  如果待搜索的配置项在各个子模块下(net, kernel, ipc, sound等一级目录)都有代码块管理，则只在目标代码块所在的子模块内搜索。

  如果目标代码**不在**任何配置项内部，则首先获取该代码块读的内核对象和其成员：
  - 如果该内核对象里有成员被配置项包裹，则在其相关配置项代码块附近搜索。
  - 如果没有，则看目标代码块附近是否有被配置项包裹的代码块，如果有则在其相关配置项代码块附近搜索。

  如果仍然没有结果，则进行配置项探索，将可能构成依赖但并不相关的配置项纳入之后的搜索范围。

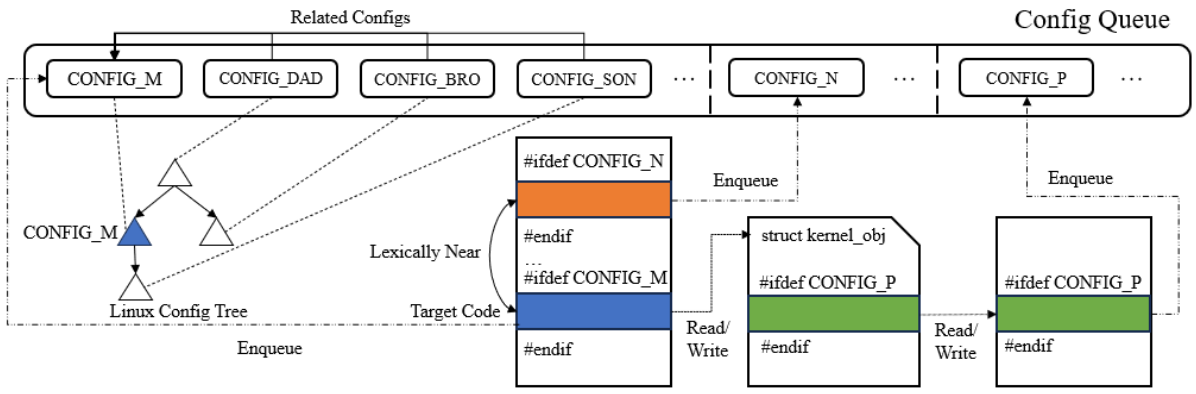
我们为每个配置项设置一个队列，该队列存储要搜索的配置项。初始化时，我们依次将配置项的父节点，子节点和兄弟节点入队。在每次搜索时，依次从队列头配置项管辖的代码

story 一致性，前面 Motivation 里限定测试的范围，比如测试架构相关代码是很重要的。

赵瑞霖 1月18日 22:36

怎么说？因为测试点位不一定在架构相关代码里，但是中间要执行的代码可能在相关代码里。（好好想想）

块中搜索隐式依赖，然后搜索过的配置项入队列尾。当搜索到构成隐式依赖代码块时，该过程结束，我们认为，下次再次搜索目标配置项的隐式依赖时，该配置项的搜索成功率较高，因此将该配置项留在队列头。如果目标代码同时被多个配置项包裹，则依次从最内层的配置项开始一层一层向外搜索。



但是，如果写语句在一个非相关配置项的管辖范围中，以上方法会失效。因此，当相关配置项所管辖的代码块都搜索完成后仍未找到时，我们最后执行一次探索操作，尝试是否能找到依赖，如果仍未找到，则搜索失败。

我们沿着内核的文件依赖关系图（File Dependency Graph），寻找其它使用了该读代码所读的变量的文件。如果这些文件里有被配置项管辖的代码块，则检查这些代码块里是否有满足依赖的写代码，如果有，证明这两个配置项在功能上存在相关性，因此将该配置项入队列。

如果两个不相关的配置项所管辖的代码却构成了隐式依赖，我们就认为这两个配置项在功能上相关，因此在配置项树上应该是相关的，所以这两个配置项可能是有错待修复的。那么我们如何识别出配置项之间的错误并汇报？

经过下面的**写语句筛选**操作后，我们检查剩余的所有可能正确写目标变量的写语句（all possible writing instructions that can correctly write the target variable）是否都在这个配置项里。如果是，那么这两个配置项可能需要修复(need fixing possibly)，这是因为如果想抵达目标点位，打开这两个配置项并执行里面的代码是唯一的方式。

在找出本该相关的配置项后，我们需要给出用户一种可能的修复方式：

- 互相依赖型：核心是“读依赖写”。
  - 如果两个配置项有着共同的祖先，说明二者管辖的功能在确实在同一范围内，可以安全地合并。
  - 如果两个配置项没有共同的祖先，检查二者管辖的代码块是否在同一目录下（最低一级目录），如果是说明二者管辖的功能在确实在同一范围内，可以安全地合并。

  先找到读的这个变量的定义语句所在的配置项。此时，在保留两个配置项原有依赖关系的同时，让写配置项成为这个配置项的子结点，读配置项再依赖于该写配置项。（如果变量就定义在读配置项里头，那就将读配置项的依赖关系加在写配置项上）
- 新增配置型：核心是“新加一个配置项”。
  - 两个配置项没有共同的祖先，且管辖的代码块不在同一目录下。这是由于两个配置项属于不同的功能模块，不能盲目地将他们连在一起，否则会发生混乱。

  此时可以新增一个配置项，该配置项的管辖范围为这个写代码块，它同时依赖于①读配置项所依赖的配置项，和②写配置项，然后读配置项依赖于该新配置项。

在完成以上操作后，下次再执行基于某配置项的搜索时，那就将修复的配置项当成相关配置项加入队列中。

另一方面，如果目标点位是个BUG()语句，那么如果某两个配置项的代码构成的依赖正好能帮助测试抵达目标点位，那么就说明这两个配置项应该相互排斥。

> 但是，如果写语句在一个非相关配置项的…

瑞霖 赵瑞霖 1月10日 09:26

得强调这确实是个错误（然后证明这个错误确实存在）即：读写依赖的语句必须在两个相关的配置项里出现，如果在不相关配置项里出现那就是错误

> 我们沿着内核的文件依赖关系图（File De…

瑞霖 赵瑞霖 1月9日 15:46 （编辑过）

这样跟“在同一文件或目录里的配置项里寻找”的方法相比，能保证正确性。同时，再也不需要什么传统的，基于控制流和数据流的静态分析方法了。

瑞霖 赵瑞霖 1月9日 19:42

（以 CVE-2024-26796 为例，写语句在 drivers/perf/riscv_pmu.c 中，而读语句在 drivers/perf/riscv_pmu_legacy.c 中。riscv_pmu_legacy.c 文件依赖于 riscv_pmu.h，而 riscv_pmu.c 也依赖于 riscv_pmu.h）

> 两个不相关的配置项所管辖的代码却有依…

瑞霖 赵瑞霖 1月9日 20:06

可能有人会问：那么两个本应相排斥但实际却没有的配置项，这种情况算不算要解决的错误？①首先，这种情况是错误。学名叫 variability bug。②这种情况下，写语句错误地写了变量。而错误地写变量后，内核究竟会不会报错是需要实际执行才能确认的。换言之，此时问题就变成了尽可能覆盖错误路径，这不是 DGF 的目标。③可能存在有些路

种子在架构间的迁移

我们希望通过少量的修改，就能让在某一架构下可执行的种子在另一架构下继续执行，从而免去了在另一种架构下再次搜索一遍隐式依赖所带来的开销。迁移后的种子和之前的同样能覆盖目标点位，因此这需要在原种子中包含的依赖（尤其是隐式依赖）在迁移后的种子中仍然存在。

内核通常会用抽象层来保证上层代码能够以相对一致的方式调用架构相关的功能。所以我们做以下assumption：

- 从抽象层向下（到硬件层面）的函数，各个架构均有着自己的实现方式，我们定义第一层架构相关的函数叫**锚函数（anchor function）**（例子：ftrace_make_call），内核中有相当数量的锚函数，从锚函数向下的的代码均使用着自己定义的变量；
- 同一锚函数在不同架构下的实现中，所使用的上层内核对象是相同的（例子：所有架构的ftrace_make_call的参数是一样的：struct dyn_ftrace *rec, unsigned long addr）；
- 架构外的代码如果和某一架构内的代码构成关联，数据流**必然**经过锚函数所使用的上层内核对象。
- 锚函数和在其执行路径下的写代码均确定后，锚函数执行路径上的大多数依赖在另一架构下仍然存在。只是因为新架构下，执行路径可能发生**部分**改变（从锚函数下去后才会变化），从而导致之前与锚函数构成依赖的系统调用的参数可能有少量变化，或者有新的系统调用与锚函数构成依赖。

基于以上假设，我们每次找到架构相关的写代码时，进行架构扩展，尝试构建架构代码迁移所需的规则数据库。以后每当提供给我们一个seed时，可以根据数据库中的规则直接做种子迁移操作。（这里目前只考虑架构外代码读，架构内代码写的依赖关系。这是因为如果是架构内代码是读时，这说明待测试的目标点位很可能是架构相关的代码，只能在当前架构下测试，所以此时不可以迁移，也就不考虑这种情况。）我们采取以下规则构建用于架构迁移的规则：

在CG上，从写代码块所在的函数开始向上回溯，当找到锚函数后，则再从其再向下搜索其可能的调用链（在目标架构的源码下搜索）（理由：assumption③），在CFG中搜索每个函数内部是否存在写同类型变量的代码块，如果存在，证明二者所在的函数在功能上可能存在较高的相似性。如果该代码块被某一配置项包裹，则将该配置项插入队列中（如果是CONFIG_X86等环境配置项[非功能配置项]，则不插入队列中，因为这类配置项管辖的代码块数量过多，不便于搜索）。

在确定写代码的位置后，继续向上检查该写代码是否存在显式和隐式依赖（①已有的依赖在该调用路径上是否仍满足；②是否存在新的依赖），以此构建正确调用写代码的可能规则（目标架构的规则和原架构的规则是一一对应的关系），以便在后续做种子迁移，规则的格式如下。

原架构的规则	目标架构的规则
	（在新架构下仍存在且无变化的依赖对）
<syscall_a, syscall_b, data_a_b>	<syscall_a, syscall_b, data_a_b>
<syscall_b, syscall_c, data_b_c>	<syscall_b, syscall_c, data_b_c>
...	...
	（在新架构下仍存在，但存在参数变化的依赖对）
<syscall_c, syscall_d, data_c_d>	<syscall_c, syscall_d, data'_c_d>
<syscall_d, syscall_e, data_d_e>	<syscall_d, syscall_e, data'_d_e>
...	...
	（仅在新架构下存在的依赖对）

径分支，如果读正确值就朝目标点位执行，如果读错误值就直接进入了 BUG()语句，这类情况因为 BUG()语句不是我们的目标点位，所以探索这些路径也不是 DGF 的目标。综上，我们不处理这种情况。

展开

瑞霖 赵瑞霖 1月9日 20:09

进一步讲③的问题。如果你真的想去探索抵达目标点位的路上有没有因为配置项不当导致的 bug，以上这种直接找 BUG()语句的方式根本就不深入，做了没什么意义；如果往深了做，那就变成如何尽可能高覆盖抵达目标点位路上的各种可能的路径，这又超出了 DGF 的范围。

两个配置项没有共同的祖先，且管辖的代…

瑞霖 赵瑞霖 1月9日 21:52

只要有一个代码块在同一目录之外那就不行。

<syscall_m, syscall_n, data_m_n> <syscall_n, syscall_dep, data_m_dep> ... <syscall_dep, syscall_anchor(调用锚函数的系统调用), data_dep_anchor(为成功调用锚函数，dep和sycall的参数范围)>	<syscall_m', syscall_n', data_m'_n'> <syscall_n', syscall_dep', data_n'_dep'> ... <syscall_dep', syscall_anchor(调用锚函数的系统调用), data_dep'_anchor'>

给定一个原架构下的种子，我们要利用上面**拓展**步骤中生成的规则进行迁移。迁移规则如下：

首先在原种子中确定syscall_anchor的位置，然后从其往前逐渐搜索在**原架构规则中出现**的系统调用syscall_p。如果它在目标架构下存在，检查它的参数是否符合要求**目标架构下**的规则data_p_anchor，对于符合要求的参数予以保留，不符合要求的参数则做mutate（如规定a!=b，此时a=b=1，则mutate a=2或b=3都可以），多余的参数去掉，缺少的参数根据规则要求随机生成。如果**目标架构**的规则中不存在这一系统调用，则将其删除；如果种子里缺少目标架构规则中的系统调用，则将其插入到seed中。不断重复这一过程，直到seed完全满足目标架构的所有规则为止。

最终替换的结果如下。可以看到规则以外的系统调用没有任何变化，这是因为其它的语句和这些语句之间没有直接的依赖关系，他们对于syscall_anchor能否正确执行没有直接的影响。此外，因为只要保证写语句在前，读语句在后，那么隐式依赖就可以被满足，所以如果某一系统调用（读）和多个系统调用（写）都有隐式依赖，只要保证写系统调用都在读系统调用前面即可。

原架构的种子	迁移后目标架构的种子
	（构成依赖但无需变化的系统调用）

syscall_a(a1, a2, ...);
syscall_b(b1, b2, ...);
...
	syscall_a(a1, a2, ...);
	syscall_b(b1, b2, ...);
	...

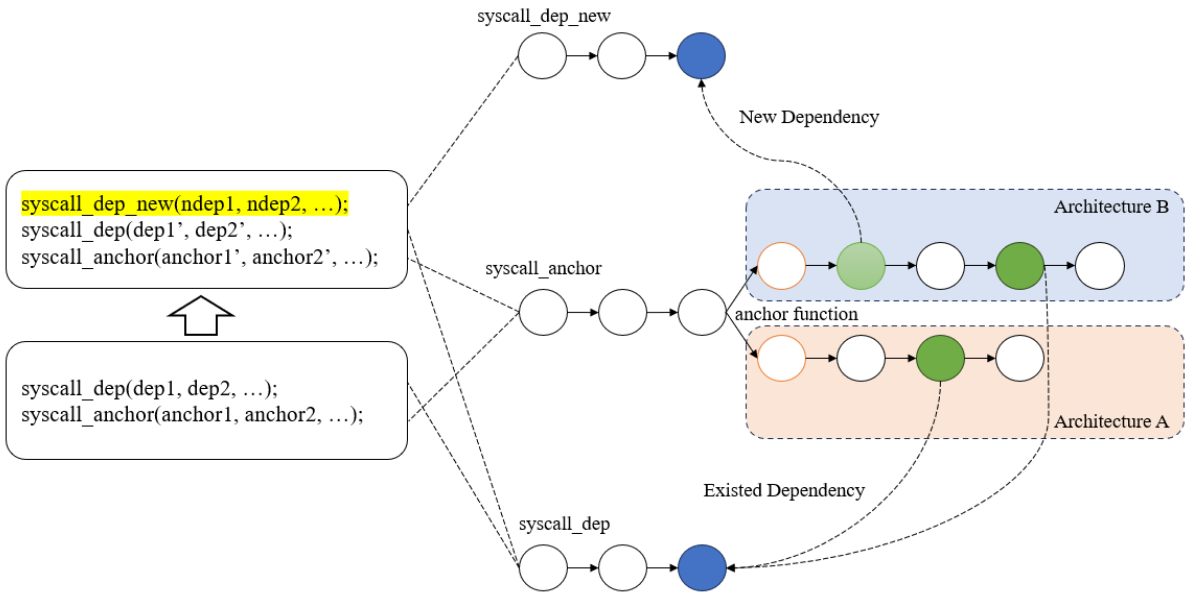
	（仍构成依赖，但参数变化的系统调用）

syscall_c(c1, c2, ...);
syscall_d(d1, d2, ...);
...
	syscall_c(c1', c2', ...);
	syscall_d(d1', d2', ...);
	...

	（新增的构成依赖的系统调用）

syscall_m(m1, m2, ...);
syscall_n(n1, n2, ...);
...
	syscall_m'(m1', m2', ...);
	syscall_n'(n1', n2', ...);
	...

syscall_dep(dep1, dep2, ...);
syscall_anchor(anchor1, anchor2, ...);
syscall_others(...);
...
	syscall_dep'(dep1', dep2', ...);
	syscall_anchor(anchor1', anchor2', ...);
	syscall_others(...);
	...



种子迁移的示例图。我们想将种子从A架构迁移到B架构，同时保证A架构下存在的隐式依赖在B架构下仍然存在（即两个种子执行的功能要保持相同）。首先我们从A架构的读语句向前回溯CG至锚函数为止，然后在B架构下寻找是否存在与A架构读语句功能相同的读语句。找到后，检查新的执行路径是否存在新的依赖，我们发现存在到syscall_dep_new的依赖。最后在迁移时，仍然存在依赖的syscall_dep保留，添加新依赖的系统调用syscall_dep_new。

写语句筛选

虽然我们并不能一次操作就准确地指出真正构成依赖的写语句，但是我们可以通过一系列操作不断缩小范围，这样生成正确覆盖隐式依赖种子的成功率会逐渐增高。为了实现这一目的，我们使用静态+动态结合的筛选方式：首先通过一些静态规则确认隐式依赖可能存在的范围，然后生成种子并监测相关内核对象值的变化，根据统计的结果进一步动态筛选规则。

一般而言，一个目标代码块读变量的方式有两种：①静态型和②动态型：

- 静态型：即var op C，C为一个常量，op为比较运算符。var是一个包含变量的表达式
- 动态型：即var1 op var2，var1和var2为包含变量的表达式。

另一方面，内核函数写一个变量的方式有两种：①静态型和②动态型，其中动态型占据了大部分[Demy]。

- 静态型：即var = C，C为一常量。
- 动态型：即var1 = var2 op C，var = n或var1 = var2 op n，op为运算符（加减乘除），n为一变量。

因此，首先检查读语句和写语句的类型，然后通过其读/写的值初步判断该写语句是否能作为candidate。

- 如果读语句和写语句都是静态型，直接判断是否满足条件即可，满足条件的直接进入规则名单，不再做进一步筛选。（大多是位运算或flags操作）
- 如果读语句是静态型，写语句是动态型。（大多是边界检查操作）则按如下规则筛选，满足条件的直接进入规则名单，不再做进一步筛选：
  - 读语句包含>/>=，选择包含加法和乘法操作的写代码（这是因为内核变量一般不包含负数，这些操作会让变量变得更大）。
  - 读语句包含</<=，选择包含减法和除法操作的写代码（这是因为这些操作会让变量变得更小）。
- 如果读语句是动态型，则无法直接判断，采用如下策略：

如果目标代码在一个配置项里，则保留同配置项下的写语句。这是因为根据Insight里的DAU特性，我们发现，如果某父配置项里定义了一些变量，供子配置项使用，则各个子配置项都会更有可能(more likely)将这些变量写成各自特定的值，以供各自功能所需。

大部分

瑞霖 2024年12月27日

67%左右

如果目标代码同时被多个配置项包裹，则保留被最多数量的配置项(在例子中，是A&B&C > B&C > C)包裹的写语句。

在通过初筛后，我们使用eBPF在读语句剩下的写语句中插入钩子函数，然后每一轮测试时记录①写语句对目标变量所写的值的范围；②读语句所需的值的范围。如果二者之间有交集，说明该写语句有概率使读语句正确触发。我们在每次seed scheduling的时候给那些有交集的seed更多的能量进行变异。

9、限制

死代码

尽管静态分析能分析出触发目标点位的可能路径，但这并不意味着实际上这些路径可以执行。这是因为由于开发人员的疏忽，内核中存在死代码，这些死代码在任何情况下都不会被执行，但却在CFG中存在。我们还无法做到在实际测试前就判断出一个路径下的代码是不是死代码。

多线程

某些写语句的正确执行需要内核中的某些线程处在特殊的状态，如死锁，竞争等。而这些状态的生成需要十分复杂的设计，且具有较大的偶然性。因此即使这些写语句在测试中能够正确执行，也并不意味着我们能立刻用seed对错误进行复现。我们希望未来能够深入研究多线程模糊测试，以尝试在某种程度上解决这一问题（We try to partially solve this problem in future）。

seed scheduling

瑞霖 赵瑞霖 1月7日 16:29

这些规则在 seed selection 时有用吗?