Driller：通过可选择的符号执行扩展模糊测试

目录

摘要

内存损坏漏洞是软件中始终存在的风险，攻击者可以利用其获得未经授权的权限访问机密信息。随着产品对敏感数据的访问变得越来越普遍，潜在的可利用漏洞的系统也越来越多，导致越来越需要软件自动审查工具。DARPA 最近资助了一项奖金数以数百万美元竞赛，进一步研究自动化漏洞发现和修补，显示了这方面研究的重要性。现有的寻找潜在漏洞的技术包括静态的，动态的，和符号执行分析，各有其优缺点。这些系统用来触发缺陷的创建输入设计都有一个共同的限制，那就是它们只能找到浅层的漏洞，并且它们都努力尝试可执行文件更深的路径。

我们给出的 Driller，是一种混合的漏洞挖掘工具，它以一种互补的方式平衡了模糊测试（fuzzing）和可选择的符号执行（selective concolic execution），用来找到更深层的漏洞。低成本的模糊测试用来训练应用的隔区，符号执行测试用来生成输入，以满足复杂的用来分隔隔区的检查。通过结合两种技术的优势，我们减轻了他们的弱点，避免了符号执行原有的路径爆炸，和模糊测试的不完备性。Driller 使用可选择的符号执行，只去探索被模糊测试机认为感兴趣的路径，并为模糊测试机不能满足的条件生成输入。我们在 DARPA Cyber Grand Challenge 资格赛发布的 126 个应用程序上测试评估了 Driller，与资格赛最高得分的队相比，它在相同时间识别了相同数量的漏洞。

第一章 绪论

尽管针对安全漏洞在增强软件的韧性方面做了很大的努力，软件中的缺陷仍是普遍存在的。事实上，近年来安全漏洞的出现已经增加到了历史新高[28]。此外，尽管引入了内存损坏和执行重定向缓解技术，软件缺陷仍占去年发现的所有的漏洞的三分之一[14]。

这些漏洞以前被想要推动安全限制并揭露保护无效的独立黑客利用，然而在现代世界，已经转移到国家和网络犯罪分子，他们利用这些漏洞来获得战略优势或利益。此外，随着物联网的兴起，运行着潜在的易受攻击软件的设备数量激增，并且在这些设备运行的软件中越来越多的发现了漏洞[29]。

虽然许多漏洞是人工发现的，但手动分析并不是一种可扩展的漏洞评估方法。为了跟上必须审查漏洞的软件数量，必然需要一个自动化的方法。DARPA 最近通过赞助两项工作来支持这一目标：VET，一个致力于二进制固件分析开发技术的程序，以及 Cyber Grand Challenge（CGC），参与者设计和部署自动化漏洞扫描引擎，它将通过利用二进制软件漏洞相互竞争。DARPA 已经为 VET 和网络大挑战提供了数百万美元的研究资金和奖金，表明了其对开发一种可行的自动化二进制分析方法的强烈兴趣。

安全研究人员一直在积极地设计自动化漏洞分析系统，存在许多方法，分为三个主要类别：静态，动态和符号执行分析系统。这些方法具有不同的优点和缺点。静态分析系统可以提供可证明的保证，即静态分析系统可以确定地表明给定的二进制代码段是安全的。然而，这样的系统具有两个基本缺点：它们不精确，导致大量的错误的正面判断，并且它们不能提供“可操作的输入”（即可以触发检测到的漏洞的特定输入的示例）。动态分析系统，例如“模糊器”（fuzzers），监控应用程序的本地执行以识别缺陷。当检测到缺陷时，这些系统可以提供可操作的输入来触发它们。然而，这些系统需要“输入测试用例”来驱动执行。没有全面的测试用例集，这需要大量的人工来生成，这种系统的可用性是有限的。最后，符号执行（concolic execution）引擎利用程序解释和约束求解技术来生成输入，以探索二进制的状态空间，以尝试到达和触发缺陷。然而，因为这样的系统能够在二进制程序中触发大量的路径（即对于条件分支，它们通常创建使得分支被采取的输入和不被采取的输入），它们受“路径爆炸”约束，这极大限制了它们的可扩展性。

由于这些缺点，现代自动化分析系统产生的大多数用来触发缺陷的输入仅能表现软件中的浅层错误。在模糊器的情况下，这是因为模糊器随机地向应用产生新的输入，并且它们可能不能成功地通过输入处理代码。另一方面，符号执行引擎通常能够重新创建正确格式化的输入以通过输入处理代码，但是由于约束于路

径爆炸，限制了它们可以分析的代码的“深度”。 因此，位于应用的更深层逻辑中的缺陷往往被这些工具所遗漏，并且通常通过人类专家的人工分析来发现[3], [9], [13].。

通过模糊测试和符号执行找到的不用漏洞类型之间的差异，也可以通过应用程序处理用户输入的方式来查看。我们提出两种不同类别的用户输入：一般输入，其具有大范围的有效值（例如，用户的名称）和特定输入，其具有有限的有效值集合（例如，上述用户名称的散列值）。应用程序对特定输入的特定值的检查有效地将应用程序分割成由这样的检查分隔的隔区。模糊测试精通于在一个区域内探索一般输入的可能值，但是很难找出所需的精确值，以满足对特定输入的检查和驱动隔区之间的执行流程。另一方面，选择性符号执行精确地确定这种特定检查所需的值，并且如果路径爆炸问题被解决，则可以推动隔区之间的执行流程。

例如，考虑处理来自用户的命令的应用程序：应用程序读取命令名称，将其与命令列表进行比较，并将用户提供的参数传递给适当的命令处理器。在这种情况下，复杂检查将是命令名称的比较：模糊器随机变化的输入将有非常小的机会发送正确的输入。另一方面，符号执行引擎将非常适合于找到正确的命令名称，但可能在参数处理代码中遭受路径爆炸。一旦确定了正确的命令名称，模糊器就更适合于探索可以发送的不同命令参数，而不会遇到路径爆炸。

我们意识到这中场景可以结合两种分析技术，利用他们的优势，同时减轻他们的弱点。例如，模糊器可以用于探索应用的初始隔间，并且当其不能进一步深入时，可以利用符号执行引擎来引导它到下一个隔间。 一旦到达，模糊器可以再次接管，探索可以提供给新隔区的可能输入。当模糊器再次停止运行时，符号执行引擎可以恢复并将分析引导到下一个隔间。通过重复这样做，执行被驱动的越来越深入程序，限制了符号执行固有的路径爆炸，并且改善动态分析的不完备性。

在这种理念的引导下，我们创建了一个系统 Driller，结合遗传输入-突变模糊器和选择性的符号执行引擎，以识别二进制中的深层漏洞。 结合这两种技术，Driller 可以以可扩展的方式运行，并绕过输入测试用例的需要。在本文中，我们将描述 Driller 的设计和实施，并评估其在 DARPA Cyber Grand Challenge 的资格赛发布的 126 个应用程序上表现的性能。

Driller 不是第一个结合不同类型分析的工作。 然而，现有技术支持非常特定类型的漏洞（而 Driller 现在可以检测到可能导致程序崩溃的任何漏洞）[21],[25], 没有充分利用动态分析提供的功能（特别是 fuzzing）[19]，或受路径爆炸问题的影响[4], [8], [10], [20].。我们可以通过相比模糊测试或者符号执行单独来说，Driller 在二进制文件中可以识别更多的漏洞，并且与 Cyber Grand Challenge 资格赛获胜队在相同时间内在相同的数据集上发现相同数量的漏洞来证明我们的方

法的有效性，此外，我们展示额外的评估，以表明如果没有 Driller 的作用（即使用传统的模糊测试或符号执行方法）这是不可能的。

总体来说，本文做出以下工作：

- 我们提出一种新的方法来提高模糊测试的效率，通过利用选择性的符号执行到达更深的程序代码，同时通过使用模糊测试缓解路径爆炸提高了符号执行的可扩展性。
- 我们设计并实施了一个工具 Driller 来演示这种方法。
- 我们通过与 Cyber Grand Challenge 资格赛获胜队在同一数据集上识别相同数量的漏洞，展示 Driller 的有效性。

第二章 相关工作

Driller 是一个有引导白盒模糊器，它建立在最先进的模糊测试技术的基础上，添加符号执行以实现有效的漏洞挖掘。由于一些其他现有的漏洞挖掘工具也结合了多种技术，我们将使用这一部分来区分 Driller 和其他利用相关技术的解决方案。

2.1 有指导的模糊测试

模糊测试最初是作为测试 UNIX 系统程序的几个工具之一引入的[23]。从那时起，它已广泛用于应用的黑盒安全测试。然而，模糊测试缺乏引导 - 基于先前输入的随机突变产生新的输入，而不能控制应用中的哪个路径应该被定向。

为了更好地将模糊器指向特定类别的漏洞产生了引导式模糊测试的概念。例如，许多研究尝试通过选择性地选择最优测试用例来改进模糊测试，在目标二进制中包含的代码的感兴趣区域上进行提升[21], [25]。具体来说，Dowser [21]使用静态分析首先识别可能导致涉及缓冲区溢出的漏洞的代码区域。为了分析代码，Dowser 对可用的测试用例进行污点跟踪，以确定这些代码区域处理哪些输入字节，并对代码区域进行符号化的探索。不幸的是，Dowser 有两个缺点：需要测试用例到达包含内存损坏漏洞的代码区域，并且它只支持缓冲区溢出漏洞。与 Dowser 不同，Driller 支持任意漏洞规范（虽然当前实现的重点是导致崩溃的漏洞），并且不需要输入测试用例。此外，Dowser 仍然遭受符号执行的路径爆炸问题，而 Driller 通过使用模糊测试来缓解这个问题。

与 Dowser 类似，BuzzFuzz [17]对抽样输入测试用例应用污点跟踪，以发现哪些输入字节由系统定义的“攻击点”处理，通常是系统调用参数和库代码。与 BuzzFuzz 不同，Driller 不依赖于到达易受攻击代码的输入测试用例，也不依赖于系统定义的“攻击点”。

在改进模糊状态的另一个尝试中，Flayer [15]允许系统在目标应用程序中随意跳过复杂的检查。这允许系统在应用程序中进行更深地模糊逻辑，而不需要编制符合目标所要求格式的输入，减掉了花费在寻找防止崩溃的合法输入的时间。类似地，Taintscope 使用校验和检测算法从应用程序中删除校验和代码，有效地“修补”分支断言，这是很难通过突变方法满足的[30]。这使得模糊器能够处理特定类别的困难约束。然而，这两种方法中，Flayer 的方法需要大量人工指导，另一种需要人工来确定在崩溃分类期间的错误正面判断。Driller 不修改目标应用程序的任何代码，意味着发现的崩溃不需要深入搜索，此外 Driller 不需要人为干预，因为它试图使用其符号执行后端来发现良好形式的输入。

另一种方法是混合模糊测试，其中有限的符号探索被用来找到“边界节点”[26]。然后采用模糊测试来执行具有随机输入的程序，随机输入被预先约束以遵循通向边界节点的路径。此方法对于确保模糊输入在二进制程序执行的早期采用不同路径很有用，但是它不处理程序中分离隔区时更深入的复杂的检查。

2.2 白盒模糊测试

其他系统尝试将模糊与符号执行混合以获得最大代码覆盖[6]，[7]，[19]，[20]。这些方法倾向于通过符号执行由模糊引擎产生的输入，收集置于该输入上的符号约束，然后否定这些约束以产生将采取其他路径的输入来增加模糊。然而，这些工具不具有 Driller 的关键点，符号执行最好用于复原输入以驱动代码在应用程序隔区之间执行。没有这种关键点，符号执行的独特能力被浪费在隔间内创建分歧路径。这些工具本质上是以串行化方式工作的符号执行引擎，每次一个路径，因此，它们深深受到路径爆炸问题的影响。

Driller 在许多实现细节中类似，我们建议可以将主要的独特的路径发现装载到一个机械化的模糊引擎中。我们限制了我们高消耗的符号执行调用，以满足允许我们的模糊器进入另外的隔区。由于我们只使用符号执行来生成模糊器无法自己生成的基本转换块，所以符号执行引擎只处理可管理的输入数量。相反，上述工具使用符号执行重复地否定约束，缓慢地分析指数级增加的转换次数，其中大多数可以由模糊器更有效地分析。

2.3 符号执行

随着近年来计算能力的不断增加，动态符号执行已经普及。由 EXE 引入[5]，于 KLEE 完善[4]，并由 Mayhem [8]和 S2E [10]应用于二进制代码，符号执行引擎解释应用程序，使用符号变量的模拟用户输入，跟踪条件跳转产生的约束，并使用约束求解器创建输入以驱动程序代码按特定路径向下执行。虽然这些系统是强大的，但是它们存在一个根本问题：如果条件分支依赖于符号值，则通常满足已采用和未采用条件。 因此，状态必须分叉，并且必须探索两个路径。 这快速导致了众所周知的路径爆炸问题，这是符号执行技术的主要抑制者。

研究员们已经尝试了各种方法来减轻路径爆炸问题。 Veritesting [1]提出了一种优化的路径合并技术来减少正在执行的路径数量，Firmalice [29]执行大量的静态分析，并将符号执行限制在小的代码段，并且与欠约束的符号执行互换精度来支持伸缩性[16]，[27]。然而，这些技术要么不能缓解路径爆炸问题（Veritesting 延迟爆炸，但这种爆炸仍然最终发生）要么产生的输入不能直接可操作（例如，由 Firmalice 完成的切分产生的输入满足一种特定切片的约束，但是没有提供输

入以在首次到达代码）。

Driller 尝试通过将大多数路径探索任务装载到其模糊引擎来减轻这种情况，使用符号执行只是为了满足应用程序中保护各个隔区之间的复杂检查的过渡。

第三章 DRILLER 概述

Driller 设计背后的核心理念是，应用程序处理两种不同类型的用户输入：通用输入，表示可以有效的大范围值，以及特定输入，表示必须采用选定几个可能值之一的输入。从概念上讲，应用程序对后一种类型的输入的检查将应用程序拆分成各个隔区。执行流在隔间之间通过针对特定输入的检查来移动，而在隔间内，应用处理一般输入。这个概念在我们的实验数据集中的实际二进制的上下文中在 4-G 节中更深入地探讨。

Driller 通过将模糊测试的速度与符号执行的输入推理能力结合起来工作。这允许 Driller 快速探索部分二进制文件，不对用户输入施加复杂的要求，同时还能够处理纯粹的符号执行的可扩展性问题，以及模糊测试对特定输入的复杂检查问题。在本文中，我们定义“复杂”检查，为那些检查太具体，不能满足来自输入变异模糊器的输入。

Driller 由多个组件组成。 在这里，我们将总结这些组件，并提供 Driller 的操作的高级示例。 在本文的其余部分，我们将深入描述这些组件。

输入测试用例

Driller 可以在无输入测试情况下运行。然而，这种测试用例的存在可以通过预引导模糊器指向某些隔区来加快初始模糊测试步骤。

模糊测试

当 Driller 被调用时，它通过启动其模糊引擎开始。模糊引擎探索应用程序的第一个隔区，直到它达到特定输入上的第一个复杂检查。在这一点上，模糊引擎“卡住”并且不能识别输入以在程序中搜索新路径。

符号执行

当模糊引擎卡住时，Driller 调用它的选择性的符号执行组件。该组件分析应用程序，预约束用户输入与由先前的模糊测试步骤发现的唯一输入以防止路径爆炸。在跟踪由模糊器发现的输入之后，符号执行组件利用其约束求解引擎来识别将迫使执行先前未被探索的向下路径的输入。如果模糊引擎在卡住之前覆盖了先前的隔区，则这些路径代表到新隔区的执行流。

重复

一旦符号执行组件识别新的输入，它们被传递回到模糊组件，其继续在这些输入上突变以模糊新的隔区。Driller 继续在模糊测试和符号执行之间循环，直到发现应用程序的崩溃输入。

3.1 示例

为了阐明 Driller 背后的概念，我们在清单 1 中提供了一个示例。在此示例中，应用程序解析通过输入流接收的包含特定数的配置文件。如果接收到的数据包含语法错误或不正确的数字，程序退出。否则，控制流基于多个新隔区之间的输入切换，其中一些包含内存损坏缺陷。

Driller 通过调用其模糊引擎并模糊测试程序的第一个隔区来开始操作。这些模糊节点在图 1 的程序的控制流程图中以阴影示出。该模糊步骤探索第一隔区并且停留在第一个复杂检查 - 与特定数的比较。然后，Driller 执行符号执行引擎，以识别将驱动通过该检查的输入执行，进入其他程序隔区。对于该示例，由图 2 中示出了由符号执行组件发现的额外转换。

之后，Driller 再次进入其模糊阶段，模糊第二个隔间（初始化代码和对配置文件中的键的检查）。 第二模糊阶段的覆盖如图 3 所示。如图所示，除了默认值之外，模糊器找不到任何关键开关。当第二次模糊调用被阻塞时，Driller 利用其符号执行引擎来发现“crashstring”和“set_option”输入，如图 4 所示。前者直接导致二进制中的错误。

重要的是要注意，虽然既没有符号执行也没有模糊测试自己可以发现这个 bug，但 Driller 可以。在这个例子中有几个区域需要 Driller 的混合方法。解析例程和初始化代码具有大量的关于高状态数据的复杂控制流推理，这将导致路径爆炸，从而将符号执行减慢到毫无作用。此外，如前所述，魔数检查通过要求高度特定的输入，太小而不能在其搜索空间中合理地发现，导致传统的模糊测试方法成为不可能。阻碍模糊方法的其他常见技术包括使用散列函数来验证输入。出于这个原因，符号执行和模糊的组合具有获得更好结果的潜力。

```
1    int main(void){
2    configt* config = readconfig();
3    if(config == NULL){
4          puts("Configuration syntax error");
5          return1;
6    }
7    if(config->magic != MAGICNUMBER){
8          puts(”Bad magic number”);
9          return2;
10   }
11   initialize(config);
13   char* directive = config->directives[0];
14   if(!strcmp(directive, "crashstring")){
15         programbug();
16   }
```

```
17  elseif(!strcmp(directive, "setoption")){
18          setoption(config->directives[1]);
19  }
20  else{
21          default();
22  }
24  return0;
25  }
```

清单 1.需要模糊测试和符号执行一起工作的示例。

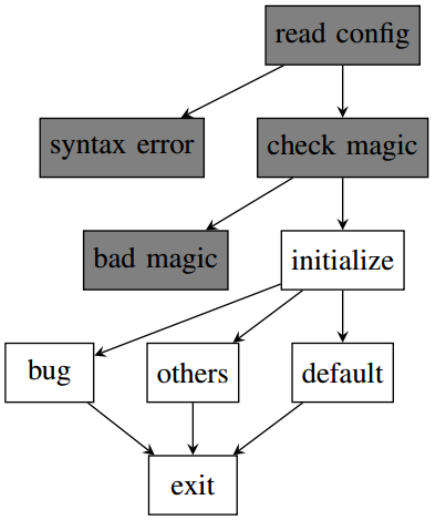


图 1.最初由模糊器找到的节点

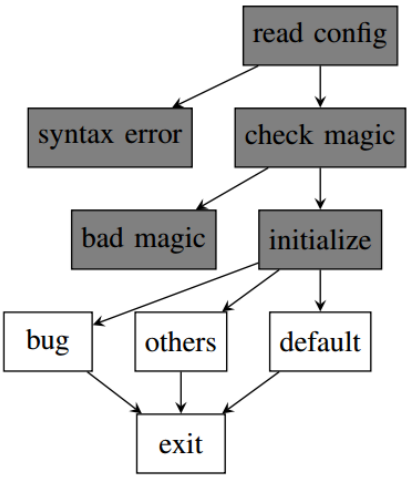


图 2.第一次调用符号执行所找到的节点

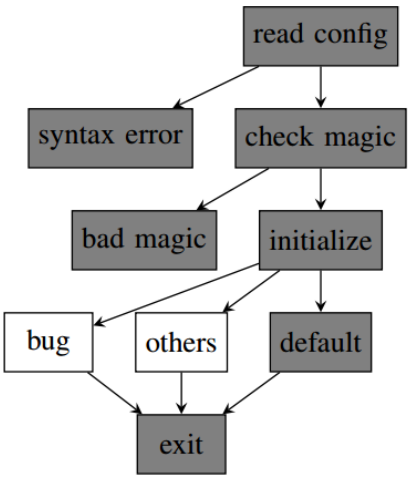


图 3.由模糊器发现的节点，补充了第一次 Driller 运行的结果。

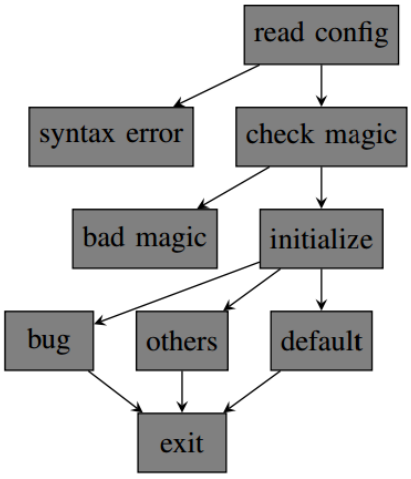


图 4.第二次调用符号执行所找到的节点。

第四章 模糊测试

模糊测试是一种使用大量输入执行应用程序的技术，检查这些输入是否导致应用程序崩溃。 为了保持执行的速度，模糊器是微创的 - 它们对底层应用执行最少的测试，并且大多从外部监视它。

近年来，对模糊引擎有许多改进。在本节中，我们将详细介绍与 Driller 的性能相关的改进。

为了实现 Driller，我们利用了一个流行的现成的模糊器，American Fuzzy Lop（AFL）[31]。 我们的改进主要涉及将模糊器与我们的符号执行引擎集成。 没有改变 AFL 的逻辑。AFL 依靠工具来对哪些路径是感兴趣的做出明智决定。该工具可以在编译时引入或通过修改 QEMU [2]，我们选择了一个 QEMU 后端，以消除对源代码可用性的依赖。虽然我们在本节讨论了 Driller 的模糊器组件（AFL）的重要特性，但我们不居功其发明或实现。

4.1 模糊器特性

现代的 fuzzer 实现了许多功能，以更好地识别崩溃输入。 在本节中，我们将列出和描述最重要的 AFL 功能，并提及 Driller 如何使用它们。

遗传模糊测试

AFL 通过遗传算法执行输入生成，根据遗传规则（转录，插入等）突变输入并通过适应度函数对它们排序。 对于 AFL，适应度函数基于唯一的代码覆盖率 - 即，触发与由其他输入触发的路径不同的执行路径。

状态转换跟踪

AFL 跟踪它从其输入看到的控制流转移集合，即源和目的基本块元组。 基于新控制流转移的发现，该输入被优先用于遗传算法中的“育种”，这意味着导致应用以不同方式执行的输入在未来输入的生成中获得高优先级。

循环“分桶”。

处理循环对于模糊引擎和符号执行引擎是一个复杂的问题。 为了帮助减少循环的路径空间的大小，执行以下启发式算法。 当 AFL 检测到路径包含循环的迭代时，触发辅助计算以确定该路径是否应当有资格进行育种。 AFL 确定执行的循环迭代的数量，并将其与能导致路径通过相同循环的先前输入进行比较。这些路径都根据它们的循环迭代计数的对数（即 1,2,4,8 等等）被放置到“桶”中。 来自每个桶的一条路径被考虑用于遗传算法中的育种。这样，与 N 个路径的普通方法相比，每个循环只需要考虑 log（N）个路径。

去随机化

程序随机化会干扰遗传模糊器对输入的评估 - 在给定随机种子下产生感兴趣路径的输入可能不会在另一输入下产生。 我们预先设置 AFL 的 QEMU 后端到一个特定的随机种子，以确保一致的执行。 之后，当发现崩溃输入时，我们使用我们的符号执行引擎来复原任何“挑战-响应”行为或依赖于随机性漏洞的缺陷。 例如，二进制中的“挑战响应”过程向用户回送随机数据，并且期望相同的数据回送回来。在不去除随机化的情况下，模糊组件可能每次都失败，并且探索非常少的路径。 如果随机性替换为常数，则程序每次都接受相同的输入，使得模糊器（或符号执行组件）自由地找到这一个值并且随后进一步探索。 在

发现崩溃后，随机性可以被替代进行符号化建模，如第 5-D4 节所述，并且可以相应地修补崩溃输入。

这些功能允许 AFL 快速发现通过应用程序的唯一路径，在应用程序的给定隔区内执行路径发现。 然而，模糊测试的限制是众所周知的。

4.2 模糊器限制

因为模糊器随机地变异输入，并且遗传模糊器改变通过二进制产生唯一路径的输入，所以它们能够快速发现处理“一般”输入的不同路径（即具有许多不同值的输入可以触发有意义的程序行为） 。 然而，生成用于通过应用中的复杂检查（即需要具有非常少的特定值之一的输入的检查）的“特定”输入对于模糊器是非常困难的。

清单 2 中的应用程序从用户读取一个值，并将其与特定值进行比较。 如果提供正确的值，应用程序将崩溃。然而，由于模糊测试的性质，模糊器满足该断言是不可能的。对于非机械化的模糊器（即选择随机值作为输入的模糊器），模糊器发现错误的可能性是 232 中的极小的 1。对于机械化的模糊器，该二进制的控制流布局将被发现为一个单一的路径。 没有优先考虑新的路径的能力，一个机械化的模糊器将被弱化到在现有路径上应用随机突变，这在本质上与非机械化的情况相同，具有相同的极小的机会成功。

```
1   int main(void)
2   {
3       intx;
4       read(0,&x,sizeof(x));
5
6       if(x==0x0123ABCD)
7           vulnerable();
8   }
```

清单 2. 对于模糊测试来说一个困难的程序。

4.3 转换到符号执行

Driller 旨在补充模糊测试的根本弱点，通过利用符号执行的力量来确定通过复杂检查所需的特定用户输入。 当模糊组件经过突变的预定量（与输入长度成正比）而没有识别新的状态转换时，我们认为它“卡住”。然后，Driller 检索模糊器在当前分区中认为“感兴趣”的输入，并在它们上调用符号执行引擎。

如果两个条件中的一个成立，则模糊器将输入识别为感兴趣：

1）输入使应用程序采取的路径是第一个触发一些状态转换的路径。

2）输入使应用程序采取的路径是第一个放置到唯一“循环桶”中的路径。

这些条件将控制传递给符号执行执行组件的输入数，降低到合理的数量，同时保持使得符号执行可以突变以到达应用程序中的下一个隔区的输入，传递时的高几率。

第五章 可选择的符号执行

当Driller确定模糊器不能找到另外的状态转换时，调用符号执行引擎。Driller使用符号执行的原因如下：模糊器在程序中没有找到新的状态转换的主要原因之一是模糊器无法生成特定的输入来满足代码中的复杂检查。符号执行引擎用于利用符号解释器将达到但不能满足复杂检查的现有输入变换为到达并满足这种检查的新输入。

当 Driller 调用符号执行引擎时，它传递所有由模糊引擎识别的“感兴趣”输入（如 4-C 部分所定义）。 符号化地跟踪每个输入，以识别模糊引擎不能满足的状态转换。 当识别到这种转换时，concolic 执行引擎产生能通过该状态转换的的输入来驱动执行。

在符号执行引擎完成处理所提供的输入之后，其结果被反馈到模糊引擎的队列中，并且控制权被传递回到模糊引擎，使得其可以快速地探索应用中新发现的隔区。

本节的其余部分将描述 Driller 符号执行的实现和我们对 Driller 出现的问题做出的具体调整。

5.1 符号执行

我们利用 angr [29]，一个最近开源的符号执行引擎，用于 Driller 的符号执行引擎。该引擎基于由 Mayhem 和 S2E [8]，[10]推广和改进的模型。 首先引擎将二进制代码转换为 Valgrind 的 VEX [24]中间表示，其被解释成符号状态表示以决定对程序代码的影响。这种符号状态使用符号变量来表示可能来自用户的输入或其他不确定的数据，例如来自环境的数据。

符号变量是可以产生许多可能的具体解决方案（例如数字 5）的变量（例如 X）。其他值，例如在程序中硬编码的常量，被建模为具体值。随着执行的进行，符号约束被添加到这些变量上。 约束是对符号值的潜在解的限制语句（例如，X <100）。 具体的解决方案是满足这些约束的任何 X 值。

分析引擎在整个执行过程中跟踪内存和寄存器中的所有具体和符号值（上述的符号状态）。在引擎到达的程序中的任何点处，可以执行约束决断以确定可能输入（在该状态下，满足的所有符号变量约束）。这样的输入当被传递到应用的正常执行时，将驱动程序运行到该点。符号执行的优点是它可以探索和找到约束求解器可以满足的任何路径的输入。 这使得它识别复杂比较的解决方案十分有效（甚至包括某些哈希函数），一个模糊器不太可能会这样的暴力方法。

Driller 的符号化内存模型可以存储具体和符号值。 它使用基于索引的内存模型，其中读取地址可以是符号的，但写入地址总是具化的。 这种由 Mayhem 推广的方法是一种重要的优化，以保持分析可行：如果读取和写入地址都是符号的，使用相同符号索引的重复读取和写入将导致符号约束的二次方级增加，或者取决于符号执行引擎的实现细节，存储的符号表达式的复杂性。 因此，符号写地址总是具体化为单个有效解。 在某些条件下，如本文的文献所提出的，符号值被具体化为单个潜在解[8]。

符号存储器的优化增加了符号执行引擎的可伸缩性，但是可能导致不完整的状态空间，以至于产生更少的可能的解决方案。不幸的是，对现实的二进制文件

进行分析，这是一个必须做出的折衷。

5.2 示例

符号执行在解决不同的问题时好于模糊器。回看来自 4-B 节的清单 3 中的演示模糊器缺陷的例子。由于通过检查需要精确的输入，保护了对弱点函数的调用，因此模糊测试在合理的时间内不能探索那段代码。

```
1   int main(void)
2   {
3       intx;
4       read(0,&x,sizeof(x));
5
6       if(x==0x0123ABCD)
7           vulnerable();
8   }
```

清单 3. 符号执行的程序可以解决的一个程序

然而，符号执行引擎将能够轻易地满足该检查并触发缺陷函数。 对于这个例子，符号执行只需要探索一小部分路径来找到一个在这个例子中达到缺陷的路径，但是对于更大的二进制和现实的例子，会有太多的路径以相同的方式探索。

5.3 限制

传统的符号执行方法，包括从程序起始就开始执行符号执行，然后用符号执行引擎来探索路径状态，以找到尽可能多的缺陷。 然而，这种方法有两个主要的限制。

首先，符号执行是很慢的。 这是由于需要解释应用程序代码（与使用模糊器原生地执行代码相反）以及约束求解步骤中涉及的开销。特别是，后一操作涉及 NP 完全问题的解决，使得潜在输入的生成（以及确定哪些条件跳转是可行的）十分耗时。

更糟的是，符号执行受到状态爆炸问题的困扰。 随着符号执行引擎探索该程序，路径数量呈指数增长，并且它很快变得不可能再探索下去。考虑清单 4 中的示例。在此程序中，当用户输入正好 25 个 B 字符时，将触发 weak（），但这是一个在符号执行框架中难以表达的条件。 这个程序的符号执行将导致巨大的状态爆炸，因为模拟 CPU 将递归调用进入到 check（）函数中。每次执行与字母 B 比较的三元操作都将模拟状态一分为二，最终导致 2^{100} 种可能的状态，这是不可行的处理量。

另一方面，选择输入是基于状态转换的遗传模糊器，不能推测程序的整个状态空间，而仅仅考虑由输入触发的状态转换。 也就是说，它将主要关注次数，例如，第 5 行的 check 函数执行成功。也就是说，不管输入中 B 字符在哪里，将状态的判断都将基于输入中的它们的数目来确定，避免路径爆炸问题。

虽然研究进展已经通过智能状态合并来减轻这个问题[1]，但普遍的问题仍然存在。

```
1   int check(char* x, int depth){
```

```
2           if(depth >= 100){
3                   return 0;
4           }else{
5                   int count = (*x == 'B') ? 1 : 0;
6                   count += check(x+1, depth+1);
7                   return count;
8           }
9   }
10
11  int main(void){
12          char x[100];
13          read(0, x, 100);
14
15          if(check(x,0) == 25)
16                  vulnerable();
17  }
```

清单 4. 在符号执行下导致路径爆炸的程序。

5.4 Driller 中的符号执行

在大多数情况下，模糊测试单独就可以充分地探索大部分路径，只要简单地通过使用随机位翻转和其他突变策略来找到它们。通过利用原生代码执行，在大多数情况下，它可以随机触发路径，它的性能优于符号执行。因此，大多数工作被从符号执行引擎交给了模糊器，这将快速找到许多路径，让符号执行引擎只是去解决更困难的约束。

当模糊测试不能发现进入新的执行路径的输入时，调用符号执行引擎。 它跟踪由模糊器发现的路径，识别切分到新隔区中的输入，并执行有限的符号探索。另外，当由模糊组件发现崩溃输入时，符号执行引擎“重新随机化”它以恢复取决于随机性和其他环境因素的部分崩溃输入。

1）**预约束跟踪**：Driller 使用符号执行跟踪来自模糊器的感兴趣的路径并生成新的输入。 这种方法的有效性的一个关键因素是，它允许 Driller 避免恢复探索中固有的路径爆炸，因为只有代表应用程序处理该输入的路径被分析。

当轨迹从模糊器传递到符号执行时，目的是发现模糊测试以前没有找到的新转换。 Driller 的符号执行引擎跟踪输入，遵循由模糊器采取的相同的路径。 当 Driller 进行条件控制流转移时，它检查是否反转该条件将导致发现新的状态转移。如果有，Driller 将生成一个示例输入，它将通过新的状态转换而不是原来的控制流来驱动执行。通过这样做，Driller 的符号执行引擎引导模糊引擎到应用程序的新隔间。 生成输入后，Driller 继续跟踪匹配路径，以查找其他新的状态转换。

2）**输入预约束**：Driller 使用预约束，以确保符号执行引擎的结果与原生执行引擎的结果相同，同时保持发现新的状态转换的能力。 在预约束执行中，输入的每个字节被约束与模糊器输出的每个实际字节匹配，例如/dev/stdin[0] == 'A'。 当发现新的可能的基本块转换时，暂时地去除预约束，允许 Driller 求解偏离该状态转换的输入。预约束对于在符号执行引擎中生成相同的轨迹是必要的，

并且使得有限的 concolic 探索可行。

```
1    int check(char* x,int depth){
2        if(depth >= 100){
3            return 0;
4        }else{
5            int count = (*x == 'B') ? 1 : 0;
6            count += check(x+1, depth+1);
7            return count;
8        }
9    }
10
11   int main(void){
12       charx[100];
13       int magic;
14       read(0, x, 100);
15       read(0, &magic, 4);
16
17       if(check(x,0)==25)
18           if(magic==0x42d614f8)
19               vulnerable();
20   }
```

清单 5. 展示需要预先约束符号输入的程序。

为了演示 Driller 中的输入预约束是如何工作的，我们使用清单 5 中的示例，它类似于 5.3 中的示例，另外，为了达到弱点函数，我们必须在第 18 行提供一个幻数（0x42d614f8）。 在输入模糊测试后，Driller 最终认识到它没有发现任何新的状态转换，因为单独的模糊器不能猜测正确的值。 当调用 concolic 执行跟踪输入时，Driller 首先约束符号输入中的所有字节以匹配跟踪到的输入字节。由于程序是以符号方式执行的，因此每个分支只有一种可能性，因此只跟随一个路径。这防止了在第 5.3 节中描述的路径爆炸。然而，当执行到达行 18 时，Driller 识别出存在在模糊化期间从未采取的备选状态转换。然后，Driller 删除在执行开始时添加的预约束，不包括将跟踪的输入以符号方式执行时放置的谓词。字符数组 x 中的字节被路径部分约束，magic 的值受等式检查 if（magic == 0x42d614f8）约束。 因此，符号执行引擎创建一个包含 25 个 B 和一个魔术值 0x42d614f8 的输入。 这通过了 18 行中的检查并到达缺陷函数 vulnerable。

3）**有限符号探索**：为了减少昂贵的符号引擎调用的次数，我们还引入了一个符号探索桩，以发现更多的状态转换，直接位于新发现的状态转换之后。 这个符号探索桩探索状态转换的周围区域，直到探索器遍历配置数量的基本块。一旦探索达到了这个数量，Driller 就会为探索器发现的所有路径确定输入。 我们认为这样做可以防止模糊器在接受 Driller 生成的输入后被“卡住”。

4）**重新随机化**：在程序运行期间引入的随机值可能回破坏如前所述的模糊尝试。 清单 6 显示了一个小程序，它测试用户是否能猜中生成的随机值。 这使得模糊测试不稳定，因为我们不能在不监视程序输出的情况下知道生成的随机值 challenge 的具体值。

```
1    int main(void){
```

```
2       int challenge;
3       int response;
4
5       challenge = random();
6
7       write(1, &challenge, sizeof(challenge));
8       read(0, &response, sizeof(response));
9       if(challenge == response)
10          abort();
11
12  }
```

清单 6. 需要重新引入随机性的程序

一旦发现了漏洞，我们使用符号执行来跟踪崩溃输入，并恢复输入字节，该输入字节需要满足目标二进制所提出的动态检查（如清单 6 的示例中的 challenge-response）。通过检查崩溃时的符号状态并找到应用程序的输出和崩溃输入之间的关系，Driller 可以确定应用程序的 challenge-response 协议。在这个例子中，我们可以看到，读取的字节被限定为等于写出的字节。 在确定这些关系之后，我们可以生成一个漏洞发现规范，处理在真实环境中发生的随机性。

第六章 评估

为了确定我们的方法的有效性，我们对大型二进制数据集进行了评估。 我们的评估目标是展示两件事：首先，Driller 显著扩展了无辅助模糊器实现的代码覆盖率，其次，这种增加的覆盖率使发现的漏洞数量增加。

6.1 数据集

我们从 DARPA 网络大挑战（CGC）[11]的资格赛中评估了 Driller，该比赛旨在“测试新一代全自动化网络防御系统的能力”[11]。 在活动期间，选手有 24 小时自主发现内存损坏漏洞，并通过给出输入，使相关应用程序处理时导致崩溃来证明。 在 CGC 资格赛数据集中有 131 个服务，但其中 5 个涉及多个二进制文件之间的通信。 由于这样的功能超出了本文的范围，我们只考虑 126 个单二进制应用程序，留下多二进制应用程序作为以后的工作。

这 126 个应用程序包含各种各样的障碍，使得二进制分析变得困难，例如复杂的协议和很大的输入空间。 它们专门用于突出程序分析技术的能力，而不仅仅是玩具应用程序用于黑客娱乐（与 Capture The Flag 黑客竞赛中通常看到的不同）。这些二进制文件的种类和深度允许对高级漏洞挖掘系统（如 Driller）进行广泛测试。 此外，顶级竞争对手的结果在线可查，赛事提供了一个测试集，通过验证后的结果评估分析系统的性能。

6.2 实验设置

我们在现代 AMD64 处理器的计算机集群上运行我们的实验。每个二进制有四个专用的模糊器节点，当模糊器需要符号执行帮助时，它将作业发送到 64 个所有二进制文件共享的符号执行节点池中。 由于可用内存的限制，我们将每个符号执行作业限制为 4 GB 的 RAM。 在我们的所有测试中，我们分析一个单一的二进制文件最多 24 小时，这是给予 CGC 资格赛团队的相同时间。 我们分析每个二进制直到发现崩溃或 24 小时过去。

所有崩溃都使用赛事提供的二进制测试工具收集和重放，以验证提交的崩溃在实际的 CGC 环境中是可重现的。 因此，这些结果是真实的，可验证的，并且可以与竞赛的实际结果相比。

6.3 实验

我们在我们的评估中总共运行了三个实验。 首先，为了以现有技术的基准性能来评估 Driller，我们尝试使用纯符号执行引擎和纯粹的模糊器进行漏洞挖掘。 然后，我们在同一数据集上评估 Driller。

实验设置如下：

基本模糊测试。 在这个测试中，每个二进制被分配 4 个内核用于进行 AFL 模糊测试，但是符号执行节点被关闭。 当模糊器无法发现新路径时，它没有帮助。 注意，我们对 AFL 的 QEMU 后端进行了更改以提高针对 CGC 二进制文件的性能，但是，如前所述，没有对 AFL 的核心逻辑进行更改。

基本符号执行。 我们使用一个现有的符号执行引擎，基于 Mayhem [8]提出的想法，用于符号执行测试。为了确保对现有技术的公平测试，优化的状态合并技术被用于帮助限制状态爆炸的影响，如 Veritesting [1]中提出的。 我们通过符号探索状态空间，从入口点开始分析每个二进制文件，检查内存损坏。当发生状态爆炸时，我们使用启发式方法来对程序进行深入探索的路径进行优先级排序，以最大化代码覆盖率。

Driller。 当测试 Driller 时，每个二进制文件被分配 4 个内核用于模糊引擎，总共 64 个内核用于符号执行组件。当 Driller 确定模糊器被“卡住”时，由于轨迹是由模糊节点请求的，所以符号执行池处理先进先出队列中的符号执行作业。符号执行跟踪被限制为一小时周期和 4GB 字节的内存限制，以避免当分析大的轨迹时资源耗尽。

我们将讨论我们对 Driller 的评估的几个不同方面。 我们将首先讨论三个实验的结果，即 Driller 对我们在数据集中找到漏洞数量的贡献。 接下来，我们将讨论 Driller 在代码覆盖率方面对现有技术的贡献。 最后，我们将重点介绍 CGC 数据集中的示例应用程序，以进行深入的案例研究，以讨论 Driller 如何提高代码覆盖率并识别该应用程序中的漏洞。

6.4 缺陷

在本小节中，我们将讨论三个实验发现的漏洞的数量，以及 Driller 在这方面的贡献。

基本符号执行实验在这个数据集上表现不佳。 在 126 个应用程序中，符号执行只发现了 16 个漏洞。

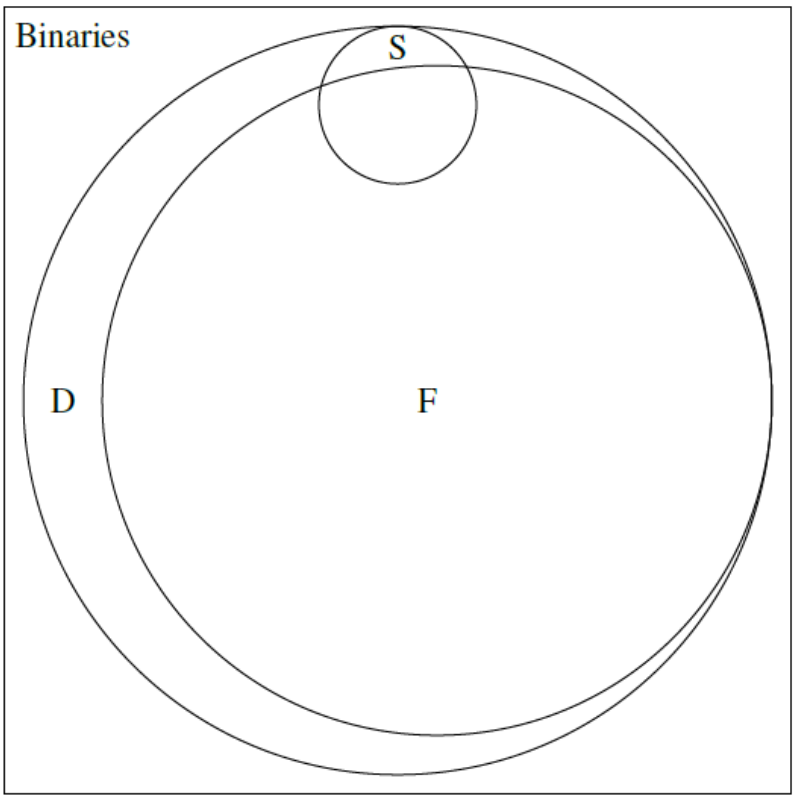
在我们的实验数据集中的 126 个 Cyber Grand Challenge 应用程序中，模糊测试足以发现 68 个的崩溃。在剩余的 58 个二进制文件中，41 个被“卡住”（即 AFL 无法识别任何新的“感兴趣路径” 如第四章所讨论，不得不求助于随机输入突变），17 个尽管继续找到新的感兴趣的输入，但没有发现崩溃。

在 Driller 的运行中，模糊器调用了“卡住”的 41 个二进制文件上的符号执行组件。图 7 显示了对这些二进制文件调用符号执行的次数。其中，Driller 的符号执行组件能够为其中 13 个应用程序生成总共 101 个新输入。利用这些额外的输入，AFL 能够恢复额外的 9 次崩溃，使 Driller 实验中确定的总崩溃数达到 77 次，这意味着 Driller 发现的漏洞相对于基本模糊测试实现了 12％的改进。

当然，在 Driller 实验中发现崩溃的大多数应用程序是与基本模糊器一起发现的。 对于通过不同方法识别的唯一崩溃，基本模糊器发现了 55 个符号执行未能发现的崩溃。 其中的 13 个漏洞与基本符号执行共享。 另外 3 个漏洞基本符号执行与 Driller 恢复的漏洞重叠，剩下应用程序，其中基本符号执行单独发现了 1 个漏洞，剩下 6 个应用程序，Driller 是唯一找到漏洞的方法。基本上，Driller 有效地合并和扩展了基线模糊和基线符号执行提供的功能，获得了比单独执行更多的结果。 这些结果示于图 5 中。

总而言之，Driller 能够识别 77 个独特应用程序中的崩溃，与基础实验的并集（71 个漏洞）相比，改善了 6 个崩溃（8.45%）。 这与竞赛中得分最高的团队确定的崩溃数量相同（并且明显高于任何其他竞争对手），在相同的时间内。没有 Driller（即使用两种基本方法），我们不会实现这些结果。 注意，我们清楚地知道，与参与团队的比较只是指示性的，并不意味着是定性的。 参与团队在严格的时间限制下运行，只有很少或没有错误的空间。 我们的实验受益于更多的准备时间，我们的技术还可以在 Driller 的发展过程中改进。

这些结果表明，优化的具有选择符号执行的模糊器改善了其在发现崩溃中的性能。 通过提高漏洞挖掘中的技术水平，Driller 能够找到比通过模糊测试和通过符号执行分别发现的并集更多的崩溃程序。虽然 6 个唯一漏洞的贡献可能看起来远低于 CGC 限定事件中的应用程序总数，但这些崩溃代表其各自二进制文件中的深层漏洞，其中许多漏洞需要多个符号执行调用来穿透几个隔区。



Method	Crashes Found
Fuzzing	68
Fuzzing ∩ Driller	68
Fuzzing ∩ Symbolic	13
Symbolic	16
Symbolic ∩ Driller	16
Driller	77

图 5. 实验结果的组成。 维恩图显示了基本模糊（AFL），符号执行和 Driller 在 CGC 数据集中查找崩溃的相对覆盖率。 标记为 F 的圆圈表示通过模糊查找发现的崩溃，S 表示通过符号执行发现的崩溃，D 表示由 Driller 发现的崩溃。 该表根据不同方法的相对有效性

及其相对于彼此的改进提供了这些结果。 可以看到，Driller 识别了由 Fuzzing 和 Symbolic Execution 发现的崩溃的超级集合。

6.5 状态转移覆盖

选择性符号执行能够克服在处理“魔数”常数和其他复杂输入检查时模糊器的根本弱点。这意味着，在模糊器不能识别新的感兴趣输入（例如，由于无法猜测散列或魔数）之后，符号执行引擎可以生成允许模糊器继续探索已被卡住的路径的输入， Driller 的这个方面可以在表 I 中观察到，其示出了在执行期间如何发现状态转换。 在符号执行能够找到新路径的应用中，单独的模糊测试只发现了平均 28.5％的块转换。

如所预期的，符号跟踪在这些二进制中仅占少量新的状态转换（平均约 15.1％），因为符号探索在范围上受限并且主要用于识别和通过感兴趣的检查。然而，由符号执行引擎产生的输入帮助模糊引擎成功地穿透这些状态转换。 模糊引擎对这些输入的后续修改允许它平均找到额外的 56.5％的状态转换。 总的来说，对于模糊器被阻塞并且最终符号执行找到新路径的应用，71.6％的状态转换由基于符号跟踪期间生成的那些的输入产生。

事实上，小数量的有价值输入将导致更大的模糊器可以探索的状态转换集合，表明 Driller 的符号执行引擎产生的输入刺激了更深入的应用程序的探索。重要的是要记住，这个数字仅适用于 41 个应用程序中的 13 个，这些应用程序变得“卡住”，并且能够通过符号执行识别新的路径。 这些百分比在我们在实验过程中的基本块总量上归一化，因为生成完整的控制流图静态地需要超出本文范围的重量级静态分析。

如在第四章中所讨论的，我们将状态转换考虑为基本块（A，B）的有序对，其中块 B 在块 A 之后立即执行。换句话说，状态转换是控制流图中的边，其中每个节点表示程序中的基本块。 很明显，如果我们发现每个状态转换，我们就有完整的代码覆盖。类似地，如果我们只发现很少状态转换，那么我们可能具有非常低的覆盖。因此，使用不同状态转换的数量作为代码覆盖的度量是合理的。在图 6 中，我们通过显示 Driller 找到的模糊器单独找不到的额外基本块数量，来展示 Driller 随时间推移提升了多少基本块覆盖。

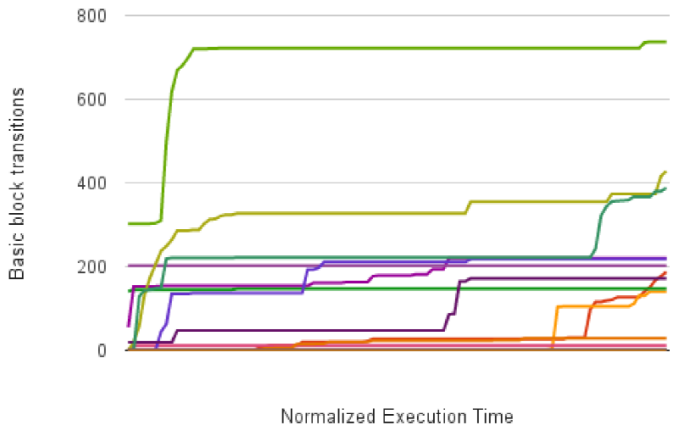


图 6. Driller 随时间发现的模糊器无法单独找到额外基本块的数量。执行时间显示为根

据二进制文件执行时间归一化后的结果，因为二进制文件执行时间由于它是否/何时崩溃而各不相同。 此图包括调用的 13 个受益于符号执行二进制文件。

6.6 程序隔区覆盖

Driller 中的符号轨迹的目标是使得模糊器能够探索二进制中的各个隔区，其可以通过对用户输入进行复杂检查来分离。我们期望看到通过调用符号跟踪器产生的输入符合在应用程序中找到新的隔间。 也就是说，由符号执行引擎产生的输入应该使得模糊器能够到达并探索新的代码区域。

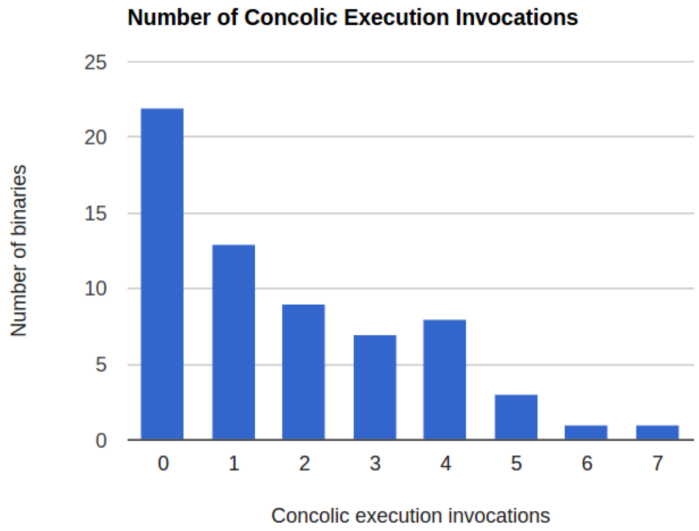


图 7. 图表显示在模糊测试本身不会使其崩溃的二进制程序中调用的符号执行的次数。

如图 5 所示，数据集中 126 个应用程序中的 68 个，没有任何需要 Driller 符号执行的困难检查。这些对应于模糊组件可以独立发现崩溃输入的或其从未被“卡住”的应用。这些往往是具有简单协议和较少复杂检查的应用。另一方面，Driller 能够至少在 13 个二进制文件中找到一个困难检查，并在 4 各二进制文件中找到多个困难检查。这些隔间由于分离它们的特定检查，所以对于基本的模糊器很难进入，但是可以通过 Driller 使用的混合方法解决。

每次调用符号执行都有可能将执行引导到应用程序中的一个新隔区。这可以通过在一个模糊循环被“卡住”，并且在随后的一轮模糊实现的覆盖上调用符号执行之前，且在符号执行组件将执行推送到下一个隔区之后，分析 Driller 的基本块覆盖来度量。在图 8 中，显示了在分析的每个阶段，调用了符号执行的每个二进制文件中，总数标准化为整个实验中发现的基本块总数的基本块比例。该图表明 Driller 确实将应用程序中的执行驱动到了新的隔区，允许模糊器快速探索更大量的代码。 在第 6.7 节中，我们为案例分析给出了一个深入的例子。

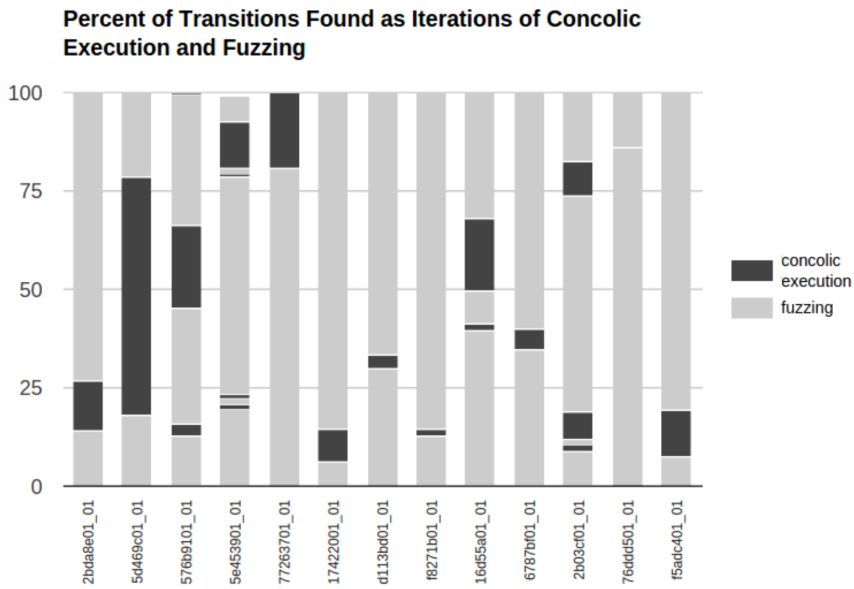


图 8. 图表显示了每次调用符号执行导致发现更多的基本块转换。
仅显示符号执行识别到额外输入的二进制文件。

6.7 案例分析

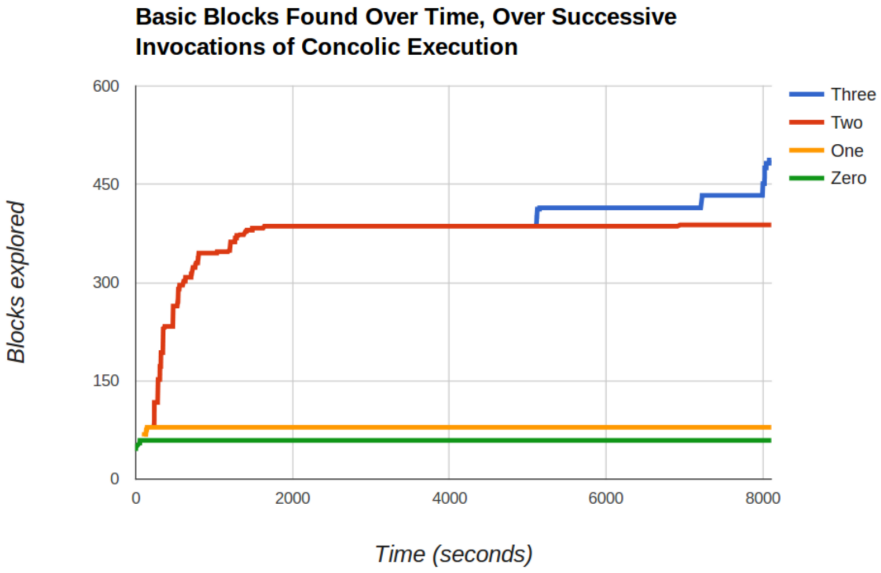


图 9. 对于二进制 2b03cf01，Driller 在大约 2.25 小时内崩溃，该图显示了随时间发现的基本块的数量。 每行代表不同数量的符号执行调用（从 0 到 3 次）。
在每次调用符号执行之后，模糊器能够找到更多的基本块。

Type of State Transition	Percentage of discovered blocks across all binaries	Percentage of discovered blocks across binaries where concolic execution found at least one input

Initial Fuzzing Run	84.2	28.4
Identified by Concolic Execution	3.3	15.1
Post-Concolic Fuzzing Runs	12.5	56.5
Total	100	100

表 1. 在所有调用了符号执行的二进制文件中，通过何种方法发现状态转移的百分比，并且这些二进制文件中的符号执行至少发现了一个输入

```
1   enum {
2       MODE_BUILD = 13980,
3       MODE_EXAMINE = 809110,
4   };
5
6   ...
7
8   RECV(mode, sizeof(uint32 t)) ;
9
10  switch(mode[0]) {
11      case MODE_BUILD:
12          ret = do_build();
13          break;
14      case MODE_EXAMINE:
15          ret = do_examine();
16          break;
17      default:
18          ret = ERR_INVALID_MODE;
19  }
```

清单 7. 2b03cf01 应用程序中的第一个复杂检查

本节将重点介绍单个应用程序，以深入解释 Driller 的操作。我们将关注标识符为 2b03cf01 的 CGC 资格赛应用程序。感兴趣的读者可以在 DARPA 的 github 库[12]中名称为 NRFIN_00017 的公共集中找到此应用程序的源代码。另外，我们提供这个二进制的调用图，我们将在这个案例中参考图 10。该图显示了 Driller 的模糊测试和符号执行组件连续调用的性能 - 通过连续的模糊测试调用发现的节点被绘制为逐渐变暗的颜色，并且由不同样式绘制的边，展示由符号执行组件重新发现的转换。节点的不同颜色表示二进制中的不同隔间。

该应用表示电源测试模块，其中客户端向服务器提供电气设计，并且服务器建立电连接的模型。这不是一个简单的二进制：它为用户提供了各种复杂的功能，并要求格式正确的输入，有许多复杂的检查。

当 Driller 对这个二进制进行模糊测试时，第一个复杂的检查导致模糊器在应用程序的一个相当小的区域中只找到 58 个基本块后立即卡住，包括一些包含初始化代码的函数。模糊引擎卡在对用户输入的检查。为了方便，对应于图 10 中的节点“A”的所讨论的片段在清单 7 中再现，尽管 Driller 直接在二进制代码上操作。

来看源代码，我们看到从主循环调用的两个主要命令，需要用户给出一个特定的32 位数字来选择“操作模式”。要调用函数 *do_build()*，用户必须提供数字 13980，要调用函数 *do_examine()*，用户必须提供数字 809110。虽然这些检查对于人来说看起来很简单，但实际上一个模糊器必须暴力破解他们。因此，模糊器猜测这些魔数的机率是微乎其微的，结果模糊组件被卡住。

在模糊器无法识别新的感兴趣路径之后，Driller 调用符号执行组件来跟踪模糊器目前收集的输入，并找到新的状态转换。Driller 找到将驱动执行上述两个函数的输入，并将它们返回到模糊器进行探索。再次，模糊器被相当快地阻塞，这次在图 10 中的节点“B”处的另一个复杂检查。Driller 的符号执行引擎被第二次调用，产生足够的新输入来通过这些检查。从这一点开始，模糊器能够在处理通用输入的应用程序大隔间内找到 271 个附加基本块，对于该应用，通用输入由与用户提供的电气设计分析相关的解析代码组成。最终，模糊器找到它在该隔间中可能存在的所有有趣的路径，并决定它不会产生进一步进展，导致调用另一个 Driller 的符号执行引擎。

这一次，Driller 找到 74 个新的基本块，并生成到达它们的输入，这依靠成功地通过模糊器以前没有产生满足的输入的检查。这些附加的基本块（由图 10 中的黑色节点表示）包括添加特定电路组件的功能。对于感兴趣的读者，清单 9 给出了对于模糊器有麻烦的，包含针对用户输入的特定检查的函数。表示这些组件的输入，必须遵守电路组件的确切规范，并且这些规范的检查是 Driller 的符号执行引擎的第三次调用所发现的。在函数检查中使用的这些常量在清单 8 中重现的代码中定义。模糊器不消耗巨大的搜索空间是不能猜测到这些常量的，因为它们是 32 位整数的特定值。然而，Driller 的符号执行可以很容易地找到这些常量，因为代码中的比较动作，在采用这些分支的路径上产生易于解决的条件。

```
1    typedef enum {
2        FIFTEEN_AMP = 0x0000000f,
3        TWENTY_AMP = 0x00000014,
4    } CIRCUIT_MODELS_T;
```

清单 8.具有显式常量的枚举定义。 为了猜测这些常数，这些特定值必须从 2^{32} 个数字的搜索空间中猜出。

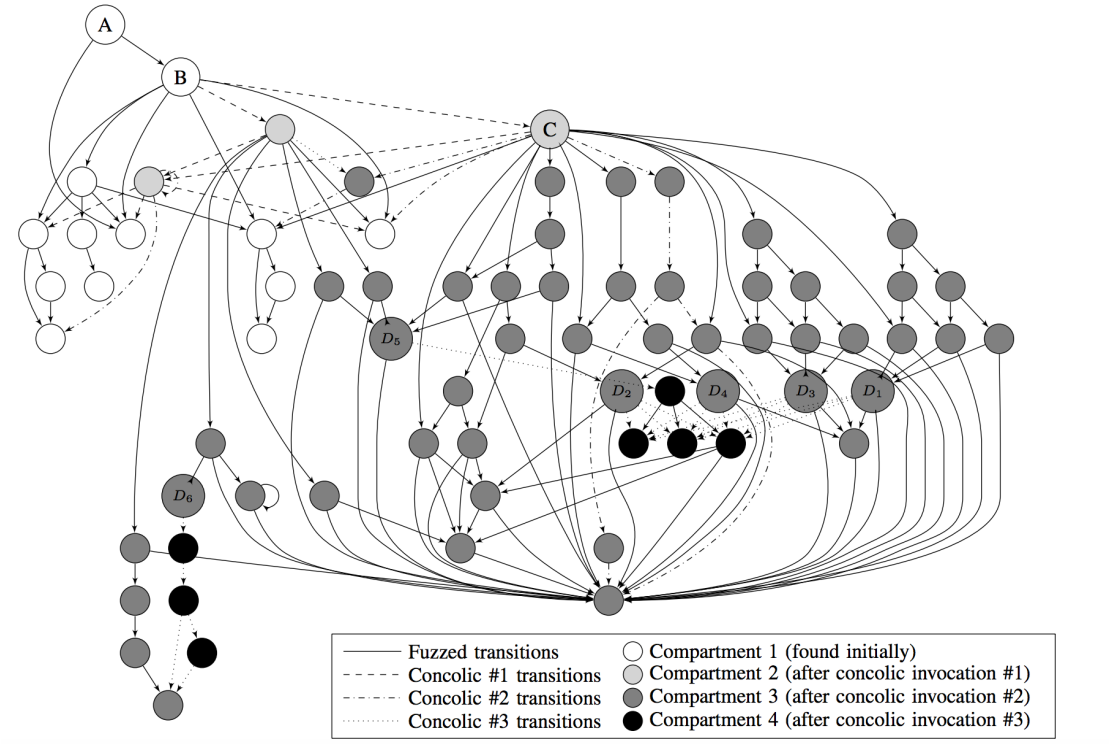


图 10. 显示 Driller 发现新隔区的过程。 每个节点是一个函数; 每个边都是一个函数调用，但返回边被排除以保持可读性。 节点“A”是入口点。 节点“B”包含一个魔数检查，需要符号执行组件来解析。 节点“C”包含另一个魔数检查。

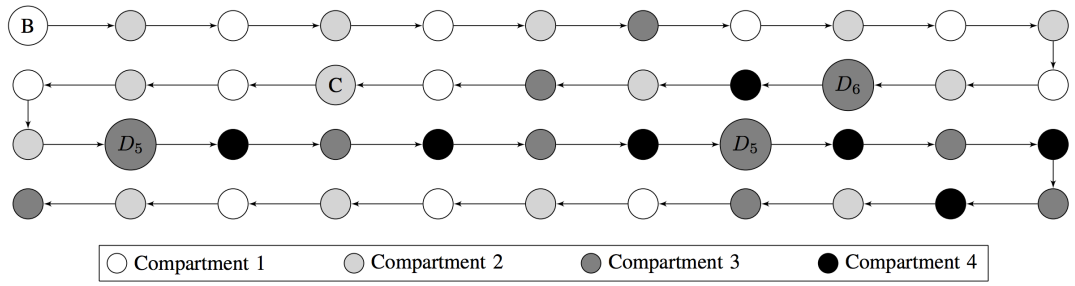


图 11. 对 CGC 应用 2b03cf01 的崩溃输入跟踪的执行流程，所通过的隔间序列。 Driller“进入”第四隔间（由黑色节点表示）的能力对于生成崩溃输入是至关重要的。

生成的去随机化崩溃输入为

“A\x00\x00\x00\x00\x00\x00\x00\x9c6\x00\x00\x18\x04\x00\x00\x18’\x00\x00A\x00\x00\x00\x00\x00\x00\x00\x9c6\x00\x00\x19\x04\x00\x00\x14\x00\x00\x00A\x00\xf8\xff\xff\xec\x00d\x96X\x0c\x00\x06\x08\x00\x00\x10\x00\x00\x00A\x00\x00\x00\x00\x00\x00\xfb\x96X\x0c\x00\x02\x08\x00\x00\x18’\x00\x00A\x00\xebA\x00\x00d\x96X\x0c\x00\x06”.

符合 DARPA CGC 漏洞利用范例格式并考虑随机性的完整漏洞范例在附录 A 中提供。

```
1    int8_t get_new_breaker_by_model_id
                (CIRCUIT_MODELS_T model_id, breaker_t *
                breaker_space, uint8_t breaker_space_idx) {
2        int8_t res = SUCCESS;
```

```
3           switch(model_id) {
4                   case FIFTEEN_AMP:
5                           create_breaker(15, breaker_space, breaker_space_idx);
6                           break;
7                   case TWENTY_AMP:
8                           create breaker(20, breaker_space, breaker_space_idx);
9                           break;
10                  default:
11                  //invalidmodelid
12                          res=−1;
13          }
14          return res;
15  }
```

清单 9. 一个带有 switch 语句的函数，用于对多个特定值测试用户输入

Driller 的新输入之后被传递回到模糊器，以便快速评估由变化产生的新覆盖。清单 10 显示了作为新输入的结果第一次执行的代码。这个新组件处理的用户输入不再是具体的，而是一般的，使其适合于模糊器。从这一点开始，模糊器继续改变这些输入，直到它触发由应用程序中缺少的清理检查引起的漏洞。

```
1   static void create_breaker
                    (uint8_t amp_rating, breaker_t *
                    breaker_space, uint8_t breaker_space_idx) {
2           breaker_space−>id = breaker_space_idx;
3           breaker_space−>amp_rating = amp_rating;
4           breaker_space−>outlets = list_create_dup();
5           if(breakerspace−>outlets == NULL) { _terminate(ERRNO_ALLOC); }
6   }
```

清单 10.由于传递特定检查而执行的代码

在语义上，该漏洞涉及在用户创建的电路中初始化新的断路器对象。稍后，电路将被测试连接性等，并且将根据组成电路的材料来调用特定于组件的逻辑。满足检查以添加断路器将扩展错误搜索覆盖范围，以包括断路器特定的代码。触发此漏洞需要在提供的电路图中包含特制的断路器组件。触发创建这些组件所需的输入是 Driller 在第三个符号执行调用中恢复的输入，最后的模糊测试调用使它们足够多，以触发缺陷的边缘情况。

崩溃输入所采用的最后路径如图 11 所示。从入口点开始，该路径经过逐渐更难到达的隔区（由节点的不同颜色表示），直到触发边缘的条件被创建。这个二进制在基本实验中没有崩溃 - 独立的模糊器从来不能通过复杂检查到达“保护”代码的区域，并且符号探索引擎在输入处理代码中经历了几乎立即发生的路径爆炸。通过结合模糊测试和符号执行的优点，Driller 对这个二进制文件大约两个小时发现崩溃。

我们给出了在这个二进制中每个符号执行调用，产生的基本块覆盖率的扩大，在图 9 中随时间绘制。

第七章 讨论

Driller 通过利用符号执行和模糊进行统一的分析。 这允许 Driller 用其补充来解决每个分析的一些缺点。 在本节中，我们将讨论 Driller 的局限性和未来的研究方向，以进一步增强自动化漏洞提取。

7.1 限制

Driller 的一个优点也是缺陷，是它借用了 AFL 的状态空间解释。AFL 通过简单地跟踪状态转换元组计算粗略的“命中计数”（遇到状态转换的次数）来表示状态。种适度的轻量状态表示，是 AFL 如此有效的原因，因为每个路径的状态仅由它遇到的状态转换元组的集合，以及它们遇到了多少次来定义。Driller 使用这个相同的数据结构来确定哪些状态转换值得解决。我们在清单 11 中举一个例子来说明这如何限制了 Driller。

```
1   int static_strcmp(char *a, char *b) {
2       for(; *a; a++,b++) {
3           if(*a != *b)
4               break;
5       }
6
7       return *a − *b;
8   }
9
10  int main(void) {
11  read(0, user_command, 10);
12
13      if (static_strcmp("first_cmd", user_command) == 0) {
14          cmd1();
15      }
16      else if (static_strcmp("second_cmd", user_command) == 0) {
17          cmd2();
18      }
19      else if (static_strcmp("crash_cmd", user_command) == 0) {
20          abort();
21      }
22
23      return 0;
24  }
```

清单 11. 限制发现新状态转换的最小状态表示的示例。

此清单演示了在多个命令处理程序中发生的状态转换。由于每个分支都依赖于 *static_strcmp*，AFL 本身将不能在 *static_strcmp* 的不同调用内区分状态转换。

因此，Driller 不会尝试在第 3 行上多次解析 *if* 语句，即使它用于不同的比较。此外，具有一个或两个额外匹配字符的输入将不被 AFL 视为感兴趣。当然，如果整个字符串是由 Driller 发现的，AFL 会对它感兴趣，并采纳它。Driller 试图在每个新的状态转换时调用的符号探索器桩（在 5.4 中描述）减轻这个问题的影响。然而，我们认为这是一个不完美的解决方案，最终可能需要更好的状态表示。

Driller 的另一个限制是用户输入在一个组件中被视为通用输入，在另一个组件中被视为特定输入。 考虑清单 12 中提供的程序。

此应用程序从用户读取命令和散列，并验证散列。此隔区（跨越第 1 到 11 行）将命令视为通用输入，将散列视为特定输入。然后，应用程序在多个阶段检查提供的命令是否是“CRASH！”。基本上，这会将 *user_command* 重新分类为特定输入，因为它必须完全匹配。 这触发了将 Driller 降到符号执行引擎的情况，如下所述。

```
1    int main(void) {
2          char user_command[10];
3          int user_hash;
4
5          read(0, user_command, 10);
6          read(0, user_hash, sizeof(int));
7
8          if (user_hash != hash(user_command)) {
9                puts("Hash mismatch!");
10               return 1;
11   }
12
13         if (strncmp("CRASH",usercommand,5) == 0) {
14               puts("Welcome to compartment 3!");
15               if (user_command[5] == '!') {
16                     path_explosion_function();
17                     if(user_command[6] == '!') {
18                           puts("CRASHING");
19                           abort();
20                     }
21               }
22         }
23
24         return 0;
25   }
```

清单 12. 在一个地方用作通用输入并在另一个地方用作特定输入的输入示例。 这个二进制的崩溃输入是“CRASH！”，后面是它的哈希。

第一阶段，进入到隔区 3 中是直接的 - Driller 的符号执行引擎将识别以“CRASH”开始的输入及其对应的散列（因为这是散列的向前计算，所以不必担心需要“破解”哈希；Driller 只需要计算它）。然而，在这之后，模糊器将不再用于探索这个隔区。 这是因为对哈希或输入的任何随机突变将可能导致执行从

分区 1 无法进行。因此，模糊器将很快被阻塞，并且 Driller 将再次调用符号执行引擎。这个调用将引导 Driller 到在第 16 行的隔区 4，执行回到模糊器。 但是，模糊器将再次失败，决定它被卡住，并触发符号执行。

这个循环将继续，使得模糊组件无用并且基本上将 Driller 降为符号探索。更糟的是，在本应用程序中，隔区 4 调用导致路径爆炸的函数（*path_explosion_function*）。没有模糊引擎的缓解效果，Driller 将不能到达隔间 5（18 和 19 行）并触发漏洞。

这表示 Driller 中的限制：在某些情况下，模糊组件可以被有效地禁用，这抢夺了 Driller 的优势。减轻这个问题的潜在未来阶段是产生“半符号”模糊输入的能力。例如，符号执行引擎可以将一组约束传递给模糊器，以确保它生成的输入符合一些规范。这将利用代生模糊的概念[18]创建“输入生成器”，以帮助模糊器到达和探索应用程序隔区。

清单 12 所示的限制显示了特定输入如何防止模糊器有效地改变通用输入。然而，对于其他类型的特定输入，即使具有多个组件，AFL 仍可以模糊较深的组件。即使在最困难的情况下，例如哈希检查，Driller 仍然能够改变与哈希无关的任何输入，例如检查哈希后的输入。我们已经料到 Driller 发现多个组件后性能会有所下降。 这是因为 AFL 不知道来自符号执行引擎的约束，所以将有一小部分的模糊周期浪费在了试图突变特定输入。

第八章 总结

在本文中，我们提出了 Driller 这样一个工具，结合最好的动态模糊和符号执行，能够有效地找到潜藏在二进制中的漏洞。我们介绍一个二进制隔区的概念，这在很大程度上分离了功能和代码。在 Driller 中，模糊器提供了一个快速且廉价的隔区概览，有效地探索循环和简单的检查，但通常不能在隔区之间转换。当考虑到循环和内部检查时，选择性的符号执行将进入状态爆炸，但它在找到二进制中隔区之间的路径时非常有效。通过结合这两种每个都会单独失败的技术，Driller 能够探索二进制内更大的功能空间。

我们在 DARPA 网络大挑战赛资格赛提供的 126 个二进制文件上评估了 Driller。Driller 发现 77 次崩溃，与基本模糊器发现的 68 次相比，有了实质性的改进。我们相信这种技术，在对所有类别的二进制文件进行通用漏洞发现上，显示了良好的前景。

参考文献

[1]. T. Avgerinos, A. Rebert, S. K. Cha, and D. Brumley. Enhancingsymbolic execution with veritesting. In Proceedings of the InternationalConference on Software Engineering (ICSE), pages 1083–1094. ACM,2014.

[2]. F. Bellard. QEMU, a fast and portable dynamic translator. In USENIXAnnual Technical Conference, FREENIX Track, pages 41–46, 2005.

[3]. S. Bucur. Improving Scalability of Symbolic Execution for Softwarewith Complex Environment Interfaces. PhD thesis, Ecole Polytechnique ´Fed´ erale de Lausanne, 2015. ´

[4]. C. Cadar, D. Dunbar, and D. R. Engler. KLEE: Unassisted andautomatic generation of high-coverage tests for complex systemsprograms. In Proceedings of the USENIX Symposium on OperatingSystems Design and Implementation (OSDI), 2008.

[5]. C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler.EXE: Automatically generating inputs of death. ACM Transactions onInformation and System Security (TISSEC), 12(2):10, 2008.

[6]. G. Campana. Fuzzgrind: un outil de fuzzing automatique. In Actes du7eme symposium sur la s ` ecurit ´ e des technologies de linformation et ´des communications (SSTIC), 2009.

[7]. D. Caselden, A. Bazhanyuk, M. Payer, L. Szekeres, S. McCamant, andD. Song. Transformation-aware exploit generation using a HI-CFG.Technical report, UCB/EECS-2013-85, 2013.

[8]. S. K. Cha, T. Avgerinos, A. Rebert, and D. Brumley. UnleashingMayhem on binary code. In Proceedings of the IEEE Symposium onSecurity and Privacy, 2012.

[9]. Y. Chen, A. Groce, C. Zhang, W.-K. Wong, X. Fern, E. Eide,and J. Regehr. Taming compiler fuzzers. In Proceedings of theACM SIGPLAN Conference on Programming Language Design andImplementation (PLDI), volume 48. ACM, 2013.

[10]. V. Chipounov, V. Kuznetsov, and G. Candea. S2E: A platform forin-vivo multi-path analysis of software systems. In Proceedings ofthe Sixteenth International Conference on Architectural Support forProgramming Languages and Operating Systems, ASPLOS XVI, pages265–278. ACM, 2011.

[11]. DARPA. Cyber Grand Challenge. http://cybergrandchallenge.com.

[12]. DARPA. Cyber Grand Challenge Challenge Repository. https://github.com/CyberGrandChallenge/samples/tree/master/cqe-challenges.

[13]. J. DeMott. Understanding how fuzzing relates to a vulnerability likeHeartbleed. http://labs.bromium.com/2014/05/14/understanding-how-fuzzing-relates-to-a-vulnerability-like-heartbleed/.

[14]. C. Details. Vulnerability distribution of CVE security vulnerabilitiesby type. http://www.cvedetails.com/vulnerabilities-by-types.php.

[15]. W. Drewry and T. Ormandy. Flayer: Exposing application internals.In Proceedings of the USENIX Workshop on Offensive Technologies(WOOT), 2007.

[16]. D. Engler and D. Dunbar. Under-constrained execution: Makingautomatic code destruction easy and scalable. In Proceedings of theInternational Symposium on Software Testing and Analysis (ISSTA).ACM, 2007.

[17]. V. Ganesh, T. Leek, and M. Rinard. Taint-based directed whiteboxfuzzing. In Proceedings of

the International Conference on SoftwareEngineering (ICSE), 2009.
[18]. P. Garg. Fuzzing - mutation vs. generation. http://resources.infosecinstitute.com/fuzzing-mutation-vs-generation/.
[19]. P. Godefroid, N. Klarlund, and K. Sen. DART: Directed automatedrandom testing. In Proceedings of the ACM SIGPLAN Conferenceon Programming Language Design and Implementation (PLDI),volume 40, pages 213–223. ACM, 2005.
[20]. P. Godefroid, M. Y. Levin, and D. Molnar. SAGE: Whitebox fuzzingfor security testing. Communications of the ACM, 55(3):40–44, 2012.
[21]. I. Haller, A. Slowinska, M. Neugschwandtner, and H. Bos. Dowsingfor overflows: A guided fuzzer to find buffer boundary violations. InProceedings of the USENIX Security Symposium, 2013.
[22]. LegitBS. DEFCON Capture the Flag. https://legitbs.net/.
[23]. B. P. Miller, L. Fredriksen, and B. So. An empirical study of the reliabilityof UNIX utilities. Communications of the ACM, 33(12):32–44, 1990.
[24]. N. Nethercote and J. Seward. Valgrind: a framework for heavyweightdynamic binary instrumentation. In Proceedings of the ACM SIGPLANConference on Programming Language Design and Implementation(PLDI), volume 42, pages 89–100. ACM, 2007.
[25]. M. Neugschwandtner, P. Milani Comparetti, I. Haller, and H. Bos. TheBORG: Nanoprobing binaries for buffer overreads. In Proceedings ofthe ACM Conference on Data and Application Security and Privacy(CODASPY). ACM, 2015.
[26]. B. S. Pak. Hybrid fuzz testing: Discovering software bugs via fuzzingand symbolic execution. Master's thesis, School of Computer Science,Carnegie Mellon University, May 2012.
[27]. D. A. Ramos and D. Engler. Under-constrained symbolic execution:Correctness checking for real code. In Proceedings of the USENIXSecurity Symposium, 2015.
[28]. Secunia. Resources vulnerability review 2015. http://secunia.com/resources/vulnerability-review/introduction/.
[29]. Y. Shoshitaishvili, R. Wang, C. Hauser, C. Kruegel, and G. Vigna.Firmalice - automatic detection of authentication bypass vulnerabilitiesin binary firmware. In Proceedings of the Symposium on Network andDistributed System Security (NDSS), 2015.
[30]. T. Wang, T. Wei, G. Gu, and W. Zou. TaintScope: A checksum-awaredirected fuzzing tool for automatic software vulnerability detection. InProceedings of the IEEE Symposium on Security and Privacy, 2010.
[31]. M. Zalewski. American Fuzzy Lop. http://lcamtuf.coredump.cx/afl/.

附录 A

案例研究中的漏洞利用范例

对于好奇的读者，我们在清单 13 中给出了准确的崩溃输入，符合 Cyber Grand Challenge 指定的漏洞证明格式。

```
1   <?xml version="1.0" standalone="no" ?>
2   <!DOCTYPE pov SYSTEM "/usr/share/cgc-docs/replay.dtd">
3   <pov>
    <cbid>2b03cf01_01</cbid>
    <replay>
4       <read>
            <length>1</length>
            <assign><var>stdout0</var><slice begin="0" /></assign>
        </read>
        <read><length>7</length></read>
5       <write><var>stdout0</var></write>
        <write>
          <data>\x00\x00\x00\x00\x00\x00\x9c6\x00\x00\x18\x04\x00\x00\x18\x27\x00\x00</data>
        </write>
6       <read>
            <length>1</length>
            <assign><var>stdout8</var><slice begin="0" /></assign>
        </read>
        <read><length>7</length></read>
7       <write><var>stdout8</var></write>
        <write>
          <data>\x00\x00\x00\x00\x00\x00\x9c6\x00\x00\x19\x04\x00\x00\x14\x00\x00\x00</data>
        </write>
        <read><length>37</length></read>
8       <read>
            <length>1</length>
            <assign><var>stdout16</var><slice begin="0" /></assign>
        </read>
        <read><length>7</length></read>
9       <write><var>stdout16</var></write>
        <write>
          <data>\x00\xf8\xff\xff\xec\x00d\x96X\x0c\x00\x06\x08\x00\x00\x10\x00\x00\x00</data>
        </write>
        <read><length>37</length></read>
10      <read>
            <length>1</length>
            <assign><var>stdout24</var><slice begin="0" /></assign>
        </read>
```

```
        <read><length>7</length></read>
11      <write><var>stdout24</var></write>
        <write>
          <data>\x00\x00\x00\x00\x00\x00\xfb\x96X\x0c\x00\x02\x08\x00\x00\x18\x27\x00\x00</data>
        </write>
    </replay>
    </pov>
```

清单 13. 案例研究中使用的应用程序的漏洞证明

该输入可以通过执行清单 14 所示的命令在 Cyber Grand Challenge 机器上运行。

```
1 cb−test −−debug −−should_core −−cb ./NRFIN_00017
                −−xml ./POV_FROM_LISTING.xml −−directory .
```

清单 14. 测试所提供的漏洞证明的命令