**班级**  1318019

**学号**  13010310043

**本科毕业设计（论文）**

**外文资料翻译**

**毕业设计题目**  基于覆盖导向原则的开源模拟器

半自动模糊测试框架

**外文资料题目**  Automated Whitebox Fuzz Testing

**学 院**  网络与信息安全

**专 业**  信息安全

**学 生 姓 名**  唐天文

**指导教师姓名**  唐青昊(校外),张宁(校内)

# 概要

模糊测试是在软件中发现安全漏洞的一种有效技术。传统上，模糊测试工具将随机突变应用于原型输入，并测试结果值。我们提出了一种替代的白盒模糊测试方法，启发了最近在符号执行和动态测试生成中的一些争议。我们的方法记录了一个完整的输入测试的实际运行，象征性地评估了记录的跟踪，并收集了输入的约束，从而获取程序如何使用这些输入。收集到的约束被逐一否定，并用约束求解器求解，产生在程序中运行不同控制路径的新输入。借助于代码覆盖最大化启发式的设计，尽可能快地找到缺陷，重复此过程。我们已经在SAGE可扩展，自动，引导执行中实现了这种算法，这是一种采用x86指令级跟踪和仿真来实现任意文件读取Windows应用程序的白盒模糊的新工具。我们描述了使动态测试生成规模到大型输入文件和长达数亿个指令的长执行轨迹所需的关键优化。然后，我们将介绍几个Windows应用程序的详细实验。值得注意的是，没有特定格式的SAGE，SAGE检测到MS07-017 ANI漏洞，这是一个广泛的黑匣子模糊和静态分析工具错过的漏洞。此外，尽管SAGE还处于开发初期阶段，但SAGE已经在大型运送的Windows应用程序中发现了30多个新错误，包括图像处理器，媒体播放器和文件解码器。这些错误中的几个是可能利用的内存访问冲突。

# 介绍

由于“浏览器错误月”在2006年7月的每一天发布了一个新的错误[25]，因此，模糊测试已成为在大型应用中发现严重安全缺陷的快速且成本选择的方法。模糊测试是一种黑盒随机测试的形式，它随机突变输入结果，并对结果数据进行程序测试[13,30,1,4]。在某些情况下，语法用于生成良好的输入，这也允许编码应用程序特定的知识和测试启发式。虽然模糊测试可以非常有效，但黑盒测试方法的局限性是众所周知的。例如，如果x是随机选择的32位输入值，那么条件语句“if（x = = 10）”的那个分支就只有232个机会中有一个被执行。这直观地解释了为什么随机测试通常提供低代码覆盖[28]。在安全上下文中，这些限制意味着潜在的严重的安全漏洞（如缓冲区溢出）可能会被错过，因为包含错误的代码甚至没有被执行。

我们提出了一个概念上简单但不同的白盒模糊测试方法。这项工作受到系统动态测试生成的最新进展的启发[16,7]。从固定输入开始，我们的算法以符号方式执行程序，从沿途遇到的条件语句收集输入约束。收集到的约束然后用约束求解器系统地否定和解决，产生在程序中执行不同执行路径的新输入。使用具有覆盖最大化启发式的新颖的搜索算法来重复该过程，旨在尽可能快地找到缺陷。例如，输入x = 0上的上述片段的符号执行产生约束x = 10。一旦这个约束被否定和解决，它产生x = 10，这给我们一个新的输入，导致程序跟随给定条件语句的分支。这允许我们对安全漏洞进行运行和测试附加代码，即使没有输入格式的特定知识。此外，这种方法自动发现并测试程序员可能无法正确分配内存或操纵缓冲区的“安全漏洞”，从而导致安全漏洞。

理论上，系统的动态测试生成可以导致完整的程序路径覆盖，即程序验证[16]。然而，实际上，搜索通常是不完整的，因为被测程序中的执行路径的数量是巨大的，因为符号执行，约束生成和约束求解必然是不精确的。 （关于为什么会出现这种情况，请参见第2节）因此，我们被迫探讨实际的权衡，本文介绍了我们认为是一个特别的甜蜜点。事实上，我们的具体方法在以前经过很好测试的大型应用中发现新的缺陷已显着有效。事实上，我们的算法发现了许多缺陷事件，我们必须解决缺陷分类问题（参见第4节），这在静态程序分析和黑匣子模糊中是常见的，但直到现在在动态测试生成的背景下还没有面临[ 16,7,31,24,22,18]。我们的另一个新颖性就是我们比以前在动态测试生成中测试更大的应用程序[16，7，31]。

我们在SAGE中实现了这种方法，可扩展，自动化，引导型E：xecution是x86 Windows应用程序的全程白盒文件模糊工具。虽然我们目前的工具专注于文件读取应用程序，但这些原则也适用于面向网络的应用程序。如上所述，SAGE能够发现超出黑盒子模糊器范围的错误。例如，没有任何特定格式的知识，SAGE能够检测到关键的MS07-017 ANI漏洞，这是通过广泛的黑盒模糊和静态分析做不到的。我们的工作有三个主要贡献：

•第2节介绍了一种用于系统测试生成的新型搜索算法，该算法针对具有大量输入文件的大型应用程序进行了优化，并展示了很长的执行跟踪，其中搜索必须不完整

•第3节讨论SAGE的实现，其符号执行算法背后的工程选择以及关键优化技术，使其能够以数亿个指令扩展到程序跟踪；

•第4节描述了我们在SAGE方面的经验，我们给出了发现缺陷的例子，并讨论了各种实验的结果。

# 一个白盒模糊测试算法

## 背景：动态测试生成

考虑图1所示的程序。该程序以4个字节为输入，当变量cnt的值在函数顶端结束时大于或等于3时，会包含一个错误。 使用4个输入字节的随机值运行程序不太可能发现错误：有5个值导致4个字节的2（8 \* 4）个可能值中的错误，即概率约为1/230 随机测试打错误，包括黑匣子模糊。

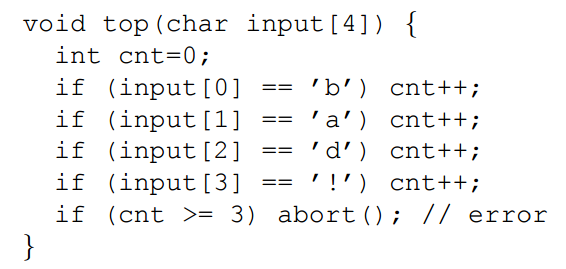


图 1 示例程序

这个问题是随机测试的典型代码：难以产生通过其所有可能执行路径驱动程序的输入值。

相反，白盒动态测试生成可以很容易地找到该程序中的错误：它包括执行程序，从一些初始输入开始，执行动态符号执行，以收集从执行中的分支语句中的谓词收集的输入的约束，然后使用约束求解器来推断先前输入的变体，以便将程序的下一个执行转向替代程序分支。这个过程被重复，直到给定的特定程序语句或路径执行[22,18]，或者直到程序的所有（或许多）可行程序路径被执行[16,7]。

对于上面的例子，假设我们开始运行函数top，其中最初的4个字母的字符串good。图2显示了函数top的所有可行程序路径的集合。最左边的路径代表程序输入good的第一次运行，对应于程序路径p，包括程序中所有条件if语句的所有4个其他分支。该路径的叶标记为0，表示运行结束时变量cnt的值。与正常执行交织在一起，符号执行收集谓词i0 != b，i1 != a，i2 != d和i3 != !,根据条件评估的方式，其中i0，i1，i2和i3分别表示输入变量input [0]，input[1]，input[2]和input[3]对应内存地址。

路径约束φp = (i0 != b,i1 != a,i2 != d,i3 != !)表示输入向量的等价类，即通过刚刚执行的路径驱动程序的所有输入向量。为了强制程序通过不同的等价类，可以计算一个不同的路径约束的解，比如说，通过否定最后一个谓词，得到的(i0 != b,i1 != a,i2 != d,i3 != !)当前的路径约束。该路径约束的解决方案是(i0 = b,i1 = a,i2 = d,i3 = !)。运行程序top与这个新的输入”goo!”产生图2中第二个最左边路径描述的新程序路径。通过重复这个过程，可以执行该程序的所有16个可能执行路径的集合。

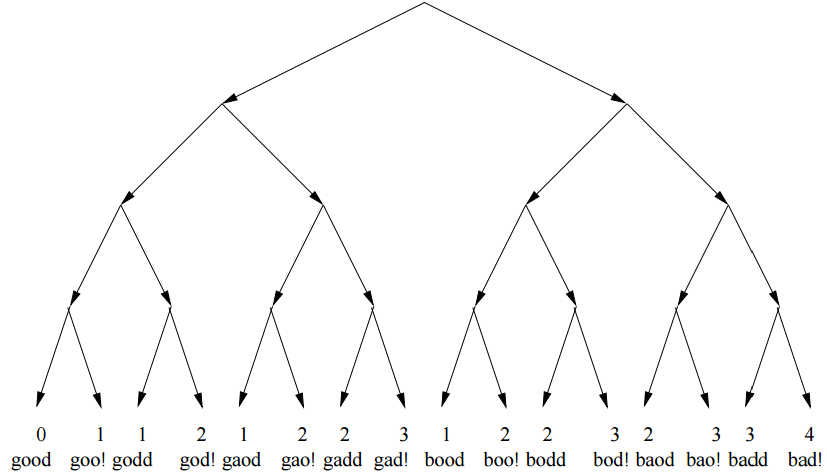


图 2对图1中变量cnt结果与输入字符串关系的搜索路径图

如果这个系统的搜索以深度顺序执行，这16个执行在图中从左到右进行了探索。第八次运行时，第一次使用cnt == 3进入abort错误，第九次运行完成完整的分支/块覆盖。

## 局限性

如上面简要描述的系统动态测试生成[16,7]有两个主要限制。

**路径爆炸**：系统地执行所有可行的程序路径不会扩展到大而实际的程序。通过在组成上执行动态测试生成[14]可以缓解路径爆炸[14]，通过隔离测试功能，将测试结果编码为使用功能输入前提条件和输出后置条件提取的功能摘要，然后在测试更高时重新使用这些摘要级功能。虽然在软件测试中使用摘要似乎是有希望的，但是在有限搜索周期内（例如，一天晚上）仍然会遇到具有数百个指令的大型应用程序测试大量应用程序时实现全面的路径保护，即使是总结。

**不完美的符号执行**：由于复杂的程序语句（指针操作，算术运算等），对程序的符号执行势必是不精确的，并且调用操作系统和库函数，这些操作系统和库函数难以或不可能推理到sym-以合理的成本，足够高的精度。每当不可能执行符号时，可以使用具体的值来简化约束并进行简化的部分符号执行[16]。随机化也可以帮助建议具体值，无论何时自动推理是困难的。每当实际执行路径与给定输入向量的符号执行预测的程序路径不匹配时，我们说发生了分歧。可以通过将预测的执行路径记录为位向量（每个条件分支结果一个位）并检查在随后的测试运行中实际采用预期路径来检测分歧。

|  |
| --- |
| 1. Search(inputSeed){ 2. inputSeed.bound = 0; 3. workList = {inputSeed}; 4. Run&Check(inputSeed); 5. while (workList not empty) {//new children 6. input = PickFirstItem(workList); 7. childInputs = ExpandExecution(input); 8. while (childInputs not empty) { 9. newInput = PickOneItem(childInputs); 10. Run&Check(newInput); 11. Score(newInput); 12. workList = workList +newInput; 13. } 14. }   15 } |

图 3 搜索算法

## 生成搜索

我们现在提出一种新的搜索算法，旨在解决这些基本的实际局限。具体来说，我们的算法具有以下突出特点：

•它旨在系统地，部分地探索用大量输入（数千个符号变量）执行的大型应用程序的状态空间，并具有许多深入的路径（数亿个指令）；

•最大限度地提高从每个符号执行（在我们的上下文中长而昂贵）中产生的新测试数量，同时避免搜索中的任何冗余；

•它使用启发式方法尽可能快地最大化代码覆盖率，目的是更快地发现错误；

•对分歧有弹性：每当发生分歧时，搜索能够恢复并继续。

这种新的搜索算法在图3和图4中分为两部分。图3的主要Search过程是标准的。它将初始输入inputSeed放在一个workList（行3）中，并运行程序来检查在第一次执行期间是否检测到任何错误（第4行）。然后通过选择一个元素（第6行）并扩展它（第7行）来处理workList中的输入（第5行），以生成具有下面在图4中描述的函数ExpandExecution的新输入。对于这些childInputs，被测程序是运行该输入。检查这个执行错误（第10行），并在被添加到按照这些分数排序的workList（第12行）之前被分配了一个Score（第11行），如下所述。

|  |
| --- |
| 1. ExpandExecution(input) { 2. childInputs = {}; 3. //symbolically execute (program,input) 4. PC = ComputePathConstraint(input); 5. for (j=input.bound； j < |PC|;j++) { 6. if ( (PC [0.. (j-1)] and not (PC[j])) has a solution I){ 7. newInput = input+ I; 8. newInput.bound =j; 9. childInputs = childInputs+ newInput; 10. } 11. return childInputs;   12 } |

图 4 计算新子代

我们的搜索算法的主要原创性就是如图4所示扩展的子代。给定input（第1行），ExpandExecution函数用该input符号地执行被测程序，并生成路径约束PC（第4行），如前所述。 PC是| PC |的连接约束，每个对应于程序中的条件语句，并使用表示输入参数值的符号变量表示（参见[16,7]）。然后，我们的算法尝试扩展路径约束中的每个约束（位置j大于或等于最初为0的input.bound参数）。这是通过检查在第j个约束PC [0 ..(j-1)]之前的路径约束的部分和第j个约束的not(PC [j])之间的连接是否是有道理。如果是这样，则使用此新路径约束的解决方案I来更新先前的解决方案input，同时保留路径约束中未涉及的输入参数的值（该更新由第7行的input+ I表示）。生成的新输入值被保存以供将来评估（第9行）。

换句话说，从初始输入inputSeed和初始路径约束PC开始，图3和图4中所示的新的搜索算法将尝试扩展所有在 PC中的约束|PC|，而不是最后一个具有深度优先搜索的约束，或者第一个具有广度优先搜索的约束。为了防止这些child子搜索冗余地探索搜索空间的重叠部分，使用bound来限制子搜索之上的子搜索在其父搜索开始的分支之上的回溯。因为每次执行都是通常的扩展了许多子代，我们称这样的搜索顺序为一个**生成搜索**。

再次考虑图1所示的程序。假设初始输入是4个字母的字符串good，则图2中最左侧的路径表示该输入上程序的第一个运行。从这个parent运行中，代数搜索产生四个第一代子代，它们对应于叶子被标记为1的四个路径。实际上，这四个路径都对应于在最左边的parent运行的原始路径约束中取消一个约束。这些第一代执行路径中的每一个依次可以通过图4的过程进行扩展，以生成（零个或多个）第二代子代。其中有六个，每个都有图2中他们（第一代）父代右侧的叶子标签2。通过重复这个过程，函数top的所有可行的执行路径最终都被精确地生成一次。对于该示例，变量cnt的值恰好表示每次运行的生成次数。

  由于图4的ExpandExecution过程扩展了当前路径约束（低于当前bound）的所有约束，而不是仅仅一个约束，因此可以最大化从每个符号执行生成的新测试输入的数量。尽管如图1所示，尽可能详细地探索小程序的所有执行路径时，这种优化可能并不重要，重要的是符号执行需要很长时间，对于大型应用程序来说，走遍所有执行路径基本无望。这一点将在第3节进一步讨论，并通过第4节中报告的实验进行说明。

在这种情况下，我们希望尽可能利用初始输入执行的第一个符号执行，并系统地探索其所有第一代子代。如果初始输入形成良好，则此搜索策略最有效。实际上，更有可能运用更多的程序代码，因此产生更多的约束被否定，从而产生更多的子代，如第4节中的实验所示。对第一个输入的重要性与所做的相似与传统的黑盒模糊测试，因此我们使用白盒模糊测试为本文引入的搜索技术。

通过使用启发式尝试尽可能快地最大限度地扩大块覆盖范围，第一个父级运行的子级的扩展本身就被优先考虑，希望能够更快地找到更多的错误。函数Score（图3中的第11行）计算与所有以前的运行相比执行newInput获得的增量块覆盖。例如，触发执行的newInput将被分配100个分数。接下来，（第12行），newInput根据分数被插入到workList中，最高的分数位于列表。注意，所有的子代们都可以相互竞争，接下来要扩大，不管他们的世代是多少。

我们的块覆盖启发式与“优先搜索”的EXE有关[7]。然而，整体搜索策略是不同的：当EXE使用深度优先搜索时，偶尔选择下一个子代来探索使用块覆盖启发式，一代检索测试每个扩展执行的所有子项，并挑选从结果的workList对其整个运行进行评分最好的一个。

用函数Score计算的块覆盖启发式有助于处理前面部分定义的分歧，即执行与预期路径约束分离，接下来要采取。单一发散的发生损害了搜索的完整性，但这不是实践中的主要问题，因为搜索对于非常大的搜索空间来说肯定是不完整的。一个更令人担忧的问题是分歧可能会阻止搜索取得任何进展。例如，从路径p分散到先前探测的路径p’的深度优先搜索将在该路径p’和随后的发散运行p之间永久地循环。相比之下，我们的代际搜索可以容忍分歧，并可以从这种病理情况中恢复。事实上，每次运行都会产生许多子代，而不是一个单一的子代，而不是一个深度搜索，而如果一个子代运行p分歧到前一个p’，那么该小孩p将有一个零分，因此被放在最后workList不损害其他非发散children的扩张。处理分歧是我们的处理大型应用程序的算法的另一个重要特征，因为符号执行必然是不完美/不完整的，如第4节所示。

最后，我们注意到，代际检索并行化，因为子代可以独立检查和评分；只需要共享workList和整体块覆盖。

# SAGE系统

上一节中提出的代数搜索算法已经在一个名为SAGE的新工具中实现，它代表可扩展，自动，引导执行。 SAGE可以通过将从文件读取的字节视为符号输入来测试在Windows上运行的任何文件读取程序。 SAGE的另一个关键的新颖之处在于它在x86二进制级别执行程序跟踪的符号执行。 该部分通过争论SAGE如何处理各种大型生产应用程序来证明这一设计选择。 这种设计决策引发了与源代码级符号执行所面临的挑战。 我们描述这些挑战，并展示在实施过程中如何处理这些挑战。 最后，我们概述了在扩展到大型程序时至关重要的关键优化。

## 系统架构

SAGE通过重复四种不同类型的任务执行代数搜索。 Tester任务通过在测试输入上执行被测程序来实现Run&Check功能，并寻找异常事件，例如访问冲突异常和极端内存消耗。只有当Tester任务没有遇到任何此类错误时，后续任务才会进行。如果Tester检测到错误，则会保存测试用例并进行自动分类，如第4节所述。

Tracer任务再次在同一个输入文件上运行目标程序，此时记录将由以下任务使用的运行日志，以重播程序Executionoffline。 Thistaskusesthe iDNA框架[3]在机器指令级别收集完整的执行跟踪。

CoverageCollector任务重播记录的执行，以计算在运行期间执行哪些基本块。 SAGE使用此信息来实现上一节中讨论的功能评分。

最后，Symbolic Executor任务通过再次重播记录的执行来实现第2.3节的函数ExpandExecution，此时收集输入相关的约束并使用con-straintsolver Disolver [19]生成新的输入。

CoverageCollector和Symbolic Executor任务都构建在跟踪重放框架TruScan [26]之上，TruScan [26]消耗由iDNA生成的跟踪文件，并且虚拟地重新执行记录的运行。 TruScan提供了几个功能，大大简化了符号执行。这些包括指令解码，提供程序符号信息的接口，监视各种input/output系统调用，跟踪堆和堆栈帧分配，以及通过程序结构跟踪数据流。

## x86下基于跟踪的约束生成

SAGE的约束生成与以前的动态测试生成实现不同[16,31,7]有两个主要方面。首先，SAGE采用基于机器码的方法，而不是基于源代码的方法，主要有三个原因：

**多种语言和构建过程。**基于源的工具必须支持被测程序的特定语言，编译器和构建过程。将该装置适应于新的语言，编译器或构建工具有很大的前期成本。覆盖许多应用程序在一个大型公司开发，具有各种不兼容的构建过程和编译器版本，这是一个后勤噩梦。相比之下，基于机器码的符号 - 执行引擎虽然复杂，但每个架构只需要执行一次。正如我们将在第4节中看到的，这个选择让我们将SAGE应用于大量的生产软件应用程序。

**编译器和后构建转换。**通过执行实际运行的二进制代码的符号执行，SAGE可以捕获目标程序中的错误，也可能捕获编译和后处理工具（例如代码混淆器和基本块变换器），可能会引入来源和最终产品的语义之间的微妙差异。

**源的不可用性**。我们可能难以获得第三方组件的源代码，或者甚至来自同一组织的不同组的组合。基于源的仪器也可以自行修改或JITed代码。 SAGE通过在机器代码级别工作来避免这些问题。虽然源代码确实有关于机器代码级别不能立即显示的类型和结构的信息，但我们不需要SAGE的路径探索的这一信息。

另外，SAGE采用基于离线跟踪的约束生成，而不是在线检测。通过在线生成，通过静态注入的仪器代码或借助动态二进制仪器工具（如Nirvana [3]或Valgrind [27]（Catchconv是后一种方法的一个例子[24]）执行程序，生成约束。）SAGE采用离线跟踪约束生成有两个原因。首先，单个程序可能涉及大量的二进制组件，其中一些程序可能受到操作系统的保护或混淆，从而很难用仪表版本替换它们。第二，大型目标程序中固有的非确定性使得调试在线约束生成变得困难。如果约束生成引擎出现问题，我们不太可能重现导致问题的环境。相比之下，SAGE中的约束生成是完全确定性的，因为它在执行跟踪中起作用，捕获在记录运行期间遇到的所有非确定性事件的结果.

## 约束生成

SAGE维护由一对存储器表示的程序的具体和符号状态，该对存储将每个存储器位置和寄存器分别关联到字节大小的值和符号标签。符号标记是表示输入值或某些指令的函数的表达式。 SAGE支持多种标签：input(m)表示m个字节输入； c表示常数； t1 op t2表示由标签t1和t2表示的值的一些算术或按位操作的结果，op代表操作类型；序列标签t0...tn,（其中n = 1或n = 3）代表字或双字大小的值对应的分组标签； subtag(t,i) ，i属于{0 ... 3}，对应于由t表示的字或双字大小的值中的第i个字节，请注意，SAGE当前没有引用符号指针解引用，SAGE定义了一个新的符号变量，每个非常数的符号标签。如果没有混淆，我们不会在本节的其余部分区分标签和其关联的符号变量。

由于SAGE重放记录的程序跟踪，它会根据每个访问指令的语义来更新具体和符号存储。除执行符号标签传播外，SAGE还对输入值产生约束。约束是符号变量之间的关系；例如，给定对应于标签input(4)的变量x，约束x <10表示输入的第五个字节小于10的事实。

当算法遇到与输入相关的附加跳转时，它创建一个约束，建模分支的结果，并将其添加到由到目前为止遇到的约束组成的路径约束。

以下简单示例说明了跟踪符号标签和收集约束的过程。

|  |
| --- |
| # read 10 byte file into a  # buffer beginning at address 1000  mov ebx, 1005  mov al, byte [ebx]  dec al # Decrement al  jz LabelForIfZero # Jump if al == 0 |

该片段的开始使用系统调用从地址1000读取一个10字节的文件到内存范围。为简洁起见，我们省略了实际的指令序列。 作为重播这些指令的结果，SAGE通过将地址1000 ... 1009分别与符号标签input(0)...input(9)相关联来更新符号存储。

两个mov指令具有将第五个输入字节加载到寄存器al中的作用。 在重播这些指令后，SAGE用符号input(5)的映射更新符号存储。 最后两个指令的效果是减少al，并且如果递减值为0，则将条件跳转到LabelForIfZero。作为重播这些指令的结果，根据分支的结果，SAGE将添加两个约束之一t = 0或t != 0，其中t =input(5)-1。如果采用分支，则添加前一个约束； 后者如果没有分支。

这导致我们成为从x86机器指令流生成约束的关键难题之一 ­—— 处理条件表达式的两阶段性质。 当进行比较时，不知道在稍后执行条件跳转指令之前如何使用它。 处理器有一个专门的寄存器EFLAGS，用于打包CF，SF，AF，PF，OF和ZF等状态标志的集合。 这些标志如何设置由各种指令的结果决定。 例如，CF——EFLAGS的第一位­——是由各种算术运算影响的进位标志。 特别地，通过减法指令将其设置为1，其第一个参数小于第二个参数。 ZF是位于EFLAGS第七位的零标志； 如果其参数相等，则由减法指令设置。 更复杂的事情，一些指令，如sete和pushf直接访问EFLAGS。

对于EFLAGS，听起来SAGE定义了描述根据(f0... fn-1)约束设置其位的n位值的{f0... fn-1}形式的位扫描器标签。 在上面的例子中，当SAGE重复dec指令时，它更新了al和EFLAGS的符号存储映射。 前者映射到input(5)-1； 后者到位向量标签（t <0 ... t = 0 ...）其中t =input(5)-1，并且两个所示的约束位于位向量的偏移量0和6处——偏移量对应于EFLAGS寄存器中CF和ZF的位置。

   另一个普遍的x86实践涉及到字节，字和双字对象之间的转换。 即使被测程序的主要代码不包含显式转换，它也将始终调用一些运行时库函数，如atol，malloc或memcpy。

SAGE通过子标签和序列标签的帮助来实现对cast的声音处理。

这通过以下示例来说明。

|  |
| --- |
| mov ch, byte [...]  mov cl, byte [...]  inc cx # Increment cx |

假设两个mov指令读取与符号标签t1和t2相关联的地址。 在SAGE重放这些指令之后，它用映射cl -> t1和ch -> t2来更新符号存储。 下一个指令将cx——包含cl和ch的16位寄存器分别作为低字节和高字节递增。 在增量之前，cx的内容可以由序列标签(t1，t2 )表示。 然后，增量的结果是字大小的标签t =((t1,t2)+ 1)。 为了最终确定inc指令的效果，SAGE用字节大小的映射cl-> subtag(t，0)和ch-> subtag(t，1)来更新符号存储。 SAGE通过约束x = x'十256\*x’’对子标记关系进行编码，其中字大小的符号变量x对应于t，两个字节大小的符号变量x'和x”对应于subtag(t，0) 和subtag(t，1)。

## 约束优化

SAGE采用了一些优化技术，其目标是提高约束生成的速度和内存使用率：标签缓存确保结构上等价的标签映射到同一物理对象；不相关的约束消除通过删除不与否定约束共享符号变量的约束来减少约束求解器查询的大小；局部约束缓存跳过一个约束，如果它已经被添加到路径约束；翻转限制建立从特定程序指令生成的约束可以翻转的最大次数；具体化将涉及按位和多重运算符的符号标签减少到相应的混合值。

这些优化在动态测试生成中是相当标准的。本节的其余部分描述了约束sub-sumption，我们发现我们发现对分析结构化文件解析应用程序特别有用的优化。

约束收敛优化跟踪从给定分支指令生成的约束。当创建新的约束f时，SAGE使用快速语法检查来确定f是否绝对意味着或者是由相同指令生成的另一个约束明确暗示。如果是这种情况，则隐含约束从路径约束中移除。

包容优化对处理诸如各种图像解析器和媒体播放器等结构化文件的许多程序具有关键影响。例如，在第4节描述的Media 2搜索之一中，我们观察到由于包含的约束数量减少了10倍。没有这种优化，SAGE耗尽内存，并用大量的冗余查询来遏制约束求解器。

我们借助下面的例子来看看约束包含优化的细节：

|  |
| --- |
| mov cl, byte [...]  dec cl # Decrement cl  ja 2 # Jump if cl > 0 |

该代码片段将一个字节加载到cl中，并将其递减到一个循环，直到它变为0.假设由mov指令读取的字节映射到符号标签t0，则3.3节中概述的算法将生成约束t1> 0 , ... , tk-1> 0和tk <=0，其中k是加载字节的具体值，对于i属于 {1 ... k}，ti + 1 = ti-1。这里，内存成本在循环迭代次数中是线性的，因为每次迭代产生新约束和新的符号标记。

包含技术允许我们删除第一个k-2约束，因为它们被以下约束所暗示。我们仍然必须坚持线性数量的符号标签，因为每个符号标签都是根据前面的标签定义的。为了实现恒定的空间行为，必须结合常量折现标注(t-c)-1 = t -(c+1)来执行约束收敛。具有约束收敛和常量折叠的算法对上述分段的净效应是具有两个约束t0-(k-1)> 0和t0-k <0的路径约束。

另一个障碍来自于多字节标签。考虑以下循环，其类似于上述循环，除了字节大小的寄存器cl被字大小的寄存器cx替换。

|  |
| --- |
| mov cx, word [...]  dec cx # Decrement cx  ja 2 # Jump if cx > 0 |

假设由mov指令读取的两个字节被映射到标签t0’和t0’’，则该片段产生约束s1> 0，sk-1> 0和sk <0，其中，si + 1 = <ti’ , ti’’> -1且ti’ = subtag(si , 0)和ti’’ = subtag(si , 1)，其中i属于{1 ... k}。 恒定折叠变得困难，因为每个循环迭代引入语法上唯一的但是非对称的冗余字长序列标签。 SAGE通过顺序标签简化来解决这个问题，它将<subtag(t , 0) , subtag( t, 1) >重写为t，避免复制等效标签并启用常量折叠。

约束包络，恒定折叠和序列标签简化足以保证上述片段生成约束(t0’ , t0’’)-(k – 1)> 0和(t0’ , t0’’)-k<0。更一般来说，这三种简单的技术使SAGE能够有效地模糊实际结构化文件解析应用程序，其中输入绑定循环模式是普遍的。

# 实验

我们首先描述我们SAGE的初步经验，其中包括SAGE发现的黑匣子模糊的错误。 受这些经验的启发，我们对两种媒体解析应用程序的SAGE行为进行了更系统的研究。特别是，我们专注于搜索的起始输入文件的重要性，我们的代际搜索与深度优先搜索的效果以及我们的块覆盖启发式的影响。 在某些情况下，我们怀疑有关具体应用测试的细节，因为错误仍在修复中。

## 初步实验

MS07-017。 2007年4月3日，Microsoft发布了解析ANI格式动画游标的代码的带外关键安全补丁。该漏洞最初于2006年12月由Deter- mina Security Research的Alex Sotirov向Microsoft报告，然后在野外使用漏洞代码后公布[32]。这只是2006年1月以来微软发布的第三个带外修补程序，表明该漏洞的严重性。Microsoft SDL Policy Weblog指出，此代码的广泛的黑盒模糊测试未能发现错误，现有的静态分析工具无法在没有过多误报的情况下找到该错误[20]。相比之下，SAGE在从形成良好的ANI文件开始的几个小时内合成了一个新的输入文件。

更详细地说，这个漏洞是由不完备的补丁导致的，MS05-006也涉及到ANI分析代码。这个错误的根本原因是无法估计从ANI文件中的anih记录读取的大小参数。不幸的是，MS05-006的补丁是不完整的。只检查第一个anih记录的长度。如果一个文件具有36个字节或更少的初始记录，则检查被满足，但是在所有的anih记录上调用了图标加载功能。不检查第二个和后续记录的长度字段，因此这些记录中的任何一个可能会触发内存损坏。

因此，测试用例需要至少两个anih记录来触发MS07-017错误。 SDL策略Weblog表示黑盒模糊测试发现MS07-017的失败是因为用于黑盒测试的所有种子文件只有一个记录，因此没有一个生成的测试用例会打破MS05- 006补丁。虽然当然可以写一个生成这样的黑盒子模糊测试用例的语法，但这需要努力，并不会超出单一的ANI格式。

|  |
| --- |
| RIFF...ACONLIST RIFF...ACONB  B...INFOINAM.... B...INFOINAM....  3D Blue Alternat 3D Blue Alternat  e v1.1..IART.... e v1.1..IART....  ................ ................  1996..anih$...$. 1996..anih$...$.  ................ ................  ................ ................  ..rate.......... ..rate..........  ..........seq .. ..........seq ..  ................ ................  ..LIST....framic ..anih....framic  on......... .. on......... .. |

图 5 左侧是用于搜索的ANI文件前缀的ASCII码值,右侧是SAGE生成器为ms07-017生成的崩溃.注意图中把LIST更改为anih的位置.

相比之下，尽管不了解ANI格式，SAGE可以从一个具有一个anih记录的良好形成的ANI文件开始，产生出现MS07-017的崩溃。我们的种子文件是从一个完善的ANI文件库中任意挑选的，我们使用一个叫做user32 .dll的小型测试驱动程序来解析测试用例ANI文件。初始测试用例在解析1279939个总指令超过10072个符号输入字节后，产生了具有341个分支约束的路径约束。 SAGE然后在7小时36分钟的搜索和7706个测试用例之后，以深度72创建了一个崩溃的ANI文件，使用一颗2 GHz AMD Opteron 270双核处理器，运行32位Windows Vista，内存为4 GB。图5显示我们的种子文件的前缀与SAGE生成的测试用例崩溃。图6显示了该测试运行的进一步统计信息。

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Test | # SymExec | SymExecT | Init. | PC | | # Tests | Mean Depth | Mean # Instr. | Mean Size |
| ANI | 808 | 19099 | 341 | 11468 | 178 | 2066087 | 5400 |
| Media 1 | 564 | 5625 | 71 | 6890 | 73 | 3409376 | 65536 |
| Media 2 | 3 | 3457 | 3202 | 1045 | 1100 | 271432489 | 27335 |
| Media 3 | 17 | 3117 | 1666 | 2266 | 608 | 54644652 | 30833 |
| Media 4 | 7 | 3108 | 1598 | 909 | 883 | 133685240 | 22209 |
| Compressed File | 47 | 1495 | 111 | 1527 | 65 | 480435 | 634 |
| OfficeApp | 1 | 3108 | 15745 | 3008 | 6502 | 923731248 | 45064 |

图 6对 7个应用程序的测试结果示例

**压缩文件格式。**我们向内部测试团队发布了SAGE的一个字体，以寻找处理压缩文件格式的代码中的错误。该文件格式的解析代码已经用blackbox fuzzing工具进行了广泛的测试，但是SAGE发现了两个严重的新错误。第一个bug是一个堆栈溢出。第二个错误是一个无限循环，导致处理应用程序消耗近100％的CPU。这两个错误在一周的时间内被修复，表明产品团队认为这些错误很重要。图6显示了在该测试代码上运行的SAGE的统计信息，并使用格式良好的压缩文件进行种子处理。 SAGE在解析由同一个团队测试的不同格式的格式不正确的文件时，也发现了两次单独的崩溃：相应的错误也在一周内提交。

**媒体文件解析。**我们将SAGE应用于四种广泛使用的媒体文件格式的解析器，我们将称之为“Media 1”，“Media 2”，“Media 3”和“Media 4”。通过几个测试会话，SAGE发现每个的这些媒体文件导致九个不同的错误报告。例如，由于程序将零字节复制到缓冲区，然后从非零偏移读取，SAGE发现读取违规。此外，从100个零字节的种子文件开始，SAGE在1403个测试用例之后合成了一个崩溃的Media 1测试用例，展示了SAGE从代码中推断文件结构的功能。图6显示了从格式良好的文件开始时，对每个解析器的SAGE搜索的大小的统计。

**Office 2007应用程序。**我们使用SAGE成功合成了作为Office 2007一部分发货的大型应用程序的崩溃测试用例。在两个10小时的搜索过程中，SAGE生成了两个不同格式的文件，生成了4548个测试用例，其中43个崩溃了应用。到目前为止，我们研究的崩溃是NULL指针解引用错误，它们显示了SAGE如何能够成功地大规模地推广程序。图6显示了其中一个格式良好的文件的SAGE搜索的统计信息。

**图像分析**。我们使用SAGE在包含在各种其他应用程序中的媒体播放器中执行映像解析代码。虽然我们的初始运行没有发现崩溃，我们使用内部工具来扫描来自SAGE生成的测试用例的踪迹，并发现了几个未初始化的值使用错误。我们向测试团队报告了这些错误，将测试结果扩展成可重复的崩溃。这个经验表明，SAGE可以发现严重的错误，不会立即导致崩溃。

## 实验设置

**测试计划**。我们专注于Media 1和Media 2解析器，因为它们被广泛使用。我们针对Media 1解析器运行了SAGE搜索，其中包含五个“格式良好”的媒体文件，从一个测试媒体文件库中选择。我们还用五个“伪造”文件测试了媒体1：由100个零字节组成的伪造-1，由800个零字节组成的伪造2，由25600个零字节组成的伪造3，由100个随机生成的字节组成的伪造4，以及伪造 - 由800个随机生成的字节组成。对于这10个文件中的每一个，我们运行了一个10小时的SAGE搜索种子与该文件建立SAGE发现的基准数量的崩溃。如果一个任务在10个小时结束时进行，我们允许它完成，导致搜索时间稍长于10个小时。对于发现崩溃的搜索，我们将重新搜索SAGE搜索10小时，但禁用了我们的块覆盖启发式。我们重复了具有五个“格式良好”的Media 2文件和伪造1文件的Media 2解析器的过程。

每个SAGE搜索使用AppVerifier [8]配置为检查堆内存错误。无论何时发生这种错误，AppVerifier会在被测试的应用程序中强制“崩溃”。然后我们收集了崩溃的测试用例，种子输入覆盖的代码块的绝对数，以及在搜索过程中添加的代码块数。我们在四台机器上进行了实验，每台机器都配有两颗双核AMD Opteron 270处理器，运行速度为2 GHz。然而，在实验过程中，我们只使用一个核心来减少非确定性任务调度对搜索结果的影响。每台机器运行32位Windows Vista，配备4 GB RAM和250 GB硬盘。

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| stack hash | wff-1 | wff-3 | wff-4 | wff-5 |
| 790577684 | X | X | X | X |
| 825233195 | X | X |  | X |
| 795945252 | X | X | X | X |
| 1060863579 | X | X | X | X |
| 1043337003 |  |  | X |  |
| 808455977 |  |  |  | X |
| 1162567688 |  |  |  | X |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| stack hash | wff-1 | wff-2 | wff-3 | wff-4 | wff-5 | bogus-1 |
| 1867196225 | X | X | X | X | X |  |
| 2031962117 | X | X | X | X | X |  |
| 612334691 |  | X | X |  |  |  |
| 1061959981 |  |  | X | X |  |  |
| 1212954973 |  |  | X |  |  | X |
| 1011628381 |  |  | X | X |  | X |
| 842674295 |  |  |  | X |  |  |
| 1246509355 |  |  | X | X |  | X |
| 1527393075 |  |  |  | X |  |  |
| 1277839407 |  |  |  |  | X |  |
| 1392730167 |  |  |  |  | X |  |
| 1951025690 |  |  | X |  |  |  |

图 7 SAGE从357个Media1崩溃文件中发现的12个不同的栈hash(下图)

及88个Media2崩溃文件中的7个不同栈hash

**分诊**。因为SAGE搜索可能会产生许多不同的测试用例，这些测试用例会出现相同的错误，所以我们通过崩溃的堆栈哈希来“存储”崩溃文件，其中包括故障指令的地址。相同的错误可能由具有不同堆栈散列的程序路径可达到相同的根本原因。我们的实验总是报告不同的堆栈散列。

**覆盖结果中的非确定性。**作为我们实验的一部分，我们测量了测试运行期间封锁的绝对数量。我们观察到，在同一程序上运行相同的输入可能导致初始覆盖率略有不同，甚至在同一台机器上。我们相信这是由于与我们的测试应用程序使用的DLL的加载和初始化相关联的非确定性。

## 观测与结果

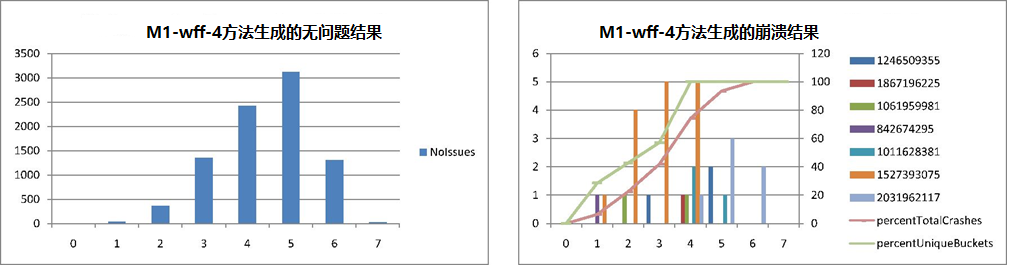


图 8 配合wff-4的对Media1生成的崩溃及测试用例直方图

附录显示了我们的实验结果表。在这里我们评论一些一般观察。我们强调，这些观察来自两个申请的样本量有限，应谨慎。

**符号执行缓慢。**我们测量了在每次搜索期间执行符号执行的总时间。我们观察到，单个符号执行任务比测试或跟踪程序慢许多倍。例如，使用wff-3播种的Media 2搜索中的符号执行任务的平均时间为25分30秒，而测试Media 2文件需要几秒钟。同时，我们还可以看到，只有一小部分的搜索时间用于执行符号执行，因为每个任务生成了许多测试用例；在Media 2 wff-3的情况下，只有25％的搜索时间花在符号执行中。这显示了一代代搜索如何有效地利用昂贵的符号执行任务。这也显示了将测试任务与更昂贵的符号执行器任务分离的好处。

**广代搜索优于深度搜索。**我们用深度优先搜索进行了几次运行。首先，我们发现当用伪造-1文件种子时，媒体1上的SAGE搜索显示出病理分歧（参见第2节），导致18分钟后过早终止搜索。经过进一步的检查，这种分歧证明是由于在路径约束中具体化了AND运算符。我们确实观察到用wff-2和wff-3播种的Media 2搜索的深度优先搜索运行10小时。深度优先搜索都没有发现崩溃。相比之下，当用wff-2播种的世代搜索没有发现任何崩溃时，用wff-3种植的世代搜索在4个桶中发现15个崩溃的文件。此外，深度优先搜索低于代码覆盖中的代数搜索：wff-2代寻找开始于51217个块，并添加12329，而深度优先搜索始于51476，仅添加398.对于wff-3，世代搜索开始于41726个块，并添加了9564个，而深度优先搜索开始于41703个块，并添加了244.这些不同的初始块覆盖来自上面提到的非确定性，但是添加的块的差异远大于开始覆盖关于代码覆盖的深度优先搜索的限制是众所周知的（例如，[23]），并且是由于搜索过于局部化。相比之下，一代人搜索探索了所有深度上的改变本地执行分支，同时探索程序的所有层。最后，我们看到，搜索时间的大部分时间用于深度优先搜索的符号执行，而不是代数搜索，因为每个测试用例需要一个新的符号执行任务。例如，对于以wff-3播种的Media 2搜索，在总共10小时35分钟内，对于18个测试用例，在符号执行中花费10小时27分钟进行深度优先搜索。请注意，从每个符号执行（如广度优先搜索）生成单个新测试的任何其他搜索算法都具有类似的执行概要文件，其中昂贵的符号执行不足，因此在给定固定时间预算的情况下执行相对较少的测试。

**分歧是常见的。**我们的基本测试设置没有测量分歧，所以我们运行了几个仪器测试用例来测量发散速率。在这种情况下，我们经常观察到分歧率超过60％。这可能是由于几个原因：在我们的实验设置中，我们将所有非线性运算（如乘法，除法和按位运算）进行具体化以提高效率，还有几个x86指令仍然不能模拟，我们不会将符号化指针的取消引用，跟踪符号变量可能不完整，我们不能像上面那样控制非确定性的所有来源。尽管如此，SAGE能够在实际应用中发现许多错误，表明我们的搜索技术是容忍这种差异的。

**恶意文件发现的错误很少。**我们收集了来自我们良好形态和假种子SAGE搜索的崩溃数据。每个种子文件发现的错误显示为图7中的堆栈散列。在Media 1,6中用作SAGE搜索种子的10个文件中，6个搜索中至少发现一个崩溃的测试用例，其中5个6个种子形成良好。此外，搜索引擎中发现的所有错误都是通过至少一个格式良好的文件来发现的。对于媒体2上的SAGE搜索，在测试的6个种子文件中，4个发现了至少一个崩溃的测试用例，并且都是格式正确的。因此，使用格式良好的文件作为模糊测试的起点的常规智慧也适用于我们的白盒方法。

**不同的文件找到不同的错误。**此外，我们观察到，没有一个格式良好的文件找到媒体1或媒体2的所有不同的错误。这表明，使用各种格式良好的文件对于发现不同的错误是重要的，因为每个搜索都不完整。

**发现的bug很浅。**对于每个种子文件，我们收集了搜索达到的最大代数。然后我们查看了哪一代搜索找到了最后一个独特的崩溃桶。对于Media 1搜索，用精心设计的文件播放搜索结果，发现4代以内的所有独特的错误，最大次数在5到7之间。因此，这些搜索发现的大多数错误都是shalloffw - 它们是可以在少数几代中达到。坠机查找媒体2搜索量达到了最大值3，所以我们没有观察到这个趋势。

图8显示了用wff-4种植的Media 1的生成的崩溃和非崩溃（“NoIssues”）测试用例的直方图。我们可以看到，执行的大多数测试是第4到第6代，但所有独特的错误都可以在第1至第4代中找到。在后来没有问题的情况下测试的测试用例数量很高，但是这些新的测试用例没有发现不同的新bug。在几乎所有的实验中，一直遵循这种行为，特别是直方图中显示的“钟形曲线”。这种世代搜索没有超越第七代，因为它仍然有许多候选输入测试在较小的世代中扩展，并且由于后代的许多测试具有较低的增量覆盖率分数。

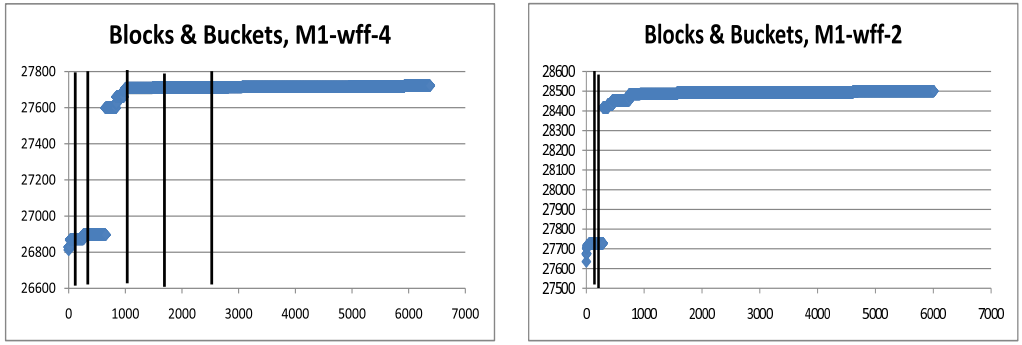


图 9 使用wff-4和wff-2为种子的Media1的代码覆盖和初始化对应的栈hash发现.左侧代表搜索中早期发现的多个版本不同的崩溃,其他竖线表示首先在这个位置找到独立崩溃

**覆盖和崩溃之间没有明确的相关性。**我们测试了每次测试运行后覆盖的绝对数，并将其与第一个测试用例的位置进行了比较，以显示崩溃的每个不同的堆栈散列。图9显示了用wff-4播种的Media 1搜索的结果；在搜索结果中，垂直条标记出新的堆栈哈希中发生了什么。虽然这个图表表明覆盖面的增加与发现新的错误有关，但是我们并没有普遍观察到这一点。其他几个搜索遵循wff-2图表显示的趋势：即使代码覆盖率稍后增加，它们也会在早期发现所有独特的错误。我们发现这是令人惊讶的，因为我们期望在探索的新代码和发现的新错误之间存在一致的相关性。在这两种情况下，尽管搜索结果仍然存在，但最后一个唯一的错误是通过搜索找到的。

**启发式块覆盖的影响。**我们比较了在使用我们的块覆盖启发式的测试运行之间搜索期间添加的块数量，从池中选择下一个子节点，而没有运行。我们观察到只有一个有利于启发式的弱势。例如，Media 2 wff-1搜索从覆盖的48494个块中添加了10407个块，而非启发式的情况则以48486个块开始，并添加了10633，几乎是一个死热。相比之下，Media 1 wff-1搜索开始于27659个块，并添加了701，而非启发式的情况从26962个块开始，仅添加了50.在10个搜索对中，3个启发式添加了更多块，而在其他的数字是足够接近几乎是一个领带。然而，如上所述，由于在代码覆盖范围内观察到的非确定性，这些数据是嘈杂的。

# 其他相关工作

模糊测试的其他扩展最近已经开始。其中大部分包括使用语法代表一组可能的输入[30,33]。概率权重可以分配给生产规则，并用作随机测试输入生成的启发式。也可以使用轻量级动态程序仪器收集的覆盖年龄数据自动定义或修改这些权重[34]。这些语法还可以包括用于角位案例的规则，以测试输入价值代码中的常见陷阱（如非常长的字符串，零值等）。使用输入语法使得可以编码关于被测应用程序的应用程序特定知识，以及测试指南，以便与其他的相比测试输入空间的特定区域。实际上，它们通常是使黑箱模糊化能够发现有趣的bug的关键，因为使用纯随机测试的概率通常非常小。但是手动编写语法是乏味，昂贵和缩小的。相比之下，我们的白框模糊方法不需要输入语法规范才能有效。然而，上一节的实验突出了给定搜索的初始种子文件的重要性。可以使用用于黑匣子模糊的语法来生成这些种子文件，以增加它们的多样性。另外请注意，由于符号执行和约束求解的代价，黑盒模糊可以生成并运行新的测试，比白盒模糊更快。因此，由于符号执行的不精确性，它可能会暴露出不会被白框模糊化的新路径。

如前所述，我们的方法建立在最近在[16,6]中引入并在[15,31,7,14,29]中扩展的系统动态测试生成的工作。主要区别在于，我们使用一种使用启发式方法的代数搜索算法在不完整的搜索中尽可能快地发现错误，并且我们测试大型应用程序，而不是单元测试小型，后者由基于跟踪的x86二进制符号执行而不是基于源的方法。这些差异可能解释了我们发现比以前报告的动态测试生成更多的错误。

我们的工作也不同于像[11]这样的工具，它们基于不产生或解决约束的动态污点分析，而只是强制分支被采取或不采取而不考虑程序状态。虽然对于一个人的审计员来说，这可能会导致虚假的程序崩溃的形式的错误，与真正执行中的“不可能发生的”数据。符号执行也是静态程序分析的关键组件，已经应用于x86二进制文件[2,10]。静态分析通常比动态分析和测试更有效，但不太精确，其互补性是众所周知的[12,15]。他们也可以合并[15,17]。静态测试生成[21]包括静态分析程序以尝试计算输入值，以沿着特定的程序路径驱动它，而不执行程序。相比之下，动态测试生成通过额外的运行时信息来扩展静态测试生成，因此更为通用和强大[16,14]。还提出了在生成漏洞签名的情况下，静态[5]或动态[9]的符号执行。

# 结论

我们引入了一种新的搜索算法，即代数搜索，用于容忍分歧的动态测试生成，并更好地利用昂贵的符号执行任务。我们的系统SAGE应用这种搜索算法来查找在Windows上运行的各种生产x86机器代码程序中的错误。然后，我们进行了实验，以更好地了解SAGE在两个媒体解析应用程序上的行为。我们发现使用各种格式良好的输入文件对于查找不同的错误很重要。我们还观察到，探索的世代是一个更好的预测因素，而不是一个测试用例是否会发现一个独特的新bug。特别地，发现的大多数独特的错误发现在少数几代之内。

虽然这些意见必须谨慎对待，但是来自有限的样本量，他们提出了一个新的搜索策略：而不是运行一定数量的小时数，可以从初始种子文件系统地搜索少数代，一旦这些测试用例耗尽，请转到新的种子文件。这个策略的承诺是，它可能会切断一代人搜索的“尾巴”，只能发现以前看到的bug的新实例，因此可能会在相同的时间内发现更多的错误。未来的工作应该尝试这种搜索方法，可能将其与我们的覆盖范围启发式组合应用于不同的种子文件，以避免重复探索相同的代码多次。调查的关键点是生成深度与代码覆盖率是否比仅代码覆盖率更好地指示何时停止测试。

最后，我们计划提高SAGE符号执行的精度和SAGE的约束求解能力。这将使SAGE能够发现目前无法达到的错误。

# 致谢

我们感谢Chris Marsh和Dennis Jeffries对SAGE的重要贡献，Hunter Hudson从一开始就争取到这个项目。 SAGE建立在TruScan团队的工作之上，其中包括Andrew Edwards和Jordan Tigani以及Disolver团队，包括Youssf Hamadi和Lucas Bordeaux，我们对此感激不尽。 我们感谢Tom Ball，Manuvir Das和Jim Larus对这个项目的支持和反馈。 各种内部测试团队在开发SAGE期间提供了宝贵的反馈意见，其中包括第4.1节所述的一些错误，我们感谢他们。 我们感谢Derrick Coetzee，Ben Livshits和David Wagner对我们论文草案的评论，Nikolaj Bjorner和Leonardo de Moura对约束解决的讨论。 感谢Chris Walker有关安全性的有益讨论。

# 引用

1. D. Aitel. The advantages of block-based proto­col analysis for security testing, 2002. http ： //[www.immunitysec.com/downloads/](http://www.immunitysec.com/downloads/) advantages\_〇f\_bl〇ck\_based\_analysis. html.
2. G. Balakrishnan and T. Reps. Analyzing memory accesses in x86 executables. *In Proc. Int. Conf. on Compiler Construction,* 2004. <http://www.cs>. wisc.edu/wpis/papers/cc04.ps.
3. S. Bhansali, W. Chen, S. De Jong, A. Edwards, and M. Drinic. Framework for instruction-level tracing and analysis of programs. In Second International Conference on Virtual Execution Environments VEE, 2006.
4. D. Bird and C. Munoz. Automatic Generation of Ran­dom Self-Checking Test Cases. IBffMSffystemsJo-ffmal, 22(3):229-245, 1983.
5. D. Brumley, T. Chieh, R. Johnson, H. Lin, and D. Song. RICH : Automatically protecting against integer-based vulnerabilities. In #DSSfSym户.on#et- work and Distributed System Security), 2007
6. C. Cadar and D. Engler. Execution Generated Test Cases: How to Make Systems Code Crash Itself. In Proceedings of SPIN’2005 (12th International SPIN Workshop on Model Checking offf Software), volume 3639 of Lecture Notes in Computer Science, San Fran­cisco, August 2005. Springer-Verlag.
7. C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. EXE: Automatically Generating Inputs of Death. In ACM CCS, 2006.
8. Microsoft Corporation. AppVerifier,2007. <http://www.microsoft.com/>

technet/prodtechnol/windows/ appcompatibility/appverifier.mspx.

1. M. Costa, J. Crowcroft, M. Castro, A. Rowstron, L. Zhou, L. Zhang, , and P. Barham. Vigilante: End- to-end containment of internet worms. In Symposium on Operating Systems Principles (SOSP), 2005.
2. M. Cova, V. Felmetsger, G. Banks, and G. Vigna. Static detection of vulnerabilities in x86 executables.

In Proceedings of the Annual Computer Security Ap­plications Conference (ACSAC), 2006.

1. W. Drewry and T. Ormandy. Flayer: Exposing ap­plication internals. In First Workshop On Offensive Technologies (WOOT), 2007.
2. M. D. Ernst. Static and dynamic analysis: synergy and duality. In Proceedings of WODA'2003 (ICSE Work­shop on Dynamic Analysis), Portland, May 2003.
3. J. E. Forrester and B. P. Miller. An Empirical Study of the Robustness of Windows NT Applications Using Random Testing. In Proceedings of the 4th USENIX Windows System Symposium, Seattle, August 2000.
4. P. Godefroid. Compositional Dynamic Test Generation.In Proceedings of POPL’2007 (34th ACM Sym­posium on Principles of Programming Languages), pages 47-54, Nice, January 2007.
5. P. Godefroid and N. Klarlund. Software Model Check­ing: Searching for Computations in the Abstract or the Concrete (Invited Paper). InProceedingso/IFM’2005 (Fifth International Conference on Integrated Formal Methods), volume 3771 of Lecture Notes in Computer Science, pages 20-32, Eindhoven, November 2005. Springer-Verlag.
6. P. Godefroid, N. Klarlund, and K. Sen. DART: Di­rected Automated Random Testing. In Proceedings ofPLDI’2005 (ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation), pages 213-223, Chicago, June 2005.
7. B. S. Gulavani, T. A. Henzinger, Y. Kannan, A. ff Nori, and S. K. Rajamani. Synergy: A new algo­rithm for property checking. In Proceedings of the 14th Annual Symposium on Foundations of Software Engineering (FSE), 2006.
8. N. Gupta, A. P. Mathur, and M. L. Soffa. Generat­ing Test Data for Branch Coverage. In Proceedings of the 15th IEEE International Conference on Auto­mated Softffaffe Engineering, pages 219-227, Septem­ber 2000.
9. Y. Hamadi. Disolver : A Distributed Constraint

Solver. Technical Report MSR-TR-2003-91, Mi­

crosoft Research, December 2003.

1. M. Howard. Lessons learned from the an­imated cursor security bug, 2007. http: //blogs.msdn.com/sdl/archive/2007/ 04/26/lessons-learned-from- the-animated-cursor-security-bug. aspx.
2. J. C. King. Symbolic Execution and Program Testing. JournaloftheACM, 19(7):385-394, 1976.
3. B. Korel. A Dynamic Approach of Test Data Gener­ation. In IEEE Conference on Software Maintenance, pages 311-317, San Diego, November 1990.
4. R. Majumdar and K. Sen. Hybrid Concolic testing. In

Proceedings of ICSE}2007 (29th International Con­ference on Softffaffe Engineering), Minneapolis, May 2007. ACM.

1. D. Molnar and D. Wagner. Catchconv: Symbolic Execution and run-time type inference for integer conver­sion errors, 2007. UC Berkeley EECS, 2007-23.
2. Month of Browser Bugs, July 2006. Web page:

<http://browserfun.blogspot.com/>.

1. S. Narayanasamy, Z. Wang, J. Tigani, A. Edwards, and B. Calder. Automatically classifying benign and harmful data races using replay analysis. In Program­ming Languages Design and Implementation (PLDI), 2007.N.
2. Nethercote and J. Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In PLDI, 2007.J. Offutt and J. Hayes. A Semantic Model of Pro­gram Faults. In Proceedings of ISSTA’96 (Interna­tional Symposium on Software Testing and Analysis), pages 195-200, SanDiego, January 1996.
3. Pex. Web page:

<http://research.microsoft.com/Pex>.

1. Protos. Web page: <http://www.ee.oulu.fi/> research/ouspg/protos/.
2. K. Sen, D. Marinov, and G. Agha. CUTE: A Concolic Unit Testing Engine for C. In Proceed­ings of FSE'2005 (13th International Symposium on the Foundations of Software Engineering), Lisbon, September 2005.
3. A. Sotirov. Windows animated cursor stack

overflow vulnerability, 2007. http: //www.

determina.com/security.research/ vulnerabilities/ani-header.html.

1. Spike. Web page: <http://www.immunitysec>. com/resources-freesoftware.shtml.
2. M. Vuagnoux. Autodafe: An act of software torture.

In 22nd Chaos Communications Congress, Berlin, Germany, 2005. autodafe . sourceforge . net.

# 其他搜索统计

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Media 1: | wff-1 | wff-1nh | wff-2 | wff-2nh | | | | wff-3 | | wff-3nh | | | | wff-4 | | | | | | | wff-4nh | | | | | | | | |  |  |  |
| NULL | 1(46) | 1(32) | 1(23) | 1(12) | | | |  | 1(32) | 1(26) | | | | 1(13) | | | | | | |  | | | | | 1(1) | | |  |  |  |  |
| ReadAV | 1(40) | 1(16) | 2(32) | 2(13) | | | |  | 7(94) | 4(74) | | | | 6(15) | | | | | | |  | | | | | 5(45) | | |  |  |  |  |
| WriteAV | 0 | 0 | 0 | 0 | | | |  | 0 | 1(1) | | | | 1(3) | | | | | | |  | | | | | 1(1) | | |  |  |  |  |
| SearchTime | 10h7s | 10h11s | 10h4s | 10h20s | | | |  | 10h7s | 10h12s | | | | 10h34s | | | | | | | 9h29m2s | | | | | | | | |  |  |  |
| AnalysisTime(s) | 5625 | 4388 | 16565 | 11729 | | | |  | 5082 | 6794 | | | | 5545 | | | | | | |  | | | | | 7671 | | |  |  |  |  |
| AnalysisTasks | 564 | 545 | 519 | 982 | | | |  | 505 | 752 | | | | 674 | | | | | | |  | | | | | 878 | | |  |  |  |  |
| BlocksAtStart | 27659 | 26962 | 27635 | 26955 | | | |  | 27626 | 27588 | | | | 26812 | | | | | | | 26955 | | | | | | | |  |  |  |  |
| BlocksAdded | 701 | 50 | 865 | 111 | | | |  | 96 | 804 | | | | 910 | | | | | | |  | | | | | 96 | | |  |  |  |  |
| NumTests | 6890 | 7252 | 6091 | 14400 | | | |  | 6573 | 10669 | | | | 8668 | | | | | | | 15280 | | | | | | | |  |  |  |  |
| TestsToLastCrash | 6845 | 7242 | 5315 | 13616 | | | |  | 6571 | 10563 | | | | 6847 | | | | | | | 15279 | | | | | | | |  |  |  |  |
| TestsToLastUnique | 168 | 5860 | 266 | 13516 | | | |  | 5488 | 2850 | | | | 2759 | | | | | | |  | | | | | 1132 | | |  |  |  |  |
| MaxGen | 6 | 6 | 6 | 8 | | | |  | 6 | 7 | | | |  | | | 7 | | | |  | | | | | 8 | | |  |  |  |  |
| GenToLastUnique | 3(50%) | 5(83%) | 2(33%) | 7 (87.5%) | | | | 4(66%) | | 3 (43%) | | | | 4(57%) | | | | | | | 3 (37.5%) | | | | | | | | |  |  |  |
| Mean Changes | 1 | 1 | 1 | 1 | | | |  | 1 | 1 | | | |  | | | 1 | | | |  | | 1 | | | | | |  |  |  |  |
| Media 1: | wff-5 | wff-5nh | bogus-1 | bogus-1nh | | | | | bogus-2 | | | bogus-3 | | | | | bogus-4 | bogus-5 | | | | | | | | | | | | |  |  |
| NULL | 1(25) | 1(15) | 0 | 0 | | | |  | 0 | | | 0 | |  | | | 0 | | | |  | | 0 | | | | | |  |  |  |  |
| ReadAV | 3(44) | 3(56) | 3(3) | 1(1) | | | |  | 0 | | | 0 | |  | | | 0 | | | |  | | 0 | | | | | |  |  |  |  |
| WriteAV | 0 | 0 | 0 | 0 | | | |  | 0 | | | 0 | |  | | | 0 | | | |  | | 0 | | | | | |  |  |  |  |
| SearchTime | 10h8s | 10h4s | 10h8s | 10h14s | | | |  | 10h29s | | | 9h47m15s | | | | | 5m23s | 5m39s | | | | | | | | | | | |  |  |  |
| AnalysisTime(s) | 21614 | 22005 | 11640 | 13156 | | | |  | 3885 | | | 4480 | |  | | | 214 | | | | | | |  | 234 | | | | |  |  |  |
| AnalysisTasks | 515 | 394 | 1546 | 1852 | | | |  | 502 | | | 495 | |  | | | 35 | | | |  | | 35 | | | | | |  |  |  |  |
| BlocksAtStart | 27913 | 27680 | 27010 | 26965 | | | |  | 27021 | | | 27022 | | |  | | 24691 | | | | | | |  | 24692 | | | | |  |  |  |
| BlocksAdded | 109 | 113 | 130 | 60 | | | |  | 61 | | | 74 | |  | | | 57 | | | |  | | 41 | | | | | |  |  |  |  |
| NumTests | 4186 | 2994 | 12190 | 15594 | | | |  | 13945 | | | 13180 | | |  | | 35 | | | |  | | 35 | | | | | |  |  |  |  |
| TestsToLastCrash | 4175 | 2942 | 1403 | 11474 | | | |  | NA | | | NA | |  | | | NA | | | |  | | NA | | | | | | |  |  |  |
| TestsToLastUnique | 1504 | 704 | 1403 | 11474 | | | |  | NA | | | NA | |  | | | NA | | | |  | | NA | | | | | | |  |  |  |
| MaxGen | 5 | 4 | 14 | 13 | | | |  | 8 | | | 9 | |  | | | 9 | | | |  | | 9 | | | | | |  |  |  |  |
| GenToLastUnique | 3 (60%) | 3(75%) | 10(71%) | 11(84%) | | | |  | NA | | | NA | |  | | | NA | | | |  | | NA | | | | | | |  |  |  |
| Mean Changes | 1 | 1 | 1 | 1 | | | |  | 1 | | | 1 | |  | | | 1 | | | |  | | 1 | | | |  | | |  |  |  |
| Media 2: | wff-1 | wff-1nh | wff-2 | wff-3 | wff-3nh | | | | wff-4 | | wff-4nh | | wff-5 | | | | | wff-5nh | | | | | | | | | | bogus1 | | | |
| NULL | 0 | 0 | 0 | 0 |  | | 0 | | 0 | |  | 0 |  | | | 0 | |  |  | | | 0 | | | | | | 0 | | | |
| ReadAV | 4(9) | 4(9) | 0 | 4(15) |  | | 4(14) | | 4(6) | |  | 3(3) | 5(14) | | | | |  | 4(12) | | | | | | | | | 0 | | | |
| WriteAV | 0 | 0 | 0 | 0 |  | | 0 | | 0 | |  | 0 |  | | | 1(1) | |  |  | | | 0 | | | | | | 0 | | | |
| SearchTime | 10h12s | 10h5s | 10h6s | 10h17s |  | | 10h1s | | 10h3s | |  | 10h7s | 10h3s | | | | |  | 10h6s | | | | | | | | | 10h13s | | | |
| AnalysisTime(s) | 3457 | 3564 | 1517 | 9182 |  | | 8513 | | 1510 | |  | 2195 | 10522 | | | | |  | 14386 | | | | | | | | | 14454 | | | |
| AnalysisTasks | 3 | 3 | 1 | 6 |  | | 7 | | 2 | |  | 2 |  | | | 6 | |  |  | | | 6 | | | | | | 1352 | | | |
| BlocksAtStart | 48494 | 48486 | 51217 | 41726 |  | | 41746 | | 48729 | | 48778 | | 41917 | | | | |  | 42041 | | | | | | | | | 20008 | | | |
| BlocksAdded | 10407 | 10633 | 12329 | 9564 |  | | 8643 | | 10379 | |  | 10022 | 8980 | | | | |  | 8746 | | | | | | | | | 14743 | | | |
| NumTests | 1045 | 1014 | 777 | 1253 |  | | 1343 | | 1174 | |  | 948 | 1360 | | | | |  | 980 | | | | | | | | | 4165 | | | |
| TestsToLastCrash | 1042 | 989 | NA | 1143 |  | | 1231 | | 1148 | |  | 576 | 1202 | | | | |  | 877 | | | | | | | | | NA  NA | | | |
| TestsToLastUnique | 461 | 402 | NA | 625 |  | | 969 | | 658 | |  | 576 |  | | | 619 | |  | 877 | | | | | | | | |
| MaxGen | 2 | 2 | 1 | 3 |  | | 2 | | 2 | |  | 2 |  | | | 3 | |  |  | 2 | | | | | | | | 14 | | | |
| GenToLastUnique | 2(100%) | 2(100%) | NA | 2 (66%) | 2(100%) | | | | 2(100%) | | 1 (50%) | |  | | | 2 | |  |  | 2 | | | | | | | | NA | | | |
| Mean Changes | 3 | 3 | 4 | 4 |  | 3.5 | | | 5 | |  | 5.5 |  | | | 4 | |  |  | 4 | | | | | | | | 2.9 | | | |

图 10搜索统计。 对于每个搜索，我们报告每种类型的崩溃次数：第一个数字是不同桶的数量，而括号中的数字是总数的崩溃测试用例。 我们还报告搜索总时间（SearchTime），总时间符号执行（AnalysisTime），符号执行任务（AnalysisTasks）的数量，块由初始文件（BlocksAtStart）覆盖，搜索期间发现的新块（BlocksAdded），测试总数（NumTests），最后一次崩溃的测试（TestsToLastCrash），最后一个独特的桶的测试（TestsToLastUnique），最大的一代达到（MaxGen），发现最后一个独特的桶的一代（GenToLastUnique），并且每个生成的测试用例（Mean Changes）的文件位置的平均数量发生了变化。