FuzzGuard: Filtering out Unreachable Inputs in Directed Grey-box Fuzzing through Deep Learning 阅读 & 笔记

同类的Fuzz工具有：aflfast和ecofuzz

Abstract

最近，定向灰盒模糊测试（directed grey-box fuzzing — DGF）在软件测试领域变得很流行。 与基于覆盖的模糊测试（其目标是增加代码覆盖范围以触发更多错误）不同，DGF旨在检查一段潜在的漏洞代码（例如，字符串操作）是否确实包含漏洞。 理想情况下，DGF生成的所有输入都应到达目标漏洞代码，直到触发漏洞为止，使用无法访问的输入去执行是浪费时间的。 不幸的是，在实际情况下，大量生成的输入无法使程序执行到目标，这极大地影响了模糊测试的效率，尤其是当越存在漏洞的代码代码嵌入受各种约束保护的代码中时。

在本文中，我们提出了一种基于深度学习的方法，可以在执行目标程序之前预测输入的可达性（即是否错过目标），帮助DGF过滤掉不可达的输入以提高模糊测试的性能。 为了运用DGF进行深度学习，我们设计了一套新技术（例如，step-forwarding方法，代表性数据选择）来解决标签数据不平衡和训练过程中时间不足的问题。 此外，我们实施了名为FuzzGuard的建议方法，并为其配备了最新的DGF（例如AFLGo）。 对45个实际漏洞的评估表明，FuzzGuard将vanilla AFLGo的模糊处理效率提高了17.1倍。 最后，为了理解FuzzGuard所学的关键特征，我们说明了它们与程序中的约束的关系。

1 Introduction

模糊测试是一种自动程序测试技术，通常分为两类：**基于覆盖率的模糊测试和定向模糊测试。** 前者的目标是实现较高的代码覆盖率，希望触发更多的崩溃。 **定向模糊测试旨在检查给定的潜在漏洞代码是否确实包含漏洞。** 在实际分析中，由于经常指定漏洞代码，因此非常常用定向模糊测试。 例如，安全分析人员通常更关注缓冲区操作代码，或者想为已知漏洞代码的给定CVE[3]生成概念验证（PoC）漏洞。 有一些定向的模糊测试工具，例如AFLGo [9]，SemFuzz [35]和Hawkeye [12]。 众所周知，**随机输入不太可能到达漏洞代码，更不用说触发漏洞了。** 因此，大多数工具会为目标程序提供工具，以观察运行时信息并利用该信息生成可能到达漏洞代码的输入。 这种模糊测试方法也称为定向灰箱模糊测试（DGF）。

理想的DGF应该生成所有可以到达漏洞代码的输入。 不幸的是，在实际情况下，大量生成的输入可能会错过目标，特别是当将漏洞代码嵌入受许多（复杂）约束（例如数千个）保护的代码中时。 面对这种情况，设计了各种技术（例如，基于退火的功率表[9]）来生成可到达的输入。 但是，即使对于最先进的DGF工具（例如AFLGo [9]），无法访问的输入比例仍然很高。 **根据我们使用AFLGo进行的评估，平均而言，超过91.7%的输入无法到达错误代码**（第6节）。

如此大量的无法到达的输入会在模糊测试过程中浪费大量时间。 理论上，**诸如符号执行[20]之类的传统程序分析方法可以使用目标程序中所有分支的约束来推断输入的执行结果。** 但是，解决约束所

花费的时间将随着约束复杂性的增加而急剧增加。 换句话说，**从程序的起点到漏洞代码的路径中的约束可能非常复杂，这使它们很难甚至不可能在有限的时间内解决。**

模式识别的成功启发[11,19,34,36]可以准确地对数百万张图像进行分类，即使以前看不见它们，我们的想法是将程序输入视为一种模式，并识别可以到达漏洞的图像码。 基本上，通过使用大量标有先前执行中的目标代码可达性的输入来训练模型，我们可以利用该模型预测新生成的输入的可达性，而无需运行目标程序。但是，由于以下原因，为DGF建立如此精确的模型具有挑战性：

Challenges。 C1：缺少平衡的标记数据。 必须获取足够且平衡的标签数据以训练深度学习模型（例如，猫和狗的分类）。 换句话说，一个物体的图像数量应与另一物体的数量接近。 但是，**在模糊测试过程中，（无法）到达的输入通常是不平衡的。 特别是在模糊测试的早期，甚至没有可达的输入**（例如，对于GraphicsMagick中的bug#7，**第一个可达的输入是在执行了超过2260万次执行之后生成的**）。 **没有平衡的标签数据，训练后的模型将容易过拟合。** 人们可能会想像图像转换的方式（例如，调整大小，变形，透视变换）那样扩展标记的数据，这可能会增加对象的图像数量以平衡训练数据而不会更改所标识的对象。 但是，**这种转换不能应用于程序输入，因为即使一位翻转也可能改变输入的执行路径并进一步影响标签（即，使可到达的输入变得不可到达）。**

C2：新生成的可及输入看起来可能与训练集中的可及输入完全不同，从而使训练后的模型无法预测新输入的可及性。 这主要是**因为新输入可能会通过以前从未见过的不同执行路径到达错误代码。** 因此，**仅沿一条执行路径使用输入来训练模型可能无法正确预测新输入的可达性。** 人们可能会想到在训练之前，沿着漏洞代码的不同执行路径生成各种输入。 不幸的是，这样的生成过程是我们无法控制的。 他可能还需要等待很长时间才能接受训练，希望能沿不同途径收集足够的投入。 但是，这可能会浪费大量时间，因为已经执行了许多无法到达的输入。

C3：**效率。** 在训练传统模式识别模型的任务中，在训练上花费的时间没有严格限制。 但是，在模糊测试过程中，如果花费在训练模型和预测输入的可及性上的时间大于花费在执行具有输入的程序上的时间，那么进行预测是没有用的。 因此，应该严格限制训练和预测的时间成本。

Our approach。 在本文中，我们克服了上述挑战，**并设计了一种方法来构建DGF模型以过滤出不可达的输入，称为FuzzGuard。** FuzzGuard的基本思想**是通过从以前的执行中学习来预测程序是否可以使用新生成的输入执行到目标漏洞代码。 如果无法获得预测结果，则定向灰箱模糊器（在本文的其余部分中简称为“模糊器”）将不再执行此输入，从而节省了实际执行所花费的时间。** 请注意，**FuzzGuard并不是要替换模糊器（例如AFLGo），而是要与模糊器一起使用以帮助其过滤出无法到达的输入。**

FuzzGuard分三个阶段工作：**模型初始化，模型预测和模型更新。**

（1）在第一阶段，模糊器生成各种输入并与它们一起运行目标程序，以检查是否触发了漏洞。 同时，FuzzGuard保存输入及其可及性，并使用标记的数据训练初始模型，这些数据可能是不平衡的（C1）。 为了解决这个问题，我们设计了一种逐步推进的方法：选择漏洞代码的主导者（称为“主导节点” [5]）作为中间阶段目标，并让执行达到主导地位 节点优先。 这样，就可以更早地获取平衡数据，以训练仅针对主导节点的某些模型，从而最大程度地缩短了执行时间。

（2）在第二阶段，在Fuzzer生成大量新输入之后，FuzzGuard使用该模型预测每个输入的可达性。 如C2中所述，训练后的模型可能不适用于新生成的输入。 为了解决这个问题，我们设计了一种有代表性的数据选择方法，用于从每一轮突变中采样训练数据，从而最大程度地减少了采样数据的数量，从而提高了效率。

（3）在第三阶段，FuzzGuard使用第二阶段收集的标记数据更新模型，以提高其准确性。 请注意，应严格限制在模型更新上花费的时间（C3）。 我们通过谨慎选择更新时间来应对这一挑战。

据我们所知，以前的模糊研究主要集中在生成各种输入，覆盖更多的代码行（CGF）或获取漏洞的代码（DGF）。 **设计了输入上的各种变异策略。 相比之下，我们的研究并未直接突变输入（我们依赖于当前的突变策略，例如AFLGo）**。 相反，我们过滤掉无法到达的输入。 这样，DGF不需要使用无法访问的输入（肯定不能触发目标错误）运行目标程序，从而提高了整体效率。

我们在AFLGo [9]（开源的最新DGF工具）的基础上实现FuzzGuard，并使用10个流行程序上的45个实际漏洞对性能进行评估。 结果表明，FuzzGuard将模糊性能提高了1.3倍至17.1倍。 **有趣的是，我们发现模糊器生成的输入越多，FuzzGuard的性能就越好。** 另外，如果目标节点更早达到平衡状态，则可以节省更多时间。 最后，**我们设计了一种方法来理解FuzzGuard提取的功能，并发现这些函数与目标程序的if语句中的约束相关联，这确实影响了代码级的执行。**

Contribution。 本文的贡献如下：

- **新技术。** 我们设计并实现了FuzzGuard，它有助于DGF过滤掉不可达的输入并节省不必要的执行时间。 据我们所知，这是第一个基于深度学习的解决方案，用于识别和删除无法访问的输入。 **FuzzGuard的核心是step-forwarding方法和代表性数据选择。** 评估结果表明，使用最先进的工具（例如AFLGO）可以节省多达88％的模糊时间。我们还发布了FuzzGuard，以帮助社区中的研究人员。
- **新的理解。 我们设计了一种方法来研究FuzzGuard中模型用于预测的功能，并发现它们与目标程序中的分支相关。** 对这种关系的理解有助于解释深度学习模型，并进一步有助于改进FuzzGuard。

2 Background

在本节中，我们简要介绍了有针对性的灰箱模糊测试的背景以及最近使用深度学习提高模糊测试性能的研究。

2.1 Fuzzing

模糊测试[27]是显示计算机程序异常的经典软件测试技术之一[32]。 **模糊测试的主要思想是向目标程序提供大量输入（即测试用例），通过观察到的异常暴露漏洞。** 在所有的模糊测试技术中，灰盒模糊测试[12]最近因其高效和合理的性能开销而变得非常流行。 由于目标不同，灰盒模糊测试通常可以分为以下两种类型：

基于覆盖的灰盒模糊测试。 这种模糊测试技术的一个主要目标是在目标程序中实现较高的代码覆盖率。 因此，一些模糊器[2, 10, 15, 16, 24, 25]旨在实现目标程序的高代码覆盖率，期望偶然触发该漏洞，即基于覆盖的灰箱模糊化（CGF）。 通常，**CGF通过使可以遍历以前未发现的程序语句的种子输入发生变异来生成输入，以提高代码的覆盖率。** 作为CGF的代表，AFL [2]使用轻量级的编译时插桩技术和遗传算法来自动发现有趣的测试用例，选择在模糊测试过程中触发新的内部状态的种子输入，并对种子进行变异以各种方式输入（例如，位和字节翻转，简单算术，堆叠的调整和拼接[22]）。

定向灰箱模糊测试。 有时，潜在的漏洞代码是已知的，因此，无需增加代码覆盖率。 在这种情况下，模糊器[9，12，35]被设计为生成输入，该输入到达用于触发特定漏洞的漏洞代码，称为定向灰箱模糊（DGF）。 **通常使用DGF，因为某些类型的代码很可能包含漏洞（例如字符串复制操作），应该在模糊测试中进一步强调。** 此外，有时会发现漏洞代码（例如，来自CVE）。 因此，这些模糊器被用来针对漏洞代码生成概念验证漏洞[35]。 **与CGF的目标不同，当前的DGF旨在生成可能达到**

特定潜在bug代码的输入，进一步期望触发该bug。 例如，**AFLGo [9]计算每个基本块之间的距离以及控制流图中从入口点到漏洞代码的路径。 然后利用距离选择适合突变的输入。**

但是，**即使对于最先进的模糊器AFLGO，仍然会花费大量时间进行不必要的执行。** 在我们的实验中，我们发现，对于一个已知位置的典型漏洞，平均超过91.7％的生成输入无法到达漏洞代码（无法到达的输入）。 使用无法访问的输入来运行目标程序非常耗时。 **如果有一个模糊器可以在不执行程序的情况下判断输入的可达性，则可以节省大量时间。** 在本文中，我们设计了一个名为FuzzGuard的过滤器，**该过滤器利用深度学习模型来实现此目标而无需实际执行。 而且，它可以适合现有的模糊器（例如AFLGo）并与它们一起工作，而无需替换它们。** 据我们所知，这是第一个基于深度学习的解决方案，用于为DGF筛选出无法到达的输入。

2.2 Deep Learning

安全研究人员将深度学习应用于模糊测试，这为解决先前研究中的难题提供了新见解。 例如，Godefroid等 **利用RNN生成具有更高代码覆盖率的程序输入**[17]。 [29]**利用RNN引导的突变过滤器来定位输入的哪一部分对代码覆盖率的影响更大。** 这样，他们可以通过对定位部分进行变异来实现更高的代码覆盖率。 Nichols等[28]表明GAN可以用来预测输入的执行路径，以提高AFL的性能[2]。 Angora[15]和NEUZZ [31]分别采用梯度下降算法来解决路径约束并学习模型以提高代码覆盖率。 所有这些研究都集中在利用深度学习来覆盖更多代码的能力上。 与它们不同的是，**我们的目标是帮助定向灰箱模糊器在实际执行之前过滤掉无法触及漏洞代码的输入。** 这样，可以节省在输入无法到达的情况下运行程序所花费的时间，从而大大提高了模糊测试的效率。 请注意，我们的工具可以适应现有的DGFtools（例如AFLGo），这意味着我们可以进一步提高模糊测试效率以及为其他模糊测试技术提高的性能。

3 Motivation

Table 1: Effectiveness of FuzzGuard.

No.	Program	Vuln. Code	$N_{Functions}$	$N_{Constraints}$	N_{Inputs}	$UR.$	$Filtered$	T_{AFLGo}	T_{+FG}	Speedup		
										FG	FG_1	FG_2
1	Bento4 v1.5.1.0	Ap4AvccAtom.cpp:83	676	1.8 K	1.8 M	38.1%	32.3%	44 h	29.9 h	**1.5**	1.3	1.4
2	Ettercap v0.8.2	ec_strings.c:182	420	41.5 K	49.2 M	99.0%	93.9%	80.5 h	6.1 h	**13.3**	1.1	8.3
3	GraphicsMagick v1.3.31	tiff.c:2375	3.3 K	170.3 K	32.1 M	95.9%	90.5%	94.8 h	11.1 h	**8.6**	5.9	7.5
4	GraphicsMagick v1.3.31	png.c:6945	4.9 K	319.8 K	30 M	96.6%	88.8%	200 h	23.4 h	**8.5**	2.0	8.2
5	GraphicsMagick v1.3.31	png.c:7503	1.5 K	21.9 K	16.4 M	99.9%	84.1%	13.2 h	3.9 h	**3.4**	1.0	3.1
6	GraphicsMagick v1.3.31	png.c:5007	4.4 K	317.1 K	16 M	99.9%	34.3%	200 h	132.3 h	**1.5**	1.0	1.5
7	GraphicsMagick v1.3.30	png.c:3810	3.1 K	168.4 K	22.6 M	99.9%	32.8%	31.9 h	22.4 h	**1.4**	1.0	1.4
8	GraphicsMagick v1.3.27	webp.c:716	10.7 K	749.3 K	67.5 M	99.5%	93.4%	200 h	15.2 h	**13.2**	9.7	9.7
9	GraphicsMagick v1.3.26	png.c:7061	4.9 K	320 K	56.9 M	98.4%	93.3%	200 h	16.2 h	**12.3**	8.4	9.4
10	GraphicsMagick v1.3.26	tiff.c:2433	4.4 K	316.4 K	78.4 M	75.3%	69.5%	200 h	66.7 h	**3.0**	2.4	2.7
11	GraphicsMagick v1.3.26	rle.c:753	9.2 K	379.3 K	17.7 M	99.4%	70.5%	30.8 h	10.4 h	**3.0**	1.5	2.6
12	GraphicsMagick v1.3.26	list.c:232	3.6 K	172.1 K	73 M	37.2%	28.4%	200 h	146.4 h	**1.4**	1.6	1.4
13	ImageMagick v7.0.8-13	msl.c:8353	85.5 K	5.4 M	7.3 M	99.2%	92.4%	200 h	15.4 h	**13.0**	3.3	12.9
14	ImageMagick v7.0.8-3	dib.c:1306	117.1 K	7, 883.1 M	3.2 M	99.9%	54.4%	200 h	91.4 h	**2.2**	1.0	1.6
15	ImageMagick v7.0.8-3	bmp.c:2062	117.3 K	6, 306.9 M	3 M	99.9%	52.3%	200 h	95.6 h	**2.1**	1.0	2.0
16	ImageMagick v7.0.7-16	webp.c:769	23.9 K	145.7 K	11.5 M	99.1%	93.9%	200 h	12.6 h	**15.9**	15.5	14.7
17	ImageMagick v7.0.7-16	webp.c:403	14.8 K	116.1 K	19.1 M	96.0%	90.7%	200 h	19.8 h	**10.1**	9.4	10.7
18	ImageMagick v7.0.7-1	tiff.c:1934	149.1 K	1.2 M	9.4 M	98.5%	92.5%	200 h	15.2 h	**13.1**	1.6	8.7
19	ImageMagick v7.0.5-5	bmp.c:894	102.4 K	926.5 K	12.9 M	64.3%	59.9%	200 h	80.5 h	**2.5**	1.7	2.5
20	Jasper v2.0.14	jp2_enc.c:309	13.9 K	17.7 M	28 M	99.4%	50.9%	200 h	99 h	**2.0**	1.7	1.9
21	Jasper v2.0.10	jpc_dec.c:1700	740	9.7 K	11.3 M	99.7%	94.3%	46.9 h	3.7 h	**12.7**	1.4	11.1
22	Jasper v2.0.10	jpc_dec.c:1881	1.7 K	36.8 K	6.1 M	99.9%	94.0%	19.7 h	1.6 h	**12.0**	1.0	10.0
23	Jasper v2.0.10	jas_seq.c:254	1.1 K	11.8 K	22.3 M	62.4%	56.0%	200 h	89 h	**2.2**	2.0	2.2
24	Libming v0.4.8	decompile.c:1930	104	5.3 K	38.6 M	99.9%	70.2%	200 h	63 h	**3.2**	1.0	3.1
25	Libming v0.4.7	parser.c:1645	75	4.7 K	32.3 M	99.8%	94.7%	200 h	11.7 h	**17.1**	8.5	14.1
26	Libming v0.4.7	parser.c:64	170	2.7 K	16.1 M	91.9%	86.6%	200 h	27.2 h	**7.3**	1.7	6.2
27	Libming v0.4.7	parser.c:3381	79	790	38.4 M	99.7%	69.9%	200 h	61.3 h	**3.3**	2.0	3.2
28	Libming v0.4.7	parser.c:3095	25	217	46.8 M	92.9%	65.7%	200 h	70 h	**2.9**	1.9	2.8
29	Libming v0.4.7	parser.c:2993	22	386	45.9 M	97.2%	64.8%	200 h	71.8 h	**2.8**	1.7	2.7
30	Libming v0.4.7	parser.c:3126	24	294	77 M	92.9%	63.6%	200 h	75.3 h	**2.7**	2.0	2.5
31	Libming v0.4.7	parser.c:3232	55	423	12.6 M	99.8%	61.3%	6.1 h	2.8 h	**2.2**	2.0	1.8
32	Libming v0.4.7	parser.c:3221	38	308	13 M	99.9%	43.2%	14 h	8.2 h	**1.7**	1.0	1.6
33	Libming v0.4.7	parser.c:3250	32	340	16.6 M	99.9%	46.0%	7.3 h	4.4 h	**1.7**	1.0	1.4
34	Libming v0.4.7	parser.c:3089	36	396	19.6 M	99.9%	43.3%	5.2 h	3.4 h	**1.5**	1.0	1.1
35	Libming v0.4.7	parser.c:3061	37	637	18.9 M	99.8%	37.2%	3.4 h	2.5 h	**1.4**	1.0	1.1
36	Libming v0.4.7	parser.c:3071	34	1.1 K	17.6 M	99.9%	33.6%	3.8 h	2.9 h	**1.3**	1.0	1.1
37	Libming v0.4.7	parser.c:3209	34	402	30.7 M	99.9%	27.7%	8.9 h	6.9 h	**1.3**	1.0	1.2
38	Libming v0.4.7	outputtxt.c:143	64	2.2 K	27.3 M	65.5%	24.6%	7.7 h	6.1 h	**1.3**	1.1	1.1
39	Libtiff v4.0.9	tif_dirwrite.c:1901	728	14.4 K	8.6 M	99.9%	91.4%	9.6 h	1.3 h	**7.4**	1.0	4.8
40	Libtiff v4.0.7	tif_swab.c:289	631	13.1 K	44.7 M	99.7%	52.8%	29.6 h	15 h	**2.0**	1.1	1.3
41	Libtiff v4.0.7	tiffcp.c:1386	728	13.3 K	15.6 M	99.9%	51.7%	8.9 h	4.6 h	**1.9**	1.0	1.7
42	Libtiff v4.0.7	tif_read.c:346	416	11.6 K	60.6 M	79.5%	36.3%	77.9 h	49.8 h	**1.6**	1.4	1.5
43	Libxml2 v2.9.4	SAX2.c:2035	418	15.7 K	92.6 M	99.9%	94.4%	200 h	17.6 h	**11.3**	1.0	5.2
44	Podofo v0.9.5	PdfPainter.cpp:1945	19.8 K	44.1 K	2.6 M	99.3%	79.7%	200 h	40.7 h	**4.9**	4.8	1.8
45	Tcpreplay v4.3.0-beta1	get.c:174	23	1.1 K	203.3 M	53.2%	49.5%	200 h	105.4 h	**1.9**	1.7	1.9
Avg.			15.5K	315.9 M		91.7%	65.1%			**5.4**	2.6	4.4

如上所述，**当前的DGF旨在生成可能到达特定漏洞代码的输入，并进一步期望触发漏洞。** 在模糊测试过程中，很多输入最终都无法到达漏洞代码（不可能触发漏洞）。 根据我们的评估，平均超过91.7%的输入未达到漏洞代码（请参见表1）。 执行数百万个无法到达的输入可能会花费很长的时间（例如，对Podofo进行模糊处理时，一百万个输入需要花费76个小时，Podofo是一个使用几种工具处理PDF文件格式的库[1]）。特别地，当目标程序的执行时间在整个模糊过程中占最大比例时，浪费的时间甚至更多。 **如果存在一种足够快的方法来预测输入的可到达性，则模糊测试不需要使用不可达的程序来执行目标程序。** 这样，可以提高模糊测试的整体性能。

受到深度学习在模式识别方面的最新成功的启发[11，19，34，36]，我们想知道深度学习是否可以应用于识别（不）可达输入。仔细比较模式识别和（不）可达输入的识别过程。，我们发现它们之间的相似之处：**它们都基于从模式中提取的先验知识或统计信息（对象的许多带标签的图像与以前执行的许多带标签的输入）对数据（某些对象与（不）可达输入进行分类）。** 但是，它们确实存在本质上的差异（例如，标记数据的分布，效率要求等），这使得无法到达的输入识别过程非常具有挑战性（请参见第1节）。

```
1  ThrowReaderException(...);
2 if (dib_info.colors_important > 256)
3  ThrowReaderException(...);
4 if ((dib_info.image_size != 0U) && (dib_info.image_size
      > file_size))
5  ThrowReaderException(...);
6 if ((dib_info.number_colors != 0) ||
      (dib_info.bits_per_pixel < 16)) {
7    image->storage_class=PseudoClass;
```

Listing 1: The vulnerable code of CVE-2018-20189.

Example： List 1给出了一个示例。 易受攻击的代码位于第6行（请参见第7节）。 因此DGF（例如AFLGo）的目标是生成尽可能多的输入并希望触发该错误。 种子输入是从AFLGo的种子语料库（例如，not_kitty.png）中选择的。 **生成1600万个输入需要13个小时，并且需要在触发bug之前对其进行测试。** 在这些输入中，只有3.500（0.02%）可以到达漏洞代码。 可能有人想到利用符号执行来生成从执行路径到目标的约束。 但是，由于多个路径可能会到达漏洞代码，因此很难生成完全约束。 即使可以生成约束，使用约束计算可达性仍然非常耗时，这甚至与运行目标程序所花费的时间相似。 **我们的想法是生成一个深度学习模型，以自动提取可访问输入的特征并识别将来可访问的输入。** 根据我们的评估，确定了将近1400万个输入（占84.1%），从而节省了9个小时的不必要执行。 还要注意，此示例的误报率和误报率分别仅为2.2%和0.3%。

范围和假设。 与以前使用深度学习对CGF进行的研究[15、17、28、29]不同，我们的方法侧重于筛选DGF中无法达到的输入。 这样，可以节省执行输入不可达的程序所需的大量时间。 请注意，我们的方法是对其他DGF工具的补充，可以与它们一起使用，而不是替换它们。 还要注意，**我们不假定输入中的小突变会产生相似或相同的行为。 经过训练的模型应表征相似输入的不同行为。**

4 Methodology

我们建议设计FuzzGuard，这是一种基于深度学习的方法，可帮助DGF筛选出无法访问的输入，而无需真正使用它们执行目标程序。 这样的数据驱动方法避免了使用传统的耗时方法（例如符号执行）来获得更好的性能。 下面我们详细介绍FuzzGuard。

4.1 Overview

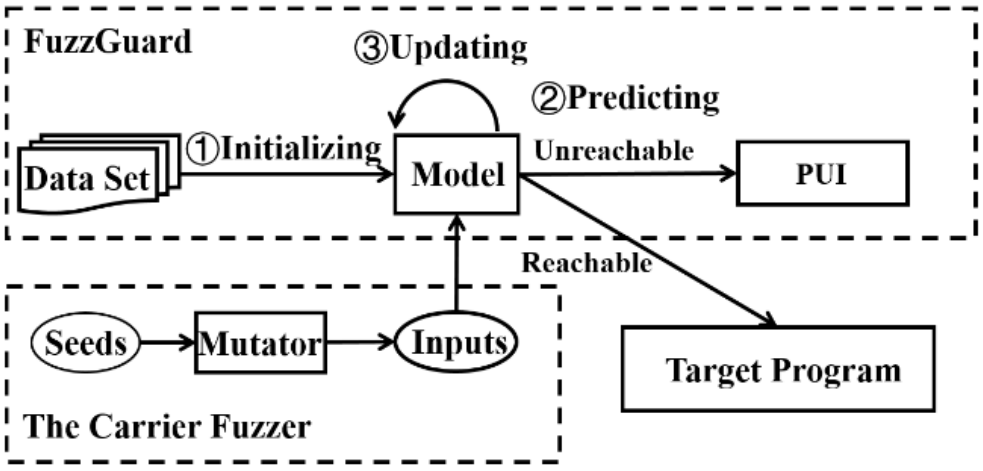

Figure 1: An overview of FuzzGuard.

FuzzGuard的概述如图1所示，其中包括三个主要短语：**模型初始化（MI），模型预测（MP）和模型更新（MU）**。 它与DGF（称为“载波模糊器(**the carrier fuzzer**)”）一起使用。 **如图1所示，在MI模型初始化中，the carrier fuzzer*生成大量输入并尝试观察到任何异常。** FuzzGuard记录程序是否可以为每个输入执行目标漏洞代码。 然后FuzzGuard使用输入及其可达性来训练模型。 **从MP模型预测来说，FuzzGuard对该模型进行了实用化，以预测新生成的输入的可达性。** 如果输入可达，则将其输入到程序中以进行实际执行。 在此过程中，FuzzGuard会观察输入是否可以真正达到目标代码。 **从MU模型更新来说，FuzzGuard会通过增量学习来进一步更新模型，以维持其效率并提高其性能。**无法访问的输入将临时保存在数据池中（称为“无法访问的输入池（PUI）”），用于模型更新后，通过更准确的模型进行进一步检查。 由于将深度学习与模糊测试结合起来并非易事，因此我们面临着新的挑战（如第1节所述）。

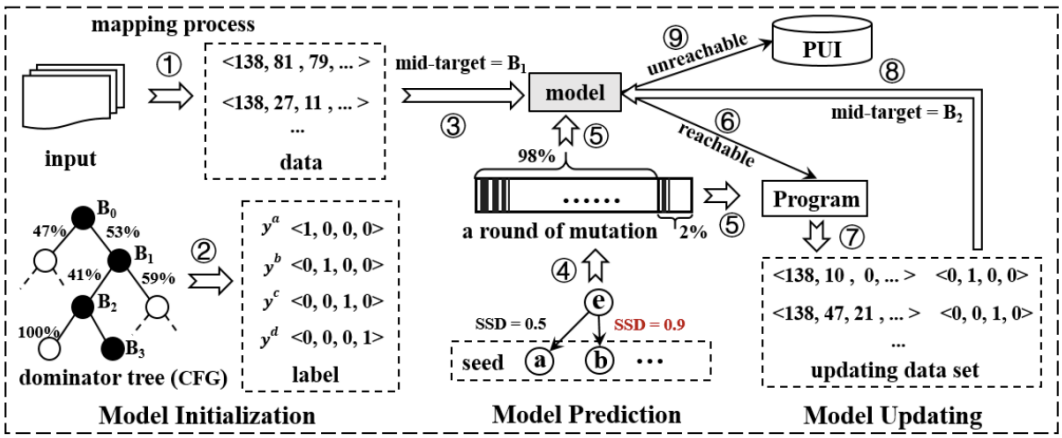

Figure 2: An example of filtering unreachable inputs by FuzzGuard.

图2显示了FuzzGuard的具体示例。 首先，**the carrier fuzzer**生成许多输入（称为“data”），并使用它们运行目标程序以获取MI模型初始化中的可达性（称为“label”）（图中的步骤1和步骤2）。 **在此过程中，理想的情况是使用平衡数据来训练深度学习模型。** 也就是说，**大约一半的输入可以到达漏洞代码，而另一半则不能。** 不幸的是，实际上，**the carrier fuzzer**几乎不会在模糊化的初始阶段生成到达漏洞代码的输入。 因此，标记的数据通常在此阶段非常不平衡。 例如，在生成了超过2200万个输入之后，只有一个输入实际上可以到达漏洞代码（表1中的第7个）。 为了解决这个问

题，我们设计了针对MI的step-forwarding方法（step2©），该方法使输入到达漏洞代码（eg B_3 在图二）的主要控制节点（在本文的其余部分中，我们使用“节点”来指代“基本块”），逐步到达目的地（eg B_3 在图二）。特别是，FuzzGuard选择一个优先支配节点（例如，B_1）作为中间阶段的目标（即称为“中间目标”），并生成一个模型以过滤出无法到达B_1的输入（步骤3©）。 通常，与B_3相比，当程序运行到B_1时可以获得更平衡的标记数据。 因此，可以更早地训练模型，并且也可以更早开始工作。 然后，MP模型预测模块会使用模型（步骤5©）判断新生成的输入（在步骤4©中）是否可以到达B1。对于可达的输入（例如，在步骤6©中使用label <0,1,0,0>进行标记），FuzzGuard使用该程序运行程序，并记录它是否真的可以到达漏洞代码（步骤7©）。 此类信息被MU模型更新步骤进一步利用以不断更新模型（步骤8©）。 将无法访问的输入放入PUI(无法访问的输入池)（步骤9©）。在测试了更多输入之后，将出现一个具有平衡标记数据的更接近于主导的节点（对于漏洞代码）（例如，在这种情况下为B_2）。 这样的过程将一直持续到漏洞代码到达并最终触发为止。 下面我们提供三个模块的详细信息。

4.2 Model Initialization

如前所述，**将深度学习应用于模糊测试的一个主要挑战是训练数据的不平衡。 通常，可达到的输入数量远远少于不可达的输入数量。** 为了应对这一挑战，我们提出了一种step-forwarding方法。 基本思想是基于以下观察：较早到达漏洞代码的主要节点，这应该较早获得平衡数据。 请注意，漏洞代码的主导节点是主导漏洞代码的节点：通往漏洞代码的每个执行路径都将通过主导地位的节点[5]。 因此，可以保证标记的主导节点的可达性。 因此，**我们可以训练一个模型来筛选出那些无法到达主要控制节点的输入（它们也无法到达漏洞代码）**。 通过这种方式，我们逐渐获得了主导节点的平衡数据，最终到达了漏洞代码。 例如，对于图2中所示的控制流图，节点表示列表1中程序中的基本块。B_0是入口点，漏洞代码在B_3中。 B_1和B_2是B_3的主导节点。 在开始模糊测试时，没有输入到达B_3，而一半的输入可以到达B1。 现在，B_1是最接近漏洞代码的平衡主导节点。 因此，我们将B_1视为目标，并使用这些输入来训练模型。 这样，可以过滤掉B_1的不可达输入，从而节省了用它们执行目标程序所花费的时间。 当模糊测试过程进一步进行时，B_2或B_3将获得平衡数据进行训练。 请注意，与旨在实现高覆盖率的CGF不同，DGF的目的是生成输入以在特定位置触发给定（潜在）漏洞。 因此，它并不关心是否在其他路径中发现了新的错误。 有趣的是，我们确实看到FuzzGuard + AFLGo仍然发现了尚未发现的bug（请参阅第6节），这些bug深入位于程序中（也靠近目标bug代码）。 普通的CGF很难在有限的时间内触发它们。

> 有一些疑问 ：当通过选择性取输入数据后，得到平衡数据进行训练后，会不会把输入进行变异从而让输入能保持经过关键节点并努力靠近下一个关键节点？还是程序只是会不断地去跑由AFLGO生成的1600万个种子，从而进行挑选和标记以及训练模型呢？
>
> FuzzGuard不会，因为FuzzGuard只是筛选AFLGo生成的种子，本身不生成新的种子，所以也不存在对种子再次进行变异“靠近下一个关键节点”的操作；因此也不存在在step-forwarding过程中进行新的变异的操作。但是现存的有基于语法的自动生成的（用语法来保证变异出来的语句有效），比如GRIMOIRE，USENIX19
>
> Thanks to kdsj.

但是，为每个主导节点训练一个模型花费的时间太长。 这主要是因为当FuzzGuard前进到下一个主要节点时，需要重新训练模型。 **我们的想法是只为所有主导节点（包括漏洞代码本身）训练一个模型。**为实现这一目标，我们在向量上标记了节点的可达性（即$B = \{B_1, B_2, \ldots, B_n\}$）。 对于每个标签，它表示为单位向量ŷ，即$\hat{y} =< y_1, y_2, \ldots, y_m >$，其中m为漏洞代码的主要控制节点数，

y_i表示第i个节点是否为最后一个节点，可通过x, y_i馈入的程序来达到。，其中$x, y_i \in 0, 1, i \in \{1, 2, \ldots, m\}$。如图2所示，对于输入a，标签表示为$y^a =< 1, 0, 0, 0 >$，这意味着到达了$B_0$，但没有到达其他点。类似地，$y^b =< 0, 1, 0, 0 >$表示输入b可以让执行到达$B_0$和$B_1$，但是$B_2$、$B_3$也没有到达.$y^d =< 0, 0, 0, 1 >$都意味着最终到达了漏洞代码。为简单起见，我们将输入的每个字节直接映射到特征向量中的元素。这种方法使FuzzGuard以统一的方式处理具有各种输入格式的不同程序。对于每个数据，可以将其表示为向量$x =< x_1, x_2, \ldots, x_n >$，其中输入的最大长度。以及$x_i = byte_i + 1(x_i \in \{0, 1, \ldots., 256\})$，其中$x_i = 0$表示输入的第i个字节不存在（即输入的长度小于n）。

> 疑问：这里的m是不是还没有确定的，按照计算的距离、以及模型更新的拟合程度来决定调整主要控制节点数m的大小？但是如果没有固定的m，对于CNN来说就无法确定标签y的向量大小，所以还是需要确定m，那这里的m是确定好的还是一半会比准确的节点数大？或者是都可以取对应程序中最大最深的函数节点数？
>
> label y的大小就是关键节点的数目，这个是cfg决定的。AFLGo是原始版本的定向模糊，和这个没什么关系。至于是否有节点被遗漏，这个主要取决于图搜索算法与你的控制流图（cfg）的准确性，cfg原文应该是静态生成的，可能对间接调用有偏差，整体来讲应该影响不大
>
> Thanks to kdsj.

在设计了数据和标签的表示形式之后，我们仔细选择了深度学习模型。**这样的模型应该善于从输入中提取特征并进行正确的分类。**回想一下图像识别的问题：图像中对象的特征是由几个像素（即输入矢量中的元素，如图2所示）的组合表示的，可以通过CNN模型很好地提取它们[11，19，34，36]。同样，可以通过程序输入中几个字节的组合来表达影响其可达性的输入特征。实际上，**目标程序中if语句中的约束使用这些字节来确定执行方向。**因此，我们的想法是利用CNN来完成分类任务。一方面，与更适合使用字节序列进行训练的RNN相比，CNN擅长处理长数据。输入时间越长，RNN模型忘记先前功能的速度就越快。另一方面，训练CNN模型的时间比训练RNN模型的时间少得多，这适合于我们的问题（训练和预测所花费的时间应少于实际执行的时间）。

因此，我们选择使用3层CNN模型（详细实现请参见第5节）。通过这种方式，**第一层可以学习每个字节之间的关系，而其他两层则可以学习高维特征（例如，组合几个字节以在输入中形成一个字段，并组合多个字段以影响程序执行）。**发现这些提取的高维特征与目标程序的if语句中的约束相关（请参见第7节）。我们还将在第8节中讨论其他机器学习模型。请注意，由于实现方式不同（以不同方式解析输入），因此需要从头开始为每个程序训练该模型。探索不同程序之间的相似性并利用这种相似性来提高训练效率也是一个有趣的话题。

这样，我们可以让 **the carrier fuzzer** 运行一段时间以收集初始训练数据集。**在初始训练数据集达到平衡后，模型可以学习到输入所有节点的可达性。**该模型的目标是学习目标函数f（即$y = f(x)$），该函数由许多卷积运算组成。卷积操作使用许多过滤器从数据中提取特征：

$$y_i = w^T x_i = \sum_{i-k<j<i+k} w_j \cdot x_j, i \in \{1, 2, \ldots, n-k\}$$

其中k是滤波器w的卷积核的宽度。 梯度下降算法将更新每个滤波器的权重以减少损耗值，以实现更准确的预测。 对于分类任务，与交叉熵[18]损失相比，均方误差（MSE）[23]损失可以平衡每个类别的错误率，从而避免单个类别的错误率特别高。 考虑到**step-forwarding**方法需要训练有素的模型来尽可能准确地预测每个主要控制节点的可达性，因此我们选择使用MSE。 因此，当$loss = \frac{1}{m}\sum_{i=1}^{m}(y_i - y_i^p)^2$的值接近于0时，我们认为分类模型中的目标函数已经收敛，并且模型已准备好预测新生成的输入。

4.3 Prediction

初始化模型后，FuzzGuard利用该模型预测每个输入的标签并过滤掉那些不可达的对象。 **对于可及对象，它们将由目标程序执行，并进一步收集为新的标记数据以进行模型更新。** 特别是对于一个输入x，我们假设该模型只能预测B_t之前的主支配节点（即中间目标），并且预测结果为y^p。以下函数f'用于检查输入是否可到达目标节点 。

$$f'(y^p, t) = \begin{cases} reachable & y_i^p = 1 \wedge i \geq t \\ unreachable & y_i^p = 1 \wedge i < t \end{cases}$$

但是，在实际情况下，即使生成许多标记数据，我们也发现预测结果不够准确。 主要原因是，**即使新生成的输入可以达到目标，它们看起来也可能与训练集中的可到达输入完全不同。** 这是可以理解的：这些输入可能来自不同的种子。 从同一种子变异而来的大多数输入彼此略有不同，而在从不同种子变异而来的输入之间可以发现许多差异。 因此，**完全使用以前执行的输入可能无法训练非常准确的模型来预测新生成的输入的可达性。** 例如，使用从种子s^1变异而来的集合S_1中的数据训练的模型可能无法预测从种子s^2变异而来的S_2中的数据的标签。

> 这里新生成的输入是通过什么方式生成的？CNN有设置生成输入的操作吗？还是AFLGO生成的？

为了解决这个问题，**我们提出了一种代表性数据选择方法，该方法从每一轮突变中选择许多代表性输入以执行和训练。** 我们将固定数量的输入（例如5％）从该轮突变中**随机采样**作为该突变的代表数据。 **这样，可以在有限的时间内将更多种子产生的输入用于训练，从而提高了模型的准确性。** 但是，在实际执行中，即使只有5％的输入也构成了很大的数字（例如，超过2万），使用这些输入执行将花费大量时间。 我们的想法是对更少的输入进行采样。 假设在两个不同的突变中，分别从两个种子s^1和s^2生成了两组输入S_1和S_2。 **如果S1的分布与S2的分布相似，我们可以选择更少的输入。 但是，我们不能直接假设仅通过两个种子的相似性，两组的分布是相似的。** 这主要是因为不同的变异策略（例如，位和字节翻转，简单的算术，堆叠的调整和拼接）可能会极大地改变种子，并使后代看起来完全不同。因此，我们的想法是将种子与相应的突变策略进行比较。 **如果两个种子相似并且策略相同，那么我们考虑从组合集中选择较少的输入。** 我们定义两个种子s1和s2之间的种子相似度（SSD），如下所示：

$$d_{s^1,s^2} = 1 - \sum_{i=1}^{8n} s_i^1 \oplus s_i^2 / 8n$$

其中n是输入的最大字节长度，s_i表示种子s的第i位。 请注意，嵌入的不同选择不会影响SSD的定义，因为SSD是使用种子而不是嵌入后的向量定义的。 这样，我们可以通过其前身种子来衡量两组输入之间的相似性。 当SSD超过阈值（θ_s）时，我们认为种子s_2与种子s_1相似，并且应从s_2突变的输入中选择较少的数据。 例如，**在图2中，我们从种子e生成的输入集中选择了较少的数据（例如2%），因为e与种子b相似（例如SSD = 90%）。** 这样，我们可以选择较少的输入进行实际执行和培训，而不会影响模型的准确性。 根据我们的评估，应用该技术平均可以节省一半的时间（第6节）。

4.4 Model Updating

为了实现在线模型更新，**我们利用增量学习[26]通过每次提供一组数据而不是一次提供所有数据来训练动态模型。** 在这种情况下，将不断使用新传入的数据来扩展现有模型的知识。 增量学习旨在适应新数据而不会忘记学习模型的现有知识，并且不需要重新训练模型。当随着**the carrier fuzzer**不断生成和执行新输入而随着时间逐渐可用训练数据集时，可以应用增量学习。**增量学习还减少了等待数据收集的时间，并过滤掉了更多无法到达的测试用例。**

应更新在线深度学习模型以保持其准确性。 每当收集到一组新的标记数据时，就有机会进行模型更新。 但是，如果模型更新过于频繁，则训练时间将很长，这将影响模糊测试的效率。 相反，如果执行的更新频率较低，则该模型可能不准确。 因此，在此过程中，我们应该仔细选择何时执行模型更新。 另外，我们应该让更新足够快。 下面我们详细说明。

当模型“outdated”时，我们将执行模型更新。 当到达新的主导节点时，过时的模型不够准确。 在第一种情况下，当模型的误报率γ超过阈值θ_f时，我们更新模型。 为此，只要执行结果与预测结果不同，我们就连续记录模型的误报率，并继续关注γ。 **更新模型后，我们将误报率重置为零并再次记录。** 另一种情况是，当有一个新的主导节点$B_i(i > t)$包含平衡的标记数据时，就该用新数据更新模型了（请参见第4.3节）。这样，**模型可以从输入中学习新功能，这些新功能使程序可以执行从未被触及的新代码。** 使用这种方法，我们可以确保模型的准确性，同时保持模型更新的合理频率。

为了避免遗漏PoC（即避免过滤出任何PoC），我们将无法访问的输入临时存储在PUI中。 更新模型后，我们将使用新模型再次检查PUI中的输入，并选择可访问的输入以执行。 根据我们的评估，该模型足够准确，不会遗漏任何PoC。

5 Implementation

在本节中，我们描述FuzzGuard的实现，包括模型初始化，模型预测，模型更新以及FuzzGuard部署的详细信息。

model initialization。 在初始阶段，**只有在收集到足够的数据后，FuzzGuard才开始训练模型。并在收集了另一组数据（而不是单个输入）之后继续更新模型。** 此类数据应保持平衡（即，可到达的输入数量与不可到达的输入数量相似）。 特别是，在训练模型之前，应将所有输入都馈送到目标程序中以进行实际执行。 FuzzGuard记录输入的可达性。 一旦获得足够的2个平衡数据，

FuzzGuard便开始训练模型。 **然后，它利用经过训练的模型来预测新生成的输入的可到达性，如果目标程序可到达，则执行目标程序，并在实际执行中记录可到达性。** 收集此类数据以更新模型以获得更好的性能。 如前所述，DGF需要目标（潜在）的漏洞代码，其位置因模糊而闻名。 要设置漏洞节点的主要控制节点，我们生成目标程序的调用图（CG）和控制流图（CFG），并根据第4节中提到的定义设置主要控制节点。我们使用NetworkX [6]从LLVM生成的CG和CFG中自动找到主导节点。

Model Prediction and Updating。 为了进一步收集数据以更新模型，我们将SSD的θ_s设置为0.85，每轮突变的默认采样率为5%。 当SSD超过阈值时，采样率将降低到$(1-\theta_s)/5$（即小于3%）。 根据我们的评估，使用此值设置阈值具有最佳性能。 考虑到不同程序的模型精度差异很大，我们根据之前的执行动态更改θ_f：$\theta_f = 1 - acc_{avg}$，其中acc_{avg}表示先前更新的模型的平均精度。

Model Implementation。 对于训练模型，我们使用PyTorch [7]实现CNN模型。 它包含三个一维卷积层（k = 3，stride(步幅)= 1）。 请注意，一维卷积层将每个输入作为行序列。 每行有1024个字节。 每个卷积层后面都有一个池化层和一个ReLU [4]作为激活函数。 我们还具有Dropout层（禁用率= 20%），以避免神经网络过度拟合。 神经网络的末尾有一个完全连接的层，用于对目标路径中的错误代码的每个节点的可达性进行评分。 另外，我们使用Adam优化器[21]帮助学习功能快速，稳定地收敛到最优解。 当学习功能的损失值变得稳定时，训练过程结束。

Algorithm 1 Function Checker()

Input: *argv*, *timeout* and *input*

1: $fault \leftarrow 0$
2: **if** $check(input)$ is *reachable* **then**
3: $\quad fault \leftarrow run\_target(argv, timeout)$
4: $\quad label \leftarrow check\_trace(input)$
5: $\quad send(label)$
6: **end if**
7: **return** *fault*

Deployment of FuzzGuard。 为了实现数据共享，我们在AFLGo中的afl-fuzz.c中添加了Checker()函数。 算法1显示`Checker()`详细信息。 函数`Checker()`里面`run_target()`的所有参数（即算法1中的`argv`，`timeout`），并接收保存在内存中的输入。 在将输入提供给目标程序之前，将其发送到FuzzGuard（即算法1中第2行的`check(input)`）。 只有当FuzzGuard返回结果表明执行路径可以到达时，才使用输入执行目标程序（算法1中的第3行）。 执行目标程序后，`Checker()`函数中的`check_trace()`（算法1中的第4行）读取输入的可达性，并将其发送给FuzzGuard进行进一步学习（算法1中的第5行）。 我们计划发布FuzzGuard，以帮助社区研究人员。

6 Evaluation

在本节中，我们评估具有45个漏洞的FuzzGuard的有效性。 将结果与普通AFLGo进行比较。 **根据实验结果，FuzzGuard可以将模糊测试的性能提高到AFLGo的17.1倍。** 然后，我们将了解性能的提升并分解FuzzGuard的性能开销。 我们还将分析FuzzGuard的准确性并显示我们的发现。

6.1 Settings

我们首先选择了15种现实程序来处理10种常见文件格式，包括网络程序包（例如PCAP），视频（例如MP4，SWF），文本（例如PDF，XML），图像（例如PNG，WEBP，JP2），TIFF）和压缩文件（例如ZIP）。 不幸的是，无法编译三个程序（即mupdf，rzip，zziplib），并且两个程序（即apache，nginx）没有提供漏洞的详细信息。 因此，我们选择了其余10个作为目标程序，并选择了过去3年中的相应漏洞。 表1显示了每个漏洞的详细信息，包括漏洞代码的程序名称和行号（“漏洞代码”列）。 对于不同的输入格式，我们使用AFLGo提供的测试用例作为初始种子文件来开始模糊测试（我们相信，使用自己选择的初始种子文件，AFLGo将表现良好）。 所有实验和测量均在两台运行Ubuntu 16.04，具有16核（Intel®Xeon®CPU E5-2609 v4 @ 1.70GHz），64GB内存和3TB硬盘以及2个GPU（12GB Nvidia）的64位服务器上执行 GPU TiTan X）和CUDA 8.0。

6.2 Effectiveness

Table 1: Effectiveness of FuzzGuard.

No.	Program	Vuln. Code	$N_{Functions}$	$N_{Constraints}$	N_{Inputs}	$UR.$	$Filtered$	T_{AFLGo}	T_{+FG}	Speedup		
										FG	FG_1	FG_2
1	Bento4 v1.5.1.0	Ap4AvccAtom.cpp:83	676	1.8 K	1.8 M	38.1%	32.3%	44 h	29.9 h	**1.5**	1.3	1.4
2	Ettercap v0.8.2	ec_strings.c:182	420	41.5 K	49.2 M	99.0%	93.9%	80.5 h	6.1 h	**13.3**	1.1	8.3
3	GraphicsMagick v1.3.31	tiff.c:2375	3.3 K	170.3 K	32.1 M	95.9%	90.5%	94.8 h	11.1 h	**8.6**	5.9	7.5
4	GraphicsMagick v1.3.31	png.c:6945	4.9 K	319.8 K	30 M	96.6%	88.8%	200 h	23.4 h	**8.5**	2.0	8.2
5	GraphicsMagick v1.3.31	png.c:7503	1.5 K	21.9 K	16.4 M	99.9%	84.1%	13.2 h	3.9 h	**3.4**	1.0	3.1
6	GraphicsMagick v1.3.31	png.c:5007	4.4 K	317.1 K	16 M	99.9%	34.3%	200 h	132.3 h	**1.5**	1.0	1.5
7	GraphicsMagick v1.3.30	png.c:3810	3.1 K	168.4 K	22.6 M	99.9%	32.8%	31.9 h	22.4 h	**1.4**	1.0	1.4
8	GraphicsMagick v1.3.27	webp.c:716	10.7 K	749.3 K	67.5 M	99.5%	93.4%	200 h	15.2 h	**13.2**	9.7	9.7
9	GraphicsMagick v1.3.26	png.c:7061	4.9 K	320 K	56.9 M	98.4%	93.3%	200 h	16.2 h	**12.3**	8.4	9.4
10	GraphicsMagick v1.3.26	tiff.c:2433	4.4 K	316.4 K	78.4 M	75.3%	69.5%	200 h	66.7 h	**3.0**	2.4	2.7
11	GraphicsMagick v1.3.26	rle.c:753	9.2 K	379.3 K	17.7 M	99.4%	70.5%	30.8 h	10.4 h	**3.0**	1.5	2.6
12	GraphicsMagick v1.3.26	list.c:232	3.6 K	172.1 K	73 M	37.2%	28.4%	200 h	146.4 h	**1.4**	1.6	1.4
13	ImageMagick v7.0.8-13	msl.c:8353	85.5 K	5.4 M	7.3 M	99.2%	92.4%	200 h	15.4 h	**13.0**	3.3	12.9
14	ImageMagick v7.0.8-3	dib.c:1306	117.1 K	7, 883.1 M	3.2 M	99.9%	54.4%	200 h	91.4 h	**2.2**	1.0	1.6
15	ImageMagick v7.0.8-3	bmp.c:2062	117.3 K	6, 306.9 M	3 M	99.9%	52.3%	200 h	95.6 h	**2.1**	1.0	2.0
16	ImageMagick v7.0.7-16	webp.c:769	23.9 K	145.7 K	11.5 M	99.1%	93.9%	200 h	12.6 h	**15.9**	15.5	14.7
17	ImageMagick v7.0.7-16	webp.c:403	14.8 K	116.1 K	19.1 M	96.0%	90.7%	200 h	19.8 h	**10.1**	9.4	10.7
18	ImageMagick v7.0.7-1	tiff.c:1934	149.1 K	1.2 M	9.4 M	98.5%	92.5%	200 h	15.2 h	**13.1**	1.6	8.7
19	ImageMagick v7.0.5-5	bmp.c:894	102.4 K	926.5 K	12.9 M	64.3%	59.9%	200 h	80.5 h	**2.5**	1.7	2.5
20	Jasper v2.0.14	jp2_enc.c:309	13.9 K	17.7 M	28 M	99.4%	50.9%	200 h	99 h	**2.0**	1.7	1.9
21	Jasper v2.0.10	jpc_dec.c:1700	740	9.7 K	11.3 M	99.7%	94.3%	46.9 h	3.7 h	**12.7**	1.4	11.1
22	Jasper v2.0.10	jpc_dec.c:1881	1.7 K	36.8 K	6.1 M	99.9%	94.0%	19.7 h	1.6 h	**12.0**	1.0	10.0
23	Jasper v2.0.10	jas_seq.c:254	1.1 K	11.8 K	22.3 M	62.4%	56.0%	200 h	89 h	**2.2**	2.0	2.2
24	Libming v0.4.8	decompile.c:1930	104	5.3 K	38.6 M	99.9%	70.2%	200 h	63 h	**3.2**	1.0	3.1
25	Libming v0.4.7	parser.c:1645	75	4.7 K	32.3 M	99.8%	94.7%	200 h	11.7 h	**17.1**	8.5	14.1
26	Libming v0.4.7	parser.c:64	170	2.7 K	16.1 M	91.9%	86.6%	200 h	27.2 h	**7.3**	1.7	6.2
27	Libming v0.4.7	parser.c:3381	79	790	38.4 M	99.7%	69.9%	200 h	61.3 h	**3.3**	2.0	3.2
28	Libming v0.4.7	parser.c:3095	25	217	46.8 M	92.9%	65.7%	200 h	70 h	**2.9**	1.9	2.8
29	Libming v0.4.7	parser.c:2993	22	386	45.9 M	97.2%	64.8%	200 h	71.8 h	**2.8**	1.7	2.7
30	Libming v0.4.7	parser.c:3126	24	294	77 M	92.9%	63.6%	200 h	75.3 h	**2.7**	2.0	2.5
31	Libming v0.4.7	parser.c:3232	55	423	12.6 M	99.8%	61.3%	6.1 h	2.8 h	**2.2**	2.0	1.8
32	Libming v0.4.7	parser.c:3221	38	308	13 M	99.9%	43.2%	14 h	8.2 h	**1.7**	1.0	1.6
33	Libming v0.4.7	parser.c:3250	32	340	16.6 M	99.9%	46.0%	7.3 h	4.4 h	**1.7**	1.0	1.4
34	Libming v0.4.7	parser.c:3089	36	396	19.6 M	99.9%	43.3%	5.2 h	3.4 h	**1.5**	1.0	1.1
35	Libming v0.4.7	parser.c:3061	37	637	18.9 M	99.8%	37.2%	3.4 h	2.5 h	**1.4**	1.0	1.1
36	Libming v0.4.7	parser.c:3071	34	1.1 K	17.6 M	99.9%	33.6%	3.8 h	2.9 h	**1.3**	1.0	1.1
37	Libming v0.4.7	parser.c:3209	34	402	30.7 M	99.9%	27.7%	8.9 h	6.9 h	**1.3**	1.0	1.2
38	Libming v0.4.7	outputtxt.c:143	64	2.2 K	27.3 M	65.5%	24.6%	7.7 h	6.1 h	**1.3**	1.1	1.1
39	Libtiff v4.0.9	tif_dirwrite.c:1901	728	14.4 K	8.6 M	99.9%	91.4%	9.6 h	1.3 h	**7.4**	1.0	4.8
40	Libtiff v4.0.7	tif_swab.c:289	631	13.1 K	44.7 M	99.7%	52.8%	29.6 h	15 h	**2.0**	1.1	1.3
41	Libtiff v4.0.7	tiffcp.c:1386	728	13.3 K	15.6 M	99.9%	51.7%	8.9 h	4.6 h	**1.9**	1.0	1.7
42	Libtiff v4.0.7	tif_read.c:346	416	11.6 K	60.6 M	79.5%	36.3%	77.9 h	49.8 h	**1.6**	1.4	1.5
43	Libxml2 v2.9.4	SAX2.c:2035	418	15.7 K	92.6 M	99.9%	94.4%	200 h	17.6 h	**11.3**	1.0	5.2
44	Podofo v0.9.5	PdfPainter.cpp:1945	19.8 K	44.1 K	2.6 M	99.3%	79.7%	200 h	40.7 h	**4.9**	4.8	1.8
45	Tcpreplay v4.3.0-beta1	get.c:174	23	1.1 K	203.3 M	53.2%	49.5%	200 h	105.4 h	**1.9**	1.7	1.9
Avg.			15.5K	315.9 M		91.7%	65.1%			**5.4**	2.6	4.4

为了展示FuzzGuard的有效性，我们使用组合了FuzzGuard的AFLGo，对10个真实程序中已知存在45个漏洞的原始AFLGo进行了评估（如表1所示）。**组合了FuzzGuard的AFLGo和vanilla AFLGo**的理想比较是比较在AFLGo组合了FuzzGuard（T_{+FG}）时使用模糊测试时间以及AFLGo（T_{AFLGo}）进行模糊测试的时间。 但是，我们不能直接使用相同的种子输入来比较AFLGo和AFLGo + FuzzGuard的模糊测试过程。这是因为突变是随机的，并且输入的生成顺序（即使来自相同的种子输入）在两个模糊测试进程上，这使得执行所花费的时间完全不同。 **因此，我们的想法是在两个不同的模糊测试过程中，使生成的输入序列相同。** 特别是，对于目标程序的漏洞，我们使用vanilla AFLGo执行模糊测试并记录所有变异的输入$I_{AFLGoin}$顺序（表1中所示的输入数$N_{Inputsis}$）的序列，直到触发目标漏洞（例如crash）为止或超时（在我们的评估中为200小时）。 在此过程中，还记录了模糊时间T_{AFLGo}（如表1所示）。 然后，我们利用配备FuzzGuard的相同输入序列（I_{AFLGo}去测试AFLGo），记录过滤后的输入$I_{filtered}$（过滤后的输入的数量为$N_{filtered}$，以及过滤后的输入与所有生成的输入的比率，$filtered = N_{filtered}/N_{Inputs}$如表1所示）。 我们还记录了FuzzGuard T_{FG}的时间成本，包括训练和预测的时间。 这样，我们便可以知道配备FuzzGuard的时间，并将时间与TAFLGO进行比较。 T_{+FG}可以计算如下：

$$T_{+FG} = T_{AFLGo} - \sum_{i \in I_{filtered}} t_i + T_{FG}$$

$I_{filtered}$是FuzzGuard过滤出的输入，t_i表示使用输入i执行目标程序所花费的时间。

请注意，I_{AFLGo}中的最后一个输入是AFLGo生成的第一个PoC（如果目标程序崩溃，例如表1中的#1和#2）或AFLGo超时之前生成的最后一个输入（没有崩溃发生，例如 表1中的#8和#9）。 我们强调FuzzGuard不知道给定的输入是否为最后一个输入。 在模糊测试过程中，FuzzGuard以与先前输入相同的方式对待最后一个输入。 与FuzzGuard相比，在I_{AFLGo}中随机删除输入的方法将随机决定是否删除最后一个输入。 从表1中我们可以看到FuzzGuard平均减少了65.1％的输入。 如果通过随机方法丢弃了相同数量的输入（65.1％），则最后一个输入（可能的PoC，例如表1中的#1和#2）也可能被丢弃，可能性为65.1％。 相比之下，**FuzzGuard的误报率为0.02％（请参见第6.3节），这意味着即使FuzzGuard丢弃了65.1％的输入，丢弃PoC的可能性也仅为0.02％。**

Landscape。 结果如表1所示。表1中的45个漏洞包括最近3年发现的27个CVE和18个新的未公开漏洞（请参见6.5节）。在我们的评估中，**当FuzzGuard对其他漏洞（例如CVE-2017-17501，表1中的第4行）执行目标模糊测试时，发现了未公开的漏洞（例如，表1中的第6行）。 请注意，此未公开漏洞的漏洞代码实际上不是此过程中的目标。 然后，我们将新发现的漏洞代码设置为目标，并尝试利用AFLGo对其进行重现。 不幸的是，在时间限制（200小时）内，AFLGo无法触发该漏洞。 AFLGo + FuzzGuard均无法触发该漏洞。 但是，AFLGo + FuzzGuard确实将时间从200小时节省到23.4小时（加速8.5倍）。** 从表中我们发现，**对于所有漏洞，FuzzGuard可以将AFLGo的运行时性能从1.3倍提高到17.1倍**（请参阅表1中的“ Speedup”列，其中$Speedup = T_{AFLGo}/T_{+FG}$）。 平均性能提高了5.4倍。 请注意，这种性能提升已添加到已经优化的DGF（即AFLGo）中。

> 这里为什么不能重现了呢，FuzzGuard不是一个定向的模糊测试工具吗？不是DFG吗？

Understanding the performance boost。 为了了解FuzzGuard对于不同程序和漏洞的性能，我们进一步研究speedup之间的关系，**模型开始训练的时间**以及**无法到达的输入的比率**之间的关系。

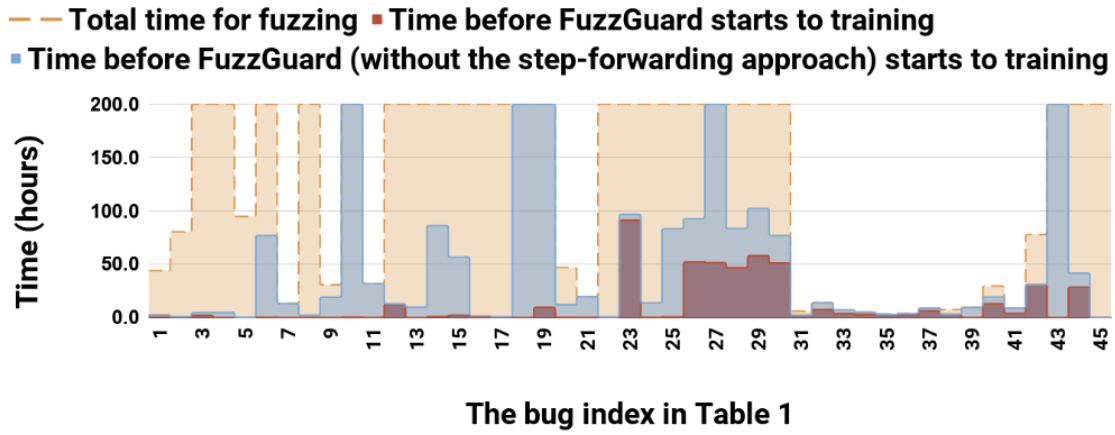

Figure 3: Start time of the first training in FuzzGuard.

- 训练模型越早，可以节省更多的时间。 图3显示了针对表1中的漏洞（红条）开始训练每个模型的时间。 我们可以看到，稍后训练的模型（例如，#20，#24，#27）获得的加速不超过3.3倍，而先前训练的模型可以实现超过17倍的加速。 **这主要是因为漏洞节点越早获得平衡的标签数据，就可以越早训练模型以筛选出无法到达的漏洞代码输入。** 结果，可以过滤出更多的输入，以节省不必要的执行时间。
- **the carrier fuzzer** 生成的输入越多，FuzzGuard的效果就越差。 例如，如表1所示，当超过40％的输入是可到达的（“ UR”列是不可到达的输入的比率）时，FuzzGuard所获得的加速小于2倍（例如，漏洞#1，表1中的#12和#45），在特殊情况下，**如果从入口点到目标漏洞代码的路径中没有if语句或约束，则所有生成的输入都是可访问的，因此这里无需训练深度学习模型。**

Complicated functions。要评估FuzzGuard在处理具有多个约束和分支的复杂函数时，我们测量了表1中每个漏洞的路径中唯一函数和约束的数量。从表中，我们可以看到平均唯一函数和约束的数量为1.55万和3.159亿。 超过50％的漏洞受到成千上万个约束的保护（例如GraphicsMagick和ImageMagick中的漏洞）。 对于这些漏洞，FuzzGuard的速度从1.4提高到15.9。 **对于受数百万级别约束的某些漏洞（例如表1中的＃13和＃18），FuzzGuard的速度提高了10倍以上。** 结果表明，FuzzGuard可以很好地处理复杂的函数，这对于解决传统的约束可能是非常耗时的。

Cost。在我们对表1中的45个漏洞的评估中，在线模型的训练平均花费60分钟， 包括13.5％的数据收集，0.5％的数据嵌入和86％的培训过程。 请注意，训练所花费的时间仅占模糊器产生输入的时间的6％（平均15个小时）。 FuzzGuard平均花费的总时间为1.4小时，仅占模糊测试总时间的9.2％（表1中的T_{+FG}）和AFLGo执行的模糊测试过程的总时间的2.5％（表1中的T_{AFLGo}）。 这样的时间段足以使模糊器处理70.4万个输入，这比直接执行目标程序进行测试要有效得多。

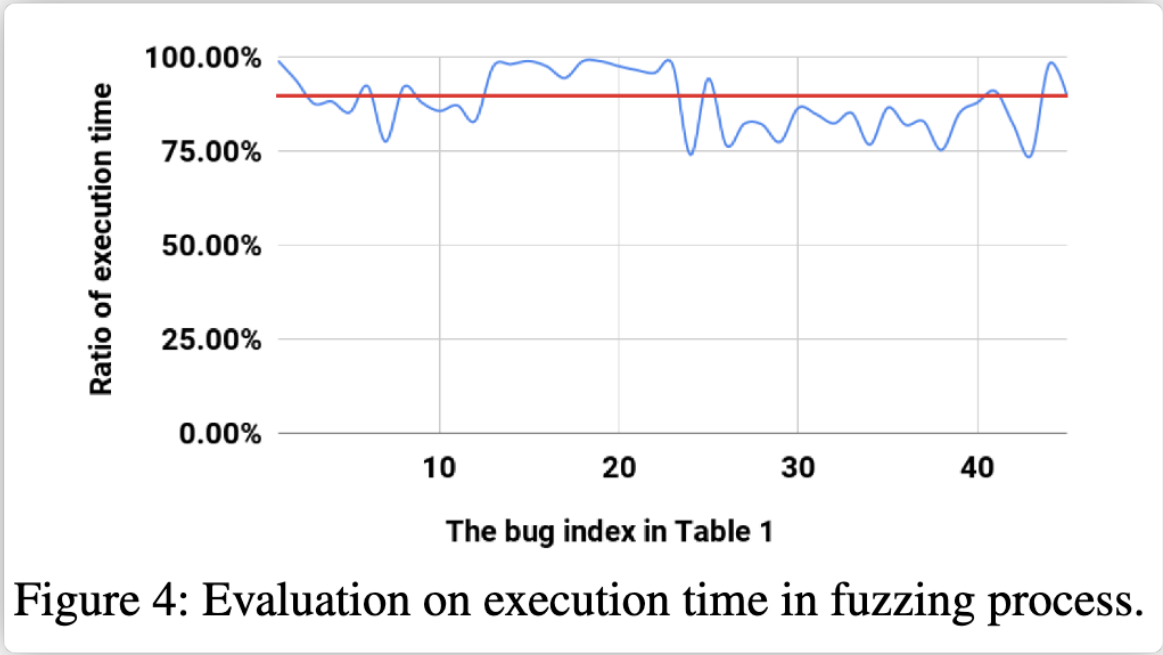

Figure 4: Evaluation on execution time in fuzzing process.

为了了解FuzzGuard可以节省的模糊时间上限，我们使用AFLGo对45个漏洞（如表1所示）执行24小时模糊测试。 从图4中可以看出，目标程序的平均执行时间超过了总模糊时间的88%，这意味着FuzzGuard可以节省的平均模糊时间上限约为88%。 FuzzGuard的时间成本应小于限制。

6.3 Accuracy

我们测量FuzzGuard的准确性。 准确性基于是否正确判断可达性。 它的行为越准确，就可以过滤出越多的输入。 请注意，不会将任何PoC遗漏，因为过滤后的输入将保存在PUI中，这将由更新的模型进一步检查。**一个更准确的模型可以在池中找到可到达的模型，并让目标程序与它们一起执行，从理论上讲，它们不会有假阴性。** 但是，在实际执行中，我们通常会为模糊设置超时时间。 在这种情况下，如果在超时之前未找到错误的负输入，则它将被忽略。 幸运的是，在我们评估45个bug时，由于模型准确，PUI中未发现PoC。 我们将误报率定义为：$fpr = N_{fp}/N_n \times 100\%$，其中$N_n$代表AFLGo生成的不可达输入的数量，$N_{fp}$是指不能达到错误代码但被FuzzGuard视为可达代码的输入数量。 负率是：$f_{nr} = N_{fn}/N_p \times 100\%$，其中$N_p$表示可访问输入的数量，$N_{fn}$是指表示可访问输入的数量，但被FuzzGuard过滤掉。f_{pr}越高，执行不可达输入的时间就越多。f_{nr}越高，在模糊测试中执行PoC的可能性就越大。 通过$acc = \frac{N_p+N_n-N_{fp}-N_{fn}}{N_p+N_n}$计算精度。

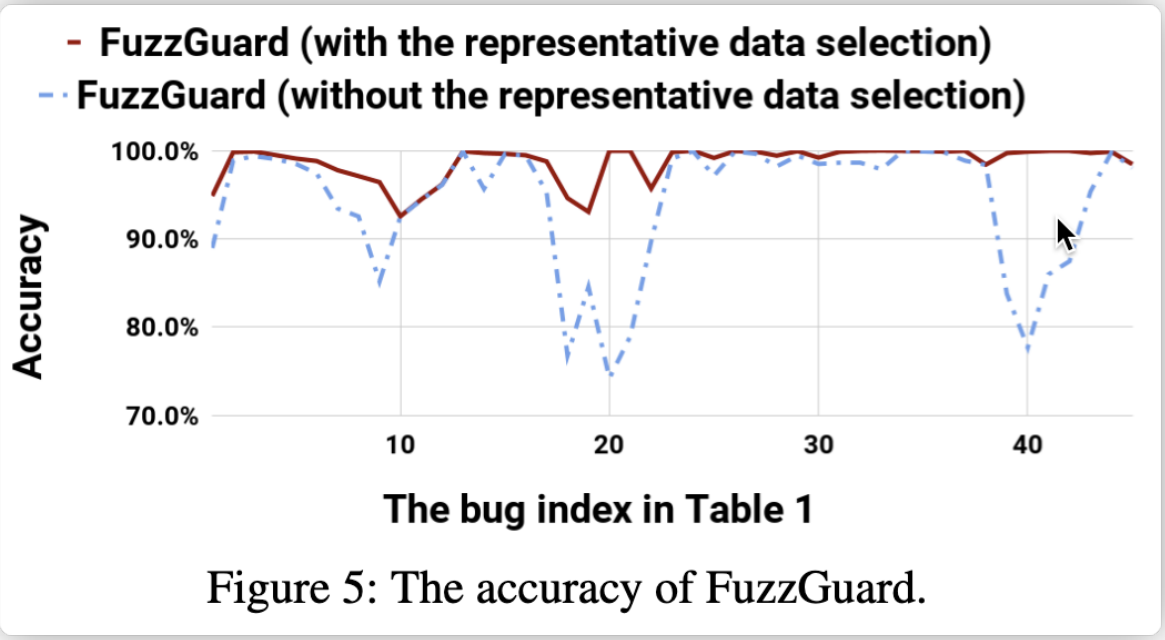

Figure 5: The accuracy of FuzzGuard.

从图5中，我们可以发现FuzzGuard非常准确（范围从92.5%到99.9%）。 平均准确度是98.7%。 所有漏洞的误报率平均为1.9%。 请注意，误报不会使PoC遗漏。 它们也不会增加执行输入所花费的时间（如果没有FuzzGuard，此类输入始终由程序执行）。 假阴性率可以忽略不计，平均为0.02%。 只有4个带有误报的漏洞，最高的是0.3%。 我们进一步手动检查那些假阴性，并确认这些输入中没有PoC。 如前所述，即使包含PoC，FuzzGuard也会将其保存到PUI中，以便通过更新的模型进行进一步测试（不会丢失PoC）。 如此精确的模型使FuzzGuard具有高性能。

假阳性和假阴性(误报)的主要原因是缺乏均衡的代表性数据。 例如，如果FuzzGuard与先前的可达输入足够相似，则FuzzGuard可以将其预测为可达（即误报）。 输入的执行路径也可以类似于漏洞代码的路径（覆盖漏洞代码的某些主要节点）。 但是输入中的某些字节最终会停止对漏洞代码的执行。**false negative**可能会使程序通过以前从未见过的执行路径到达目标漏洞代码。 如果模型可以了解这些新的执行路径，则预测将更加准确。 在我们的评估中，经过长时间的模糊处理后，看不见的路径数量减少了，这可能是误报率低的原因。

6.4 Contribution of Individual Techniques

为了研究step-forwarding方法和代表性数据选择的个人贡献，我们针对表1中的所有bug评估了使用和不使用每种技术时的性能提升。特别是，为了比较起见，对于要测试的每个bug，我们使用相同的输入顺序。 我们首先在没有逐步方法的情况下进行评估，并记录性能的提高（表1中的FG1列）。 然后，我们不使用表示数据选择，而是记录相应的性能提升（表1中的FG2列）。 结果表明，与普通的AFLGo实施相比，**FuzzGuard（使用这两种技术）可以实现5.4倍的加速，而没有step-forwarding方法的FuzzGuard和没有代表性数据选择的FuzzGuard只能分别实现2.6倍和4.4倍的加速。**

我们还作了进一步分析。 众所周知，**step-forwarding方法旨在帮助FuzzGuard在模糊测试过程中更早地获取平衡数据，从而进一步使训练过程更早开始。** 因此，我们想衡量step-forwarding方法可以提供多少帮助。 我们记录了采用和不采用step-forwarding方法的第一次训练的开始时间（见图3）。 图中的x轴显示了表1中的漏洞索引，y轴给出了以小时为单位的开始时间。 从图上我们发现，如果不使用step-forwarding方法，由于缺少平衡数据，FuzzGuard无法触发针对14个漏洞（例如，#5，#6和#7）的训练过程。 对于其他漏洞，即使训练过程已开始，与使用step-forwarding方法的模型相比，开始时间平均也要延迟17.4小时。 这也延迟了筛选过程，并最终影响了整体性能。

关于代表性数据选择，我们还测量其对模型准确性的影响。 对于每个漏洞，我们都会记录是否使用代表数据选择模型的准确性。 结果如图5所示。x轴显示bug索引，y轴显示模型的准确性。 从图中可以看出，平均而言，代表性数据选择的准确性提高了4.4％。 在某些情况下（图5中的#14，#21和#40），在没有代表性数据选择的情况下，模型的准确性急剧下降。**基于上面的单独评估，我们发现Fuzz-Guard既需要step-forwarding方法又需要代表性数据选择 效率和准确性。**

6.5 Findings

有趣的是，**在我们的评估中，我们发现了23个未公开的bug（其中4个是零日漏洞）。 请注意，未公开漏洞的漏洞代码实际上不是我们的目标。**

FuzzGuard的目标是通过删除不可达的输入而不是触发新的bug来提高模糊测试的效率。
FuzzGuard + AFLGo发现的所有漏洞甚至可以最终由AFLGo发现。 未公开的漏洞已在相应程序的新版本中修复。对于四个零日漏洞，我们成功获取了CVE号。 当对其他漏洞执行目标模糊测试时，将触发该漏洞。 例如，在CVE-2017-17501的模糊测试过程中发现了CVE-2018-20189; 和CVE-2019-7663在CVE-2016-10266的模糊测试过程中找到。 此外，在验证CVE-2016-9831时，我们还发现CVE-2019-7581和CVE-2019-7582。在手动分析了未公开的bug和零日漏洞之后，我们发现它们的位置非常接近漏洞代码（即目标在有针对性的模糊测试）。 对于示例，列表2和列表3分别显示了触发CVE-2017-17501和CVE-2018-20189的调用堆栈。 两个调用堆栈的前8个主要控制节点相同，而只有最后一个基本块不同。 我们猜测漏洞代码附近的代码更有可能包含一个新漏洞。

```
0x665abb in WriteOnePNGImage coders/png.c:7061
0x677891 in WriteMNGImage coders/png.c:9881
0x479f3d in WriteImage magick/constitute.c:2230
0x47a891 in WriteImages magick/constitute.c:2387
0x42bb9d in ConvertImageCommand magick/command.c:6087
0x43672e in MagickCommand magick/command.c:8872
0x45eeaf in GMCommandSingle magick/command.c:17393
0x45f0fb in GMCommand magick/command.c:17446
0x40c895 in main utilities/gm.c:61
```

Listing 2: The sequence of calls to trigger CVE-2017-17501.

```
0x548b71 in WriteOnePNGImage coders/png.c:7263
0x551d97 in WriteMNGImage coders/png.c:9881
0x450f60 in WriteImage magick/constitute.c:2230
0x4515da in WriteImages magick/constitute.c:2387
0x4215bc in ConvertImageCommand magick/command.c:6087
0x427e48 in MagickCommand magick/command.c:8872
0x44113e in GMCommandSingle magick/command.c:17393
0x441267 in GMCommand magick/command.c:17446
0x40be26 in main utilities/gm.c:61
```

Listing 3: The sequence of calls to trigger the zero-day vulnerability.

7 Understanding

我们的评估结果表明，**FuzzGuard可以非常有效地过滤出无法到达的输入，平均准确率为98.7%**。我们想从特征中了解为什么FuzzGuard具有如此出色的性能。 如果FuzzGuard掌握的特征是合理的，则FuzzGuard的结果也是可以理解的。 为了实现此目标，我们的想法是从模型中提取特征并进行手动分析。 但是，众所周知，由深度神经网络提取的高维特征很难直接理解。 受显着性图[8]的启发，我们的想法是将特征投影到单个字节（称为关键特征），并检查关键特征是否会影响目标程序的执行。

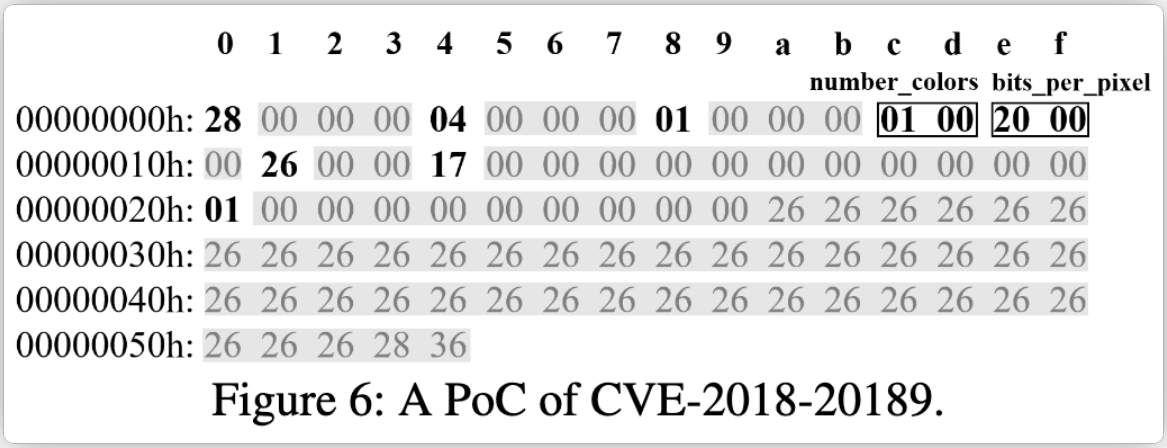

Figure 6: A PoC of CVE-2018-20189.

特别地，**为了获得关键特征，我们设计了一种基于掩码的方法来获取模型使用的输入的相应关键字节。** 基本思想如下：我们使用掩码（即与输入长度相同的向量）覆盖输入x的字节（覆盖字段设置为0）。 如果覆盖的输入具有与未覆盖的输入相同的预测结果（即$f(mask \cdot x) = f(x)$，其中f是FuzzGuard使用的CNN模型），则覆盖的字段将不会影响预测结果，因此表示它们不是关键特征。通过逐步增加输入字段的覆盖范围，我们最终可以获取所有关键特征。 此时的遮罩称为最大遮罩。例如，一个输入如图6所示。该掩码将输入的阴影部分的值设置为0。当$f(x) = f(m \cdot x)$时，阴影部分不会影响输入X的可达性 。 因此，我们将更多字节加阴影并重复此过程。 这里的问题是涵盖的字段组合太多。 因此，我们的想法是利用梯度下降来计算最大遮罩。 特别地，我们根据预测标签y^p与x的实际标签y之间的偏差来调整遮罩，直到$y^p = y$。 为了利用这种方法，我们设计了一个损失函数，该函数不仅考虑预测值与实际值之间的偏差，而且还考虑了遮罩中的覆盖率，如下所示：

$$loss = \frac{\sum_{i=1}^{n} mask_i}{n} + \frac{\sum_{i=1}^{m} (y_i^x - y_i)^2}{m}$$

其中，n是掩码的字节数，m是第4.3节中提到的y的长度。 当y^p和y之间的差距最小并且所覆盖的字节数最大时，x中的未覆盖字节是关键特征，这是影响FuzzGuard查看的可达性的输入字段。 这样，可以将关键特征与目标程序中的约束条件进行比较，以检查关键特征是否真的可以影响执行。

举个例子，图6中显示了CVE-2018-20189的PoC（PNG文件）。此PoC中的关键特征未加阴影。经过手动分析，我们验证了输入中偏移从 0x0e 到 0x0f（bits_per_pixel in List 1）中的字段决定了第六行中分支的执行方向； 输入中的偏移 0x0c到0x0d（number_colors in List 1）中的字段会影响执行。 例如，当`bits_per_pixel <16` or `number_colors != 0`时，将执行漏洞代码。当`bits_per_pixel > 8`时将触发该漏洞。 通过以上分析，**我们可以确定关键特征确实会影响输入的可达性，这意味着当此类输入的数量足以训练时，模型就成功地将字段捕获为特征。**

```
1  ThrowReaderException(...);
2 if (dib_info.colors_important > 256)
3  ThrowReaderException(...);
4 if ((dib_info.image_size != 0U) && (dib_info.image_size
       > file_size))
5  ThrowReaderException(...);
6 if ((dib_info.number_colors != 0) ||
       (dib_info.bits_per_pixel < 16)) {
7    image->storage_class=PseudoClass;
```

Listing 1: The vulnerable code of CVE-2018-20189.

8 Discussion

Benefit to input mutation。 当前大多数的模糊器都集中在对输入进行突变以增强模糊性能（例如，AFL [2]，AFLFast [10]和AFLGo [9]）。 与他们不同的是，我们的想法是帮助DGF过滤掉无法到达的输入。有趣的是，我们发现我们的方法也有可能帮助他们优化输入突变的策略。 **如果模糊器知道影响执行的输入中的字段，则可以对其进行突变以使程序执行到达漏洞代码。 修改其他字段将无助于此过程**。 基于对FuzzGuard提取的功能的了解，我们发现FuzzGuard可以学习影响执行的字段（请参见第7节）。 因此，FuzzGuard可以在输入突变的过程中进一步帮助DGF。

Learning models。 直观上，卷积架构使用局部模式。 但是，只要CNN具有足够的神经网络层，它实际上就可以处理非局部模式。 RNN与之类似：**当具有足够的图层时，它可以处理非局部模式；否则，它将忘记以前的功能**。 但是，RNN处理长数据的开销非常大。 因此，我们选择使用三层CNN。 在我们的评估中，结果显示CNN取得了良好的性能（平均1.9％的假阳性率和0.02％的假阴性率），这可能表明输入中的大多数关键特征是局部模式（例如，图6中的fieldbits_per_pixel）。 这是可以理解的：对于if语句中的单个约束，通常依赖于输入中的本地字节来进行决策。

> 文中提到fuzzer知道影响执行的输入中字段，那对于这些字段是不是可以用于网络协议的模糊测试协议字段中?

Memory usage。 从理论上讲，我们可以将无法访问的输入永远保留在内存中，以避免丢失PoC。 但是，在实际情况下，内存是有限的。 因此，我们的想法是删除那些根本无法到达错误代码的输入。 换句话说，如果数十次更新模型将输入判断为“无法访问”，则很可能无法到达错误代码。 这样，我们可以节省内存，同时保持准确性。 根据我们的评估，不会以这种方式丢弃PoC。

9 Related Work

Traditional Fuzzers.。 近年来，提出了许多最新技术。 **AFL [2]是其中的代表性CGF模糊器**，为其他模糊器提供了指导。 例如，Böhme等[10]使用马尔可夫模型构建模糊过程。 它选择行使低频执行路径的种子，然后对其进行变异以覆盖更多代码以查找漏洞。 FairFuzz[24]与AFLFast [10]类似，但

是它提供了新的变异策略（即，覆盖，删除和插入）。 Gan等。 [16]通过更正AFL中的路径覆盖计算来解决AFL中的路径冲突问题。 AFL的另一个变体是AFLGo [9]，**它选择执行路径更接近目标路径的种子，并对它们进行变异以触发目标漏洞。** Chen等[12]**通过种子选择和突变的新策略来改善AFLGo。** 一些研究人员通过传统程序分析来提高有效性。 例如，Li等[25]使用静态分析和仪器来获取执行期间的幻数位置，并将其应用于突变以提高测试用例的执行深度。 Chen等[13]使用**动态技术（例如，彩色污点分析）来发现漏洞。** Rawat等[30]使用静态和动态分析技术来获取控制流和数据流信息，以提高突变的有效性。 Chen等[14]发现内存布局以执行准确的模糊测试。 与他们的工作不同，我们利用基于深度学习的方法来筛选不可达的输入，以提高模糊测试的性能。

Learning-based Fuzzers. 也有一些使用智能技术的模糊器。 例如，You等 [35]**从CVE描述中提取脆弱的信息并触发Linux内核中的错误。** Wang等[33]通过**概率上下文相关语法（PCSG）从大量程序输入中学习语法和语义特征，然后从该PCSG生成程序输入。** 同样，以前有一些研究[17，28，29]**训练静态模型以通过生成更可能触发错误的输入来改进模糊器的突变策略。** Godefroidet等 [17]**应用RNN通过大量的测试用例来学习程序输入的语法，并进一步利用所学的语法来产生新的输入。** Rajpal[29]**利用LSTM模型来预测输入中合适的字节，并根据以前的模糊测试经验对这些字节进行突变以最大化边缘覆盖率。** Nichols等。 [28] **Traina GAN模型可以预测输入的执行路径。** Chenet al[15]**应用梯度下降算法来解决路径约束问题，并在漏洞代码的输入中找到关键字节。** She等[31]**还利用梯度下降平滑神经网络模型并学习程序中的分支以提高程序覆盖率。** 与这些研究主要致力于使输入突变以实现高代码覆盖率或有效地达到目标错误代码的研究不同，**FuzzGuard的目标是帮助DGF过滤掉无法访问的输入，该输入与其他模糊器是互补且兼容的，而不是替代它们。**

10 Conclusion

最近，DGF可以有效地发现具有潜在已知位置的错误。 **为了提高模糊测试的效率，当前的大多数研究都集中在对输入进行突变以增加达到目标的可能性上，但是在过滤掉无法到达的输入方面做得很少。** 在本文中，我们提出了一种基于深度学习的方法，称为FuzzGuard，它可以预测程序输入的可达性而无需执行程序。 我们还提出了一套新颖的技术来应对缺乏代表性标签数据的挑战。 对45个实际漏洞的结果表明，FuzzGuard可以使速度提高多达17.1倍。 我们进一步展示了FuzzGuard学习的关键特征，这些特征确实会影响执行。