一种约束制导的机器学习框架漏洞检测方法

刘 昭[1] 邹权臣[1] 于 恬[1] 王 旋[1] 张德岳[1] 孟国柱[2] 陈 恺[2]

[1](北京奇虎科技有限公司 AI 安全实验室 北京 100015)

[2](中国科学院信息工程研究所 北京 100195)

摘 要 随着机器学习在社会各领域中自主决策场景的广泛应用,人们对机器学习框架中潜在漏洞的担忧也在日益增加.然而,由于其复杂的实现,针对框架的系统化、自动化测试成为一项艰巨的任务.现有对机器学习框架测试的研究在生成有效测试数据方面尚不成熟,导致测试数据无法通过合法性校验并因此无法检测到目标漏洞.本文提出了 ConFL,一种基于约束的机器学习框架模糊测试工具.ConFL 能够自动从框架源代码中提取约束而无需任何先验知识.在约束的指导下,ConFL 可以生成能够通过校验的有效输入,并执行到框架更深层次的代码逻辑.此外,本文设计了一种算子分组调度技术来提高模糊测试的效率.为了证明 ConFL 的有效性,本文主要在 TensorFlow 框架上评估了其性能.测试发现,与现有的 SOTA 工具相比,ConFL 能够覆盖更多的代码行,并生成更多有效的测试数据;在相同版本的 TensorFlow 框架上,ConFL 能检测出更多的已知漏洞.此外,ConFL 在不同版本的 TensorFlow 中发现了 84 个未知漏洞,这些漏洞全部被官方修复并被分配了 CVE 编号,其中包括 3 个严重漏洞,13 个高危漏洞.最后,本文还在 PyTorch 和 PaddlePaddle 中进行了通用性测试,迄今为止发现了 7 个漏洞.

关键词 机器学习框架;约束提取;算子测试;模糊测试;漏洞检测

中图法分类号 TP391 **DOI 号** 10.11897/SP.J.1016.2024.01120

Constraint-Guided Vulnerability Detection Techniques for Machine Learning Framework

LIU Zhao[1] ZOU Quan-Chen[1] YU Tian[1] WANG Xuan[1] ZHANG De-Yue[1]
MENG Guo-Zhu[2] CHEN Kai[2]

[1](*AI Security Lab*, *Qihoo* 360, *Beijing* 100015)

[2](*Institute of Information Engineering*, *Chinese Academy of Sciences*, *Beijing* 100195)

Abstract The increasing integration of machine learning (ML) in various sectors for decision-making automation brings to light significant concerns regarding the vulnerabilities in ML frameworks. Such vulnerabilities pose a considerable risk, potentially undermining the integrity and reliability of ML applications in critical areas. Testing these frameworks, however, is notably challenging due to their complex implementations. The intricacy of these systems often masks vulnerabilities, making them difficult to detect with conventional methods. Historically, fuzzing ML frameworks has been met with limited success. The primary challenge in this area has been the effective extraction of input constraints and the generation of valid inputs. Traditional approaches often result in prolonged fuzzing periods, which are not only inefficient but also insufficient in reaching the deeper, more complex execution paths where critical vulnerabilities might lie. In

收稿日期:2023-06-30;在线发布日期:2024-01-15.本课题得到国家科技创新 2030——"新一代人工智能"重大项目(2020AAA0104300)资助.**刘昭**,硕士,主要研究领域为软件安全、AI 安全.E-mail: liuzhao3@360.cn.**邹权臣(通信作者)**,博士,副研究员,主要研究领域为 AI 安全、软件与系统安全.E-mail: zouquanchen@126.com.**于 恬**,硕士,主要研究领域为软件安全.**王 旋**,硕士,主要研究领域为软件安全.**张德岳**,硕士,主要研究领域为 AI 安全、软件安全.**孟国柱**,博士,副研究员,主要研究领域为 AI 安全、软件安全.**陈 恺**,博士,研究员,主要研究领域为 AI 安全、软件安全.

response to these challenges, our paper introduces ConFL (Constraint Fuzzy Lop), a novel, constraint-guided fuzzer designed specifically for ML frameworks. ConFL marks a significant advancement in the field of ML framework testing. Its ability to automatically extract constraints from source codes is a groundbreaking feature. This automation is particularly beneficial as it eliminates the need for prior knowledge of the framework's inner workings, thus democratizing the testing process. The constraint-guided approach of ConFL is instrumental in generating valid inputs that are more likely to pass through the initial layers of verification in ML frameworks. This capability enables ConFL to delve deeper into the operator code's pathways, thus uncovering vulnerabilities that would otherwise remain hidden in traditional testing methods. Moreover, ConFL innovates with a unique grouping technique designed to enhance fuzzing efficiency. This technique organizes the testing process in a more structured manner, allowing for a more thorough and systematic exploration of the framework's vulnerabilities. Our evaluation of ConFL's performance, primarily on the TensorFlow framework, has yielded impressive results. ConFL demonstrates a superior capability in covering more code lines and generating a greater number of valid inputs compared to state-of-the-art (SOTA) fuzzers. This increased efficiency is crucial in the practical application of fuzzing in ML frameworks, as it translates to more robust and secure ML applications. In the realm of known vulnerabilities within the TensorFlow framework, ConFL has shown exceptional prowess. It has successfully detected a larger number of vulnerabilities than existing fuzzers. But perhaps more importantly, ConFL has identified 84 previously unknown vulnerabilities across various versions of TensorFlow. These newly discovered vulnerabilities, which include 3 of critical severity and 13 of high severity, have been significant enough to warrant new CVE (Common Vulnerabilities and Exposures) ids. The versatility of ConFL is further demonstrated by its application to other ML frameworks such as PyTorch and Paddle. In these frameworks, ConFL has already identified 7 vulnerabilities, indicating its potential as a universal tool for ML framework testing. In conclusion, ConFL represents a significant step forward in securing ML frameworks. Its automated, constraint-guided approach not only makes the fuzzing process more efficient but also more effective in uncovering deep-seated vulnerabilities. As ML continues to permeate various sectors, tools like ConFL will be vital in ensuring the security and reliability of ML-driven systems.

Keywords　machine learning framework; constraints extraction; operator testing; fuzzing; vulnerability detection

1　引　言

机器学习技术目前被广泛应用于图像分类[1]、语音识别[2]、自然语言处理[3]等领域，给人们的生活带来极大的便利. 智能服务需要软硬件基础设施的支持，而机器学习(Machine Learning, ML)框架是基础设施的核心. ML 框架如 TensorFlow[4]、PyTorch[5]、Caffe[6]为开发人员提供了功能全面的应用程序接口(Application Programming Interface, API)，用于数据处理、模型训练和部署推理，加速了智能服务的构建. 其中，TensorFlow 是谷歌公司推出的一款端到端开源框架，其下载使用量已经突破亿次[7]，目前是开发者使用最多的框架之一.

尽管 ML 框架很受欢迎，但它们不可避免地存在常见的软件漏洞，如栈溢出、堆溢出和除零等问题. 如 TensorFlow 到目前为止已经有 432 个 CVE[8]漏洞，这些漏洞可能导致敏感信息泄露、任意代码执行等严重后果. 随着智能应用数量的持续增加，与 ML 框架相关的安全风险也会显著增加. 因此，检测 ML 框架中的隐藏的安全漏洞并及时修复，对降低安全风险至关重要.

然而，由于 ML 框架功能的复杂性，发现其中的安全漏洞极具挑战性的. 典型的 ML 框架由两部分组

成:(1)面向开发者,用于快速构建神经网络的前端,这部分主要由 Python 语言实现;(2)执行矩阵计算、模型优化等任务的后端,这部分主要由 C/C++语言实现.算子作为 ML 框架的计算单元,在模型训练与推理时都被使用,因此是测试的主要对象.然而,算子具有多种类型的参数,同时不同参数之间可能存在依赖关系,这增加了生成合法测试数据的难度.常规的模糊测试工具,如 Peach[9]、AFL[10] 和 libFuzzer[11],需要大量的人力依据规范构建测试模板,这使得它们在测试 ML 框架方面受到限制.

最近,一些工作在 ML 框架测试方面取得了进展.DocTer[12] 从 API 文档中提取辅助信息,并使用它们来指导测试输入生成.FreeFuzz [13] 使用插桩来跟踪每个覆盖算子的动态信息,然后利用这些信息进行模糊测试.DeepRel[14] 在 FreeFuzz 的基础上构建,在相似算子之间共用有效测试数据.DocTer、FreeFuzz 和 DeepRel 部分或完全依赖于算子文档,而算子文档和真实运行的代码逻辑是脱离的,可能存在滞后或错误.因此,以上的方法无法覆盖全部算子并且存在生成错误算子参数测试数据的情况.例如,DocTer 仅实现了 33%的有效输入生成率.IvySyn[15] 自动识别算子的 C++代码实现位置,在其中添加模糊测试代码后进行基于变异的测试,具有类型感知的变异.一旦检测到出现内存错误时,IvySyn 会生成漏洞验证程序.然而,当面临复杂的代码逻辑时,IvySyn 并不能有效地生成漏洞验证数据.

针对上述问题,本文提出了一种自动从源代码中提取算子约束的方法,并开发原型工具 ConFL,其工作流程如图 1 所示.本文选择 Python 前端作为测试入口,用于测试后端 C/C++算子代码.ConFL 首先遍历源代码中的所有算子,收集算子名称、调用链和参数名称等信息.接下来,ConFL 使用静态污点分析从源代码中提取约束.依据约束类型的不同,ConFL 共提取以下四种约束:环境类约束、依赖类约束、验证类约束和逻辑类约束.最后,ConFL 使用算子信息和提取到的约束构建两种类型的模糊测试模板:(1)控制模板.用于确定算子的上下文执行环境与控制流;(2)数据模板.用于指定算子参数的维度、类型和数值.在这些约束的指导下,ConFL 能够生成结构和语义全部有效的高质量测试输入,进而测试算子深层代码逻辑中的漏洞.

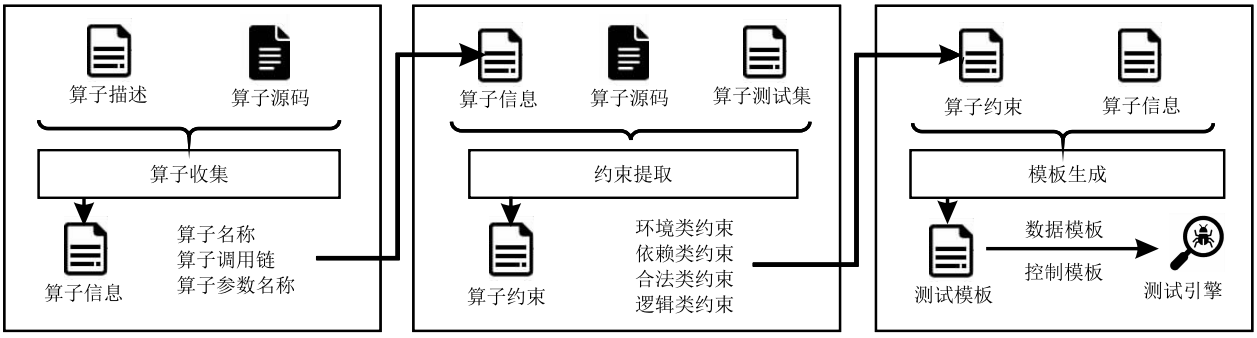

图 1 ConFL 整体工作流程

本文评估了 ConFL 在 TensorFlow 上的有效性.在各个评估维度上,ConFL 均优于 DocTer、FreeFuzz、DeepRel 和 IvySyn.ConFL 取得了更高的算子执行成功率和代码覆盖率,表明它能够生成更有效的输入,执行更多的测试用例,而不会出现参数校验错误或异常,进而可以探索更广泛的代码路径.此外,ConFL 拥有更强的漏洞检测能力,在 ML 框架中可以发现更多的漏洞.实验结果显示,ConFL 在不同版本的 TensorFlow 中发现了 84 个以前未知的漏洞,其中包括 3 个严重漏洞和 13 个高危漏洞,所有这些漏洞都被分配了新的 CVE 编号.本文还用 ConFL 扩展测试了 PyTorch 和 Paddle,迄今已发现 7 个漏洞.本文的主要贡献包括:

(1)提出了一种高效的算子收集和约束提取方法:ConFL 有效地从机器学习框架中收集算子,提取环境类约束、依赖类约束、验证类约束和逻辑类约束,构建全面的约束树.

(2)提出了一种约束制导的测试数据生成方法:利用提取的约束,ConFL 以更有指导性和高效的方式生成测试输入,与随机生成方法或最先进的模糊工具相比,取得了更高的执行成功率和代码覆盖率.

(3)实现了更有效的 ML 框架漏洞检测:通过高质量的测试输入,ConFL 可以在相同版本的 ML 框架中发现比其他工具更多的已知漏洞;同时 ConFL 还发现了 84 个未知漏洞,全部漏洞均已被厂商确认和修复.

2　相关工作

2.1　机器学习框架描述

ML 框架作为机器学习领域中的关键基础设施，通过提供预构建的库和工具，简化了创建、训练和部署机器学习模型的流程. 框架整体由四层组成，即运行时层、计算层、网络层和设备层. 运行时层接收、构建和编排计算图，计算层实现不同计算操作，网络层实现不同组件之间的通信，设备层支持各种硬件设备，如 CPU、GPU、TPU 等.

算子是 ML 框架的核心组件，负责训练推理过程中对张量或数据执行数学计算，如负责执行卷积的 Conv3D 算子，执行池化的 MaxPool 算子. 算子内核通常由 C/C++语言开发实现，同时为开发者提供 Python、Java 等语言接口，方便开发人员通过前端算子接口来构建神经网络. 在计算过程中，算子通过处理用户传入的不同参数执行不同的逻辑功能. 以 TensorFlow 中提供的 LoadAndRemapMatrix(LARM)[16] 和 BoostedTreesCalculateBestFeatureSplit(BTCBFS)[17] 两个算子为例，在分析源代码中的描述文件后，发现算子 LARM 需要 9 个参数来实现从 checkpoint 文件中加载数据的功能，其中参数 *old_tensor_name* 指定了需要加载数据条目的名称；算子 BTCBFS 需要 9 个参数来计算每个特征的增益.

2.2　应用程序接口测试

先前的工作专注于对操作系统调用和库函数接口进行测试，包括云服务 API[18]，操作系统内核接口[19-21]和本地库接口[22]. 例如，NTFuzz[23]是一个类型感知的 Windows 内核模糊测试框架，可以自动推断 Windows 内核系统调用的类型信息，通过类型信息生成数据测试. APICRAFT [24]利用静态和动态信息收集函数的控制和数据依赖关系，采用多目标遗传算法将收集的依赖关系组合起来，构建高质量的模糊驱动程序.

虽然系统接口和算子之间存在相似之处，但现有的模糊测试工具不适用于 ML 框架中的算子. 首先，算子参数是特定的数据类型，例如张量，需要专用的测试数据生成方法生成. 其次，ML 框架算子存在单个参数的合法性约束和多个参数间的相互依赖关系，通用的系统接口模糊测试工具无法有效处理此类约束问题.

2.3　机器学习框架测试

近些年来，随着机器学习框架的广泛使用，其安全问题也得到广泛关注，研究人员针对 ML 框架的安全性测试也取得了一系列的进展.

除了上文提到的 DocTer、FreeFuzz、DeepRel 和 IvySyn，Xiao 等人[25]、Tan 等人[26]研究了 ML 框架第三方依赖库的安全问题，但没有对 ML 框架主体的安全性进行深入研究. Xie 等人[27]提出了一种通用的覆盖率导向的 ML 框架测试工具 DeepHunter，该工具利用多个可扩展的覆盖标准作为反馈来指导测试数据生成. 然而，DeepHunter 的数据生成方法属于随机变异，变异过程中缺乏合法性的约束，导致生成的测试样本合法性降低，检测效率难以达到预期.

本文的方法不仅专注于实现更高的代码覆盖率，而且确保生成的测试输入符合目标 ML 框架约束. 通过从算子的源代码中自动提取约束，ConFL 可以生成更全面更准确的测试输入，从而提高了模糊测试过程中漏洞挖掘的效率.

3　方法设计

3.1　问题分析

对算子进行安全性测试面临着许多挑战. 首先，算子具备不同的功能，如文件管理或矩阵计算，不同功能的算子需要不同的测试方式. 其次，算子在不同的架构(如 CPU、GPU)上的代码实现有所不同，这进一步增加了对所有算子进行测试的复杂程度. 最后，算子通常需要处理高维空间数据，如图像或文本序列，这使得生成有效的合法化测试输入数据变得尤为困难.

算子的参数具有不同的类型，主要的参数类型称为张量，它是一个具有统一类型(如 int、float 或 string)的多维数组. 如果人工编写用于模糊测试的测试模板，则需要收集算子参数的数据类型，包括 float、int、char、string 和 bool. 除此之外，考虑到 ML 框架的计算特性，还需要包括列表类型和张量类型. 图 2 给出了如何使用随机生成的测试数据生成 BTCBFS 算子测试模板的示例.

然而，在运行 BTCBFS 测试脚本时，遇到了类型错误“InvalidArgumentError: hessian dim should be <0, got -1.”.

在测试算子 BTCBFS 时，通过分析文档中算子参数的范围，缩小了随机数据生成的范围. 例如，参数 *stats_summary* 是四维的张量，*logits_dimension*

```
tensorflow.raw_ops.BoostedTreesCalculateBestFeatureSplit(
    node_id_range=[1,7],
    stats_summary=[[[[2.0]], [[3.]], [[3.]]]],
    l1=[0.0],
    l2=[0.0],
    tree_complexity=[1.0],
    min_node_weight=[0.7],
    logits_dimension = 2,
    split_type = 'equality'
)
```

图 2 测试 BTCBFS 算子的案例

是大于 0 的整数. 虽然可以生成相对规范化的测试数据,但由于参数间数据关系的不合法性,导致算子仍无法执行. 一旦测试输入无效,计算过程将在 Python 前端终止,使得测试算子的深层逻辑代码变得困难. 同时,错误消息中的"hessian"一词不在算子参数名称列表中,这使得修正测试数据更加困难.

3.2 方法设计

本文研究算子安全性检测方法,重点研究基于约束的算子参数数据合法性生成技术. 由于从算子文档中提取约束是不完整的,因此本文选择从源代码中自动提取算子约束. 本文的目标是利用约束生成合法的测试输入,以通过 C++后端的校验,检测算子深层代码漏洞.

为了实现这一目标,本文开发了原型工具 ConFL,它由三个模块组成,如图 1 所示.

算子收集. ConFL 旨在测试算子,因此第一步是收集算子信息. 该模块自动遍历 ML 框架的所有算子并收集信息,包括算子名称、算子调用链和参数名称. 基于算子收集模块的数据可以构建测试模板,用于模糊测试. 该模块将在第 4.1 节中进一步介绍.

约束提取. 此模块用于从源代码、文档中提取算子的四类约束,分别是环境类约束、依赖类约束、验证类约束和逻辑类约束. 环境和依赖类约束用于限制算子的执行上下文,以确保它们可以访问执行所需的必要资源. 验证类和逻辑类约束用于生成有效和多样化的测试输入,以使算子的深层代码能够执行. 该模块将在第 4.2 节中详细描述.

模板生成. 在获取算子信息和约束信息后,ConFL 能够依据以上数据生成测试模板. 测试模板分为两种类型:数据模板和控制模板. 数据模板指定参数的形状、类型和值,而控制模板指定算子的执行环境. 通过使用算子信息,ConFL 为模板构建骨架,然后根据约束信息为不同的参数构建依赖关系. 该模块将在第 4.3 节中详细描述.

4 关键技术实现

第 4 节以 TensorFlow 框架为例,详细介绍三个模块的具体实现,并在第 4.4 节介绍对其他框架的适配方法.

4.1 算子收集

现有的测试工具如 DocTer、FreeFuzz 和 DeepRel 部分依赖或完全依赖从算子文档中收集算子信息,而算子文档和真实运行的代码逻辑是脱离的,其中可能存在算子描述缺失或描述错误的问题. 同时,开发者可以调用文档中未描述的算子接口实现功能. 因此,ConFL 选择更为准确的框架源码作为主要对象,收集算子信息.

ML 框架如 TensorFlow 按照开发语言可以分为两部分:供开发人员使用的前端,这部分包含各种编程语言实现的接口,如 Python、Java. 而后端则负责执行计算,为了提高计算效率,通常使用 C/C++实现. 本文提出的技术选择使用通过 Python 前端作为入口,测试 C/C++后端代码,原因如下:

(1) 大多数开发人员使用 Python 前端接口构建神经网络进行模型训练和推理,与框架的使用场景一致.

(2) Python 提供了出色的语言特性. 与从 C++后端测试的 IvySyn 不同,ConFL 选择 Python 前端作为算子的入口. Python 前端具有丰富的类型,如 int、float、tensor 等,可以辅助指导生成有效的测试参数.

(3) 算子收集过程自动化. 为每个算子编写 C/C++测试代码耗费人力,需要对不同的数据类型进行细致的描述,而使用 Python 接口构建测试代码则更容易实现自动化. 本文提出的技术使用 Python 语言的反射机制自动化地生成算子测试模板,避免大量人力投入,可显著提升测试效率.

ML 框架使用 Pybind11[28]、SWIG[29] 等方法将 Python 代码与 C/C++代码连接起来,在框架执行期间,将 C/C++部分作为模块的形式加载到 Python 运行时中. 通过选择 Python 前端作为入口点,ConFL 利用 Python 的反射机制自动遍历函数、类和模块,获取算子包目录. 通过算子名称和包目录,ConFL 分析算子的签名并提取参数名称,通过融合

以上收集的信息，生成算子的测试模板.

算法 1 中的 getMods 与 getOps 函数详细展示了 ConFL 收集算子的过程. 在收集算子的过程中，ConFL 构建了模块-算子的树形结构，其中每个叶子节点代表算子，根节点代表 ML 顶层框架模块，而从根节点到叶子节点的路径代表算子的调用路径.

算法 1.　算子收集.

输入：一个特定的 ML 库 *mlLib*

输出：一个模块-算子树 *tree*

```
1.  tree.init(mlLib)
2.  FUNCTION getMods(parentMod)
3.   modules ← Queue()
4.   modules.put(mlLib)
5.   WHILE modules not empty DO
6.    parentMod←modules.get()
7.    mods←getModMembers(parentMod)
8.    parentModPath←getModInfo(parentMod)
9.    FOREACH mod IN mods DO
10.    IF mod is duplicated THEN
      //存在相同模块时，保留首次出现的调用路径
11.     CONTINUE
12.    END
13.    modules.put(mod)
14.    tree.addNode(mod, parentMod)
15.   END
16.  END
17. RETURN modules
18. FUNCTION getOps(modules):
19.  WHILE modules not empty DO
20.   parentMod←modules.get()
21.   ops←getOpMembers(parentMod)
22.   parentModPath←getModInfo(parentMod)
23.   FOREACH op IN ops DO
24.    IF op is duplicated THEN
      //存在相同算子时，保留最短的调用路径
25.     IF len(path(op)) >
       len(path(parentModPath+op.name)) THEN
26.      tree.moveNode(op, parentMod)
27.     END
28.    END
29.    tree.addNode(op, parentMod)
30.   END
31.  END
32. RETURN tree
```

getMods 函数采用广度搜索算法遍历所有模块，构建模块调用树. 从算法的第 5 行开始，迭代遍历所有可用的模块. 首先从模块队列中获取需要解析的目标模块(第 6 行)，然后在第 9～17 行将模块添加到树中，用于下一步获取每个算子的完整调用路径. 由于不同模块可能存在调用同一子模块的情况，当检测到重复出现模块时，保留首次出现的模块调用路径(如第 10～12 行所示). 最后，返回包含 ML 库中所有模块的队列.

遍历收集完模块后，ConFL 通过 getOps 从模块中收集算子. 算法中第 19 行开始遍历 getMods 收集到的模块，从每个模块中获取算子(第 21 行)，与 getMods 类似，第 24～28 行考虑了出现重复算子的情况，ConFL 选择保留调用路径最短的算子. 最终，在 29 行将算子拼接到模块-算子树中，用于后续算子测试模板生成.

ConFL 通过 Python 前端接口自动分析 C/C++后端的安全性，在收集算子的过程中，使用了以下三种方式进行优化.

(1) 相同模块或者算子被加载到内存后，可能处于内存的同一地址. 通过判断算子在内存中的地址，判断是否为同一个算子. 如果不同调用路径调用同一算子，ConFL 选择保存较短的调用路径.

(2) 对在仅使用 Python 代码实现功能的算子，ConFL 将不保存其调用路径. 例如，tensorflow.experimental.dtensor.job_name()不执行 C/C++后端的代码，因此不会进行测试.

(3) 在 Python 前端部分，不同的算子接口可能对应相同的 C++函数. 为了避免后续重复检测的问题，ConFL 通过算子参数和算子调用路径来进一步去重. 如图 3 所示的三个算子 tensorflow.reshape、tensorflow.raw_ops.Reshape 以及 tensorflow._compat.readers.array_ops.gen_array_ops.reshape 共享相同的参数，通过判断三个算子在 C++代码中的调用路径，发现它们具有相同的调用链. 因此，ConFL 只选择 tensorflow.raw_ops.Reshape 生成测试模板.

```
OP: tensorflow.reshape(tensor=[1,2], shape=[1,2])
Path: Dispatch -> TFE_Py_FastPathExecute -> TFE_Py_Execute
OP: tensorflow.raw_ops.Reshape(tensor=[1,2], shape=[1,2])
Path: TFE_Py_FastPathExecute -> TFE_Py_Execute
OP: tensorflow._compat.readers.array_ops.gen_array_ops.reshape
    (tensor=[1,2], shape=[1,2])
Path: TFE_Py_FastPathExecute -> TFE_Py_Execute
```

图 3　算子调用链

在生成算子测试模板的过程中，不同的 ML 框架前端可能具有不同的实现类型. 以 Python 前端为例，算子的实现以函数或类的形式；对于函数，调用语句

将自动构造.对于类,首先生成类的实例,然后生成调用代码.

在 TensorFlow 2.8 版本中,ConFL 能够收集 9689 个算子接口,并且自动为所有算子接口生成测试模板,此过程无需人工干预.对比官方文档后,发现 ConFL 收集到的算子包含所有官方提供的算子,此外,结果中还包含官方文档中未提供的算子数据.

4.2 算子约束提取

算子的运行时执行路径取决于输入数据和执行环境.ConFL 将算子成功执行所需的条件定义为约束,通过提取算子实现代码中的约束条件,依据其设置执行环境并生成输入数据以确保算子代码功能的全面覆盖.我们按照约束类型将其分为四种类型,包括环境类约束、依赖类约束、验证类约束和逻辑类约束.基于这四类约束,ConFL 能够实现更高的代码覆盖率,并能够发现深层次执行路径中隐藏的漏洞(在第 5.6 节中详细描述).

4.2.1 环境类约束

ML 框架根据不同的硬件环境或者优化选项具备多种可选择的运行环境.表 1 所示为 TensorFlow 框架中所有环境类约束.

表 1 环境类约束

约束类型	类型	约束
执行模式	Eager execution	-
	Graph Execution	@tf.function
	XLA	@tf.function(jit_compiler=True)
架构模式	CPU	-
	GPU	tf.device('/device:GPU:2')
	TPU	TPUClusterResolver(tpu='')

以 TensorFlow 为例,其执行模式分为Eager Execution 和 Graph Execution[30].Eager execution 是一种命令式编程模式,在这种模式下,TensorFlow 的操作在 Python 调用时立即执行,这种模式更直观、更灵活,可以更方便地进行调试.Graph Execution,也称为静态计算图,是 TensorFlow 中传统的执行方式.在这种模式下,用户首先定义一个表示模型或算法的计算图,然后 TensorFlow 使用会话的方式优化和执行该图.Graph Execution 通过并行、分布式执行和高效内存分配等各种优化提供性能优势.自 TensorFlow 2.0 以来,Eager Execution 被设置为默认的模式,但用户仍然可以通过 tf.function 装饰器使用图模式,将用户的 Python 代码转换为静态图.这使用户可以利用图优化的高性能,同时保持动态图的灵活性.此外,TensorFlow 使用名为 Accelerated Linear Algebra(XLA)[31]的领域特定编译器来加速线性代数计算.

4.2.2 依赖类约束

依赖类约束是指算子在实际执行之前必须满足条件的参数约束.如果参数的类型和数据不符合算子的要求,则算子无法成功执行.依赖类约束具体细分为资源类依赖约束和操作类依赖约束.

(1)资源依赖约束

在机器学习框架算子的计算过程中,需要各种类型的参数.在 TensorFlow 中,除了 int 和 float 这样的简单类型之外,还有 resource 和 variant 等特殊类型.resource 类型是可变的、动态分配资源的句柄,而 variant 类型表示任意类型的数据.根据运算符参数的复杂程度,本文将类型分为基本类型和复合类型两类.表 2 展示了 TensorFlow 中的所有类型.

表 2 TensorFlow 中的类型

基本类型	复合类型	
	基本 Tensor	资源型 Tensor
bool	DT_INT8/16/32/64	DT_RESOURCE
int	DT_UINT8/16/32/64	DT_VARIANT
float	DT_BOOL	CODE
string	DT_COMPLEX64/128	FILE
char	DT_QINT8/16/32	
	DT_QUINT8/16	
	DT_HALF	
	DT_FLOAT	
	DT_DOUBLE	
	DT_BFLOAT16	
	DT_STRING	

在 TensorFlow 和 Paddle 这样的机器学习框架,算子描述文件通常用于动态生成代码或者跟踪算子代码的更改.其中 TensorFlow 的算子描述文件为 ops.pbtxt,包含算子名称、参数名称和类型,通过分析其中的参数信息来获取对应的类型.以算子 LARM 为例,参数 ckpt_path 是 DT_STRING 类型,num_rows 是 DT_FLOAT 类型,同时能够获取算子的返回值为 DT_FLOAT 类型.在解析完各参数的属性后,生成如图 4 的类型描述.

```
'ckpt_path': ['DT_STRING'],
'old_tensor_name': ['DT_STRING'],
'row_remapping': ['DT_INT64'],
'col_remapping': ['DT_INT64'],
'initializing_values': ['DT_FLOAT'],
'num_rows': ['int'],
'num_cols': ['int'],
'max_rows_in_memory': ['int']
```

图 4 算子 LARM 的参数类型

除此之外，不同的算子之间还存在依赖关系，即一个算子的输出被用作另一个算子的输入. 然而这种参数构造在文档或源代码中并没有反映出来. 本文将这种约束称为资源依赖约束. 通过分析算子类型，并保存成功执行的算子的输出类型，将资源依赖约束抽象出来用以构造正确的参数.

算子的执行可能依赖于不同类型的文件，手动构造文件数据是一项耗时的任务. 例如，LARM 算子在执行过程中需要加载 ckpt 格式的模型文件. 由于输入参数表示模型文件的存储路径，其数据类型为字符串，如果字符串中保存的文件路径发生变化，算子在执行过程中将无法读取路径指向的模型文件数据，导致执行失败. 为了解决这个问题，本文提出了利用测试用例自动提取相关文件约束的方法. TensorFlow 包含大量的测试用例，在测试特定的算子时，测试用例将预先部署算子执行所需要的资源，例如生成指定的文件. LARM 测试的代码存储在 checkpoint_ops_test. py 中，其中包含如图 5 代码.

```
class LoadAndRemapMatrixTest(test.TestCase):
 def setUp(self):
  matrix = variable_scope.get_variable(
   'matrix',
   dtype=dtypes.float32,)
 save = saver.Saver([matrix])
 save.save(...)
```

图 5　算子 LARM 的测试用例

通过对 LARM 算子测试代码进行插桩并监控执行过程中的参数数值变化，可以提取在执行测试用例时生成的相应文件.

(2) 操作依赖约束

根据分析，大多数算子具有很少的调用依赖性，可以进行单独测试. 但是某些参数可能是特殊类型，需要其他算子生成的结果作为它们的输入. 通过关键字匹配识别如 pop、push 和 close 这样的单词，提取算子实体，并将具有相同实体的算子以一个组为单位进行测试. 例如，与 Stack 相关的算子，有 StackPop、StackPush 和 StackClose，都共享 Stack 主体，并通过内置测试序列构造相关数据进行测试. 而且这些操作同一主体的算子存在着顺序上的约束，例如，Push 操作首先需要初始化 Stack，而 Pop 操作需要初始化的堆栈作为参数以及堆栈中的数据，需要执行 Push 操作. 将确保算子顺利执行的前置操作称为操作依赖约束.

算子名称通常从语义上会描述其功能，例如 StackClose、StackPush 和 StackPop. 通过进行词性分析，确定算子名称中的操作和实体，并将作用于同一实体的不同算子视为一组. 在算子执行顺序方面，如果测试用例检测到 Hook 方法通过特定顺序调用集合中的算子，则会保存相关序列；如果测试用例中不存在相关测试，则通过随机执行确定操作依赖关系.

在词性确定过程中，由于单词的位置影响其词性判断，因此在分词后将序列循环左移，并保存识别出的所有动词. 最后，依据筛选出来的所有动词列表，删除初始算子名称中的动词，对剩余的单词进行聚类，判断相同的实体，结果如图 6 所示.

```
LookupTableFind ['Find']
LookupTableRemove ['Remove']
ReaderReadUpTo ['Read']
ReaderRestoreState ['Restore']
Stack ['Stack']
StackClose ['Stack', 'Close']
StackPush ['Push']
```

图 6　算子的名称和操作

4.2.3　验证类约束

环境类约束和依赖类约束主要用于设置算子的执行环境，以便算子具有执行时所需的资源，但算子的执行条件也与输入参数的约束相关. 例如，在 BTCBFS 中，获取每个参数的大小、类型和数值约束. 除此之外，还发现参数之间也存在相互约束. 例如，BTCBFS 算子中的 *stats_summary* 参数的第三个值需要大于 *logits_dimension*.

算子中的验证类约束是指使算子正确执行并产生预期的输出的参数必须满足的条件，这些约束在确保数据完整性、维护 API 稳定性以及防止算子执行期间的产生错误或异常方面起着至关重要的作用. 如果输入参数不满足语义规则，则测试用例通常会在语义检查中失败，并在算子的浅层代码中出现故障. 因此，通用生成式模糊测试生成的输入仅有少量达到算子执行阶段，而深层次的漏洞通常隐藏在其中，导致大部分的算子代码并未被触及.

本文提出了一种算子约束提取技术，通过分析算子的源代码，定位语义检查语句，提取与参数进行比较的特定值，并最终作为验证类约束. 验证类约束可以分为类型约束和数值约束，它们作为后续测试阶段生成有效参数的指南. 该过程包括三个主要步骤：首先，使用 clang 将源代码编译为 LLVM[32] 的

中间表示(IR);其次,指定污点源、传播方式和污点汇聚点;最后,在污点汇聚点处提取约束.

在 Tensorflow 中,不同算子是通过扩展 OpKernel 并重写 Compute 方法来实现的.算子参数分为 input 和 attr 两种类型,input 是具有可变属性的张量,而 attr 则不可变.如图 7 所示,算子 BTCBFS 在构造函数中使用 context->GetAttr(图 7 中的点框)接收 attr 参数,在 Compute 方法中使用 context->input(图 7 中的实心框)接收 input 参数.本文通过解析对应的结构,设置污点分析的 source 位置.如 GetAttr 函数,其第一个参数是算子在 Python 前端接口中对应的参数名称,第二个参数表示存储该参数所用的变量名称.通过对 TensorFlow 框架源码的解析,我们确定了七种 source 点形式(如图 8 所示),具体分为三类:获取输入(*input*、*input_list*)、获取可变输入(*mutable_input*、*mutable_input_list*)以及获取属性(*GetAttr*).

```
explicit BoostedTreesCalculateBestFeatureSplitOp(…) {
  OP_REQUIRES_OK(context, context->GetAttr("logits_dimension", &logits_dim_));
}

void Compute(…){
  OP_REQUIRES_OK(context, context->input("stats_summary ", &stats_summary_t));①

  const int32_t hessian_dim  = stats_summary_t->dim_size(3) - logits_dim_;②

  OP_REQUIRES(context, hessian_dim > 0, errors::InvalidArgument("……"));③

  OP_REQUIRES_OK(context, context->input("l1", &l1_t));❶

  const auto l1 = l1_t->scalar<float>()();❷

  if (logits_dim_ > 1) {
    OP_REQUIRES(context, l1 == 0, errors::InvalidArgument("……"));❸
  }
}
```

图 7 算子 BTCBFS 参数约束提取过程

```
context->input(INDEX)
context->input("VARNAME", &VAR)
context->input_list("VARNAME", &VAR));
context->mutable_input(INDEX, _);
context->mutable_input(VARNAME, &VAR, _));
context->mutable_input_list("VARNAME", &VAR));
context->GetAttr("VARNAME", &VAR));
```

图 8 污点源

由于算子主要使用输入参数进行计算,因此将获取参数的函数返回结果指定为污点源.在使用 LLVM 分析 IR 时,如 load、store 和 getelementptr 指令,是污点传播分析的主要目标.当指令的操作数中存在污点变量时,该指令的返回变量被标记为污点.图 7 中存在两条污点传播:(1)在①处获取参数 *stats_summary_t*,之后在②计算并将结果保存在 *hessian_dim* 中,最后在③判断是否大于零;(2)在❶处获取 *l1_t* 参数,之后在❷进行类型转换为 *l1*,最后在❸判断是否等于 0.

识别污点汇聚点是污点分析的关键,经过分析,我们发现大多数 ML 框架在源代码中使用宏来评估算子参数的合法性.如算子 BTCBFS 使用 OP_REQUIRES 宏来确定 *stats_summary* 和 *logits_dimension* 参数之间的关系,当参数数据无法满足约束条件时,则将报告错误并终止计算.

OP_REQUIRES 宏的第二个参数是一个表达式,如图 7 中的③处,第三个参数是一个错误输出语句.当表达式判断为 false 时,程序会调用输出函数打印错误语句,并退出流程.在使用 LLVM 分析 IR 时,该结构表示为条件跳转指令,且跳转目标的基本块包含一个错误输出函数或检查函数.因此,ConFL 基于以下三个特征来确定污点汇聚点:

(1)污点汇聚点是一个条件跳转指令;

(2)跳转条件包含污点变量;

(3)跳转目标基本块中存在错误输出函数或检查函数.

在满足上述特征的污点汇聚点处,提取跳转条件作为算子约束.最后,将提取的约束简化和修订为可读形式.如图 7 所示,通过数据流分析,在③处判断 *stats_summary_t* 的维度数据与 *logits_dim_* 的关系,进而解析为约束:$stats\_summary\_t.shape(3) > logits\_dim\_$.

4.2.4 逻辑类约束

本文将算子源码中通过解析分支语句提取的约束称为逻辑类约束,如图 7 中的 IF 语句,判断输入参数 *logits_dim_* 是否大于 1,如果判断为真,则对 *l1* 进行合法性校验.

ConFL 通过构建约束树来增加对逻辑类约束的支持,具体流程如算法 2 所示.首先判断 if 语句中是否存在污点,如果存在,则将约束添加到约束树中,然后分析 if 语句块内是否存在合法性验证语句.基于构造的约束树,ConFL 选择其中一个分支来生成模糊测试模板.

算法 2. 约束提取.

输入:污点 *taint*,语句 *stmt*

输出:参数的约束 *C*

1. FUNCTION *HandleFor*(*taint*, *stmt*):
   //如果 for 的循环体和循环条件中有污点,将循环条件加入约束 *C* 中
2. IF *taint* in *stmt.body.sink* THEN
3. *C.add*(*convertToCons*(*stmt.body.sink*))
4. END

```
5. FUNCTION HandleIf(taint, stmt)
   //如果 if 的 body 中有污点，则将跳转条件和 sink 加入到约束 C 中
6.  IF stmt.body.sink THEN
7.    C.add(convertToCons(stmt.body.sink))
8.    C.add(convertToCons(stmt.cond))
9.  END
```

然而，提取的约束是依据后端 C/C++代码生成的中间表示，即 LLVM IR. ConFL 直接调用 Python 前端接口进行测试，表示约束的 IR 不能直接用于生成 Python 前端参数的数据，因此需要将 IR 提升为 Python 前端形式，使约束在模糊测试执行期间被正确使用. ConFL 通过解析 Python 与 C/C++间的数据映射，将 IR 约束中的参数对应到 Python 前端中，使提取后的约束能够辅助测试数据的生成.

以 LARM 算子为例，ConFL 生成的约束条件如图 9 所示.

```
len(ckpt_path_t) == 1
len(row_remapping.shape()) == 1
len(row_remapping) == num_rows
len(col_remapping) == num_cols
```

图 9　LARM 算子的约束信息

上述约束既包括参数的形状要求，也包括参数之间的依赖关系. 例如，*row_remapping* 必须是一维的，其长度应等于 *num_rows* 的值.

4.3　测试模板生成

基于算子信息和算子约束条件，ConFL 生成算子模糊测试模板. 这些模板不包含算子参数的具体模糊数据，而是构建一个测试框架. 在实际测试过程中，ConFL 根据模板选择相应的测试数据. 根据功能，算子模板分为控制模板和数据模板.

控制模板主要设置算子的可执行环境、参数位置、参数类型和依赖信息. ConFL 首先对不存在操作类约束的算子生成控制模板，因为此类算子不依赖于其他算子，进而可以直接测试. 当算子存在确定的操作类约束序列时，ConFL 将自动构建序列控制模板. 如测试 StackPop 算子时，需要依赖 Stack 算子进行 stack 初始化操作，此时需要在 StackPop 前引入 Stack 模板. 当算子存在操作类约束序列，但排序不确定时，ConFL 将在边际确定的两个算子之间随机插入任意调用序列. 如 Stack 相关的算子 StackPop、StackPush 和 StackClose，当测试算子 StackPush 时，其参数依赖于 Stack 算子结果，此时序列开头节点为 Stack 算子，结尾为 StackPush 算子，中间可以插入任意数目的 StackPop、StackPush 操作.

此外，在生成控制模板时，ConFL 提出了一种数据复用的分组测试，将相似的算子设置为同组进行调度测试. 在 ML 框架中，不同的 Python 接口可能共同使用框架中相同的 C/C++后端，例如，Python 中的 ArgMax 和 ArgMin 都对应于 C++中的 ArgOp 类. ArgOp 类通过判断传入的参数来使用两种相似的代码处理逻辑来处理 ArgMax 与 ArgMin. 不同 ML 框架中的算子注册机制规定了添加算子的方式，例如 PaddlePaddle 的 REGISTER_OPERATOR、MindSpore 的 REGISTER_PRIMITIVE_EVAL_IMPL 和 TensorFlow 的 REGISTER_KERNEL_BUILDER. 通过这样的注册方法，ConFL 建立了不同语言中算子之间的对应关系，实现了参数数据的重用. 如前面所述，ArgMax 和 ArgMin 共享相同的参数：input、dimension 和 output_type. 通过将同一后端实现的算子分为一组，可以实现参数的迁移，如将 ArgMax 的参数数据应用于 ArgMin，进而避免了重复生成 ArgMin 的数据，提升了整体的测试效率. 同时，通过第 4.2.2 节中介绍的词性分析，确定算子名称中的操作和实体，设置待测算子的上下文依赖.

图 10 为 LARM 算子的控制模板，其中包含收集到的算子名称以及参数信息，并且设置了算子的依赖环境、可执行环境、参数位置. 控制模板用于控制算子的执行上下文，与数据模板结合共同测试算子安全性.

```
func = {{ op.name }}
{{ op.deps }}
para = {
    {% for arg in op.args %}
        '{{ arg.name }}': {{ arg.type }}
    {% endfor %}
}
with tf.device({{ op.env }}):
    func(**para):
```

图 10　LARM 算子的控制模板

数据模板是一种用于生成测试数据的模板，它包含了算子的名称和参数列表. 根据第 4.2 节提取的约束条件，数据模板用占位符表示算子参数中具

有特定要求的可变部分.在生成测试数据时,数据模板中的占位符会被具体的数据替换,从而生成多样化的测试数据.

ConFL 首先根据算子参数依赖进行拓扑排序,处理无父节点的参数,其约束信息用于限制单一参数数据生成,如 LARM 算子中的 *num_cols* 参数需要为 int 类型,对应生成的单参数数据模板为"'*num_cols*': DI".之后 ConFL 遍历拓扑排序中的参数依赖,处理参数之间的关系,如 LARM 算子中的 *col_remapping* 参数,该参数存在约束"*len*(*col_remapping*)==*num_cols*"(图 9),要求 *col_remapping* 中元素个数等同于 *num_cols*.由于单参数 *num_cols* 的数据模板已经生成,因此可以直接引用并生成数据模板"'*col_remapping*': [DI] * *num_cols*",其中 DI 是符合参数 *col_remapping* 类型的数据占位符,在具体测试时使用整数来替代.该模板表示 *col_remapping* 的长度必须等于 *num_cols*,并且由 DI 类型数据填充 *col_remapping*.针对 LARM 算子生成的数据模板如图 11 所示.

```
'ckpt_path':STR|PATH,
'old_tensor_name': STR,
'row_remapping': LST,
'col_remapping': [DI]*num_cols,
'initializing_values': LST,
'num_rows': DI,
'num_cols': DI,
'max_rows_in_memory':DI,
```

图 11 LARM 算子的数据模板

通过保存表示数据的形状和类型符号,将生成的数据进行分类,以防止创建过多的重复参数.由于 ML 框架中存在众多的计算步骤,ConFL 在生成具体值时从特殊值集合中选择值(例如边界值、零、大整数).这种方法减少了生成参数的范围,并防止不同的数据执行相同的路径,同时保留了漏洞检测能力.然后,ConFL 验证参数是否满足约束条件.如果不满足,将采取有针对性的修改措施,对形状或数值进行修改,同时保留原始数据特征,与重新生成数据相比更为高效.

通过应用控制与数据模板,可以生成以下测试数据(如图 12 所示).

4.4 其他 ML 框架适配

ConFL 的设计只需进行少量的修改就可以适配其他 ML 框架,具有通用性.

```
para = {
        'ckpt_path':'bundle_checkpoint',
        'old_tensor_name':'some_scope/matrix',
        'row_remapping': [1],
        'col_remapping': [2147483649] ?1073741824,
        'initializing_values': [],
        'num_rows': 1,
        'num_cols': 1073741824,
        'max_rows_in_memory':?1,
}
```

图 12 基于约束生成的 LARM 算子的参数

在算子收集功能方面,由于 Python 的反射机制适用于所有以 Python 作为前端的框架,因此 ConFL 可以自动收集算子信息而无需特殊适配.

在约束提取方面,环境类约束来自于专家经验,通常采用阅读文档的方式提取,如针对 PyTorch 可以提取环境类约束"module.to(torch.device('cuda'))".依赖约束同样可以通过判断文件操作或者分析算子名称来提取.在验证类约束方面,如 PyTorch 和 PaddlePaddle 在源代码中同样使用宏进行参数验证,其中 PyTorch 使用 TORCH_CHECK 进行参数验证,Paddle 使用 PADDLE_ENFORCE_EQ 等进行参数验证,通过解析不同宏的代码,从中提取算子参数约束.由于其他 ML 框架中同样使用 if、for 等分支循环结构,因此 ConFL 同样支持逻辑类约束的提取.

在测试模板生成方面,ConFL 通过以上提取到的信息构建控制类模板,不同 ML 框架的控制类模板基本相似.在构建数据类模板时需要针对不同的 ML 框架设置不同类型数据的生成方式.最终复用 ConFL 的测试引擎即可完成对其他 ML 框架的适配.

5 实验评估

5.1 实验设置

算子收集部分由 1078 行 Python 代码实现解析源代码的功能.在获取接口时,修改 Python 解释器——CPython,以获取函数调用链并确定接口是否调用了后端 C 函数.之所以选择 CPython 而不是 pybind11 来确定是否调用 C 函数,是因为一些接口使用 SWIG,并且不同的 Python 函数名称可以传递给同一个 C 语言接口,例如函数 TFE_Py_FastPathExecute.

在约束提取部分,使用 523 行 Python 代码提取环境类约束和依赖类约束.为了提取验证类约束和

逻辑类约束，使用 1k 行 C++代码基于 LLVM 实现了路径不敏感的污点分析. 此外，使用了 2k 行 Python 代码实现了算子测试模板生成和算子测试输入生成.

本节使用了第 4 节介绍的方法对 TensorFlow 2.8 进行评估，主要关注以下四个方面：

(1) 算子收集能力评估：评估 ConFL 收集到的算子数量与其它工具相比的优势.

(2) 算子约束提取能力评估：评估 ConFL 提取算子约束的种类和数量.

(3) 算子测试数据有效性评估：评估 ConFL 基于算子约束生成的测试数据与随机生成以及 SOTA 工具生成的测试数据在代码覆盖率指标上的优势.

(4) 漏洞检测能力评估：评估 ConFL 在实际框架中的漏洞发现能力，并分析其相对于 SOTA 工具的优势.

用于运行实验的计算机配备有 Intel Xeon E5-2630 2.20 GHz CPU、Tesla P4 GPU、128 GB RAM、Ubuntu 20.04 LTS 和 Python3.8.

5.2　算子收集能力评估

与从文档中收集算子信息不同，ConFL 通过分析 ML 框架的源代码来收集算子，并且可以直接调用收集到的算子进行测试. 当适配不同版本的 ML 框架时，ConFL 会自动提取该版本的算子，而无需重新收集公开的代码片段或者重新分析算子文档.

ConFL 主要测试 raw_ops 模块，包含 1355 个算子. 其中，24 个算子已经被弃用或无意义，例如 Abort 算子. 在剩下的 1331 个算子中，有 65 个依赖于 TPU，例如 SendTPUEmbeddingGradients. 虽然 ConFL 理论上能够检测这些算子，但由于硬件限制，没有对这类算子进行实验. 最终选择了 1266 个非 TPU 环境依赖的算子作为测试目标. 此外，ConFL 还可以测试其他模块，例如 ConFL 在 IO 模块中发现了 CVE-2020-26269.

图 13 显示了算子参数个数的分布情况，平均每个算子需要接收 5 个参数. 976 个算子具有 2 到 6 个参数，其中最常见的是 3 个参数，出现在 232 个算子中. 此外，有 4 个算子的参数数目超过 20 个.

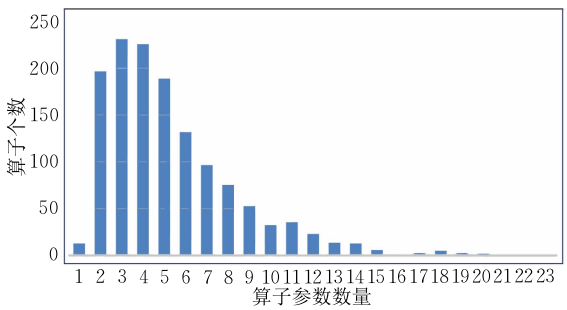

图 13　算子参数统计

DocTer 在 Tensorflow 的 2.1.0 版本中通过分析文档共收集到 1008 个算子，而 ConFL 在同版本中通过分析源码能够收集 1217 个算子. 同时 FreeFuzz 在 2.4 版本中发现 688 个算子，而 ConFL 在同版本中能够发现 1334 个算子. ConFL 的算子发现能力要优于目前的检测工具，因此相对于其他工具能更全面地检测框架源码中的安全问题.

5.3　算子约束提取能力评估

ConFL 采用基于约束制导的测试数据生成方法，因此约束的种类与数目影响测试数据的生成. ConFL 共提取四类 1646 条约束，具体如表 3 所示.

表 3　约束计数

约束类型	约束数量/条
环境类约束	6
依赖类约束	23
验证类约束	1519
逻辑类约束	98

本文通过解析 ML 框架文档收集了 6 条环境类约束，分为硬件环境约束与运行环境约束两部分. 硬件环境如 CPU、GPU 等，运行环境如动态图、静态图等. 虽然环境类约束的数量相对较少，但非常重要，加入环境类约束能够调用不同处理逻辑，进而覆盖更多的代码，并发现新的漏洞.

ConFL 在分析算子信息后提取了 23 条依赖类约束. 通过使用依赖类约束，能够让之前无法执行成功的部分算子成功执行. 算子能否执行成功还与参数能否通过校验相关，由于算子类型和数量多种多样，ConFL 共提取了 1519 条验证类约束. 同样，引入逻辑类约束能够构建完整的约束树，覆盖更多的代码.

验证类约束分为两种类型：单个参数的约束和多个参数之间的约束. 另外，约束也从不同的角度描述参数的属性，*ndim* 表示维数，*shape* 表示张量的形状，*size* 表示元素数量，*value* 表示具体值，*dtype* 表示参数的类型. 如表 4 所示，在单个参数约束中，对 *ndim* 的校验数目最多，其次是 *size*. 如在算子 ArgMax 的验证类约束中，约束“*dimension*.*ndim*==0”代表参数 *dimension* 的 *ndim* 值需要为 0.

表 4　单个参数的验证类约束分类

ndim	*shape*	*size*	*value*	*dtype*
837	92	202	92	15

在表 5 中展示了多个参数之间的情况，行和列指定不同参数属性之间的约束. 如算子 BatchToSpaceND 中的多参数约束“*input*. *ndim* > *block_shape*. *shape*[0]”，代表 *input* 参数的维数需要大于 *block_shape* 参数维度的第一个值.

表 5 参数之间的验证类约束分类

	ndim	*shape*	*size*	*value*
ndim	1			
shape	10	199		
size	3	6	12	
value	11	17	-	12

ConFL 共收集了四种类型的约束共 1646 条，其中验证类约束数量最多，占总约束的 90% 以上，说明源码中对参数的校验较为严格，提取出的约束从单个参数和参数之间的依赖两个维度对参数进行限制. 在下一节中详细说明约束对测试样本生成的有效性.

5.4 算子测试数据有效性评估

在本实验中，将 ConFL 与随机生成（Atheris[33]）方法以及最先进的（SOTA）模糊测试工具产生的代码覆盖率进行比较. 考虑到各种 SOTA 模糊测试工具不能覆盖所有算子，首先在 1266 个算子上与 Atheris 进行比较实验. 然后，选择所有 SOTA 模糊测试工具都能覆盖的 400 个算子，进行与 SOTA 模糊测试工具的比较实验.

5.4.1 与随机生成方法的覆盖率对比

本实验分别使用 Atheris 和 ConFL 为每个算子生成 10000 个测试输入，并记录成功执行的次数. 成功执行是指程序正常退出或触发崩溃，不成功执行是指程序由于参数错误（例如 Python 代码异常）而失败. 实验结果显示，Atheris 在所有测试的算子中共成功执行了 669249 次，而 ConFL 达到了3534170次，增长率为 428.08%，表明 ConFL 使用提取出的约束对测试数据进行限制，显著提高了输入的有效性.

此外，本文研究了代码覆盖增长率与参数数量之间的关系. 图 14 显示了 ConFL 相对于 Atheris 的增长率. 所有算子的平均增长率为 625.94%；5 个参数的算子具有最高的增长率，为 1,417.94%. 尽管随着参数数量的增加，增长率会下降，但 ConFL 仍然明显优于 Atheris. 这表明，当算子具有少量参数时，随机生成的数据可以使算子成功执行. 然而，随着参数个数的增加，随机生成数据方法的限制变得更加明显. 当参数数量在大于等于 7 时，Atheris 的成功执行次数基本均为 0，无法生成合法化的测试数据. ConFL 虽然没有继续保持高增长率，但依旧可以成功生成合法化测试数据.

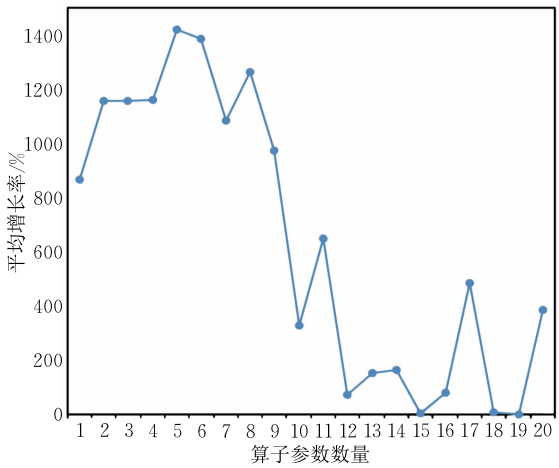

图 14 平均增长率

相比于 Atheris 随机生成数据的方式，ConFL 生成的数据更加合法. Atheris 随机生成的数据难以通过算子代码中的合法性检验，进而无法全面检测算子的安全性. 基于约束生成合法性数据的 ConFL，能够执行到算子较为深层的代码逻辑中，进而检测更为复杂的安全问题，验证了基于约束生成的数据的有效性.

同时，实验统计了 ConFL 与 Atheris 的代码覆盖率. 如图 15 所示，实验中对 1266 个算子进行了 20 min 的模糊测试，发现 Atheris 在 20 min 的覆盖率趋于稳定，仅覆盖了 4929 行代码，这表明大多数输入都是无效的，最终程序终止在参数的合法性检查处. 相比之下，ConFL 的代码覆盖率不仅在前 5 min 内迅速增加，而且在随后的测试中保持稳定增长的趋势. 在有限的时间内，ConFL 相比于 Atheris 覆盖率提高了 228.83%，体现出 ConFL 在生成有效输入方面的能力.

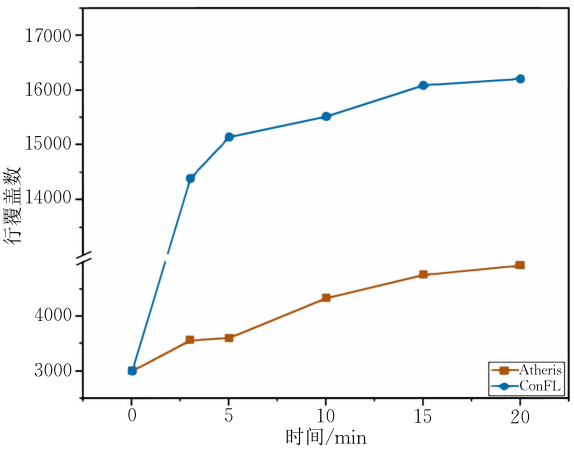

图 15 ConFL 与 Atheris 的覆盖率对比

5.4.2　与 SOTA fuzzer 的覆盖率对比

本文将 ConFL 与最先进的模糊测试工具(包括 DocTer、FreeFuzz、DeepRel 和 IvySyn)进行比较. 由于不是所有的 fuzzer 都能覆盖以上 1226 个算子，因此随机选择 400 个所有 fuzzer 都可以覆盖的算子作为 benchmark，以 Atheris 作为基线，对 benchmark 的每个算子进行 20 min 的测试. 随后，记录 benchmark 中所有算子的代码覆盖率.

如图 16 所示，测试结果显示 ConFL 在代码覆盖率指标上始终优于 DocTer、FreeFuzz、DeepRel 和 IvySyn. 在测试的前 5 min 内，不同工具测试得到的覆盖率快速增长，ConFL 在 5 min 时覆盖率达到 5513 行，比 Ivysyn 的覆盖率高出 985 行. 在 5 min 之后，ConFL 的覆盖率仍逐渐上升，这表明约束有助于生成算子的有效输入，并大大提高了执行成功率. 此外，本文中基于约束为算子运行提供了多样的执行环境和优质的测试数据，相对于最先进的模糊测试工具显著提高了算子的代码覆盖率. 环境类约束和依赖类约束设置算子的执行环境，保证算子及时获取到所需的资源；验证类约束确保生成的数据能够通过参数的合法性检验，例如对维度、类型和数值的约束，从而进入到更深层次的代码中进行运算，以覆盖更多的代码；逻辑类约束用于覆盖更多的分支逻辑，通过设置不同的约束条件，可以引导算子在不同的逻辑分支中执行，从而增加覆盖率. 因此，ConFL 在生成有效输入和探索更广泛的代码路径方面更有效.

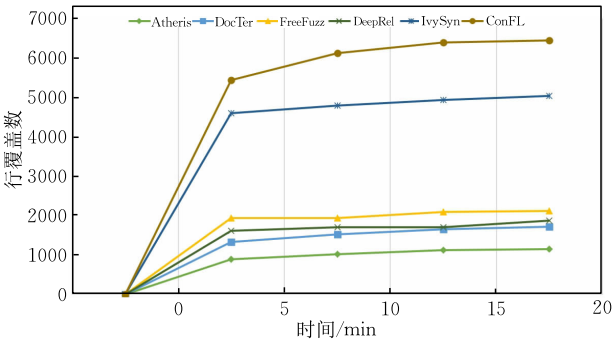

图 16　ConFL 与 SOTA fuzzer 的覆盖率对比

5.5　漏洞检测能力评估

在已知漏洞发现能力对比中，我们根据漏洞公告，对漏洞信息进行清洗、修正，形成测试集. 我们选定在 TensorFlow 2.8 版本中进行漏洞验证，发现测试集中存在 94 个漏洞验证程序可执行成功，其中浮点数漏洞 4 个，异常漏洞 57 个，段错误 33 个. 由于 TensorFlow 在解析不合法的数据时会自行异常崩溃，不会导致严重的安全问题，因此选择段错误和浮点数异常漏洞共计 37 个漏洞作为数据集，分析 SOTA 工具检测 2.8 版本的 37 个漏洞的能力.

通过对每个算子测试 1000 次，对比结果如表 6 所示，结果表明，ConFL 得益于完整的算子接口收集、基于约束的精确化数据，能够检测出 32 个已知漏洞，而 DocTer 只能发现 6 个已知漏洞. DeepRel 与 FreeFuzz 没有发现已知漏洞，经过分析发现，两种检测工具的算子收集能力无法全部覆盖已知漏洞算子，同时生成的测试数据难以满足算子的参数校验，因此在已知漏洞的检测能力方面大不如 ConFL.

表 6　已知漏洞发现能力对比结果

工具名称	发现漏洞个数
ConFL	32
DocTer	6
IvySyn	8
DeepRel	0
FreeFuzz	0

经过进一步分析，发现目前针对 ML 框架算子测试的工具缺少生成特定类型的能力，如 numpy.nan 类型，同时缺少针对特定格式文件变异的能力，如 gif 文件.

5.6　漏洞检测结果分析

在 ConFL 对 TensorFlow 框架进行漏洞检测时，不能能够有效发现已知漏洞，还能够检测未知漏洞. ConFL 成功地在 TensorFlow 框架中发现了 84 个漏洞，所有这些漏洞都已经得到官方确认并分配了 CVE 编号.

我们通过人工分析的方式，定位漏洞位置并分析漏洞类型. 具体步骤如下：(1) 配置操作系统选项，使得程序异常退出时，将崩溃时的内存状态保存为 core dump 文件；(2) 通过 gdb 加载 core dump 文件，结合寄存器和内存状态确定漏洞位置；(3) 通过 gdb 进行动态调试，逐步分析导致漏洞的原因.

5.6.1　漏洞类型分析

根据表 7 所示，通过使用 gdb 调试 84 个漏洞，分析函数调用栈与寄存器信息，按照类型对其进行分类，前五名分别是越界(OOB)、空指针异常(NPE)、浮点异常(FPE)、整数溢出(IOF)和释放后使用(UAF).

表 7　TensorFlow 框架中数量在前 5 名的漏洞类型

漏洞类型	示例 CVE 编号	示例算子
OOB	CVE-2021-41226	SparseBincount
NPE	CVE-2021-41209	Conv2D
FPE	CVE-2022-21725	AvgPoolGrad
IOF	CVE-2022-21733	StringNGrams
UAF	CVE-2021-37652	BoostedTreesCreateEnsemble

(1)越界(OOB).当操作访问超出预期的内存边界时,将会导致数据损坏、程序崩溃等问题.越界漏洞是数量最多的漏洞类型,占总量的35.71%,主要原因在于缺少对待访问内存的合法性验证,如TensorFlow框架的CVE-2021-41226,是SparseBincount算子中缺少对数值取值范围的验证,从而访问预期内存之外的内存地址,造成越界漏洞.

(2)空指针异常(NPE).当程序尝试通过空指针引用访问或操作对象时,将会导致安全问题.此类漏洞占比23.81%,在ML框架中主要由于错误的维度,造成某些数据结构指针为nullptr,如TensorFlow框架的CVE-2021-41214,RaggedCross算子的*dense_input*参数是一个零维tensor,导致保存维度的结构体指针为空,后续继续操作该指针导致空指针异常.

(3)浮点数异常(FPE).FPE漏洞涉及浮点运算中的错误,例如除零或整型溢出,导致崩溃或不正确的计算,进而影响ML框架的可靠性.FPE类型漏洞占比为11.9%,如TensorFlow框架的CVE-2022-21725,AvgPoolGrad算子处理*strides*参数不当导致浮点数异常.

(4)整数溢出(IOF).IOF漏洞发生在整数操作产生的值太大或太小而无法由整数类型表示时,导致数据损坏、崩溃或其他意外后果.整数溢出漏洞占总漏洞的10.71%,在于算子中缺少对整数值的限制,如TensorFlow框架的CVE-2022-21733,StringNGrams算子处理*pad_width*参数时忽略负整数的影响,从而导致整数溢出.

(5)释放后使用(UAF).当程序继续使用一个已释放内存对象时,会发生UAF漏洞,导致崩溃、数据损坏或其他意外后果,此类漏洞占比2.38%,如TensorFlow框架的CVE-2021-37652,BoostedTreesCreateEnsemble算子处理参数不当导致释放后使用漏洞.

本章节对ConFL发现TensorFlow的84个漏洞进行了深入的分析,对漏洞类型进行了分类,帮助开发人员提高TensorFlow框架的安全性、稳定性和可靠性.

5.6.2 漏洞成因分析

通过分析ConFL在TensorFlow中检测到的84个漏洞,确定了这些漏洞的因果关系,如表8所示,按照触发漏洞的属性分为三类:维度、类型和数值.

与维度相关的漏洞,一种是零维向量导致NPE、OOB和FPE,例如CVE-2021-37672;以及一个大的索引导致的OOB和NPE漏洞,例如CVE-2021-37655.在类型导致的漏洞中,不正确的张量类型值会触发拒绝服务(DoS)漏洞,如CVE-2020-26268所示.由数值引起的漏洞分为零、大整数和负数,当*tensor*中的数据或参数值为零时,可能导致FPE和OOB漏洞,如CVE-2022-21725;大整数值可能导致OOB、IOF和类型混淆(TC)漏洞,如CVE-2022-21727;负值可能导致OOB和IOF漏洞,如CVE-2022-21733.

表8 漏洞成因

类别	输入类型	漏洞类型	示例CVE编号
维度	零维度	NPE, FPE, OOB	CVE-2021-37672
	较大的索引	OOB, NPE	CVE-2021-37655
类型	字符串	DoS	CVE-2020-26268
数值	零	FPE,OOB	CVE-2022-21725
	大整数	OOB, IOF, TC	CVE-2022-21727
	负数	OOB, IOF	CVE-2022-21723

经过对漏洞的分析,发现当前的ML框架存在以下问题:

(1)开发者优先考虑性能和功能,而忽略了安全性,导致缺乏全面的用户输入验证,特别是对于空数组和空指针.

(2)机器学习算法的计算性质导致频繁出现浮点数异常和整数溢出问题.

(3)ML框架算子参数之间存在相互依赖关系,即使参数满足自身约束条件,仍有可能多个参数共同计算,产生安全问题.

5.6.3 漏洞案例分析

第5.6.3节将以ConFL发现的BTCBFS算子漏洞为例,说明ConFL如何生成有效输入,并对真实世界中的ML框架进行高效的漏洞检测.

BTCBFS的约束条件表明,参数*split_type*是一个字符串,只有两个有效值:"inequality"和"equality".同时,*stats_summary*和*logits_dimension*参数之间存在验证类约束:*stats_summary*的维度与*logits_dimension*的值相关.ConFL根据以上算子约束自动生成测试模板,并为*split_type*、*stats_summary*和*logis_dimension*生成有效测试数据,避免在浅层代码逻辑测试时耗费大量时间和计算资源.

最后,ConFL发现了位于BTCBFS算子中的越界读漏洞,同时能够自动化生成漏洞验证程序(图17).从漏洞验证程序中可以看出,算子BTCBFS不同参数及其类型信息均正确,同时,ConFL在严格保证生成合法数据的前提下,如参数*split_type*为特定字

符串‘*equality*’，生成了部分多样化、稀有化的测试数据，如 *node_id_range* 参数中的 0x400000，这也是能够发现 BTCBFS 算子中出现越界读漏洞的关键.

```
tensorflow.raw_ops.BoostedTreesCalculateBestFeatureSplit(
        node_id_range=[0x400000,0x400001],
        stats_summary=[
                [[[2.0, 3.0]], [[3., 3.]], [[3., 3.]]],
                [[[3., 4.]], [[5., 6.]], [[6., 6.]]]
        ],
        l1=[0.0],
        l2=[0.0],
        tree_complexity=[1.0],
        min_node_weight=[0.7],
        logits_dimension = 1,
        split_type = 'equality'
)
```

图 17　BTCBFS 的 PoC

经过对应的源代码分析发现，如图 18 所示，参数 *node_id* 的数据来源于输入参数 *node_id_range*，当其数据大小超出了 *stats_summary* 的范围，将导致 *stats_mat* 的指针指向一个超出控制范围的地址. 后续操作如果对该地址的内容进行读取或者修改，将导致敏感信息泄露或者任意命令执行等安全问题.

```
    ConstMatrixMap stats_mat(&stats_summary(node_id, 0, 0, 0), ...);
    const Eigen::VectorXf total_grad =
stats_mat.leftCols(logits_dim).colwise().sum();
```

图 18　在 BTCBFS 中导致漏洞的源代码

5.6.4　其他 ML 框架漏洞

本文在第 4.4 节中介绍了 ConFL 适配其他 ML 框架的方法，包含 PyTorch 和 PaddlePaddle. 迄今为止，ConFL 已经在这些框架中发现了 7 个漏洞(表 9)：在 PaddlePaddle 中，ConFL 共发现了 4 个漏洞；在 PyTorch 中，检测到了 3 个越界(OOB)漏洞. 这些结果表明，ConFL 对其他 ML 框架具有通用性.

表 9　其他 ML 框架中检测到的漏洞

框架	漏洞类型	算子	ISSUE-ID
PaddlePaddle	OOB	gather_tree	33382
		strided_slice	33006
	FPE	Pool3d	33036
	DF	split	30942
Pytorch	OOB	quantized_lstm_cell	50037
		_remove_batch_dim	50038
		Native_layer_norm	50090

ConFL 具备提取单个参数约束与多个参数间约束的能力，并生成有效的测试数据，这对于发现工业界中常用的 ML 框架漏洞非常有效.

6　结束语

本文介绍了一种基于约束制导的 ML 框架算子测试方法并形成原型工具 ConFL，用于检测 ML 框架中隐藏的安全漏洞. ConFL 通过分析源代码以收集算子，从源代码中提取算子约束，并自动构建算子测试模板，根据提取的约束生成有效测试数据. 通过评估，ConFL 展示了在生成有效参数方面的出色能力. 此外，该方法已成功地检测出 TensorFlow 框架中的 84 个未知漏洞，以及 PyTorch 和 PaddlePaddle 的 7 个未知漏洞. 本节阐述下一步的工作计划.

(1) 约束求解的优化. 利用提取的约束条件，后续可采用高效的约束求解技术生成有效的测试输入. 从而更多地覆盖算子代码并提高发现深层漏洞的可能性.

(2) 算子优化. ML 框架在实际计算中，会有算子优化的情况，例如在实际计算过程中的算子融合，旨在减少内存占用并提高效率. 目前，ConFL 单独测试算子，未来应考虑算子融合后的场景.

(3) 文件变异. 一些算子需要特定的文件格式作为参数，例如 DecodeJEPG 的图像文件或 DecodeWAV 的音频文件. 为了更有效地测试具有复合类型参数的算子，计划在未来的工作中添加文件格式变异.

参　考　文　献

[1] Dosovitskiy A, Beyer L, Kolesnikov A, et al. An image is worth 16x16 words: Transformers for image recognition at scale. arXiv preprint arXiv:2010.11929, 2020

[2] Kaldi, http://kaldi-asr.org/, 2023

[3] Vaswani A, Shazeer N, Parmar N, et al. Attention is all you need//Proceedings of the Advances in Neural Information Processing Systems. Long Beach, USA, 2017: 5998-6008

[4] Tensorflow. https://www.tensorflow.org/, 2023

[5] Paszke A, Gross S, Massa F, et al. Pytorch: An imperative style, high-performance deep learning library//Proceedings of the Advances in Neural Information Processing Systems. Vancouver, Canada, 2019: 8024-8035

[6] Jia Y, Shelhamer E, Donahue J, et al. Caffe: Convolutional architecture for fast feature embedding//Proceedings of the 22nd ACM International Conference on Multimedia. 2014: 675-678

[7] Daily download quantity of tensorflow package. https://pypistats.org/packages/tensorflow, 2023

[8] TensorFlow Security. https://github.com/tensorflow/tensorflow/tree/master/tensorflow/security, 2023
[9] Peach. https://www.peach.tech/, 2023
[10] AFL. https://github.com/google/AFL, 2023
[11] Libfuzzer. https://llvm.org/docs/LibFuzzer.html, 2023
[12] Xie D, Li Y, Kim M, et al. DocTer: Documentation-guided fuzzing for testing deep learning API functions//Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis. 2022: 176-188
[13] Wei A, Deng Y, Yang C, et al. Free lunch for testing: Fuzzing deep-learning libraries from open source//Proceedings of the 44th International Conference on Software Engineering. Pittsburgh, USA, 2022: 995-1007
[14] Deng Y, Yang C, Wei A, et al. Fuzzing deep-learning libraries via automated relational API inference//Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering. Singapore, 2022: 44-56
[15] Christou N, Jin D, Atlidakis V, et al. IvySyn: Automated vulnerability discovery for deep learning frameworks. arXiv preprint arXiv:2209.14921, 2022
[16] BoostedTreesCreateQuantileStreamResource. https://www.tensorlow.org/api_docs/python/tf/raw_ops/BoostedTreesCreateQuantileStreamR source, 2023
[17] BoostedTreesCalculateBestFeatureSplit. https://www.tensorflow.org/api_docs/python/tf/raw_ops/BoostedTreesCalculateBestFeatureSplit, 2023
[18] Atlidakis V, Godefroid P, Polishchuk M. Restler: Stateful rest api fuzzing//Proceedings of the 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE). Montreal, Canada, 2019: 748-758
[19] syzkaller. https://github.com/google/syzkaller, 2023
[20] Corina J, Machiry A, Salls C, et al. Difuze: Interface aware fuzzing for kernel drivers//Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security. Dallas, USA, 2017: 2123-2138
[21] Han H S, Cha S K. Imf: Inferred model-based fuzzer//Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security. Dallas, USA, 2017: 2345-2358
[22] Babić D, Bucur S, Chen Y, et al. Fudge: fuzz driver generation at scale//Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. San Francisco, USA, 2019: 975-985
[23] Choi J, Kim K, Lee D, et al. NTFuzz: Enabling type-aware kernel fuzzing on windows with static binary analysis//Proceedings of the 2021 IEEE Symposium on Security and Privacy (SP). San Francisco, USA, 2021: 677-693
[24] Zhang C, Lin X, Li Y, et al. APICraft: Fuzz Driver Generation for Closed-source SDK Libraries//Proceedings of the 30th USENIX Security Symposium. Vancouver, Canada, 2021: 2811-2828
[25] Xiao Q, Li K, Zhang D, et al. Security risks in deep learning implementations//Proceedings of the 2018 IEEE Security and Privacy Workshops (SPW). San Francisco, USA, 2018: 123-128
[26] Tan X, Gao K, Zhou M, et al. An exploratory study of deep learning supply chain//Proceedings of the 44th International Conference on Software Engineering. Pittsburgh, USA, 2022: 86-98
[27] Xie X, Ma L, Juefei-Xu F, et al. Deephunter: A coverage-guided fuzz testing framework for deep neural networks//Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis. Beijing, China, 2019: 146-157
[28] pybind11. https://github.com/pybind/pybind11, 2023
[29] Swig. https://github.com/swig/swig/, 2023
[30] Graph execution vs. eager execution. https://www.tensorflow.org/guide/intro_to_graphs#graph_execution_vs_eager_execution, 2023
[31] XLA. https://www.tensorflow.org/xla, 2023
[32] Lattner C, Adve V. LLVM: A compilation framework for lifelong program analysis & transformation//Proceedings of the 2004 International Symposium on code generation and optimization. Palo Alto, California, 2004: 75-86
[33] Atheris. https://github.com/google/atheris, 2023

LIU Zhao, M. S. His main research interests include software security and AI security.

ZOU Quan-Chen, Ph. D., associate professor. His main research interests include software security and AI security.

YU Tian, M. S. Her main research interests include software security.

WANG Xuan, M. S. Her main research interests include software security.

ZHANG De-Yue, M. S. His main research interests include AI security and software security.

MENG Guo-Zhu, Ph. D., associate professor. His main research interests include software security and AI security.

CHEN Kai, Ph. D., professor. His main research interests include software security and AI security.

Background

This paper studies the analysis and detection of the security of machine learning framework. The previous work tested a small subset of operators in the machine learning framework. At the same time, the generated operators test data is relatively random, and it is difficult to generate legal test data. On one hand, those methods cannot detect all operators' source code. On the other hand, existing methods can only detect shadow bugs, and cannot find the hidden deeper security vulnerabilities.

In order to overcome the shortcomings of current detection methods and find the deeper vulnerabilities of machine learning frameworks, in the paper, We propose a constraint-based approach to test operators. By analyzing operator information from the source code and the operator's description, we can get all operators' call path. Then, by tracking the flow of operator parameters, we construct four types of constraints on operators and parameters, and generate more legal test data based on the constraints to discover deeper security issues.

The security risks of machine learning frameworks need to be taken seriously. Developers use machine learning frameworks to provide users with intelligent services. When security issues arise in the framework, it will affect the information security of a large number of developers and massive users of intelligent services. The method proposed in this paper can effectively detect security issues in machine learning frameworks, thereby protecting user privacy and security.

This work is supported by the National Key Rgyqzwz D Program of China with No. 2020AAA0104300.