RESTful API 中批量赋值漏洞的自动化黑盒测试

Davide Corradini*、Michele Pasqua‡ 和 Mariano Ceccato§维罗纳大学计算机科学系——意大利维罗纳 电子邮件： *davide.corradini@univr.it、 ‡michele.pasqua@univr.it、 §mariano.ceccato@univr.it

arXiv:2301.01261v1 [cs.CR] 3 Jan 2023

论文已被以下会议录接收发表：*45th* IEEE/ACM 软件工程国际会议（ICSE 2023）

本文档是同行评审前的初步版本。最终版本可在出版商网站上找到。

本文引用格式：

D. Corradini, M. Pasqua 和 M. Ceccato，《RESTful API 中批量赋值漏洞的自动化黑盒测试》 s”，2023年 IEEE/ACM45th 软件工程国际会议（ICSE），2023年。

摘要——批量赋值是 RESTful API 中最突出的一类漏洞。这种漏洞源于常见 Web 框架的错误配置：攻击者可以利用命名约定和自动绑定来构造恶意请求，从而写入机密资源并（大规模地）覆盖本应为只读和/或机密的数据。

本文采用黑盒测试视角，自动检测 RESTful API 中的批量赋值漏洞。执行场景完全基于 OpenAPI 规范生成，该规范列出了可用操作及其消息格式。我们使用聚类对相似操作进行分组，并识别只读字段；后者是批量赋值的候选目标。随后，通过实例化抽象测试模板自动生成交互序列，以尝试利用潜在漏洞。最后，运行测试用例，并通过专用判定器评估其执行结果，以确定漏洞是否能够被成功利用。

所提出的新方法已在一组使用不同编程语言编写的案例研究中实现并进行了评估。评估结果表明，该方法在检测植入漏洞方面非常有效，准确率极高。

索引词—REST API、安全测试、黑盒测试、自动化软件测试、批量赋值

I. 引言

RESTful API（简称 REST API）正成为访问面向 Web 的资源以及在公共互联网中互连软件系统的标准技术。它们通过 HTTP 协议在 Web 上运行，通常交换 JSON 负载，因此也称为 Web API。

鉴于 REST API 作为连接不同计算机系统的核心集成技术，承担着至关重要的职责，因此应尽早揭示其实现中的缺陷和漏洞。事实上，整个集成系统的安全性建立在其安全性以及参与组合的原子 REST API 的可靠性之上。

正如 OWASP 基金会在年度调查中所报告的，REST API 面临着一些已知的威胁[1]。尤其是，批量赋值是这一编程领域最突出的特有漏洞之一。这种漏洞源于常见 REST API 框架的错误配置；这些框架通常会自动将输入数据字段（由潜在攻击者控制）绑定到内部数据表示（例如数据库列）。若成功利用批量赋值漏洞，攻击者便可操纵私人数据，前提是他们能够猜出 REST API 所使用的内部数据结构（例如数据库表和列）的名称。

自动化测试 REST API 时，一种广泛采用的视角是黑盒测试 [1]–[5]，它不需要访问 API 的源代码。本文同样采用黑盒方法，因为这种方法能够成功测试以不同编程语言实现、基于任意 REST 框架且可能由闭源组件构成的 REST API。事实上，黑盒视角能够独立于 REST API 的内部架构（通常采用微服务架构）对其进行测试。

本文提出了一种新方法，用于自动测试 REST API 是否存在批量赋值漏洞。我们分析待测 REST API 的规范，以确定哪些数据不应被传入请求覆盖，即哪些数据应为只读，因此可能成为批量赋值攻击的目标。首先，我们通过聚类识别具有相似数据的操作组，然后利用一些启发式方法识别只读

[1]https://owasp.org/www-project-api-security/

字段。随后，通过实例化抽象测试模板，自动生成具体的测试场景。最后，安全判定器会监控这些测试用例的执行，以揭示漏洞何时暴露。

实证评估表明，我们的方法在检测批量赋值漏洞方面非常准确；事实上，在几乎所有纳入考察的案例研究中，都检测出了所有漏洞，且没有假阳性。

据我们所知，现有文献中尚无自动化黑盒方法可用于检测 REST API 中的批量赋值漏洞。

本文结构安排如下。第II节介绍 REST API 和批量赋值所需的背景知识；第III 节介绍我们针对批量赋值提出的新型测试方法。第IV 节介绍实验设计的定义，第V 节应用该设计并讨论收集到的结果。在第VI节将我们的方法与相关工作进行比较后，第VII节对本文作总结。

II. 背景

A. RESTful API

RESTful API（简称 REST API）是一种遵循 REST（表述性状态转移）架构风格的 Web API [6]。REST API 提供统一接口，用于创建（*C*）、读取（*R*）、更新（*U*）和删除（*D*）资源（称为 CRUD 语义）。资源由 HTTP URI 标识，CRUD 操作通常映射到 HTTP 方法 POST、GET、PUT（或 PATCH）和 DELETE。

例如，可以考虑 VAmPI[2]，，这是一个用于管理图书馆用户和图书的开源 REST API。指向用户资源的 HTTP URI 是 /用户，HTTP 操作 GET /users 和 POST /users/register 分别用于获取已注册用户列表和在系统中注册新用户。

REST API 可以接受输入参数，以指定执行某项操作所需的其他信息，例如要检索的用户标识符（例如，/users/{用户名}），或者通过 POST 方法在请求正文中提交要在系统中注册的结构化对象。

B. OpenAPI 规范

OpenAPI[3] 定义了用于描述 REST API 的标准。根据 OpenAPI，API 服务通过结构化文件（ YAML 或 JSON ）进行描述，该文件规定了如何使用 URI 访问 API、采用何种身份验证方案，以及 API 中所有可用操作的详细信息。具体而言，文件还会给出每项操作在请求中使用的输入参数（及其模式）以及响应模式。

图 1 展示了我们示例中 OpenAPI 规范的一个片段*VAmPI*。在开头的标题中，指定了版本、许可证和应用程序接口的基础 URL（出于篇幅考虑，代码中未展示），此后 OpenAPI 规范

[2]https://github.com/erev0s/VAmPI [3]https://www.openapis.org/

```
/用户:
  get:
    操作 ID: 检索_所有_用户
    响应:
      '200':
        description: 查看所有用户的详细信息
        内容:
          application/json:
            模式:
              类型: 数组
              项目:
                类型: 对象
                属性:
                  管理员:
                    类型: 布尔值
                  电子邮件:
                    类型: 字符串
                  密码:
                    类型: 字符串
                  用户名:
                    类型: 字符串
/用户/注册:
  post:
    操作 ID: 注册_新_用户
    请求体:
      内容:
        application/json:
          必填: true
          模式:
            类型: 对象
            prop属性:
              用户名:
                类型: 字符串 g
              密码:
                类型: 字符串
              电子邮件:
                类型: 字符串
```

图 1. VAmPI 的 OpenAPI 规范部分内容。

包含一个路径数组，即应用程序接口中可用 URL 路径的列表。在我们的示例中，我们只关注两个 URL 路径：/用户，以及 /用户/注册。每个路径支持一个或多个 HTTP 方法。路径与 HTTP 方法共同构成一个操作，通常由操作 ID 标识。例如，/用户 中的 GET 方法（检索所有用户）用于检索系统中所有已注册用户的列表，而 /用户/注册 中的 POST 方法对应“注册用户”操作，用于在系统中注册新用户。

操作的输入和输出与一个模式相关联，该模式规定其类型，并可选择性地规定一组值约束（例如，对数值参数设定最小值或最大值）。类型可以是原子类型（例如整数和字符串），也可以是结构化类型（即复合对象）。例如，注册新用户操作需要一个请求正文，其中包含一个由三个字符串字段组成的 JSON 对象：用户名、密码和电子邮件。类似地，检索所有用户操作的响应模式是一个 JSON 数组，其中包含多个 JSON 对象，每个对象由四个字段组成：三个字符串字段（用户名、密码和电子邮件），以及一个布尔字段（管理员）。

C. 批量赋值漏洞

开发者通常会依赖框架来开发 REST API。这些框架适用于所有主流 Web 编程语言，提供一系列功能和自动化工具，以加快并简化.....的开发

REST API。在最常用的框架中，有 Spring（Java）、Express.js（JavaScript）、FastAPI（Python）、Flask（Python）、Laravel（PHP）和 Slim（PHP）。

REST API 框架支持的一种常见做法是使用命名约定，将 HTTP 请求中的输入数据映射到 REST API 的后端数据表示中，例如，在输入参数与数据库列同名时将二者对应起来。此功能通常默认启用，并通过简化数据流来加快 REST API 的开发。

根据 OWASP 基金会的说法，如果外部用户能够通过利用自动参数绑定的错误配置，操纵原本只读的资源值，那么该 REST API 就被认为易受批量赋值攻击。

例如，VAmPI API 基于 Flask 框架构建，该框架会自动将 HTTP 请求中的 JSON 用户对象映射到 User Python 类，随后再将其映射到用户数据库表。

用户表（以及 User 类）包含一个 admin 布尔列（类字段），该列原本应为只读。也就是说，虽然用户的其他属性可由终端用户编辑（例如密码或电子邮件地址），但 admin 字段不应通过此操作进行更改；事实上，此操作的 OpenAPI 规范并未将其列为输入参数。

但是，如果 REST 框架配置不当（例如采用默认配置），框架就会在注册用户的请求中，自动将额外的管理员 HTTP 参数绑定到用户数据库表中相应的管理员列。攻击者可以伪造一个发往 POST /users/register 操作的请求，该操作原本用于注册新用户；攻击者在请求中注入一个 OpenAPI 规范未提及的额外管理员参数。该额外参数 admin 会自动关联到用户表中的管理员列，因此，攻击者控制的 HTTP 参数值会覆盖数据库中的合法值。

自动绑定能够提高生产效率：它免去了开发人员手动配置输入/输出数据与数据库表/列之间关系的工作。事实上，这是所有主流 REST API 框架都提供的一项常见功能。然而，盲目使用此功能往往会导致批量赋值漏洞，也称为自动绑定漏洞或对象注入漏洞。

如今，REST API 框架提供了一种缓解此类漏洞的方法：允许开发者明确指定哪些 HTTP 输入参数可以映射到内部数据表示。不过，开发者应了解批量赋值漏洞，以便正确应用这一缓解措施。事实上，批量赋值漏洞仍然十分普遍，并被列为 REST API 中最常见的漏洞之一。

III. 方法

OWASP 基金会提供了一些用于检测和缓解批量赋值漏洞的指南 [7]，即使在无法访问 API 源代码的情况下（也就是从黑盒视角出发）也是如此。这些指南要求识别可能在后端创建或更新资源的 HTTP 请求，并通过分析后端返回的响应，识别敏感对象字段（例如 `isAdmin` 属性）。这些指南旨在供开发者手动使用，因此并未提供任何有关如何自动化检测批量赋值漏洞的提示。

因此，我们提出一种安全测试方法，以自动发现 REST API 中潜在的批量赋值漏洞。通常，REST API 的源代码并不可用（或仅部分可用）。此外，REST API 通常由许多动态分配的分布式组件构成（这是微服务架构的特点），使得源代码分析在计算上颇具挑战。基于这些原因，我们采用黑盒视角，即我们的方法不要求能够访问源代码。这也意味着，无论实现所使用的编程语言或框架为何，我们的方法都适用于任何 REST API。我们的方法仅有以下要求：(i) 提供 OpenAPI 规范，以获取操作及其输入/输出消息格式；以及 (ii) 能够以黑盒方式通过 HTTP 访问被测 REST API，以发送测试请求并接收响应。

我们用于测试批量赋值漏洞的策略由以下三个主要步骤组成。

1) 识别只读属性。我们解析被测 REST API 的 OpenAPI 规范，列出操作及其输入/输出属性。随后对操作进行聚类，将处理相似数据资源的操作归为一组。然后，我们比较处理相似资源的操作，以找出只读属性，即那些出现在读取操作的输出中、但未出现在写入操作输入中的属性。这些字段是批量赋值漏洞的候选对象。

2) 测试用例生成。我们自动生成测试场景，将请求组织成序列，旨在尝试覆盖只读属性。通过从抽象测试模板出发，计算具体交互来生成测试用例。

3) 安全判定器。在被测 REST API 上执行测试用例，并由我们的安全判定器监控执行过程。当测试用例成功覆盖某个只读参数时，判定器便会报告批量赋值漏洞已被成功利用。下文将详细介绍上述所有步骤。

A. 只读属性的识别

测试批量赋值漏洞的第一步，是根据 OpenAPI 规范识别只读属性，因为它们可能成为成功攻击的目标。具体来说，这需要

表 I 用于聚类的操作参数布尔编码。

操作	管理员	电子邮件	密码	用户名
注册用户	否	True	True	True
检索所有用户	True	True	True	True

识别操作是读取还是写入数据（即推断操作的 CRUD 语义），以及这些操作是否针对同一资源执行。

操作的 CRUD 语义与资源。我们提出一种自动化流程，用于推断操作的 CRUD 语义以及操作所作用的资源。此信息仅通过检查 OpenAPI 规范来收集。

推断操作的 CRUD 语义。为推断操作的读取/写入语义，我们依据其 HTTP 方法进行判断，具体如下。我们假设 POST 方法用于创建资源（标注为 C），而 GET 方法用于读取资源。当返回架构类型为单个对象时，我们将读取单个资源标注为 R；当返回模式类型为对象数组时，我们将一次读取多个资源称为批量读取（标注为 RM）。此外，PUT 和 PATCH 方法被视为用于更新资源（标注为 U），而 DELETE 方法（标注为 D）被视为用于删除资源。

关于图 1中的示例，操作 `GET/users` 被标记为批量读取（RM），事实上，其返回类型是一个数组；而操作 `POST /users/register`被识别为创建（C）。

尽管 REST API 并不受必须遵循此映射关系的约束，但程序员通常会遵循这一惯例：反过来通常被视为一种反模式和不良做法 [8]。

按资源类型对操作分组。要检测只读属性，理解哪些操作以读取或写入模式处理同一类型的资源至关重要。事实上，如图1所示，操作定义并未说明这些操作处理的是用户资源。此外，处理同一类型资源的操作很少会针对完全相同的一组字段进行操作。例如，注册新用户有三个输入参数，而检索所有用户有四个输出参数。

为了将具有最相似参数的操作归为一组，我们使用聚类。我们收集 OpenAPI 规范中所有操作的输入/输出参数名称。使用 Porter 词干提取算法对这些名称进行规范化处理[9], ，仅保留单词的词根（例如，处理单数和复数形式）。之后，删除重复名称，并将其余名称按字母顺序排列，形成一个全局列表。最后，如表 I 所示，每个操作都编码为一个布尔值数组；当全局列表中的第 i 个参数是该操作的输入或输出参数时（根据 OpenAPI 规范判定），数组中的第 `true` 个元素设为 i。

然后，我们将期望最大化（EM）聚类算法应用于这种操作表示。EM 聚类算法会根据共同参数自动确定最佳聚类数量，以及每个聚类包含哪些操作。在我们的示例中，检索所有用户和注册新用户这两个操作很可能会被分配到同一个用户资源聚类中，因为它们的输入/输出参数集非常相似。

检测资源标识符。最后，仅仅了解哪些操作涉及同一资源类型（例如用户）还不够；我们还应检测在测试场景中的后续操作里是否使用了同一资源实例（例如同一个用户资源）。为此，我们检测充当资源（唯一）标识符的参数，并将其称为资源 ID；REST API 通常会使用这类参数。

我们采用一种简单但有效的启发式方法来识别资源标识符，该方法依赖于常见的命名惯例[4]。在某个操作的输入/输出参数中，我们将以 id 或 name 为后缀的字段视为资源标识符。如果多个字段符合这一简单模式，且模式中存在不同层级的嵌套，我们会选择嵌套层级较少的字段。如果发现多个字段处于同一层级，则优先选择以 id 为后缀的字段。如果仍无法唯一确定，则选择第一个字段。

在我们的示例中，所有针对 user 资源的操作都会将字段 username 选作资源标识符。

规范注解。一旦识别出 CRUD 语义、资源类型和资源 ID，就会将它们保存在 OpenAPI 本身中。为此，我们设计了一种注解语法，用于将这些信息添加到 OpenAPI 规范中。注解基于如下自定义扩展字段。

- x-crudOperationSemantics 用于为某个操作标注其 CRUD 语义。此字段可以是 create、read、update、delete、read-multi 或 none[5]。
- x-crudResourceType 用于标注操作所处理的资源类型。此注解接受字符串值。
- x-crudResourceIdentifier 用于标注在操作中用作资源标识符的参数。

或者，开发者可以手动为 OpenAPI 添加这些自定义标签，以跳过我们的自动检测。事实上，尽管指南和最佳实践为我们识别 CRUD 语义提供了依据，但对资源类型和资源 ID 的检测仍可能不够准确。

只读参数检测。收集完所有必要信息后，我们就可以开始实际识别只读参数。检查经过插桩的规范中的标签（通过聚类或手动检查），将处理同一类型资源的操作归为一组。只读参数就是那些在读取操作中作为输出提供（R 或 RM）

[4]https://restfulapi.net/resource-naming/

[5]注解 none 表示该操作不存在 CRUD 语义，因为它不涉及任何资源（例如，登录操作）。

的参数，但它们从未在同一组的创建操作（C）或更新操作（U）中用作输入。

在我们的示例中，管理员参数会被归类为只读字段，因为可以通过读取操作 GET /users 访问该参数，但无法通过写入操作 POST /users/register 修改它。

B. 测试用例生成

识别出只读字段后，我们需要生成确实尝试覆盖这些字段的执行场景。该测试方法依赖于采用以下记法的抽象测试模板。

- 对于类型为 τ 的资源，涉及τ资源的操作请求分别标记为 C_τ、 R_τ、 RM_τ、 U_τ 和 D_τ，分别表示旨在创建、读取、读取多个资源、更新或删除类型为 τ 的资源的请求。
- 当通过在操作请求 C_τ 中添加参数 f 来修改该请求时，会在操作请求上标注加号C_τ^{+f}；该参数未记录在 C 的规范中，添加它是为了尝试覆盖只读属性 f。
- 可选（但建议执行）的请求序列会标注问号。它们可用于清理 REST API 中的测试数据。

抽象测试模板被形式化为一系列标签，用于描述操作和资源类型。具体而言，我们确定了两种抽象测试模板：更新注入序列和创建注入序列。

模板 1：更新注入序列

$$< C_\tau, R_\tau, U_\tau^{+f}, R_\tau, (D_\tau, R_\tau)^? >$$

此序列旨在测试更新操作 U_τ 中潜在的批量赋值漏洞。序列从在 REST API 中创建资源开始。随后，对该资源执行读取操作，以检查资源是否已成功创建。第三个操作是更新操作U_τ^{+f}，其中注入只读参数 f 。接下来的读取操作 R_τ 旨在通过比较注入前后 f 的值，检查漏洞利用是否成功。如果只读参数发生变化，则表明存在批量赋值漏洞。

此外，可以删除已创建的资源，以清理测试副作用（即测试数据）。在这种情况下，最后一次读取操作会检查资源是否已成功删除，或者是否仍然存在（即删除操作失败）。

模板 2：创建注入序列

$$< C_\tau^{+f}, R_\tau, (D_\tau, R_\tau)^? >$$

此序列旨在测试创建操作 C_τ 中可能存在的批量赋值漏洞。序列中的第一个操作尝试通过指定一个额外的只读参数来创建资源。随后，一个读取操作会检查新建资源中的只读参数是否已成功被覆盖。

与前一种情况一样，序列末尾有两个可选请求，用于清理 REST API 的状态：删除测试资源，并检查是否已成功删除。

请注意，两个抽象测试模板中的最后两个请求（D_τ，R_τ）$^?$并未直接涉及批量赋值漏洞，实际上它们是可选的。例如，如果被测 REST API 中没有可用的删除操作 D_τ ，则必须跳过这个可选的清理序列。

从抽象测试用例到具体测试用例。抽象模板序列必须实例化，才能在被测 REST API 上执行。对于序列中的每个抽象操作，我们都必须生成相应的 HTTP 请求，包括输入参数的值。

输入值。请求的输入参数（资源 ID 除外）的值取自 OpenAPI 规范，或通过随机生成获得。OpenAPI 规范为每个参数提供一组示例值和参数默认值。此外，对于枚举参数，通常还会提供一组受支持的枚举值。若有这些值，我们便会采用。如果没有这些值，或者这些值无法生成成功的请求（例如，示例值不正确），我们就改用随机值，并确保这些值符合规范中定义的参数格式。对于未指定特定格式的字符串参数，我们会尝试根据参数名称推断其格式。例如，如果某个字段名为 email，我们会尝试生成有效的电子邮件地址。我们支持 20 多种字符串格式，包括日期、时间戳、电子邮件地址、电话号码、URL、UUID 等。

资源 ID 值。测试模板中的所有操作都应针对同一个资源实例进行操作。例如，创建资源后，后续的读取操作应访问刚创建的资源。为确保这一点，我们会检查创建操作 C_τ 的输出中的资源 ID 字段，并将其用作后续读取、更新和删除操作 R_τ、 U_τ和 D_τ的输入。由于序列中的所有操作使用的资源 ID 都相同，为了简化记法，我们没有注明每个操作处理的是哪个资源 ID。

然而，有时创建操作 C_τ的输出不包含新建资源的资源 ID，而该 ID 本应供后续读取操作 R_τ使用。为解决这一问题，只要条件允许，我们就会使用多资源读取操作 RM_τ ，在创建 C_τ前后列出类型为 τ 的所有资源。我们预计可以通过比较这两个列表，找出新创建的资源。两个抽象测试模板据此修改如下：

Template 1': $< RM_\tau, C_\tau, RM_\tau, U_\tau^{+f}, R_\tau, (D_\tau, R_\tau)^? >$
Template 2': $< RM_\tau, C_\tau^{+f}, RM_\tau, (D_\tau, R_\tau)^? >$

具体测试用例。从抽象模板实例化一个具体测试用例，需要成功执行相应序列中的每个操作。然而，

序列中的某一步可能有多个候选操作，例如，有多个操作 R_τ 可以读取类型为 τ 的资源。可以使用不同的输入值测试候选操作，直至其成功（状态码为 2XX）。测试一个操作的最大尝试次数（MAX_OPERATION_ATTEMPTS）可配置。如果所有尝试都无法获得成功响应，我们就认为整个模板实例化失败。系统会从头开始重新实例化模板，直至成功执行所有操作，最多尝试 MAX TEMPLATE ATTEMPTS 次。_ _

尝试次数越多，成功实例化序列的概率就越高，但测试生成所需的时间也会更长。

C. 安全判定器

安全判定器负责对测试用例执行进行分类，并判断它们是否能够揭示被测 REST API 中的批量赋值漏洞。为此，判定器会验证序列中的所有操作是否使用了相同的资源标识符，以及注入是否通过不同于默认值的值成功完成。下文将介绍判定器的这三个方面。

相同资源标识符检查。如前所述，测试场景中的所有操作按构造都应通过使用相同的资源 ID 字段值来处理同一个资源实例。安全判定器必须检查执行测试用例时是否满足这一约束。

覆盖只读字段。当判定器检测到只读值已被成功覆盖时，就会报告批量赋值漏洞。实际上，判定器会检查注入的更新请求 U_τ^{+f} 或注入的创建请求C_τ^{+f}是否能够覆盖只读参数 f 的值。具体做法是，将只读参数 f 的注入值与后续读取请求 R_τ 中观察到的值进行比较。如果两个值相同，则该测试用例成功写入了只读字段 f。

默认值注入。将抽象测试模板实例化为具体测试用例时，应为所有输入字段选择值。为只读参数选择值时，我们可能会无意中猜中其默认值，尤其是对于可选值集合有限的布尔值或枚举类型。如果后续读取操作发现资源的只读字段中包含注入值，我们的判定器可能会报告存在漏洞。然而，也有可能是攻击失败了，而只读参数的值与正常情况下的值相同，即其默认值。为检查这种情况并避免假阳性，我们会使用不同的只读参数注入值，重复执行判定器归类为注入成功的每个测试用例。我们的目标是确保至少使用两个不同的只读字段值成功完成注入，并避免因默认值导致假阳性。

识别其他缺陷。尽管该判定器旨在通过运行测试用例来揭示批量赋值漏洞，但它也能够检测其他类型的缺陷。

注入是指通过添加未记录的字段来违反规范的请求。因此，正确实现的 REST API 应返回 4XX 状态码，并拒绝包含意外参数f的格式错误的请求，因为该请求不符合已记录的请求格式。相反，2XX 状态码意味着 REST API 将注入的、因而无效的请求视为有效请求并予以接受。即使没有发生注入，仅接受格式错误的请求本身也是一种缺陷。此外，5XX 响应状态码表示未得到正确处理的内部服务器错误（例如，由未受控异常导致）。这表明可能存在实现缺陷，但这与批量赋值漏洞有所不同。

IV. 实验设置

本节介绍用于对我们的方法进行实证验证的实验设置。

A. 研究问题

首先，我们的方法需要识别操作的 CRUD 语义、这些操作处理的资源类型，以及它们用作资源标识符的参数。因此，我们首先通过实证研究评估此类信息自动识别的准确率。

RQ1：自动识别操作的 CRUD 语义、资源类型和资源标识符参数的准确率是多少?

其次，我们的方法旨在揭示 REST API 中的批量赋值漏洞，这一目标将通过以下研究问题进行探究。

RQ2：揭示 REST API 中批量赋值漏洞的准确率是多少?

最后，我们的方法应能够处理真实世界中可投入生产的 REST 服务，而这类服务可能既复杂又规模庞大。因此，最后一个研究问题探讨了该方法在大型 REST API 上的可扩展性。

RQ3：所提出的批量赋值漏洞检测方法能否扩展到大型 REST API?

B. 案例研究

实验验证在一组 REST API 上进行。然而，考虑到测试用例旨在利用安全漏洞，对公开托管的 API（例如 *APIs.guru*[6] 上的 API）进行测试并不合乎道德，因为成功攻击可能会损害其完整性。因此，我们选择了可以下载并在受控环境中运行的案例研究。

基于这些原因，我们寻找托管在 GitHub 上的开源项目，以便在本地编译和安装。这样我们也能完全控制 REST API 状态和数据，从而将 API 恢复到初始状态，并且，

[6]https://apis.guru/browse-apis/

表 II 用于评估该方法准确率的易受攻击 REST API。

案例研究	编程语言	框架	# 操作	# 漏洞
VAmPI [10]	Python	Flask	12	1
OWASP [11]	Java	Spring	10	4
Toggle [12]	ASP.NET	.NET Core	16	2
书店 [13]	Java	Spring	5	1
增删改查 [14]	Node.js	Express	4	2

因此避免先前注入所引发的潜在副作用。

我们使用以下基础字符串查询 GitHub 搜索引擎：“REST”、“RESTful API”、“批量赋值”、“自动绑定”、“对象注入”、“OpenAPI”和“Swagger”。随后，我们还添加了代表常用于实现 REST API 的框架的查询字符串，例如“swagger- ui”、“SpringFox”、“swagger-jsdoc”和“flask-swagger”。在搜索结果中的 REST API 里，我们选取了包含 OpenAPI 规范的 API，这是我们的黑盒测试方法所要求的。

我们下载、编译并运行了这些潜在案例研究，以剔除编译或运行失败的项目。经过这最后一轮筛选，最终选定的案例研究如表 II 所示。其中包含 5 个 REST API，使用不同的编程语言（Java、ASP.NET、Python 和 Node.js）编写，基于不同的框架和数据库管理系统，共有 26 个端点和 47 个操作。

该表还列出了各案例研究中的漏洞数量。VAmPI 和 OWASP 原本就包含一些漏洞（分别为 1 个和 4 个），而其他案例研究则没有，因此我们按照以下流程手动植入漏洞。我们编辑 REST API 中资源的内部表示（例如，资源 user），添加一个名称任意的新字段。然后，我们编辑处理此类资源的写入操作代码（例如，PUT /user），以启用自动映射。最后，我们编辑该资源类型的读取操作及其在 OpenAPI 规范中的相应部分，将新增字段加入可读取字段列表。

此外，我们还为所有案例研究准备了无漏洞（安全）版本。VAmPI 支持通过环境布尔变量启用或禁用漏洞。OWASP 的代码库同时包含易受攻击版本和修补版本。对于其余案例研究，我们修复了易受攻击的版本，并采用了框架的缓解功能。

最终，我们的方法要求以正确的 OpenAPI 规范作为输入。然而，在某些情况下，开源 REST API 的规范存在错误或不完整。因此，在运行实验之前，一组专家（包括本文的一位作者和两位外部贡献者）已对照源代码检查了所有 OpenAPI 规范。团队进行了一些修正，例如，在缺少响应模式时补充了这些模式。同一团队还为所有操作制定了 CRUD 语义的真实值、资源类型和资源标识符。

最后，为了回答第三个研究问题，我们考虑了一组更复杂的案例研究，即 10 个主流谷歌 REST API（Gmail、Google Analytics、Calendar、Classroom、Custom Search、Drive、Fitness、Google 我的商家、Search Console 和 YouTube）。

C. 实验流程

我们在 RestTestGen [1] 之上将我们的方法实现为一种测试策略 MassAssignmentSecurityTestingStrategy。RestTestGen 是一款开源的 REST API 测试工具（可在 GitHub[7]，上获取，您也可以在那里找到批量赋值策略），并提供了一些便利实用的功能，助力我们实现该方法。这些功能包括可靠的 OpenAPI 规范解析器、构造并发送 HTTP 请求至被测 REST API 的流程，以及接收和解析相应 HTTP 响应的流程。

我们将模板实例化的参数配置为以下值：MAX_OPERATION_ATTEMPTS =12，以及 MAX_TEMPLATE_ATTEMPTS = 3。在试运行实验中，这些值恰好在测试时间与成功识别漏洞之间实现了最佳权衡。

案例研究已被置于可测试状态，即使用一些数据对其状态进行了初始化，为测试做好准备。大多数案例研究都附带特定脚本，而另一些则需要进行若干次手动 HTTP 交互，以提供初始化数据。

我们的测试策略已在每个案例研究上执行，既针对易受攻击版本，也针对安全版本。该过程重复进行了十次，以控制我们的方法中非确定性算法所导致的潜在随机变化。每次测试会话结束后，我们都会将待测 REST API 恢复到完全相同的状态。

D. 指标

在实验中，我们计算并收集了以下指标。

CRUD 识别的正确性。对于每个 REST API，我们根据人工提供的真实值，计算正确识别出 CRUD 语义的操作数量与规范中具有 CRUD 语义的操作总数之比。

聚类正确性。我们手动检查聚类结果，以确定每个簇中的大多数操作所处理的资源类型。处理的资源类型不同于该多数类型的操作，被视为分配到了错误的簇。该指标的计算方式为：分配到正确簇的操作数除以 CRUD 操作总数。

资源 ID 识别的正确性。此指标的计算方式为：正确识别出

[7]https://github.com/SeUniVr/RestTestGen

资源 ID 字段的操作模式数量，除以包含资源 ID 的 CRUD 操作的模式总数。操作可以有输入模式和/或输出模式。

漏洞检测的准确率。为了衡量批量赋值漏洞检测的准确率，我们使用标准的信息检索指标。

- 真阳性（TP）：正确检测到的漏洞数量。
- 假阳性（FP）：被报告为漏洞、但实际上并非易受攻击的案例数量（误报）。
- 假阴性（FN）：被漏检的漏洞数量（工具未报告）。
- 精确率（Pr）：正确检测到的漏洞数量与报告的漏洞数量之比（Pr = TP / (TP + FP)）。
- 召回率（Re）：正确报告的漏洞数量与漏洞总数之比（Re = TP / (TP + FN)）。

E. 效度威胁

本文讨论影响研究结果效度的主要威胁。

结论效度威胁涉及影响得出正确结论能力的问题。为降低此类威胁，我们采用了标准的信息检索指标（即精确率和召回率）。此外，为了提高数据集中样本的异质性，我们纳入了使用多种编程语言编写、基于不同框架的软件项目。

内部效度威胁涉及可能影响结果的主观因素，例如用于比较我们方法结果的黄金标准定义。为降低这种主观性，我们检查了操作的源代码，以确定其实际的 CRUD 语义、读取/写入资源的类型，以及用作资源标识符的字段（如果有）。

构念效度威胁涉及数据收集和分析程序。为减轻此类威胁，我们确保在案例研究中注入故障时，严格遵循 OWASP 基金会所阐述的漏洞定义。

外部效度威胁关系到我们的研究结果能否推广。尽管我们分析了以不同编程语言编写、基于不同 REST 框架的 REST API，但由于我们只考虑了开源项目，分析结果仍可能存在偏差。我们的结果未必普遍适用于其他软件项目，例如闭源工业软件。只有通过包含更多样本的重复研究，才能确认我们的研究结果是否适用于实验设置之外的情况。

V. 实验结果

本节介绍实验验证的结果。本实验的复现包可在线获取 [15]。

表 III CRUD 语义识别、操作聚类和资源 ID 识别的正确性。

案例研究	**CRUD**	聚类	资源标识符
VAmPI	100%	100%	67%
OWASP	100%	80%	100%
Toggle	88%	88%	100%
书店	100%	100%	100%
CRUD	100%	100%	100%
平均值	98%	94%	93%

A. RQ1：CRUD 语义、资源类型和资源标识符的准确率

在开始实际的测试用例生成之前，应仅基于 OpenAPI 规范计算一些初步结果。第一个研究问题针对所有案例研究考察了这一步的准确率。我们将工具自动生成的标签与黄金标准进行了比较，结果见表 III。对于每个案例研究（第一列），表中报告了 CRUD 语义识别的正确性（第二列）、聚类正确性（第三列）以及资源 ID 识别的正确性（第四列）。

几乎所有案例研究中，CRUD 语义的识别准确率均为 100%，平均正确率达到 98%。如此高的准确率得益于所观察到的 REST API 对 HTTP 方法的规范使用。尽管 Toggle 的准确率仍然很高（即 88%），但其中一些操作的语义识别有误。经人工调查发现，由于返回类型不常见，多项读取操作被自动标记为读取。这些操作返回的不是对象数组，而是单个 JSON 对象，其中又包含许多对应于资源的子对象。

聚类正确性也相当高，平均值为 94%。被分配到错误聚类中的操作仅为删除操作。这是因为删除操作通常只需要待删除资源的资源标识符，无需其他输入。当针对同一资源类型的其他操作所需的输入参数远多于删除操作时，聚类算法很可能会将删除操作单独分配到一个聚类中（该聚类仅包含删除操作本身）。不过，我们的测试方法并不严格要求存在删除操作（模板 1 和模板 2 中的 D_τ 均为可选项）。

资源 ID 的识别也相当准确，平均正确率为 93%。检测不准确是因为某些案例研究中采用了不符合惯例的自定义命名。例如，VAmPI 使用字段 book_title 作为资源标识符，而不是更直观的 book_id 或书名。_

基于这些结果，我们可以这样回答 RQ1：

表 IV 揭示批量赋值漏洞的准确率。

案例研究	Safe		易受攻击					
	测试	FP	测试	TP	FP	FN	Pr	Re
VAmPI	4.0	0.0	4.0	1.0	0.0	0.0	100%	100%
OWASP	8.0	0.0	7.4	3.6	0.0	0.4	100%	90%
Toggle	2.0	0.0	2.0	2.0	0.0	0.0	100%	100%
书店	2.0	0.0	2.0	1.0	0.0	0.0	100%	100%
CRUD	2.0	0.0	2.0	2.0	0.0	0.0	100%	100%

我们的方法在分析 OpenAPI 规范时表现出较高的准确率：98% 的操作正确检测出了增删改查语义，94% 的操作在处理相同类型的资源时被归为一组，93% 的资源 ID 得到了正确识别。

B. RQ2：漏洞检测的准确率

然后，我们将测试方法应用于每个案例研究的两个变体，即存在漏洞和不存在漏洞的变体。我们将判定器对自动生成的测试用例的评估结果与黄金标准进行了比较。表 IV 从真正例（TP）、假正例（FP）、假负例（FN）、精确率（Pr）和召回率（Re）等方面展示了漏洞检测的准确率。该表还报告了每个案例研究的各个变体中检测到的只读字段所能成功生成的具体测试用例数量。表中报告的是每次测试会话重复十次的平均值（因此会出现小数值）。

批量赋值漏洞测试的准确率非常高。在几乎所有案例研究中，所有漏洞都被检测到，且没有假阳性。只有在 OWASP 的十次运行中有一次（因此表中的平均 TP 是小数），我们的方法漏掉了少数漏洞。对测试执行轨迹进行人工检查后发现，在 OWASP 的一次特定执行中，我们的方法只能实例化八个序列中的两个，因此无法测试相应的漏洞。

作为安全测试的副作用，我们的测试还可能发现其他缺陷，例如在除 Toggle 之外的所有案例研究中出现的崩溃（状态码 5XX）。总共出现了 24 个状态码为 500 的 HTTP 响应。通过人工检查 HTTP 请求和响应，我们发现它们源自 5 个不同的缺陷。

根据这些结果，我们可以得出对 RQ2 的回答：

使用自动化黑盒测试揭示批量赋值漏洞的准确率非常高，因为在我们的案例研究中，所有漏洞都被检测出来，且没有假阳性；唯一的例外是，在一项案例研究的十次执行中，有一次未能检测出来。

表 V 用于评估方法可扩展性的谷歌 REST API。

案例研究	# 操作	时间（秒）	# 只读字段
Gmail	68	3.0	23
Analytics	88	5.0	166
Calendar	37	2.0	11
Classroom	61	5.0	15
Custom Search	2	1.0	66
Drive	48	3.0	49
Fitness	13	1.4	4
Google 我的商家	50	7.4	527
Search Console	11	1.0	10
YouTube	76	8.4	110
总计	**454**	**37.2**	**981**

C. RQ3：方法的可扩展性

为了研究我们方法的可扩展性，我们对表中列出的 10 项主流谷歌服务的规范运行了静态分析，V，这些服务总共包含 454 个操作。为减轻聚类引入的非确定性的影响，我们对每项服务分别进行了 5 次测试。请注意，出于伦理方面的考虑，我们无法运行动态测试用例生成部分，也就是说，我们旨在避免对正在生产环境中运行的服务发起注入攻击并获得成功的风险。

对所有服务进行静态分析平均耗时 37.2 秒，总共发现了 981 个只读字段。虽然我们无法运行自动化测试用例生成部分，但我们预计，使用少量测试用例验证这些候选漏洞不会花费太长时间。因此，我们可以推测，对于大型 REST API，可扩展性不会成为我们方法的问题。

因此，我们可以这样回答 RQ3：

我们的方法似乎能够很好地扩展到大型 API：分析规模相当大的 OpenAPI 规范仅耗时 37.2 秒，并检测出 981 个只读参数（总计 454 个操作），这些参数可能是批量赋值漏洞的候选项。

VI. 相关工作

尽管大多数关于 REST API 自动化测试的文献都聚焦于功能测试，但旨在自动化安全测试、以检测 REST API 中潜在安全漏洞的方法也才刚刚开始涌现 [2]–[5]，。

具体而言，Mai 等人 [3] 使用蜕变关系来解决预言机问题：他们定义了 22 个与系统无关的蜕变关系，以自动化 Web 系统中的安全测试。他们的方法并非黑盒方法，也不针对 REST API。此外，批量赋值漏洞并未纳入其蜕变关系目录。

Atlidakis 等人 [2] 提出了四条安全规则，用于描述 REST API 的理想属性，旨在测试 Azure REST API 的若干安全相关方面。具体而言，他们的目标是测试以下四种情况：(i)

释放后使用，即资源被删除后仍可读取；(ii) 资源泄漏，即资源创建失败后仍可读取；(iii) 资源层级，即资源可通过错误的父资源访问；以及 (iv) 用户命名空间，即某个用户的私有资源可被另一用户访问。尽管这种方法有效，但它针对的是特定的安全问题，并未涵盖批量赋值漏洞。OWASP 基金会指出，批量赋值漏洞是最需要缓解的重要 REST API 漏洞之一。

Luo 等人 [4], [5] 关注访问控制。在他们的第一项工作中， [4], 他们的目标是将权限划分问题简化为 RESTful 函数分类问题。他们提出了一种基于遗传算法的 REST API 分类方法（ RestSep ）。在第二项工作中, [5], 他们提出了一种策略语言（RestPL），用于表达 REST API 的授权策略。RestPL 策略可以根据实际请求自动生成，从而减少对用户的干预。访问控制当然是测试工具应当关注的重要方面，但它与批量赋值缺陷无关。

据我们所知，目前还没有能够针对批量赋值漏洞自动测试 REST API 的方法。

其余关于功能测试的文献主要分为两个不同的研究方向。

其中一个方向是白盒方法，这类方法依赖 REST API 源代码，通过静态分析或对代码进行插桩来收集执行轨迹和度量值。对此，Arcuri [16] 提出了一种利用进化算法生成测试用例的全自动解决方案，该方案需要 OpenAPI 规范以及待测 REST API 的 Java 字节码。该方法已实现为名为 EvoMaster 的工具原型，并通过引入一系列新颖的可测试性变换加以扩展，旨在为常用 API 调用 [17] 提供指导。遗憾的是，在 REST API 场景中，白盒方法往往并不实用，因为通常无法获取源代码。

另一方面，黑盒方法不需要任何源代码，而在使用闭源组件和库时，这往往正是实际情况。模糊测试工具 [18]–[22] 是黑盒测试工具，它们从先前记录的 API 流量出发生成新的测试：通过对新的流量进行模糊测试并重放，以查找缺陷。其中一些工具还会利用被测服务的 OpenAPI 规范 [19]–[21]。

Godefroid 等人 [23] 提出了一种方法，利用 JSON 请求体架构和高级模糊测试规则，对请求体载荷进行智能模糊测试。尽管它们是自动化黑盒工具，但其目标是为测试生成输入值，因此无法作为独立的测试工具使用（Godefroid 等人 [23] 提出的方法除外，该方法已在 RESTler 中实现）。

Ed-douibi 等人 [24] 提出了一种基于模型的方法，用于 REST API 黑盒自动测试用例生成。模型从 REST API 的 OpenAPI 规范中提取，用于生成正常测试用例（其输入值符合模型）和故障测试用例（其输入值违反模型）。

Karlsson 等人 [25] 提出了 QuickREST，这是一种用于 RESTful API 属性测试的工具。他们从 OpenAPI 规范出发生成测试用例，旨在验证被测 API 是否符合规范中记录的某些属性（即定义），例如状态码和模式。

Segura 等人 [26] 提出了另一种用于 REST API 测试的黑盒方法，其判定器基于请求与响应之间的（蜕变）关系。例如，他们向同一个 REST API 发送两条查询，其中第二条查询的条件比第一条更严格（例如，通过添加约束）。第二条查询的结果应当是第一条查询结果中条目的真子集。如果结果并非子集，判定器就会揭示一个缺陷。不过，这种方法仅适用于面向搜索的 API。此外，该技术只能实现部分自动化，因为用户需要手动识别要利用的蜕变关系以及要测试的输入参数。

Corradini 等人 [1] 提出了 RestTestGen，这是一款面向 REST API 的自动化黑盒测试用例生成工具。该方法支持测试正常场景和错误场景，其测试生成策略基于操作依赖图；该图对 API 中可用操作之间的数据依赖进行编码。

Atlidakis 等人 [27] 提出了 RESTler，这是一款由微软研究院开发的有状态 REST API 模糊测试工具。RESTler 通过推断规范中描述的不同请求类型之间的生产者-消费者关系，生成有状态请求序列。它还会动态分析响应，以便智能地构建请求序列，并避免生成会导致错误的序列。

Martin-Lopez 等人 [28] 提出了 RESTest，这是另一款用于 RESTful API 的自动化黑盒测试工具。该工具的特点是支持参数间依赖。事实上，一些 REST API 不仅会对输入值施加约束，还会限制输入值的组合方式，以便构造有效请求。

最后，Laranjeiro 等人 [29] 提出了 bBOXRT，这是一款用于 RESTful API 的黑盒健壮性测试工具。bBOXRT 旨在通过观察被测服务在收到无效请求时的行为，评估 REST API 的健壮性。该工具提供了一个故障模型，其中包含 57 种不同的变异，可应用于各种类型的输入参数（数字、字符串、布尔值、日期、时间、数组等）。

上述所有 REST API 黑盒测试工具都不是为了发现安全漏洞，尤其不是为了发现批量赋值漏洞。不过，可以扩展这些工具，以实现本文提出的方法。为此，我们选择在 RestTestGen 的基础上应用我们的测试策略，因为它提供了模块化且易于扩展的架构。

VII. 结论

数据的可靠性和机密性取决于那些旨在实施适当数据访问策略的 REST API 是否得到了正确且安全的实现。批量赋值是 REST API 中最常见的漏洞之一，通常由 Web 框架配置错误导致。这些漏洞可能使攻击者能够直接覆盖 REST API 后端中的私有内部数据结构，例如敏感的数据库列。

我们提出了一种基于黑盒测试的自动化方法，用于揭示批量赋值漏洞。该方法从 OpenAPI 规范出发，通过聚类识别很可能作用于同一资源的操作。我们比较各簇的内容，以确定那些不应在写入操作中公开的资源字段，即根据开发者意图应为只读的数据。随后，我们将抽象测试模板转化为具体测试用例，针对每个疑似只读字段检测批量赋值漏洞。据我们所知，这是首项提出利用自动化黑盒测试揭示批量赋值漏洞的工作。

实验结果表明，如果未对 Web 框架进行适当配置，各种编程语言都容易出现这类漏洞。作为未来工作，我们计划将实验验证扩展至更完整的开源项目集合（涵盖更多编程语言和 Web 框架），以及工业合作伙伴提供的闭源工业项目。此外，我们计划将该方法扩展到 REST API 的其他编程缺陷和安全漏洞，例如对象级授权失效和访问控制错误。最后，我们计划与开发者开展用户研究，以确定报告 REST API 漏洞的最佳方式，从而支持快速、准确地修复这些缺陷。

参考文献

[1] D. Corradini、A. Zampieri、M. Pasqua、E. Viglianisi、M. Dallago 和 M. Ceccato，《RESTful API 中名义场景和错误场景的自动化黑盒测试》，Software Testing, Verification and Reliability，1月。2022 [ ] 10 1002 1808 。在线。获取地址：

[2] https://doi.org/ 。 /stvr。V. Atlidakis、P. Godefroid 和 M. Polishchuk，《检查云服务 REST API 的安全属性》，载于第13届 IEEE 软件测试、验证与确认国际会议，ICST 2020，葡萄牙波尔图，2020年10月24日至28日。IEEE，2020年，第387–397页。 [ ] 10 1109 46399 2020 00046 在线。获取地址：

[3] https://doi.org/ 。 /ICST 。 。P. X. Mai、F. Pastore、A. Goknil 和 L. Briand，《面向 Web 系统的蜕变安全测试》，载于2020年第13届 IEEE 软件测试、验证与确认国际会议（ICST），2020年，第186–197页。[4] Y. Luo、T. Puyang、X. Sun、Q. Shen、Y. Yang、A. Ruan 和 Z. Wu，《RestSep：面向 RESTful API 的测试导向权限划分方法》，载于2017年 IEEE Web 服务国际会议，ICWS 2017，美国夏威夷州檀香山，2017年6月25日至30日，I. Altintas 和 S. Chen 编。IEEE，2017年，第548–555页。 [在线]。获取地址：https://doi.org/ 10 1109 2017 64 。 /ICWS. 。Y. Luo、H. Zhou、Q. Shen、

[5] A. Ruan 和 Z. Wu，《RestPL：面向任意 RESTful API 的面向请求的策略语言》，载于IEEE Web 服务国际会议，ICWS 2016，美国加利福尼亚州旧金山，2016年6月27日至7月2日，S. Reiff-Marganiec 编。IEEE 计算机学会，2016年，第666–671页。[在线]。获取地址：https://doi.org/10.1109/ICWS.2016.92

[6] R. T. Fielding，架构风格与基于网络的软件架构设计。加利福尼亚大学欧文分校博士论文，2000年，第7卷。[7] OWASP 基金会，《批量赋值测试》，2022年，https://owasp.org/www-project-web-security-testing-guide/latest/4-WebApp _ 应用程序

[8] F.Palma、J. Dubois、N. Moha 和 Y.-G. Gu´eh´eneuc，《REST 模式与反模式的检测：一种基于启发式方法的方法》，载于 国际服务导向计算会议。施普林格，2014年，第 230–244 页。[9] P. Willett，《Porter 词干提取算法：过去与现在》，*Program*，2006年。[10] github.com/erev0s，《VAmPI 案例研究》，2022年，https://github.com/erev0s/VAmPI。 [11] github.com/mattiasanti99，《OWASP 案例研究》，2022年，https://github.com/mattiasanti99/vulnerability OWASPproject。 [12] github.com/pdonatilio，《Toggle 案例研究》–，2022年，https://github.com/pdonatilio/ToggleAPI。 [13] github.com/todo，《书店案例研究》，2022年，https://github.com/todo/Bookstore。 [14] github.com/lucianopereira86，《增删改查案例研究》，2022年，https://github.com/lucianopereira86/CRUD-NodeJS-Sequelize-Swagger-MySQ L. [15]复现包，《RESTful API 批量赋值漏洞的自动化黑盒测试》，2022年，https://doi.org/10.5281/zenodo.7348518。[16] A. Arcuri，《使用 Evo-master 自动生成 RESTful API 测试用例》，ACM 软件工程与方法学汇刊（TOSEM），第28卷，第1期，第3页，2019年。[17] A. Arcuri 和 J. P. Galeotti，《现有 API 的可测试性变换》，载于第13届 IEEE 软件测试、验证与确认国际会议（ICST 2020），葡萄牙波尔图，2020年10月24日至28日。IEEE，2020年，第153–163页。 [在线]。参见：https://doi.org/ 10 1109 46399 2020 00025 。/ICST。. API 模糊测试器。[18] 2022 （） API 模糊测试器。https://github.com/KissPeter/APIFuzzer。[19] Fuzz-Lightyear，《Fuzz-Lightyear》，2022年，https://github.com/Yelp/fuzz-lightyear。[20] Fuzzy-Swagger，《Fuzzy-Swagger》，2022年，https://github.com/namuan/fuzzy-swagger。[21] Swagger-Fuzzer，《Swagger-Fuzzer》，2022年，https://github.com/Lothiraldan/swagger-fuzzer。[22] TnT-Fuzzer，《TnT-Fuzzer》，2022年，https://github.com/Teebytes/TnT-Fuzzer。[23] P. Godefroid、B. Huang 和 M. Polishchuk，《智能 REST API 数据模糊测试》，载于 ESEC/FSE '20：第28届 ACM 欧洲软件工程会议与软件工程基础研讨会联合会议，线上会议，美国，2020年11月8日至13日，P. Devanbu、M. B. Cohen 和 T. Zimmermann 编。ACM，2020年，第725–736页。 [ ] 10 1145 3368089 3409719 在线。参见：

[24] https://doi.org/。/。H. Ed-Douibi、J. L. C. Izquierdo 和 J. Cabot，《REST API 测试用例的自动生成：一种基于规范的方法》，2018年第22届 IEEE 国际企业分布式对象计算会议（EDOC），第181–190页，2018年。[25] S. Karlsson、A. Causevic 和 D. Sundmark，《QuickREST：为基于 OpenAPI 描述的 RESTful API 进行基于属性的测试生成》，载于2020年第13届 IEEE 软件测试、验证与确认国际会议（ICST），2020年，第131–141页。[26] S. Segura、J. Parejo、J. Troya 和 A. Ruiz-Cort´es，《RESTful Web API 的蜕变测试》，IEEE 软件工程汇刊，第44卷，第11期，第1083–1099页，2018年。[27] V. Atlidakis、P. Godefroid 和 M. Polishchuk，《RESTler：有状态 REST API 模糊测试》，载于第41届国际软件工程会议论文集，系列：ICSE '19。美国新泽西州皮斯卡塔韦：IEEE 出版社，2019年，第748–758页。 [在线]。参见：https://doi.org/ 10 1109 2019 00083 。/ICSE。. A.

[28] Martin-Lopez、S. Segura 和 A. Ruiz-Cort´es，《RESTest：RESTful Web API 的黑盒基于约束测试》，载于服务导向计算——第18届国际会议（ICSOC 2020），阿拉伯联合酋长国迪拜，2020年12月14日至17日，论文集，系列：计算机科学讲义，E. Kafeza、B. Benatallah、F. Martinelli、H. Hacid、A. Bouguettaya 和 H. Motahari 编，第12571卷。施普林格，2020年，第459–475页。 [在线]。参见：https://doi.org/ 10 1007 978 3 030 65310 1_33 。/----- N.

[29] Laranjeiro、J. Agnelo 和 J. Bernardino，《用于 REST 服务健壮性测试的黑盒工具》，IEEE Access，第9卷，第24 738–24 754页，2021年。 [在线]。参见：https://doi.org/10.1109/ACCESS.2021.3056505