计算机科学
COMPUTER SCIENCE

基于黑盒插桩的闭源数据库管理系统的模糊测试技术研究

李忠杰, 梁皓天, 贾浩阳, 王清贤, 曹琰

引用本文

李忠杰, 梁皓天, 贾浩阳, 王清贤, 曹琰. 基于黑盒插桩的闭源数据库管理系统的模糊测试技术研究[J]. 计算机科学, 2026, 53(2): 133-144.

LI Zhongjie, LIANG Haotian, JIA Haoyang, WANG Qingxian , CAO Yan. Research on Fuzz Testing Techniques for Closed-source DBMSs Based on Black-box Instrumentation [J]. Computer Science, 2026, 53(2): 133-144.

计算机科学 COMPUTER SCIENCE
http://www.jsjkx.com
DOI:10.11896/jsjkx.241200060

基于黑盒插桩的闭源数据库管理系统的模糊测试技术研究

李忠杰[1] 梁皓天[1] 贾浩阳[1] 王清贤[1] 曹 琰[1,2]
1 郑州大学网络空间安全学院 郑州 450002
2 嵩山实验室 郑州 450000
(aizawanachu@gmail.com)

摘 要 数据库管理系统(Database Management Systems,DBMSs)是被广泛用作管理业务数据的应用软件,其安全性至关重要,任何形式的数据泄露或损坏都可能导致重大安全问题。目前针对闭源DBMS漏洞检测的公开研究成果相对较少。为了实现闭源DBMS的有效测试,提出了基于文法结构的变异和基于语义规则的变量填充的方法来批量生成测试数据集,根据提供的原始语料生成语法和语义高度正确的复杂SQL查询,使得输入数据能够深入探索DBMS的深层逻辑;同时提出了基于Pin的动态覆盖率分析方法来收集闭源DBMS的实时覆盖率,根据覆盖率反馈指导模糊测试的种子调度。基于上述方法实现了面向闭源DBMS的自动化测试原型工具OFuz。使用OFuz对Oracle和SQL Server两种DBMS进行测试,实验结果验证了该工具的有效性,在生成测试集和覆盖统计方面相比其他工具效果更优。

关键词:数据库安全;闭源数据库管理系统;黑盒测试;模糊测试;覆盖率统计
中图分类号 TP309

Research on Fuzz Testing Techniques for Closed-source DBMSs Based on Black-box Instrumentation

LI Zhongjie[1],LIANG Haotian[1],JIA Haoyang[1],WANG Qingxian[1] and CAO Yan[1,2]
1 School of Cyberspace Security and Engineering,Zhengzhou University,Zhengzhou 450002,China
2 Songshan Laboratory,Zhengzhou 450000,China

Abstract DBMSs are widely used application software for managing business data,and their security is critical. Any form of data leakage or corruption could lead to significant security issues. Currently,there are relatively few public research findings on vulnerability detection for closed-source DBMSs. To enable effective testing of closed-source DBMSs,a novel approach has been developed. It proposes methods based on grammar structure mutation and semantic rule-based variable filling to generate test datasets in batches,creating syntactically and semantically correct complex SQL queries from provided raw corpora. These inputs allow for in-depth exploration of the deep logic of DBMSs. Additionally,a dynamic coverage analysis method based on Pin is introduced to collect real-time coverage data for closed-source DBMSs,using feedback from the coverage to guide seed scheduling in fuzz testing. Based on these methods,an automated testing prototype tool for closed-source DBMSs,named OFuz,has been developed. Experiments conducted on Oracle and SQL Server validate the effectiveness of OFuz,demonstrating superior performance in test dataset generation and coverage analysis compared to other tools.

Keywords Database security,Closed-source database management systems,Black-box testing,Fuzz testing,Coverage analysis

1 引言

数据库管理系统(DBMS)是每个企业都会使用到的管理各类数据资料的软件。近年来,企业对数据安全的关注愈加频繁,保障企业的数据安全成为必不可少的环节。2016年由MySQL的漏洞引发的Friend Finder Network事件导致超过4亿用户信息被泄露;2017年的MongoDB勒索攻击事件使得大量企业的数据丢失和业务中断[1];2019年因内部DBMS的漏洞导致的Desjardins Group数据泄露事件使得近300万名客户的个人信息可以在未经授权的情况下被访问。这些事件都表明了DBMS潜在漏洞的危害性。为了防止此类恶意行为,及时发现并且修复DBMS中的漏洞,各大DBMS厂家都通过独立的团队定期对自己的产品做全面的测试工作,同时也会获取用户的意见从而诊断和修复潜在的漏洞,定期将补丁发送给客户群体[2]。

模糊测试属于一种常见的漏洞挖掘方法[3]。著名的模糊

到稿日期:2024-12-09 返修日期:2025-03-18
基金项目:嵩山实验室资助项目(232102210124,ZZK202403002-03);河南省科技攻关项目(232102210124)
This work was supported by the Songshan Laboratory-Funded Project(232102210124,ZZK202403002-03) and Henan Province Science and Technology Research Project(232102210124).
通信作者:曹琰(ieycao@zzu.edu.cn)

测试框架 AFL 已经证实，依靠这样的随机输入的测试方法能够发现很多软件中以前从来没有被人为找到的潜在漏洞。

模糊测试已经被应用于测试 DBMS，例如 SQLsmith，Squirrel[4]和 DynSQL[5]等虽然实现技术各不相同，但都可以通过不断生成 SQL 查询语句来检查系统的内存错误；SQLancer[6]和 Troc[7]使用基于事先创建的预言机来检测 DBMS 的逻辑错误和事务隔离性错误；APOLLO[8]通过对多个不同版本的 DBMS 进行性能对比来检测性能回归问题；Amoeba[9]基于等价语义也可以挖掘 DBMS 中的性能 bug。此外，还有 Grand[10]和 GDsmith[11]这种专门用于图形类 DBMS 的测试工具，甚至近期还诞生了如 BUZZBEE[12]等可以尝试用于非关系型 DBMS(如 redis)的测试工具。这些研究成果帮助开发人员找到了不同类型 DBMS 的各式各样的深层 bug 和漏洞[13]。但是这些测试工具都仅可应用于开源 DBMS 的测试任务，尝试将这些工具应用于闭源 DBMS 的漏洞检测就会出现一些问题。首先，每一种 DBMS 的输入空间是符合其自身方言的 SQL 查询，而前面所列举的这些工具所生成的输入都只支持一些特定的开源 DBMS 软件，最常见的如 MySQL，PostgreSQL 等，仅靠这些生成数据无法探测 Oracle 和 SQL Server 等 DBMS 的深层逻辑。其次，在模糊测试中存在着一项重要的“反馈机制”[14]，用于指导每轮次测试开始时选取的种子优先级。由于大多数 DBMS 测试工具的测试对象都是开源 DBMS，因此可以利用一些插桩技术，使用提供的定制编译工具来编译目标程序的源码，例如 AFL++[15]中的 afl-gcc 编译器直接可以用来编译 C 程序，Angora[16]使用 LLVM 的 Pass 来实现上下文敏感的分支计算等。这类定制编译工具会在目标程序中进行汇编指令层面的插桩，从而可以统计出程序执行流程中涉及到的基本块或者路径的数量，根据这个指标来判断测试样例的表现。

但上述方法都是针对开源数据库 DBMS 系统，尚无针对闭源数据库 DBMS 系统的公开研究成果。为了能够使用模糊测试来检测 Oracle，SQL Server 等闭源 DBMS 中的漏洞，需要解决两大难点：1)如何构造有效且复杂程度足够高的 SQL 测试集[17]；2)如何实时统计目标 DBMS 的覆盖率。针对这两个问题，本文提出了对应的解决方案。具体地，使用基于文法结构的变异和基于约束规则的变量填充来生成语法语义正确的 SQL 查询，进而探索 DBMS 的输入空间；使用 Intel Pin 来统计黑盒测试环境下 DBMS 软件的执行路径信息。在此基础上，开发了一个闭源 DBMS 模糊测试系统，在 Oracle 11g 和 SQL Server 2016 上对其进行测试，并评估了实验结果。

2 问题分析

传统模糊测试技术具有盲目性，不适合直接测试 DBMS。DBMS 对于接收的输入数据有着严格的校验，这些校验规则会判断输入数据是否是符合语法规范的正常 SQL 查询，只有正确的 SQL 查询才会被 DBMS 送往深层处理并执行[18]。通常来说，DBMS 执行 SQL 查询时是按照语句之间的分隔符(如分号“;”)先将查询分为一条一条的单独语句再串行执行，而 SQL 语句的单独执行可以分为解析、优化、评估和执行 4 个阶段[19]。

在解析阶段，DBMS 会严格地对 SQL 语句的语法和语义进行校验，如果有任何的解析错误就会返回错误消息并且终止语句的执行，解析无错误则会生成抽象语法树，抽象语法树描述了语句的结构和内容[20]。之后进入优化阶段，DBMS 的优化器会将语句进行简化或者重写来实现进一步改进。在评估阶段就会生成多个执行方案，由 DBMS 根据当前环境来判断此时最优的方案，最后执行。

另外，在模糊测试中如何准确地获取执行路径的信息来不断指导后续测试也是一个重要的方面[21]。例如，使用 AFL 进行测试前，需要利用 AFL 工具集中的定制编译器(afl-gcc 和 afl-g++)来编译目标程序的源码。而闭源 DBMS 是一个完整且封闭的产品，如 Oracle 产品交由用户安装时也没有可以手动进行编译的机会，用户想要启动服务只需要使用 service命令运行事先安装好的脚本文件即可，真正的启动过程是由脚本文件按顺序依次启动各类组件，用户无法像编译插桩那样实时获取软件的执行信息，也就没有反馈信息来正确指导模糊测试的流程。

2.1 生成测试集

在 SQL 查询执行的整个流程中，解析阶段控制着 SQL 语句是否能够进入到 DBMS 深层逻辑，只有符合 SQL 语法和语义的输入才能构建抽象语法树并进入之后的处理逻辑[22]。解析阶段一共有两层校验，即文法校验和语义校验。文法校验判断输入语句中的单词是否合法，以及该语句是否是所属语言中的合法句型。如图 1 所示，假如一条正常的语句“SELECT c1 FROM t WHERE c2 > 1”在进入文法解析阶段时，会先被分为多个单词：SQL 关键词“SELECT”“FROM”和“WHERE”，变量“c1”“t”和“c2”，运算符“>”以及字面量“1”。这些单词都是符合规范的。

图 1 SQL 语句分解示例

Fig. 1 Example of SQL statement decomposition

然后对其进行句型的匹配，发现这条 SQL 语句中有 SELECT 子句、FROM 子句和 WHERE 子句。其中 SELECT 子句和 FROM 子句都是仅由一个 SQL 关键词加一个单独变量构成的子句，而 WHERE 子句中包含 SQL 关键词和一个比较大小的表达式，句子中的所有元素都符合 SQL 文法中的句型，因此这条语句就可以通过语法校验。

之后，这条语句会被进行语义解析，此时 DBMS 会结合当前的运行环境以及事先定义好的约束规则来判断语句是否合法[23]。例如“t”表中是否有字段“c1”和“c2”，字段“c2”的类型是否可以与常量“1”进行比较等。如果这些约束都满足，那么 DBMS 就会判断这条语句是一个合法的输入，就会继续执行后续的优化、评估和执行阶段。综上所述，若尝试对 DBMS 进行深度检测就必须使得送往程序的输入查询顺利通过解析阶段，即测试系统必须能够生成文法和语义都正确无误的 SQL 查询。

2.2 覆盖率统计

在使用 AFL++进行模糊测试的大多数情况下，都是在

拥有目标程序的源代码的前提下实施测试的。在源码的编译阶段，会进行汇编指令层面的插桩[24]，以此来收集运行过程中程序的覆盖率并将其作为指导。

本系统同样是基于 AFL++作为驱动框架开发的，但测试闭源 DBMS 时没有办法实行白盒测试那样的指令插桩。AFL++虽然提供了 Qemu 的运行模式，但 AFL++产生的数据不能直接发送给 DBMS，因此在整个模糊测试系统中，设计了一个单独的客户端进程用于协调两个进程之间的数据传送。

总体启动的流程为：在终端启动 afl-fuzz 后，AFL++按照计划读入种子文件，调用基于 AFL++开发的其他功能组件完成整个变异操作，生成完整的测试样例，然后发送给 DBMS 执行，得到执行结果后重复新一轮的测试。而单独设计的 DBMS 通信中间件（以下简称为中间件）就充当从 AFL++获得 SQL 查询发送给 DBMS 并且从 DBMS 获得结果返回给 AFL++的角色，如图 2 所示。

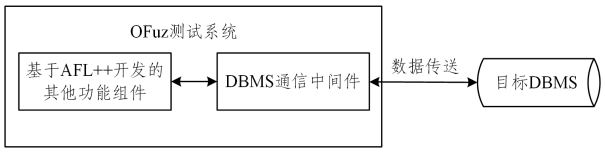

图 2 OFuz 中的通信中间件

Fig. 2 Communication middleware in OFuz

按照上述计划，一开始 AFL++启动时 afl-fuzz 的输入参数中的目标程序应该是中间件的可执行程序的位置，但是 AFL++的 Qemu 模式下默认只会对输入参数中的目标程序统计其覆盖率信息，在程序的运行过程中表现为在 AFL++检测到有“-Q”参数的输入时，会重新组建一行命令，然后将新的命令行传给内部的 Qemu 组件启动运行[25]，导致中间件的启动以及覆盖率统计都是在 Qemu 组件中进行，仅在 AFL++中无法进行细粒度的分隔。Qemu 进程的输入参数中仅接收了中间件的可执行文件，因此在启动之后也仅仅统计中间件的覆盖率，这与一开始想要统计 Oracle 的路径执行信息的目的也不相符。

为了解决前文提到的两个主要问题，分别提出了对应的解决方案。使用基于文法结构的变异和基于语义规则的变量填充保证生成的 SQL 查询的文法和语义的正确性，使用 Pin Tool 去探测 DBMS 系统运行过程中的执行路径信息。

3 基于文法结构的变异

虽然不加任何限制也可以直接使用 AFL++的模糊策略，但是 AFL++的确定性变异和非确定性变异都会破坏输入种子的原有结构，若种子是 SQL 查询则就会导致其完全不能正常运行。AFL++默认是面向所有类型的应用程序的，因此在变异过程中其不关心输入数据的类型。即使是最简单的 SELECT col FROM tab，在使用内部策略随机变异时也会进行随机的字节或字符替换，处理之后的结果可能就变成了 SFLECT col FROM tab。仅仅一个字符的改变，就会使得这条 SQL 语句不能再正常执行，因为 DBMS 不能识别一个拼写错误的关键词“SFLECT”。

模糊器的默认变异策略专注于改变输入种子的字符甚至字节层面，对于有着高度文法结构的语言来说，这种行为会很大概率破坏原有的文法结构，导致其不再是一条正确的语句[26]。如果能把变异策略施加的对象改为更上面的抽象出的文法层次，就可以很大程度地增加变异处理之后的语句的正确性。

为了实现这一目的，首先需要设计一个抽象层（Abstract Representation）来描述语句的文法结构，为了方便描述，将其简称为 ABR。ABR 需具备两个特性：等价性和简便性。等价性指该抽象层必须与 SQL 查询有着相同的表现力，使得 ABR 与 SQL 查询可以进行无损失的互相表达，而简便性要求 ABR 与 SQL 查询互相转换时尽可能简单。

一条简单的 ABR 包含一个操作符和至多两个操作数，这种简单的表示形式使得在对 ABR 进行增加、删除和替换等操作时只需要对其操作数做相应修改即可。一般来说，每条 SQL 语句可以分解为多条 ABR，每条 ABR 的左右操作数仍然为另外的 ABR。因此，在实际工作中，ABR 整体被设计为一种树状结构，每个节点使用以下成员属性存储必要的值。

1）left_ABR，right_ABR：它们分别代表树中的左右子节点，用来存储当前 ABR 的左右操作数，操作数的本质同样也是 ABR。

2）op：是当前 ABR 的操作符，操作符的值在解析文法结构时获取，包含 SQL 语言中的关键词和算术运算符等，操作数连接上操作符就构成了 ABR 的基本结构。

3）abr_t：用来表示 ABR 的类型。ABR 的类型也是由其文法结构决定的，其数值取自对 SQL 查询进行文法解析时，生成的对应 AST 的节点。例如，一个 SQL 查询（包含了一组由分隔符隔开的多条 SQL 语句）的 abr_t 值为 StmtList，而单独选择的语句的 abr_t 的值为 SelectStmt。

4）data_t：存储基本数据的类型。当 SQL 语句分解为不可再分的 ABR 时，就需要存储基本数据的信息，如表示字段的 ColumnName 和表示表的 TableName。

5）value：存储基本数据的值，如 SQL 语句中的变量和字面量的数值等。

图 3 展示了以 SQL 查询 SELECT c1 FROM t1，t2 WHRER c2＞1 为例，分解为 ABR 的操作。

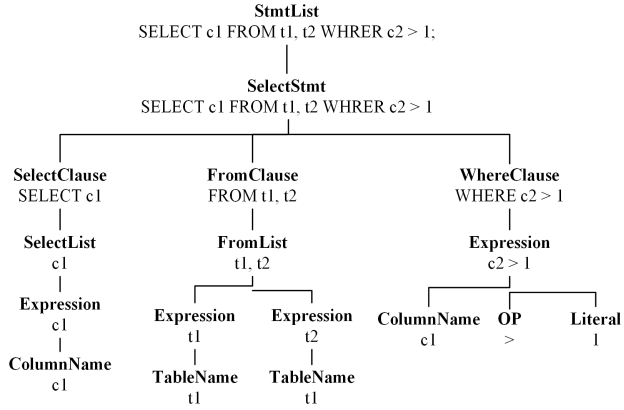

图 3 SQL 语句转化为 ABR 的示例

Fig. 3 Example of SQL statement transformation to ABR

在图 3 所示的例子中，SQL 查询中只包含一条语句，类型为 SelectStmt，然后将整条语句的各个子句进行细分，使得

语句中的每个部分都能构建属于自身的 ABR。在该分解图中，SelectStmt 这个类型的 ABR 一共拥有 3 个子节点，而之前说明的 ABR 节点只有左右两个操作数。这是因为在实际操作中，该例子中的 SelectStmt 的左操作数依然是 SelectClause，但是右操作数的 ABR 类型命名为“复合类型”，ABR_t 的值为 Component，而在 Component 节点中又单独包含了 FromClause 和 WhereClause 类型的 ABR。SQL 查询中包含了各种类型的结构，在遇到父节点分解为多个地位相同的子节点的情况时，都采用这样的复合节点来表示。

以这样方式的 ABR 来表示原本的 SQL 查询可以展示出相同的表现力，而且将 SQL 查询和 ABR 相互转换也属于比较方便的操作。但若按照这个思路进行工作，则需要有一个专门用来解析 SQL 文法的工具。目前对于 SQL 文法解析的研究大部分都停留在以开源 DBMS 为对象的程度，主要原因是开源 DBMS(如 MySQL，PostgreSQL 等)的代码库中有官方发布的文法解析器，这为其他用户提供了很大便利，用户想要开发出适配自己工程的解析工具，可以直接参考官方给出的各种文法规则。而不同方言的 SQL 文法差异比较大，对于商用 DBMS 的 Oracle 和 SQL Server 来说，就不存在官方给出的文法规则，也就不太容易构建出对应的文法解析器。

为了解决这个问题，实际工作中收集了 Oracle 和 SQL Server 的各种类型的常用语句，然后分析这些语句，总结了各种规则，借助 ANTLR4 等辅助工具，成功实现了可以正确解析大部分常规语句的文法解析器。ANTLR4 需要用户事先写好自定义的文法规则来生成词法和语法解析器，一共包含 lexer 和 parser 两个文件，并且都以 g4 扩展名结尾。其中 lexer 的作用是将输入的文本分割为单词，parser 的作用是使用单词构建语法结构。SQL 语言结构整体是按照 Program-StmtList-Stmt 的层级进行划分的，而 Stmt 中就包含了各个种类的单条语句，例如简单的 select 语句的语法规则可以表示为：

```
select_stmt
    :SELECT(DISTINCT | UNIQUE | ALL)? selected_list from_clause? where_clause?
        group_by_clause? order_by_clause?
```

其中不仅有 select 子句自身的成分，又可能包含 from 子句和 where 子句等。这样的语法规则最终会被单词和句型匹配识别，在完成 lexer 和 parser 之后，借助 ANTLR4 就可以将这两个文件编译为 C++程序可用的源码文件，分别代表词法和语法解析器。

基于 ABR 的设计，提前准备好输入的种子(为常规 SQL 查询)后，模糊测试的过程中变异操作就不再采用 AFL++默认使用的变异器，取而代之的是针对 DBMS 方言的基于文法结构的变异策略。所有的 SQL 查询在经过语法解析器的处理之后，都会转化为 ABR 的树形结构。这里，在进行变异操作之前，还需要先对输入的 SQL 查询进行预处理，主要是为了在变异过程中将 SQL 查询的文法和语义分开。这一阶段中重点关注文法结构，而对于语义方面的处理将在之后的流程进行。预处理的操作即为提取 ABR 中的所有变量，并且用占位符进行替代。前述例子的 SELECT c1 FROM t1，t2 WHERE c2>1 在预处理(本质是提取 ABR 的变量，因为 ABR 和 SQL 的表现力相同，为了方便就使用 SQL 语句为例进行表述)结束后就会变为 SELECT $ FROM $，$ WHERE $>1。

在随机变异阶段，系统会从每个 ABR 树的根节点开始遍历，所有子节点都会以一个随机的概率决定其是否进行变异操作。变异操作一共包含 3 种：增加、删除和替换。对于已经决定实施变异操作的目标节点，系统会使用这 3 种操作分别进行处理。删除操作会先找到当前节点的父节点，然后取消父节点对当前节点的引用关系；增加和替换操作需要从系统存储的 ABR 字典中随机找到一个与当前节点的 abr_t 相同的另外一个 ABR 节点，之后增加操作会将找到的新节点的所有子节点的引用复制给当前节点，而替换操作则会找到当前节点的父节点，然后将父节点对于当前节点的引用替换为 ABR 字典中找到的新节点。abr_t 标识着每个节点的 ABR 类型，反映了文法解析中 AST 节点的类型，因此相同 abr_t 的节点进行增加、删除和替换操作之后，能够很大程度地保留变异之后 SQL 查询的文法正确性。基于文法结构的变异算法的伪代码如算法 1 所示。

算法 1 基于文法结构的变异算法

```
输入：种子 SQLSeed
输出：结构化的 ABRTree
1. function SyntaxBasedMutation(SQLSeed)
2.     ABRTree ← ParseToABR(SQLSeed) // 将 SQL 查询解析为 ABR
3.     PlaceholderReplacedTree ← ReplaceVariablesWithPlaceholders(ABRTree) // 将 ABR 中的变量替换为占位符
4.     for each Node in PlaceholderReplacedTree do
5.         if RandomProbability() < MutationThreshold then// 生成随机数，决定是否需要进行变异操作
6.             MutationType ← RandomSelect({Addition, Deletion, Replacement}) // 从增加、删除和替换操作中随机选择一种变异策略
7.             if MutationType==Addition then // 增加操作
8.                 NewNode ← SelectFromDictionary(Node.abr_t) // 从 ABR 字典中选出一个相同类型的 ABR
9.                 InsertNode(Node.Parent, NewNode) // 将新 ABR 的子节点的引用复制给当前节点
10.            else if MutationType==Deletion then // 删除操作
11.                RemoveNode(Node) // 取消父节点对该节点的引用
12.            else if MutationType==Replacement then // 替换操作
13.                ReplacementNode ← SelectFromDictionary(Node.Type.abr_t)
14.                ReplaceNode(Node, ReplacementNode) // 将原本 ABR 对子节点的引用修改为新节点的子节点的引用
15.            end if
16.        end if
17.    end for
18.    return ABRTree
19. end function
```

系统在运行过程中会保存一个 ABR 字典，用于存储所有已预处理的 ABR。这个字典的 key 是不同类型 ABR 的abr_t，value是一个列表，里面存放着对应于字典 key 的同一类

ABR。在变异流程中需要进行增加或者替换操作时，就可以由提供的 abr_t 来寻找列表中同类型的 ABR。

4 基于语义规则的变量填充

在经过前述变异处理后，生成的 SQL 查询可以保证文法层面上的正确性，但却只有结构而没有数据，表示变量的部分仍然被占位符所占据。后续使用多项语义规则来约束这些变量填充，将占位符重新替换为具体的变量名。正确填充变量才能保证语义的正确，文法和语义都正确才能保证 SQL 查询的正确性。

SQL 查询中语义规则取决于语句本身的结构，这些规则确保各个变量在适当的上下文中使用[27]。在设计面向 Oracle 和 SQL Server 的模糊测试系统时，本文参考了一些开源 DBMS 的语义解析流程，决定主要专注于对变量的适用范围进行约束。这些约束规则主要包括表名字段名检查、字段类型检查、别名范围检查、聚合函数检查、子查询范围检查、视图和临时表检查。所有的约束规则之间存在优先级，在分析过 SQL 语句的 AST 之后找到适用的规则，然后按照优先级先后对 SQL 语句中的变量进行赋值。

图 4 展示了一种示例，假如经过文法结构变异的处理之后生成的 SQL 语句为 SELECT $，COUNT($) FROM $ WHERE $ > 1 GROUP BY $ 形式，当前 SQL 中的变量位置暂时仍由占位符“$”替代，现在要对其进行实际变量的填充。首先需要进行变量名字的赋予，对于每个占位符，将其全部替换为一个独一无二的标识符，目的是为了将每个变量都区分开。为了方便起见，直接使用一个自增的计数器实现。替换后的 SQL 语句变为 SELECT v1，COUNT(v2) FROM v3 WHERE v4 > 1 GROUP BY v5，其中以 v 开头的变量名是目前还未确定，需要之后替换为真实环境下的变量名。然后对 AST 进行分析，可知适用于该 SQL 语句的约束有 3 项：表名字段名约束、字段类型约束和聚合函数约束。且表名字段名约束的优先级最大，字段类型约束和聚合函数约束优先级次之。

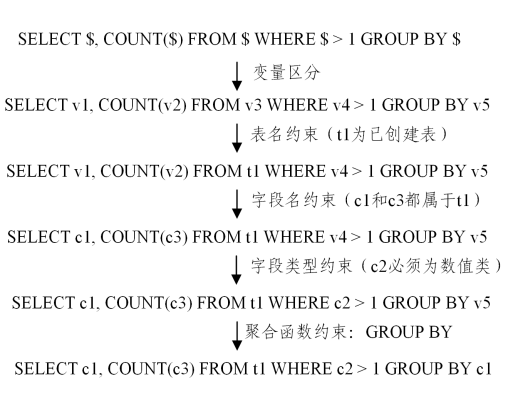

图 4 规则约束的变量填充示例

Fig. 4 Example of variable filling based on rule constraints

表名字段名约束规定，FROM 子句和 WHERE 子句中使用的表名必须是已经创建过的有效表，且 SELECT 子句中的所有字段名必须来自于 FROM 子句中的表。如果同时存在多个候选变量名，那么就随机挑选一个使用。字段类型约束规定，任何类型的子句中出现表达式，就需要根据运算符号来选择符合指定类型的变量。按照这个规则，需要从整个数据库的主题中找到一个已经创建过的表，假如找到的是 t1 表，此时将 SQL 语句更新为 SELECT v1，COUNT(v2) FROM t1 WHERE v4 > 1 GROUP BY v5，SELECT 子句中含有两个未确定变量 v1 和 v2，根据规则 v1 和 v2 必须要从 t1 所包含的字段中选取，t1 中可能包含多个字段，这时就会有多个候选值，从中随机选择两个。同样，WHERE 子句随机取一个字段即可，而且 WHERE 子句中的 v4 变量的上下文是一个数学表达式，还需要对其进行字段类型的约束，表达式是进行数值常量的大小比较，故应该挑选数值类的字段类型进行填充。此外，v1 和 v2 之间不存在约束关系，故 v1 和 v2 也可以是同一个值。依照这些规则，当 v1 和 v2 取不同值 c1 和 c3，WHERE 子句中的 v4 取数值类字段值 c2 时，SQL 语句就变为 SELECT c1，COUNT (c3) FROM t1 WHERE c2 > 1 GROUP BY v5。此时 SQL 语句中只剩下 v5 还未被初始化，聚合函数规则中有关 GROUP BY 的使用规定了 GROUP BY 子句的目标对象必须是来自于 SELECT 子句中的除聚合函数外的所有字段，因此 v5 初始化为 c1。如此，整个 SQL 语句中的变量全部填充完毕，最后生成的结果为 SELECT c1，COUNT(c3) FROM t1 WHERE c2 > 1 GROUP BY c1。整个流程的伪代码如算法 2 所示。

算法 2 基于语义规则的变量填充算法

输入：包含占位符的 PlaceholderTree，数据库的元数据 DBMetadata

输出：填充完变量的 PlaceholderTree

```
1. function VariableFilling(PlaceholderTree, DBMetadata)
2.     for each Node in PlaceholderTree do
3.         if Node.IsPlaceholder() then
4.             ApplicableRules ← DetermineRules(Node, DBMetadata)
               // 基于上下文决定约束规则
5.             for each Rule in ApplicableRules do
6.                 CandidateValues ← ApplyRule(Rule, DBMetadata,
                   Node.Context) // 获得满足约束规则的变量值
7.                 if CandidateValues.IsEmpty() then
8.                     continue
9.                 else
10.                    SelectedValue ← RandomSelect(CandidateValues) //
                       从生成的有效的变量值中随机选择
11.                    Node.value ← SelectedValue // 将变量值分配给占位符
12.                    break
13.                end if
14.            end for
15.            if Node.value==null then
16.                Node.value ← GenerateFallbackValue(Node.abr_t) //
                   为未能通过任何规则确定具体值的占位符节点生成一个
                   默认值
17.            end if
18.        end if
19.    end for
20.    return PlaceholderTree
21. end function
```

5 覆盖率反馈

基于文法结构的变异和基于语法规则的变量填充已经可以确保产出文法和语义都高度正确的 SQL 查询，将这两项功能集成实现 AFL++提供的接口标准可以开发出一个自定义变异器，在 AFL++运行时选择使用自定义变异器，而不是用默认的变异策略，就能先读入包含 SQL 查询的文件，然后交由变异器来解析结构，生成测试样例，再交给 DBMS 去运行。按照上述方法操作也可以进行测试，只是测试效果不佳，因为目前还未考虑程序执行路径的反馈信息，无法指导系统向着更优秀的方向推进测试流程。

5.1 计算覆盖率

选择使用 Intel Pin 工具来帮助获取目标程序运行时的路径信息。Intel Pin 是一个强大的二进制分析工具，允许用户在不修改源代码的情况下分析和调试应用程序[28]。本实验主要使用 Pin 的动态二进制插桩功能，来获取 Oracle 和 SQL Server 在运行过程中的基本块信息。Pin 是一个独立的功能组件，不能直接通过源码的形式集成到系统中，只能使用命令行的方式启动。Oracle 和 SQL Server 在系统中属于提前启动的服务进程，因此需要使用 Pin 的 attach 模式附加到 DBMS 进程上。

此外，还需要指定一个由用户编写的 Pin Tool 来决定 Pin 在探测进程时实施的工作。Pin 为用户提供了一些回调函数，用来处理探测行为，如表 1 所列，需要实现的有 Trace 函数、docount 和 Finish 函数等。

表 1 Pin Tool 内部函数功能实现

Table 1 Implementation of internal functions in Pin Tool

函数名	实现逻辑
Trace	pin 进程在探索到基本块时就执行，内部遍历基本块
docount	对每个基本块进行处理，存储基本块标识去重
Finish	每轮测试执行过后需要将结果输出到文件，清理资源

Trace 函数是 Pin 工具的回调函数，用于对目标程序的每个动态 trace 进行插桩。在 Pin 中 trace 代表的是一个在程序执行过程中不包含控制流改变的指令序列。Trace 函数会在程序每次遇到新的 trace 时被调用，这个函数是插入自定义逻辑的重要入口点。在 Trace 函数中，为了防止 DBMS 进程以外的链接库等的影响，先要对当前 trace 的所属地址进行验证，保证计算基本块的 trace 都属于目标镜像。

在每个 trace 中，使用 TRACE_BblHead 和 BBL_Next 可以分别获取到第一个基本块和当前基本块的下一个基本块，对于遍历到的每个基本块都执行 BBL_InsertCall 回调函数，内部需要调用基本块的计数函数 docount。在每一轮模糊测试开始时就初始化一个用于统计每轮基本块数量的静态变量和一个用于存储基本块标识的列表，docount 函数内部在遍历基本块时要增加总数量，同时计算基本块的哈希值作为唯一标识存放在列表中。这样可以同时记录每一轮测试中执行到的全部基本块(包含重复基本块)数量和唯一基本块数量。

在黑盒测试的过程中，若要统计比较精确的覆盖率，还需要知晓 DBMS 整个程序的代码基本块数量。由于没有已知的经验数据，在实际操作中只能通过一些逆向工具来进行估算，如使用 IDA 对安装目录的文件进行分析。在 Oracle 的安装目录/u01/app/oracle 下有很多文件，与 Oracle 进程运行相关的大部分文件都存放在 xe/bin 目录下，去除掉一些与程序运行主进程无关的实例初始化的脚本文件和数据存储类文件，剩余 46 个二进制文件，包含主进程 oracle、监听器 tnslnr、恢复管理器 rman 等启动 Oracle 服务时连带着一起启动的数据控制组件。将所有组件全部输入 IDA 中进行分析，导入统计脚本，脚本会分析文件中包含的所有函数，找到代码块中的指令的入口点并将入口点存储在一个列表中，最后计算数量，使用该值来代表每个文件中的基本块数量。经过计算，所有文件的基本块数量总和为 677042。类似地，在 SQL Server 的安装目录 MSSQL 中，着重分析 Binn 目录下的与主服务 MSSQLSERVER 运行相关的 sqlservr，sqlceip 和 sqlwriter 等可执行文件，以及使用到的各种动态库，计算出的基本块数量总和为 878614。

得到 DBMS 进程相关的所有基本块数量之后，在每轮测试结束的 Finish 函数中，将记录的唯一基本块数量写入文件中，作为基本块数量的有效值，就可以计算测试流程的覆盖率数值。覆盖率的计算式如下：

$$CoverageRate=\frac{A}{B}\times 100\% \tag{1}$$

式(1)中包含的变量有：

1)测试过程中由 Intel Pin 工具收集的唯一基本块数量。

2)目标 DBMS 的基本块总数，使用静态分析工具 IDA Pro 预先计算。

此覆盖率数值在模糊测试系统中不仅用于量化测试深度，还作为反馈机制的重要依据，被 AFL++主进程应用到种子调度策略中，指导种子的优先级调度，提高模糊测试效率。

5.2 将覆盖率应用至 AFL++的种子调度

在 Pin Tool 中获得每次测试执行过程中探索的基本块数量，同时获取到静态统计的程序基本块总数量，就可以获得目标程序的执行覆盖率。本文实验以这些数据为依据来判断种子的优秀程度，以便后续测试中能够使用表现更好的种子。由于整个模糊测试系统是基于 AFL++开发的，其内部有默认的种子调度策略，需要对其进行替换。

高质量的种子会很大程序上提高模糊测试的效率[29]，以 FAST 模式运行的 AFL++的原本调度策略为例，在每一轮模糊测试结束后就对队列中的条目进行评分，评分依据是每个种子是否能够触发新的路径以及触发的路径的深度等。随后以不同概率随机判断是否跳过对这些不同分类的种子的选取，直到选择到第一个符合条件的种子进行新的测试。这些调度涉及到主 While 循环中的 update_bitmap_score、select_next_queue_entry 等函数。

为了适配新的基本块覆盖率统计，对其替换后的调度策略为：在每轮测试开始前，在 AFL++精简队列的操作之后，检索目标 DBMS 的进程，然后将 Pin 附加上去，Pin Tool 就能获取每个 trace 中得到的基本块数量并将其记录在文件中，模糊测试完成之后，使用 detach 将 Pin 和 DBMS 进程分离。读取文件中的基本块数量，比较历史测试中探测到的基本块数

量较大的一部分种子，对其 favor 属性进行标记，这部分种子在生成随机数判断是否跳过时，对其放宽限制，增加其成果作为新一轮测试种子的概率，之后每一轮的模糊测试都重复这样的过程。

6 系统实现及实验测试

基于上述方法，本文设计并实现了一种面向 Oracle 和 SQL Server 的模糊测试系统 OFuz。测试方法为：首先读取用户提供的种子语料；然后交给语法解析器进行结构分析并生成 ABR，再通过变异器将 ABR 改造为各种不同结构，之后送给变量填充器补充变量并还原为完整的 SQL 查询；最后将 SQL 查询送往目标 DBMS 测试，同时每轮测试将 Pin 附加到 DBMS 进程分析覆盖率，根据覆盖率统计结果指导后续测试流程。在将 SQL 查询送往 DBMS 执行的过程中，通过 DBMS 返回的错误码来判断是否存在 crash（崩溃）或 hang（执行超时）等行为。

图 5 展示了 OFuz 中包含的组件之间的交互关系，整体来看，OFuz 整个系统包含 6 个主要功能组件，分别为 AFL＋＋驱动框架、语法解析器、结构变异器、变量填充器、覆盖率统计器，以及通信中间件。

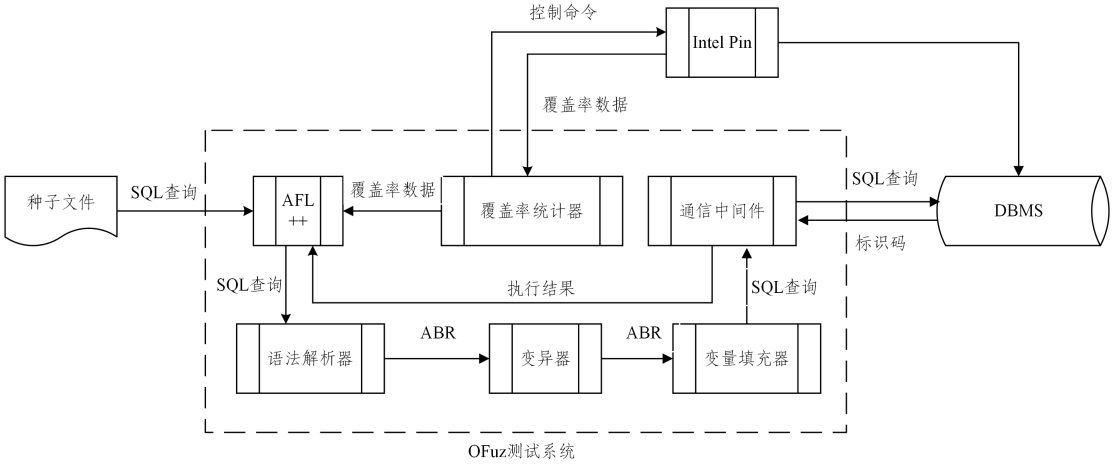

图 5 OFuz 测试系统结构

Fig. 5 Architecture of OFuz testing system

AFL＋＋框架：系统的驱动框架，使用其调度策略来协调和管理模糊测试流程。

语法解析器：对输入的种子进行解析，构建出 ABR 并存储。

结构变异器：使用基于文法结构的变异策略对 ABR 施加增加、删除和替换，将其改变为其他结构。

变量填充器：使用多种约束规则对 ABR 中的占位符进行赋值，并逆向还原为 SQL 查询。

覆盖率统计器：将 pin 附加到 DBMS 进程上，统计每轮测试的执行流程中探索到的基本块数量。

通信中间件：负责将生成的 SQL 查询送往 DBMS 执行，并将执行结果的标识码传给 AFL＋＋。下文将从几个方面来评价 OFuz 的性能表现。

1）OFuz 相比其他黑盒 DBMS 自动测试工具有什么优点？其通用性如何？（见 6.1 节）

2）Ofuz 生成测试数据的效率如何？（见 6.2 节）

3）OFuz 的基于文法结构的变异和基于语义规则的变量填充策略能否有效地生成 SQL 查询？（见 6.3 节）

4）OFuz 的基于 Intel Pin 的黑盒覆盖率统计组件是否能有效工作，其统计结果的实时性和精确性如何？（见 6.4 节）

使用 OFuz 进行测试，目标程序的版本是 Oracle-XE-11.2.0和 SQL Server 2016，整个测试过程持续 24 小时，因为在实验中观察到 OFuz 在 24 小时之后对于目标程序的覆盖率探索值基本不再变化。实验使用的输入种子来源于官方指导给出的 SQL 查询，在进行测试的过程中，依据 DBMS 执行 SQL 查询后返回的错误码和状态信息进行判断，以此为依据统计了变异策略构建的 SQL 查询在目标 DBMS 真实程序中通过语法检验的比例。

6.1 特点及通用性

相比其他 DBMS 测试工具，OFuz 的特点是非常明显的。其他工具无法用于闭源 DBMS 的测试，主要原因有两个：首先，这些工具依赖源码插桩和覆盖率反馈机制，而闭源 DBMS 无法进行源码级插桩，使得这些工具难以有效评估测试覆盖率；其次，现有工具并未针对闭源 DBMS 的 SQL 方言进行适配，缺乏针对性语法解析，无法批量生成语法和语义均高度正确的 SQL 查询。相比之下，OFuz 专门针对闭源 DBMS 开发，不仅支持自动化 SQL 查询生成，还可通过扩展语法解析器，适配 Oracle 和 SQL Server 以外的其他 DBMS。

OFuz 系统对测试 DBMS（如 Oracle 和 SQL Server）有很强的通用性，主要体现在两个方面。

首先，OFuz 中基于 ANTLR4 的语法解析器和基于约束规则的变量填充器为系统的扩展性奠定了基础。语法解析器通过解析数据库的方言特性生成对应的语法规则文件，可适配不同类型的关系型数据库。

其次，变量填充策略使用通用的约束规则描述方法，仅需对目标数据库的元数据进行调整，即可针对其他闭源数据库生成文法与语义均正确的测试数据，从而提升工具的普适性。

最后，OFuz 采用 Intel Pin 工具实现动态覆盖率统计，完

全依赖二进制插桩，无需目标程序源码支持，这使得工具对多种闭源数据库具有通用性。在实验中，Pin 工具的动态分析能力能够实时探测目标数据库的执行路径，适用于不同数据库的逻辑分析任务。实验结果表明，OFuz 可通过少量定制快速适配其他闭源数据库，同时覆盖率统计功能无需依赖特定的数据库架构，为工具在广泛测试场景中的应用提供了技术保障。

6.2 测试数据生成效率对比分析

为了对比分析 OFuz 生成测试数据的效率，使用开源 DBMS 的代表性测试工具 SQLsmith 和 Squirrel 进行了测试，以此作为参考。但由于 SQLsmith 和 Squirrel 不支持闭源 DBMS 的测试，因此本实验选择它们都支持的 MySQL 作为测试对象。原因是，尽管不同 DBMS 采用不同的 SQL 方言（如 Oracle 使用 PL/SQL，SQL Server 采用 T-SQL，而 MySQL 具有自身的 SQL 变体），但在查询函数、连接、约束等核心结构上，它们均遵循 SQL99 标准。此外，MySQL 作为一种广泛应用的数据库，具备良好的代表性。因此，本实验选用 MySQL 5.7 作为 SQLsmith 和 Squirrel 的测试对象，在相同硬件环境下运行 24 小时，以对比不同工具的测试数据生成效率。不同工具在不同数据库环境下的表现如表 2 所列。

表 2 不同工具生成 SQL 查询效率的对比

Table 2 Comparison of SQL query generation efficiency among different tools

DBMS 测试工具	生成 SQL 查询数量	生成 SQL 查询的正确数量
SQLsmith	102000	100000
Squirrel	95000	59000
OFuz(Oracle)	114000	67000
OFuz(SQL Server)	98000	54000

结合生成的查询数量和通过校验的有效数量进行分析，可以看出 OFuz 在闭源 DBMS 领域具有较强的适应性，且能够在黑盒环境下实现高效的 SQL 查询生成。

在 Oracle 数据库的测试中，OFuz 共生成了 114076 个 SQL 查询，根据 AFL++接收到的错误码数量可知，其中一共有 67035 个 SQL 查询是通过 Oracle 主程序的语法校验的，语法校验通过率达到 58.7%。在 SQL Server 数据库中，OFuz 的表现同样出色，共生成了 98018 个查询，通过校验的数量为 54498，校验通过率为 55.6%。这种稳定的表现说明：尽管测试环境更具挑战性，OFuz 仍然保持着稳定的语法正确性，并成功生成了大量可执行的 SQL 查询。

需要注意的是，SQLsmith 和 Squirrel 只可以针对开源 DBMS（实验使用 MySQL）进行测试，而 OFuz 面向闭源 DBMS（Oracle 和 SQL Server），两者的测试环境存在显著不同，因此测试通过率的对比并不能直接衡量工具的优劣，尤其是 OFuz 需要应对更具挑战性的闭源 DBMS 测试环境。SQLsmith 和 Squirrel 在 MySQL 上的测试可以利用其开源代码库中的官方语法解析器及相关组件作为参考，按照官方 SQL 语法规则构造查询生成逻辑，从而大幅提升 SQL 语句的正确性。此外，SQLsmith 仅能生成单条 SQL 语句，而不涉及复杂查询结构，因此其语法正确率相对更高。而 OFuz 由于测试闭源 DBMS，无法直接获取 Oracle 和 SQL Server 的语法解析源码，只能通过人工收集并分析 SQL 语法，基于 ANTLR4 解析器构建适配规则，这种方式在闭源环境下不可避免地存在一定局限性。因此，相比 SQLsmith 和 Squirrel 依赖官方解析器带来的正确率，OFuz 在闭源 DBMS 上进行 SQL 语法适配的难度更大，从而导致生成 SQL 查询的正确率没有明显优势。但 OFuz 在黑盒环境下生成的复杂 SQL 查询仍能保证较高的生成效率和正确率，实现了适用于闭源 DBMS 的 SQL 语法解析，这是其他工具做不到的。

综合比较 3 种工具的表现，OFuz 在闭源 DBMS 测试方面具有独特优势，并能够适应 Oracle 和 SQL Server 的 SQL 语法，填补了现有工具在闭源数据库测试领域的空白。此外，与 SQLsmith 相比，OFuz 的生成复杂度更大，能够覆盖更广泛的测试场景。本文在 6.3 节对 SQL 生成复杂度进行了详细分析，并通过实验对比不同工具生成的 SQL 结构，进一步验证了 OFuz 在生成复杂查询方面的优势。与 Squirrel 相比，OFuz 的 SQL 查询生成效率明显更高。

6.3 SQL 查询生成有效性分析

在测试 Oracle 和 SQL Server 的实验中，OFuz 分别生成了 114076 和 98018 个 SQL 查询，其中通过语法校验的数量分别为 67035(58.7%)和 54498(55.6%)。由于目前还没有类似实验进行 Oracle 和 SQL Server 的模糊测试，因此不能将该数据和其他实验数据进行严格的对照，这里用 DBMS 模糊测试领域的其他实验数据进行比较。大多数 DBMS 模糊测试工具都是对开源 DBMS 进行测试的，如目前最优秀的一些测试框架 SQLsmith，Squirrel 等，这些工具在生成 SQL 查询方面都有着很高的正确率，如图 6 所示。

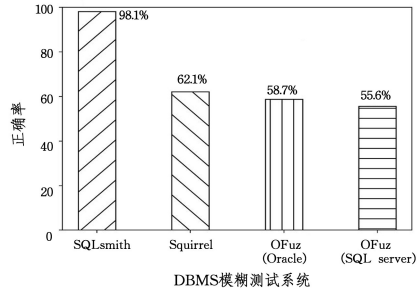

图 6 生成 SQL 查询的正确率对比

Fig. 6 Comparison of the correctness of generated SQL queries

由图 6 可知，SQLsmith 的正确率远高于其他工具，这是因为 SQLsmith 只能生成单条 SQL 语句构成的查询，复杂度不高的 SQL 语句难以发现深层次漏洞[30]。虽然测试结果显示，Squirrel 生成的 SQL 查询的正确率要稍高于 OFuz，但是 Squirrel 只针对一些开源 DBMS 测试，其利用源码插桩，而 OFuz 不需要程序源码即可进行黑盒测试。开源 DBMS 的语法解析器的实现可以参考官方提供的语法解析逻辑，相比商用非开源 DBMS 的语法解析器的实现来说难度较小。

在 OFuz 中语法解析器属于必要组件，如果直接去掉 Oracle 和 SQL Server 的语法解析器进行测试，则仅仅依靠

AFL++的变异策略根本无法生成正常的测试样例。若使用通用 SQL 语法解析器(仅实现 SQL99 标准而不针对某一特定方言而设计的)替换掉现有的语法解析器，则系统能够识别和构建的语法结构都很简单，而且大多数都是单条语句，最终导致 OFuz 能够探索到的程序深度很有限，无法充分发掘程序的深层漏洞。

另外，针对前面提到的 SQL 查询的复杂度，对这 3 种工具进行了评价，结果如表 3 所列。目前还没有一个广泛认可的关于 SQL 查询的复杂度的判断标准，这里决定采用每个查询中相关联的 SQL 语句数量的平均值来作为判断标准。一个 SQL 查询中会包含多条单独的语句，这里的“相关联”分为直接关联和间接关联。在一个 SQL 查询中，直接关联指后面的语句直接使用了前面所创建的元素，最简单的直接关联的例子就是前面某条语句插入了某条记录，后面就会出现某条语句选择该条记录；而间接关联指后面的语句使用了前面创建的元素中的子元素或关联元素，例如创建一个表之后，对其字段名进行修改，属于使用子元素的情况，而触发器引起的一些处理结果被后续语句拿去使用，则属于使用关联元素的情况。为了能够比较准确地统计到这个数值，在每一轮测试生成 SQL 查询的变量填充过程前，先创建一个空字典，key 代表变量名字，value 代表被引用到的次数。填充变量的过程中将每一个已经赋值的变量填入字典中，如果后续语句使用到同一个变量，就对其次数值加 1。在每一轮结束后，统计字典中最大的计数值作为本次 SQL 查询的语句关联度。收集到足够多的关联度之后，求它们的平均值作为实验所得的平均复杂度。

表 3 不同工具生成 SQL 查询的对比

Table 3 Comparison of SQL queries generated by different tools

DBMS 测试工具	生成 SQL 查询中平均复杂度	生成 SQL 查询的正确率/%
SQLsmith	1.0	98.1
Squirrel	4.3	62.1
OFuz(Oracle)	4.6	58.7
OFuz(SQL Server)	4.4	55.6

SQLsmith 的每个查询中只有单条语句，因此复杂度为 1。以此为基准，Squirrel 和 OFuz 的查询复杂度分别为 4.3 和 4.5 左右，其中 Squirrel 的复杂度来源于对 MySQL 的测试结果，OFuz 对 Oracle 和 SQL Server 两种软件都进行了测试。

通过实验验证，OFuz 的基于文法结构的变异和基于语义规则的变量填充策略能够有效生成语法和语义均正确的 SQL 查询。这些生成的查询在语法复杂性和语义完整性上优于通用 SQL 解析器生成的结果，尤其是在闭源 DBMS 的特定方言测试中展现出显著的适配性。实验结果表明，OFuz 能够生成多样化的 SQL 查询，不仅覆盖了基本的语法结构，还能够探索到复杂的程序路径，从而为闭源数据库的深层漏洞挖掘提供了强有力的支持。

6.4 覆盖率统计器的性能评估

覆盖率统计器是 OFuz 系统中的重要性能组件。通过这一组件，OFuz 系统能够实现覆盖率驱动的种子优选，从而更高效地探索未覆盖路径。在模糊测试中，覆盖率统计的效果直接影响到测试的效率和深度。实验将从覆盖率统计器的有效性、实时性和精确性 3 个方面，对其性能进行详细分析和讨论。

6.4.1 有效性验证

使用 OFuz 分别对 SQL Server 和 Oracle 测试覆盖基本块数量进行对比。以 Oracle 测试为例，每次运行过程中 pin 统计到的已探索基本块数量为 30 000～160 000，可以判断有少量 SQL 查询不会被送入 DBMS 的深层处理逻辑，此情况下探索基本块数量大概在 60 000 左右；而在语法正确的情况下，整体来说能够探索到的基本块数量与 SQL 查询的复杂程度成正相关。为了测试 OFuz 能够探索到的最大限度的基本块数量，对已有的统计基本块数量的 Pin Tool 做了一些修改，在 Trace 函数中对测试流程中遇到的所有基本块数量进行累加，并且每隔 2 个小时就向指定文件输出一次当前计算的结果。仍然使用相同的环境运行 30 小时，结果显示在整个测试流程中的前 24 个小时之内 OFuz 所能探测到的基本块数量呈现一直增长的趋势，尤其是在前 16 个小时之内增长速度最快。而在 24 小时之后，测试系统所能探索到的基本块数量总和趋于平稳，最大值在 440 000 左右，结合通过静态方式计算出的基本块数量总和，最终计算出的代码覆盖率为 65%。实验的增长曲线如图 7 所示。

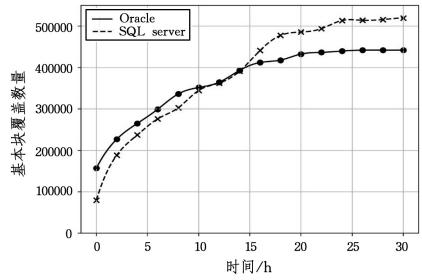

图 7 覆盖数量增长曲线

Fig. 7 Growth curves of covered blocks

覆盖率统计器是 OFuz 的性能组件。为评估其作用，屏蔽覆盖率统计器，改用 AFL++ 的 Qemu 模式运行系统，并在相同环境下进行测试。由于 AFL++和 DBMS 进程之间存在中间件负责数据传输，Qemu 模式无法直接统计 DBMS 主进程的覆盖信息，从而实现了对覆盖率统计器的屏蔽。测试结果如图 8 和图 9 所示。

可以看到，相同实验环境下持续运行期间无覆盖率统计器时，程序探索到的基本块数量最多为原始 OFuz 系统探索到的数量的一半。在 SQL Server 的测试结果中，可以明显地看到基本块的探索数量呈阶梯形式增长，出现这种情况的原因是，在没有覆盖率统计器的数据指导时，系统探索程序覆盖率是比较盲目的行为，可能会在某个时刻生成一个比较优秀的测试样例，但是在大部分时间内，都只是在重复生成一些普通的用例进行执行。

实验结果充分验证了覆盖率统计器在 OFuz 系统中的有效性。在多轮实验中，覆盖率统计器成功记录了被测数据库的基本块覆盖数量，并显著提升了测试系统的路径探索能力。

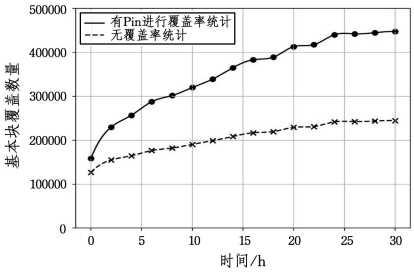

图 8 覆盖率统计器对 Oracle 测试的影响

Fig. 8 Impact of the coverage tracker on Oracle testing

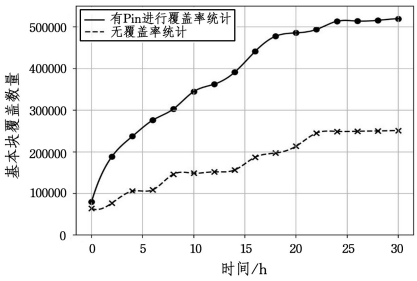

图 9 覆盖率统计器对 SQL Server 测试的影响

Fig. 9 Impact of the coverage tracker on SQL Server testing

6.4.2 实时性分析

由于 OFuz 系统的覆盖率统计基于 Intel Pin 工具实现，Pin 在工作过程中不可避免地会产生一部分开销。实验中，对 OFuz 的覆盖率统计实时性进行了量化分析，主要考察其统计延迟。统计延迟定义为覆盖率统计器收集执行路径信息并完成基本块数量计算所需的时间。这一延迟发生在测试样例执行之后，且将计算结果反馈给 AFL++进行种子调度之前的这一段时间。

由于长时间测试过程中生成的统计数据过多，因此以时间段进行划分，整个测试过程依然是 24 小时，每个小时分别计算一次该时间段内所有执行过的 SQL 查询的覆盖率统计的平均延迟。以 Oracle 测试过程为例，测试系统在 24 小时内一共执行了约 110 000 个 SQL 查询，计算可知平均每小时内可以执行 4583 个查询。在每个小时时间段内执行的 SQL 查询的平均延迟如图 10 所示。

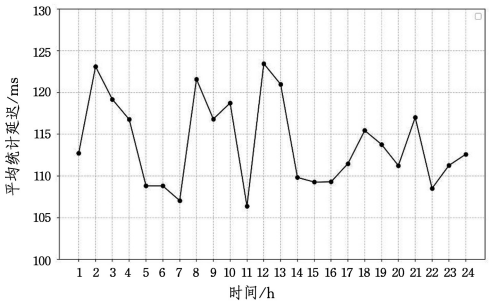

图 10 Oracle 测试中覆盖率统计器平均延迟

Fig. 10 Average latency of the coverage tracker in Oracle testing

从测试结果可以看出，在每个小时内平均统计延迟范围为 110～120 ms，整体测试过程的 24 小时内波动较小，标准差为±2.5 ms。长时间运行未观察到显著的性能退化现象，表明了覆盖率统计器工具在持续运行和统计操作中的性能稳定性。

此外，在相同硬件环境下对 SQL Server 也进行了覆盖率统计实时性测试。在 24 小时的测试中，每小时的统计延迟平均值范围为 115～130 ms，相比 Oracle 测试稍有上升。原因可能是 SQL Server 更大的程序规模导致插桩和数据处理的时间增加。在对 SQL Server 的长时间测试中同样未观察到显著的性能退化，验证了覆盖率统计器在不同数据库场景下的实时性和稳定性。

实验结果表明，覆盖率统计器的统计延迟占每次 SQL 查询总时间的 12%～15%，这部分时间开销对模糊测试流程的影响较小。这种低延迟特性使得覆盖率统计器能够在较高频率下反馈覆盖率信息，为种子调度和未覆盖路径探索提供了重要支持。以上实验数据验证了 OFuz 的覆盖率统计器在整个测试系统中具备良好的实时性。

6.4.3 精确性验证

在覆盖率统计的精确性分析中，动态统计结果的可重复性是一个关键维度。由于 OFuz 的覆盖率统计器主要应用于种子调度，其核心目标是有效区分测试用例的优劣，而非追求覆盖率统计的绝对准确性。因此，只需验证覆盖率统计器在相同实验条件下是否能够生成一致的统计结果，即可证明其对测试流程的指导能力和策略优化的可靠性。

由于 OFuz 的覆盖率统计器位于文法变异之后，生成 SQL 查询的随机性可能会显著影响覆盖率统计结果的可重复性。为了更准确地评估统计器的精确性，本实验仅将初始的种子 SQL 交由 DBMS 执行，去掉原本的变异过程（即将语法解析器、变异器和变量填充器无效化），通过使用统一的测试输入和多轮实验验证等方法，仅关注覆盖率统计器对相同测试场景的反馈结果。

实验方法是：提前选取一组涵盖不同复杂度和功能类型的 SQL 查询作为固定种子并排序，将固定种子按顺序直接交由 DBMS 运行，同时覆盖率统计器进行统计，重复运行 3 次。每轮实验中，记录每一个 SQL 查询运行过后的已覆盖基本块数量。以 SQL Server 为例，实验结果如表 4 所列。

表 4 覆盖基本块数量 3 轮统计

Table 4 Three-round statistics of covered basic blocks

测试轮次	总覆盖基本块数量	相对误差/%
第一轮	459 852	0.04
第二轮	460 212	0.09
第三轮	460 169	0.07
平均值	460 078	—
标准差	±173	0.08

此外，还对实验中使用的单个 SQL 查询的 3 轮基本块覆盖数量进行了统计和分析，结果显示，每个 SQL 查询在 3 次实验中的覆盖基本块数量差异极小；标准差最大为±200 基本块，说明覆盖率统计器在多次实验中的表现高度一致；单个 SQL 查询的覆盖基本块数量范围为 60 000～120 000，与整体实验数据一致，充分体现了覆盖率统计器在不同类型查询中的适应性。

以上实验结果充分验证了 OFuz 基于 Intel Pin 的黑盒覆盖率统计组件的正确性和可靠性。在多轮实验中，覆盖率统

计器准确记录了被测试数据库的已执行基本块数量。此外,覆盖率反馈的实时性显著提升了种子调度效率,使得测试系统在有限时间内探索到更多的深层逻辑路径。这表明,覆盖率统计组件能够正确工作,并且在 OFuz 系统的模糊测试流程中发挥了至关重要的作用。

结束语 虽然 OFuz 展现了比较优秀的性能,但是在开发的过程中,还有一些其他思路可以帮助工具进行改进和优化。

1)语法解析器方面。OFuz 是专门针对闭源 DBMS 的自动化测试开发的工具,其对关系型数据库都是通用的,只是目前 OFuz 中的语法解析器是针对 Oracle 和 SQL Server 所使用的方言开发的,其中种子的收集都来源于官方指导文档中的例子。在以后的工作中,计划设计一种工具来自动收集一些库中的 SQL 查询用于分析。另外,后续若能针对其他方言扩展出其相应的语法解析器,那么 OFuz 就能够支持 Oracle 和 SQL Server 以外的其他 DBMS 的测试。

2)覆盖率统计方面。在之后的工作中,可以通过逆向分析等方法进一步研究 pin 附加在 DBMS 进程上时 Oracle 和 SQL Server 的各种数据处理对 pin 的探测反应。若能获取到更明显的特征,就可以设计出统计覆盖率更加精确的 Pin Tool 组件。此外,将 pin 附加到 DBMS 进程上在一定程度上会产生额外开销,如果对 DBMS 内部处理特征了解得更加详细,那么就能够将这部分时间开销继续减少,继续提升程序的运行效率。

参 考 文 献

[1] SENDNER C,IFFLÄNDER L,SCHINDLER S,et al. Ransomware detection in databases through dynamic analysis of query sequences[C]// 2022 IEEE Conference on Communications and Network Security(CNS). IEEE,2022:326-334.

[2] WANG M,WU Z,XU X,et al. Industry practice of coverage-guided enterprise-level DBMS fuzzing[C]// 2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice. IEEE,2021:328-337.

[3] WANG Q Y,XU J C,LI Y W,et al. A Review of Smart Fuzzing:Problem Exploration and Method Classification [J]. Chinese Journal of Computers,2024,47(9):2059-2083.

[4] ZHONG R,CHEN Y H,HU H,et al. SQUIRREL:Testing Database Management Systems with Language Validity and Coverage Feedback[C]// ACM SIGSAC Conference on Computer and Communications Security(ACM CCS). 2020:955-970.

[5] JIANG Z M,BAI J J,SU Z D,et al. DynSQL:Stateful Fuzzing for Database Management Systems with Complex and Valid SQL Query Generation[C]// 32nd USENIX Security Symposium. 2023:4949-4965.

[6] RIGGER M,SU Z D,ASSOC U. Testing Database Engines via Pivoted Query Synthesis[C]// 14th USENIX Symposium on Operating Systems Design and Implementation(OSDI). 2020:667-682.

[7] DOU W S,CUI Z Y,DAI Q W,et al. Detecting Isolation Bugs via Transaction Oracle Construction[C]// 45th IEEE/ACM International Conference on Software Engineering(ICSE). 2023:1123-1135.

[8] JUNG J H,HU H,ARULRAJ J,et al. APOLLO:Automatic Detection and Diagnosis of Performance Regressions in Database Systems[J]. Proceedings of the VLDB Endowment, 2019,13(1):57-70.

[9] LIU X Y,ZHOU Q,ARULRAJ J,et al. Automatic Detection of Performance Bugs in Database Systems using Equivalent Queries[C]// ACM/IEEE 44th International Conference on Software Engineering(ICSE). 2022:225-236.

[10] ZHENG Y Y,DOU W S,WANG Y C,et al. Finding Bugs in Gremlin-Based Graph Database Systems via Randomized Differential Testing[C]// 31st ACM SIGSOFT International Symposium on Software Testing and Analysis(ISSTA). 2022:302-313.

[11] HUA Z Y,LIN W,REN L Y,et al. GDsmith:Detecting Bugs in Cypher Graph Database Engines[C]// 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis(ISSTA). 2023:163-174.

[12] YANG Y,CHEN Y,ZHONG R,et al. Towards Generic Database Management System Fuzzing[C]// 33rd USENIX Security Symposium(USENIX Security 24). 2024:901-918.

[13] PHAM V,BÖHME M,SANTOSA A E,et al. Smart Greybox Fuzzing[J]. IEEE Transactions on Software Engineering,2021,47(9):1980-1997.

[14] LIANG J,WU Z Y,FU J Z,et al. Survey on Database Management System Fuzzing Techniques [J]. Journal of Software,2025,36(1):399-423.

[15] FIORALDI A,MAIER D,EIßFELDT H,et al. AFL++:Combining incremental steps of fuzzing research[C]// 14th USENIX Workshop on Offensive Technologies(WOOT 20). 2020.

[16] CHEN P,CHEN H. Angora:Efficient Fuzzing by Principled Search[C]// 39th IEEE Symposium on Security and Privacy (SP),IEEE,2018:711-725.

[17] LIANG J,WU Z,FU J,et al. Mozi:Discovering DBMS Bugs via Configuration-Based Equivalent Transformation[C]// Proceedings of the IEEE/ACM 46th International Conference on Software Engineering. 2024:1-12.

[18] LIANG Y,LIU S,HU H,et al. Detecting Logical Bugs of DBMS with Coverage-based Guidance[C]// 31st USENIX Security Symposium. 2022:4309-4326.

[19] PAN Q F,XU C. Advances in SQL Execution Techniques Based on Query Compilation [J]. Journal of Computer Research and Development,2024,61(7):1754-1770.

[20] TRICKEL E,PAGANI F,ZHU C,et al. Toss a Fault to Your Witcher:Applying Grey-box Coverage-Guided Mutational Fuzzing to Detect SQL and Command Injection Vulnerabilities[C]// 44th IEEE Symposium on Security and Privacy(SP),IEEE,2023:2658-2675.

[21] WANG J H,SONG C Y,YIN H,et al. Reinforcement Learning-based Hierarchical Seed Scheduling for Greybox Fuzzing[C]//. 28th Annual Network and Distributed System Security Symposium(NDSS). 2021.

[22] WANG A Q,YANG B,ZHANG J H,et al. A Survey of SQL Injection Attack Detection and Defense Technology [J]. Journal

of Information Security Research, 2023, 9(5): 412-422.
[23] BA J S, RIGGER M. Testing Database Engines via Query Plan Guidance[C]// 45th IEEE/ACM International Conference on Software Engineering(ICSE). 2023: 2060-2071.
[24] BLAZYTKO T, ASCHERMANN C, SCHLOGEL M, et al. GRIMOIRE: Synthesizing Structure while Fuzzing[C]// 28th USENIX Security Symposium. 2019: 1985-2002.
[25] WANG W T, SUN J J, WAN Y F, et al. Fuzzing for Binary Software Based on Program Analysis [J]. Computer Systems and Applications, 2025, 34(1): 294-307.
[26] LIU X, ZHOU Q, ARULRAJ J, et al. Testing dbms performance with mutations[J]. arXiv: 2105. 10016, 2021.
[27] FU J Z, LIANG J, WU Z Y, et al. Griffin: Grammar-Free DBMS Fuzzing[C]// 37th IEEE/ACM International Conference on Automated Software Engineering(ASE). 2022.
[28] ZHANG J, ZHANG C, XUAN J F, et al. Recent Progress in Program Analysis [J]. Journal of Software, 2019, 30(1): 80-109.
[29] FU J, LIANG J, WU Z, et al. Sedar: Obtaining High-Quality Seeds for DBMS Fuzzing via Cross-DBMS SQL Transfer[C]// Proceedings of the IEEE/ACM 46th International Conference on Software Engineering. 2024: 1-12.
[30] LI J, WANG K, CHEN Y, et al. Detecting DBMS Bugs with Context-Sensitive Instantiation and Multi-Plan Execution[J]. arXiv: 2312. 04941, 2023.

LI Zhongjie, born in 1996, postgraduate, is a member of CCF (No. N1001G). His main research interest is database vulnerability discovery.

CAO Yan, born in 1983, associate professor, Ph.D supervisor, is a member of CCF (No. 17447S). His main research interest is vulnerability discovery.

(责任编辑:何杨)