介绍 KLEE，并展示其使用方法和应用

计算机科学与技术学院
西安电子科技大学

小组成员：
HASANBAYEV AGABEK
EZIZOV MEKAN

Outline

什么是 Symbolic Execution?

定义

Symbolic Execution（符号执行）是一种程序分析技术，它使用符号变量而非具体值作为程序输入，通过跟踪符号在程序中的变化来探索所有可能的执行路径。

传统测试

- 使用具体输入值
- 覆盖率有限
- 难以覆盖边界条件

符号执行

- 使用符号变量
- 自动探索多个路径
- 理论上可以达到更高的路径覆盖

将程序输入视为代数变量，通过约束求解器确定满足特定路径条件的输入值。

主要目标

- **路径覆盖**：自动探索程序的多条执行路径
- **缺陷检测**：发现内存错误、空指针引用、除零错误等
- **测试生成**：自动生成高覆盖率的测试用例
- **程序验证**：验证程序是否满足特定属性

覆盖率优势

传统测试：手动选择输入 → 覆盖率通常较低。
符号执行：自动探索路径 → 覆盖率通常更高（视程序复杂度而定）。

符号执行过程

1. **符号化输入**：将输入变量替换为符号变量
2. **路径探索**：遇到条件分支时，创建多个执行路径
3. **约束收集**：为每条路径收集路径约束
4. **约束求解**：使用 SMT 求解器求解约束

```
int check(int x) {
    if (x > 10) {
        return 1;
    } else {
        return 0;
    }
}
```

- 路径 1 约束：$x > 10 \rightarrow$ 可能解例如 $x = 11$
- 路径 2 约束：$x \leq 10 \rightarrow$ 可能解例如 $x = 5$

KLEE 概述

- 基于 LLVM 的符号执行引擎
- 最初由斯坦福/帝国理工等学术团队开发并维护
- 支持 C/C++ 程序分析，生成高覆盖率测试用例
- 开源，具备活跃社区和多种扩展

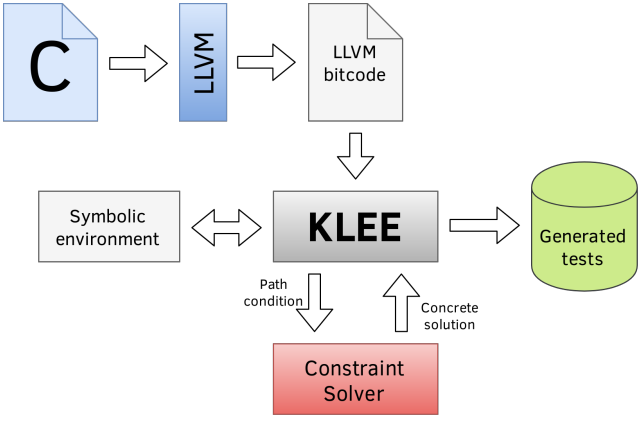
架构特点

- 基于 LLVM IR
- 支持多种求解器（SMT）
- 提供 POSIX 运行时模型
- 精确的内存建模

核心优势

- 高精度路径探索
- 内存错误检测
- 自动测试用例生成
- 良好的可扩展性

KLEE 架构


C
LLVM
LLVM bitcode
Symbolic environment
KLEE
Generated tests
Path condition
Concrete solution
Constraint Solver

Ubuntu 安装步骤（示例）

```
# 安装依赖
sudo apt-get install build-essential cmake bison flex libncurses-dev python3 zlib

# 安装 LLVM 14
sudo apt-get install clang-14 llvm-14 llvm-14-dev llvm-14-tools

docker pull klee/klee
docker run --rm -ti --ulimit='stack=-1:-1' klee/klee

# 验证安装
klee --version
```

注意事项

建议使用 Docker 镜像以简化安装和版本依赖问题

示例程序

```
int get_sign(int x) {
  if (x == 0)
    return 0;

  if (x < 0)
    return -1;
  else
    return 1;
}

int main() {
  int a;
  // 标记 a 为符号变量
  klee_make_symbolic(&a, sizeof(a), "a");
  return get_sign(a);
}
```

编译与运行

```
# 编译为 LLVM 字节码（示例）
clang -I /path/to/klee/include -emit-llvm -c -g test

# 使用 KLEE 执行
klee --output-dir=output test.bc

# 查看输出目录
ls output/
ktest-tool output/test000001.ktest
```

结果说明:
KLEE 会为不同路径生成若干 `ktest` 文件，例如：`test000001.ktest`（触发 $x == 0$），`test000002.ktest`（触发 $x < 0$），`test000003.ktest`（触发 $x > 0$）。

KLEE output —explanation

- **output directory is "..."** —where KLEE writes generated tests and diagnostics (e.g., `klee-out-0`).
- **Using STP solver backend** —the SMT solver selected (can be changed via `-solver-backend`).
- **ERROR: ... ASSERTION FAIL** —KLEE found a path that triggers an assertion (reports file and line).
- Final **done:** lines are statistics (instructions executed, completed paths, tests generated).
- Note: number of generated tests may be less than completed paths if tests are redundant or do not cover new states.

测试用例生成:

- 自动生成高覆盖率测试用例
- 回归测试：检测代码修改引入的新 bug
- 自动发现边界条件问题

Coreutils 案例

- KLEE 在 GNU Coreutils 上的实验发现了众多 bug（论文中已有报告）
- 包括：内存越界、空指针、未定义行为等

	COREUTILS		BUSYBOX	
Coverage (w/o lib)	**KLEE tests**	**Devel. tests**	**KLEE tests**	**Devel. tests**
100%	16	1	31	4
90-100%	40	6	24	3
80-90%	21	20	10	15
70-80%	7	23	5	6
60-70%	5	15	2	7
50-60%	-	10	-	4
40-50%	-	6	-	-
30-40%	-	3	-	2
20-30%	-	1	-	1
10-20%	-	3	-	-
0-10%	-	1	-	30
Overall cov.	84.5%	67.7%	90.5%	44.8%
Med cov/App	94.7%	72.5%	97.5%	58.9%
Ave cov/App	90.9%	68.4%	93.5%	43.7%

Table 2: Number of COREUTILS tools which achieve line coverage in the given ranges for KLEE and developers' tests (library code not included). The last rows shows the aggregate coverage achieved by each method and the average and median coverage per application.

漏洞挖掘

- 缓冲区溢出检测：精确跟踪内存访问
- 格式化字符串漏洞分析
- 权限提升路径验证
- 协议实现的规范验证

优势

- 精确路径分析
- 能定位触发条件

局限

- 路径爆炸（branch explosion）
- 对复杂环境/系统调用需建模
- 约束求解可能耗时

学术研究

- 程序合成、形式化验证、编译器优化验证、硬件验证等

工业应用

- 与模糊测试结合（如 OSS-Fuzz 的思想）
- 驱动/内核模块验证、嵌入式系统安全分析等

领域	应用	效果
操作系统	内核模块验证	发现多处 bug
网络协议	协议实现分析	提高安全性
编译器	优化验证	确保正确性
区块链	智能合约分析	检测逻辑漏洞

主要结论

- 符号执行通过符号输入与约束求解实现程序的系统化分析
- KLEE 提供基于 LLVM 的符号执行引擎，适用于 C/C++, Python 等程序

未来方向

- 与机器学习结合优化路径选择
- 分布式/云化的符号执行以提升规模
- 更好地建模系统/外部交互

谢谢大家！

参考资料：

- KLEE 官方网站: `https://klee-se.org/`
- KLEE 论文: “KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs”
- GitHub 仓库: `https://github.com/klee/klee`