LibFuzzer实战教程：从0到1 Fuzzing一个真实的C++模块

简介

什么是模糊测试 (Fuzzing)?

模糊测试（Fuzzing）是一种自动化的软件测试技术。它通过向程序提供大量随机、无效或非预期的输入（称为"fuzz"）并监视其异常（如崩溃、断言失败、内存泄漏等）来发现软件中的 Bug。

LibFuzzer 是一个内置于 LLVM/Clang 编译器中的、覆盖率引导的模糊测试引擎。它非常适合 Fuzzing C/C++ 库和模块。

对于libfuzzer等模糊测试工具来说，windows会有很多bug，一般linux系统没有bug，且配置过程较为简单，所以我们在windows上提供了详细的教程，同时也提供了linux的版本，不过强烈推荐大家使用Linux版本的教程。

对于每一个"步骤"，都有Windows和Linux两个教程，如果你看到了🖥️的图标，说明这是Windows版本的教程，可以直接下拉直到看到🐧的图标

我们的实验目标

本教程将带你完成一个完整的 Fuzzing 流程：

1. **准备工具**：安装 Fuzzing 所需的 Clang 编译器。
2. **编写 Driver**：编写 Fuzzing "驱动程序"（Driver），将 LibFuzzer 的随机数据"喂"给我们选择的模块。
3. **编译运行**：编译并运行 Fuzzer，寻找潜在的 Bug。
4. **分析结果**：学会如何解读 Fuzzer 的输出。

步骤 1: 准备环境

🖥️ Windows：

安装 Clang:

Fuzzing 严重依赖现代 C++ 编译器。**Clang 是必需的**，因为它内置了 LibFuzzer 和 AddressSanitizer (ASan)。

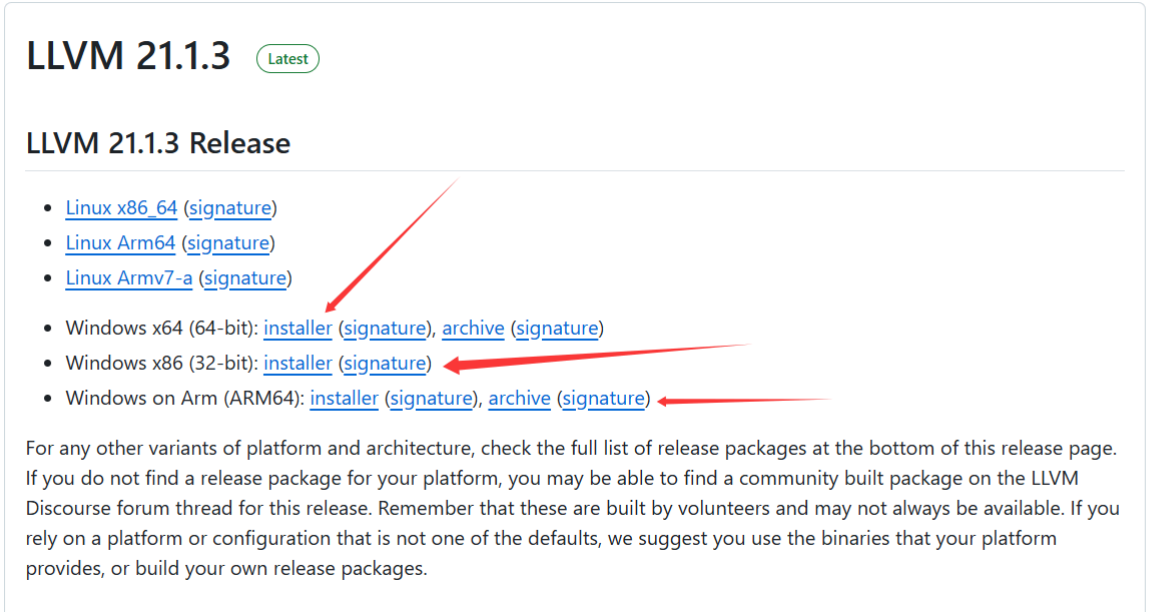
步骤 1：访问 LLVM 下载页面

打开你的浏览器，访问 LLVM 官方下载页面： https://llvm.org/builds/ 或者直接访问 GitHub 上的发布页面（通常更新最快）： https://github.com/llvm/llvm-project/releases

如果你能使用github，则**推荐**使用这个链接：https://github.com/llvm/llvm-project/releases

步骤 2：选择下载版本

根据你的配置选择以下版本：



步骤 3：运行安装程序

下载完成后，运行你下载的 `.exe` 安装程序。

1. 你会看到一个欢迎界面，点击 "Next"。
2. 阅读并同意许可协议 (License Agreement)，点击 "Next"。

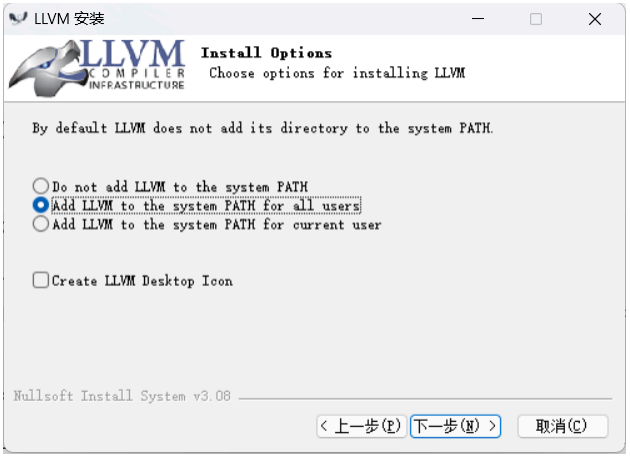
步骤 4：配置安装选项

在安装选项界面，你会看到几个复选框。**这一步至关重要**：

- **"Add LLVM to the system PATH"** (将 LLVM 添加到系统 PATH)

  请务必勾选这个选项！ * **"Add LLVM to the system PATH for all users"** (为所有用户添加) - 如果你是管理员并且希望所有用户都能用，选这个。
    - **"Add LLVM to the system PATH for current user"** (为当前用户添加) - 通常选这个就足够了。

  勾选此选项后，你才能在命令行 (cmd 或 PowerShell) 中直接运行 `clang` 命令。
- （可选）"Create LLVM shortcut" (创建 LLVM 快捷方式) - 这不重要。
- （可选）"Add LLVM to the system library path" (添加 LLVM 到系统库路径) - Fuzzing 时不需要。



步骤 5：选择安装路径

选择一个你喜欢的路径。点击 "Next"。

步骤 6：开始安装

点击 "Install" 并等待安装程序完成。

步骤 7：验证安装

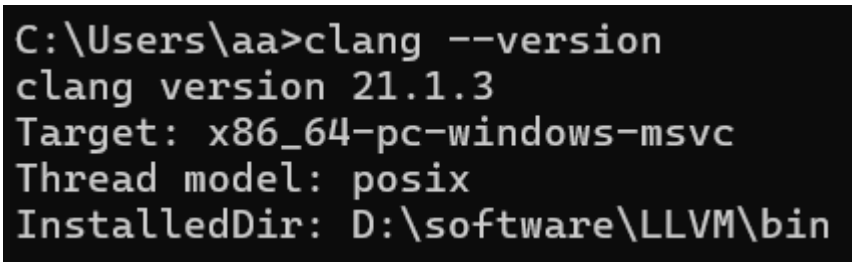
安装完成后，你需要验证 Clang 是否已成功安装并配置在你的 PATH 中。

1. **重要： 打开一个新的**命令提示符 (cmd) 或 PowerShell 窗口**（win+R，然后输出cmd）**。
    - *如果你有已经打开的窗口，请先关闭它们再重新打开，这样才能加载新的 PATH 环境变量。*
2. 在新的命令行窗口中，输入以下命令并按回车：

    Bash

    ```
    clang --version
    ```

3. 如果你看到类似下面的输出，说明安装成功了：

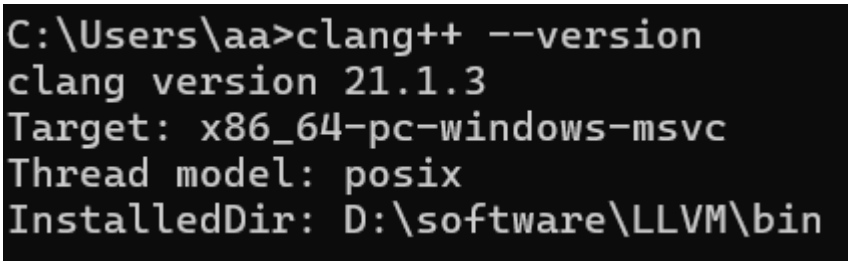
    

4. 你也可以顺便检查 C++ 编译器 `clang++`：

    Bash

    ```
    clang++ --version
    ```

    (它应该显示和 `clang --version` 一样的信息)

现在，你就可以在 Windows 上使用 Clang 来编译你的 Fuzzer 了。

安装Visual Studio

从 LLVM 官网下载的 Clang默认是为 **MSVC** (Microsoft Visual C++) 工具链构建的。

在 Windows 上，一个完整的 C++ 编译环境需要：

1. **编译器 (Compiler)**：例如 `clang++` 或微软的 `cl.exe`。
2. **C++ 标准库 (Standard Library)**：包含所有的头文件，如 `<cstdint>`, `<iostream>`, `<vector>`。
3. **链接器 (Linker)**：将编译好的代码（.obj 文件）和库文件链接成最终的 .exe 文件。

Clang 本身只是一个编译器（第1点），因此，需要下载第2、3点，因此这里我们选择Visual Studio，当然也可以自行选择其他工具，只是这个方法**更方便**。

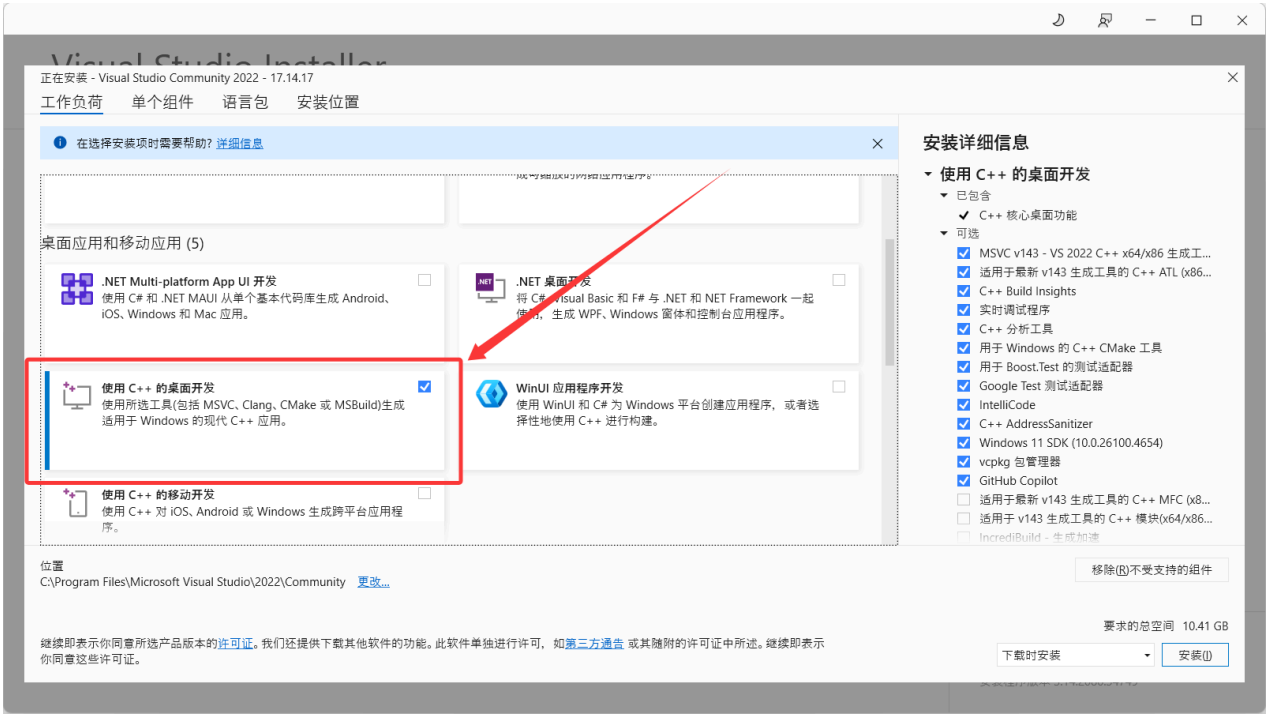
以下是安装教程：

第一步：下载 Visual Studio 安装器 (Installer)

1. 打开浏览器，访问 Visual Studio 官方下载页面： https://visualstudio.microsoft.com/zh-hans/downloads/
2. 在页面上，找到 **"Visual Studio 2022"**。
3. 在 **"Community" (社区版)** 选项下方，点击 **"免费下载"** 按钮。

第二步：运行安装器并选择 "工作负载"

1. 双击你刚下载的 `.exe` 文件。
2. 系统可能会提示你授权，点击 "是" (Yes)。
3. 安装器会首先进行一些准备工作（下载和安装 "Visual Studio Installer" 本体），点击 "继续" (Continue) 并等待它完成。
4. 准备完成后，你会看到一个标题为 **"正在安装 - Visual Studio Community 2022"** 的窗口，上面有很多方框选项。这些选项被称为 **"工作负载" (Workloads)**。
5. **【核心步骤】** 在 "工作负载" 标签页下，找到并**用鼠标勾选 "使用 C++ 的桌面开发" (Desktop development with C++)** 这个方框。



第三步：(可选) 检查安装详细信息

在你勾选了 "使用 C++ 的桌面开发" 之后，可以看一眼右侧的 "安装详细信息" (Installation details) 栏：

1. **确保 "MSVC v143"**（或最新版的 v14x）**编译器工具集**被勾选了。
2. **确保 "Windows SDK"**（例如 Windows 11 SDK 或 10 SDK）被勾选了。

> **注意：** 通常情况下，你只要勾选了 "使用 C++ 的桌面开发" 这个主工作负载，上述这些必需的组件都会被自动勾选，你一般不需要额外操作。

第四步：开始安装

1. 在窗口的右下角，你可以选择安装位置。
2. 点击右下角的 **"安装"** (Install) 按钮。
3. Visual Studio Installer 会开始下载和安装你选择的组件。

第五步：安装完成与启动

安装完成后，安装器会提示你重启电脑。**建议你立即重启**，以确保所有环境变量都生效。

🐧 Linux

在 Linux (以 Ubuntu/Debian 为例) 上，环境准备要简单得多。你不需要单独下载安装包，也不需要 Visual Studio。使用系统的包管理器 (apt) 即可一次性安装所有必需品。

- **步骤 1：打开终端 (Terminal)**
- **步骤 2：安装 Clang 和 build-essential** `build-essential` 包含了 `g++` 编译器、C++ 标准库头文件和 `make` 等构建工具，Clang 会自动使用它们。

  ```
  # 1. 更新你的包列表
  sudo apt update

  # 2. 安装 Clang, 相关的 libfuzzer 工具, 以及 C++ 构建套件
  sudo apt install clang build-essential
  ```

  这个命令会同时安装 Clang 编译器、LibFuzzer 工具链以及 C++ 标准库/头文件。
- **步骤 3：验证安装** 在同一个终端中，输入以下命令：

  ```
  clang --version
  clang++ --version
  ```

  如果你看到 Clang 的版本信息，就说明安装成功了。你可以在任何标准终端中继续操作。

步骤 2: 准备目标代码

现在，我们来创建一个专门用于 Fuzzing 的工作目录。

🖥️ Windows

1. **创建一个单独的文件夹**：

   ```
   mkdir fuzz_lru_test
   cd fuzz_lru_test
   ```

2. **把测试的目标文件放到文件夹中**：

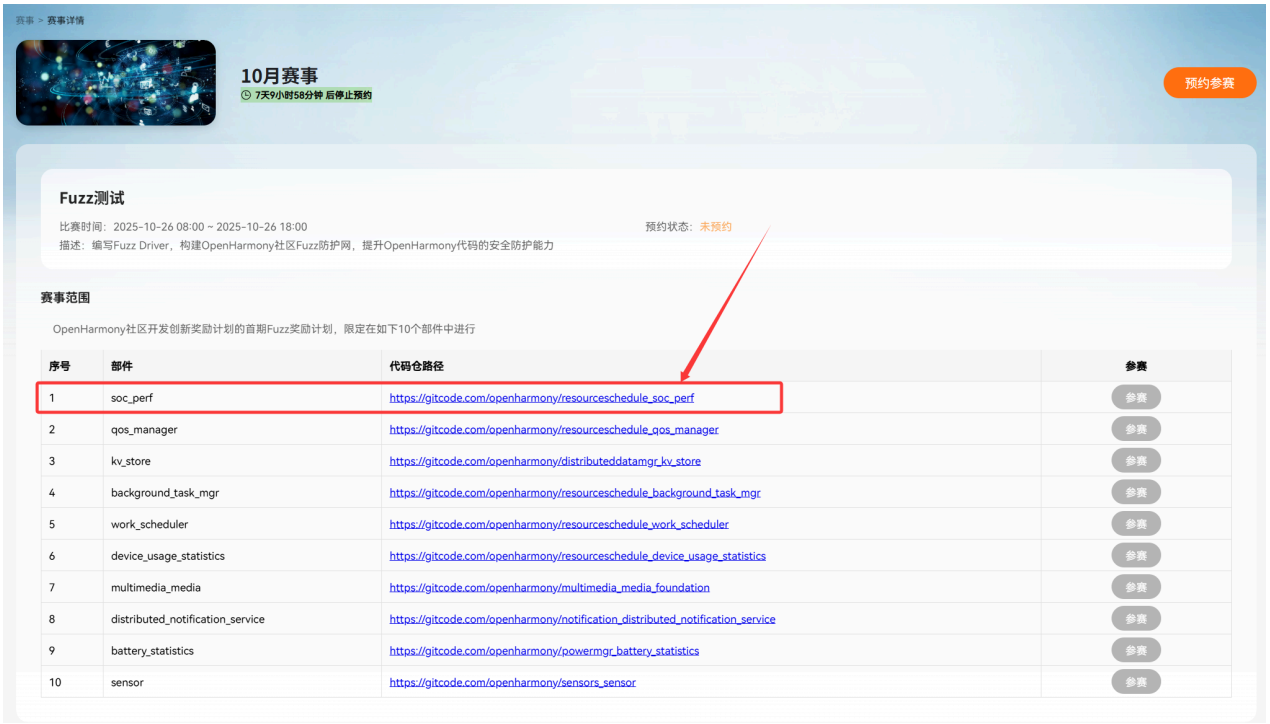
   由于本次实验和华为的比赛主题相同，所以我们选择一个比赛中的开源项目作为测试目标，可以通过下面的方

式手动下载代码，也可以直接从群里发送的附件“fuzz_lru_test.zip”中直接获取完整代码，或者通过这个链接从GitHub上下载“https://github.com/SYSUSELab/Software-Test-Course/tree/main/Lab6”，这个步骤只是获取一个统一的代码，不需要大家报名比赛。

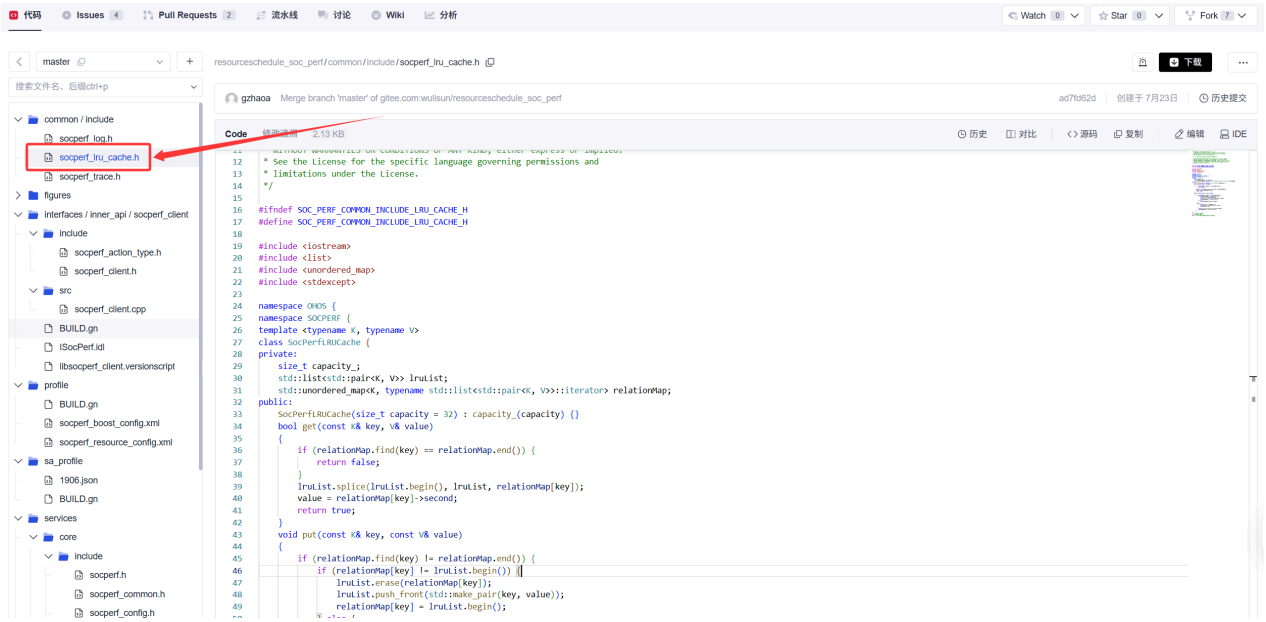
打开鸿蒙比赛官网：https://devbounty.openharmony.cn/index

点击“10月赛事”

点击进入第一个代码仓库**（本次实验以仓库一作为示例）**



将这个代码下载下来，这个代码没有项目中的其他代码依赖，较为简单，因为我们作为实验教学示例。



现在，我们的目录结构如下：

```
fuzz_lru_test/
└── socperf_lru_cache.h
```

Linux

创建一个专门用于 Fuzzing 的工作目录。

```
# 1. 创建目录
mkdir fuzz_lru_test
cd fuzz_lru_test
```

把测试的目标文件放到文件夹中：（从鸿蒙比赛官网下载 `socperf_lru_cache.h`）

假设文件下载到了你的 `~/Downloads` 目录，你可以使用 `cp` 命令将其复制到当前目录：

```
cp ~/Downloads/socperf_lru_cache.h .
```

现在，你的目录结构如下（使用 `ls` 命令查看）：

```
fuzz_lru_test/
└── socperf_lru_cache.h
```

步骤 3: 分析 Fuzz "攻击面"

(此步骤在 Windows 和 Linux 上相同)

我们的目标是 `SocPerfLRUCache` 类。LibFuzzer 提供的是**原始字节流**（`const uint8_t *Data, size_t Size`）。我们必须将这些字节“翻译”成对 `put` 和 `get` 的调用。

- **输入**：我们需要生成 `K` (Key) 和 `V` (Value)。
- **操作**：我们需要决定是调用 `put` 还是 `get`。
- **序列**：我们不应该只调用一次，而应该模拟一系列操作来测试缓存的复杂状态。

为了轻松地将原始字节转换为结构化数据（如 `int`, `bool`, `string`），我们将使用 `FuzzedDataProvider`。

步骤 4: 编写 Fuzz Driver

(此步骤在 Windows 和 Linux 上相同)

Fuzz Driver 是连接 LibFuzzer 和目标代码的“胶水”。

1. **创建 Driver 文件**： 在 `fuzz_lru_test` 目录中，创建第二个文件 `fuzz_lru.cpp`。
2. **编写 Driver 代码**： 将以下代码粘贴到 `fuzz_lru.cpp` 中。请仔细阅读注释，它解释了每一行。

```
#include <cstdint>
#include <cstddef>
#include <string>

// 1. 包含 FuzzedDataProvider，这是 LibFuzzer 的一个辅助工具
#include <fuzzer/FuzzedDataProvider.h>

// 2. 包含我们要测试的目标代码
#include "socperf_lru_cache.h"

// 定义我们要 Fuzz 的 Key 和 Value 类型
// 我们选择简单的类型：uint32_t 作为 Key，std::string 作为 Value
using FuzzKey = uint32_t;
using FuzzValue = std::string;

// 3. 这是 LibFuzzer 的主入口点
// 每次执行，LibFuzzer 都会生成新的 Data 和 Size
extern "C" int LLVMFuzzerTestOneInput(const uint8_t *Data, size_t Size) {

    // 4. 初始化 FuzzedDataProvider
    FuzzedDataProvider fdp(Data, Size);

    // 5. 设置缓存容量，也由 Fuzzer 决定（比如 1 到 128 之间）
    // 我们不使用 `static` 缓存，而是为每次输入创建一个新缓存
    // 这样可以测试不同容量和构造函数
    size_t capacity = fdp.ConsumeIntegralInRange<size_t>(1, 128);
    OHOS::SOCPERF::SocPerfLRUCache<FuzzKey, FuzzValue> cache(capacity);

    // 6. 模拟一系列操作，直到 Fuzzer 提供的
    //    数据耗尽 (fdp.remaining_bytes() > 0)
    //    我们最多执行 200 次操作，防止超时
    int operations = 0;
    while (fdp.remaining_bytes() > 0 && operations++ < 200) {
```

```
        // 7. 让 Fuzzer 决定下一步是 'put' 还是 'get'
        bool is_put = fdp.ConsumeBool();

        if (is_put) {
            // 模拟 'put'
            // 从 Fuzzer 数据中提取 Key
            FuzzKey key = fdp.ConsumeIntegral<FuzzKey>();

            // 从 Fuzzer 数据中提取 Value (最多 100 字节)
            FuzzValue value = fdp.ConsumeRandomLengthString(100);

            // 调用目标 API
            cache.put(key, value);

        } else {
            // 模拟 'get'
            // 从 Fuzzer 数据中提取 Key
            FuzzKey key = fdp.ConsumeIntegral<FuzzKey>();

            // 准备一个变量来接收 'get' 的结果
            FuzzValue out_value;

            // 调用目标 API
            cache.get(key, out_value);
        }
    }

    // 8. 必须返回 0
    return 0;
}
```

现在，你的目录结构应该是：

```
fuzz_lru_test/
├── lru_cache.h
└── fuzz_lru.cpp
```

步骤 5: 编译 Fuzzer

🖥️ Windows

这是最关键的一步。我们将使用 `clang++` 并启用所有必要的 "Sanitizers"（卫生工具）来查找 Bug。

打开你的终端，**确保你在 `fuzz_lru_test` 目录下**：

通常需要在开始菜单中搜索：x64 Native Tools Command Prompt for VS 2022（具体x64或者x86由你的电脑决定，注意，如果直接在visual studio打开命令行，有可能打开的版本不正确，所以推荐在开始菜单中搜索）

然后进入到你的文件夹目录下：

```
cd /d D:\research_related\研究生相关事务\助教课程\fuzz_lru_test
```

然后运行：

```
clang++ -g -O1 -fsanitize=fuzzer,undefined -std=c++17 fuzz_lru.cpp -o fuzz_lru.exe
```

让我们分解这个命令：

- `clang++`：使用 Clang C++ 编译器。
- `-g`：包含调试信息。当 Fuzzer 崩溃时，这能给我们提供准确的行号。
- `-O1`：一级优化。官方推荐，在速度和代码可调试性之间取得了良好平衡。
- `-fsanitize=fuzzer`：**核心！** 告诉 Clang 链接 LibFuzzer 引擎。
- `-fsanitize=undefined`：**极其重要！**
  - `undefined` (UBSan) 开启未定义行为卫生器，用于检测（如整数溢出、空指针解引用）。
- `-std=c++17`：使用 C++17 标准（`lru_cache.h` 中的代码需要它）。
- `fuzz_lru.cpp`：我们的 Driver 文件。
- `-o fuzz_lru`：将输出的可执行文件命名为 `fuzz_lru`。

注意：我们**不需要**编译 `lru_cache.h`，因为它是一个纯头文件，`fuzz_lru.cpp` 在 `#include` 它时已经将其编译进去了。

出现以下输出就是正确的：

```
D:\research_related\研究生相关事务\助教课程\fuzz_lru_test>clang++ -g -O1 -fsanitize=fuzzer,undefined -std=c++17 fuzz_lru
.cpp -o fuzz_lru.exe
   正在创建库 fuzz_lru.lib 和对象 fuzz_lru.exp
```

如果有其他输出，可以暂时忽略，一般不影响运行

Linux

1. **打开你的标准终端。** (不需要特殊的命令提示符)
2. **进入到你的文件夹目录下**:

```
cd /path/to/your/fuzz_lru_test
# 例如: cd ~/fuzz_lru_test
```

3. **运行编译命令**: 我们使用一个更健壮的命令，**同时启用 AddressSanitizer (ASan) 和 UndefinedBehaviorSanitizer (UBSan)。**

```
clang++ -g -O1 -fsanitize=fuzzer,address,undefined -std=c++17 fuzz_lru.cpp -o fuzz_lru
```

命令分解:

- `clang++`, `-g`, `-O1`, `-std=c++17`, `fuzz_lru.cpp`: 与 Windows 相同。
- `-fsanitize=fuzzer,address,undefined`: **核心!**
  - `fuzzer`: 链接 LibFuzzer 引擎。
  - `address`: 开启 AddressSanitizer (ASan)，检测内存错误 (如缓冲区溢出、释放后使用等)。
  - `undefined`: 开启 UBSan，检测未定义行为。
- `-o fuzz_lru`: **区别点。**将输出的可执行文件命名为 `fuzz_lru` (Linux 不使用 `.exe` 后缀)。

(编译成功后，终端同样会安静地返回提示符)

步骤 6: 运行 Fuzzer

Windows

1. **创建语料库 (Corpus)**: Fuzzer 需要一个目录来存放它发现的"有趣"的输入。

```
mkdir corpus
```

2. **开始 Fuzzing!**

```
# 运行 Fuzzer，'corpus' 是它存放和读取输入的目录
.\fuzz_lru.exe corpus/
```

3. **观察输出**: 你将看到 LibFuzzer 开始运行，每秒执行成千上万次:

```
INFO: Seed: 123456789
INFO: Loaded 1 modules   (42 inline 8-bit counters): 42 [0x...
INFO: -max_len is not provided; libFuzzer will not generate inputs larger than 4096
bytes
INFO: A corpus is not provided, starting from an empty corpus
#2       INITED cov: 10 ft: 11 corp: 1/1b exec/s: 0 rss: 28Mb
#4       NEW    cov: 12 ft: 13 corp: 2/3b lim: 4 exec/s: 0 rss: 28Mb L: 2/2 MS: 1
ChangeByte-
#8       NEW    cov: 15 ft: 16 corp: 3/7b lim: 4 exec/s: 0 rss: 28Mb L: 4/4 MS: 1
InsertByte-
...
#1048576 pulse  cov: 40 ft: 52 corp: 15/123b lim: 4096 exec/s: 524288 rss: 35Mb
```

- `cov:` (Coverage): 覆盖的代码分支。这个数字增长是好事!
- `corp:` (Corpus): 语料库中的文件数。
- `exec/s:` (Executions per Second): **每秒执行次数。**这是 Fuzzer 的速度。

4. **让它运行**： 让 Fuzzer 运行一段时间，比如 60 秒。你可以按 `Ctrl+C` 停止它，或者使用参数：

```
# 运行 Fuzzer 60 秒
.\fuzz_lru.exe corpus/ -max_total_time=60
```

Linux

1. **创建语料库 (Corpus)**： (命令相同)

```
mkdir corpus
```

2. **开始 Fuzzing! 区别点**：在 Linux 上，你需要使用 `./` 来执行当前目录下的程序。

```
# 运行 Fuzzer
./fuzz_lru corpus/
```

3. **观察输出**： (输出格式与 Windows 相同)

```
INFO: Seed: 123456789
...
#2       INITED cov: 10 ft: 11 corp: 1/1b exec/s: 0 rss: 28Mb
#4       NEW    cov: 12 ft: 13 corp: 2/3b lim: 4 exec/s: 0 rss: 28Mb L: 2/2 MS: 1
ChangeByte-
...
```

4. **让它运行**： 你可以按 `Ctrl+C` 停止它，或者使用参数：

```
# 运行 Fuzzer 60 秒
./fuzz_lru corpus/ -max_total_time=60
```

步骤 7: 分析结果

(此步骤在 Windows 和 Linux 上相同)

你有两种可能的结果：

结果 A: 未发现崩溃 (最常见，也是这次你得到的结果)

如果 Fuzzer 一直运行直到你停止它，并且**没有**显示红色的 `ERROR` 信息，那么恭喜你！ 这说明 `SocPerfLRUCache` 在 LibFuzzer 强大的攻势下（在 ASan 和 UBSan 的监视下）表现稳健。

这也是一次成功的 Fuzzing！ 你的测试增强了你对这段代码的信心。

结果 B: 发现崩溃 (！成功！)

如果 Fuzzer 突然**自动停止**，并在屏幕上打印出一份详细的、通常是红色的错误报告，那么恭喜你，你发现了一个 Bug！这种情况将在接下来的步骤 8 中模拟。

步骤 8: 验证 Fuzzer！(引入并发现一个 Bug)

为了验证我们的 Fuzzer 设置确实能有效发现 Bug，我们可以故意在 `socperf_lru_cache.h` 中引入一个简单的 Bug，然后看 Fuzzer 能否快速找到它。

8.1 手动添加一个 Bug

(此步骤在 Windows 和 Linux 上相同)

我们将添加一个非常经典的 Bug：**空指针解引用 (Null Pointer Dereference)。**这种 Bug 会导致程序立即崩溃，非常容易被 Fuzzer 发现。

1. 打开你的 `socperf_lru_cache.h` 文件。
2. 找到 `put` 函数。
3. 在 `put` 函数的**最开头**，添加以下代码：

```
    void put(const K& key, const V& value)
    {
        // ======================================================
        //           !!! 我们在这里添加一个 Bug !!!
        //
        // 当 Fuzzer 生成的 value 字符串长度超过 20 时，
        // 我们就触发一个空指针解引用。
        if (value.size() > 20) {
            int *p = nullptr; // 创建一个空指针
            *p = 42;          // 尝试写入空指针 -> 100% 崩溃！
        }
        // ======================================================
```

```
        // ... (函数剩下的代码保持不变) ...
        if (relationMap.find(key) != relationMap.end()) {
            // ...
        } else {
            // ...
        }
    }
```

4. **保存** `socperf_lru_cache.h` 文件。

8.2 重新编译 Fuzzer (带 Bug 的版本)

🖥️ Windows

回到你的"**开发者命令提示符**"窗口，使用这段命令，重新编译你的 Fuzzer（这会把带有 Bug 的代码编译进去）：

```
clang++ -g -O1 -fsanitize=fuzzer,undefined -fno-sanitize-recover=all -std=c++17 fuzz_lru.cpp -o fuzz_lru.exe
```

🐧 Linux

回到你的终端，使用 Linux 的编译命令重新编译 (同样添加 `address` 和 `-fno-sanitize-recover=all`)：

```
clang++ -g -O1 -fsanitize=fuzzer,address,undefined -fno-sanitize-recover=all -std=c++17 fuzz_lru.cpp -o fuzz_lru
```

8.3 清理语料库并重新运行 Fuzzer (寻找 Bug)

🖥️ Windows

为了让 Fuzzer 更快地找到新 Bug，最好从一个空的语料库开始。

1. **（推荐）** 删除旧的语料库文件夹：

   ```
   rmdir /s /q corpus
   ```

2. **（必须）** 重新创建语料库文件夹：

   ```
   mkdir corpus
   ```

3. **运行 Fuzzer！**

   ```
   .\fuzz_lru.exe corpus/
   ```

🐧 Linux

```
# （推荐） 删除旧的语料库文件夹 (Linux 使用 rm -rf)：
rm -rf corpus

# （必须） 重新创建语料库文件夹：
mkdir corpus

# 运行 Fuzzer!
./fuzz_lru corpus/
```

8.4 分析崩溃报告！

(此步骤在 Windows 和 Linux 上的输出几乎相同)

这一次，Fuzzer 应该会在**几秒钟内**（甚至更快）**自动停止**，并且**在屏幕上打印出类似这样的错误报告**：

```
INFO: Running with entropic power schedule (0xFF, 100).
INFO: Seed: ...
INFO: Loaded 1 modules ...
INFO: A corpus is not provided, starting from an empty corpus
#2       INITED cov: ...
.\socperf_lru_cache.h:50:13: runtime error: store to null pointer of type 'int'
SUMMARY: UndefinedBehaviorSanitizer: undefined-behavior .\socperf_lru_cache.h:50:13
==12345== libFuzzer: deadly signal
    #0 0x... in OHOS::SOCPERF::SocPerfLRUCache<...>::put(...) .\socperf_lru_cache.h:50
    #1 0x... in LLVMFuzzerTestOneInput fuzz_lru.cpp:45
    #2 0x... in fuzzer::Fuzzer::ExecuteCallback(...)
    ...
NOTE: libFuzzer has rudimentary signal handlers.
      Combine libFuzzer with AddressSanitizer or similar for better crash reports.
artifact_prefix='./'; Test unit written to ./crash-a1b2c3d4e5f6a1b2c3d4e5f6
```

关键信息解读：

- `runtime error: store to null pointer`：明确告诉你 Bug 类型（写入空指针）。
- `.\socperf_lru_cache.h:50:13`：精确指出了 Bug 在代码中的位置（第 50 行，第 13 列）。
- `SUMMARY: UndefinedBehaviorSanitizer: ...`：确认是 UBSan 捕获了这个未定义行为。
- `Test unit written to ./crash-...`：**最重要！** Fuzzer 已经为你保存了导致崩溃的输入文件。

8.5 验证 Crash 文件

🖥️ Windows

现在，回到你的命令行窗口（`fuzz_lru_test` 目录下），输入 `dir`：

```
dir
```

你应该会看到一个新文件，例如 `crash-a1b2c3d4e5f6a1b2c3d4e5f6`。

你可以用这个文件来**精确复现**崩溃：

```
.\fuzz_lru.exe crash-bfd69a2f755407a9a6ac4a5a1c09386becf2f5ae
```

它会立刻再次打印出同样的错误报告。

例如：

```
D:\research_related\研究生相关事务\助教课程\fuzz_lru_test>.\fuzz_lru.exe crash-
bfd69a2f755407a9a6ac4a5a1c09386becf2f5ae
INFO: Running with entropic power schedule (0xFF, 100).
INFO: Seed: 3380496516
INFO: Loaded 1 modules   (510 inline 8-bit counters): 510 [00007FF7E368C008,
00007FF7E368C206),
INFO: Loaded 1 PC tables (510 PCs): 510 [00007FF7E364A720,00007FF7E364C700),
.\fuzz_lru.exe: Running 1 inputs 1 time(s) each.
Running: crash-bfd69a2f755407a9a6ac4a5a1c09386becf2f5ae
.\socperf_lru_cache.h:50:13: runtime error: store to null pointer of type 'int'
SUMMARY: UndefinedBehaviorSanitizer: undefined-behavior .\socperf_lru_cache.h:50:13
```

成功！ 你现在已经验证了你的 Fuzzer 设置能够有效地发现 Bug！

Linux

回到你的命令行窗口，输入 `ls`：

```
ls -l crash-*
```

你应该会看到一个新文件，例如 `crash-a1b2c3d4e5f6a1b2c3d4e5f6`。 你可以用这个文件来精确复现崩溃 (使用 `./`)：

```
./fuzz_lru crash-bfd69a2f755407a9a6ac4a5a1c09386becf2f5ae
```

(它会立刻再次打印出同样的错误报告)

总结

你已经完成了你的第一个 Fuzzing 实验！

我们回顾了整个流程：

1. **准备环境**：安装 `clang++`。
2. **隔离目标**：从一堆代码中选择了简单、独立的 `lru_cache.h`。
3. **编写 Driver**：使用 `FuzzedDataProvider` 将随机字节流转换成对 `put` 和 `get` 的调用。
4. **编译**：使用 `-fsanitize=address,undefined` 编译。
5. **运行和分析**：启动 Fuzzer 并学会了如何分析崩溃报告。