Hull & WAAP

WASM Judge 的应用

Aberter0x3F (aberter0x3f@sjtu.edu.cn)

TUIQUN ‘26

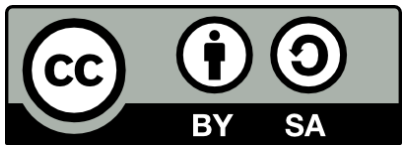
2026-07-12

Copyright

本文档内容采用

知识共享 署名–相同方式共享 4.0 协议

CC BY-SA 4.0



Outline

Hull 引言

去年的问题

根据 fstqwq 的意见，本项目如果要普及需要一个与它非常适配的情景.

基于 Nix 的本地造题系统?

从 WASM Judge 到 Hull

WASM Judge 已经实现了评测尺度稳定, 允许安全并行. 但作为造题系统, 仍需要解决 **题目信息声明**, **工具链依赖管理**, **自动化打包流程**.

我们需要一个能做到上述几点的设计.

Why to choose Nix?

或许我们可以从 Nix 是什么来回答.

Why to choose Nix?

Nix 是依赖管理工具.

```
inputs = {
  nixpkgs.url = "github:NixOS/nixpkgs/nixos-26.05"; cplib = ...; hull
= ... ;
};

outputs = {self, nixpkgs, hull, cplib}: let ... in {
  perSystem = forEachSystem (system: {
    hullProblems = {
      major = hullLib.evalProblem ./problem/major { };
      stone = hullLib.evalProblem ./problem/stone { };
      count = hullLib.evalProblem ./problem/count { };
    };
    hullContests.default = hullLib.evalContest ./contest.nix { };
  });
};
```

`flake.nix` + `flake.lock` 可声明, 锁定工具链.

Why to choose Nix?

Nix 是函数式编程语言.

```
{
  validator = { src = ./validator.23.cpp; tests = ...; };
  checker = { src = ./checker.23.cpp; tests = ...; };
  testCases = {
    random-small = {
      generator = "rand";
      arguments = [ "--n-max=100" ];
    };
  };
  traits = { n_le_100 = { descriptions.en = "$N <= 100$."; }; };
  subtasks = [
    { traits = { n_le_100 = true; }; fullScore = 0.4; }
    { fullScore = 0.6; }
  ];
```

Why to choose Nix?


```
  solutions.std = {
    src = ./solution/std.23.cpp;
    mainCorrectSolution = true;
    subtaskPredictions = {
      "0" = { score, ... }: score == 1.0;
      "1" = { score, ... }: score == 1.0;
    };
  };
  documents = { ... };
  targets = {
    default = hull.problemTarget.common;
    hydro = hull.problemTarget.hydro { statements.en =
"statement.en.pdf"; };
  };
}
```

代替各种 problem.json, problem.yaml, 声明式定义题目.

Why to choose Nix?

Nix 是脚本语言.

```
$ ls nix/lib/problemTarget
nix/lib/problemTarget/
├── hydro/
│   ├── checker.c
│   └── default.nix
├── legacy/
│   ├── luogu/
│   │   ├── default.nix
│   │   └── wrapper.c
│   ├── uoj/
│   │   ├── default.nix
│   │   └── wrapper.c
│   ├── cms.nix
│   ├── domjudge.nix
│   ├── hydro.nix
│   └── lemon.nix
├── lemon/
│   ├── default.nix
│   └── watcher.c
├── uoj/
│   ├── supervisor/
│   │   ├── src/
│   │   ├── Cargo.lock
│   │   └── Cargo.toml
│   ├── README.txt
│   ├── default.nix
│   ├── judger.mk
│   └── prepare.c
├── common.nix
└── default.nix
```

自定义 judger? 外接自定义 target?

Why to choose Nix?

题目是一份可执行规格.

$$\text{声明} \to \text{构建} \to \text{分析} \to \text{验证} \to \text{交付}$$

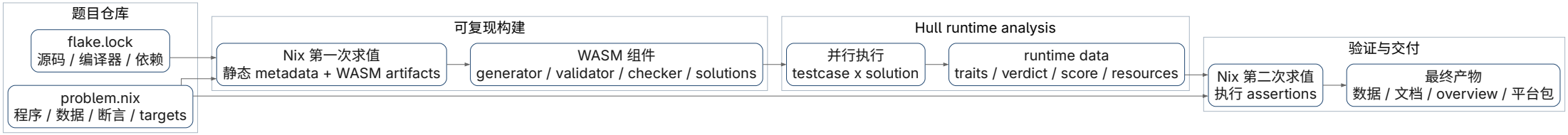
We can do them all in one.

声明式题目模型

程序	generator, validator, checker, solutions
数据	test cases, traits, subtasks
验证	组件测试, 解法预测, 运行断言
交付	题面, overview, 单题与比赛 targets

这些部分共同参与同一次求值. validator 输出的 traits 决定测试点所属的 subtask; main correct solution 生成标准输出; 其他解法的实际得分必须符合声明中的预测. 文档和平台包则读取同一份运行分析结果.

构建流程



第一次 Nix 求值产生静态 metadata 和待实现的 WASM artifacts. Rust runtime 实现这些 artifacts, 并行运行 validator, checker 和 solutions. 分析结果加入 Nix store 后回注同一组 modules, 由第二次求值检查完整断言并构建 target.

Hull Features

通用

沙盒式评测, 构建, 验证, 打包.

正常造题系统该有的我们都有.

Traits

- ×: 声明每个 test case 属于哪些 subtask.
- ✓: 声明每个 subtask 要求有什么 “特性”, 然后自动匹配.

```
traits = {
  w_eq_1 = { };
  n_le_100 = { };
};

subtasks = [
  {
    traits = { w_eq_1 = true; };
    fullScore = 0.2;
  }
  {
    traits = { n_le_100 = true; };
    fullScore = 0.3;
  }
  { fullScore = 0.5; }
];
```

```
inline auto traits(const Input
&input) ->
std::vector<cplib::validator::Trait>
{
  return {
    {"w_eq_1", [&input]() {
      for (const auto &e:
input.edges) if (e.w != 1) return
false;
      return true;
    }},
    {"n_le_100", [&input]() { return
input.n <= 100; }},
  }
}
```

渲染到题面为:

#	Score	w_eq_1	n_le_100
0	0.2	✓	?
1	0.3	?	✓
2	0.5	?	?

Traits

- 防止 subtask 分错 / 分漏, 每个 test case 自动匹配到最严格的 subtask.
- 搬题时无需考虑 subtask 对应关系, 改为自动化分配.
- Hack 直接加入对应 subtask.

Typst 集成

嵌入 traits & subtask 表格, 样例, 题目信息. 基于 template 排版渲染等基本功能支持.

正常造题系统该有的我们都有.

Typst 集成

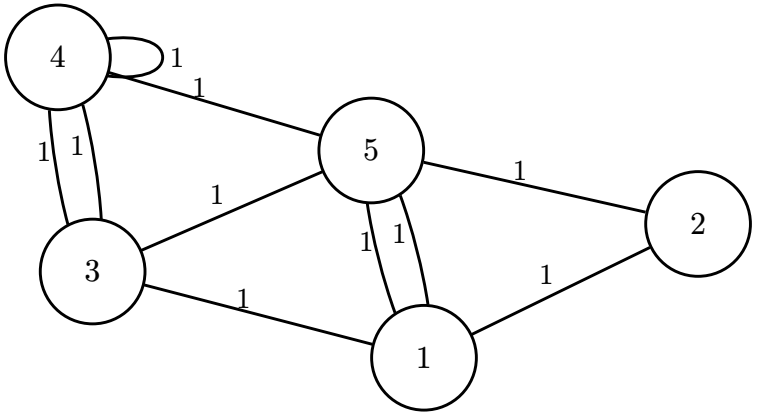
```
struct TestCaseInput {
  std::int32_t idx, n, m;
  std::vector<Edge> edges;

  static auto read(cplib::var::Reader &in, std::int32_t tc_idx) -> TestCaseInput {
    std::int32_t n, m;
    std::tie(n, std::ignore, m, std::ignore) = in(cplib::var::i32("n", 1, 2e5), cplib::var::space,
cplib::var::i32("m", 1, 2e5), cplib::var::eoln);
    auto edges = in.read(cplib::var::Vec(cplib::var::ExtVar<Edge>("edges", n), m,
cplib::var::eoln));
    in.read(cplib::var::eoln);

    if (in.get_trace_level() >= cplib::trace::Level::FULL) {
      in.attach_tag(
        "hull/graph",
        cplib::json::Map{{"name", std::format("graph_{}", tc_idx)},
            {"nodes", std::views::iota(1, n + 1) | std::views::transform([](std::int32_t x) ->
std::string { return std::to_string(x); })},
            {"edges", edges | std::views::transform([](const auto &e) -> cplib::json::Map {
              return {{"u", std::to_string(e.u)}, {"v", std::to_string(e.v)}, {"w",
std::to_string(e.w)}, {"directed", false}}; })}});
      in.attach_tag("hull/case", tc_idx);
    }

    return {.idx = tc_idx, .n = n, .m = m, .edges = std::move(edges)};
  }
};
```

Typst 集成

通过 validator 中的规则直接实现特殊渲染. 例如 Codeforces 风格的 test case 分组染色, 以及图可视化.

样例 1

input	output
3 5 10 1 2 1 1 3 1 3 4 1 4 5 1 3 5 1 1 5 1 2 5 1 5 1 1 3 4 1 4 4 1 5 10	4 1 2 3 4 4 1 2 3 4 4 1 2 3 4



graph_0

hull/case: testcase 分组染色

hull/graph: 图结构可视化

Custom judger

Hull 原生支持用 judger 自定义评测流程而非预制有限种评测流程. 或者说, 现有的传统题, IO 交互题, 提交答案题在 Hull 中都视作一种特殊的 judger.

Eg: UOJ 新年的军队 使用两阶段评测:

$$\text{执行 } 1 \to \text{检查 } 1 \to \text{变换输入} \to \text{验证 } 2 \to \text{执行 } 2 \to \text{检查 } 2$$

资源使用取两个阶段的最大值, 输出两个文件 `first` 与 `second`.

题目可以把前一阶段输出转换为后一阶段输入, 再由 validator 检查转换结果. 最终仍由统一 runtime data 表达整个评测过程.

Custom target

Nix 语言具有极高扩展性.

```
targets = {
  default = hull.problemTarget.common;
  hydro = hull.problemTarget.hydro { statements = { en =
"statement.en.pdf"; }; };
  lemon = hull.problemTarget.lemon { solutionExtNames = lib.mapAttrs
(_: _: "cpp") config.solutions; };
  uoj = hull.problemTarget.uoj { };
  uojAarch64 = hull.problemTarget.uoj {
    targetSystem = "aarch64-linux";
  };
  myTarget = import ./myTarget.nix {}; # 自定义外接 target
};
```

遇到未适配的 OJ 可以编写附加 target 手动适配, 而无需更改 Hull 自身源码.

并行执行

分析任务图

运行时分析包含三类独立任务:

- validator tests
- checker tests
- test cases × solutions

这些任务在有界 Rayon 线程池中并行执行.

Hull 还在每个测试点内并行评测多份解法. Tick 独立于共享墙钟和任务抢占, 因而并行度变化保持时限语义稳定. Custom target 的 scheduler 复用同一并行模型.

并行执行

Kunpeng 实测

硬件	双路 Kunpeng 920, 128 cores
比赛	6 题
评测规模	test case $\times$ solution $>$ 3000
runtime 用时	$<$ 60 s
整场打包用时	$<$ 90 s

该实验来自一场 6 题 2 天真实 NOI 风格模拟赛. load average 长期超过 100, 说明调度器能够有效利用超算节点的多核并行能力. 相较 tuack 达到数十倍效率提升.

并行执行

IOI 赛制的 NOIP

假设全国 NOIP 改用 IOI 赛制 (4 题 5 小时, 每人每题最多提交 50 次), 使用 Hull judger 的基于 WASM 的评测方案, 需要多少台节点?

NOIP 2025 全国共 7,546 个正式参赛者, 四题共 90 个 test case. 通过数学建模不同水平的选手的答题策略, 得到提交次数和提交时间分布. 最终结论为: 平均每人提交 10.81 次, 最后 30 分钟的提交占比 19.78%, 假设全程节点正常工作, 若要保证 95% 的提交在 2 分钟内出结果, 则至少需要 332.7 个 Kunpeng 920 核心 (2.6 个 ARM 超算节点) 或 78.5 个 Intel Xeon Platinum 8358 核心 (1.2 个 x86_64 超算节点).

代码参考项目目录下的 `noip-ioi-capacity/` 目录.

AI Agent Friendly

都 2026 了, 还在人工出题?

我们制作了一个 skill 给 AI 阅读，详细列出了每个步骤的最佳实践. 以后直接直接让 AI 生成题目后人工修改就好了.

Using an AI agent is the recommended way to create a Hull problem. Point an agent that supports Agent Skills to the Hull skill index and ask it to use `author-hull-problems`. The skill archive is also available directly.

AI Agent Friendly

要求 → 题面与解法 → 程序组件 → 数据与子任务 → 校准

- 从用户要求出发, 补齐 workspace 与 Hull 配置.
- 同步生成正式题面、std 与暴力解法, 再实现 validator、checker、generator 等 CPLib 组件.
- 以 traits 描述数据性质, 组织 testcase、subtask 与预测, 依据验证结果迭代校准 limits.
- 最终构建题面和 target, 复核程序、数据、计分语义与交付产物.

GPT 5.6 Sol high 仅根据一句 prompt, 即可将一道 AtCoder ARC C 改编为完整 OI 风格题目, 自动完成题面、std、暴力、数据以及 subtask / traits 划分.

调用示例

- 使用 author-hull-problems skill, 将 https://codeforces.com/contest/XXXX/problem/X 转换为 NOI 风格的模拟赛题目, 要求题面全部重写为中文的形式化题面. 在合理的范围内加大数据范围, 时间限制 1e10 tick. 设置多种且合理的部分分.
- 使用 author-hull-problems skill, 使用 idea [pasted text] 和 std [pasted text], 生成完整的 ICPC 风格题目. 显示语言使用英文.
- 搬运题目 https://loj.ac/p/XXXX, 直接使用原版数据范围, 数据全部重造为强力数据, 直接把原版题面翻译为中文.

AI Agent Friendly

题目	题面	暴力	std	C/V/G	数据	S/T
ULR3 A	D3*	M2		M2		
ULR3 F	D3*	M2		M2*		
省集 A	D3	G54	G54	G54	G54	G54
省集 B	D3	G54	G54P	G54	G54*	
省集 C	D3	G54		G54	G54*	M3
未公开 1	G55	G55	G55	G55	G55	G55
未公开 2	G55	G55		G55	G55*	G55

`C/V/G` = checker / validator / generator, `S/T` = subtask / traits.

空白表示人类实现, * 表示有人类参与.

D3 = DeepSeek V3.2; M2 = Gemini 2.5 Pro; M3 = Gemini 3.1 Pro; G54 = GPT-5.4; G54P = GPT-5.4 Pro; G55 = GPT-5.5.

Hull 部署与比较

部署支持

评级标准

Platinum	部署 Hull runtime, 完整保留评测流程与资源语义
Gold	转换为平台原生评测, 保留 subtask 语义, 只支持传统, 交互, 提答三种常见题目类型
Silver	转换为平台原生评测, subtask 语义受限或无法覆盖三种常见题目类型
Bronze	只能表达基础 batch/pass-fail 模型

评级检查 custom judger, 多输出, validator traits, overlapping subtasks, 逐点限制和资源报告.

部署支持

Platinum: Runtime 随题部署

题目包 → Hull scheduler + WASM runners + Nix closure → OJ

Hydro	`proot` 映射私有 Nix store
UOJ	supervisor 启动完整闭包
Lemon	watcher + special judge + 完整闭包

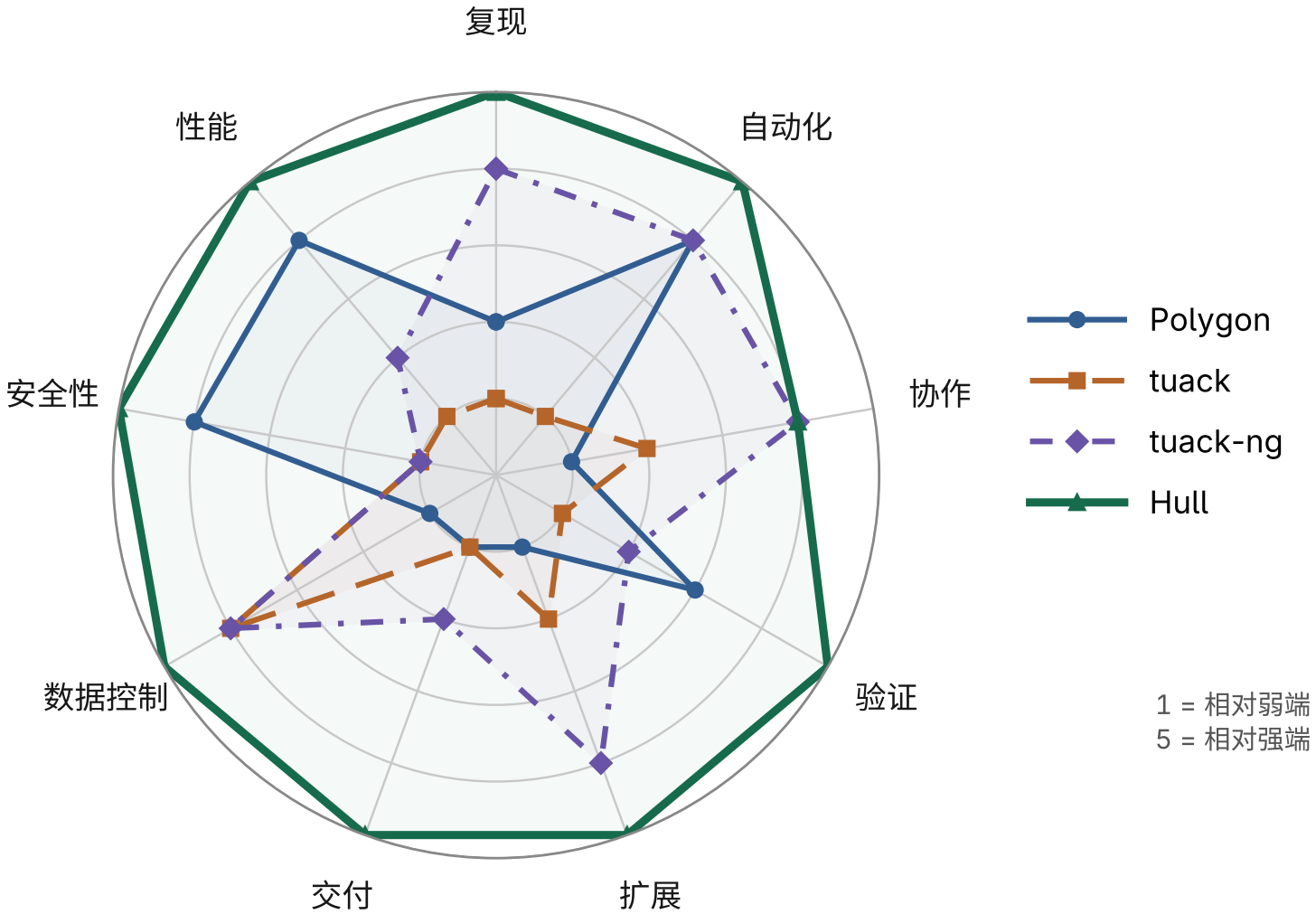
保留 custom judger, 多输出, validator 产生的 traits, overlapping subtasks, 逐点限制和资源报告. OJ 负责接收提交和展示结果, Hull runtime 继续负责完整评测语义. 使用自定义的支持并行加速的评测流程.

部署支持

等级	Targets	语义边界
Gold	UOJ (L), CMS, Luogu	保留 subtask 与部分分
Silver	Hydro (L), Lemon (L)	按平台模型表达分值, 输出和限制
Bronze	DOMjudge	表达 batch 与 pass-fail

(L) 表示 Legacy, 适用于无法取得平台管理员或后台特殊配置权限的情况的使用平台原生评测方案的兼容 target 配置.

同类项目比较


复现
自动化
协作
验证
扩展
交付
数据控制
安全性
性能
Polygon
tuack
tuack-ng
Hull
1 = 相对弱端
5 = 相对强端

同类项目比较

Polygon

- 集中式服务.
- 强制在线操作 WebUI / API, 遇到平台宕机则无法工作.
- 私有线性 revision, 不支持 Git, SVN 等常见版本控制系统.
- 单一 package family, 仅支持导出一种部署格式.
- 评测资源使用官方服务器, 以保证各客户端评测稳定.
- **平台管理员位于私有题目信任域.** *Big Mike is watching you!*

同类项目比较

Tuack

- 本地 Python 工具.
- 依赖宿主工具链.
- QA 和执行模型较弱.
- 平台适配严重陈旧, 缺少现代 OJ target 支持.
- 无法保证因各用户主机速度引起的性能 mismatch.
- 仅使用 ulimit 作为资源限制, 直接使用子进程启动评测进程并在外层轮询 real time 和 memory.

同类项目比较

Tuack-NG

- 本地 Rust CLI.
- 依赖宿主工具链.
- 可重放 seed. 具有一定的可复现性.
- **不支持 validator 检验**.
- Target 数目较少.
- Git 协作和扩展结构较强, 但无法保证因各用户主机速度引起的性能 mismatch.
- **无评测沙盒**, 直接使用子进程启动评测进程并在外层轮询 real time 和 memory.

同类项目比较

Hull

- Local-first 与 Git-native: 题目源文件留在作者控制的目录中, 可直接接入 review、hooks 与 CI.
- Nix 固定依赖和构建环境, WASM tick 固定评测尺度, 最终产物可 bit-to-bit 复现.
- validator / checker tests、solution predictions、traits 与 subtasks 共同构成一致性验证体系.
- Custom judger 可表达多阶段、多输出评测; custom target 可将 Hull runtime 与完整语义部署到 OJ.
- 声明式配置和 `author-hull-problems` skill 让 Agent 能从最小题目规格推进到数据、校准与交付.

Local-first 趋势

本地文件 → Git → hooks / CI → Agent → 发布平台

集中式 Web GUI 正在失去默认创作入口地位. 平台退回同步, 审批与发布控制面.

题面, 解法, validator, checker 和 generator 放回本地目录. Git 保存权威历史, hooks 和 CI 执行检查, 平台只在需要协作或发布时同步.

Hull 结论

目前的应用

Hull 已被应用于山东三轮省集的命题, 以及 UOJ Long Round 3 的 A, F 题命题工作.

预计将在今年于 UOJ 举办一场 UOJ WASM Round, 尽请期待!

WAAP 背景与动机

现有代码查重方案的局限

静态文本分析

目前的查重系统 (如 JPlag, Moss) 主要基于源码文本或抽象语法树 (AST) .

- **标识符重命名**: 通过批量替换变量名、函数名, 轻松绕过基于 Token 的匹配.
- **结构混淆**: 利用 OLLVM 等工具进行控制流平坦化, 使 AST 结构完全改变.
- **等价逻辑替换**: 将 `for` 循环重写为 `while` + `goto`, 或展开递归函数, 导致静态特征完全失效.

现有代码查重方案的局限

动态分析

- **平台依赖性强**: 依赖 `ptrace` (Linux) 调试 API (Windows)，难以跨平台.
- **性能开销大**: 基于 Hook 的系统调用拦截会严重拖慢评测速度.
- **安全性风险**: 在评测机直接运行不可信的二进制代码进行动态插桩，存在沙盒逃逸风险.

WAAP 的核心理念

固定输入，将代码在实际运行时的表现编码为向量.

WAAP 算法设计详解

Winnowing 指纹生成算法

记录运行时所有指令类型, 忽略操作数, 映射为枚举值.

$$A = [t_1, t_2, t_3, ..., t_N]$$

设定窗口大小 W (如 8) , 计算连续 W 个指令的 Rolling Hash.

$$h_i = H(t_i, t_{i+1}, ..., t_{i+W-1})$$

设定块大小 B (如 128) . 在每个滑动窗口覆盖的范围内, 选取 **Hash 值最大** 的那个作为指纹.

$$f_j = \max\{h_k \mid k \in \text{Range}_j\}$$

Winnowing 指纹生成算法

算法特性

- **压缩率**: 数据量减少为原来的 $\frac{1}{B}$, 空间复杂度 $O\left(\frac{N}{B}\right)$.
- **抗干扰**: 如果选手在代码中插入了一条垃圾指令, 只会影响包含该指令的 W 个 Hash 值. 只要这 W 个 Hash 值没有成为新的局部最大值, 最终生成的指纹序列 F 甚至可能 **完全不变**.

相似度计算模型

将生成的指纹序列转换为向量空间模型 (Vector Space Model).

统计指纹序列 F 中每个特征 Hash 出现的频率, 构建稀疏向量.

$$\boldsymbol{V}_A = \{(h, \text{count}) \mid h \in F_A\}$$

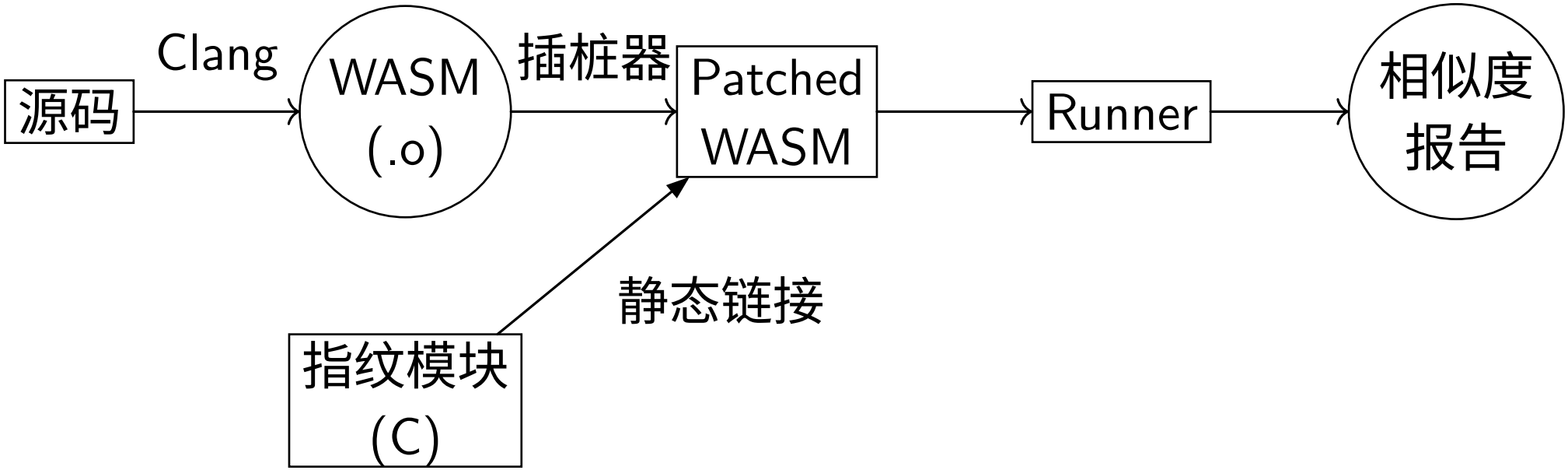
计算两个代码向量的夹角余弦值, 值越接近 1 表示越相似.

$$\text{Sim}(A, B) = \frac{\boldsymbol{V}_A \cdot \boldsymbol{V}_B}{\|\boldsymbol{V}_A\| \|\boldsymbol{V}_B\|}$$

相比于编辑距离 (Levenshtein) 的 $O(N^2)$ 复杂度, 向量点积仅需 $O(\frac{N}{B})$, 能够实现大规模比对.

WAAP 系统架构与实现

处理流程图



模块内记录

如果在 WASM 中每执行一条指令就调用 `call host_function()`, 频繁的 VM 上下文切换会导致运行速度极慢, 无法用于实际评测.

我们将记录逻辑下放到 WASM 模块内部:

1. 编写一个 C 语言模块, 包含 `record(type)` 函数, 内部维护一个 Ring Buffer 和 Hash 状态机.
2. 通过修改二进制 (Binary Patching) , 在每个 Basic Block 和关键指令前插入 `call $record`.
3. 仅在程序结束时, 通过一次 Host Call 导出内存中的指纹数组.

指令归一化策略

为了进一步消除编译器优化的干扰，我们将 WASM 指令映射为粗粒度的类别：

类别	包含指令	设计意图
Control	`if`, `br`, `call`, `block`	捕获程序的逻辑骨架
Arith	`add`, `sub`, `mul` (i32/i64)	忽略类型宽度，关注计算本质
Mem	`load`, `store`	捕获数据访问模式
Bit	`and`, `or`, `shl`, `xor`	位运算特征
Ignored	`const`, `nop`, `drop`	**抗干扰**：忽略常量防止魔改

WAAP 抗攻击分析

场景 1: 变量重命名与类型混淆

攻击手段: 选手将 `int a` 改为 `long long b`, 并将所有变量名替换为随机字符串.

WAAP 表现:

- WASM 编译后变量名消失, 转为 Index.
- 指令分类时合并了 `i32` 和 `i64` 的算术指令 (如 `i32.add` 和 `i64.add` 均视为 `ADD`) .
- 生成的指令序列完全一致, 相似度 100%.

场景 2: 控制流平坦化

攻击手段:

使用 OLLVM 将清晰的 `if-else` 逻辑打碎成一个巨大的 `switch-case` 状态机.

WAAP 表现:

- 静态分析会看到 CFG (Control Flow Graph) 变得极其复杂.
- 动态执行流关注的是 **实际执行的路径**.
- 尽管中间插入了分发块 (Dispatcher) 的指令, 但核心计算逻辑的指令 **执行顺序** 依然被保留.
- Winnowing 算法会保留核心逻辑片段的 Hash, 相似度依然很高.

场景 3: 死代码注入与循环展开

攻击手段: 在代码中随机插入大量无用的计算 (如 `int x = 1*2*3...`) , 或手动展开循环.

WAAP 表现:

- 死代码: 由于 Winnowing 选取局部 Hash 最大值, 插入的垃圾指令大多产生的 Hash 值较小, 会被算法筛掉.
- 循环展开: 循环展开本质上是指令序列的重复. 向量空间模型基于频率统计, 重复的特征只会增加对应维度的权重, 而不会改变向量的方向 (余弦相似度对向量长度不敏感) .

WAAP 总结

应用前景

注意到这个方案把选手程序转化成了一组向量，其实际起到的作用是一个 embedding.

相似度只是两个向量最基本的特征.

- 聚类?
- 解法自动归类?
- OI 外的应用?

总结

WASM 的下一步应用

WASM 下运行结果与硬件无关，Serverless OJ?