

# WebAssembly Differential Testing at Scale

A Multi-Runtime Fuzzing and Minimization Pipeline



## Bug and Vulnerability Discovery in WebAssembly Runtimes

## Differential Testing Framework Design & Evaluation

(A cross-runtime differential testing system for Wasmtime, Wasmer, and V8)

Group 2

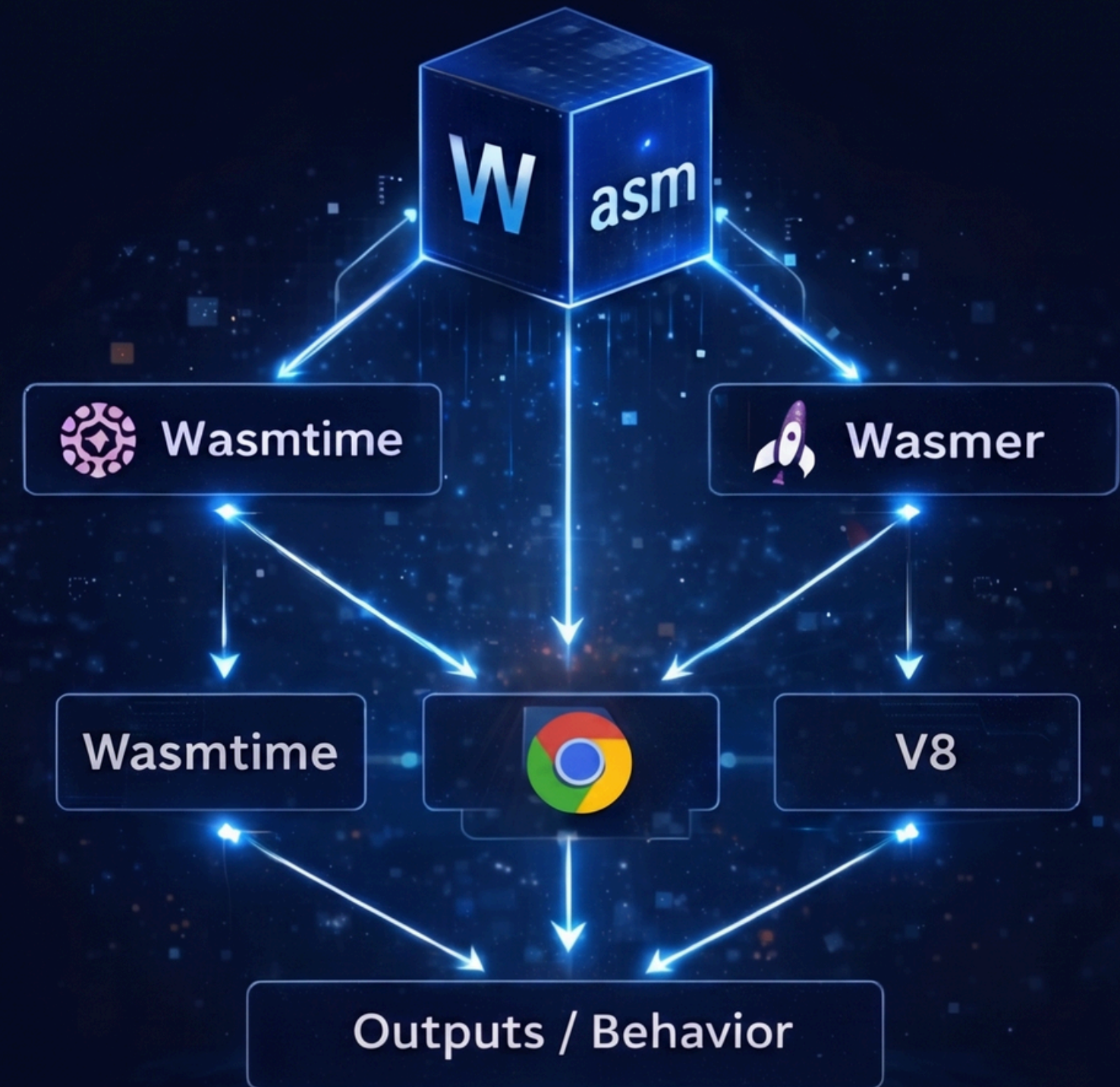
- Anika
- Ronghao
- Sahil
- Sajid



# Why Test WebAssembly Runtimes?

- WebAssembly runs across multiple independent runtimes (**Wasmtime**, Wasmer, V8)
- Same program SHOULD behave identically everywhere
- In practice:
  - different validation behavior
  - different trap handling
  - different output semantics
- These inconsistencies can indicate:
  - correctness bugs
  - spec interpretation differences

**Key Idea:** If two runtimes disagree on the same program, at least one of them is **wrong** (or underspecified).





# Project Objective

Build an end-to-end differential testing framework for WebAssembly runtimes that:

- Executes the same Wasm program across multiple runtimes

(Wasmtime, Wasmer, V8 subset)

- Captures execution results:

- stdout
- stderr
- exit codes
- execution time

- Normalizes runtime-specific outputs into comparable categories:

- ok
- trap
- validation\_error
- runtime\_error

- Detects behavioral mismatches across runtimes

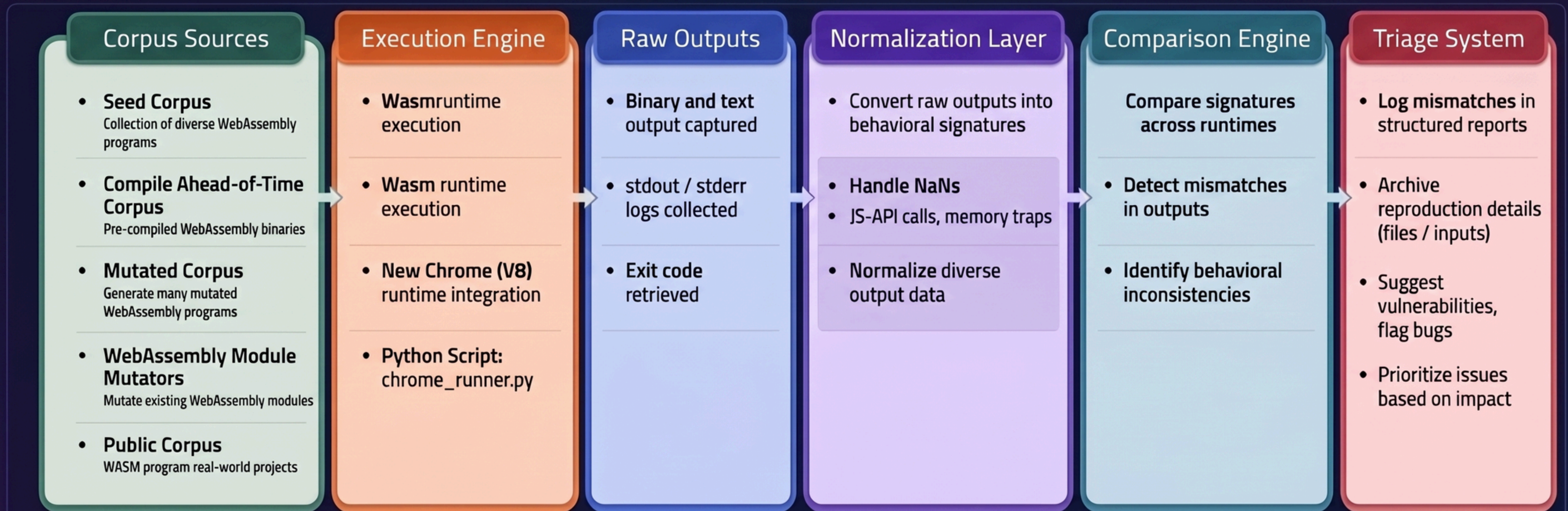
- Generates structured analysis artifacts:

- summary.json
- mismatches.json
- metrics.md

Goal: Move from raw execution noise → structured, comparable, and analyzable behavior across runtimes



# End-to-End Framework Architecture

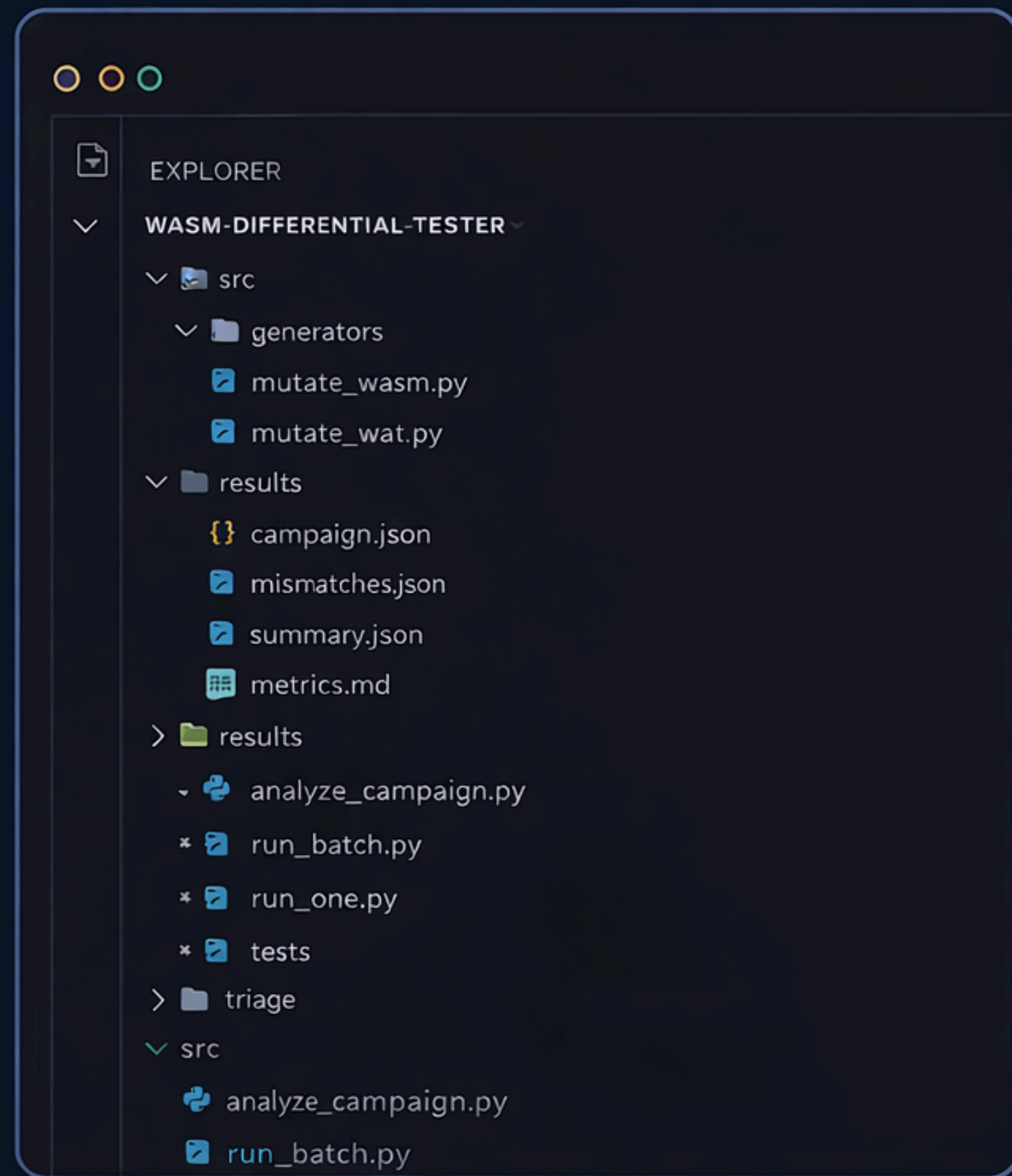


## Key Insight:

A single WebAssembly program flows through multiple runtimes, is normalized into a structured behavioral signature, and then compared to uncover real inconsistencies.



# Framework Components (Actual Implementation)



## Core Components of Our Framework

- **Corpus Layer**
  - Seed programs, public corpus, compiler-generated, mutated inputs
- **Execution Layer**
  - Wasmtime, Wasmer, Chrome ( V8)
  - Same Wasm executed across all runtimes
- **Data Capture Layer**
  - stdout, stderr, exit code, execution time
  - Stored in JSONL format
- **Normalization Layer**
  - Converts outputs into structured signatures:  
`✓ | trap | validation_error | runtime_error`
- **Comparison Engine**
  - Compares behavior across runtimes
- **Analysis & Reporting**



**Key Idea:** Same program → Different **runtime** ➡ ➡ Compare behavior



# Test Pipeline Overview

```
src > python3 src/run_batch.py
9   from dataclasses import asdict, dataclass
10  from pathlib import Path
11  from typing import Any, Dict, List, Optional, Tuple
12
13
14  @dataclass
15  class RunRecord:
16      test_id: str
17      wasm_path: str
18      runtime: str
19      runtime_version: str
20      status: str          # ok | trap | timeout | validation_error | runtime_error | err
21      exit_code: Optional[int]
22      duration_ms: int
23      stdout: str
24      stderr: str
25
26
27  TRAP_PATTERNS = [
28      re.compile(r"divide by zero", re.IGNORECASE),
29      re.compile(r"unreachable", re.IGNORECASE),
30      re.compile(r"out of bounds", re.IGNORECASE),
31      re.compile(r"wasm trap", re.IGNORECASE),
32      re.compile(r"\btrap\b", re.IGNORECASE),
33      re.compile(r"RuntimeError", re.IGNORECASE),
34  ]
35
36  VALIDATION_PATTERNS = [
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS POSTMAN CONSOLE

```
● (.venv) sahil@Sahils-MacBook-Air wasm-differential-tester % python3 src/run_batch.py
Wrote 216 run records to results/campaign.jsonl
● (.venv) sahil@Sahils-MacBook-Air wasm-differential-tester % python3 src/analyze_campaign.py
Wrote: results/summary.json
Wrote: results/mismatches.json
Wrote: results/metrics.md
Wrote triage bundles to: results/triage
Mismatches: 1 / 104
○ (.venv) sahil@Sahils-MacBook-Air wasm-differential-tester %
```

Example: return\_42.c → return\_42.wasm

C Program / WAT Input

Compiled to WebAssembly  
(.wasm)

Executed Across Runtimes  
(Wasmtime | Wasmer | Chrome V8)

Raw Outputs Captured  
(stdout, stderr, exit code, time)

Normalization Layer  
(ok | trap | validation\_error | runtime\_error)

Comparison Engine  
(signature + stdout equivalence)

Analysis Output  
(summary.json, mismatches.json)



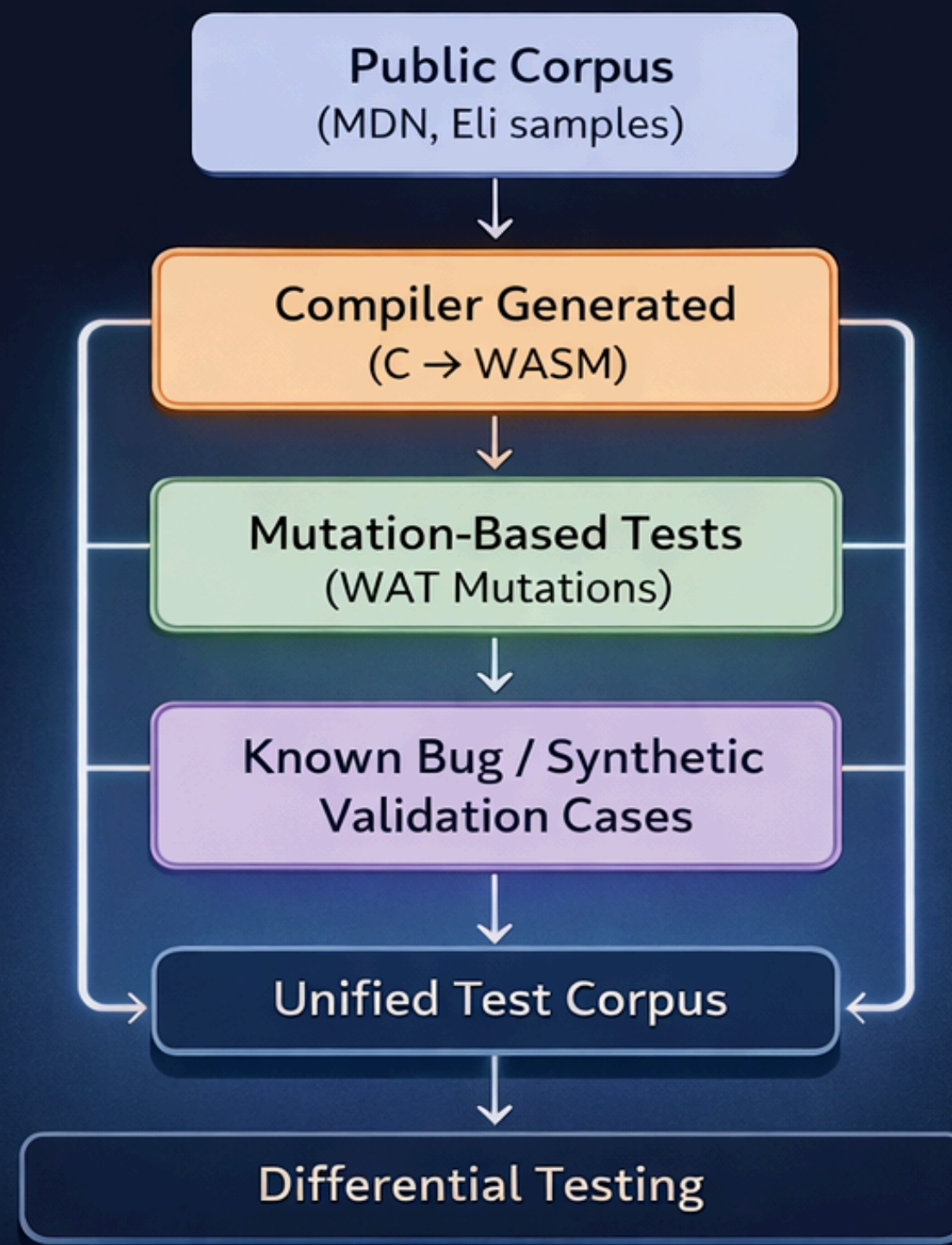
# Corpus Strategy: Multi-Source Test Design

## Why Corpus Design Matters

- Differential testing is only as strong as its inputs
- Poor inputs → no bugs
- Diverse inputs → higher chance of divergence

## Our 4-Layer Corpus Strategy

- 1 **Public Corpus** (Real-world programs)
  - Source: MDN + Eli Bendersky examples
  - arithmetic, recursion, memory operations
  - **Purpose:** baseline correctness
- 2 **Compiler-Generated Corpus**
  - C → WASM (via clang)
  - signed/unsigned division, bitwise ops
  - **Purpose:** realistic compiled behavior
- 3 **Mutation-Based Corpus**
  - WAT-level mutations:
    - instruction swaps (**add** → **sub**)
  - constant changes
  - **Purpose:** validate framework effectiveness





# Compiler-Generated Corpus ( $C \rightarrow \text{WASM}$ )

## Why Compiler-Generated Inputs?

- Real-world programs are often compiled from C/C++
- Helps simulate realistic execution paths
- Produces structured and valid WebAssembly modules

### Process Pipeline:

- Write simple C programs
- Compile using clang  $\rightarrow$  wasm
- Generate .wasm binaries
- Feed into differential testing pipeline

### Examples we used:

- signed / unsigned division
- bitwise operations
- memory load/store
- recursion

```
tests > compiler_generated > src > C mutual_recursion.c
1  int is_even(int n);
2  int is_odd(int n);
3
4  int is_even(int n) {
5      if (n == 0) return 1;
6      return is_odd(n - 1);
7  }
8
9  int is_odd(int n) {
10     if (n == 0) return 0;
11     return is_even(n - 1);
12 }
13
14 __attribute__((export_name("entry")))
15 int entry() {
16     return is_even(8);
17 }
```

### Command Used:

```
clang --target=wasm32 -o mutual_recursion.wasm mutual_recursion.c
```

To generate realistic inputs, we compiled C programs into WebAssembly using clang. This allowed us to test how real compiled binaries behave across different runtimes.



# Why We Avoided Binary Mutation

## Binary-Level Mutation Challenges

- Direct .wasm mutation often breaks structure
- Generates invalid or malformed modules
- Leads to validation errors instead of execution differences

## Observed in Our Experiments

- Majority of mutated binaries → validation\_error
- Very low execution coverage
- No meaningful runtime divergence

## Conclusion

Binary mutation is noisy, low-signal

## Our Design Decision

### Shift mutation to WAT-level (text format)

- Preserves structure
- Produces valid modules
- Enables meaningful testing



# WAT-Level Mutation Engine (Our Approach)

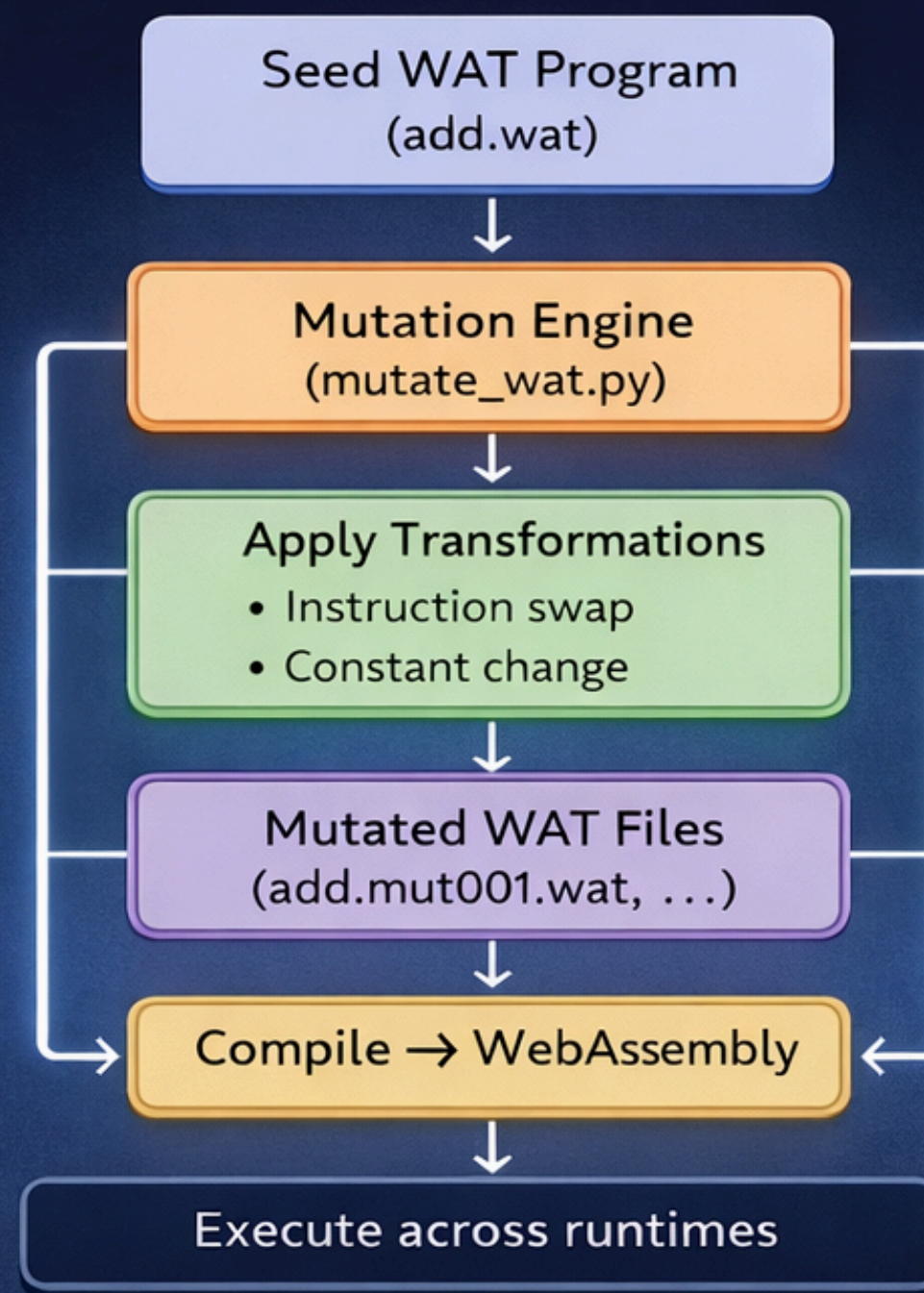
## Why WAT-Level Mutation?

- Operates on structured text format (.wat)
- Preserves module validity
- Enables controlled semantic changes

## Our Mutation Strategy

- **Input**
  - Seed programs (add.wat, call.wat, recursion.wat)
- **Mutation Types**
  - **Instruction Mutation**
    - i32.add → i32.sub
    - i32.mul → i32.add
  - **Constant Mutation**
    - i32.const 1 → i32.const 2
  - **Logic Variation**
    - Modify recursion / branching
- **Output**
  - Mutated .wat files
  - Compiled → .wasm
  - Executed across

```
{  seed: 'add.wat',
  "mutput::": "add.mut001.wat",
  "mutation": {
    "from": "i32.add",
    "to": "i32.sub"
  }
}
```



WAT-level mutation ensures executable diversity while preserving structural correctness.



# What Worked Best?

## Empirical Evaluation of Testing Strategies

### WAT-Level Mutation

(Best Performer)

- Generated valid yet behaviorally diverse programs
- Preserved structural **correctness** → high execution rate
- Enabled **controlled** semantic changes (instruction swap, constants)
- Contributed most to:
  - Higher “ok” **execution cases**
  - Increased runtime coverage

**Key Insight:** Structure-aware mutation is critical for effective differential testing

### Compiler-Generated Corpus

(High Quality Baseline)

- Produced well-formed, realistic **Wasm** modules via clang
- Covered:
  - Arithmetic operations
  - Memory interactions
  - Recursion and control flow
- Provided:
  - Stable execution baseline
  - Reduced noise from invalid inputs

**Key Insight:** Compiler-generated inputs improve reliability but lack diversity

### Public Corpus

(Real-World Coverage)

- Included **standard** Wasm examples and edge cases
- Helped validate:
  - Function exports
  - Memory **behavior**
  - Control flow correctness
- Challenges:
  - Missing entry points (`_start`)
  - Argument mismatch issues

**Key Insight:** Real-world samples require normalization and handling logic

**Final Strategy:** Combining structured inputs + mutation gave the best balance of coverage, validity, and **diversity**.



# Execution Harness

- Executes WebAssembly modules across multiple runtimes:

- Wasmtime (CLI)
- Wasmer (CLI)
- Chrome V8 (Playwright/Chromium)

- Uses subprocess-based orchestration:

- Isolated execution per runtime
- Prevents crashes from affecting pipeline

- Captures detailed execution signals:

- stdout → program output
- stderr → runtime errors
- exit\_code → execution status
- duration\_ms → performance metric

- Produces structured run records (JSONL):

- One record per runtime per test
- Enables deterministic analysis

```
{  
  "test_id": "25fbd3de98919c36b7d5b5e954d0bb03c210e728a3eaedda0c5526f2289c051d",  
  "wasm_path": "tests/compiler_generated/wasm/subtraction.wasm",  
  "runtime": "wasmer",  
  "runtime_version": "wasmer 7.0.1",  
  "status": "ok",  
  "exit_code": 0,  
  "duration_ms": 8,  
  "stdout": "42",  
  "stderr": ""  
}
```



# Real Challenges Faced

## 1. High Rate of Invalid Wasm (Binary Mutation Failure)

- Initial binary mutation produced mostly malformed modules
- Result:
  - Extremely high validation\_error rates
  - Very low meaningful execution coverage
- Root Cause:
  - Random byte mutations break Wasm structure

### 👉 Solution:

- Switched to WAT-level mutation
- Preserved syntax → improved execution success

## 2. Runtime Invocation Inconsistencies

- Different runtimes require different execution models:
  - Missing\_start function
  - Exported functions require explicit invocation
- Example:
  - Wasmer fails without --invoke
  - Wasmtime silently succeeds

### 👉 Solution:

- Built invocation handling logic
- Standardized execution across runtimes



# Real Challenges Faced

## 3. Output Inconsistency Across Runtimes

- Same program produced:
  - Different stdout formats
  - Or no output at all
- Example:
  - One runtime prints return value
  - Another returns silently

- 👉 Solution:
- Designed stdout normalization layer
  - Introduced:
    - `equivalent_stdout()` logic

## 4. False Positives in Differential Testing

- Initial mismatches caused by:
  - Logging differences
  - Formatting inconsistencies
- Not actual behavioral differences

- 👉 Solution:
- Built signature-based comparison system
  - Compared:
    - status (ok/trap/validation\_error)
    - normalized outputs

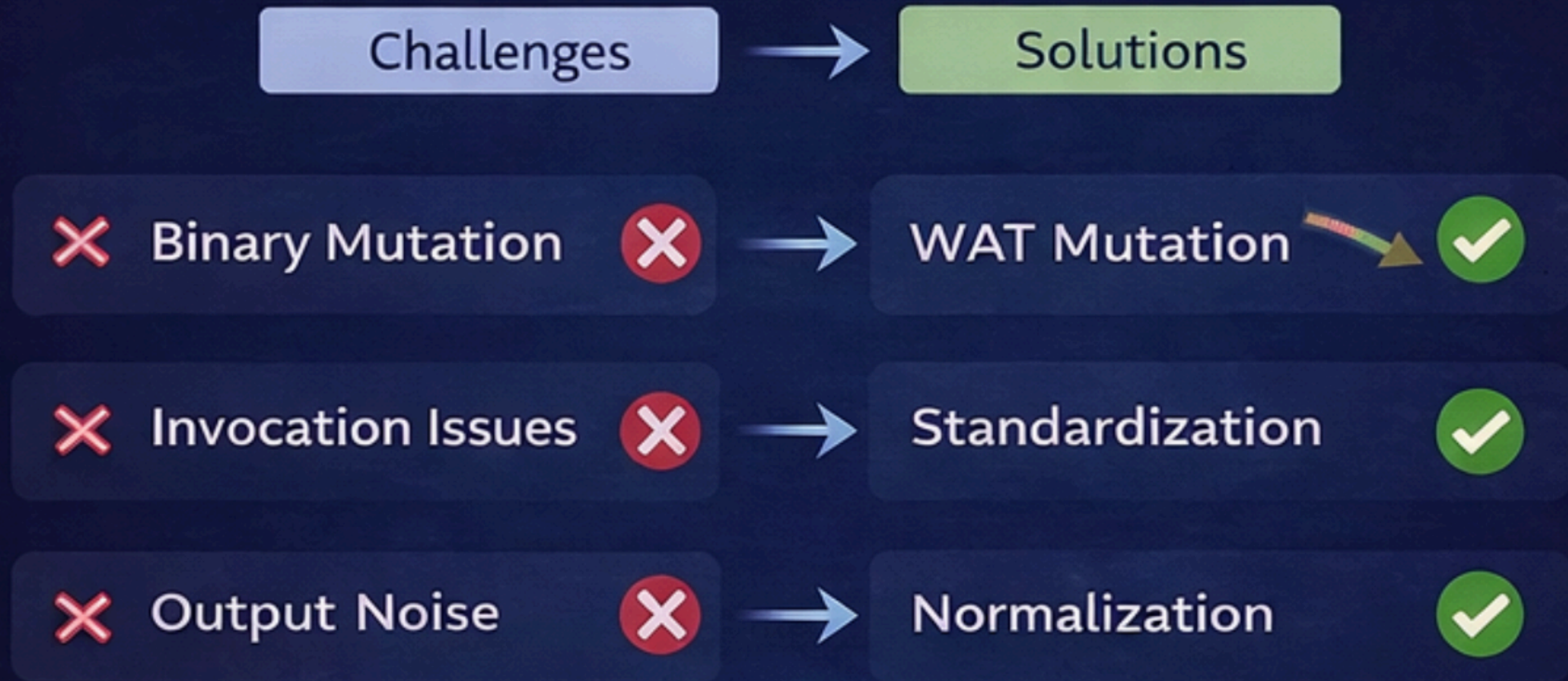
## 5. Low Mismatch Discovery (Reality Check)

- Despite large corpus:
  - 0 real mismatches initially found
- Challenge:
  - Hard to discover real bugs in modern runtimes

- 👉 Solution:
- Introduced:
    - Synthetic mismatch case (NaN canonicalization)
    - Known bug replay strategy



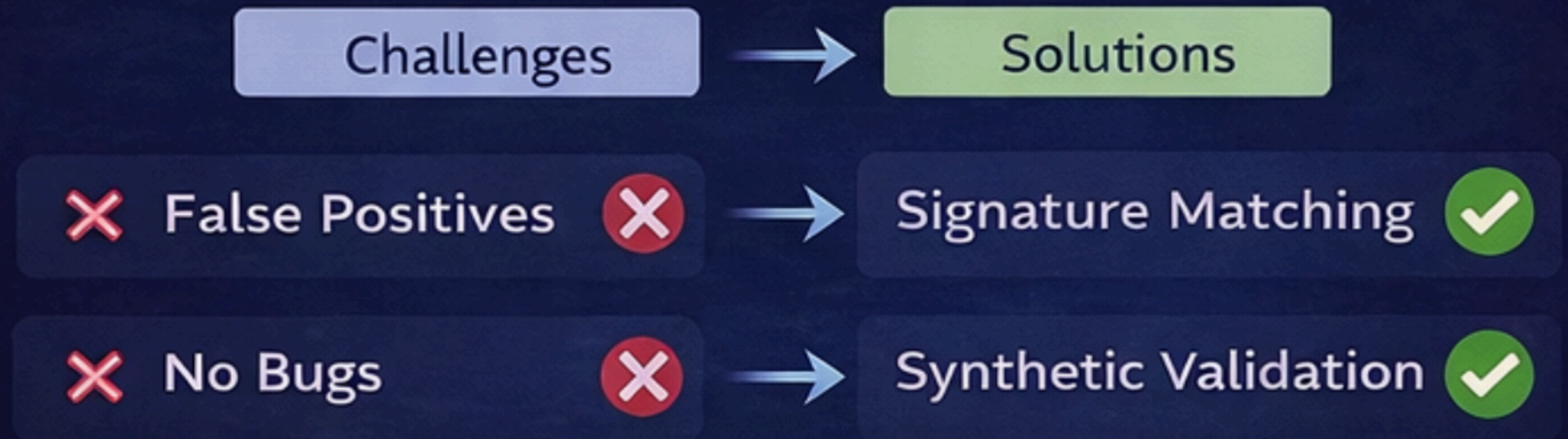
# Challenges & Solutions (1/2)



Handling runtime diversity requires **structured mutation** and **systematic** invocation logic.



# Challenges & Solutions (2/2)



Differential testing necessitates robust normalization and validation strategies.



# Engineering Fixes

Key adjustments that made the system reliable

| Problem                                                                                                                                                                                                                           | Fix                                                                                                                                                                                                                                                   | Impact                                                                                                                                                                                            |
|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Entrypoint Normalization</b> <ul style="list-style-type: none"><li>Missing entrypoints:<br/>Different runtimes <i>expected</i><br/>different entrypoints (<code>_start</code>, <code>exports</code>)</li></ul>                 | <b>Dynamic detection in 'run_batch.py':</b> <ul style="list-style-type: none"><li>Detects: <code>__start</code></li><li>exported functions (e.g., <code>'entry'</code>, <code>'main'</code>)</li></ul>                                                | <b>Higher execution success</b> <ul style="list-style-type: none"><li>Eliminated execution failures due to missing entrypoints</li><li>Increased valid execution coverage significantly</li></ul> |
| <b>Manifest Argument Support</b> <ul style="list-style-type: none"><li>No argument support:<br/>Modules requiring arguments could not be tested</li></ul>                                                                         | <b>Manifest system:</b> <ul style="list-style-type: none"><li>Defined: function to invoke</li><li>arguments to pass</li></ul>                                                                                                                         | <b>Real-world testing</b> <ul style="list-style-type: none"><li>Supported complex test cases</li><li>Improved realism of testing pipeline</li></ul>                                               |
| <b>Classification Ordering Fix</b> <ul style="list-style-type: none"><li>Misclassification:<br/>Mismatch between <code>validation_error</code>, <code>trap</code>, <code>runtime_error</code> → caused false mismatches</li></ul> | <b>Ordered logic</b> <ul style="list-style-type: none"><li>Introduced strict classification priority:<pre># classification priority if validation_error:     status:= "validation_error" elif trap: elif runtime_error &gt; ok" else:</pre></li></ul> | <b>Fewer false positives</b> <ul style="list-style-type: none"><li>Reduced false positives</li><li>Made comparison engine deterministic</li></ul>                                                 |



# Scaling Up: Mutation Challenges in Wasm Testing

## 1 Large-Scale Mutation Run

- Generated and executed 1,427 test programs
- 2,856 total runtime executions
- Compared across Wasmtime and Wasmer

## 2 What We Observed

- 1,300+ cases per runtime ended in validation\_error
- Only a small fraction reached meaningful execution
- 34 raw mismatches were observed, but many came from invalid or low-quality inputs

## 3 Key Insight



Naive mutation creates mostly invalid Wasm modules. In WebAssembly differential testing, input quality matters more than input quantity.

## 4 Our Decision

- Treat the large-scale run as an exploratory experiment
- Use it to motivate structure-aware mutation
- Present a curated, valid subset for the main evaluation and demo



**1427**  
tests



**2856**  
executions



**>90%**  
validation errors



**34**  
raw mismatches



**Conclusion:** Structure-aware, validity-preserving inputs gave us the best balance of coverage, executability, and meaningful comparison.



# Key Observations from Actual Results

After running our campaign, these were the main observations from the generated results:

- We executed 100+ comparable tests across Wasmtime and Wasmer, producing 200+ execution records.
- A large portion of failures were validation errors, showing that input validity was the main bottleneck.
- After applying our normalization layer, we found no natural semantic mismatches across the main corpus.
- This suggests that the two runtimes behaved consistently for the tested inputs.
- To verify that our framework can still catch real divergences, we then introduced a synthetic mismatch case, which was successfully detected in the next step.



# Runtime Behavior Analysis

## What our framework actually observed

```
results > {} summary.json > ...
1  {
2    "num_tests": 104,
3    "num_records": 216,
4    "records_per_runtime": {
5      "wasmtime": 108,
6      "wasmer": 108
7    },
8    "status_counts_by_runtime": {
9      "wasmtime": {
10       "ok": 37,
11       "trap": 7,
12       "validation_error": 64
13     },
14     "wasmer": {
15       "ok": 37,
16       "trap": 7,
17       "validation_error": 64
18     }
19   },
20   "trap_categories_by_runtime": {
21     "wasmtime": {
22       "divide_by_zero": 1,
23       "oob_memory": 3,
24       "unreachable": 3
25     },
26     "wasmer": {
27       "divide_by_zero": 1,
28       "oob_memory": 3,
29       "unreachable": 3
30     }
31   },
32   "validation_error_categories_by_runtime": {
33     "wasmtime": {
34       "type_mismatch": 5,
35       "unknown": 56,
36       "invalid": 3
37     },
38     "wasmer": {
39       "type_mismatch": 5,
40       "unknown": 56,
41       "invalid": 3
42     }
43   },
44   "runtime_error_categories_by_runtime": {},
45   "num_mismatches": 1
46 }
```

```
results > {} mismatches.json > ...
1  [
2    {
3      "test_id": "8b60bb8168f30dc6edd370f887608d409f215db7323bd70b9504beca01fb4acd",
4      "wasm_path": "tests/known_bug_replay/wasm/synthetic_nan_canonicalization_mismatch.wasm",
5      "reason": "Mismatch in: stdout",
6      "by_runtime": {
7        "wasmtime": {
8          "test_id": "8b60bb8168f30dc6edd370f887608d409f215db7323bd70b9504beca01fb4acd",
9          "wasm_path": "tests/known_bug_replay/wasm/synthetic_nan_canonicalization_mismatch.wasm",
10         "runtime": "wasmtime",
11         "runtime_version": "wasmtime 41.0.3 (db1c043b5 2026-02-04)",
12         "status": "ok",
13         "exit_code": 0,
14         "duration_ms": 4,
15         "stdout": "",
16         "stderr": ""
17       },
18       "wasmer": {
19         "test_id": "8b60bb8168f30dc6edd370f887608d409f215db7323bd70b9504beca01fb4acd",
20         "wasm_path": "tests/known_bug_replay/wasm/synthetic_nan_canonicalization_mismatch.wasm",
21         "runtime": "wasmer",
22         "runtime_version": "wasmer 7.0.1",
23         "status": "ok",
24         "exit_code": 0,
25         "duration_ms": 7,
26         "stdout": "NaN",
27         "stderr": ""
28       }
29     }
30   ]
31 }
```



# Cross-Engine Coverage with Headless Chrome



Why a third runtime:

- V8 is the most-deployed wasm engine in the world (every browser)
- Two runtimes can disagree, but you can't tell who's wrong
- Three runtimes enable majority voting as a debugging signal

| MAJORITY VOTING                      |      |     |
|--------------------------------------|------|-----|
| Wasmtime                             | ok → | ∅   |
| Wasmer                               | ok → | NaN |
| Chrome V8                            | ok → | NaN |
| CONCLUSION                           |      |     |
| Wasmtime is the outlier → Bug signal |      |     |



# Cross-Engine Coverage with Headless Chrome

How it works:

1. Launch headless Chromium with Playwright
2. Inject a JS harness: WebAssembly.compile → instantiate → call export → capture result/exception
3. Wrap output as RunRecord — identical schema to wasmtime/wasmer, drops cleanly into the existing comparison logic



# Cross-Engine Coverage with Headless Chrome

## Two integration considerations

- Chrome's error wording differs from server runtimes (e.g. "expected type f32, found i32" vs "type mismatch") — extended the normalization patterns to recognize both
- Symmetry with the other runtimes: missing-export must produce a record, not silently disappear from the comparison

```
# Validation error categories
VALIDATION_CATS = [
    ("type_mismatch", re.compile(r"type mismatch", re.IGNORECASE)),
    # chrome wording: "expected type f32, found i32" / "not enough arguments on the stack"
    ("type_mismatch", re.compile(r"expected (?<type>)?\w+, found", re.IGNORECASE)),
    ("type_mismatch", re.compile(r"not enough arguments on the stack", re.IGNORECASE)),
    ("unknown_opcode", re.compile(r"unknown (opcode|operator)", re.IGNORECASE)),
    ("section_size_mismatch", re.compile(r"section size mismatch", re.IGNORECASE)),
    ("trailing_bytes", re.compile(r"trailing bytes", re.IGNORECASE)),
    ("unexpected_end", re.compile(r"unexpected end", re.IGNORECASE)),
    ("parse_failure", re.compile(r"failed to parse|unknown binary version", re.IGNORECASE)),
    ("malformed", re.compile(r"\bmalformed\b", re.IGNORECASE)),
    ("unexpected", re.compile(r"unexpected", re.IGNORECASE)),
    ("validation_failure", re.compile(r"validation", re.IGNORECASE)),
    ("invalid", re.compile(r"\binvalid\b", re.IGNORECASE)),
    ("invalid", re.compile(r"illegal opcode", re.IGNORECASE)),
    ("invalid", re.compile(r"local index out of bounds", re.IGNORECASE)),
    ("invalid", re.compile(r"unknown local", re.IGNORECASE)),
]
```

```
def run_chrome_record(wasm_path: str, timeout_s: int, manifest: Dict[str, Any]) -> Optional[RunRecord]:
    try:
        import sys
        sys.path.insert(0, str(Path(__file__).parent.parent))
        from runners.chrome_runner import run_chrome
    except ImportError:
        return None

    entry_info = choose_entrypoint(wasm_path, manifest)
    if entry_info is None:
        # symmetric with wasmtime/wasmer: record the same synthetic validation_error
        return make_missing_entry_record("chrome", "chromium", wasm_path)
```



# Minimizer

- Binary blob is useless as a bug report
- Need the smallest bytes that still trigger the same mismatch

```
1 chunk_size = body_size / 4
2 while chunk_size >= 1:
3     while improved:
4         for each offset i in body:
5             try removing bytes [i, i + chunk_size)
6             if same mismatch still reproduces → keep cut
7             else → skip
8     chunk_size //= 2
```

cuts quarter of the file), then halve chunk size each pass until single-byte precision



# What Made the Minimizer Robust

## *i.* **Single source of truth for runtime invocation**

The minimizer imports `run_wasmtime` / `run_wasmer` / `run_chrome_record` directly from the runner module — detection and reduction can never drift apart.

## *ii.* **Shape-preserving reduction**

Required invariant: the set of normalized signatures across runtimes must stay identical at every step — guarantees the minimal output reproduces the same bug, not just any bug.

## *iii.* **Per-iteration manifest injection**

Each iteration constructs a temp manifest mapping the temp path to the original's export + args — lets the reduction loop reuse the production invocation path without special cases.



# DETECTION → REDUCTION → REPRODUCTION

## i. DETECTION

Framework detects the disagreement

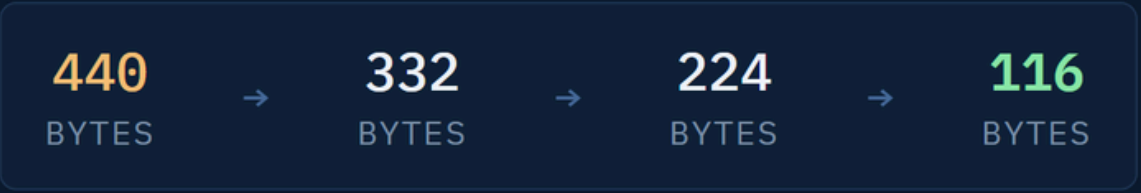
```
results/mismatches.json
{
  "test_id": "8b60bb8168f30dc6edd370f887608d409f215db7323bd70b9504beca01fb4acd",
  "wasm_path": "tests\\known_bug_replay\\wasm\\synthetic_nan_canonicalization_mismatch.wasm",
  "reason": "Mismatch in: stdout chrome='NaN' vs wasmtime=''; stdout wasmer='NaN' vs wasmtime='',
  "by_runtime": {
    "wasmtime": {
      "test_id": "8b60bb8168f30dc6edd370f887608d409f215db7323bd70b9504beca01fb4acd",
      "wasm_path": "tests\\known_bug_replay\\wasm\\synthetic_nan_canonicalization_mismatch.wasm",
      "runtime": "wasmtime",
      "runtime_version": "wasmtime 42.0.1 (6844a83b5 2026-02-25)",
      "status": "ok",
      "exit_code": 0,
      "duration_ms": 13,
      "stdout": "",
      "stderr": ""
    },
    "wasmer": {
      "test_id": "8b60bb8168f30dc6edd370f887608d409f215db7323bd70b9504beca01fb4acd",
      "wasm_path": "tests\\known_bug_replay\\wasm\\synthetic_nan_canonicalization_mismatch.wasm",
      "runtime": "wasmer",
      "runtime_version": "wasmer 7.0.1",
      "status": "ok",
      "exit_code": 0,
      "duration_ms": 14,
      "stdout": "NaN",
      "stderr": ""
    },
    "chrome": {
      "test_id": "8b60bb8168f30dc6edd370f887608d409f215db7323bd70b9504beca01fb4acd",
      "wasm_path": "tests\\known_bug_replay\\wasm\\synthetic_nan_canonicalization_mismatch.wasm",
      "runtime": "chrome",
      "runtime_version": "chrome 120.0.6321.114",
      "status": "ok",
      "exit_code": 0,
      "duration_ms": 15,
      "stdout": "NaN",
      "stderr": ""
    }
  }
}
```

Wasmtime returns nothing. Wasmer and Chrome both return NaN.

## ii. REDUCTION

Minimizer shrinks the file

```
PowerShell
Minimizing: tests/known_bug_replay/wasm/synthetic_nan_canonicalization_mismatch_padded.wasm
Original size: 440 bytes (body: 432 bytes)
Reduced to 332 bytes (removed 108B at offset 108)
Reduced to 224 bytes (removed 108B at offset 108)
Reduced to 116 bytes (removed 108B at offset 108)
Minimized size: 116 bytes
Repro pack saved to: repros\\synthetic_nan_ca
```



440 bytes down to 116. Each cut validated against all three runtimes — 74% reduction.

## iii. REPRODUCTION

Self-contained repro package

```
explorer
repros
  2b84a50760667582
    minimal.wasm
    original.wasm
    repro.md
  8b60bb8168f30dc6
```

```
repro.md
## Expected vs Observed

| Runtime | Status | stdout | stderr (truncated) |
|-----|-----|-----|-----|
| wasmtime | ok | '' | '' |
| wasmer | ok | 'NaN' | '' |
| chrome | ok | 'NaN' | '' |

## Files
- Original: `tests\\known_bug_replay\\wasm\\synthetic_nan_canonicalization_mismatch_padded.wasm` (440 bytes)
- Minimal: `minimal.wasm` (116 bytes)
```

Minimal binary, original binary, markdown report. Ready to file as a bug ticket.



# System Impact & Future Work

## System Impact

- Built an end-to-end differential testing framework
- Enabled cross-runtime validation (Wasmtime, Wasmer, Chrome V8)
- Designed a modular and extensible architecture
- Achieved structured and reproducible testing pipeline
- Successfully detected runtime divergence (synthetic validation case)

## Future Work

- Add more runtimes (WasmEdge, V8 standalone, Wasm3)
- Integrate coverage-guided fuzzing (AFL / libFuzzer style)
- Expand real-world bug replay corpus
- Automate test generation at scale (10K+ programs)
- Build better dashboard / visualization UI for results

From a testing framework → to a scalable WebAssembly reliability platform.





# What We Learned Through This Project

- Raw runtime outputs cannot be compared directly.

Normalization was essential to distinguish real semantic mismatches from superficial reporting differences.

- Input quality matters more than input quantity.

Running thousands of tests is not useful if most inputs are invalid and fail during validation.

- Structure-aware mutation is far more effective than naive binary mutation.

WAT-level mutation gave us more valid and meaningful executable test cases.

- Differential testing needs careful harness design.

Entry-point handling, invocation consistency, and output capture strongly affect correctness of comparison.

- No mismatch does not mean no value.

Even when natural mismatches were not found, the framework still revealed important insights about runtime consistency and corpus quality.

- Validation through a synthetic mismatch was important.

It confirmed that the framework can detect real divergences when they actually occur.

