# Detecting Compiler-Introduced Security Bugs via IR Mutation and Coverage-Guided Fuzzing

**Selim Oh**, Seulbae Kim

selim0h@postech.ac.kr, seulbae@postech.ac.kr

Eurosec 2026 | April 27 | Edinburgh, UK

POSTECH CompSec Lab

# What are CISBs?

Compiler-Introduced Security Bugs (CISBs)

```
void process_data(int offset) {
    // Security check intended to detect integer wrapping
    if (offset + 10 < offset) {
        abort();
    }
    /* Subsequent security-critical logic */
}
```

# What are CISBs?

Compiler-Introduced Security Bugs (CISBs)

```
void process_data(int offset) {
    // Security check intended to detect integer wrapping
    if (offset + 10 < offset) {
        abort(); // Silently optimized away!
    }
    /* Subsequent security-critical logic */
}
```

# What are CISBs?

Compiler-Introduced Security Bugs (CISBs)

```
void process_data(int offset) {
    if (offset < 0)
        return;
    /* Subsequent security-critical logic */
}
```

**Compiler's No-UB optimization assumption**
conflicts with **the developer's security intent!**

# How to detect CISBs – Challenges

## Challenge 1: Reachability

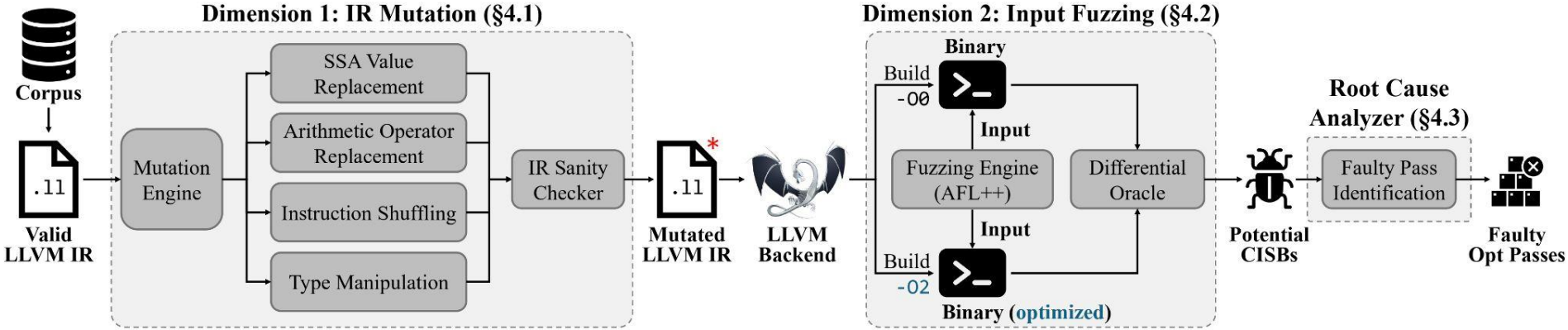
Source-level fuzzers are often blocked by frontend optimization, failing to stress deep middle-end optimization logic.

## Challenge 2: Input-Dependency

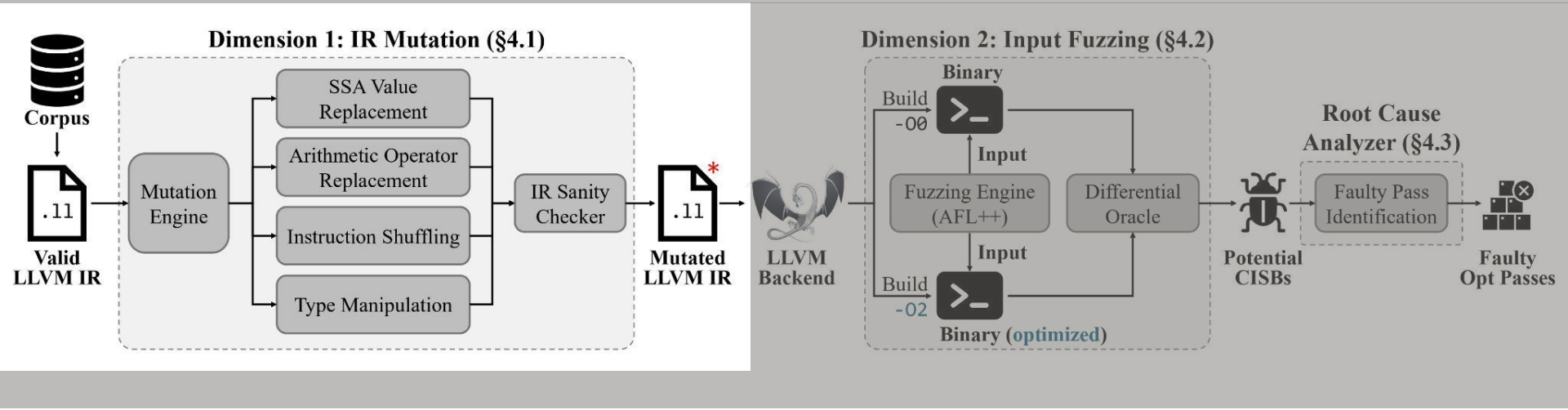
Many CISBs only manifest under specific runtime values.

**Directly target LLVM IR and explore the input space using fuzzing.**

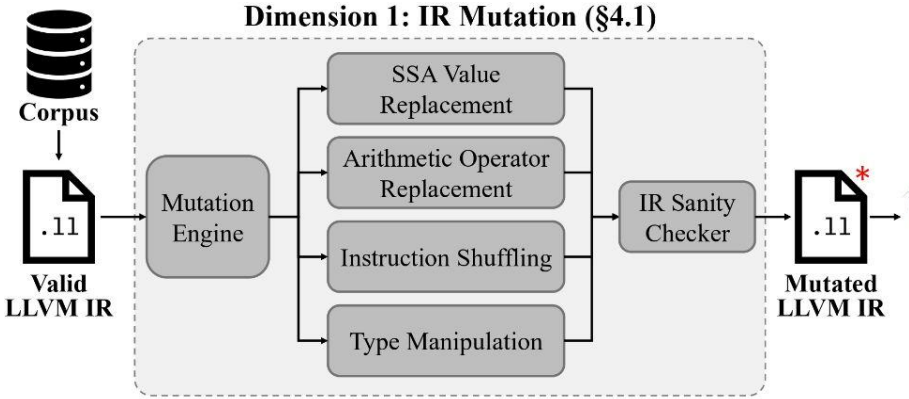
# CISBFuzz


Corpus
.ll
Valid LLVM IR
Dimension 1: IR Mutation (§4.1)
Mutation Engine
SSA Value Replacement
Arithmetic Operator Replacement
Instruction Shuffling
Type Manipulation
IR Sanity Checker
.ll
*
Mutated LLVM IR
LLVM Backend
Dimension 2: Input Fuzzing (§4.2)
Binary
Build -O0
Input
Fuzzing Engine (AFL++)
Differential Oracle
Input
Build -O2
Binary (optimized)
Potential CISBs
Root Cause Analyzer (§4.3)
Faulty Pass Identification
Faulty Opt Passes

# CISBFuzz – Dimension 1: IR Mutation



# CISBFuzz – Dimension 1: IR Mutation




ORIGINAL IR

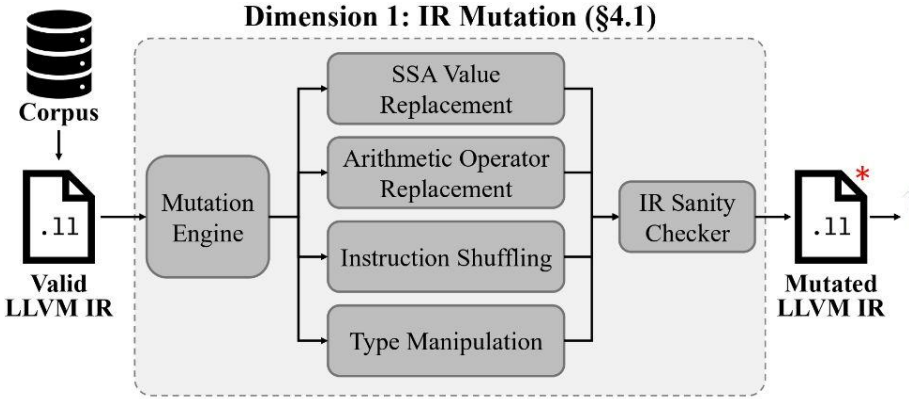
```
%cmp = icmp sgt i32 %a, 10
br i1 %cmp, label %T, label %F
```

MUTATED IR (SSA VALUE REPLACEMENT)

```
%cmp = icmp sgt i32 %a, undef
br i1 %cmp, label %T, label %F
```

SSA Value Replacement: Injecting **undef** into a comparison

# CISBFuzz – Dimension 1: IR Mutation




ORIGINAL IR

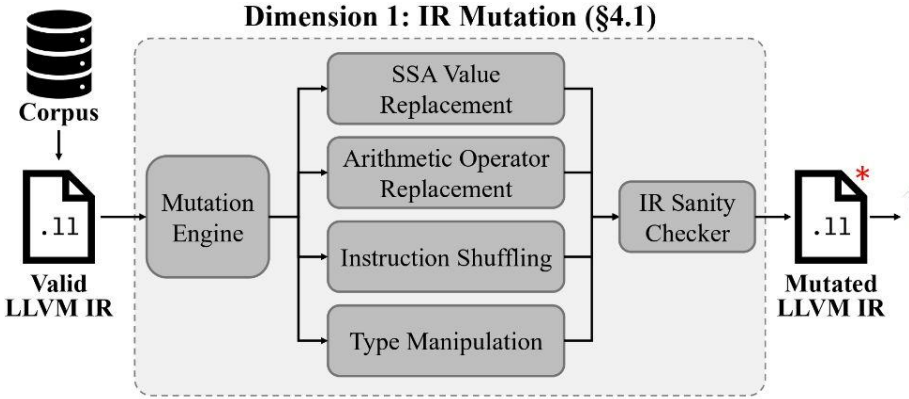
```
%res = add nsw i32 %a, %b
```

MUTATED IR (AOR)

```
%res = add i32 %a, %b
```

AOR: Toggling overflow flags to alter semantics

# CISBFuzz – Dimension 1: IR Mutation




ORIGINAL IR

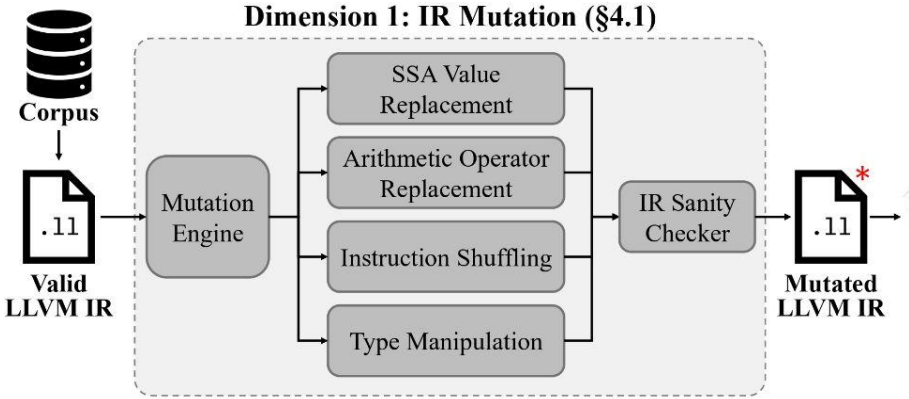
```
store i32 10, ptr %p
%val = load i32, ptr %q
```

MUTATED IR (Reordered)

```
%val = load i32, ptr %q
store i32 10, ptr %p
```

Instruction Shuffling: Reordering independent instructions

# CISBFuzz – Dimension 1: IR Mutation




ORIGINAL IR

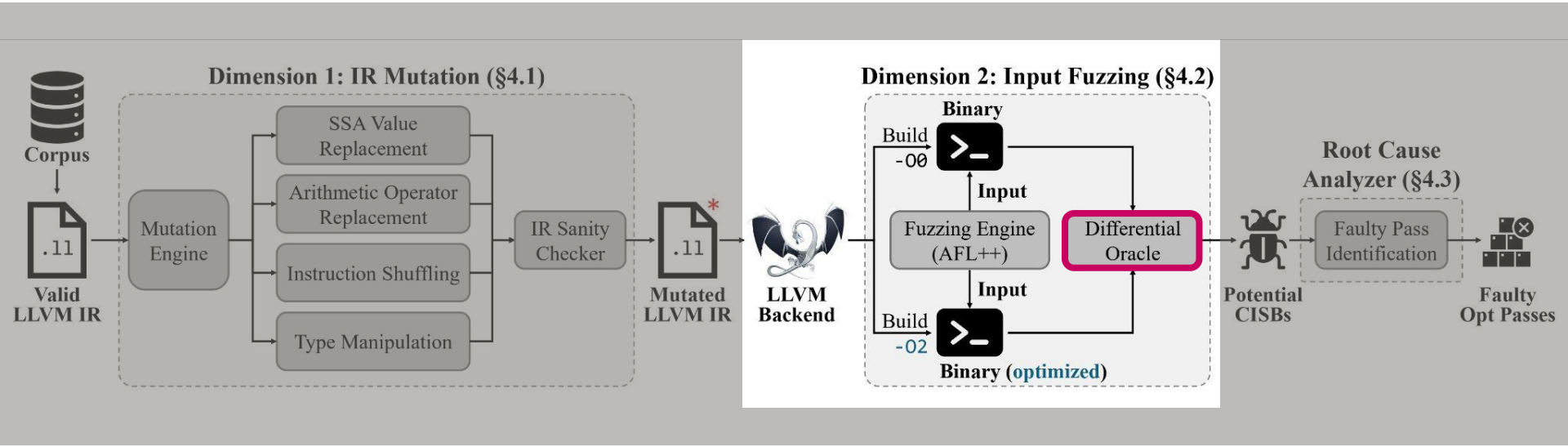
```
%res = add i32 %a, %b
```

MUTATED IR (i32->i26)

```
%1 = trunc i32 %a to i26
%2 = trunc i32 %b to i26
%3 = add i26 %1, %2
```

Type Manipulation: Changing types to irregular ones

# CISBFuzz – Dimension 2: Input Fuzzing



# CISBFuzz – Dimension 2: Input Fuzzing

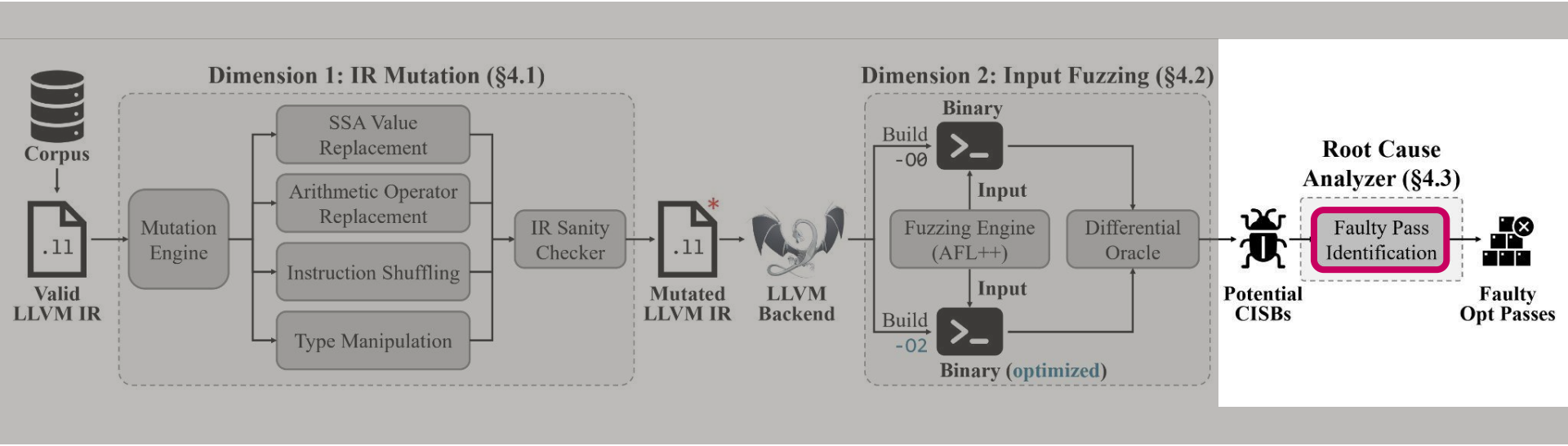
```
def detect_cisb(mutant_ir, fuzz_input):
    # 1. Generate two binaries from the same Mutated IR
    bin_base = compile(mutant_ir, opt_level="-O0")
    bin_opt = compile(mutant_ir, opt_level="-O2")

    # 2. Execute both with the same fuzzer-generated input
    res_base = execute(bin_base, fuzz_input)
    res_opt = execute(bin_opt, fuzz_input)

    # 3. Differential Analysis
    if res_base != res_opt:
        return CISB_DETECTED # Output Divergence (Logic Error)
    if is_crash(res_opt) and not is_crash(res_base):
        return CISB_DETECTED # Optimization-induced Crash

    return NO_BUG
```

# CISBFuzz – Root Cause Analyzer



# CISBFuzz – Root Cause Analyzer

## Binary Search Strategy for pinpointing the culprit

### 01. Bisect Pipeline

Selectively limit the number of optimization passes using LLVM's `opt-bisect-limit`.

### 02. State Check

For each midpoint in the binary search, verify if the security gate still exists in the IR.

### 03. Convergence

**Check remains:**
Culprit is in the **later half**.

**Check erased:**
Culprit is in the **earlier half**.

# Evaluation: New CISBs Found

### Experimental Setup

- Built on LLVM 18
- 34 seed cases → **340 mutants**

### Noise Filtration

IR Sanity Checker **filtered 110** semantically invalid cases

### Input Fuzzing

**1 hour** per mutant

## **Key Finding: 23 New CISBs** discovered in LLVM 18

- All identified bugs are **silent logic errors**, not compiler crashes
- **Root Cause:** Semantic misinterpretation of undef propagation
- Successfully reported to **LLVM developers**

# Case Study: Silent Security Check Erasure (SSCE)

**ORIGINAL IR**

```
define i32 @security_gate(i32 %user_id) {
entry:
%id.addr = alloca i32, align 4
%val = add nsw i32 undef, 10 ; Source
store i32 %user_id, ptr %id.addr, align 4
%loaded = load i32, ptr %id.addr, align 4
%cmp = icmp sgt i32 %val, %loaded ; Use of undef
br i1 %cmp, label %passed, label %failed
```

**EarlyCSE**

**OPTIMIZED IR**

```
define i32 @security_gate(i32 %user_id) {
entry:
%id.addr = alloca i32, align 4
store i32 %user_id, ptr %id.addr, align 4
br i1 false, label %passed, label %failed ; Folded
```

# Discussion

## 1. Can current frontends really emit these buggy IR patterns?

- Reachability vs. Robustness
  - IR-to-Backend pipeline must remain against all valid IR.
  - Evolution of languages (e.g., Rust) can lead to new IR emissions.
  - Our focus: Stress-testing backend pipeline

## 2. Is this a "Real Bug"?

- LLVM Developer Consensus: UB prevention is the programmer's sole responsibility, thus behaviors are ***intended optimizations.***

**⇒ Need for Secure-by-default compilers that warn of or freeze security-sensitive undef values.**

# What’s next?

## 1. System Enhancement & Automation

**End-to-End Automation**

**Pass-Aware Root Cause Refinement**

**Guided Mutation**

# What’s next?

## 2. Expanding Scope & Mitigation

**Broader Bug Coverage**

**Compiler-Level Mitigation**

# Conclusion

IR Mutation + Coverage-guided Fuzzing = CISB Discovery

Uncovered 23 previously unknown CISBs in the LLVM 18 optimizer

Primarily driven by the aggressive exploitation of undef semantics

Vulnerabilities are difficult to pinpoint due to cross-pass propagation.

Static analysis and single-execution tests are insufficient; coverage-guided fuzzing is essential.

# Questions

CompSec Lab @ POSTECH
https://compsec.postech.ac.kr/

Selim Oh
https://selim0h.github.io/