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

Selim Oh
selim0h@postech.ac.kr
Pohang University of Science and Technology
Pohang, South Korea

Seulbae Kim*
seulbae@postech.ac.kr
Pohang University of Science and Technology
Pohang, South Korea

Abstract

Compilers translate source code into executable binaries without altering their intended semantics. To improve performance, modern compilers apply a wide range of optimizations, such as eliminating redundant computations and removing code deemed unnecessary, that analyze and transform programs under strong semantic assumptions. However, aggressive optimizations designed to maximize performance can inadvertently introduce security vulnerabilities that did not exist in the source code. These are referred to as Compiler-Introduced Security Bugs (CISBs), and they are particularly dangerous as they often silently eliminate security-critical checks (*e.g.*, bounds checks) under the guise of removing dead code. Existing research on CISBs has largely relied on source-level mutation or formal verification, which often fails to exercise deep optimization logic or lacks scalability.

In this paper, we propose CISBFuzz, a two-dimensional automated testing framework designed to proactively detect CISBs within the `LLVM`'s optimization pipeline. CISBFuzz first mutates the `LLVM` IR (Intermediate Representation) of pre-optimization test programs to directly stress-test optimization passes. It then applies coverage-guided input fuzzing to the optimized binaries compiled from the mutants to trigger CISBs whose manifestation depends on specific runtime inputs. Running CISBFuzz on `LLVM 18`, we uncover 23 previously unknown CISBs.

CCS Concepts

• **Security and privacy → Systems security**.

Keywords

Compiler Security, Compiler-Introduced Security Bugs

ACM Reference Format:
Selim Oh and Seulbae Kim. 2026. Detecting Compiler-Introduced Security Bugs via IR Mutation and Coverage-Guided Fuzzing. In *19th European Workshop on Systems Security (EuroSec '26), April 27–30, 2026, Edinburgh, Scotland Uk*. ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3803525.3804981

*Corresponding Author

EuroSec '26, Edinburgh, Scotland Uk

ACM ISBN 979-8-4007-2603-3/2026/04
https://doi.org/10.1145/3803525.3804981

1 Introduction

Compilers are the bedrock of software security. Developers rely on the assumption that if they write secure source code, the resulting binary will also be secure. However, when modern compilers, such as `LLVM` and `GCC`, optimize code to improve runtime performance, they often exploit Undefined Behavior (UB) semantics, assuming that programs will never execute undefined operations. When this assumption clashes with defensive programming practices, the compiler may optimize away essential security checks, such as `NULL` pointer checks or integer overflow guards, and end up introducing Compiler-Introduced Security Bugs (CISBs) [8].

Detecting CISBs is notoriously difficult. Traditional software testing approaches focus extensively on finding bugs in application code, rather than in the compiler itself. In contrast, existing compiler testing tools such as `Csmith` [9] generate random C programs to expose robustness issues in compilers. Such an approach often struggles to trigger specific corner cases in the compiler's middle-end (*i.e.*, the IR optimization pipeline) because the frontend may normalize, rewrite, or reject such patterns before optimization occurs. Even when programs successfully reach the optimizer, there is no guarantee that optimization-induced security bugs will manifest at runtime. Furthermore, formal verification tools like `Alive2` [6] provide strong guarantees but are computationally expensive, making them difficult to apply to large-scale fuzzing campaigns.

To bridge this gap, we present CISBFuzz, a novel framework that operates directly on the `LLVM` Intermediate Representation (IR). Instead of mutating source code, CISBFuzz mutates valid `LLVM` IR to generate optimization-sensitive patterns. It then employs a coverage-guided fuzzer to explore the input space of the optimized binaries, allowing us to detect input-dependent CISBs that manifest only under specific runtime conditions.

Our contributions are as follows:

- **Two-Dimensional Detection Framework:** We introduce a hybrid approach that combines semantics-aware IR mutation with fuzzing. This enables the detection of both optimization crashes and logic errors where security checks are silently removed.
- **Noise Reduction via Cross-Validation:** We implement an automated validation oracle that cross-checks results against unoptimized (`-O0`) binaries, effectively filtering out 86% of false positives caused by invalid mutants.
- **Challenges in Root Cause Isolation for `undef`:** We report 23 new CISBs found in LLVM 18. Our root cause analysis reveals a systemic weakness regarding `undef` propagation. Our analysis reveals that identifying the specific faulty pass is non-trivial due to `undef` propagation. We demonstrate how current isolation techniques can lead to misattribution (*e.g.*, to `IPSCCP`), as multiple passes allows `undef` to reach the backend.

```
1 void process_data(int offset) {
2   // Security check intended to detect integer wrapping
3   if (offset + 10 < offset) {
4     abort(); // Silently optimized away!
5   }
6   /* Subsequent security-critical logic */
7 }
```

Figure 1: An example of a CISB involving integer overflow. The compiler assumes that signed overflow never occurs, leading it to conclude that the condition is always false and removing the safety guard.

2 Motivating Example

We first describe how a seemingly benign compiler optimization can introduce a security vulnerability by examining a real Compiler-Introduced Security Bug (CISB) involving integer overflow.

The Code. Figure 1 shows a simplified code snippet in which an integer overflow check is performed (Line #3) to prevent wrapping. The developer explicitly includes the overflow check, *i.e.*, `if (offset + 10 < offset)` at Line #3, to ensure that the program aborts before executing security-critical logic if the input would lead to an overflow. This pattern is a common defensive programming practice intended to prevent memory corruption or logic errors caused by integer wrapping.

The Bug. Modern compilers, such as `GCC` and `LLVM`, perform optimizations under the assumption that the source program does not exhibit Undefined Behavior (UB). This assumption is often referred to as the No-UB model [8]. Under this model, the compiler processes the code as follows:

(1) Compiler observes the signed addition `offset + 10` (Line #3).
(2) According to the C standard, signed integer overflow is UB. Therefore, under the No-UB assumption, the compiler infers that the addition will never wrap around to a value smaller than the original offset.
(3) Consequently, the compiler deduces that the condition `offset + 10 < offset` at Line #3 must always evaluate to `false`.
(4) Applying dead-code elimination optimization, the compiler removes the safety guard (`abort()`) from the generated binary.

Security Impact. While this transformation is semantically correct given strict compliance with the language specification, it introduces a *security vulnerability*. If an attacker can control the input such that `offset` is close to `INT_MAX`, the resulting binary will bypass the intended safety check, allowing the subsequent logic to proceed with a wrapped-around value. This discrepancy between the developer's security intent (protecting against hardware-level wrapping) and the compiler's optimization assumptions (assuming standard-compliant behavior) exemplifies the core characteristic of Compiler-Introduced Security Bugs (CISBs).

2.1 Challenges of Detecting CISBs

This motivating example highlights two fundamental challenges that make CISBs difficult to detect using existing testing techniques.

Challenge 1: Reaching Optimization-Sensitive States. CISBs originate from interactions within the compiler's middle-end optimization pipeline, where transformations operate directly on `LLVM` IR under assumptions about undefined or underspecified semantics. While the frontend aggressively validates and normalizes source programs, this process both filters inputs and rewrites program structure, limiting the ability of source-level testing to precisely control the IR patterns presented to the optimizer. Consequently, many optimization-sensitive states are difficult to reach or reproduce from source code. Targeting `LLVM` IR directly is therefore necessary to systematically stress-test optimization passes responsible for CISBs.

Challenge 2: Input-Dependent Manifestation. Even when an optimization silently removes or corrupts a security check, the resulting vulnerability may not manifest under a single execution or fixed test input. For example, as illustrated in Figure 1, the elimination of the overflow check does not cause any observable failure for the vast majority of integer inputs where overflow does not occur. Many CISBs preserve functional correctness for most inputs and only manifest when specific inputs interact with the optimized-away logic. Systematically uncovering such bugs therefore requires exploring the input space of optimized binaries.

3 Background

LLVM and Intermediate Representation (IR). The `LLVM` compiler infrastructure follows a modular three-phase design: the frontend (*e.g.*, `Clang`), the optimizer (middle-end), and the backend. The core of this architecture is the `LLVM` IR, a low-level, strongly typed, RISC-like instruction set. The optimizer performs a series of transformations (passes) on the IR to improve performance. Our research targets this middle-end optimizer, as it is responsible for the most aggressive semantic transformations.

Undefined Behavior and Optimization. In `LLVM`, Undefined Behavior (UB) allows the compiler to assume that certain program states or executions never occur. Within `LLVM` IR, this model is realized through special values such as `undef`, which denotes an uninitialized or unconstrained value. Optimizers may use `undef` to perform dead code elimination (DCE) or constant folding. For example, if a variable is `undef`, the compiler might assume it can be any value that simplifies the code. However, if a developer writes a security check involving a variable that the compiler determines to be `undef` the compiler may decide the check is redundant and remove it. This disconnect between the developer's intent and the compiler's assumption is a primary source of CISBs.

Compiler Fuzzing. Fuzzing is an automated software testing technique that provides invalid, unexpected, or random data as inputs to a program to monitor for exceptions. In compilers, fuzzing generally falls into two categories: source-level and IR-level. Source-level fuzzers, such as `Csmith`, generate random C programs to test the compiler starting from the frontend. While effective for finding backend crashes, they are often limited by frontend constraints and simplifications, making it difficult to stress specific middle-end optimizations. In contrast, IR-level fuzzing (*e.g.*, `alive-mutate`) applies mutations directly to the Intermediate Representation, enabling fine-grained control over instruction sequences and types. CISBFuzz builds upon this IR-level mutation strategy but improves it by integrating coverage-guided input fuzzing. Unlike previous tools that typically rely on a single execution of the generated binary, CISBFuzz uses `AFL++` to execute the compiled binary with diverse inputs, enabling the discovery of bugs triggered only by specific runtime values.

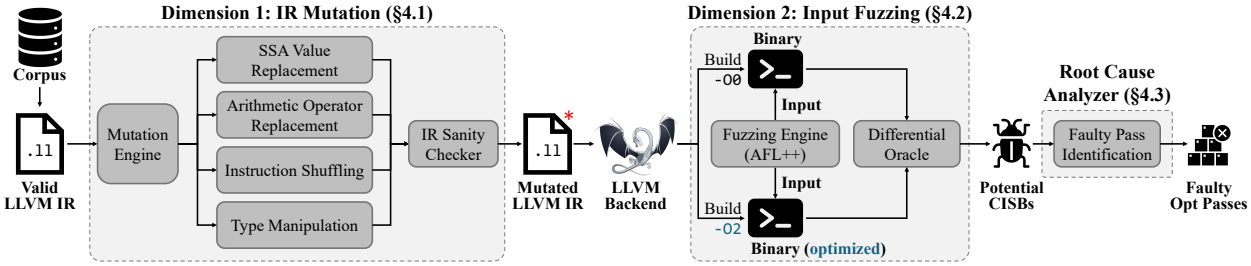


Figure 2: The overview of CISBFuzz pipeline.

4 Design

We propose CISBFuzz, an automated framework designed to expose Compiler-Introduced Security Bugs (CISBs) by combining semantics-aware IR mutation with coverage-guided fuzzing. The framework operates on the premise that aggressive optimizations often exploit undefined behavior semantics to eliminate seemingly redundant code, inadvertently removing essential security checks.

Figure 2 illustrates the overall workflow of CISBFuzz, which integrates the two-dimensional detection strategy. The framework first takes a valid `LLVM` IR as input and generates a diverse set of optimization-sensitive mutants through our mutation engine (Dimension 1). These mutants are subsequently processed through a coverage-guided fuzzing and differential testing loop to identify potential security discrepancies (Dimension 2).

4.1 Dimension 1: IR Mutation

Unlike traditional source-level fuzzers (*e.g.*, `Csmith`) that generate random C code, our framework operates directly on the `LLVM` IR to effectively stress-test optimization passes.

Mutation Operators. Specifically, our mutation engine selects and applies one of the following mutation operators to a seed program's `LLVM` IR to generate mutants that are syntactically valid:

(1) **SSA Value Replacement** (Figure 3a): This operator randomly replaces operands at Single Static Assignment (SSA) use sites with dominating values, fresh constants, or `undef` values.
(2) **Arithmetic Operator Replacement (AOR)** (Figure 3b): It modifies arithmetic instructions and toggles overflow flags (*e.g.*, `nsw`, `nuw`), creating corner cases for arithmetic-related passes.
(3) **Instruction Shuffling** (Figure 3c): It reorders independent instructions within a basic block, challenging the compiler's dependency analysis and instruction scheduling.
(4) **Type Manipulation** (Figure 3d): It alters integer bit-widths, such as changing `i32` to non-standard types like `i26`, forcing the compiler to handle irregular types that often trigger edge cases in backend lowering.

IR Sanity Checker. While our mutation engine handles the generation of syntactically valid IR, we implemented an additional layer to filter out semantically broken mutants, *e.g.*, the ones that crash regardless of optimization passes. Algorithm 1 details this noise filtration process. Before subjecting a mutant to the computationally expensive fuzzing loop (Dimension 2), it is compiled with optimization disabled (`-O0`) and is executed against a predefined set of heuristic inputs, including common interesting values [4].

Algorithm 1 IR Sanity Checker for Noise Filtering

Input: Mutant IR M, Compiler CC (*e.g.*, `clang`)
Output: Decision: KEEP or DISCARD

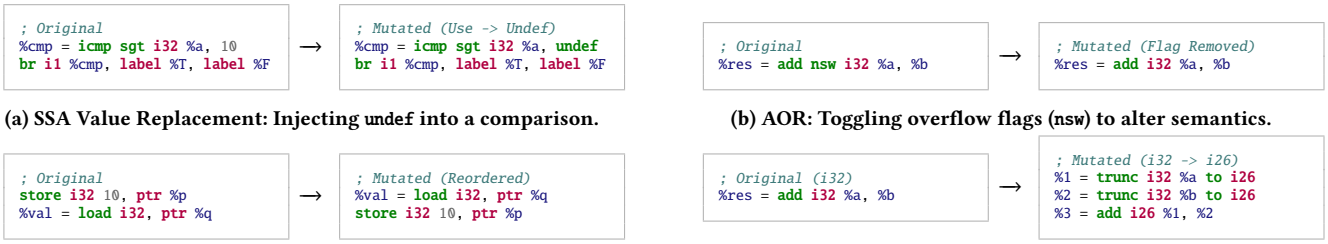
$Inputs \leftarrow \{0, 1, -1, \text{INT\_MAX}, \text{INT\_MIN}\}$
▷ Step 1: Compile with -O0 to preserve semantics
$Binary \leftarrow \text{Compile}(CC, M, \text{"-O0"})$
if $Binary$ is $\emptyset$ (Compilation Error) **then**
  return DISCARD ▷ Syntactically Invalid
end if
▷ Step 2: Execute with heuristic edge inputs
for each $input$ in $Inputs$ **do**
  $Result \leftarrow \text{Execute}(Binary, input)$
  if $Result$ is CRASH **or** TIMEOUT **then**
    return DISCARD ▷ Semantically Broken (Noise)
  end if
end for
return KEEP ▷ Valid Candidate for Fuzzing

This ensures that any crash is due to the mutation's logic itself, not an optimization bug. If the binary crashes or times out on these inputs, the mutant is classified as noise (semantically broken) and discarded immediately. By pruning these invalid mutants early, CISBFuzz focuses its fuzzing resources on the candidates that are more likely to harbor true CISBs.

4.2 Dimension 2: Input Fuzzing and Oracle

IR-level mutation alone is insufficient to expose CISBs, as optimizations often introduce *latent* security flaws that manifest only under specific runtime conditions. In particular, optimizations may preserve functional correctness for most inputs while silently miscompiling security-critical logic at boundary cases. As a result, CISBs frequently remain dormant unless the optimized binary is executed with carefully chosen inputs that activate the optimized-away checks. To systematically uncover such input-dependent CISBs, CISBFuzz incorporates coverage-guided input fuzzing as a second, orthogonal dimension. Coverage-guided fuzzing automatically generates and mutates program inputs while monitoring execution feedback (*e.g.*, edge coverage) to steer exploration toward previously unseen execution paths [4]. This enables CISBFuzz to efficiently explore corner-case behaviors that are difficult to anticipate or enumerate manually.

Coverage Tracking. Because CISBFuzz operates on `LLVM` IR, standard coverage instrumentation passes used by existing fuzzers (*e.g.*, `AFL++`) can be directly applied during compilation. This allows the fuzzer to observe fine-grained execution feedback from the optimized binaries and guide input generation toward paths affected by optimization-induced logic changes.



(a) SSA Value Replacement: Injecting `undef` into a comparison.

(b) AOR: Toggling overflow flags (`nsw`) to alter semantics.

(c) Instruction Shuffling: Reordering independent instructions.

(d) Type Manipulation: Changing types to irregular ones (`i26`).

Figure 3: Examples of the four mutation operators applied to LLVM IR snippets.

Bug Oracle. CISBFuzz employs a principled differential testing oracle that compares the runtime behavior of binaries compiled with aggressive optimizations (*e.g.*, `-O2`) against a semantically equivalent, unoptimized baseline (`-O0`). Any divergence in observable behavior, such as inconsistent outputs, or crashes in the optimized binary, indicates a potential CISB.

Execution Modes. To accommodate different types of mutated IRs, CISBFuzz operates in two execution modes:

- **Single Execution:** For non-interactive test cases that do not require an explicit input, CISBFuzz performs a single run of the optimized binary to detect immediate crashes or inconsistencies.
- **Input Fuzzing:** For programs that consume external inputs (*e.g.*, via `argv`), CISBFuzz fuzzes the optimized binary with `AFL++` to detect input-dependent errors.

4.3 Root Cause Analyzer: Faulty Pass Identification

To support actionable debugging, CISBFuzz includes an automated root cause analyzer that identifies the optimization pass (or minimal set of passes) responsible for introducing a detected CISB. Given a bug-inducing mutant IR and the corresponding differential failure, the analyzer localizes where in the optimization pipeline the security-critical discrepancy first emerges.

Method. CISBFuzz replays a compilation while selectively controlling `LLVM` optimization passes. Concretely, it compiles the bug-inducing IR under (1) single-pass configurations (*e.g.*, only `GVNPass`), and (2) incremental prefixes of the standardized pipeline (*i.e.*, enabling passes in their original order up to a cutoff). For each configuration, CISBFuzz applies the same differential oracle used for detection (output divergence, crash, or hang) and optionally inspects the generated assembly to confirm that the security check has been eliminated or corrupted.

We identify the faulty pass as the earliest optimization pass that reliably reproduces the CISB:

- **Single-pass culprit**: If enabling one pass in isolation is sufficient to reproduce the CISB, we report the pass as the culprit.
- **Interaction culprit**: If no single pass reproduces the CISB, but the CISB appears under a pipeline prefix, we report the minimal sequence of passes that triggers the discrepancy. This provides a compact starting point for debugging pass interactions, where one pass establishes a precondition and a subsequent pass materializes the vulnerability.

5 Evaluation

Setup. We implemented CISBFuzz as a modular toolkit interacting with the `LLVM 18` infrastructure. For IR mutation, we built our core mutation engine on top of `alive-mutate` [3], which is effective at generating syntactically valid `LLVM` IRs. For input space exploration, we utilized `AFL++` [4]. During the fuzzing loop, mutated IRs are compiled into executables using `afl-clang-fast`.

5.1 Overall Effectiveness: New Bugs Found

We evaluated the bug-finding effectiveness of CISBFuzz using 34 seed test cases known to contain security-sensitive code patterns (*e.g.*, integer overflow checks). Although CISBFuzz is designed to allow continuous testing, we constrained Dimension 1 (§4.1) to 10 mutants per seed in this evaluation due to processor limitations to run input fuzzing. From the 340 generated mutants, the IR Sanity Checker first filtered out 110 semantically invalid cases as noise. CISBFuzz then applied input fuzzing for one hour per mutant to the remaining 230 valid mutants in Dimension 2 (§4.2). Through this two-stage filtering and detection pipeline, CISBFuzz identified 23 CISBs that were triggerable and reproducible. Notably, while these are all silent logic errors, CISBFuzz did not encounter any crashes (*e.g.*, segmentation faults) within `LLVM`. This underscores that the systemic vulnerabilities discovered are rooted in the semantic misinterpretation of `undef` propagation rather than traditional code defects in the compiler infrastructure. We have responsibly reported all identified bugs to the `LLVM` developers. These results shows that CISBFuzz can effectively uncover unknown vulnerabilities within production-grade compilers, including `LLVM 18`.

5.2 Impact of Fuzzing

Dimension 2 uses coverage-guided fuzzing for input-dependent CISBs. Among the 23 validated CISBs, 4 cases (17.4%) were found only through fuzzing, while the remaining 19 cases were detectable via single execution (Table 1). We argue that the importance of fuzzing will scale alongside IR complexity. As mutated programs become more sophisticated, the state space governing logic corruption will expand beyond what single-execution testing can reach.

Table 1: Contribution of Execution Modes.

Method	Single Execution	Fuzzing	Total
CISBs Found	19 (82.6%)	4 (17.4%)	23 (100%)

```c
int test(int a) {
    if (a + 10 > a) { // Overflow Check
        puts("ok");
    }
    return 0;
}
```

Figure 4: Example code with an overflow check.

This will make coverage-guided fuzzing a dominant requirement for exposing deep, input-dependent vulnerabilities in future compiler generations.

Figure 4 illustrates the code that triggers one of the CISBs detected by CISBFuzz, which we refer to as the Silent Security Check Erasure (SSCE) case. In this example, the compiler corrupted a bounds check, *i.e.*, `if (a + 10 > a)`. While the unoptimized binary correctly executes `puts("ok")` for inputs like `0`, the optimized binary silently eliminates the entire `if` block and fails to execute the check for *all* inputs. This discrepancy was automatically flagged by our differential oracle, highlighting the effectiveness of fuzzing for detecting logic-level CISBs that manifest as silent semantic deviations between optimized and unoptimized binaries.

5.3 Accuracy of Root Cause Analysis

To evaluate the effectiveness of our root cause analyzer (§4.3), we examined how it attributes responsibility for detected CISBs. In several cases, the Root Cause Analyzer successfully isolated a specific culprit. For instance, as shown in Figure 5, a mutant containing an uninitialized variable (`undef`) was processed by the `EarlyCSE` pass.

In this specific case, the Root Cause Analyzer correctly identified `EarlyCSE` as the root cause, as the pass exploited the latitude provided by `undef` to "optimize away" the security-critical comparison (`icmp`). By replacing a conditional branch with an unconditional one, the optimizer effectively bypassed the intended security logic.

However, our investigation also revealed a limitation in pinpointing a single "culprit" for all CISBs. While CISBFuzz might attribute a bug to a specific pass (*e.g.*, `IPSCCP` or `EarlyCSE`), we observed that the same vulnerability could often be reproduced by substituting other passes like `InstCombine` or `GVN`. This indicates that many CISBs are not the result of a logic error in a single pass, but rather a collective consequence of `undef` propagation across the pipeline.

Furthermore, the actual materialization of the vulnerability often occurs in the `LLVM` Backend. The backend's instruction selection phase may optimize away initialization for `undef` operands, leading to operations on residual garbage values in registers. Because multiple passes allow `undef` to flow unsanitized until it reaches the backend, standard isolation techniques based on toggling individual passes are prone to misidentification. This highlights the need for more sophisticated, flow-sensitive analysis methods to distinguish between the pass that first propagates an undefined value and the one that ultimately renders it exploitable.

5.4 Case Study: Silent Security Check Erasure

To illustrate how CISBFuzz detects the complete removal of security logic, we present a case study on Silent Security Check Erasure (SSCE). SSCE represents a deterministic bypass where the compiler statically decides to eliminate a security boundary.

```llvm
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
```

(a) Mutant IR (Original)

```llvm
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
```

(b) Optimized IR (after EarlyCSE)

Figure 5: Comparison of IR before and after EarlyCSE. The optimizer identifies that the comparison depends on undefined behavior and eliminates the security-critical check entirely.

Vulnerability Pattern. The seed program contains a standard integer overflow check, `if (a + 10 > a)`, designed to ensure that the addition does not wrap around before executing critical logic (Figure 4). In a secure execution, the `puts("ok")` statement should only be reachable if the condition holds true.

Mutation. The IR mutator introduced an `undef` value into the comparison logic. Specifically, it modified the IR such that the first operand of the addition was replaced by an undefined constant, effectively transforming the check into `if (undef + 10 > a)`. While this change preserves the syntactic structure of the program, it introduces semantic ambiguity that optimization passes can exploit.

Optimization & Erasure. As shown in Figure 5, when the `EarlyCSE` pass processes this mutant, it performs constant folding and control-flow simplification. Because the result of any arithmetic operation involving `undef` is also `undef`, the optimizer treats the entire comparison (`icmp sgt`) as a "don't-care" condition.

In this instance, the optimizer chose to fold the comparison to `false`. Consequently, the conditional branch (`br i1 %cmp`) was replaced by an unconditional jump to the `if.end` label. The entire `if.then` block, containing the security-critical `puts` call, was stripped from the IR.

Security Impact. The impact of this erasure is a deterministic security bypass. Our differential oracle flagged this CISB because the unoptimized binary (`-O0`) successfully printed `"ok"`, while the optimized binary silently skipped the check regardless of the input.

6 Discussion

Reachability of Discovered CISBs. It is possible that a compiler frontend may not generate the buggy IR patterns CISBFuzz discovered. However, the primary objective of this work is to stress-test the IR-to-Backend optimization pipeline. Even if certain IR patterns are not frequently emitted by current frontends, the rapid evolution of languages (*e.g.*, Rust, Swift) and their respective frontends means that the IR-Backend boundary must remain robust against all valid IR. Our backend-centric approach proactively identifies systemic vulnerabilities that could be exploited by future frontends or specialized tools like decompilers and obfuscators.

Bug Reporting and Developer Feedback. We reported all 23 identified CISBs to the LLVM maintainers. The reports were closed with the consensus that the observed behaviors, *i.e.*, the removal of security checks in the presence of Undefined Behavior (UB), are intended optimizations according to the language specification and preventing UB is the sole responsibility of the programmer. However, we would like to argue that the compiler toolchain could play a more collaborative role in security. Considering the sheer complexity of modern software and the elusive nature of CISBs, a "secure-by-default" compiler that warns of or freezes security-sensitive `undef` values would provide a much-needed safety net for defensive programming.

7 Related Work

Compiler Testing. Seminal work by Yang *et al.* [9] (`Csmith`) demonstrated the effectiveness of fuzzing for finding compiler crash bugs. However, `Csmith` generates random C code, which often fails to stress-test specific middle-end optimization logic due to frontend normalization. Chen *et al.* [1] provide a comprehensive survey of various compiler testing techniques, categorizing them into generation-based and mutation-based approaches. While their survey highlights the breadth of the field, it underscores the relative scarcity of research specifically targeting security-critical logic bugs introduced by optimizations.

CISB Analysis. Xu *et al.* [8] formally defined "Silent Bugs" and highlighted the severity of CISBs through post-hoc analysis. While their work provided the taxonomy and patterns that guide our research, it focused on classifying existing bugs rather than providing a proactive methodology for discovery. To address the limitations of synthetic test cases, Li *et al.* [5] proposed injecting real-world code snippets into the testing pipeline. In contrast, CISBFuzz focuses on synthesizing optimization-sensitive patterns directly at the IR level to proactively uncover CISBs, exercising corner cases that are difficult to capture by simply harvesting existing code.

Source-Level Greybox Fuzzing. Recent advances have integrated greybox fuzzing into the compiler testing domain. `Grayc` [2] leverages coverage-guided fuzzing on C source code using libFuzzer to find crashes and miscompilations. While effective at exploring the frontend and early optimization passes, source-level fuzzing often struggles to bypass the syntactic barrier imposed by the compiler's frontend normalization. CISBFuzz operates directly on the IR, bypassing these restrictions to exercise deep optimization logic that is often unreachable through source-level mutations.

IR and Compiler Backend Fuzzing. Our work is closely related to `alive-mutate` [3], which mutates `LLVM` IR to detect miscompilations. However, while `alive-mutate` relies on formal verification using `Alive2` (which can be computationally expensive and bounded), CISBFuzz leverages its mutation engine for security-oriented execution fuzzing. This allows us to detect input-dependent vulnerabilities that static verification might miss. Similarly, `IRFuzzer` [7] targets LLVM backend code generation by mutating IR for diverse hardware architectures. While `IRFuzzer` primarily aims to find backend crashes and functional defects, CISBFuzz bridges the gap between IR optimization and runtime behavior by integrating `AFL++` [4]. This enables the detection of logic errors that causes security vulnerability due to specific runtime values.

8 Future Work

While CISBFuzz demonstrates the feasibility of proactively detecting CISBs, the current prototype exposes several limitations that motivate concrete directions for future work.

End-to-End Automation. Currently, CISBFuzz's the IR mutation, input fuzzing, and root cause analysis are executed as separate stages and require manual coordination. Future work will integrate these components into a fully automated pipeline that continuously generates mutants, prioritizes candidates for fuzzing, and triggers root cause analysis upon detecting a differential failure. This end-to-end automation will minimize manual effort in bug discovery and triage, enabling CISBFuzz to scale to long-running testing campaigns and larger compiler configurations.

Pass-Aware Root Cause Refinement. Our root cause analyzer localizes CISBs to the minimal pass sequence that reproduces the bug, providing a practical starting point for debugging. However, as noted §5.3, this can misattribute systemic issues involving `undef` propagation to individual passes. A concrete next step is to augment pass-level attribution with data-flow–aware tracing of `undef` values across passes, enabling CISBFuzz to distinguish between (1) passes that merely propagate `undef` and (2) passes or backend stages that unsafely materialize it. This would allow more precise localization of responsibility and reduce false attribution during bug triage.

Guided Mutation. The current IR mutation strategy relies on generic, semantics-aware operators. As CISBFuzz accumulates real CISB instances, future work can leverage these findings to design pattern-driven or feedback-guided mutation strategies that preferentially target IR constructs known to be security-sensitive (*e.g.*, branch conditions, bounds checks, and pointer validity checks). Such guidance could significantly improve fuzzing efficiency and reduce reliance on broad heuristic mutation.

Broader Bug Coverage. Currently, our findings are centered on `undef`-related issues. We aim to extend our detection capabilities to identify a broader range of CISBs stemming from localized logic errors within individual optimization passes. by refining our oracle.

Compiler-Level Mitigation. Beyond detection, a longer-term goal is to translate CISBFuzz findings into automated mitigation strategies. One concrete direction is to prototype compiler-level defenses that automatically insert guard instructions or equivalent sanitization at security-critical control-flow boundaries identified by CISBFuzz, ensuring secure optimizations.

9 Conclusion

We presented CISBFuzz, a two-dimensional framework that combines semantics-aware IR mutation with coverage-guided fuzzing to detect Compiler-Introduced Security Bugs (CISBs). Using CISBFuzz, we found 23 previously unknown CISBs in the `LLVM 18` optimizer.

Our findings show that these bugs primarily stem from aggressive exploitation of `undef` semantics, leading to both elimination of security-critical logic and unsafe materialization during backend code generation. We also observe that such vulnerabilities are difficult to localize due to cross-pass propagation and backend manifestation. Finally, the presence of input-dependent CISBs highlights the necessity of input fuzzing, as static analysis or single-execution testing alone is insufficient to expose these security violations.

References

[1] Junjie Chen, Jibesh Patra, Michael Pradel, Yingfei Xiong, Hongyu Zhang, Dan Hao, and Lu Zhang. 2020. A survey of compiler testing. *ACM Computing Surveys (CSUR)* 53, 1 (2020), 1–36.

[2] Karine Even-Mendoza, Arindam Sharma, Alastair F Donaldson, and Cristian Cadar. 2023. Grayc: Greybox fuzzing of compilers and analysers for c. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 1219–1231.

[3] Yuyou Fan and John Regehr. 2024. High-Throughput, Formal-Methods-Assisted Fuzzing for LLVM. In *2024 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 349–358.

[4] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. 2020. {AFL++}: Combining incremental steps of fuzzing research. In *14th USENIX workshop on offensive technologies (WOOT 20)*.

[5] Shaohua Li, Theodoros Theodoridis, and Zhendong Su. 2024. Boosting compiler testing by injecting real-world code. *Proceedings of the ACM on Programming Languages* 8, PLDI (2024), 223–245.

[6] Nuno P Lopes, Juneyoung Lee, Chung-Kil Hur, Zhengyang Liu, and John Regehr. 2021. Alive2: Bounded Translation Validation for LLVM. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 65–79.

[7] Yuyang Rong, Zhanghan Yu, Zhenkai Weng, Stephen Neuendorffer, and Hao Chen. 2024. IRFuzzer: Specialized fuzzing for LLVM backend code generation. *arXiv preprint arXiv:2402.05256* (2024).

[8] Jianhao Xu, Kangjie Lu, Zhengjie Du, Zhu Ding, Linke Li, Qiushi Wu, Mathias Payer, and Bing Mao. 2023. Silent bugs matter: A study of {Compiler-Introduced} security bugs. In *32nd USENIX Security Symposium (USENIX Security 23)*. 3655–3672.

[9] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and Understanding Bugs in C Compilers. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*. 283–294.