EBFuzz: Fuzzing Hidden Structural Assumptions in Compilers with Extended Inline Assembly

Jiangchang Wu
School of Computer Science
State Key Laboratory for Novel Software Technology,
Nanjing University
Nanjing, China
jiangchangwu@smail.nju.edu.cn

Yibiao Yang*
School of Computer Science
State Key Laboratory for Novel Software Technology,
Nanjing University
Nanjing, China
yangyibiao@nju.edu.cn

Abstract

Compilers are critical infrastructure whose correctness affects every program they translate. Automated compiler testing has uncovered thousands of bugs in production compilers such as GCC and LLVM. However, existing compiler fuzzers primarily explore programs with conventional control-flow and data-flow structures because they generate inputs through ordinary language constructs. Consequently, an important testing dimension remains largely unexplored: the handling of legal but structurally unusual programs. This paper presents EBFuzz, a compiler fuzzer designed to systematically explore this neglected testing dimension. Our key insight is that GNU C extended inline assembly provides a rare mechanism for influencing the control-flow and data-flow information used by compiler analyses. Because the compiler cannot fully analyze the semantics of inline assembly, it must conservatively model the effects described by programmer-provided declarations. EBFuzz exploits this capability to systematically construct legal yet structurally unusual programs through control-flow edge injection and data-flow perturbation. The resulting programs remain valid and compilable, yet are difficult or impossible for existing compiler fuzzers to generate through conventional source-level mutations. We evaluate EBFuzz on GCC and LLVM against representative compiler testing approaches spanning random generation, coverage-guided mutation, and real-world code injection. EBFuzz discovers 80% more unique bugs than the strongest baseline and is the only approach that continues finding new bugs beyond the first fifteen hours of testing. It achieves 5.9 and 5.6 percentage points higher branch coverage on GCC and LLVM, respectively, with the largest gains occurring in compiler components responsible for processing control-flow and data-flow information. Overall, EBFuzz discovered 10 previously unknown compiler bugs, 9 of which have been confirmed or fixed by developers.

CCS Concepts: • **Software and its engineering → Software testing and debugging**; *Compilers.*

Keywords: compiler testing, fuzzing, inline assembly

ACM Reference Format:
Jiangchang Wu and Yibiao Yang. 2026. EBFuzz: Fuzzing Hidden Structural Assumptions in Compilers with Extended Inline Assembly. In *The 2026 ACM SIGOPS Annual Technical Conference (SIGOPS ATC 2026), November 15–18, 2026, Hong Kong, China.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3828523.3857139

1 Introduction

Compilers are critical components of modern software infrastructure. Virtually all software is processed by a compiler before deployment. A bug in the compiler can silently introduce incorrect behavior into any program it translates, potentially affecting millions of downstream users [23, 27, 29]. Given the scale and complexity of modern production compilers such as GCC and LLVM, each comprising millions of lines of code, exhaustive manual verification is infeasible. Automated compiler testing is therefore essential [16, 29].

Over the past decade, compiler testing techniques have achieved remarkable success. Random program generators such as Csmith [29] and YARPGen [16] generate large numbers of valid programs and identify compiler bugs through differential testing, either across optimization levels or across different compilers. Equivalence-modulo-input (EMI) techniques [11] generate diverse program variants while preserving program semantics. Coverage-guided mutators such as GrayC [5] apply grammar-aware transformations to maximize compiler code coverage. Collectively, these techniques have uncovered thousands of previously unknown bugs in production compilers.

Despite their success, existing compiler fuzzers share an important limitation: they generate programs primarily through ordinary language constructs. Although these techniques explore a wide range of source-level features and execution behaviors, the resulting programs typically exhibit conventional control-flow and data-flow structures. Loops

*Corresponding author.



This work is licensed under a Creative Commons Attribution 4.0 International License.

SIGOPS ATC 2026, Hong Kong, China
© 2026 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-2818-1/2026/11
https://doi.org/10.1145/3828523.3857139

follow familiar patterns, control transfers are expressed through standard language constructs, and data dependencies remain largely consistent with programmer intent. Consequently, compiler code responsible for handling legal but structurally unusual programs receives comparatively little testing. As a result, hidden structural assumptions embedded in compiler analyses and optimizations may remain undiscovered. For example, optimization passes may implicitly assume that control-flow edges arise from ordinary language constructs or that data-flow relationships follow patterns commonly produced by source-level code.

This limitation is significant because modern compilers do not merely translate source code. Internally, they construct and manipulate representations that capture control-flow and data-flow information, which serve as the foundation for optimization and code-generation decisions. While conventional programs exercise the common cases of these analyses, compiler implementations must also correctly process unusual yet valid structures. Such assumptions are often embedded in optimization and analysis passes. Although these assumptions rarely cause problems during normal compilation, they can lead to incorrect behavior when confronted with legal programs exhibiting uncommon structures. The continued discovery of such bugs in compilers [4, 23] suggests that this testing dimension remains insufficiently explored.

Our key insight is that GNU C extended inline assembly provides a unique opportunity to explore this neglected testing space. Unlike ordinary language constructs, extended inline assembly allows source programs to influence the control-flow and data-flow information used by compiler analyses. Because the compiler cannot fully analyze the semantics of inline assembly, it must rely on programmer-provided declarations describing control-flow transfers, data dependencies, register usage, and side effects. These declarations directly affect the compiler's internal representations and optimization decisions, even though the behavior of the underlying assembly code cannot be fully verified. As a result, extended inline assembly provides a rare mechanism for creating compiler-visible structures that are difficult to obtain through conventional source-level mutations.

This paper presents EBFuzz (Edge-Breaking Fuzz), a compiler fuzzer that systematically exploits this capability to generate legal yet structurally unusual programs and expose hidden structural assumptions in compiler analyses and optimizations. EBFuzz introduces two complementary mutation families. The first, control-flow edge injection, leverages `asm goto` declarations to create unconventional control-flow structures. The second, data-flow perturbation, manipulates compiler-visible declarations of data dependencies, register usage, and side effects to alter data-flow relationships and expose assumptions in optimization and register-allocation logic. The resulting programs remain fully legal and compilable, yet they exercise compiler behaviors associated with uncommon control-flow and data-flow structures that existing fuzzers rarely reach.

We evaluate EBFuzz on GCC and LLVM and compare it against representative compiler testing approaches spanning random generation, coverage-guided mutation, and real-world code injection. EBFuzz discovers 80% more unique bugs than the strongest baseline and is the only approach that continues finding new bugs beyond the first fifteen hours of testing. It achieves 5.9 and 5.6 percentage points higher branch coverage on GCC and LLVM, respectively. Furthermore, EBFuzz discovered 10 previously unknown compiler bugs, 9 of which have been confirmed or fixed by developers.

Contributions. This paper makes three contributions:

- **Novelty:** We identify hidden structural assumptions in compiler analyses and optimizations as an under-explored testing dimension and introduce a methodology for systematically exercising these assumptions through GNU C extended inline assembly.
- **Prototype:** We design and implement EBFuzz, a compiler fuzzer that generates legal yet structurally unusual programs through two complementary mutation families: control-flow edge injection and data-flow perturbation.
- **Effectiveness:** We conduct an extensive evaluation on GCC and LLVM, demonstrating that EBFuzz substantially improves both coverage and bug-finding effectiveness. EBFuzz discovered 10 previously unknown compiler bugs, 9 of which have been confirmed or fixed by developers.

Paper Organization. The rest of this paper is structured as follows. Section 2 describes the background and our motivation. Section 3 formalizes the edge-breaking methodology, the two families of edge injection patterns, and the implementation of EBFuzz. Section 4 presents our extensive evaluation of EBFuzz. Section 5 discusses limitations and threats to validity. Section 6 discusses related work. Section 7 presents our conclusion.

2 Background and Motivation

This section first discusses the origin of hidden structural assumptions in compiler analyses and optimizations. We then explain why GNU C extended inline assembly forms a unique interface through which source programs can directly influence compiler-visible control-flow and data-flow structures. Next, we present a motivating compiler bug exposed by EBFuzz that illustrates how violating such assumptions can trigger failures in production compilers. Finally, we summarize the classes of structural assumptions targeted by EBFuzz.

2.1 Hidden Structural Assumptions in Compilers

Modern compilers do not operate directly on source code. Instead, they translate programs into internal representations that encode control-flow and data-flow information and use these representations to guide analysis, optimization, and code generation. The control-flow graph (CFG) captures

Table 1. Structural assumptions of optimization passes, organized by edge domain.

Edge type	Optimization pass	Structural assumption	ID
Control-flow (E_c)	Loop vectorization, Scalar Evolution (SCEV)	Every loop is reducible with a unique header that dominates all blocks in the loop body.	A_1
	Loop peeling, loop unrolling	Every loop has exactly one exit edge; the exit count determines epilogue placement.	A_2
	Dominator computation, shrink-wrapping	Every block (except entry) has a unique immediate dominator consistent with the CFG.	A_3
	Jump table generation, basic block reordering	Outgoing branch probabilities sum to 1 for every block.	A_6
Data-flow (E_d)	SSA construction, dead code elimination	Every use of a variable has exactly one reaching definition.	A_4
	Register allocation, live range splitting	If a variable is live at two points, it is live on every path between them; live ranges are contiguous.	A_5
	Alias analysis, store-to-load forwarding	Memory accesses via pointers of incompatible types access disjoint regions (Type-Based Alias Analysis, TBAA).	A_7

```c
char *a;
int b;

void foo (char *) {
 if (b)
  __builtin_abort();
}

void bar (char *c) {

 foo (a);

 if (c)
  bar (a);


 if (b)
  __builtin_abort();
}
```

(a) Original Program $\mathcal{P}$.

```c
char *a;
int b;

void foo (char *) {
 if (b)
  __builtin_abort();
}

void bar (char *c) {
asm goto("":::d); // add
 foo (a);
asm goto("":::d); // add
 if (c)
  bar (a);
asm goto("":::d); // add
d:                // add
 if (b)
  __builtin_abort();
}
```

(b) Mutated Program $\mathcal{P}'$.

Figure 1. Motivating example of EBFuzz. The mutated program $\mathcal{P}'$ (right) was generated by EBFuzz and triggers GCC bug #123386.

potential execution transfers among basic blocks, while data-flow information describes the definition, propagation, and consumption of values. Consequently, many compiler analyses and optimization passes rely on structural properties of these representations.

Although such properties are rarely specified explicitly, they arise naturally because most source programs exhibit well-structured control flow and consistent data dependencies. As a result, compiler passes often operate under implicit assumptions about the structures they encounter. These assumptions are typically satisfied during normal compilation. However, legal programs may still induce unusual control-flow or data-flow structures that violate such expectations. When this occurs, compiler passes may execute rarely tested code paths, potentially leading to incorrect transformations or internal compiler failures.

2.2 GNU C Extended Inline Assembly

Existing compiler fuzzers generate programs primarily through ordinary language constructs and therefore rarely produce unusual control-flow or data-flow structures. Even highly diverse random generators and mutation strategies tend to preserve conventional execution and dependency patterns.

GNU C extended inline assembly provides a rare exception. Unlike ordinary language constructs, it allows programmers to explicitly declare control-flow transfers, data dependencies, register usage, and side effects that are incorporated into the compiler's internal analyses. Because the compiler cannot fully reason about the semantics of arbitrary assembly code, it must conservatively model the effects described by these declarations when constructing and optimizing its internal representations.

Consequently, extended inline assembly offers a unique mechanism for influencing the control-flow and data-flow information observed by compiler analyses. It enables the construction of compiler-visible structures that are difficult or impossible to express through conventional source-level code, making it particularly well suited for exploring rarely exercised compiler behaviors.

GNU C extended inline assembly exposes several constructs through which programmers can communicate such information to the compiler. The general syntax of an extended assembly block is shown below:

```c
__asm__ [volatile] [goto] (
    "assembly template"
    : [output]   /*"=r"(var)*/
    : [input]    /*"r"(var) */
    : [clobbers] /*"memory" */
    : [labels]   /* label   */
```

```
7 );
```

Among the available constructs, EBFuzz exploits three that directly affect the compiler's control-flow and data-flow models [8]:

- *Goto labels*: `asm goto("":::L);`
  An `asm goto` statement declares a potential control-flow transfer from the assembly block to label `L`. The compiler must conservatively incorporate the corresponding edge into the CFG, even though it cannot determine whether the assembly code will actually transfer control to the specified target. As a result, `asm goto` can introduce CFG edges that do not naturally arise from ordinary language constructs.
- *Operand constraints*: `int x; asm("": "+r"(x));`
  Operand constraints specify how variables participate in an assembly block. In this example, `+r` declares that `x` is both read and written by the assembly code. Although the compiler cannot inspect the assembly body itself, it must incorporate these declarations into its def-use chains, data-flow analyses, and register-allocation decisions. Operand constraints therefore provide a mechanism for directly influencing compiler-visible data-flow relationships.
- *Clobber lists*: `asm("":::"memory");`
  Clobber lists declare side effects that are not captured by explicit operands. In this example, the `"memory"` clobber informs the compiler that the assembly code may modify arbitrary memory-visible state. The compiler must therefore conservatively invalidate certain assumptions about memory contents and account for the declared side effects during optimization. Clobber specifications thus provide an additional mechanism for perturbing compiler-visible data-flow information.

Collectively, these constructs establish a trusted interface through which source programs can influence the control-flow and data-flow information used by compiler analyses. EBFuzz systematically exploits this interface to generate legal programs that challenge hidden structural assumptions in compiler optimizations.

2.3 Motivating Example

Consider the program in Figure 1a. It is short, well-structured standard C, and every modern compiler compiles it without error at every optimization level. Figure 1b presents a real compiler bug triggered by a program generated by EBFuzz. The injected `asm goto` statements are individually valid and the resulting program compiles successfully under `-O0`. However, GCC crashes with an internal compiler error at higher optimization levels.

The crash is caused by the injected declarations, which create an unusual control-flow structure that does not arise from ordinary C constructs. In particular, one injected edge shares the same source and destination as an existing fallthrough edge.

Although the resulting program is legal, this topology violates an implicit structural assumption made by GCC's optimization pipeline. When the affected pass processes the CFG, it encounters a case that was not anticipated by the implementation, ultimately triggering a compiler crash.

This example illustrates the key observation behind EBFuzz: extended inline assembly can create legal compiler-visible structures that violate assumptions rarely exercised by existing compiler fuzzers.

2.4 Classes of Hidden Structural Assumptions

The motivating example is not an isolated bug. Our analysis of optimization passes and historical compiler failures suggests that similar issues often stem from recurring classes of hidden structural assumptions. The resulting classes are not intended to form a complete taxonomy. Rather, they capture representative structural properties relied upon by a broad range of compiler analyses and optimizations. Each class was chosen because it maps to a mechanism expressible in GNU C extended inline assembly and to a pass that an injected asm declaration can target.

Table 1 organizes the seven assumption classes by the domain they govern and assigns each a label (A_1 through A_7) for cross-reference. The four control-flow assumptions constrain the structure of the CFG. Every loop must be reducible with a unique header that dominates all blocks in the body (A_1) and must have exactly one exit (A_2). Every non-entry block must have a unique immediate dominator (A_3). Outgoing branch probabilities from any block must sum to one (A_6). The three data-flow assumptions constrain the relationships among values. Every variable use must have exactly one reaching definition, forming complete def-use chains (A_4). Live ranges must be contiguous: if a variable is live at two program points, it is live on every path between them (A_5). Memory accesses via pointers of incompatible types must access disjoint regions, per type-based alias analysis (A_7). Each assumption is a contract: the pass is written under the expectation that the input IR satisfies the stated property, and when that property is violated the pass can produce incorrect code or crash [23].

Existing compiler fuzzers overwhelmingly satisfy these assumptions by construction and therefore rarely exercise the code responsible for handling assumption violations. The goal of EBFuzz is to systematically generate legal programs that challenge these assumptions through extended inline assembly, thereby exploring a testing space that remains largely unexplored by existing compiler fuzzers.

3 Methodology

This section formalizes EBFuzz's edge-breaking methodology and describes its implementation. We first define the compiler's internal program representation and the structural assumptions that optimization passes make about edges.

We then introduce the two families of edge injection patterns that EBFuzz uses to violate those assumptions. Finally, we describe the pipeline that realizes these patterns as concrete asm injections.

3.1 Problem Formulation

3.1.1 Program Representation. Let P be a C program and f a function within P. The compiler translates f into an internal representation that can be abstracted as a graph $G_f = (B, E_c, E_d, V)$, where:

- $B = \{b_1, b_2, \ldots, b_n\}$ is the set of basic blocks.
- $E_c \subseteq B \times B \times [0, 1]$ is the set of control-flow edges, where each edge (b_i, b_j, p) represents a potential execution transfer with branch probability p.
- $E_d \subseteq V \times B \times B$ is the set of data-flow edges, where each edge (v, b_{def}, b_{use}) represents a reaching definition of variable v from b_{def} to a use in b_{use}.
- V is the set of variables visible in f.

The control-flow and data-flow subgraphs are coupled: a data-flow edge (v, b_{def}, b_{use}) exists only if a control-flow path connects b_{def} to b_{use} without redefining v.

3.1.2 Edge Assumptions of Optimization Passes. Each optimization pass π operates on G_f under implicit structural assumptions about its edges. When an assumption is violated, the pass may produce incorrect IR or abort with an internal compiler error. We identify seven classes of such assumptions below:

1. **Reducibility (A_1).** For any loop $L \subseteq B$, the subgraph induced by L is reducible: a unique header block $h(L) \in L$ dominates all blocks in L, and all back-edges target $h(L)$. Loop optimization, SCEV analysis, and loop vectorization assume reducibility.
2. **Single-Exit Loop (A_2).** For any loop L, $|\{(b_i, b_j) \in E_c \mid b_i \in L \wedge b_j \notin L\}| = 1$. Loop vectorization, loop peeling, and loop unrolling require a unique exit to compute iteration counts.
3. **Dominator Completeness (A_3).** Every block $b \in B$ (except entry) has a unique immediate dominator $idom(b)$. SSA construction, code motion, and shrink-wrapping rely on dominator consistency.
4. **Def-Use Chain Completeness (A_4).** Every use of v has exactly one reaching definition. SSA construction, dead code elimination, and register allocation assume complete, acyclic def-use chains.
5. **Live Range Continuity (A_5).** If v is live at b_1 and b_n, it is live on every path between them that does not redefine v. Register allocation assumes contiguous live ranges.
6. **Branch Probability Normalization (A_6).** For any block b, $\sum_{(b,b',p) \in E_c} p = 1$. Jump table generation and block reordering assume normalized probabilities.
7. **Type-Based Alias Disjointness (A_7).** Memory accesses via pointers of incompatible types access disjoint regions. Alias analysis and store-to-load forwarding rely on TBAA.

Programs produced by random generators (Csmith, YARPGen) and by equivalence-preserving transformations (EMI) satisfy all seven assumptions by construction. Compiler passes are thoroughly tested on such well-structured programs, but the error-handling paths for assumption violations are rarely exercised.

3.1.3 Edge Injection via Extended Asm. The goal of EBFuzz is to systematically violate the seven assumptions (A_1–A_7) by injecting asm declarations that the compiler must model but cannot verify. We model an asm injection as a transformation on G_f. A CFG edge injection τ_c inserts a new control-flow edge:

$$\tau_c(G_f, b_i, \ell_j) \mapsto G'_f = (B, E_c \cup \{(b_i, b_j, p_{asm})\}, E_d, V) \quad (1)$$

where b_i contains the injected `asm goto` and b_j contains the target label ℓ_j. A data-flow edge injection τ_d inserts or modifies data-flow edges:

$$\tau_d(G_f, v, b_i, op) \mapsto G'_f = (B, E_c, E_d \cup \{(v, b_i, \cdot)\}, V) \quad (2)$$

where $op \in \{\text{output}, \text{input-output}\}$ declares a new definition or anchors an existing one.

The central problem: given a well-structured seed P_0 with graph G_f satisfying all seven assumptions A_1–A_7, find a sequence of injections $\tau_1, \ldots, \tau_k \in \mathcal{T}_c \cup \mathcal{T}_d$ such that the transformed graph violates at least one assumption, causing the compiler to crash or miscompile. To address this problem, we design seven injection patterns: three targeting control-flow assumptions (τ_c) and four targeting data-flow assumptions (τ_d), each instantiable at multiple locations within a seed program.

3.2 Injection Patterns

EBFuzz implements three τ_c patterns that perturb E_c and four τ_d patterns that perturb E_d. The two families differ in their primary mechanism: a τ_c injection is designed to add unusual control-flow edges, and a τ_d injection is designed to add inconsistent data-flow edges.

3.2.1 Control-Flow Edge Breaking (τ_c). Irreducible loop construction (targeting A_1, A_3). A label is placed immediately before a loop header, and an `asm goto` inside the loop body targets that label while a natural C `goto` jumps backward to the same label. This creates a back-edge to a non-header block and yields an irreducible CFG that breaks dominator consistency. SCEV, loop vectorization, and related analyses that assume header domination can abort or produce incorrect IR when they encounter this topology.

```
1 eb_t1:          /* label before the for loop */
2 for (...) {
3   __asm__ volatile goto("": : : "memory": eb_t1)
        ;
4   goto eb_t1;  /* backward jump to the pre-
        header */
```

```
5    ...}
```

Multi-exit loop forgery (targeting A_2). An `asm goto` is inserted inside a loop body; its target label is placed outside the enclosing loop nest. The injection produces a second exit edge for the loop, violating the single-exit assumption. Loop vectorization and loop peeling, which require a unique exit to compute trip counts and epilogues, can fail or compute incorrect epilogue code under this condition.

```
1 for (...) {
2   __asm__ volatile goto("": : : "memory":eb_t2);
3   ...}
4 eb_t2:              /* outside the loop */
```

Unlike the irreducible loop pattern, where the label sits before the loop header and a C `goto` creates the back edge, this pattern places the label after the loop and does not require a C `goto`. The two patterns differ in label placement and back-edge construction.

Heterogeneous edge superposition (targeting A_3, A_6). An `asm goto` is added to target a label that the same basic block already reaches, either through fallthrough or through a natural C `goto`. The block therefore ends up with two outgoing edges that share the same source and target, so the successor is no longer reached through a single distinguishable edge. The compiler also assigns independent branch probabilities to each outgoing edge, allowing their probabilities to sum to more than one and violate normalization. Edge splitting for shrink wrapping, jump table generation, and branch-probability verification can fail assertions or trigger internal errors on these structures.

```
1 __asm__ volatile goto("": : : "memory": eb_t1);
2 // ... natural C code ...
3 goto eb_t1;
```

3.2.2 Data-Flow Edge Perturbation (τ_d). Phantom definition injection

(targeting A_4). An output constraint such as `"=r"(v)` is attached to an `asm goto` to declare a new definition at the asm site. The asm-declared definition can reach downstream uses alongside the original definition, producing multiple reaching definitions for the same use and violating def-use completeness. The compiler must treat the declared output as a genuine reaching definition and carry it into SSA construction and the def-use chains. SSA construction and dead code elimination passes that assume a single reaching definition per use can abort or produce incorrect transformations.

```
1 __asm__ volatile goto("": "=r"(var): : : label);
```

Dead variable resurrection (targeting A_5). An input-output constraint such as `"+r"(v)` is applied to a value that the compiler has constant-folded, followed by an immediate clobber that forces a spill and reload. This sequence materializes a short-lived live range that is born and killed at the same point, fragmenting live-range continuity. The compiler must treat the value as live at the asm site and allocate a register for it, even though constant propagation has already removed the source-level live range. Register allocation and live-range splitting can fail or insert incorrect spill and reload sequences when live ranges are non-contiguous.

```
1 ({ __asm__ volatile ("": "+r" (fold_const));
2    __asm__ volatile goto (""
3      : : :"x0", "x3", "memory": eb_t1);});
```

Shared dependency forgery (targeting A_5). A matching constraint such as `"0"` is used to force two operands to share the same physical register where their live ranges partially overlap. The forced aliasing conflicts with the interference graph expectation that simultaneously live values occupy distinct registers. The compiler must place both operands in the same register at the asm site, inserting a copy when their live ranges cannot be coalesced. The register allocator can be driven into infeasible assignments or incorrect spilling decisions by such constraints.

```
1 __asm__("":"=r"(tmp):"0"((unsigned long)(src)));
```

Type-based alias erasure (targeting A_7). An asm matching constraint is used to copy a pointer value across the asm boundary so the compiler must trust value equality but cannot propagate the source pointer's type provenance. The loss of type-derived disambiguation forces alias analysis to degrade to conservative may-alias semantics. Store-to-load forwarding and scheduling passes that rely on TBAA can then behave conservatively or encounter unforeseen aliasing logic.

```
1 __asm__ volatile("": "=r"(tmp): "0"(ptr));
```

3.3 EBFuzz Pipeline

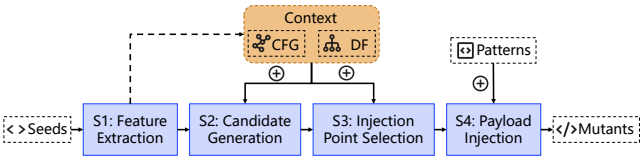


Figure 2. EBFuzz Pipeline.

EBFuzz is organized as a four-stage pipeline (Figure 2) that transforms a seed C program into a mutated variant ready for compiler testing.

Feature extraction performs libclang AST analysis and LLVM IR emission with debug metadata on the seed program. AST analysis identifies anchor points where asm injections are syntactically legal: loop preheaders, variable declarations, return paths, call expressions, cleanup boundaries, conditional branches, and compound statements. IR analysis classifies each variable's liveness state: register-bound (active in SSA registers), constant-folded (literal values tracked via debug metadata), or memory-resident (stack-allocated).

Candidate generation collects injection targets from all enabled strategies. Each of the seven τ patterns is implemented as an independent strategy class specifying a candidate predicate, target anchor types, and an asm template.

Algorithm 1 Injection point selection and application

Require: Seed source P, context ctx, strategies $\mathcal{S}$, budget K, mutation probability p_{mut}
Ensure: Mutated source P'
1: $pts \leftarrow \bigcup_{S \in \mathcal{S}} S$.candidate_lines($ctx$)
2: Partition pts by enclosing function → map $M : f \mapsto$ lines
3: **for** each function $f \in M$ **do**
4: $quota(f) \leftarrow \lfloor K/|M| \rfloor$
5: **end for**
6: Distribute remaining $K \bmod |M|$ to functions with largest $|M(f)|$
7: **for** each function $f \in M$ **do**
8: Shuffle $M(f)$; $selected(f) \leftarrow \{\}$
9: **for all** line $\ell \in M(f)$ **do**
10: **if** random() $< p_{mut}$ **and** $|selected(f)| < quota(f)$ **then**
11: $selected(f)$.add(ℓ)
12: **end if**
13: **end for**
14: **end for**
15: Sort all selected ℓ by line number descending
16: **for all** $\ell \in$ sorted_selected (bottom to top) **do**
17: $P \leftarrow S_\ell$.generate(P, ℓ, ctx)
18: **end for**
19: **return** P

Register clobber lists and constraint names are instantiated using architecture-specific pools covering x86-64, AArch64, and RISC-V, so templates adapt to the target architecture at injection time.

Injection point selection balances mutation pressure across the program to prevent large functions from monopolizing the global injection budget. EBFuzz partitions all candidate lines by their enclosing function and calculates an equal share of the budget for each function. It then distributes any modulo remainder to the functions with the largest candidate sets and applies randomized sampling within each partition to ensure diverse anchor selection across consecutive runs. Algorithm 1 formalizes this allocation and processing procedure.

The algorithm first aggregates syntactically legal candidates from all enabled strategies (line 1). It builds a partition map to compute a strict injection quota for every function (lines 2–4). Within each partition, the algorithm shuffles the candidates and applies a mutation probability to select targets until it hits the specific function quota (lines 7–11). The selection process guarantees that the fuzzer perturbs control-flow and data-flow constraints uniformly across the entire seed program rather than clustering injections locally. Finally, the algorithm sorts all selected points in descending line-number order for application (lines 15–17). Because each asm injection inserts new text into the source file, applying mutations strictly from bottom to top guarantees that the line indices of pending upper injection points remain exactly aligned with the original context.

Payload injection instantiates the strategy-specific asm template at each selected point. Label placement is guarded to keep the mutated source syntactically valid: labels are not placed between an `if` and its `else`, on function declaration lines, or inside C99 for-loop declarations where they would split a variable's scope. Adding a new strategy requires only a candidate predicate and an asm template; the dispatch loop is strategy-agnostic.

The mutated source is compiled against GCC and LLVM across multiple optimization levels. Once a variant triggers a bug, C-Reduce minimizes the program before reporting.

4 Evaluation

This section evaluates EBFuzz against trunk builds of GCC and LLVM. Section 4.1 reports the bugs discovered by EBFuzz and characterizes them by compiler component. Sections 4.2 to 4.4 then address three controlled research questions:

1. **RQ1.** How does EBFuzz compare to existing fuzzing tools in bug discovery?
2. **RQ2.** Does EBFuzz achieve higher branch coverage in compiler passes than baseline testing techniques?
3. **RQ3.** How does the execution throughput of EBFuzz compare to existing generators?

Hardware. Our evaluation was conducted on a workstation with an Intel 48-core 2.30 GHz CPU, 120 GiB RAM, running Ubuntu 22.04.2 LTS. The compilers under test were trunk builds of GCC and LLVM (Clang).

Testing process. Our testing process is fully automated and runs continuously. EBFuzz first extracts structural anchors and liveness information from seed programs to build the injection context, then dispatches strategy-specific asm templates to produce program variants. Each variant is compiled against GCC and LLVM across multiple optimization levels. The AArch64 and RISC-V cross-compilers are also exercised, and the generated binaries are executed under QEMU emulation. Once a variant triggers a compiler bug, we use C-Reduce [19] to minimize the program. If the minimized program triggers a previously unknown compiler bug, we report it to the developers. The only manual step in the testing workflow is analyzing the reduced test cases and filing bug reports.

4.1 Bug Finding

EBFuzz reported 12 compiler bugs across GCC and LLVM (Clang) over the entire testing campaign, of which 10 were previously unknown and unique. Two reports (175314 and 123394) were marked as duplicates by developers. Table 2 lists each bug with its compiler, the affected pass, the injection pattern that triggered it, and its current resolution status. Among the 10 unique bugs, 6 have been fixed or have

Table 2. Compiler bugs discovered by EBFuzz.

ID	Compiler	Pass	Pattern	Family	Status
150384	Clang	SelectionDAG	Dead var resurrection	τ_d	Confirmed
175314	Clang	Clang Sema	Irreducible loop	τ_c	Duplicate
175516	Clang	LTO	Heterogeneous edge	τ_c	Confirmed
175540	Clang	IR verifier	Irreducible loop	τ_c	Fixed
179238	Clang	Clang Sema	Irreducible loop	τ_c	Confirmed
200987	Clang	ScalarEvolution	Phantom definition	τ_d	Fixed
201252	Clang	CodeGenPrepare	Phantom definition	τ_d	Fix submitted
123386	GCC	Shrink Wrapping	Heterogeneous edge	τ_c	Fixed
123394	GCC	CFG expansion	Irreducible loop	τ_c	Duplicate
125438	GCC	DWARF Frame Generation	Irreducible loop	τ_c	Reported
125477	GCC	Tree Vectorization	Multi-exit loop	τ_c	Fixed
125545	GCC	Tree SSA Reassociation	Phantom definition	τ_d	Fixed

developer-submitted patches, 3 have been confirmed by developers, and 1 remains unconfirmed. The bug marked "Fix submitted" in Table 2 (201252) is an LLVM bug for which a developer has submitted a pull request on GitHub with the fix under review at the time of writing.

Responsibility for a failure lies with the pass that mishandles the edge, not with an earlier pass for permitting the structure to persist. We do not attribute a failure to a pass merely because it crashed there: asm-declared edges are legal input, every pass must handle legal IR, and abnormal `asm goto` edges must survive the pipeline because they carry runtime semantics that canonicalization cannot remove.

The bugs span 6 distinct passes in Clang and 4 in GCC, covering front-end, middle-end, and back-end stages. The τ_c patterns (irreducible loop, multi-exit loop, heterogeneous edge) account for 6 of the 10 unique bugs, while the τ_d patterns (phantom definition, dead var resurrection) account for 4. For five of the ten, developer triage identified the faulty pass. For the other five we report the pass where the failure surfaces.

The irreducible loop pattern accounts for 3 bugs (175540, 179238, 125438), all of which involve `asm goto` edges whose target is not the loop header. The failure surfaces in the IR verifier for 175540, in Clang's Sema for 179238, and in DWARF frame generation for 125438. Bug 125438 is also reachable through data-flow perturbation. Its minimized test case contains only an `asm goto` into a loop body, and a data-flow constraint that alters the register pressure enough to change the shrink-wrapping decisions on the same CFG structure triggers the same failure.

The multi-exit loop pattern accounts for bug 125477, where the second exit edge reaches the GCC loop vectorizer. The maintainer located the fault in loop versioning, and the diagnosis is quoted below.

The phantom definition pattern accounts for 3 bugs (200987, 201252, 125545) through `"=r"` output constraints that create spurious definition sites. Two of the three abort in passes that use dominator information to decide where to insert code, because the extra definition site makes the insertion point they compute fail to dominate the use it was chosen for. These are scalar evolution in LLVM (200987) and reassociation in GCC (125545). In 201252 the code generation pass that merges address computations inspects a basic block whose terminator is not yet well formed, a state that the injected edge can produce, and asserts on it.

The heterogeneous edge pattern accounts for 2 bugs (175516, 123386) via degenerate `asm goto` edges whose source and destination coincide with an existing fallthrough edge. For 123386, GCC maintainers traced the failure to the pass that splits fallthrough edges for shrink wrapping. Its splitting logic handles the degenerate `asm goto` incompletely and leaves the call frame information inconsistent for the DWARF pass that runs later.

Bug 150384 is the sole instance of the dead var resurrection pattern, where an `"+r"` constraint on a vector-typed variable forces SelectionDAG to fault.

Several of these bugs existed in compiler versions across multiple release cycles before being triggered by EBFuzz. Bug 125477 was a regression introduced in GCC 10 and resided in all subsequent releases through GCC 16. Bug 125545 existed in GCC 14 through 16. Bug 150384 reproduces in Clang 17 through 20 and remains unfixed on trunk. These multi-release latencies indicate that the corresponding code paths contain edge conditions reachable only under the irregular structures produced by extended asm injection.

Developer responses to several of these bug reports directly corroborate the edge-breaking framework of Section 3. The GCC maintainer diagnosis report of bug 125477[1] is representative:

> *"The issue is that we try to version an outer loop but loop_version fails as we cannot copy it as there's the abnormal dispatcher inside it."*

[1] https://gcc.gnu.org/bugzilla/show_bug.cgi?id=125477

"Abnormal dispatcher" is compiler-internal terminology for a control-flow edge whose source is not the loop header, precisely the structure that the multi-exit loop pattern (τ_c, targeting A_2) constructs. This confirms that the bug is not a random memory error but a direct consequence of the structural edge violation that EBFuzz systematically injects.

4.2 RQ1: Baseline Comparison

To quantify EBFuzz's bug-finding effectiveness relative to existing tools, we ran a controlled 24-hour comparison [1]. We selected four baselines that represent the three dominant paradigms of automated compiler testing. Csmith [29] and YARPGen [16] are random program generators: Csmith is the most widely deployed such tool, having reported over 400 bugs in GCC and LLVM, while YARPGen extends the random generation approach with a more expressive type system and generation grammar. GrayC [5] represents coverage-guided mutation: it applies grammar-aware transformations to seed programs with the explicit goal of maximizing compiler code coverage. Creal [14] represents real-world code injection: it extracts functions from open-source C projects and grafts them into seed programs to expose the compiler to code patterns drawn from production software. Together, these four tools span the space of existing compiler testing techniques against which EBFuzz's edge-breaking approach should be evaluated. Each tool was run for 24 hours using seeds generated by Csmith and YARPGen. Csmith and YARPGen used their default differential testing modes. GrayC [5] and Creal [14] were configured following the procedures described in their respective papers, with identical seed corpora where applicable. All campaigns used the same hardware and compiler builds. We use stable releases rather than trunk builds here because the bugs that Creal and GrayC target have been actively patched; running these tools against trunk would substantially undercount their bug-finding capability relative to their published results. We select GCC 13 and Clang 17, matching the versions used in the Creal evaluation [14]. The campaigns were not run concurrently to avoid resource contention. The EBFuzz run used here is a fresh 24-hour campaign distinct from the extended testing reported in Section 4.1.

We applied a uniform deduplication procedure across all tools to ensure a fair comparison. We deduplicate compiler crashes by matching the top three stack frames and the assertion condition string. We deduplicate miscompilation bugs by performing git bisect on each minimized test case to identify the commit that introduced the incorrect behavior, treating cases that converge to the same commit as the same bug. All bug counts reported in this section are unique counts after this deduplication procedure.

Table 3 reports the unique bug counts. EBFuzz found 9 unique bugs across both compilers, outperforming Creal (5), GrayC (3), and YARPGen (1). Csmith found no new bugs during this period. EBFuzz detected more bugs in GCC (6) than in LLVM (3), demonstrating an asymmetry consistent with findings in prior work [5, 23].

Table 3. The number of detected bugs in a 24-hour campaign.

Tool	GCC	LLVM
EBFuzz	6	3
Creal	3	2
GrayC	2	1
YARPGen	1	0
Csmith	0	0

Figure 3 shows the overlap in unique bugs discovered by EBFuzz, Creal, and GrayC. One bug was found by all three tools, indicating a shared reachable bug that each tool triggers through a different mechanism. EBFuzz and Creal share one additional bug, while Creal and GrayC share none in this 24-hour window. EBFuzz's 7 exclusive bugs, against Creal's 3 and GrayC's 2, confirm that the edge-injection attack surface is largely orthogonal to the coverage-guided and real-world-code-based approaches of the prior tools. YARPGen's sole discovery is exclusive to YARPGen.

Figure 4 plots cumulative unique bugs over the 24-hour window. Markers indicate the precise time at which each new unique bug was first observed. EBFuzz exhibits two distinct phases: a rapid initial phase (hours 0 to 6, 5 bugs found) followed by a sustained phase (hours 6 to 24, 4 additional bugs) in which bugs are discovered at a lower but persistent rate. GrayC plateaus after roughly six hours, and Creal after roughly fifteen; neither approaches EBFuzz's sustained discovery rate. EBFuzz's seven injection patterns provide orthogonal attack vectors that sustain discovery beyond this point.

The single YARPGen bug was found at hour 3 and no further discoveries followed. Csmith produced a flat line at zero throughout the 24-hour window, consistent with the saturation effect documented in the GrayC evaluation [5]: Csmith and YARPGen, having been deployed against GCC and LLVM for years, rarely trigger new bugs on contemporary compiler versions.

4.3 RQ2: Code Coverage Analysis

We measure compiler source code coverage to determine whether EBFuzz exercises code paths that existing tools leave untouched. GCC was built with `-coverage` (gcov instrumentation) and LLVM with source-based coverage (`llvm-cov`). Each tool was run in ten independent 24-hour trials on the same machine, following the evaluation practice recommended by Klees et al. and Schloegel et al. [9, 21], and the corpora accumulated across the trials were replayed against the instrumented builds to collect line and branch coverage.

Figure 5 reports branch coverage for all three tools across 8 GCC modules and 7 LLVM modules, averaged over ten 24-hour trials per tool. Error bars denote ± one standard

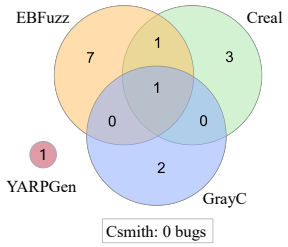


Figure 3. Unique bugs found by different techniques.

deviation across the ten trials. For the six overall tool and compiler combinations, the standard deviation over the ten trials is between 0.35 and 0.45 percentage points, and the range between the lowest and highest trial is between 1.1 and 1.4 percentage points. At the overall compiler level, EBFuzz achieves 58.2% branch coverage on GCC and 59.3% on LLVM, compared to 52.3% and 53.7% for Csmith and 49.2% and 50.3% for YARPGen; these numbers are computed over the entire instrumented compiler rather than the modules shown in Figure 5. The aggregate improvement is modest (5.9 and 5.6 percentage points over Csmith) because a large fraction of compiler code (front-end parsing, IR construction, common utilities) is well exercised by all three tools. Per-trial coverage differs significantly between EBFuzz and each baseline (two-sided exact Mann-Whitney U test, ten trials per tool, $p < 10^{-4}$ in all four comparisons).

The per-module improvements over Csmith are concentrated in precisely the compiler components that implement the edge assumptions identified in Section 2. The largest gains on GCC appear in Shrink Wrapping (34.9% to 49.0%), DWARF Frame Generation (27.8% to 40.8%), Local Register Allocator (LRA) Constraints (38.9% to 51.3%), and CFG Manipulation (42.0% to 53.2%). On LLVM, the largest gains appear in CFI Instruction Insertion (31.1% to 43.5%), Shrink Wrapping (38.0% to 49.6%), and Greedy Register Allocation (45.8% to 54.7%). These modules are responsible for processing CFG edges, register allocation under constraints, and debug frame generation, all of which are directly exercised by EBFuzz's asm injection patterns. Csmith and YARPGen, which generate only well-structured programs, cannot produce the irregular edge topologies or unusual register constraints needed to reach the untested branches in these modules.

The gap narrows in passes where Csmith is already strong. Loop Optimization (56.6% vs. 57.8%) and Instruction Combine (61.2% vs. 62.5%) differ by less than two percentage points; Csmith's structured loops and arithmetic expressions already cover these passes. Register Copy Propagation (55.3% vs. 57.3%) and Scalar Evolution (57.1% vs. 58.5%) show similar narrow gaps. The coverage advantage is largest in edge-processing code reached only by EBFuzz and negligible in general-purpose optimization code that existing tools already cover.

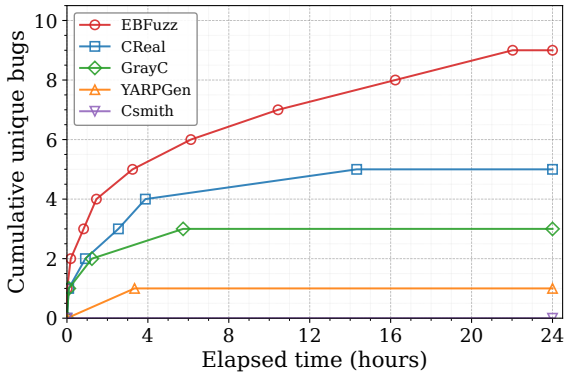


Figure 4. Cumulative unique bugs discovered over the 24-hour campaign.

4.4 RQ3: Efficiency

Table 4 reports efficiency metrics for all five tools collected from the same 24-hour runs used in the baseline comparison.

EBFuzz processes 2,012 variants per hour, higher than Csmith (1,146/h) and YARPGen (1,470/h) but below GrayC (2,897/h). The throughput is constrained by three factors. First, the feature extraction stage runs a libclang AST parse and an LLVM IR emission pass on every seed to extract structural anchors and liveness classifications (Section 3), adding approximately 0.8 seconds of preprocessing per seed. Second, the injection strategies produce programs with inline assembly, which forces the compiler to invoke the assembler and register allocator on every compilation. Third, the deduplication step requires compiling each crashing variant at multiple optimization levels to construct a robust crash signature, multiplying the effective compilation budget per variant.

Despite the lower throughput, EBFuzz found more unique bugs than GrayC (9 vs. 3). In this experiment, EBFuzz's higher bug yield despite lower throughput suggests that the specificity of the injection strategy contributed more to bug discovery than the raw throughput.

EBFuzz achieves an 87.3% compile success rate, lower than the near-perfect rates of Csmith and YARPGen (both 100%) and GrayC (99.4%). The 12.7% failure rate has two main causes. First, some injection combinations produce unsatisfiable register constraints: for instance, requesting a register operand in a machine mode that the target architecture does not support for that constraint character. Second, multi-strategy injection can occasionally produce asm templates whose combined clobber sets cover all available registers, leaving no register for the compiler to allocate to the operand constraints. Both failure modes are detected at compile time and produce a clean error diagnosis, so they do not inflate the crash count or compromise the oracle. Failed variants are cheap: a compilation attempt that fails with a syntax or constraint error typically terminates in under 0.3 seconds.

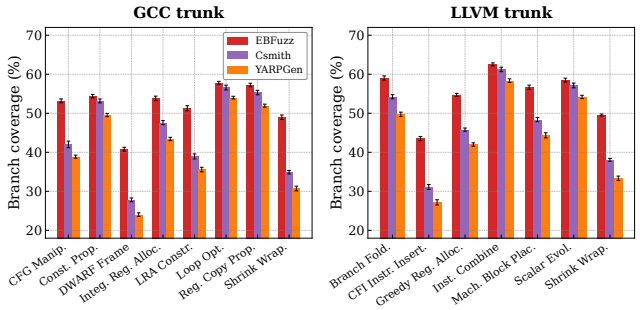


Figure 5. Branch coverage by module for EBFuzz, Csmith, and YARPGen on GCC and LLVM. Error bars denote one standard deviation across the ten trials.

EBFuzz achieves higher throughput than Csmith and YARPGen while demonstrating substantially higher bug-finding effectiveness. The 12.7% compilation failure rate is an acceptable cost: failed variants are cheap to detect, and the edge-injection strategy that causes the failures is also what enables EBFuzz to reach compiler code paths that the baselines do not reach.

5 Discussion

5.1 Scope of Bug Detection

All bugs discovered by EBFuzz are internal compiler errors (ICEs). This reflects a property of the current injection strategies: the asm declarations produce facts about control-flow and data-flow that violate the algorithm preconditions of targeted optimization passes, causing the compiler to abort before producing output. The seeds used in our evaluation, generated by Csmith and YARPGen, produce deterministic programs whose execution does not depend on runtime input. Richer seed programs augmented with runtime guards that selectively disable injected paths would enable EBFuzz to detect miscompilations through differential testing.

Fixing these bugs does not reduce the performance of the code that GCC and LLVM generate for ordinary programs. The fixes modify optimization logic rather than code generation, and they take the form of guards and bail-outs that are evaluated only on the exceptional paths that the injected structures reach. The committed fix for bug 125477 is representative: a loop-copy check before versioning, applied only to outer loops that contain an abnormal dispatcher. Programs that match this pattern may lose an optimization they would otherwise receive. We accept this trade-off because correctness takes priority.

5.2 Inline Assembly in Practice

The exact synthetic topologies EBFuzz generates are rare in hand-written code. The constructs they are built from are not, and compilers must handle the legal IR derived from them. Inline assembly implements context switches, atomics and syscalls in the Linux kernel and glibc. The `asm goto` form implements static keys (`jump_label.h`)[2], which the kernel uses for rarely toggled branches in hot paths, and tracepoints, which appear inside loops. The `static_cpu_has` macro (`cpufeature.h`)[3] implements runtime CPU feature checks with a single asm block that declares two `asm goto` edges, which is one of the shapes EBFuzz produces. Ordinary compiler tests do not exercise these paths, and mishandling them breaks real infrastructure code.

Table 4. Efficiency metrics for the 24-hour runs.

Tool	Variants/h	Compile Success	Bugs
EBFuzz	2,012	87.3%	9
Creal	1,487	91.5%	5
GrayC	2,897	99.4%	3
YARPGen	1,470	100%	1
Csmith	1,146	100%	0

Plain C can produce some of these structures but not all of them. Goto-heavy C can produce irreducible CFGs, and union-based type punning can defeat type-based alias analysis (TBAA). Degenerate edges that share a source and destination with a fallthrough edge, and def-use edges for writes that never execute, require an asm declaration. This is also why the dimension stayed untested: Csmith and YARPGen deliberately emit well-structured programs.

5.3 Strategy Design and Scalability

The current mutation strategies target the seven predefined assumption classes (A_1–A_7) derived from an internal analysis of GCC and LLVM. The combinatorial search space of possible injection patterns is massive. Relying strictly on manual template design inherently limits test coverage. Automated template synthesis guided by coverage feedback could address this limitation. A machine-learning based strategy selector could discover novel structural perturbation patterns that human designers might overlook and allocate the mutation budget more efficiently across deep compiler pipelines [6].

5.4 Generalization

The principle of exploiting a trust boundary between the source program and the compiler's internal model generalizes beyond C. Rust's `asm!` macro and C++ inline assembly support similar operand constraint and clobber syntax; adapting EBFuzz to these languages would require a new AST extraction frontend and adjustments to the asm template syntax to match each language's inline assembly conventions.

[2] https://github.com/torvalds/linux/blob/master/arch/x86/include/asm/jump_label.h
[3] https://github.com/torvalds/linux/blob/master/arch/x86/include/asm/cpufeature.h

Recent work on Rust compiler fuzzing [7] confirms growing interest in testing non-C compiler ecosystems. At the IR level, direct manipulation of LLVM IR (inserting unreachable edges, modifying PHI nodes, altering metadata) could bypass syntactic constraints entirely, provided the modified IR remains well-formed. The asm trust boundary exists in any compiler supporting inline assembly with operand constraints (MSVC, ICC); only the register names and constraint syntax differ. The principle of testing a component by exploiting the gap between its declared contract and its internal implementation extends beyond compilers; recent work on sanitizer testing exploits a related trust boundary [13].

5.5 Threats to Validity

We identify the following threats to the validity of our study.

5.5.1 External Validity. Representativeness of Subjects. We selected GCC and LLVM, the two most dominant C/C++ compilers, which is standard practice [5, 29], and evaluated them across three architectures (x86-64, AArch64, RISC-V) at multiple optimization levels. Seeds were generated by Csmith and YARPGen, the most widely adopted tools in the field [16, 29]. Applicability to other compilers (e.g., ICC, MSVC) and to real-world codebases remains open for future work.

Scope of Structural Assumptions. The seven assumption classes (A_1–A_7) are derived from our analysis of GCC and LLVM optimization passes. Other compilers, or future versions of GCC and LLVM, may rely on structural assumptions beyond the current taxonomy. The injection patterns are manually designed and tied to GNU C `asm goto` syntax; compilers without this support are not reachable by the current approach.

5.5.2 Internal Validity. Completeness of the Injection Strategy Set. EBFuzz implements 7 injection patterns (3 τ_c, 4 τ_d) targeting the identified assumption classes. While this set discovered 10 unique bugs across 10 distinct compiler passes, it is not exhaustive. Additional patterns may target the same assumptions more effectively or address assumptions not in the current taxonomy.

Randomness in Injection Point Selection. EBFuzz uses randomized sampling to select injection points (Algorithm 1), introducing non-determinism across runs. We mitigate this by using full 24-hour campaigns that generate thousands of variants and by repeating coverage experiments across ten independent trials.

5.5.3 Construct Validity. Precision of Bug Deduplication. We deduplicate crashes relying exclusively on compiler stack frame hashes and explicit assertion messages. We execute git bisect to deduplicate miscompilations by matching exactly identical point-of-failure inducing commits. These standard methods are approximations: distinct root causes may produce identical traces, and a single cause may manifest at different call sites. We took measures to mitigate the occurrence of duplicate false positives by carefully analyzing crash messages. We did not encounter any instances of false deduplication when testing with GCC and LLVM.

Code Coverage as a Proxy for Thoroughness. Branch coverage indicates exercised code paths but does not guarantee bug discovery. The consistent coverage gains in edge-processing modules, combined with the corresponding distribution of discovered bugs across those same modules, provide converging evidence that the coverage increase reflects meaningful exploration of bug-prone compiler code.

6 Related Work

This section reviews prior work on compiler testing and related aspects of compiler reliability.

6.1 Compiler Testing Techniques

Csmith [29] generates random, semantically valid C programs that are free of undefined behavior and compiles them at multiple optimization levels to detect miscompilations. EMI [11] mutates programs in regions that are dead for a given input, preserving semantics while varying program structure. Subsequent work extends EMI by inserting code into inactive regions [12] or synthesizing equivalent code in active regions [22]. YARPGen [16, 17] generates C/C++ programs targeting scalar optimization bugs. SPE [32] enumerates variable usage patterns in skeleton programs. GrayC [5] applies semantic-aware mutations guided by compiler branch coverage. Creal [14] constructs semantics-preserving test programs from real-world code snippets. MetaMut [18] uses large language models to generate program mutators. Chen et al. [3] improve generation efficiency through memoized configuration search. Wu et al. [28] address the orthogonal problem of eliminating slow-executing test programs to improve throughput. These techniques do not deliberately perturb the compiler's internal graph structures, and the programs they produce seldom exhibit the irregular edge topologies that EBFuzz systematically injects.

6.2 IR-Level and Specialized Testing

IRFuzzer [20] generates valid LLVM IR directly and uses instruction-selection coverage as feedback to guide mutations toward backend code paths. Optimuzz [10] combines optimization-directed fuzzing with translation validation to detect miscompilations. Both require well-formed IR or well-defined semantics as input. EBFuzz operates at the C source level through extended inline assembly, injecting edge structures that are legal C but produce IR that violates the structural assumptions of optimization passes. IRFuzzer injects edges into valid IR and relies on the IR verifier to maintain

well-formedness; EBFuzz injects edges through asm declarations that the compiler accepts as legal C but encodes as IR edges that violate pass-level invariants beyond what the IR verifier checks.

6.3 Other Aspects of Compiler Reliability

Beyond optimization correctness, researchers have studied debug information correctness [15, 26], debugger toolchain reliability [31], and missed optimization detection [24, 25]. COTest [2] explores compiler optimization options to boost testing effectiveness, while Yang et al. [30] isolate compiler faults by differentiating compilation configurations. These approaches target different dimensions of compiler quality than EBFuzz, which focuses on pass robustness against irregular internal edge structures.

7 Conclusion

This study introduces a novel compiler testing approach. It exploits the trust boundary of GNU C extended inline assembly to expose hidden structural assumptions in compiler analyses and optimizations. To achieve this, we strategically inject `asm goto` labels, operand constraints, and clobber lists into existing C test programs. The resulting programs contain irregular control-flow topologies and inconsistent data-flow signals. These structures exercise compiler code paths that are structurally unreachable by programs produced through random generation, equivalence-preserving transformation, or coverage-guided mutation. We implement this concept as a prototype called EBFuzz. Through extensive evaluation on trunk builds of GCC and LLVM, EBFuzz detected a total of 10 unique bugs across both compilers, 9 of which have been confirmed or fixed by the developers.

Data-Availability Statement

EBFuzz and its usage documentation are available at: **https://doi.org/10.5281/zenodo.23242185**.

Acknowledgments

We thank the anonymous reviewers and the shepherd for their valuable feedback. This work was supported in part by the National Natural Science Foundation of China (Grants 624B2067, 62572226, and 62072194), the Jiangsu Natural Science Foundation (Grant BK20231402), the NJU-Huawei Joint Innovation Lab, the Fundamental and Interdisciplinary Disciplines Breakthrough Plan of the Ministry of Education of China (No. JYB2025XDXM118), the "111 Center" (No. B26023), and the Collaborative Innovation Center of Novel Software Technology and Industrialization.

References

[1] Junjie Chen, Wenxiang Hu, Dan Hao, Yingfei Xiong, Hongyu Zhang, Lu Zhang, and Bing Xie. 2016. An empirical comparison of compiler testing techniques. In *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*. ACM, 180–190. doi:10.1145/2884781.2884878

[2] Junjie Chen and Chenyao Suo. 2022. Boosting Compiler Testing via Compiler Optimization Exploration. *ACM Trans. Softw. Eng. Methodol.* 31, 4 (2022), 72:1–72:33. doi:10.1145/3508362

[3] Junjie Chen, Chenyao Suo, Jiajun Jiang, Peiqi Chen, and Xingjian Li. 2023. Compiler Test-Program Generation via Memoized Configuration Search. In *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023*. IEEE, 2035–2047. doi:10.1109/ICSE48619.2023.00172

[4] Karine Even-Mendoza, Cristian Cadar, and Alastair F. Donaldson. 2022. CsmithEdge: more effective compiler testing by handling undefined behaviour less conservatively. *Empir. Softw. Eng.* 27, 6 (2022), 129. doi:10.1007/S10664-022-10146-1

[5] 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, ISSTA 2023, Seattle, WA, USA, July 17-21, 2023*. ACM, 1219–1231. doi:10.1145/3597926.3598130

[6] Fengjuan Gao, Hongyu Chen, Yuewei Zhou, and Ke Wang. 2024. Shoot Yourself in the Foot - Efficient Code Causes Inefficiency in Compiler Optimizations. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering, ASE 2024, Sacramento, CA, USA, October 27 - November 1, 2024*, Vladimir Filkov, Baishakhi Ray, and Minghui Zhou (Eds.). ACM, 1846–1857. doi:10.1145/3691620.3695548

[7] Hongyan Gao, Yibiao Yang, Maolin Sun, Jiangchang Wu, Yuming Zhou, and Baowen Xu. 2025. ClozeMaster: Fuzzing Rust Compiler by Harnessing LLMs for Infilling Masked Real Programs . In *2025 IEEE/ACM 47th International Conference on Software Engineering, ICSE 2025*. IEEE / ACM, 712–712. doi:10.1109/ICSE55347.2025.00175

[8] GCC developers. 2026. Extended Asm — Assembler Instructions with C Expression Operands. https://gcc.gnu.org/onlinedocs/gcc/Extended-Asm.html. Accessed May 20, 2026.

[9] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. 2018. Evaluating Fuzz Testing. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018*, David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang (Eds.). ACM, 2123–2138. doi:10.1145/3243734.3243804

[10] Jaeseong Kwon, Bongjun Jang, Juneyoung Lee, and Kihong Heo. 2025. Optimization-Directed Compiler Fuzzing for Continuous Translation Validation. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 627–650. doi:10.1145/3729275

[11] Vu Le, Mehrdad Afshari, and Zhendong Su. 2014. Compiler validation via equivalence modulo inputs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, Michael F. P. O'Boyle and Keshav Pingali (Eds.). ACM, 216–226. doi:10.1145/2594291.2594334

[12] Vu Le, Chengnian Sun, and Zhendong Su. 2015. Finding deep compiler bugs via guided stochastic program mutation. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2015, part of SPLASH 2015, Pittsburgh, PA, USA, October 25-30, 2015*. ACM, 386–399. doi:10.1145/2814270.2814319

[13] Shaohua Li and Zhendong Su. 2024. UBFuzz: Finding Bugs in Sanitizer Implementations. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1, ASPLOS 2024, La Jolla, CA, USA, 27 April 2024- 1 May 2024*. ACM, 435–449. doi:10.1145/3617232.3624874

[14] Shaohua Li, Theodoros Theodoridis, and Zhendong Su. 2024. Boosting Compiler Testing by Injecting Real-World Code. *Proc. ACM Program. Lang.* 8, PLDI (2024), 223–245. doi:10.1145/3656386

[15] Yuanbo Li, Shuo Ding, Qirun Zhang, and Davide Italiano. 2020. Debug information validation for optimized code. In *Proceedings of the 41st*

ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020. ACM, 1052–1065. doi:10.1145/3385412.3386020

[16] Vsevolod Livinskii, Dmitry Babokin, and John Regehr. 2020. Random testing for C and C++ compilers with YARPGen. *Proc. ACM Program. Lang.* 4, OOPSLA (2020), 196:1–196:25. doi:10.1145/3428264

[17] Vsevolod Livinskii, Dmitry Babokin, and John Regehr. 2023. Fuzzing Loop Optimizations in Compilers for C++ and Data-Parallel Languages. *Proc. ACM Program. Lang.* 7, PLDI (2023), 1826–1847. doi:10.1145/3591295

[18] Xianfei Ou, Cong Li, Yanyan Jiang, and Chang Xu. 2024. The Mutators Reloaded: Fuzzing Compilers with Large Language Model Generated Mutation Operators. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 4, ASPLOS 2024, Hilton La Jolla Torrey Pines, La Jolla, CA, USA, 27 April 2024 - 1 May 2024*. ACM, 298–312. doi:10.1145/3622781.3674171

[19] John Regehr, Yang Chen, Pascal Cuoq, Eric Eide, Chucky Ellison, and Xuejun Yang. 2012. Test-case reduction for C compiler bugs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, Beijing, China - June 11 - 16, 2012*. ACM, 335–346. doi:10.1145/2254064.2254104

[20] Yuyang Rong, Zhanghan Yu, Zhenkai Weng, Stephen Neuendorffer, and Hao Chen. 2025. IRFuzzer: Specialized Fuzzing for LLVM Backend Code Generation. In *Proc. IEEE/ACM International Conference on Software Engineering (ICSE)*. 1986–1998. doi:10.1109/ICSE55347.2025.00130

[21] Moritz Schloegel, Nils Bars, Nico Schiller, Lukas Bernhard, Tobias Scharnowski, Addison Crump, Arash Ale Ebrahim, Nicolai Bissantz, Marius Muench, and Thorsten Holz. 2024. SoK: Prudent Evaluation Practices for Fuzzing. In *IEEE Symposium on Security and Privacy, SP 2024, San Francisco, CA, USA, May 19-23, 2024*. IEEE, 1974–1993. doi:10.1109/SP54263.2024.00137

[22] Chengnian Sun, Vu Le, and Zhendong Su. 2016. Finding compiler bugs via live code mutation. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2016, part of SPLASH 2016, Amsterdam, The Netherlands, October 30 - November 4, 2016*, Eelco Visser and Yannis Smaragdakis (Eds.). ACM, 849–863. doi:10.1145/2983990.2984038

[23] Chengnian Sun, Vu Le, Qirun Zhang, and Zhendong Su. 2016. Toward understanding compiler bugs in GCC and LLVM. In *Proceedings of the 25th International Symposium on Software Testing and Analysis, ISSTA 2016, Saarbrücken, Germany, July 18-20, 2016*. ACM, 294–305. doi:10.1145/2931037.2931074

[24] Theodoros Theodoridis, Manuel Rigger, and Zhendong Su. 2022. Finding missed optimizations through the lens of dead code elimination. In *ASPLOS '22: 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, 28 February 2022 - 4 March 2022*. ACM, 697–709. doi:10.1145/3503222.3507764

[25] Theodoros Theodoridis and Zhendong Su. 2024. Refined Input, Degraded Output: The Counterintuitive World of Compiler Behavior. *Proc. ACM Program. Lang.* 8, PLDI (2024), 671–691. doi:10.1145/3656404

[26] Theodore Luo Wang, Yongqiang Tian, Yiwen Dong, Zhenyang Xu, and Chengnian Sun. 2023. Compilation Consistency Modulo Debug Information. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2, ASPLOS 2023, Vancouver, BC, Canada, March 25-29, 2023*. ACM, 146–158. doi:10.1145/3575693.3575740

[27] Jiangchang Wu, Yibiao Yang, Maolin Sun, and Yuming Zhou. 2025. Unveiling Compiler Faults via Attribute-Guided Compilation Space Exploration. In *Proceedings of the 2025 USENIX Annual Technical Conference, USENIX ATC 2025, Boston, MA, USA, July 7-9, 2025*. USENIX Association, 1109–1125. https://www.usenix.org/conference/atc25/presentation/wu-jiangchang

[28] Jiangchang Wu, Yibiao Yang, and Yuming Zhou. 2023. Boosting Compiler Testing via Eliminating Test Programs with Long-Execution-Time. In *IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2023, Taipa, Macao, March 21-24, 2023*. IEEE, 593–603. doi:10.1109/SANER56733.2023.00061

[29] 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, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*. ACM, 283–294. doi:10.1145/1993498.1993532

[30] Yibiao Yang, Qingyang Li, Maolin Sun, Jing Yang, Jiangchang Wu, and Yuming Zhou. 2025. Isolating Compiler Faults through Differentiated Compilation Configurations. *IEEE Transactions on Software Engineering* (2025), 1–16. doi:10.1109/TSE.2025.3569530

[31] Yibiao Yang, Maolin Sun, Jiangchang Wu, Qingyang Li, and Yuming Zhou. 2025. Debugger Toolchain Validation via Cross-Level Debugging. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1, ASPLOS 2025, Rotterdam, The Netherlands, 30 March 2025 - 3 April 2025*. ACM, 280–294. doi:10.1145/3669940.3707271

[32] Qirun Zhang, Chengnian Sun, and Zhendong Su. 2017. Skeletal program enumeration for rigorous compiler testing. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*. ACM, 347–361. doi:10.1145/3062341.3062379