CLower: Detecting Compiler Pessimization Bugs through Redundant Memory Accesses

JIANHAO XU, Southeast University, China
KUNBO ZHANG, State Key Laboratory for Novel Software Technology, Nanjing University, China
MATHIAS PAYER, EPFL, Switzerland
KANGJIE LU, University of Minnesota, USA
BING MAO, State Key Laboratory for Novel Software Technology, Nanjing University, China

Compilers are expected to generate optimized code, but they sometimes introduce pessimizations, quality-degrading redundant instructions. These bugs not only incur performance overhead but also, critically, expand the attack surface by introducing unexpected side effects (e.g., redundant memory accesses) without breaking compilation correctness. Existing bug-finding methods are neither designed for nor effective at identifying such security-sensitive pessimizations.

This paper presents CLower, a novel, black-box approach for automatically detecting compiler pessimizations via redundant memory accesses. CLower's core insight is that any extra global memory accesses in a fully optimized binary, compared to the source, indicate a pessimization. To reliably distinguish compiler-introduced redundancy from source-level redundancy, we generate random C programs in which each global variable has a predetermined, controlled number of memory accesses. CLower then executes the instrumented binary and verifies whether superfluous accesses have been introduced during compilation.

We applied CLower to GCC and LLVM, reporting 23 unique bugs (21 in GCC, 2 in Clang), with 16 confirmed as new pessimization bugs. Our evaluation shows that CLower accurately detects diverse, impactful pessimization bugs, the majority of which (75%) also manifest for heap-allocated objects, demonstrating that the underlying compiler flaws are general and not limited to global memory. Furthermore, we identify a systematic conflict between compiler optimizations and pessimization bugs, which causes many such bugs to remain hidden in compiler versions. This study sheds light on the under-explored area of compiler pessimization and provides a practical tool for improving compiler quality.

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

Additional Key Words and Phrases: Compiler Pessimization, Randomized Testing

ACM Reference Format:
Jianhao Xu, Kunbo Zhang, Mathias Payer, Kangjie Lu, and Bing Mao. 2026. CLower: Detecting Compiler Pessimization Bugs through Redundant Memory Accesses. *Proc. ACM Program. Lang.* 10, OOPSLA1, Article 142 (April 2026), 27 pages. https://doi.org/10.1145/3798250

1 Introduction

Compilers are fundamental to software performance and security. Decades of research and engineering have been devoted to developing advanced optimizations [3, 19], with the singular goal of making generated code faster and smaller. Yet, sometimes, compilers achieve the opposite: they

Authors' Contact Information: Jianhao Xu, Southeast University, Nanjing, China, jianhao.xu@seu.edu.cn; Kunbo Zhang, State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, China, absoler@smail.nju.edu.cn; Mathias Payer, EPFL, Lausanne, Switzerland, mathias.payer@nebelwelt.net; Kangjie Lu, University of Minnesota, Minneapolis, USA, kjlu@umn.edu; Bing Mao, State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, China, maobing@nju.edu.cn.

ACM 2475-1421/2026/4-ART142
https://doi.org/10.1145/3798250

```
int f, a, b, c;
void func(){
    if (f) {
        a = 1;
        b = 2;
        c = 3;
    } else {
        a = 2;
        b = 4;
        c = 6;
    }
}
```

(a) C code

```
func():
mov     0x2f06(%rip),%eax # <f>
neg     %eax
sbb     %ecx,%ecx
add     $0x2,%ecx
cmpl    $0x1,0x2ef8(%rip) # <f>
sbb     %edx,%edx
mov     %ecx,0x2efc(%rip) # <a>
and     $0x2,%edx
add     $0x2,%edx
cmpl    $0x1,0x2ee3(%rip) # <f>
sbb     %eax,%eax
mov     %edx,0x2ee3(%rip) # <b>
and     $0x3,%eax
add     $0x3,%eax
mov     %eax,0x2ed3(%rip) # <c>
retq
```

(b) Assembly

Fig. 1. A compiler pessimization bug found by CLower

actively degrade code quality by introducing redundant instructions [35], a phenomenon known as compiler pessimization. Unlike missed optimization opportunities [29], a pessimization produces objectively worse code than an unoptimized version. This phenomenon is illustrated in Figure 1, where GCC's O1 output contains multiple redundant loads of global variable f (lines 2, 6, 11 in assembly), a clear degradation from the original, simpler C code.

Compiler pessimization bugs affect numerous programs and cause significant performance degradation. More critically, these bugs introduce unique security implications by expanding attack surfaces in the correctness-security gap [13], where compilers preserve functionality while violating source-level security guarantees. These vulnerabilities are particularly dangerous because they evade both compiler verification [22, 45] and source-level detection [43].

Memory-related pessimizations are particularly severe because they introduce deterministic side effects with direct security implications. Compiler-introduced stores can violate concurrency correctness in multi-threaded contexts [28], while introduced loads may escalate otherwise benign data races into exploitable vulnerabilities. Although data races constitute undefined behavior according to the C/C++ standard, low-level systems code, such as the Linux kernel, often includes intentional, performance-motivated data races that are assumed to be safe. Real-world incidents confirm that compiler-introduced extra loads have been exploited to bypass security checks [39] or trigger null-pointer dereferences [16, 20], demonstrating that such pessimizations result in practical security violations. Beyond concurrency, they can potentially undermine software security mechanisms [18] that assume atomic memory access patterns, such as some Control-Flow Integrity (CFI) implementations. A critical example in Figure 2 shows how a single redundant load (line 5 in assembly) of an indirect jump index creates a "time-of-check to time-of-use" (TOCTTOU) window between the bounds check (line 1) and the second fetch (line 5). Under an attacker model where memory can be modified between these accesses, this violates the atomicity assumption implicit in many CFI implementations, creating an opportunity for control-flow hijacking [42]. The attack succeeds because the extra load violates CFI's underlying assumption that the index is fetched atomically for the control-flow transfer. While the actual attack surface depends on runtime context

```
    switch (mca_p->port_format) {
    case SDP_PORT_NUM_ONLY:
        SDP_PRINT("...");
    default:
        SDP_PRINT("...");
        break;
    }
```

(a) C code (in Firefox)

```
cmp $0x6, 0x118(%r13)
pop %rcx
pop %rsi
ja $0x9d6720
mov 0x118(%r13), %eax
lea 0x4fee16c(), %rdx
movsxd (%rdx, %rax, 4), %rax
add %rdx, %rax
jmp %rax
```

(b) Assembly with a double fetch

Fig. 2. A security-related compiler pessimization bug that allows the bypass of CFI protection

(e.g., concurrency, CFI), the vulnerability pattern itself, an extra access introduced by the compiler, is systemic and detectable across diverse scenarios. Such patterns demand systematic detection, as their potential exploitation depends on contextual conditions widely present in real-world code. These predictable attack behaviors underscore the critical importance of early detection for memory-related pessimizations.

Identifying compiler pessimization bugs remains a significant challenge, primarily due to the **lack of an effective test oracle**. Existing work on missed optimizations, such as Barany et al. [4] and Theodoridis et al's [36], has uncovered many compiler bugs, but their approaches cannot detect pessimizations. Barany et al.'s method searches for predefined missed optimization patterns, explicitly excluding pessimizations, while Theodoridis et al.'s focuses on unoptimized dead code blocks, which do not overlap with pessimization bugs. Recent efforts, such as ProgramMarkers [37] and Gao et al. [15], propose test oracles for pessimization-like bugs by refining or degrading source code and observing whether the compiler generates worse output for better input. However, these methods are limited to specific semantic patterns due to their constrained code generation strategies, preventing them from effectively detecting security-related pessimizations. ProgramMarkers relies on injected keywords like `__builtin_unreachable`, while Gao et al.'s approach modifies code using predefined deoptimization rules. As a result, neither can effectively identify pessimizations involving broader semantics, such as memory access. A central challenge in detecting memory-related pessimizations lies in **distinguishing compiler-introduced redundancies from those present in the source code**. Prior work on memory access monitoring, including DeadSpy [9], RedSpy [38], LoadSpy [34], and Toddler [30], target programmer inefficiencies and do not attribute redundancies to the compiler. CIDetector [35] studies compiler performance bugs via memory access patterns but requires heavy manual analysis to disentangle source-level from compiler-introduced redundancies, which limits its scalability. Cmmtest [28] detects concurrency bugs in GCC by comparing memory traces between source and compiled code, confirming four bugs. While it could be extended to track redundant memory accesses, failing to address the redundancy distinction challenge would lead to severe false negatives, as compiler-eliminated redundancies would mask compiler-introduced ones. This entanglement also **undermines differential testing** across compilers or optimization levels. Divergent yet legal optimizations of source redundancies lead to benign discrepancies, while uniformly eliminated source redundancies can hide newly introduced accesses across all tested compilers, resulting in systematic false negatives. In summary, despite these efforts, no existing method effectively detects security-critical pessimizations, particularly those involving memory access, while remaining both scalable and precise.

This paper introduces a straightforward and widely applicable method for automatically detecting compiler pessimizations at the level of instructions. The core insight of our test oracle is that a compiler is likely introducing pessimizations if one complete optimization pipeline produces binaries with more global memory accesses than those inherently present in the source code. While we focus on bugs related to memory accesses, prior work such as LoadSpy [34] has demonstrated that redundant memory accesses are a significant indicator of various software inefficiencies. The key to reliable detection lies in **disentangling compiler-introduced redundancies from source-level ones**. To achieve this, CLower generates random C programs in which every global variable has a **predetermined and controlled number of memory accesses**. By instrumenting the compiled binary and comparing the dynamically observed access counts against the predetermined baselines, CLower can reliably identify compiler-introduced extra accesses, thus isolating pessimization bugs from source-level noise.

This approach is scalable, treating compilers as black boxes, and can test any C compilers without requiring modifications. Furthermore, it does not rely on differential testing between compilers, thus enabling testing for an individual compiler and avoiding the oversight of common bugs shared among compilers or optimization levels. Additionally, unlike Theodoridis et al's [36] or ProgramMarkers [37], it examines compiler behavior beyond existing dead code in the source, facilitating the detection for more prevalent bugs with real-world impact. Our method provides a precise and definitive test oracle, and the verification phase is simple and fast. Our empirical analysis demonstrates the usefulness of CLower in uncovering a wide range of compiler pessimizations. We have uncovered numerous new compiler pessimization bugs in the latest compilers. Specifically, in GCC 13.2.0, we reported 21 unique cases and 16 confirmed including 7 fixed, while in LLVM 17.0.6, we reported 2 new cases with none confirmed.

During regression testing, we also observe a systematic conflict between compiler optimizations and compiler pessimization bugs. Specifically, the effects of a compiler pessimization bug can be nullified by subsequent compiler optimizations, thereby hiding the unfixed bug. However, these hidden bugs have the potential to resurface and impact performance as the compiler evolves over time. This conflict highlights the existence of hidden pessimization bugs that current approaches fail to target. To investigate further, we conducted a study on a benchmark generated from our random code generation process. Our findings revealed that: (i) hidden pessimization bugs are prevalent, with many historical bugs remaining unresolved and hidden in the latest compilers. (ii) many regression pessimization bugs have been hidden for a long time (about 2 years on average) before surfacing in the latest versions.

Our key contributions are:

- A novel, black-box testing tool, CLower, for finding compiler pessimization bugs through the lens of redundant memory accesses.
- An extensive evaluation of CLower, demonstrating its practical utility by uncovering 16 confirmed pessimization bugs in GCC.
- A thorough analysis of the detected bugs, highlighting their explicit performance overhead and security implications in concurrent scenarios.
- The identification of systematic conflict between compiler optimizations and pessimization bugs, explaining the root cause of many hidden bugs that evade existing detection methods.

2 The CLower Approach: Detecting Pessimization Bugs

2.1 Overview and Key Insights

CLower is built on two key insights that together enable precise and scalable detection of memory-related pessimizations.

Insight 1: A Precise Test Oracle. The core of our detection method is the observation that a compiler introduces a pessimization if its complete optimization pipeline produces a binary with

more global memory accesses than are present in the source code. We focus on **global** variables for several reasons: their accesses have critical security implications in concurrent contexts [28, 43]; they yield fewer false positives compared to locals due to stronger source-binary consistency; and they avoid reliance on often-unreliable debugging information [12]. Furthermore, as evaluated in subsection 3.6, bugs found in global memory often generalize to heap objects.
Insight 2: Isolating Compiler-Introduced Redundancy. To distinguish compiler-introduced redundancies from those present in the source, we generate programs where **the number of global accesses is predetermined and controlled**. This design prevents a critical masking effect that undermines existing detection approaches and leads to false negatives. When source code already contains redundancies, a compiler may legally remove them; if it simultaneously introduces new ones, the two changes can cancel out, leaving the total access count unchanged and hiding the bug entirely. Differential testing cannot overcome this limitation, as compilers may optimize source redundancies in different but legitimate ways, or all may eliminate the same redundancies, both scenarios hide compiler-introduced accesses. By ensuring each global variable is accessed only a predetermined number of times in the source, we establish a deterministic baseline where any extra access in the binary is unambiguously a compiler-introduced pessimization, effectively eliminating this class of false negatives.

Thus, we propose CLower, a tool designed to identify compiler's unnecessary introduction of memory accesses. CLower generates random C code in which all **global variables** have a predetermined number of memory accesses and dynamically checks if the compiled binary exhibits more accesses than expected. Figure 3 depicts the workflow of CLower, which comprises two main components the Test Program Generator and the Dynamic Checker.

- The Test Program Generator creates random C code where all global variables have a predetermined number of memory accesses. The generated C code comprises two parts: the testing part, which contains random semantics under restriction for testing purposes, and the metadata part, which contains access information of the tested variables for runtime usage.
- The Dynamic Checker verifies the memory accesses in the compiled binary, thereby testing the used compiler. It instruments the compiled test program to record every memory access during runtime. After recording the memory accesses, the Dynamic Checker compares the actual access numbers of each global variable with the predetermined numbers to identify pessimization bugs. Additionally, it generates a bug summary containing details such as the accessed variables, expected and actual access numbers, and opcodes of relevant instructions. This summary aids in test case reduction and regression testing efforts. These approaches enable the validation of pessimization bugs and the generation of bug reports by simply running the instrumented binary code.

We address two key design questions in CLower:

Q1: How to restrict the access number of a global variable? We adopt a two-fold approach: **(i) access-type separation:** We enforce separate restrictions for memory loads and stores. Each test case constrains only one type of access, enhancing the diversity of generated code for testing each kind. **(ii) single-source-occurrence:** We ensure the restricted memory accesses occur only once within the source code context (e.g., function or loop bodies). This mandates that tested accesses for each global variable are limited to a **single location** in the source.

This design is motivated by: **(i) higher throughput:** It simplifies code generation and dynamic validation by avoiding complex static analysis to compute access numbers across control flow branches. **(ii) focused bug detection:** It targets the most common scenarios. Our analysis of GCC's test suite shows that 63.7% of test cases have fewer than two accesses per global variable.

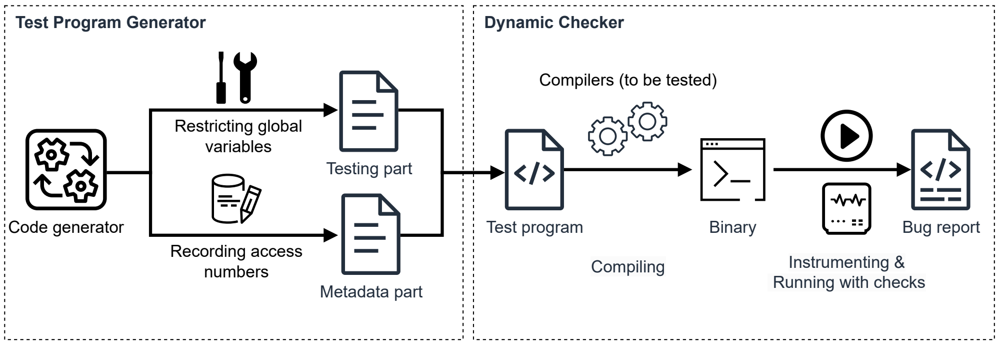


Fig. 3. The Workflow of CLower

This approach also avoids false negatives from compiler optimizations (e.g., copy propagation) that might eliminate redundant accesses, which could otherwise hide pessimization bugs.

Q2: How to pass access information of tested variables for runtime usage? Our solution involves: **(i) code partitioning:** The generated code is divided into: (i) a testing part containing random semantics, (ii) a metadata part that stores variable information (name, size, address) and predetermined access numbers via dedicated function arguments. These arguments are read during dynamic instrumentation. **(ii) dynamic access recording:** Although source-level accesses are restricted to one occurrence, the actual runtime count (affected by loops or function calls) is calculated and recorded in the metadata. This ensures accurate comparison with runtime accesses.

This design enables the dynamic validator to obtain all necessary information directly from the binary, streamlining the validation process.

2.2 Test Program Generator

The Test Program Generator creates random C programs where each global variable has a predetermined number of memory accesses (either one load or one store) in the source code. This design provides a precise test oracle: any additional access in the compiled binary indicates a compiler-introduced pessimization.

Program structure and Metadata. As illustrated in Figure 4, each generated program follows a structured design that cleanly separates testing logic from runtime metadata:

- **Testing Part:** Contains the core semantics under test, including global variables (e.g., g), randomly generated **entry function** (e.g., `func`) that perform the restricted memory accesses and other random functions called by the entry function.
- **Metadata Part:** Defines two global arrays that serve as communication channels with the dynamic checker:
  - `varInfos[]`: Stores variable information (address, size, name).
  - `accessInfos[]`: Records expected access patterns (address, length, count).
- **Initialization Sequence:** The `main` function executes a precise initialization routine:

(1) Calls `get_info()` to expose array addresses to instrumentation tools.
(2) Invokes `record_var()` and `record_access()` to populate metadata arrays.
(3) Executes the testing logic via `func()`.

This design ensures the dynamic checker can precisely correlate observed memory accesses with their source-level expectations during program execution.

```
/* testing part     */
int g;
void func() {
    read(g);
}

/* metadata part   */
struct Var{  // the variable information array
    unsigned long addr, size;
    char name[10];
}varInfos[MAX_GLOBAL_NUM];

struct Access{  // the access information array
    unsigned long addr, length;
    int cnt;
}accessInfos[MAX_ACCESS_NUM];

int main() {
    get_info(&varInfos, &accessInfos); // runtime addresses of two arrays
    record_var(&g, sizeof(g), "g");  // recording variable information
    record_access(&g, sizeof(g), 1); // recording access information
    func(); // calling the testing part
}
```

Fig. 4. The structure of CLOWER's generated programs

Generating Random Code with Access Restrictions. We build upon Csmith [45], fundamentally extending its random generation workflow to enforce global access restrictions and compute the metadata required for our test oracle. As illustrated in Figure 5, our process is a pipeline of staged analyses and generations.

The generation begins at the highest level with the **entry function** and other functions it calls. A function's signature is first created, after which its body is generated. Crucially, after the entire function body is complete, we perform a **parameter & loop analysis**. This analysis produces key **function summaries**: The **parameter summary** catalogs all dereferences of function parameters, which is essential for tracking implicit accesses at call sites. The **loop summary** records the execution bounds of loops. These summaries are then used in the subsequent **access number calculation** phase. Here, we statically compute the precise dynamic execution count for each restricted global access, factoring in loop iterations and function calls. This calculated count is embedded as **main function metadata**, forming the baseline for runtime validation by the dynamic checker.

The key to our approach is the integration of access control at the point of **statement generation**. For each new statement (e.g., an assignment `[] = []`), the generator must select a variable from the pool. Crucially, before any variable is used, our system performs a series of **access checks** to ensure it complies with the single-access-per-global restriction. This involves consulting the fine-grained access records to determine if the candidate variable (or the memory it points to) has already been accessed in the restricted access type (load/store). Only after a variable passes these checks is it selected, and the statement is generated. Subsequently, the successful access is immediately recorded in the **access recording** module, updating the state for future checks. This tight integration, where generation, checking, and recording are interleaved at the statement level, ensures that the strict access invariant is maintained throughout the construction of complex code structures, including those involving pointers, function calls, and loops.

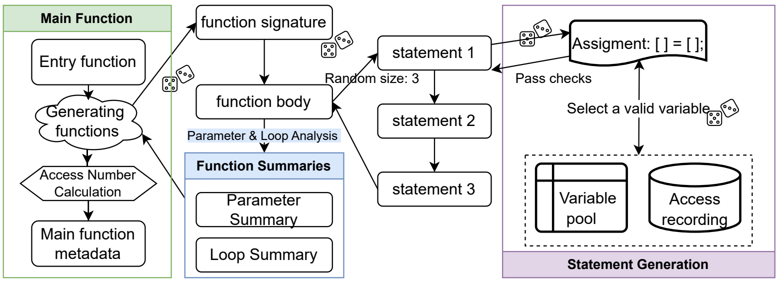


Fig. 5. Code generation workflow in CLower, extending Csmith with access restrictions

Ensuring the restricted number of memory accesses. During random code generation, we (i) enforce that each test case restricts either memory loads or memory stores, (ii) ensure that the tested type of memory access (load/store) on global variables occurs only once within the source code. Figure 6 illustrates how CLower generates an assignment statement while maintaining the restricted number of memory accesses. As depicted, when write access is restricted, and a global variable is required on the left-hand side of the assignment statement, CLower first selects (or creates) a variable that has never been written to. This ensures that the ongoing memory access of this candidate variable will not violate the imposed restriction. Subsequently, the system rechecks if the entire statement complies with our restrictions and ensures that no other statements are affected in a way that would violate these restrictions. Once validated, the selected variable can be used, and this access is recorded for future reference. In summary, we implement the restrictions as three integrated components within the test program generation process:
(I) Check before selection. This component ensures that a random global variable is selected in a manner that does not violate memory access restrictions. Specifically, CLower selects a never-read or never-written global variable (global variable that has never been read from or written to), depending on whether the restriction is on memory loads or stores. Both direct selection of the variable and selection through a pointer should be considered. To determine if a variable can be used for accessing, i.e., if it has never been loaded or stored, our system has to address two main challenges. **Challenge 1: maintaining the accessed status of tested variables**. Overcoming this allows us to establish a set of candidate variables. Csmith's analysis infrastructure can be reused to track memory accesses and resolving memory aliases during generation. However, Csmith treats some complex data structures as a whole in some analysis. Directly borrowing such analysis would prevent our system from detecting more fine-grained pessimization bugs. Therefore, CLower also tracks each array item and structure field individually. Accordingly, before variable selection, CLower adds index-sensitive checks for arrays and field-sensitive checks for structs and unions. For example, when restricting memory writes, to check if `*p` can be selected as the left-hand side of an assignment, CLower will check its point-to facts, i.e., `v1` or `v2`. If these two variables have never been written to, `*p` can be selected for generation. After the generation, we track the read of `p` and the write of `v1` and `v2`, updating the accessed status accordingly.

Challenge 2: potential global access through function arguments. In addition to direct global memory accesses within a function, there are implicit global memory accesses through

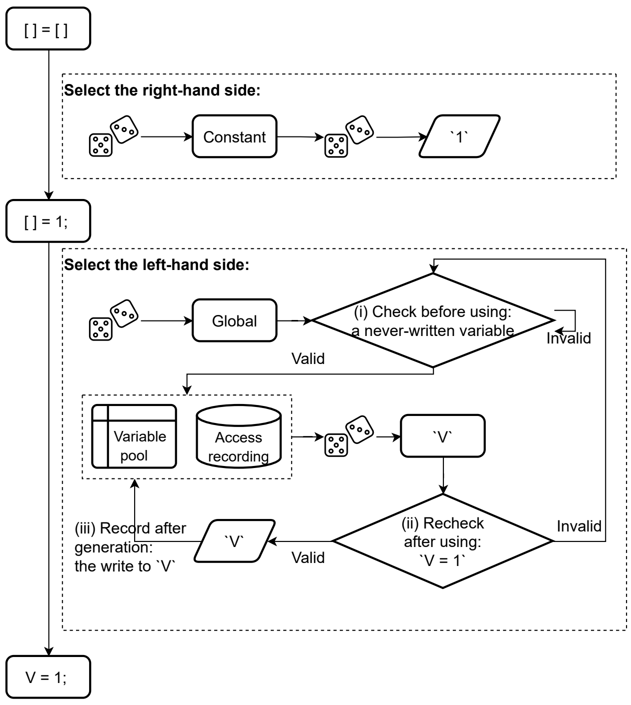
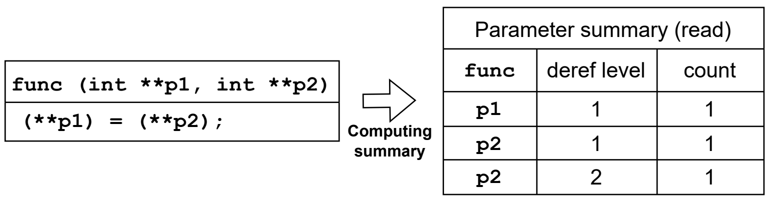

[ ] = [ ]
Select the right-hand side:
Constant
`1`
[ ] = 1;
Select the left-hand side:
Global
(i) Check before using: a never-written variable
Invalid
Valid
Variable pool
Access recording
`V`
(ii) Recheck after using: `V = 1`
Invalid
Valid
`V`
(iii) Record after generation: the write to `V`
V = 1;

Fig. 6. CLower's generating a statement with restrictions

pointer-type arguments. These potential global memory accesses are not determined until the function is called. To track and restrict these accesses, we (i) summarize a 'parameter-summary' for every function after its generation and (ii) postpone the restriction check to the time when the function is called and the involved argument is selected. The 'parameter-summary' records the numbers of all dereferences, of the tested type (load/store), of parameters for every function. Figure 7 shows the parameter-summary of an example function. As depicted, for func, all levels of read dereferences (load) of the two parameters, i.e., p1 and p2, are recorded. Before selecting arguments for a call to this function, CLower adds checks based on this 'parameter-summary' to ensure there will be no new reads to already-read variables. For example, since func introduces a read to *p2 and **p2, CLower should not select a pointer value (e.g., v1) as the argument for p2 if this pointer's single dereferences (*v1) or double dereferences (**v1) have been read before.



```
func (int **p1, int **p2)
 (**p1) = (**p2);
```

Parameter summary (read)		
func	deref level	count
p1	1	1
p2	1	1
p2	2	1

Fig. 7. The parameter-summary of an example function

(II) Recheck after using&generation. This rechecking component ensures that the generated statement is valid within language specifications and our restrictions. For language specifications, CLower checks the validity of the statement after variable selection and the generation of each basic block to prevent undefined behaviors. For our restrictions, CLower verifies that the statement itself satisfies our restrictions and does not cause other statements to violate them. Specifically, CLower mainly checks if **back edges** introduce unexpected memory accesses. Since code is generated incrementally, back edges can affect the soundness of memory access analysis on previously generated code. CLower places this check after the generation of each basic block. If the check is not passed, i.e. some back edge introduces extra memory accesses, CLower will delete the statement containing the introduced accesses and all the following statements in the basic block. It is noteworthy that for the code generated by CLower, only for-loops can introduce back edges in the control flow graph. This is because: (i) to avoid generating non-terminating programs, CLower disables `goto` statements and circular calls. (ii) CLower follows Csmith's approach of using only for loops to generate loops.
(III) Record after generation. This phase finalizes the test case by recording all access information and computing the expected execution counts.

Access tracking. During code generation, CLower tracks every permitted memory access. We reuse Csmith's memory alias analysis to precisely track accesses, and extend it for individual elements of arrays and fields of structures. This fine-grained tracking is essential for detecting subtle pessimization bugs. The recording is maintained per statement, allowing for clean rollback if a statement is deleted during post-generation checks.

Computing expected access counts. While each global variable is accessed only once in the source code, its actual execution count is determined by loops and function calls. After generating the entry function, CLower performs a static analysis to compute these dynamic counts. The calculation uses the function's call graph, parameter summaries, and loop structures to determine the final execution frequency for each restricted access in the entry function.

Embedding metadata. The final step is to embed the calculated counts and variable information into the test program's metadata section. As shown in Figure 4, this is achieved through generated function calls (`record_var`, `record_access`) that populate the global arrays `varInfos` and `accessInfos`. A call to `get_info` exposes these arrays to the dynamic checker, enabling runtime verification against the predetermined expectations.

2.3 Dynamic Checker

The dynamic checker is designed to verify if the execution of a compiled test case shows extra memory access with comparison to the predetermined ones. We build the dynamic checker with Intel Pin [26], a dynamic binary instrumentation tool.

The checker has implemented three phases: **(i) getting the information of tested variable and predetermined access numbers**. The checker instruments the compiled code according to the test program structure depicted in Figure 4. It first gets the runtime address of the metadata arrays. Then it can fetch all predetermined information from this two global arrays after executing all the calls to `record_access`. This information includes starting addresses, sizes and access numbers of all tested variables. **(ii) getting the actual number of global memory accesses**. The checker instruments all global memory accesses and records the starting addresses and sizes of all these accesses. **(iii) comparing**. The checker compares the expected memory access with the actual ones for unexpected memory access introduced by compilation. Once the actual memory accesses **exceed** the scope of expected accesses, CLower will report a potential bug.

2.4 Implementation

We implemented CLower by extending the Csmith-2.3.0 test-case generator with approximately 2,700 lines of C++ code to enforce global variable access restrictions. The dynamic checker is built with Intel Pin [26]. Approximately 3,000 lines of supporting scripts in Python, shell, and Perl orchestrate the entire testing workflow, including test generation, dynamic checking, and result post-processing.

3 Evaluation

To evaluate CLower, we aim to address the following research questions:

- RQ1: What is the **practical utility** of CLower?
- RQ2: How is the **diversity of bugs** found by CLower?
- RQ3: Does our **test oracle** accurately identify pessimization bugs, or does the introduction of extra global memory accesses by a compiler indicate a pessimization bug?
- RQ4: Do bugs reported by CLower cause **observable performance degradation**?
- RQ5: Do bugs reported by CLower cause **reproducible security-violating behaviors** under concurrency scenarios?
- RQ6: Do performance bugs reported by CLower demonstrate **generalizability** to impact non-global variables?

3.1 RQ1: Practical Utility

To test the practical utility of CLower, we deployed it to assess its efficiency and effectiveness of the most popular open-source compilers, namely GCC and LLVM(Clang).

Experimental Environment: We conducted our experiments on a system based on Intel(R) Xeon(R) CPU E5-2620v4 (X86-64) running Ubuntu 20.04. The latest compiler versions available at the time of testing were used, namely GCC 13.2.0 and Clang 17.0.6. `O2` optimization option is selected for the efficiency experiment, while `O1`,`O2` and `O3` are selected for the effectiveness experiment. For test case reduction, we utilized C-Reduce [31] along with our customized interestingness test script. This script is designed to determine whether the features required by the user are still preserved in the reduced code.

Efficiency. To evaluate the efficiency of CLower, we employ it to generate 100,000 random C programs with restricted loads and 100,000 with restricted stores. Table 1 shows the efficiency result. The random code generation process takes 3,965.1 seconds, while the dynamic checking part, including compilation and instrumentation of test code, requires a total of 5,262 seconds. The bottleneck lies in the test case reduction, a known time-consuming task in compiler testing [10]. CLower is designed to be easily checked by simply running the compiled binary, which also facilitates redundant processing. On average, CLower takes 333 seconds per test case for reduction, whereas Theodoridis et al.'s method [36] takes 4-8 hours. This advantage increases with the number of test cases. From these observations, we can conclude that the main pipeline of CLower is efficient enough for practical utility.

Table 1. Time spent on 100,000 restricted Load and 100,000 restricted Store test cases

	Generation	Dynamic Checking	Reduction
Time Spent(s)	3,965.1	5,262	333/case

Effectiveness. We evaluate CLower's effectiveness by generating 100,000 random C programs with restricted loads and 100,000 with restricted stores, using the same settings as our efficiency tests. As shown in Table 2, CLower identified potential bugs in 817 GCC cases and 179 Clang cases. Following test case reduction and manual analysis for bug deduplication, we categorized these findings into 22 potential pessimization bugs for GCC and 2 for Clang, all of which have been reported to their respective compiler communities. The current status of these reported bugs shows significant developer engagement. At the time of writing, the compiler developers have: confirmed 16 pessimization bugs in GCC (including 5 store-related cases); marked 1 reported case as a duplicate of another our confirmed report; fixed 7 bugs. We also identified 2 performance bugs that did not meet our pessimization criteria, as they did not degrade code quality relative to the source. Compiler developers marked both of them as confirmed.

When compared to state-of-the-art approaches, CLower demonstrates competitive detection capabilities for pessimization bugs. Gao et al. [15] identified 17 confirmed GCC bugs (with none fixed at publication), while ProgramMarkers [37] detected 27 confirmed GCC bugs (with 12 fixed at publication). As evidenced in Table 3, CLower particularly excels in detecting memory-access-related compiler bugs, uncovering 18 instances compared to cmmtest's [27, 28] 11 confirmed cases. Notably, in the specialized domain of GCC concurrency bugs—the primary focus of cmmtest—our approach outperformed the prior work, identifying 5 concurrency bugs versus cmmtest's 4. It is worth emphasizing that at least 3 of the 5 concurrency bugs found by CLower have existed in the compiler since before the time of Cmmtest's evaluation, underscoring their elusive nature and the effectiveness of our method in uncovering long-standing issues. All of our discovered concurrency bugs involve cases where compilers incorrectly introduced additional global store operations, thereby violating C/C++ concurrency correctness. These comprehensive results validate CLower's capability in effectively detecting both general compiler pessimizations and memory-access-specific compiler bugs, demonstrating its value as a precise testing framework for modern optimizing compilers.

Table 2. A summary of CLower's pessimization-bug-finding result

Category	**GCC**	**LLVM**
100,000 Load&Store cases each	—	
Bug-triggering test cases	568	151
Reported cases	22	2
New reports	21	2
Confirmed pessimization bugs	**16**	0
Fixed bugs	7	0
Confirmed other performance bugs	**2**	0
Won't-fix cases	2	0
Unconfirmed cases	1	0
Duplicate bugs	1	0
Confirmed pessimization bugs	1	0
Fixed bugs	1	0

Table 3. Comparison of memory-access-related compiler bug detection between cmmtest and CLower. "CC-Memory" represents general memory-access-related compiler bugs, while "CC-Concurrency" denotes compiler concurrency bugs.

Tool	Confirmed Bugs	
	CC-Memory	CC-Concurrency
cmmtest [27, 28]	8	4
CLower	18	5

Table 4. Compiler components implicated in the identified bugs

Compiler Component	Number of Related Bugs
middle-end	6
tree-optimization	1
rtl-optimization	6
target	5

Takeaway. The evaluation demonstrates that CLower is effective in detecting compiler pessimization bugs and memory-access-specific compiler bugs, is efficient in terms of time and resource utilization, and is scalable to different compilers. These characteristics validate the practical utility of CLower.

3.2 RQ2: Bug Diversity

In this section, we evaluate the diversity of pessimization bugs discovered by CLower from various perspectives, including the involved compiler components, root causes, and semantic diversity of test cases.

Compiler Components: To identify the compiler components potentially responsible for the identified bugs, we analyze the responses received from compiler developers following our bug reports. The results, presented in Table 4, indicate that the pessimization bugs detected by CLower span multiple optimization stages within the compiler pipeline, from high-level middle-end optimizations down to target-specific code generation.

Notably, the bugs are distributed across all major compiler components, with the `middle-end` and `rtl-optimization` layers each containing 6 bugs, the `target` component containing 5 bugs, and the `tree-optimization` layer containing 1 bug. This distribution indicates that redundant memory access bugs are not confined to any single optimization phase but rather manifest throughout the compiler's optimization pipeline. The presence of bugs in the `target` component further suggests that some pessimizations arise from architecture-specific optimizations, highlighting the need for cross-platform testing methodologies.

Root Cause. After analyzing the buggy compiler components, we categorized the root causes into two types: (i) Implementation mistakes within individual optimizations: This pertains to instances where an optimization inadvertently introduces redundant code. (ii) Scheduling mistakes among different optimizations: This occurs when compilers fail to schedule subsequent optimizations after those known to leave optimization opportunities. Some optimizations provide analysis and

```c
union U6 {
   short  f0;
   int f1;
};
union U6 g_13;
void func_1() {
  static const union U6 b[] =
    {{.f0 = 2}, {.f0 = 2}};
  g_13 = b[1];
}
```

(a) C code

```
func_1():
movl $0x0,0x2ef2(%rip) # <g_13>
mov  $0x2,%eax
mov  %ax,0x2ee6(%rip)  # <g_13>
retq
```

(b) Assembly

Fig. 8. A pessimization bug introducing a useless store of a union

conditions [46] for other optimizations and should not be used alone. These two causes provide a comprehensive explanation of pessimization bugs. Therefore, the fact that CLower successfully identifies bugs encompassing both these root causes indicates a certain level of diversity in CLower's ability.

Semantic Diversity. By analyzing the test code of the detected pessimization bugs, CLower demonstrates its ability to identify bugs triggered in various semantic structures. These include if condition structures, switch statements, usage of union structures, for loop structures, and function calls. The detection of bugs in loop structures and function calls is an important indicator of CLower's capability to handle complex redundant memory accesses. Examples of if condition structures and switch statements are shown in Figure 1 and Figure 2, respectively. CLower also detects bugs involving the usage of union structures(Figure 8), loop structures and function calls. These examples highlight the diverse semantic structures in which CLower successfully identifies pessimization bugs.

Takeaway. The evaluation demonstrates that CLower can discover bugs exhibiting diversity across multiple aspects, including various types of compiler components, different root causes, and a wide range of semantics.

3.3 RQ3: Accuracy

To evaluate the accuracy of the test oracle used by CLower, which also assesses how the introduction of extra global memory accesses by a compiler can indicate a pessimization bug, we conducted a false positive analysis on the bugs reported to GCC by CLower.

We conduct this false positive analysis based on (i) manual analysis of all potential bugs reported by CLower and (ii) discussions with the compiler communities. The manual analysis uses two criteria: (i) whether there is a performance impact, and (ii) whether the compiled code is worse.

The false positives identified in this manual analysis phase can be categorized into two situations. Additionally, we identified a third category based on feedback from compiler developers.

(I) Compiler Performance Bugs Due to Insufficient Optimization: These bugs are not pessimization bugs as the compiler only misses some optimization opportunities in the original code but does not produce worse binary code. In these cases, the additional memory access detected by CLower is introduced when the compiler attempts to optimize insufficient source code through certain transformations, but these optimizations are still insufficient to achieve tangible performance benefits. Although they are not classified as pessimization bugs, they still represent compiler

```
unsigned v1;
long long v2;

short func_1() {
    v2 = v1;
    return v2;
}
```

(a) C code

```
func():
mov    0x2f12(%rip),%eax # <v1>
mov    %rax,0x2f03(%rip) # <v2>
movzwl 0x2f04(%rip),%eax # <v1>
retq
```

(b) Assembly

Fig. 9. A missed optimization opportunity related to Copy Propagation

```
char g;
int v;
void foo(char p_17);
void func_1() { foo(g); }

void foo(char c) {
    v = (char)(2 - c) & c;
}
```

(a) C code

```
<func_1>
foo():  # inlined part
mov    $0x2,%eax
sub    0x2ef5(%rip),%al # <g>
and    0x2eef(%rip),%al # <g>
movsbl %al,%eax
mov    %eax,0x2ee2(%rip)# <v>
func_1():
retq
```

(b) Assembly

Fig. 10. A missed optimization opportunity related to Inlining

performance bugs (missed optimization opportunities). It is noteworthy that we have reported two bugs of this type, and all have been confirmed as compiler performance bugs.

- Missing Post-processing after Copy Propagation: This is a common optimization technique that involves replacing all occurrences of target variables in direct assignments with their corresponding values [2]. While this replacement may increase the number of accesses to the target variables, missing post-processing after this optimization can result in missed optimization opportunities when there are repeated loads. As illustrated in Figure 9, the accesses of variable ‘v2’ are replaced with those of ‘v1’. Although additional accesses are detected, this change does not worsen the code while still leaving opportunities for further optimizations.
- Missing Post-processing after Inlining: Inlining is a common optimization technique that involves replacing a function call with the actual body of the function at the call site. However, if post-processing is overlooked after this replacement, it may introduce memory access and cause missed optimization opportunities. The example in Figure 10 demonstrates how the accesses of parameter ‘c’ are replaced by direct accesses to ‘g’ within the body of ‘`func_1()`’. These direct accesses can be optimized by storing the value in a register.

(II) Usage of Paddings: These bugs are not optimization bugs, as the extra memory accesses detected by CLower do not require additional instructions; instead, they arise from the padding introduced by widening and combining memory accesses.

- Paddings due to Load Widening and Combination. Occasionally, compilers opt to widen a short load or combine multiple short loads to align with a larger size, typically the word size of the host CPU. As illustrated in Figure 11, gcc-13.2.0 demonstrates an interesting optimization pattern when evaluating the logical OR expression `s.f1 || s.f2`. The compiler generates a single 8-byte load operation that accesses the contiguous memory region starting at `s.f1`, encompassing not only the target fields (`s.f1` and `s.f2`) but intervening padding bytes. These cases serve as noteworthy findings that demonstrate CLower's capability to precisely track memory accesses at the byte level. Moreover, this compiler optimization pattern may introduce potential security concerns. The additional loaded padding bits could contain uninitialized data [23]. Particularly concerning are scenarios where sensitive data leaks occur through copying of these uninitialized memory regions to lower-privilege areas [25].

(III) Benign Performance Losses. For bug reports that meet our manual criteria, some may not indicate actual bugs. That is, performance loss due to redundant memory access can sometimes be benign. It is worth noting that performance loss reported by CLower is evaluated at the whole-compilation level, but not the optimization level. **Optimization-level performance loss is common and not considered a bug**. It does not confuse CLower as it should be resolved by follow-up optimizations within a whole compilation. CLower tests the entire compilation with in-product options (O1, O2, O3). When CLower detects such temporary performance losses persisting after a whole compilation, they mostly indicate a bug due to missing post-processing, as shown in Figure 9 and Figure 10. However, there are still benign performance loss at the whole compilation level. They are exceptions to our test oracle. Based on feedback from compiler developers on these reports (labeled 'Won't Fix'), we categorize these false positives into two types:

- Probabilistic Performance Loss for General Improvement: Some performance losses are expected and probabilistic, aiming for overall performance enhancement. CLower reported one case in this category involving a compiler feature of loop-invariant code motion. Here, the compiler moves side-effect-free code fragments outside the loop, even when it is unclear if the loop will execute. This can result in extra memory accesses if the loop is not executed.
- Performance Loss at Lower Optimization Levels: Certain performance losses are acceptable at lower optimization levels and can be resolved with higher optimization levels. CLower reported one instance where a redundant load observed at the O1 level would be addressed by the Partial Redundancy Elimination available at the O2 level. Although this issue is considered a compiler feature, it can be fixed by improving the existing O1 optimization implementation.

False Positive Statistics and Review on the Test Oracle. False positives occur because our oracle flags every compiler-introduced extra memory access, including cases that are not pessimizations. As shown in Table 2, after deduplication (without filtering), CLower reports 21 unique new GCC bugs, 16 of which have been confirmed as true pessimization bugs, resulting in a false positive rate of 23.8%. Of the 5 false positive cases, 2 were still confirmed as performance bugs, 1 was attributed to padding usage, 2 were benign performance losses. Overall, only 3 of the 22 reported cases are truly irrelevant, a rate we consider acceptable for manual triage. Regarding the accuracy of our test oracle, if we consider the introduction of extra global memory accesses as an indication of pessimization bugs, the accuracy is 76.2% (16/21). When considering it as an indicator of performance bugs, the accuracy increases to 85.7% (18/21). These results suggest that the test oracle is sufficiently accurate for practical use.

Takeaway. Through a false positive analysis, we determine that CLower exhibits sufficient accuracy to avoid irrelevant false positives. Consequently, we can conclude that the introduction of extra global memory accesses by a compiler typically signifies a performance issue, often manifesting as a pessimization bug.

```
struct S {
  long long f0;
  char  f1;
  int f2;
} s;

_Bool func() {
  return (s.f1 || s.f2);
}
```

(a) C code

```
func():
movabs $0xffffffff000000ff,%rax
and    0x2f17(%rip),%rax  # <s+0x8>
setne  %al
retq
```

(b) Assembly

Fig. 11. A Load-Combination example introduced by gcc

3.4 RQ4: Performance Impact and Prevalence

We evaluate the performance impact of CLower-detected bugs through two complementary analyses: their prevalence in real-world code and their static performance cost.

Direct dynamic measurement is infeasible for random-generation-based compiler testing, as it requires precise patches for each bug to establish a baseline, a requirement unmet by prior work [15, 36, 37].

Methodology for measuring real-world occurrences. Determining whether compilers introduce unnecessary memory accesses in real-world programs presents challenges, unlike the constrained random C programs generated by CLower. To address this, we focused on identifying pessimization bugs characterized by distinct binary features that can be summarized as simple code patterns with minimal instructions. Customized binary analysis was employed to detect these bugs effectively. For our analysis, we selected five well-known C language projects, namely SPEC CPU 2017 test suite, Linux Kernel-6.8, PostgreSQL-15.2, FFmpeg-6.0 and nginx-1.25.5 as our test subjects. These projects represent different types of performance-sensitive systems. We compiled them with the tested compiler i.e. gcc-13.2.0 and their default configurations. To ensure the accuracy of this evaluation, the chosen bugs should hold binary features that are significant enough to be only caused by our bugs. Specifically, we selected three pessimization bugs to evaluate their real-world impact:

- Conditional execution: This bug is characterized by multiple adjacent repeated comparisons followed by 'SBB' instruction. (Figure 1)
- Redundant sinking: This bug is characterized by adjacent meaningless moves preceded by a conditional jump targeting the second one. (Figure 12)
- Redundant zeroing: This bug is characterized by redundant set-zero before the effective assignment, usually happens in union assignment (Figure 8) or struct initialization. Besides, the zeroing and the assignment should correspond to the same line in the source code.

Result of real-world occurrences. The results, presented in Table 5, show that a total of 4,789 locations of the three tested pessimization bugs across SPEC CPU 2017, Linux Kernel-6.0, PostgreSQL-15.2, and FFmpeg-6.0. These findings underscore the significant presence of code segments in real-world programs that can trigger compiler pessimization bugs found by CLower.

Methodology for measuring performance degradation. Due to the inherent non-determinism in process scheduling [5] and the impracticality of creating isolated patches for each bug, dynamic performance measurement is infeasible. We instead adopt the static methodology of Gao et al. [15],

```
int g, *p, c;
void func() {
  *p = 5;
  g = 1;
  if (c)
    *p = 0;
  else
    ;
}
```

(a) C code

```
int g, *p, c;
void func() {
  *p = 5;
  g = 1;
  int tmp;
  if (c)
    tmp = 0;
  else
    tmp = *p;
  *p = tmp;
}
```

(b) transformed code

```
func():
mov   0x2ef1(%rip),%rax # <p>
xor   %edx,%edx
movl  $0x5,(%rax)
mov   0x2edb(%rip),%ecx # <c>
movl  $0x1,0x2ee1(%rip) # <g>
test  %ecx,%ecx
jne   401165 <func+0x25>
mov   (%rax),%edx
mov   %edx,(%rax)
retq
```

(c) Assembly

Fig. 12. A pessimization bug of GCC's redundant sinking

Table 5. The number of real world locations of chosen bugs found by CLower

	CPU2017	vmlinux	postgres	ffmpeg	nginx
Conditional execution	2,965	246	24	1,295	6
Redundant sinking	62	1	0	3	0
Redundant zeroing	38	148	7	0	8
Total	3,065	395	31	1,298	14

quantifying performance impact through two metrics: total instruction count increase and processor cycle estimates from LLVM's x64 scheduling model (llvm-mca). This approach provides deterministic measurements while avoiding runtime noise.

Results of measuring performance degradation. Our static analysis shows that, per minimal test case, the identified bugs introduce an average of 3.21 extra instructions and 0.72 additional processor cycles. While most redundant accesses degrade performance, we observed one case where an extra global access appeared beneficial by merging a local memory access. Nevertheless, this still constitutes a pessimization as it may violate compiler correctness by introducing unexpected memory operations. Overall, these results confirm that CLower-detected bugs impose measurable performance costs through redundant instruction execution.

Takeaway. By searching affected code samples in widely used real-world programs, we demonstrate that although derived from randomly generated code, at least some of the bugs detected by CLower can indeed impact real-world programs with considerable frequency. Furthermore, our evaluation shows that the bugs identified by CLower can affect performance by increasing the number of processor cycles. By focusing on these two aspects, we highlight CLower cause observable performance degradation on real-world programs.

3.5 RQ5: Security Implications in Concurrency Contexts

This research question examines whether CLower-detected bugs can induce **reproducible security violations** under concurrency. The security implications vary by bug type: **redundant**

store bugs directly violate compiler correctness and concurrency guarantees in multi-threaded contexts, while **redundant load bugs** only manifest security impact under specific race conditions. Since CLower generates single-threaded test cases, we must evaluate the latter under controlled concurrency to assess their full security implications.

Methodology. We focus on redundant load bugs, as their security impact is conditional. Our evaluation employs a two-phase approach: **(i) Controlled Race Injection:** We transform the original single-threaded test cases by introducing concurrent writes to the affected memory locations. This creates deterministic race conditions, though undefined behavior in C, that allow us to stress the compiler's handling of concurrent memory accesses. **(ii) Behavior Verification:** We recompile the modified programs and use CLower's dynamic checker to confirm that the extra load operations persist under these race conditions. This methodology is designed to expose compiler behavior to concurrent access patterns, focusing on vulnerability potential rather than runtime race probability. The results demonstrate whether these pessimizations can escalate benign data races into exploitable security vulnerabilities.

Results. Our evaluation confirms that all CLower-detected bugs manifest reproducible security impacts when exposed to concurrent execution with intentional race conditions. These findings corroborate cmmtest's [28] approach of identifying concurrency vulnerabilities through single-threaded memory access monitoring. The observed vulnerability pattern reflects real-world performance-critical scenarios, as demonstrated by incidents such as a Linux kernel memory corruption case [33] and another null pointer dereference case [20]. We emphasize that while our results verify this vulnerability pattern, comprehensive detection in real world programs would require additional race detection and program analysis techniques beyond our current scope.

Takeaway. CLower's bug detection extends beyond performance degradation, revealing two security impacts: (i) **concurrency violations**: five confirmed bug patterns directly break C concurrency model correctness in multi-threaded contexts, (ii) **race-condition vulnerabilities**: all remaining bug patterns demonstrate reproducible security-violating behaviors under controlled race conditions.

3.6 RQ6: Generalizability

While CLower currently detects compiler-introduced redundant accesses to global memory, the underlying optimization flaws may extend to other memory semantics. This research question evaluates whether the identified bugs generalize beyond global variables, particularly investigating their manifestation with **heap-allocated variables**. Determining this generalizability is essential for understanding the full scope of CLower's impact across different programming paradigms and memory usage patterns.

Methodology. To assess whether the detected bugs affect heap-allocated memory, we systematically migrated each bug-inducing global variable to an equivalent heap-based representation while preserving the original bug-triggering semantics. We employed two distinct migration strategies based on the variable's usage pattern:

- Directly accessed globals: For global variables referenced directly via their symbolic names within function bodies, we replaced them with heap objects accessed through global pointers, maintaining the same access patterns while changing the underlying allocation.
- Indirectly accessed globals: For global variables accessed via parameter pointers, we modified the calling context to pass pointers to newly allocated heap objects instead of the original global addresses.

Results. Our evaluation reveals that **75%** of the pessimization bugs detected by CLower can be reproduced when global variables are replaced with heap-allocated objects. This significant proportion suggests that the underlying compiler optimization flaws are not specific to global memory semantics but represent more fundamental issues in the compiler's optimization passes.

A plausible explanation for this generalizability is that these bugs involve additional dereferences to pointers that reference memory of unknown provenance from the compiler's perspective. Whether this memory resides in the global data section or the heap appears irrelevant to the erroneous optimization decisions.

Takeaway. The compiler pessimizations detected by CLower exhibit broader impact than initially anticipated: a clear majority (75%) manifest not only with global variables but also affect heap-allocated objects. This finding demonstrates that these optimization bugs transcend specific memory allocation schemes, representing generalized flaws in compiler optimization logic that affect multiple memory semantics.

4 A study of hidden bugs

During the regression testing of GCC and Clang using CLower, we observed the disappearance and reappearance of some pessimization bugs across different compiler versions, indicating a hidden nature of these bugs.

The Conflict. Through the root cause analysis of the hidden bugs, We identified a systematic conflict between compiler optimizations and pessimization bugs. Specifically, subsequent compiler optimizations can nullify the effects of a pessimization bug, hiding the unfixed bugs. However, the **underlying implementation defects** causing the pessimization bug may persist in the compiler, leading to its reappearance in later versions. This conflict complicates the detection of pessimization bugs: while pessimization bugs often become hidden due to subsequent optimization passes, attempting to expose them by disabling these clean-up optimizations would inevitably capture legitimate intermediate redundancies as false positives.

The Research Questions. To delve deeper, we formulated two research questions: RQ1: Are historical bugs not triggered in the latest compiler version hidden but not fixed? RQ2: Have the latest regression bugs been hidden before their detection?

Methodology. This study utilized a benchmark generated from our random code generation process, comprising 100,000 restricted load and 100,000 restricted store C code. We employed the rigorous enough test oracle (of CLower) to discern whether a pessimization bug is triggered for a test case. Note that this oracle filters legitimate intermediate redundancies by requiring all reported bugs to manifest under standard optimization levels (O1-O3). Different optimization scheduling was achieved using compilation options provided by the compilers, with `O2` serving as the base scheduling configuration. We tested various versions of GCC (from 8.x to 13.x) and Clang (from 11.x to 17.x). To address RQ1, our approach involved: (i) Running CLower on historical compiler versions to identify all test cases triggering potential pessimization bugs. (ii) Identifying test cases that did not trigger such bugs in the latest compiler version and tuning compilation options to determine if CLower could detect hidden bugs. For RQ2, we selected all bugs discovered in the latest compiler versions and adjusted compilation options on historical compilers to detect any pessimization occurrences.

Result. For RQ1, 1,100 of all 200,000 test cases triggered a pessimization on historical compilers but nothing on the latest compilers. By tuning the latest compiler's options, CLower detected pessimizations in 127 test cases. Regarding RQ2, of the 719 test cases triggering pessimizations in the latest compilers, 707 had never been triggered before. Upon tuning, we found that 32 of these cases had been hidden for a significant duration 2.01 years before resurfacing in the latest compilers. And the other 12 cases were present in older versions, disappeared, and then reappeared in the latest versions. In summary, our study revealed the prevalence of hidden bugs, some of which remain hidden for extended periods before resurfacing. This indicates the potential amount of unexposed bugs.

5 Discussion

5.1 Limitations

CLower has false negatives in detecting pessimization bugs due to design trade-offs in three main aspects. **(i) Global variables.** Bugs exclusive to local variables will be ignored. The reasons for this design choice are discussed in subsection 2.1. Additionally, since pointers to global and local (especially heap) variables are treated similarly by compilers, bugs found by CLower are sometimes common to other kinds of variables, e.g. 73.3% of the bugs found by CLower still work when replacing the involved global with heap variables. **(ii) Single location access.** Bugs where the introduced memory access to a global variable requires the variable to occur multiple times in the source code in the same way (either multiple loads or multiple stores) will be ignored. However, bugs related to high register pressure will not be ignored, as we keep multiple global variables and do not restrict the other kind of memory accesses (stores/loads). As explained in subsection 2.1, this choice allows for higher throughput and focuses on the most common problems, providing a more reliable test oracle. **(iii) Memory access.** Bugs that do not involve introducing extra memory accesses will be ignored. This is a shared limitation of many performance bug detection tools [9, 30, 34, 35, 38]. However, their findings highlight the prevalence of redundant memory access among compiler-introduced inefficiencies. For instance, CIDetector's evaluation demonstrates that concentrating on redundant loads and stores is highly effective for identifying various compiler-introduced inefficiencies. Similarly, LoadSpy further reveals that redundant loads are a significant indicator of diverse software inefficiencies, with its evaluation showing that dynamically addressing all redundant memory loads can achieve a 1.79x speedup for the entire program.

5.2 Developer feedback and Mitigations

The pessimization bugs found by CLower remain challenging to debug and fix. For all reported bugs, compiler developers have classified and initially debugged most of them. Although efforts have been made to fix them, only 7 out of 16 confirmed bugs have been successfully resolved at the time of writing. The average duration for fixing these bugs has been 418 days.

Although root causes are difficult to address, these bugs can be mitigated by **improving subsequent optimizations**. Many of the bugs we found were left as future work by developers, who anticipate that enhanced optimization techniques may eventually prevent these issues. For example, 66.7% of the bugs we found involved redundancies within fewer than four instructions. Peephole optimization, which inspects a small set of instructions and attempts to replace inefficient parts with better code, can be improved to be a mitigation.

The vulnerabilities arising from CLower-detected pessimizations in the correctness-security gap can often be addressed through source-level compiler barriers. The Linux kernel employs this approach via its `READ_ONCE` and `WRITE_ONCE` macros, which combine compiler barriers and `volatile` keywords to secure memory accesses in race-prone scenarios. However, comprehensive protection remains challenging due to the randomness of pessimization occurrence requires blanket protection of all potentially risky accesses. Missing protections frequently lead to vulnerabilities, as demonstrated by real-world cases including memory corruption [33] and null pointer dereferences [20]. While avoiding all race conditions prevents exploitation in concurrent contexts, unexpected memory accesses remain exploitable through other attack vectors [18, 42]. This underscores that compiler-side fixes constitute the fundamental solution for eliminating these security impacts at their source.

5.3 On the Nature of Pessimization Bugs: Implementation vs. Design

The vast majority of pessimizations uncovered by CLower are **implementation bugs**, errors that occur during the translation of a sound optimization design into compiler code. These are the random, localized mistakes that inevitably creep into complex systems. CLower excels at finding these precisely because they manifest as concrete, observable deviations (extra memory accesses) from the expected behavior of a correct design.

In contrast, bugs stemming from a flawed **optimization design** itself are rare in our findings. When reported, developers often classify these cases as 'Won't Fix' (see Section 3.3), as they represent conscious—though potentially performance-limiting—design choices rather than correctable implementation errors. This distinction explains the high confirmation rate of our reports and underscores that CLower primarily targets the prevalent class of implementation-level defects.

This focus is where CLower's greatest value lies. While design flaws can be reasoned about and corrected systematically, implementation bugs are scattered and unpredictable. Automating the detection of these implementation-level pessimizations addresses a problem that is both severe and otherwise intractable through manual inspection alone.

5.4 Long-standing bugs

The bugs detected by CLower are not version-specific anomalies; many are long-standing issues that have persisted undetected across multiple compiler versions. Our analysis reveals that the confirmed bugs uncovered by CLower had gone undetected for an average of over 3,000 days prior to our discovery. For instance, the five CLower-detected compiler concurrency bugs summarized in Table 3 can be reproduced in versions as early as GCC 4.0.4, 4.4.7, 4.7.3, and 5.0, indicating these critical defects had remained latent in production compilers for over a decade.

5.5 Post-finding Analysis

This process includes the reduction of test cases and bug deduplication. It is the bottleneck of CLower in confirming more bugs. CLower can produce hundreds of problematic test programs within an hour, but most of them are related to known bugs.

The reduction of test cases is a known time-consuming task in compiler testing [10]. CLower leverages C-Reduce [32] to shrink the code from an average of 1,000 lines to an average of 50 lines. On average, CLower takes 333 seconds per test case for reduction, which is significantly faster than the method by Theodoridis et al. [36], based on differential testing, which takes 4-8 hours. CLower is more efficient because the reduction process involves repeatedly checking that the bug persists, and CLower is designed to easily verify this by simply running the compiled binary.

The bug deduplication part is heavily labor-intensive [11]. We hand-coded source/binary code patterns of found bugs to match potential new bugs. This semi-automated approach helps save human labor, but more automated methods are heavily needed in this direction.

6 Related Work

Compiler correctness testing. Compiler correctness testing has been widely studied [10], with test generation being a key component. Csmith [45], YARPgen [24] and fuzz4all [41] have been proposed to generate valid and diverse test programs. However, CLower targets pessimization bugs rather than correctness bugs, and thus extends existing generators to enforce restricted memory accesses. Given the different targets and technical foundations, CLower is not directly comparable to these tools.

For bug-specific approaches, Eide et al. [14] introduced a valuable differential testing method for volatile variable bugs, comparing access summaries across optimization levels. While this approach provides an effective foundation for detecting optimization issues, it faces challenges when extended to non-volatile accesses: compilers may legally eliminate non-volatile accesses, causing false negatives when source-level redundancy removal hides compiler-introduced redundancy. In contrast, our restricted generation strategy explicitly controls source-level redundancy, enabling a reliable test oracle.

Compiler concurrency correctness testing. Cmmtest [28] identified four GCC concurrency bugs by comparing source-level and compiled memory traces, detecting eliminations, introductions, and reordering of memory operations. While sharing detection capability for introduced global stores, CLower demonstrates superior effectiveness (five confirmed cases) through its specialized test generation methodology. By producing programs with constrained memory access patterns, CLower minimizes false negatives caused by compiler optimizations eliminating redundant accesses. Complementary to these approaches, C4 [40] employed fuzzing to uncover atomic-operation violations, discovering two new and two historical GCC bugs.

Performance testing. Existing approaches to performance analysis fall into two categories: program monitoring and compiler-specific testing. Tools like DeadSpy [9] and LoadSpy [34] detect inefficiencies in application code but cannot identify compiler-induced issues. For compiler-specific testing, current methods employ distinct strategies: **(i) benchmarking:** Benchmarks like SPEC 2017 [7] and PolyBench [21] require significant expert labor to confirm compiler bugs, posing scalability issues. **(ii) assembly feature comparison**: Barany et al. [4] employ cross-compiler assembly feature comparison to detect deviations from predefined optimization patterns. However, this binary-level differential analysis cannot reliably distinguish pessimizations. **(iii) optimization differential testing**: While such approaches like Mod [46] and Theodoridis et al.'s [36] have advanced missed optimization detection, their cross-compiler comparison methodology cannot detect bugs that manifest uniformly across all tested compilers. Mod's reliance on manually-curated optimization pairs introduces scalability bottlenecks through required compilation-level instrumentation. Dead's exclusive focus on Dead Code Elimination restricts its detection capability to basic block-level optimizations, leaving instruction-level pessimizations undetected. **(iv) metamorphic testing**: Recent work applies metamorphic testing to detect pessimizations by comparing compiler output across refined/degraded source variants. ProgramMarkers [37] injects optimization hints (e.g., `__builtin_unreachable`), while Gao et al. [15] applies systematic code degradation. However, both approaches suffer from inherent semantic limitations: ProgramMarkers requires specific syntactic markers, and Gao et al.'s method depends on pre-defined transformation rules. These constraints artificially restrict the input space, preventing detection of memory-access-related pessimizations that fall outside their transformation patterns. **(v) memory operation testing**: CIDetector [35] employs dynamic instrumentation to identify redundant memory operations, including potential pessimizations. However, this approach relies on high-quality benchmarks and requires extensive manual analysis to filter source-level redundancies.

Correctness-security gap. The correctness-security gap, first formalized by D'Silva et al. [13], describes a critical phenomenon where compiler optimizations preserve functional correctness while violating source-level security properties. Xu et al. [43] later demonstrated the severe security consequences of this gap across multiple domains. CLower advances this line of research by introducing the first automated approach to detect **compiler defects contributing to this gap**, thereby expanding both its theoretical understanding and practical implications.

Secure Compilation and Translation Validation. CLower shares the overarching goal of secure compilation, ensuring that compilation preserves security properties—but addresses a distinct and complementary problem. Foundational work in formal verification [1, 6] provides rigorous, design-level guarantees for compiler passes. These methods could, in principle, be extended to reason about security violations arising from redundant memory accesses. However, their application faces several practical challenges in the context of our target bugs. First, the primary source of the pessimizations we uncover is **implementation bugs** in complex, real-world compilers like GCC and LLVM, rather than flaws in the high-level design of optimization passes. Formal methods are less suited to finding such scattered, low-level implementation errors that occur during the translation of a sound design into code. Second, scaling these formal frameworks to verify the entirety of the C language and highly optimizing compilers (such as GCC) remains a formidable, open research challenge. In contrast, CLower offers a practical, automated testing approach that scales to these complex systems.

Compared to translation validation methods, the security property CLower checks, the absence of extra memory accesses, is a quantitative, low-level property. This differs from the high-level, trace-based hyper-properties (e.g., on I/O behavior) often targeted by secure translation validation methods [8], which are not designed to detect such fine-grained instruction-level inefficiencies and their security side effects.

Study of Hidden Bugs. Yann et al. [17] studied hidden bugs in FPGA synthesis tools and revealed similar patterns of bug recurrence. The parallel is relevant given the shared optimization challenges between compilers and FPGA toolchains.

7 Conclusion

This paper introduces a novel approach, CLower, for detecting compiler pessimization bugs through redundant memory accesses. Our work specifically targets security-critical pessimizations involving memory operations, which can introduce deterministic side effects that expand the attack surface in real-world code. A key challenge in this domain is disentangling compiler-introduced redundancies from those already present in the source code. To address this, CLower enforces a predetermined number of memory accesses per global variable during test generation, establishing a deterministic baseline that unambiguously attributes any extra access to the compiler. We apply CLower to test GCC and LLVM compilers, uncovering a total of 16 confirmed pessimization bugs, with 7 of them fixed at the write time. Evaluation results demonstrate that CLower exhibits sufficient accuracy and can target a diverse set of pessimization bugs with both performance and security impacts. Additionally, we identify a systematic conflict between pessimization bugs and compiler optimizations, which serves as a significant factor contributing to hidden pessimization bugs. This paper represents the first step towards studying and addressing compiler pessimization bugs related to memory accesses and lays the groundwork for future research in this area.

Acknowledgments

We thank the anonymous reviewers for their feedback. This work was supported, in part, by grants from the Chinese National Natural Science Foundation (62402107, 62572231, 62172201, 62032010), SNSF PCEGP2_186974, ERC CodeSan 850868, NSF awards CNS-2045478 and CNS-2247434, the Start-up Research Fund of Southeast University and the Big Data Computing Center of Southeast University. Any opinions, findings, conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the sponsors.

8 Data-Availability Statement

Our archived artifact on Zenodo[44] contains all the necessary code, tools, and supplementary resources to replicate the experiments and results presented in our paper.

References

[1] Carmine Abate, Roberto Blanco, Deepak Garg, Catalin Hritcu, Marco Patrignani, and Jérémy Thibault. 2019. Journey beyond full abstraction: Exploring robust property preservation for secure compilation. In *2019 IEEE 32nd Computer Security Foundations Symposium (CSF)*. IEEE, 256–25615.

[2] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. 2006. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., USA.

[3] Amir H Ashouri, William Killian, John Cavazos, Gianluca Palermo, and Cristina Silvano. 2018. A survey on compiler autotuning using machine learning. *ACM Computing Surveys (CSUR)* 51, 5 (2018), 1–42.

[4] Gergö Barany. 2018. Finding missed compiler optimizations by differential testing. In *Proceedings of the 27th International Conference on Compiler Construction* (Vienna, Austria) *(CC 2018)*. Association for Computing Machinery, New York, NY, USA, 82–92. doi:10.1145/3178372.3179521

[5] Edd Barrett, Carl Friedrich Bolz-Tereick, Rebecca Killick, Sarah Mount, and Laurence Tratt. 2017. Virtual machine warmup blows hot and cold. *Proceedings of the ACM on Programming Languages* 1, OOPSLA (2017), 1–27.

[6] Gilles Barthe, Benjamin Grégoire, and Vincent Laporte. 2018. Secure compilation of side-channel countermeasures: the case of cryptographic “constant-time”. In *2018 IEEE 31st Computer Security Foundations Symposium (CSF)*. IEEE, 328–343.

[7] James Bucek, Klaus-Dieter Lange, and Jóakim v. Kistowski. 2018. SPEC CPU2017: Next-generation compute benchmark. In *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*. 41–42.

[8] Matteo Busi, Pierpaolo Degano, and Letterio Galletta. 2019. Translation validation for security properties. *arXiv preprint arXiv:1901.05082* (2019).

[9] Milind Chabbi and John Mellor-Crummey. 2012. DeadSpy: a tool to pinpoint program inefficiencies. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization* (San Jose, California) *(CGO ’12)*. Association for Computing Machinery, New York, NY, USA, 124–134. doi:10.1145/2259016.2259033

[10] 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.

[11] Yang Chen, Alex Groce, Chaoqiang Zhang, Weng-Keen Wong, Xiaoli Fern, Eric Eide, and John Regehr. 2013. Taming compiler fuzzers. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Seattle, Washington, USA) *(PLDI ’13)*. Association for Computing Machinery, New York, NY, USA, 197–208. doi:10.1145/2491956.2462173

[12] Giuseppe Antonio Di Luna, Davide Italiano, Luca Massarelli, Sebastian Österlund, Cristiano Giuffrida, and Leonardo Querzoni. 2021. Who’s debugging the debuggers? exposing debug information bugs in optimized binaries. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 1034–1045.

[13] Vijay D’Silva, Mathias Payer, and Dawn Song. 2015. The Correctness-Security Gap in Compiler Optimization. In *2015 IEEE Security and Privacy Workshops*. 73–87. doi:10.1109/SPW.2015.33

[14] Eric Eide and John Regehr. 2008. Volatiles are miscompiled, and what to do about it. In *Proceedings of the 8th ACM international conference on Embedded software*. 255–264.

[15] 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*. 1846–1857.

[16] Zilin Guan. 2024. fgraph: Add READ_ONCE() when accessing fgraph_array[]. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=d65474033740ded0a4fe9a097fce72328655b41d.

[17] Yann Herklotz and John Wickerson. 2020. Finding and understanding bugs in FPGA synthesis tools. In *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 277–287.

[18] Intel. 2018. Bounds Check Bypass / CVE-2017-5753 / INTEL-SA-00088. https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/advisory-guidance/bounds-check-bypass.html.

[19] Sesha Kalyur and GS Nagaraja. 2016. A survey of modeling techniques used in compiler design and implementation. In *2016 International Conference on Computation System and Information Technology for Sustainable Solutions (CSITSS)*. IEEE, 355–358.

[20] Naresh Kamboju. 2024. function_graph: Add READ_ONCE() when accessing fgraph_array[]. https://github.com/torvalds/linux/commit/63a8dfb889112ab4a065aa60a9a1b590b410d055.

[21] Jeyhun Karimov, Tilmann Rabl, and Volker Markl. 2018. PolyBench: The First Benchmark for Polystores. In *Performance Evaluation and Benchmarking for the Era of Artificial Intelligence - 10th TPC Technology Conference, TPCTC 2018, Rio de Janeiro, Brazil, August 27-31, 2018, Revised Selected Papers (Lecture Notes in Computer Science, Vol. 11135)*, Raghunath Nambiar and Meikel Poess (Eds.). Springer, 24–41. doi:10.1007/978-3-030-11404-6_3

[22] Vu Le, Mehrdad Afshari, and Zhendong Su. 2014. Compiler validation via equivalence modulo inputs. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Edinburgh, United Kingdom) *(PLDI '14)*. Association for Computing Machinery, New York, NY, USA, 216–226. doi:10.1145/2594291.2594334

[23] Juneyoung Lee, Yoonseung Kim, Youngju Song, Chung-Kil Hur, Sanjoy Das, David Majnemer, John Regehr, and Nuno P Lopes. 2017. Taming undefined behavior in LLVM. *ACM SIGPLAN Notices* 52, 6 (2017), 633–647.

[24] Vsevolod Livinskii, Dmitry Babokin, and John Regehr. 2020. Random testing for C and C++ compilers with YARPGen. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 196 (11 2020), 25 pages. doi:10.1145/3428264

[25] Kangjie Lu, Chengyu Song, Taesoo Kim, and Wenke Lee. 2016. UniSan: Proactive Kernel Memory Initialization to Eliminate Data Leakages. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (Vienna, Austria) *(CCS '16)*. Association for Computing Machinery, New York, NY, USA, 920–932. doi:10.1145/2976749.2978366

[26] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation* (Chicago, IL, USA) *(PLDI '05)*. Association for Computing Machinery, New York, NY, USA, 190–200. doi:10.1145/1065010.1065034

[27] Robin Morisset, Pankaj Pawan, and Francesco Zappa Nardelli. 2013. cmmtest: Hunting Concurrency Compiler Bugs. https://fzn.fr/projects/cmmtest/gcc-bugs.html.

[28] Robin Morisset, Pankaj Pawan, and Francesco Zappa Nardelli. 2013. Compiler Testing via a Theory of Sound Optimisations in the C11/C++11 Memory Model. *ACM SIGPLAN Notices* 48, 6 (June 2013), 187–196. doi:10.1145/2499370.2491967

[29] Tipp Moseley, Dirk Grunwald, and Ramesh Peri. 2009. Optiscope: Performance accountability for optimizing compilers. In *2009 International Symposium on Code Generation and Optimization*. IEEE, 254–264.

[30] Adrian Nistor, Linhai Song, Darko Marinov, and Shan Lu. 2013. Toddler: detecting performance problems via similar memory-access patterns. In *Proceedings of the 2013 International Conference on Software Engineering* (San Francisco, CA, USA) *(ICSE '13)*. IEEE Press, 562–571.

[31] John Regehr, Yang Chen, Pascal Cuoq, Eric Eide, Chucky Ellison, and Xuejun Yang. 2012. Test-case reduction for C compiler bugs. In *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation*. 335–346.

[32] John Regehr, Yang Chen, Pascal Cuoq, Eric Eide, Chucky Ellison, and Xuejun Yang. 2012. Test-case reduction for C compiler bugs. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation* (Beijing, China) *(PLDI '12)*. Association for Computing Machinery, New York, NY, USA, 335–346. doi:10.1145/2254064.2254104

[33] Jan Stancek. 2013. mm: prevent mmap_cache race in find_vma(). https://github.com/torvalds/linux/commit/b6a9b7f6b1.

[34] Pengfei Su, Shasha Wen, Hailong Yang, Milind Chabbi, and Xu Liu. 2019. Redundant loads: a software inefficiency indicator. In *Proceedings of the 41st International Conference on Software Engineering* (Montreal, Quebec, Canada) *(ICSE '19)*. IEEE Press, 982–993. doi:10.1109/ICSE.2019.00103

[35] Jialiang Tan, Shuyin Jiao, Milind Chabbi, and Xu Liu. 2020. What every scientific programmer should know about compiler optimizations?. In *Proceedings of the 34th ACM International Conference on Supercomputing* (Barcelona, Spain) *(ICS '20)*. Association for Computing Machinery, New York, NY, USA, Article 42, 12 pages. doi:10.1145/3392717.3392754

[36] Theodoros Theodoridis, Manuel Rigger, and Zhendong Su. 2022. Finding missed optimizations through the lens of dead code elimination. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS '22)*. Association for Computing Machinery, New York, NY, USA, 697–709. doi:10.1145/3503222.3507764

[37] Theodoros Theodoridis and Zhendong Su. 2024. Refined Input, Degraded Output: The Counterintuitive World of Compiler Behavior. *Proceedings of the ACM on Programming Languages* 8, PLDI (June 2024), 671–691. doi:10.1145/3656404

[38] Shasha Wen, Milind Chabbi, and Xu Liu. 2017. REDSPY: Exploring Value Locality in Software. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems* (Xi'an, China) *(ASPLOS '17)*. Association for Computing Machinery, New York, NY, USA, 47–61. doi:10.1145/3037697.3037729

[39] Felix Wilhelm. 2020. Xen XSA 155: Double fetches in paravirtualized devices. https://insinuator.net/2015/12/xen-xsa-155-double-fetches-in-paravirtualized-devices/.

[40] Matt Windsor, Alastair F Donaldson, and John Wickerson. 2021. C4: the C compiler concurrency checker. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 670–673.

[41] Chunqiu Steven Xia, Matteo Paltenghi, Jia Le Tian, Michael Pradel, and Lingming Zhang. 2024. Fuzz4all: Universal fuzzing with large language models. *Proc. IEEE/ACM ICSE* (2024).

[42] Jianhao Xu, Luca Di Bartolomeo, Flavio Toffalini, Bing Mao, and Mathias Payer. 2023. WarpAttack: Bypassing CFI through Compiler-Introduced Double-Fetches. In *2023 IEEE Symposium on Security and Privacy*. IEEE.

[43] 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.

[44] Jianhao Xu, Kunbo Zhang, Mathias Payer, Kangjie Lu, and Bing Mao. 2026. *Artifact for OOPSLA'26 paper: "CLower: Detecting Compiler Pessimization Bugs through Redundant Memory Accesses"*. doi:10.5281/zenodo.18503229 https://doi.org/10.5281/zenodo.18503229.

[45] 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* (San Jose, California, USA) *(PLDI '11)*. Association for Computing Machinery, New York, NY, USA, 283–294. doi:10.1145/1993498.1993532

[46] Yi Zhang. 2023. Detection of Optimizations Missed by the Compiler. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (San Francisco, CA, USA) *(ESEC/FSE 2023)*. Association for Computing Machinery, New York, NY, USA, 2192–2194. doi:10.1145/3611643.3617846

Received 2025-10-10; accepted 2026-02-17