# WhiteFox: White-Box Compiler Fuzzing Empowered by Large Language Models

CHENYUAN YANG, University of Illinois at Urbana-Champaign, USA
YINLIN DENG, University of Illinois at Urbana-Champaign, USA
RUNYU LU, Huazhong University of Science and Technology, China
JIAYI YAO, Chinese University of Hong Kong, China
JIAWEI LIU, University of Illinois at Urbana-Champaign, USA
REYHANEH JABBARVAND, University of Illinois at Urbana-Champaign, USA
LINGMING ZHANG, University of Illinois at Urbana-Champaign, USA

Compiler correctness is crucial, as miscompilation can falsify program behaviors, leading to serious consequences over the software supply chain. In the literature, fuzzing has been extensively studied to uncover compiler defects. However, compiler fuzzing remains challenging: Existing arts focus on black- and grey-box fuzzing, which generates test programs without sufficient understanding of internal compiler behaviors. As such, they often fail to construct test programs to exercise intricate optimizations. Meanwhile, traditional white-box techniques, such as symbolic execution, are computationally inapplicable to the giant codebase of compiler systems. Recent advances demonstrate that Large Language Models (LLMs) excel in code generation/understanding tasks and even have achieved state-of-the-art performance in black-box fuzzing. Nonetheless, guiding LLMs with compiler source-code information remains a missing piece of research in compiler testing.

To this end, we propose WhiteFox, the first white-box compiler fuzzer using LLMs with source-code information to test compiler optimization, with a spotlight on detecting deep logic bugs in the emerging deep learning (DL) compilers. WhiteFox adopts a multi-agent framework: *(i)* an LLM-based analysis agent examines the low-level optimization source code and produces requirements on the high-level test programs that can trigger the optimization; *(ii)* an LLM-based generation agent produces test programs based on the summarized requirements. Additionally, optimization-triggering tests are also used as feedback to further enhance the test generation prompt on the fly. Our evaluation on the three most popular DL compilers (*i.e.,* PyTorch Inductor, TensorFlow-XLA, and TensorFlow Lite) shows that WhiteFox can generate high-quality test programs to exercise deep optimizations requiring intricate conditions, practicing up to 8 times more optimizations than state-of-the-art fuzzers. To date, WhiteFox has found in total 101 bugs for the compilers under test, with 92 confirmed as previously unknown and 70 already fixed. Notably, WhiteFox has been recently acknowledged by the PyTorch team, and is in the process of being incorporated into its development workflow. Finally, beyond DL compilers, WhiteFox can also be adapted for compilers in different domains, such as LLVM, where WhiteFox has already found multiple bugs.

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

Additional Key Words and Phrases: White-box Testing, Fuzzing, Large Language Models, Code Analysis

Authors' Contact Information: Chenyuan Yang, University of Illinois at Urbana-Champaign, Champaign, USA, cy54@illinois.edu; Yinlin Deng, University of Illinois at Urbana-Champaign, Champaign, USA, yinlind2@illinois.edu; Runyu Lu, Huazhong University of Science and Technology, Wuhan, China, lry89757@gmail.com; Jiayi Yao, Chinese University of Hong Kong, Shenzhen, China, jiayiyao@link.cuhk.edu.cn; Jiawei Liu, University of Illinois at Urbana-Champaign, Champaign, USA, jiawei6@illinois.edu; Reyhaneh Jabbarvand, University of Illinois at Urbana-Champaign, Champaign, USA, reyhaneh@illinois.edu; Lingming Zhang, University of Illinois at Urbana-Champaign, Champaign, USA, lingming@illinois.edu.


ACM 2475-1421/2024/10-ART296
https://doi.org/10.1145/3689736

**ACM Reference Format:**
Chenyuan Yang, Yinlin Deng, Runyu Lu, Jiayi Yao, Jiawei Liu, Reyhaneh Jabbarvand, and Lingming Zhang. 2024. WhiteFox: White-Box Compiler Fuzzing Empowered by Large Language Models. *Proc. ACM Program. Lang.* 8, OOPSLA2, Article 296 (October 2024), 27 pages. https://doi.org/10.1145/3689736

## 1 Introduction

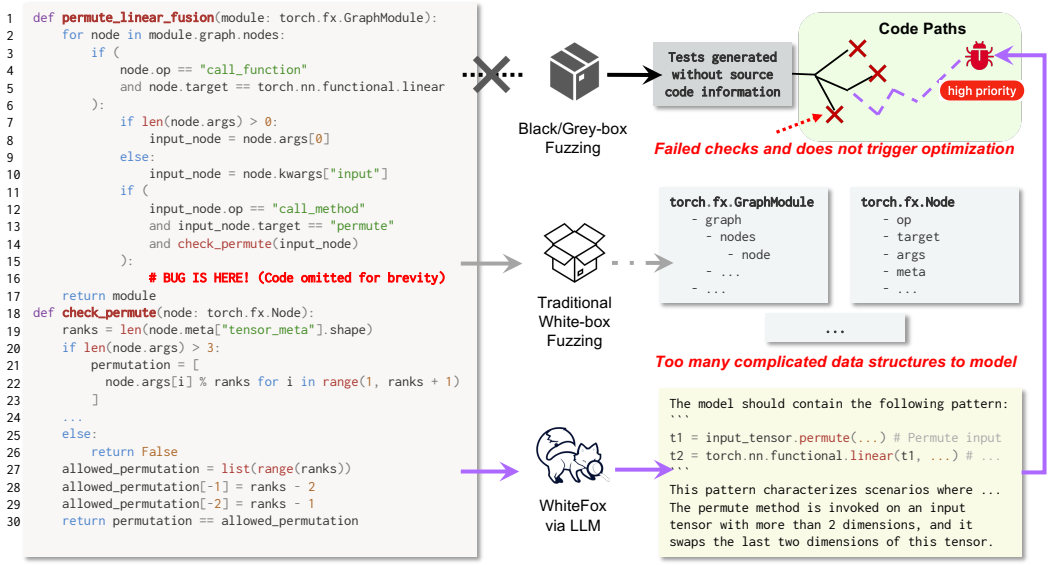



Fig. 1. Motivating example.

Modern compilers [9, 24, 38, 63, 69, 75] play a critical role in translating high-level programming languages into efficient machine code. However, incorrect or misapplied optimizations can lead to subtle and hard-to-detect bugs, and even vulnerabilities [15, 86, 89]. For instance, compiler misoptimizations have led to severe security vulnerabilities, *e.g.*, system hanging, memory errors, and information leaks, in the Linux kernel [86] and safety-critical deep learning (DL) applications/systems [1, 15, 60, 68]. Given the ubiquity of compilers in software development, it is vital to ensure the correctness of compiler optimizations. To this end, fuzzing (or fuzz testing) [73, 92] has been applied to automatically generate a large number of test inputs, aiming to explore compiler defects [71]. To date, a large body of fuzzing tools has been tailored for different languages and compilers [21, 45, 50, 88], highlighting their success by finding a large number of real-world compiler bugs.

In the literature, researchers have proposed various fuzzing techniques to incorporate the knowledge about the system under test (SUT) during test generation [11, 21]. They are generally classified into three categories according to the extent of SUT knowledge visible to the fuzzer: black-box [45, 50, 88], grey-box [11, 21, 44], and white-box fuzzing [36, 66]. Black-box fuzzing has zero knowledge about the internal workings of the SUT and simply considers system input/interface information. Consequently, the inputs are generated without complying with the intended structures or behaviors of the SUT. In contrast, white-box fuzzing techniques, by inspecting the source code of the SUT, aim to synthesize test cases to exhaustively explore all possible code paths. Grey-box fuzzing lies between black- and white-box fuzzing. By leveraging limited program information of

the SUT (*e.g.*, code coverage), grey-box fuzzers attempt to efficiently produce tests that are more likely to exercise new program behaviors. While in theory, we can apply all such approaches for compiler fuzzing, each approach grapples with its challenges and limitations owing to the immense complexity and scale of modern compiler systems. For instance, the widely-used LLVM [38] compiler is implemented with 14M lines of code, and the popular DL compiler, TensorFlow [75], has 3.5M lines.

**Challenges.** Black-box fuzzing, without knowledge of the internal workings, struggles due to the intricate conditions required to trigger optimizations. Simply generating random inputs, without any guidance, often proves impractical for reaching the deep corners of optimization logic. For example, a recent study [11] shows that the black-box fuzzer for C compilers, Csmith [88], which produces test-cases through grammar-based generation, is significantly less effective than coverage-guided fuzzing. Grey-box fuzzing, though better informed by source code instrumentation to achieve higher code coverage than its black-box counterpart, frequently falls short of fully understanding the nuanced criteria required to trigger particular optimizations. This shortcoming stems from the fact that compiler optimizations typically hinge on meeting precise and strict conditions. Vanilla coverage-driven strategies might not navigate these specific states effectively. Moreover, grey-box compiler fuzzing [11, 21] even fail to generate semantically correct inputs, leading to the discovery of mostly front-end crash bugs. On the other hand, traditional white-box fuzzing, which relies on strict analysis of the SUT source code, becomes daunting with modern compilers. The sheer complexity of modern optimization techniques, combined with the vast landscape of programming paradigms and hardware targets, makes modeling all behaviors an uphill task. For instance, symbolic execution [36] executes a program by using symbolic variables in place of concrete values, enabling the systematic exploration of every potential execution path. However, when applied to compiler systems, it becomes infeasible to designate every variable as symbolic. Even if such a feat were achievable, the million-line scale of compiler codebase inevitably leads to *path explosion*, rendering the approach highly challenging.

Furthermore, traditional compiler fuzzing techniques are typically tailored to specific languages/compilers. Yet, designing and implementing a fuzzing framework for a new compiler is both time-intensive and laborious. For instance, Csmith [88] is comprised of tens of thousands of lines of code through years of development. Given the unique characteristics of each target language/compiler, reusing the efforts of one fuzzing implementation for a different input language/compiler often presents significant challenges.

**Motivation.** Figure 1 presents a motivating example of the optimization `permute_linear_fusion` in PyTorch Inductor [63]. This optimization fuses the `permute` and `linear` operators when the *permute* method is invoked on an input tensor with more than two dimensions, specifically swapping the last two dimensions. On the left side of the figure, we see its source code implementation. Here, the constraints required to trigger this optimization are explicitly detailed with nested `if` condition statements and a helper function `check_permute`. However, when applying fuzzing techniques to test this optimization, black/grey-box fuzzing struggles to generate models that align the `permute` and `linear` operators with these constraints. For instance, consider a scenario where a grey-box compiler fuzzer produces a model with the `linear` operator and covers the first `if-check` (Line 3-6, Figure 1) in this optimization. Even if a black/grey-box fuzzer repeatedly selects this test as a seed for mutation, it is challenging to successfully mutate the model to invoke the `permute` method on a tensor—specifically, to swap its last two dimensions—where the output should then serve as the input for `linear`. This is because both black-box and grey-box fuzzing are unaware that the models should include the `permute` and `linear` operators in this specific way, due to the absence of guidance from the source code implementation. As there are thousands of operators in PyTorch, such fuzzers will likely choose a different operator than `permute` or apply `permute` in many other ways. As a

result, the generated models often fail validation checks and cannot activate this optimization, let alone discover deep bugs in it (Line 16, Figure 1). On the other hand, though the white-box techniques have the potential to trigger this optimization theoretically, it is *impractical* to apply traditional program analysis to extract constraints from the detailed source code due to the data structure complexity in compilers, *e.g.,*`torch.fx.GraphModule` and `torch.fx.Node`. These structures are further composed of several other intricate classes with diverse attributes (*e.g.,* `args`, `shape`, and `rank`). Additionally, the intricate constraints are often expressed in hierarchical conditions (*e.g.,* nested `if` statements) and even complex check functions. Therefore, it is extremely challenging, if not impossible, to accurately extract and express these constraints symbolically for constraint solvers, let alone apply any formal method to solve them.

**Insight.** Can we scale white-box fuzzing to fully test optimizations for any compilers? We address this question based on the insight that Large Language Models (LLMs) [8, 22, 42, 58, 59, 90] are pre-trained on a vast array of code spanning various programming languages. This broad foundation enables them to excel in comprehending and generating code across diverse languages [6].Therefore, for the `permute_linear_fusion` optimization, different from typical white-box fuzzing, we can leverage LLMs to summarize the requirements for the models that could trigger it based on the source code information, as highlighted in the yellow text box of Figure 1. Subsequently, we can leverage the generated requirement description to further prompt LLMs to create the corresponding inputs, which are PyTorch models in this case. In our experiments, the generated tests indeed triggered the `permute_linear_fusion` optimization and even detected a previously unknown bug in it! Notably, this bug was confirmed by developers and labeled as *high-priority*.

**Proposal.** We present WhiteFox, the first *white-box* compiler fuzzing approach via Large Language Models (LLMs) to fully test the core optimization modules in DL compilers, which represent the fastest-evolving segment in the field of compilers. As discussed, existing approaches to white-box testing cannot scale to model the behavioral information of complex compiler systems. Therefore, the key idea of WhiteFox is to leverage LLMs to automatically infer the requirements of test programs that can trigger the compiler optimizations based on their source code implementation. LLMs, having been pre-trained on an extensive corpus comprising natural languages and a variety of programming languages, possess the ability to comprehend and succinctly summarize optimization source code. The input to WhiteFox is the source code that implements compiler optimizations. First, an LLM-based analysis agent automatically analyzes and summarizes the testing requirements for triggering optimizations. Subsequently, an LLM-based generation agent produces numerous meaningful test programs guided by the generated requirements. To generate test programs that can directly exercise corresponding optimizations, WhiteFox further employs a feedback-loop mechanism that uses optimization-triggering tests as few-shot examples to guide future test generation.

**Summary.** This work makes the following contributions:

- **Novelty.** We introduce a new dimension of white-box compiler fuzzing by using LLMs as both optimization source code analyzers and test input generators. To our best knowledge, this work is the first to demonstrate that LLMs can transform the low-level implementation information into the corresponding high-level test programs, making it practical to employ the white-box fuzzing techniques to test complex DL compilers. Furthermore, beyond DL compiler testing, WhiteFox can also be adapted for white-box fuzzing of other compilers, and even complex, real-world software systems in general, inspiring future work in this promising direction.
- **Approach.** While our approach is general and applicable to various compiler systems, we implement WhiteFox as a practical fuzzer for the three most popular DL compilers, PyTorch Inductor, TensorFlow-XLA, and TensorFlow Lite. We utilize GPT4 as an analysis agent to summarize the

requirements based on the source code, and StarCoder as a generation agent to create diverse test inputs. Our artifact is available at https://github.com/ise-uiuc/WhiteFox.
- **Study.** We extensively compared WhiteFox with state-of-the-art fuzzers on the target DL compilers. Our result shows that WhiteFox can practice 2.5x more optimizations than the baselines. By now, WhiteFox detects *101* bugs, with *92* already confirmed as previously unknown and *70 already fixed*. Of these, 10 bugs are labeled as *high-priority* by the developers.
- **Impact.** WhiteFox has been acknowledged by the official PyTorch team and is currently being integrated into their development pipeline, demonstrating the real-world applicability of our approach. Furthermore, our generality study adapts WhiteFox for testing LLVM and reveals multiple bugs in this popular C/C++ compiler, showing the broader impact of our work beyond DL compilers.

## 2 Background and Related Work

Fuzzing has been extensively studied for testing both traditional compilers and the emerging deep learning (DL) compilers. In this section, we first review related work for compiler fuzzing without LLMs. We then discuss the recent advances of LLM-based testing.

### 2.1 Compiler Fuzzing

The main challenge to compiler fuzzing is to synthesize and diversify syntactically and semantically valid programs as the input to the compiler. In the literature of traditional compiler fuzzing, *grammar-based* techniques aim to synthesize syntactically valid random programs via generation rules that comply with the language grammar. Such techniques have been widely used for fuzzing compilers of programming languages including C/C++ (*e.g.,* Csmith [88] and YARPGen [50]), JavaScript (*e.g.,* LangFuzz [30] and jsfunfuzz [65]), and Python (*e.g.,* PyRTFuzz [43]). However, grammar-based approaches often require massive engineering efforts to implement rules that ensure the validity of the generated programs [88] and may still fail to synthesize realistic yet complicated test programs. Therefore, *mutation-based* techniques [19, 39, 40, 94] propose to mutate valid seed programs for generating new input programs that can exercise deeper code paths in the compiler. Besides traditional compiler fuzzing, various fuzzing techniques have also been proposed for the emerging DL compilers. DL compilers compile DL models, whose generation requires valid compositions of tensor operations. In early work, DL models are either directly curated from existing open-source models [61] or created using simple shape-preserving operators [28, 82]. To diversify operators in model generation, recent work explicitly define constraints for operator constructions either manually [27, 45] or automatically [46].

Though comprehensive, most of the work discussed above is still black-box fuzzers. Consequently, their generated test programs often fail to practice the internal nuances of intricate compiler behaviors, especially in the important optimization passes. Therefore, recent grey-box compiler fuzzers [11, 21, 47] have been proposed to integrate code coverage guidance to discover interesting input programs that can explore deeper compiler behaviors. PolyGlot [11] proposes to integrate coverage feedback into its fuzzing loop to test compilers at the intermediate representation (IR) level.Tzer [47] proposes a coverage-directed fuzzing approach that jointly mutates both IR files and optimization passes, specifically targeting the TVM DL compiler [9]. More recently, GrayC [21] combines coverage feedback with mutation operators to test C compilers.

To achieve more explicit test generation, white-box testers by theory can tailor test programs to specifically exercise certain optimization by inspecting the compiler logic from the source code. Yet, to our best knowledge, there is hardly any competitive white-box compiler fuzzer in existence, largely due to the unmanageable scale of compiler systems which makes detailed program analysis

almost impossible. While recent hybrid fuzzing techniques [12, 13, 34, 35, 49, 91] integrate general-purpose fuzzing [26, 30, 52] with concolic execution [25, 67], showing promise in different testing domains, their effectiveness in compiler fuzzing remains limited compared to compiler-specific fuzzers [7, 11, 39, 47]. Besides the path explosion issue due to the compiler-scale complexity, another key limitation is that hybrid fuzzing typically operates on binary inputs, overlooking the nuanced semantics present in source code, which is crucial for compiler testing.

### 2.2 LLM-Based Testing

With the recent advancements in Large Language Models (LLMs), there has been a surge in efforts to leverage LLMs for unit test generation [93]. For example, TeCo [56], built upon a fine-tuned CodeT5 [90] model, has been introduced to aid developers in completing unit tests for the given code and context. TestPilot [64] explores adaptive zero-shot test generation with Codex [8] for testing JavaScript. MuTAP [16] leverages zero-shot and few-shot learning to generate test cases using Codex and llama-2-chat, and further evaluates the generated test cases through mutation testing. ChatUniTest [10] performs conversation-driven test generation with ChatGPT [58]. CodaMosa [41] employs the capability of LLMs to help with the search-based software testing (SBST) [23] by generating unit tests for uncovered methods. ChatGPT-SBST [74] performs a comprehensive study on using ChatGPT [58] for SBST.

While LLM-based unit test generation techniques incorporate source code, they are specifically designed for particular modules (*e.g.,* functions/classes) of the projects under test. In contrast, our approach targets the important problem of compiler testing at the entire system level (a totally different problem from unit testing), and relies solely on the source code implementation of optimizations as *guidance* for fuzzing. Also, unit test generation demands detailed and comprehensive information about the module being tested, including all associated data structures, while our guidance can be imprecise, partial, or even incomplete. In addition, we are the first to leverage LLMs to bridge the gap between the low-level optimization implementation and the high-level input programs for compilers. Lastly, LLM-based unit test generation can only assist developers and requires further manual refinement, because generating reliable unit-test oracles remains notably difficult or infeasible for even the current best LLMs [14, 59]. In contrast, WhiteFox directly leverages various automated oracles for traditional compiler fuzzing, *e.g.,* differential testing [54].

Beyond unit test generation, applying LLMs to fuzz software systems is another emerging trend [17, 18, 72, 84]. For example, TitanFuzz [17] demonstrates for the first time that modern LLMs can be directly leveraged to perform both grammar- and mutation-based fuzzing of real-world systems. Very recently, Fuzz4All [84] has also demonstrated that LLMs can serve as the universal fuzzers for various types of software systems. However, these approaches often neglect white-box information, specifically the source code implementation of the software under test. Consequently, they tend to concentrate on the compiler's front-end rather than the intricate logic within the middle and back-end components. In contrast, WhiteFox is the first white-box compiler fuzzing approach, which leverages low-level optimization implementations for guided fuzzing, and has been demonstrated to outperform the recent TitanFuzz (§ 6.1).

## 3 Design

Figure 2 depicts the overview of WhiteFox, consisting of three main components: *Requirement Summarization*, *Test Generation*, and *Feedback Loop*. Overall, WhiteFox takes the *source code* of optimization passes from the tested compiler as inputs and generates high-quality test programs via LLMs. To that end, during the *Requirement Summarization* phase, an *analysis LLM* is used to summarize the requirements of the test programs to trigger the optimization by examining its source-code implementation (§ 3.1). Next, the analyzed requirements are used to prompt a *generation*

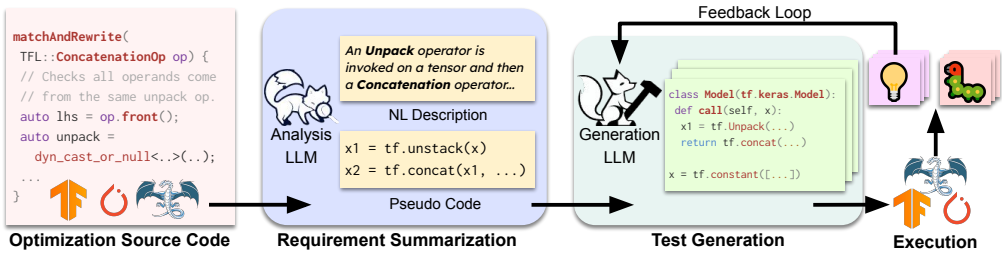



Fig. 2. Overview of WhiteFox.

*LLM*, which synthesizes test programs to specifically practice corresponding optimizations (§ 3.2). The generated test programs are then compiled and executed, and WhiteFox observes whether they can indeed activate the corresponding optimizations via instrumentation. If a test program triggers any optimization, it will be incorporated into WhiteFox's feedback loop as an example to guide the *generation LLM* towards more optimization-targeted generation in subsequent iterations (§ 3.3). To detect compiler bugs, every test program is executed with test oracles including result inconsistency, as well as compile- and execution-time crash (§ 3.4).

Notably, WhiteFox employs a multi-agent framework: *(i)* an LLM-based analysis agent is prompted to infer the conditions for triggering optimizations by inspecting the implementation code; and *(ii)* an LLM-based generation agent is prompted to create a large number of meaningful test programs. This design allows us to balance the trade-off between the costs and benefits that different LLMs provide. For example, we can let the analysis LLM be one with broad knowledge and reasoning ability (in both natural language and code) and let the generation LLM be one that is specialized for efficient program generation.

### 3.1 Requirement Summarization

While it is possible to directly generate tests from the optimization source code with LLMs, it does not perform well (as shown in our evaluation § 6.2.1). This is because the optimization source code has the following two undesirable features: *(i)* The implementation source code for the optimization is often lengthy. It often contains **redundant** information like detailed comments explaining the code behavior. Furthermore, it may contain code that is **unrelated** to the characteristics of triggering input, such as implementation details (e.g., using data structures like `std::<vector>` to gather operands), the IR transformation after the optimization is triggered, and error logging. Such long source code not only makes it time-consuming for LLMs to process but also adds to the difficulty for LLMs to understand, as LLMs are known to struggle with long context [48]. Additionally, some source code may exceed the limited context window available to LLMs. *(ii)* The implementation code is written at a **low level**, involving large quantities of domain-specific modules, IRs, and helper functions. For example, from Figure 2, we can observe that the optimization source code relies on TensorFlow Lite-specific module `TFL`, which contains various low-level operators (*e.g.*, `TFL::ConcatenationOp` and `TFL::UnpackOp`) and functions (*e.g.*, `dyn_cast_or_null`). These IRs are translated from the input programs written at a higher level. They are meant for internal use within a compiler, and none of them are visible at the high level.

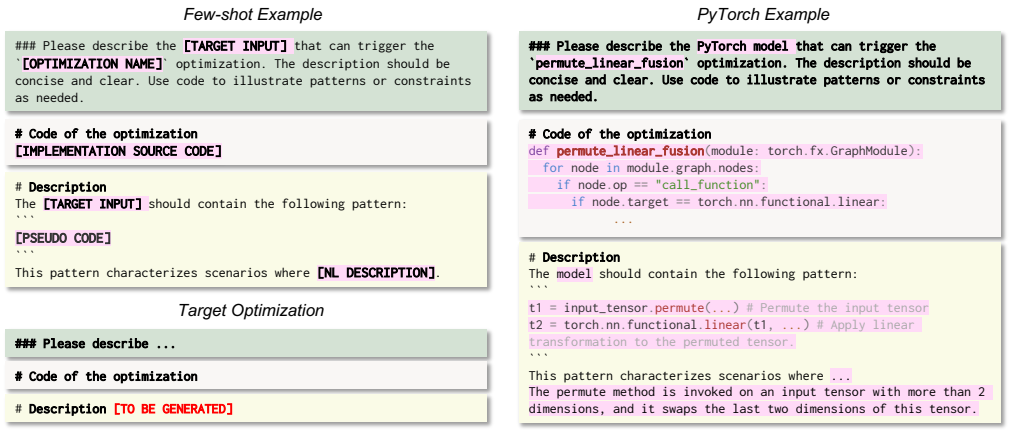
To address the above challenges, we propose to leverage an analysis LLM to provide **a clear, concise, and comprehensible summary** of how the optimization can be triggered, which will assist in subsequent test generation. We considered two types of summary formats: **natural language** and **pseudo-code**. Natural language description can be more concise than the original optimization source code, as it only summarizes the key aspects of requirements and ignores

redundant or unrelated information. Furthermore, it is much more comprehensible to LLMs, since LLMs have been trained on enormous natural language data. On the other hand, pseudo-code can concisely describe certain patterns (especially when there are long sequences of function calls) and is closer to the structure of tests that we want to generate ultimately. Additionally, we map the low-level implementation to a high-level summarization using natural language or pseudo code resembling user code. For instance, `TFL::ConcatenationOp` is a low-level TensorFlow Lite IR used in the implementation source code of TensorFlow Lite optimizations (in Figure 2), which corresponds to the high-level TensorFlow public API `tf.concat`, and semantically means joining data from multiple input tensors. In our natural language description/pseudo code, instead of using the low-level `TFL::ConcatenationOp`, we directly use "concatenation operator"/`tf.concat` to summarize.

While both natural language and pseudo-code can help shorten the context and remove low-level information to avoid confusing the LLM, each has its own unique strengths. For example, the `permute_linear_fusion` optimization from PyTorch Inductor requires that *"the tensor method permute is invoked first, and then the torch.nn.functional.linear function is invoked on the permuted tensor"*. Such a NL description is not as straightforward as the pseudo-code format for this case. Specifically, *"the tensor method permute"* could be simply represented as `input_tensor.permute(...)` in pseudo-code. However, for the description *"it swaps the last two dimensions of this tensor"*, it is pretty challenging to leverage pseudo-code to describe it clearly and briefly. To combine and maximize their strengths, we adopt a hybrid format that blends NL and pseudo-code to describe the requirements for triggering optimization, rather than relying solely on either format. This mixed format provides the analysis LLM with greater flexibility to utilize NL and/or pseudo-code as needed for each component of requirements, resulting in a more expressive and higher-quality summarization. Our experimental findings (§ 6.2.1) also support that the mix of NL and pseudo-code achieves the best performance in this task.

Despite all these aforementioned benefits of a high-quality optimization summary, it is worth noting that the requirement summarization is a very comprehensive and challenging task, even for domain experts. First, it requires understanding the logic of the implementation code and rephrasing it with semantic-preserving natural language or pseudo-code. Second, the mapping from low-level implementation code to high-level information necessitates a broad background knowledge of the corresponding programming language and compiler. This ranges from understanding general technical terminology (*e.g.,* "variable arguments" and "tail calls"), to in-depth domain-specific knowledge (*e.g.,* from low-level LLVM IRs to high-level C++ grammar). As demonstrated in our evaluation (§ 6.2.1), the most powerful LLM to date (namely GPT4) is capable of performing this challenging analysis process (while the current open-source models, such as StarCoder, still lag far behind). This capability stems from its extensive training on vast datasets, enabling it to gain a broad knowledge and implicit understanding of various programming languages and systems. Additionally, its proficiency in performing these domain-specific analyses in our work likely results from its training on the source code of these open-source compiler systems.

Therefore, WhiteFox first leverages the analysis LLM to infer the requirement of high-level inputs that could trigger the optimization, utilizing its implementation written in the low-level source code. More specifically, for each optimization, we use few-shot in-context learning [5] to prompt the analysis LLM to generate its trigger requirements for the inputs in the mixed format of NL and pseudo-code. Figure 3(a) presents the *general* few-shot prompt template used to summarize requirements for optimizations in target compilers. This prompt template starts with the instruction *"Please describe the* `[TARGET INPUT]` *that can trigger the* `[OPTIMIZATION NAME]` *optimization..."*, where `[TARGET INPUT]` is the input format specific to the target compiler. Then it is followed by the source code of the optimization implementation and concludes with the description of requirements that




(a) Prompt template for requirement summarization.

(b) Requirement summarization prompt in PyTorch Inductor.

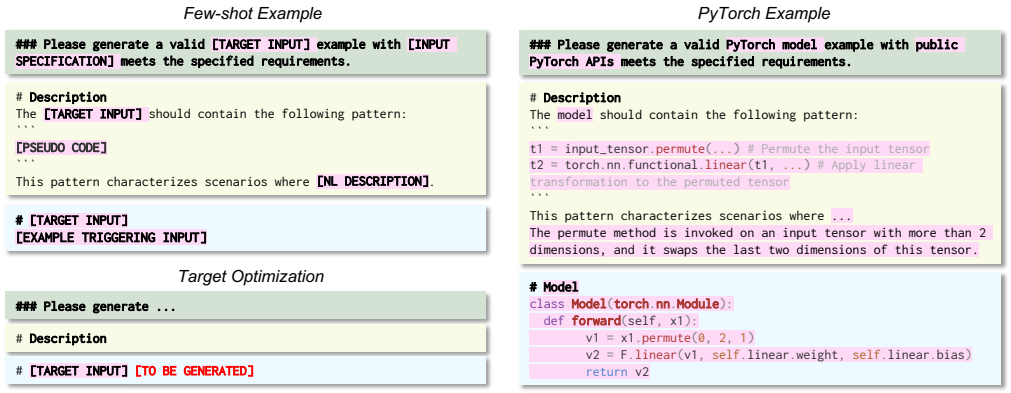
Fig. 3. Requirement summarization prompt in WhiteFox.

the input should fulfill. Note that the description is in the mixed format of NL and pseudo-code, consisting of `[PSEUDO CODE]` and `[NL DESCRIPTION]`. The target optimization has the same structure as the few-shot examples, but its requirements field is left empty, awaiting generation by the LLM. Take the prompt of requirement summarization in PyTorch Inductor as an example, whose few-shot prompt is shown in Figure 3(b). The expected input format for PyTorch Inductor is a PyTorch model; therefore, `[TARGET INPUT]` is populated with "PyTorch model". Following this, the source code for the example optimization, `permute_linear_fusion`, is provided. Finally, example descriptions are given in the mix of pseudo-code and NL formats to outline the constraints necessary to trigger the example optimization. Note that such few-shot examples can guide the analysis LLM to generate desired output formats. For instance, the example description (emphasized with a yellow box in Figure 3(b)) provides the LLM with a clearer illustration of expected formats. Furthermore, they facilitate the learning process of analysis LLM by providing the example mappings from low-level optimization implementation to high-level input program requirements.

More formally, let $\mathcal{P}_A$ be the analysis LLM that models the probability of token sequences. It takes the following types of information as inputs: *(i)* $I_i$, the instruction on summarizing an optimization $O_i$; *(ii)* $C_i$, the source code implementation of $O_i$; *(iii)* $R_i$, the summarized trigger pattern or requirement of the optimization $O_i$. Let $E_K$ be the few-shot prompt prefix consisting of $K$ example optimizations: $E_K = (I_1, C_1, R_1) \circ (I_2, C_2, R_2) \circ \ldots \circ (I_K, C_K, R_K)$. Let $O_t$ be the target optimization. The probability distribution of the generated requirement $R_t$ can be defined as $\mathcal{P}_A(R_t \mid E_K, I_t, C_t)$.

## 3.2 Test Generation

By utilizing the requirement description of the optimization, WhiteFox employs the power of LLM to generate test inputs that can effectively trigger the corresponding optimization for bug detection. Similar to requirement summarization, we leverage few-shot in-context learning to generate the test inputs specific to each optimization based on the requirements. Figure 4(a) shows the *general* prompt template used in WhiteFox for test generation. In the template, the structure of few-shot examples comprise an instruction, specifically: *"Please generate a valid* `[TARGET INPUT]` *example*




(a) Prompt template for test generation (b) Test generation prompt in PyTorch Inductor.

Fig. 4. Test generation prompt in WhiteFox.

*with* `[INPUT SPECIFICATION]` *meets the specified requirements."* Then, it details the requirements for optimization activation and concludes with an illustrative input that practices the *example* optimization. Following the few-shot examples, the target optimization has a similar structure while the test input to it is *to be generated* by the LLM. Figure 4(b) shows the test generation prompt used in PyTorch Inductor. The format of test input to the PyTorch Inductor is a PyTorch model (`[TARGET INPUT]`) with public PyTorch APIs (`[INPUT SPECIFICATION]`). Next, we specify the requirements for the test inputs to activate the optimization `permute_linear_fusion`. This is complemented by an illustrative model that can trigger this optimization.The provided few-shot examples aid the LLM in generating the test in the desired format, such as a `torch.nn.Module` composing public PyTorch APIs. Furthermore, the example could help the LLM learn the relationship between the requirement description of the optimization and the corresponding test input that can trigger it. In Figure 4(b), the model input highlighted in a blue box (*# Model*) contains the code `x1.permute(0, 2, 1)`. This corresponds to the requirement: *"The permute method is invoked on an input tensor with more than 2 dimensions, and it swaps the last two dimensions of this tensor"*. Such examples elucidate for the LLM: *(i)* the meaning of "permute method is invoked on an input tensor"—which should not be `torch.permute(x1, ...)`, and *(ii)* the implication of "swaps the last two dimensions of this tensor"-which is `permute(0, 2, 1)` for the tensor with three dimensions.

Again, let $\mathcal{P}_G$ be the generation LLM that models the probability of token sequences. Recall that $I$ represents the summarization instruction and $R$ denotes the requirement for triggering an optimization. Let $E_K$ be the few-shot prompt prefix consisting of $K$ example optimizations: $E_K = (I_1, R_1, T_1) \circ (I_2, R_2, T_2) \circ \ldots \circ (I_K, R_K, T_K)$, where $T_i$ is a valid test input that triggers the optimization $O_i$. Let $O_t$ be the target optimization. The probability distribution of the generated test $T_t$ can be defined as $\mathcal{P}_G(T_t \mid E_K, I_t, R_t)$.

### 3.3 Feedback Loop

The aim of utilizing the optimization implementation to guide the LLM in test generation (§ 3.2) is to produce diverse tests that can effectively exercise the target optimization, with the ultimate aim of uncovering potential bugs within it. While the prompt in Figure 4 contains example optimizations from the same compiler under test (and their trigger tests), these few-shot examples lack specific guidance for the target optimization.

**Algorithm 1:** Example Selection Algorithm Inspired by Thompson Sampling

```
Function Select(TriggerTests, N):
    for t ∈ TriggerTests do
        Sample θ_t ~ Beta(α_t, β_t)
    ExampleTests ← argmax_t(θ, N)
    return ExampleTests
Function Update(ExampleTests, NewTriggerTests, NumTrigger, NumNotTrigger):
    for t_example ∈ ExampleTests do
        α_example ← α_example + NumTrigger
        β_example ← β_example + NumNotTrigger
    α_avg ← avg({α_example | t ∈ ExampleTests})
    β_avg ← avg({β_example | t ∈ ExampleTests})
    for t_new ∈ NewTriggerTests do
        α_new ← α_avg
        β_new ← β_avg
    TriggerTests ← TriggerTests ∪ NewTriggerTests
```

The motivation behind incorporating a feedback loop is that the generated tests themselves are valuable feedback from the SUT, and can serve as guidance for fuzzing a target optimization in future iterations. In each iteration, when newly generated tests are discovered for the target optimization, we collect them as candidates for few-shot examples for future test generation prompts. By incorporating such successful triggering tests into the prompt, we enhance a targeted guidance to the generation LLM, enabling it to produce more inputs that trigger the target optimization.

*Target Optimization*

```
### Please generate different valid [TARGET INPUT] example with
[INPUT SPECIFICATION] meets the specified requirements.

# Description
The [TARGET INPUT] should contain the following pattern:
```
[PSEUDO CODE]
```
This pattern characterizes scenarios where [NL DESCRIPTION].

# [TARGET INPUT]
[EXAMPLE TRIGGERING INPUT #1]

# [TARGET INPUT]
[EXAMPLE TRIGGERING INPUT #2]

# [TARGET INPUT]
[EXAMPLE TRIGGERING INPUT #3]

# [TARGET INPUT] [TO BE GENERATED]
```

Fig. 5. Prompting for test generation with feedback.

Based on the inspiration, WhiteFox incorporates tests that have successfully triggered the corresponding optimization as supplementary examples during iterative test generation. The target compiler is instrumented to record triggered optimizations for each test input. If the test input successfully practices the corresponding optimization, it will be incorporated as a candidate example in the subsequent iteration of test generation for that specific optimization. More specially, in each iteration, if there are triggering inputs for the current optimization, WhiteFox selects several triggering inputs as examples (defaulting to 3). These example triggering inputs will be plugged into the prompt shown in Figure 5, along with the instruction and requirement summary of the target optimization (same instruction and summary as the previous prompt), and will be used to generate the next batch of test inputs. This feedback design aids the LLM in generating tests that have a higher likelihood of triggering the targeted pattern, which is demonstrated in our ablation study (§ 6.2.2). In the case where there are no triggering inputs for a particular optimization,

WhiteFox continues to use the initial prompt (Figure 4) in subsequent iterations until it finds an input capable of triggering that optimization.

To further investigate this, we observe that not all triggering examples are equally effective in guiding the LLM to generate new valuable triggering tests. One useful signal for assessing their effectiveness is the triggering rate of the newly generated tests when we employ them as few-shot examples. To effectively select triggering examples with an evolving knowledge of example effectiveness, it is crucial to find a balance between exploration and exploitation. Exploration is critical because it not only allows us to evaluate under-explored options but also helps to produce a diverse set of tests for fuzzing. On the other hand, some level of exploitation is desirable, as it enables us to fully harness the potential of effective examples. To address this, WhiteFox adopts an (adapted) Multi-Armed Bandit (MAB) algorithm, Thompson Sampling [78], as the selection strategy for triggering examples to balance the exploiting and exploration trade-off. Each triggering example is conceptualized as an arm in the MAB framework. The main assumption here is that each triggering example is associated with a probability representing the triggering rate, which quantifies the proportion of generated tests capable of triggering the optimization when utilizing the given example. During the fuzzing loop, our objective is to estimate the probability associated with each triggering example, with the aim of using the most effective example to achieve more valuable triggering tests. More specifically, following the classical Thompson Sampling algorithm [78], when we do not have any prior information about an arm, we choose standard beta distribution [53] $B(1,1)$ (or equivalently Uniform$(0,1)$) for its prior distribution. The beta distribution is parameterized by two shape parameters $\alpha > 0, \beta > 0$, which represents the number of successes and failures in historical trials. The probability density function of beta distribution can be formally written as follows:

$$f_{\alpha,\beta}(x) = \frac{1}{B(\alpha,\beta)} x^{\alpha-1}(1-x)^{\beta-1}$$

where $B(\alpha,\beta)$ is a constant for normalization. After drawing a new sample and observing the reward (in our case, 1 if a generated test successfully triggers the targeted optimization and 0 otherwise), the posterior probability can be conveniently updated by increasing $\alpha$ or $\beta$ by one, depending on whether the sample was a success or failure. More formally, if our prior belief about $X$ is represented by $f_{\alpha,\beta}$, the posterior distribution of $X$ will be updated to $f_{\alpha+1,\beta}$ or $f_{\alpha,\beta+1}$ after we observe $X = 1$ or $X = 0$.

Algorithm 1 shows the example test selection process. Firstly, we sample $\theta_t$ from each of the posterior distributions (Line 2-3). Subsequently, we opt for top-$N$ arms with the highest sampled value, which are the chosen example tests (`ExampleTests`) in this iteration (Line 4). Unlike conventional scenarios where a single arm is chosen, we simultaneously select multiple test examples at a time to construct a single few-shot prompt for generating a batch of new tests. Consequently, when we observe the number of triggering tests among all newly generated tests, we use this information to update the posterior of each of `ExampleTests` (Line 7-9). We next initialize the new trigger tests (`NewTriggerTests`) using the mean values of $\alpha$ and $\beta$ of the distribution of the `ExampleTests` (Line 10-14), to reduce the overhead of re-exploring the distribution of `NewTriggerTests` from scratch. This comes from our assumption that the effectiveness of the new test is highly correlated with those few-shot examples, as the new test is generated by the LLM based on those specific examples, potentially inheriting valuable code patterns from such "parent" examples [5]. In the end, we update the pool of existing trigger tests with newly found tests (Line 15).

### 3.4 Test Oracle

Bugs are manifested in the following symptoms under WhiteFox.

**Result inconsistency.** During compilation, programs are iteratively transformed to semantically equivalent yet more efficient code through an array of optimization passes. However, miscompilation can silently happen due to logical defects in the pass implementation, leading to undesired semantic inconsistency in the produced machine code. Such semantic inconsistency can be manifested through differential testing, as is commonly used in prior compiler testing work [39, 40, 45, 47, 50]. Specifically, for each test program that is both compilable and executable, given the same set of inputs to the program (if required), we cross-check the produced outputs over the optimized and non-optimized (or minimally optimized) versions of the program.

**Crash.** Following prior work [11, 17, 21, 40, 45, 47, 81, 88], it is undesirable to let the compiler and the compiled executable crash unexpectedly. Consequently, WhiteFox actively captures crashing signals at both the compile- and execution-time for the test program, including process aborts, segmentation faults, and unexpected internal exceptions (*e.g.,* `INTERNAL_ASSERT_FAILED` in PyTorch).

To summarize, we compile each test input in two modes: with and without optimization. We consider the following conditions as bug candidates:

- Crashes during either optimization compilation or optimized program execution.
- Discrepancies in compilation status (pass/fail) between the two modes.
- Different program outputs between the two modes.

## 4 Implementation

**Optimization collection.** We start to gather optimization-specific compiler source code by specifying the relevant directories. For example, the source code of optimization passes for PyTorch Inductor is managed under the `torch/_inductor` directory, and that for TensorFlow-XLA is mainly placed under `tensorflow/compiler/xla/service`. We next identify code fragments (*e.g.,* functions) that perform optimizations through simple keyword pattern matching. For instance, operator fusion is an important optimization in DL compilers and we collect the relevant functions by searching "fusion" or "fuse". In addition, auxiliary functions invoked by the main optimizations are also collected since they may unveil essential conditions for activating the optimizations. Curating and identifying optimization-relevant code fragments is required to drive WhiteFox; however, it shall be easy for compiler developers who have a deep understanding of the code being maintained.

**Instrumentation.** To gather the triggering information for the feedback loop (§ 3.3), we instrument each collected optimization function by inserting a logging statement at the function entry. As such, when compiling a test program, from the logs a sequence of activated optimization passes can be obtained.

**Analysis and generation LLMs.** While our approach is general and thus agnostic to the LLMs being employed, our tool WhiteFox is built on the state-of-the-art GPT4 [59] and StarCoder [42]. Specifically, we utilize GPT4 [59] as the analysis LLM for its recognized excellence in code comprehension and proficiency in natural language processing tasks [6]. For each optimization, we let the analysis LLM generate one requirement description with the temperature set to zero via the OpenAI APIs. Meanwhile, we choose StarCoder [42] (StarCoder-15B) to be the generative LLM, which is an open-source model with 15.5B parameters and a context length of 8K. In each iteration, we let StarCoder generate a batch of ten test inputs with the temperature set to one through the HuggingFace APIs [32]. The model choices allow us to balance the trade-off between the costs and benefits that different LLMs provide: *(i)* GPT4 is a powerful unified LLM (*i.e.,* with broad knowledge and reasoning ability over both natural language and code) but costly, making it suitable as the analysis LLM where its use is a one-time effort; *(ii)* StarCoder is an affordable LLM specialized for code and is thus suitable for efficient continuous test generation.

**Few-shot prompting.** For the requirement summarization and initial test generation few-shot prompt specific to each target compiler, we opt for one-shot prompting, for minimal manual efforts involved in prompt construction and affordable LLM context size. To accomplish this, we select an optimization from each target compiler. Subsequently, we manually write the requirement description and a demonstrative input test capable of triggering the optimization. This serves as the one-shot example in the prompts for both requirement summarization and test generation. One exception is that PyTorch Inductor has two distinct types of optimizations (7 utilizes a conventional optimization check function, and 54 involves a pattern matcher).Therefore, we separately design two prompts for each type and choose the corresponding prompt for each optimization based on its type. For the feedback prompt, the requirement is produced by the analysis LLM, and the sample tests are created by the generation LLM. We use three-shot as our default setting in the feedback prompt. Overall, constructing prompts for each target compiler is relatively straightforward. We only included a single example per compiler to illustrate the task format, requiring minimal effort. It is even easier for compiler developers who are familiar with optimization logic. Notably, such examples might already exist in test suites for many compilers. For comparison, many traditional compiler fuzzing techniques even require numerous example tests as seeds [39, 40, 94].

## 5 Evaluation

### 5.1 Research Questions

We investigate the following research questions in our experiments:

- **RQ1:** How does WhiteFox compare to state-of-the-art DL compiler fuzzers?
- **RQ2:** Are all the key components in WhiteFox effective?
- **RQ3:** Is WhiteFox able to detect real-world bugs?

### 5.2 Experimental Setup

**Compilers under test.** Our main targets are the three most popular DL compilers: PyTorch Inductor [63], TensorFlow Lite [76] and TensorFlow-XLA [77], within PyTorch [62] and TensorFlow [75], two of the most widely used DL frameworks. With our best effort, we collect all possible optimizations from these compilers. For TensorFlow-XLA, whose optimization implementations tend to be lengthy, we only choose the optimizations that consist of less than 400 lines due to the limit of LLM context window size. Table 1 lists the overview of the tested DL compilers.

Table 1. Details of target DL compilers.

| | # Optim. | Source lang. | Test lang. | Nightly ver. |
|---|---|---|---|---|
| PyTorch Inductor | 61 | Python | Python | 20230509 |
| TensorFlow Lite | 13 | C++ | Python | 20230507 |
| TensorFlow-XLA | 49 | C++ | Python | 20230507 |

**Baselines.** We compare WhiteFox with state-of-the-art DL system fuzzers, including LLM-based TitanFuzz [17] and symbolic rule-based NNSmith [45]. Since most optimizations are triggered by a sequence of operators, we do not include the comparison with API-level DL library fuzzers [83, 85], which are not intended for testing optimizations. We evaluate each baseline tool using its default configuration, *e.g.,* a 4-hour input generation time for NNSmith. We retain the default settings for all compared baselines because they were selected by the original authors as the optimal parameters for achieving high performance while minimizing saturation.

Notably, there are no practical white-box fuzzers specifically targeting DL compilers to the best of our knowledge. For general-purpose directed or hybrid fuzzing approaches (*e.g.,* QSYM [91] and Pangolin [31]), since they are far less effective for large-scale/complicated compiler systems compared to compiler-specific techniques [7, 11, 21], we exclude them from our baselines. More importantly, to the best of our knowledge, there is no directed or hybrid fuzzing approach for DL compilers. One possible reason could be the inherent complexity of DL compilers, making it prohibitively difficult to craft such tools. Specifically, DL compilers are often developed in diverse programming languages (*e.g.,* C++, Python, and CUDA) and rely heavily on various backend libraries (*e.g.,* Triton [79], oneDNN [57], and MKL-DNN [55]). Additionally, generating arbitrary inputs for DL compilers is extremely difficult for general-purpose fuzzers due to dual requirements: satisfying language syntax/semantics (e.g., Python's dynamic typing and syntax checks) and tensor/operator constraints for valid computational graphs [17, 45]. As a result, we opt to compare with the current best techniques for fuzzing DL compilers, *i.e.,* TitanFuzz [17] and NNSmith [45].

**Ablation variants.** Multiple WhiteFox variants are evaluated in our ablation study. Considering that PyTorch Inductor boasts the most optimizations, we conduct our ablation study exclusively to PyTorch Inductor. For requirement generation, we consider four variants: **WF-Mix** (the default setting of WhiteFox), **WF-NL**, **WF-Code**, and **WF-Impl**. For each optimization, we let the generation LLM generate 100 test inputs, guided by different requirement formats. Our default setting, **WF-Mix**, describes the requirements in the mixed format of natural language and pseudo-code generated by the analysis LLM. By contrast, the requirements used in **WF-NL** (resp. **WF-Code**) are the natural language (resp. pseudo-code) description extracted from the mixed format, for a fair comparison. Besides, we also evaluate the performance of directly feeding the generation LLM with the implementation source code, *i.e.,* the **WF-Impl** variant. Regarding the feedback loop, in addition to our default configuration, which uses feedback with Thompson Sampling, we contemplate two alternative variants: one without any feedback (**WF-No-Feedback**) and another that incorporates feedback but employs a simple uniform random selection (**WF-Naive**). Furthermore, we revisit the decision of using GPT4 as the analysis LLM by introducing an additional variant, **WF-SC**, which employs StarCoder as the analysis LLM.

**Environment.** Our test-bed runs Ubuntu 20.04.5 LTS with 64-core CPUs, 256 GB RAM, and NVIDIA RTX A6000 GPUs.

**Fuzzing budget.** Our default setting is to generate a total of 1000 tests for each optimization in 100 iterations. In each iteration, WhiteFox by default generates a batch of 10 tests based on optimization triggering feedback from previous iterations. If the optimization was triggered in previous iterations, WhiteFox picks three triggering inputs used as few-shot examples in the feedback-guided prompt (Section 3.3, Figure 5) for the following iterations. Otherwise, WhiteFox uses the default few-shot prompt (Section 3.2, Figure 4) to generate tests.

### 5.3 Metrics

Following prior work [11, 17, 20, 21, 45, 47, 50], we use the *number of detected bugs* as our metric, which essentially reflects the goal of fuzzing – finding more bugs. Meanwhile, the primary goal of our approach is to effectively test the optimizations within compilers. As such, we also let the *number of triggered optimizations* and the *number of (optimization-)triggering tests* be our principal metrics. Specifically, an optimization is deemed "triggered" if its corresponding optimization function (§ 4) logs its presence during fuzzing. Meanwhile, a test qualifies as a "triggering test" only if during its execution, any of the optimizations are triggered. Given that optimization bugs can only manifest when the optimization is activated, similar to the concept of coverage, a higher number of *triggering tests* correlates with an increased likelihood of bug discovery.

Table 2. Comparison with baselines under the default setting.

| | | # Optim. | # Triggered optim. | # Triggering tests | # Tests | Time/hour |
|---|---|---|---|---|---|---|
| PyTorch Inductor | WhiteFox | 61 | **41** | **21,469** | 61,000 | 41.1 |
| | WhiteFox-Mini | | 39 | 1,737 | 6,100 | 4.2 |
| | TitanFuzz | | 4 | 5,519 | 521,251 | 76.6 |
| | NNSmith | | 5 | 47 | 12,084 | 4.9 |
| TF Lite | WhiteFox | 13 | **12** | 2,801 | 13,000 | 18.1 |
| | WhiteFox-Mini | | 10 | 305 | 1,300 | 1.1 |
| | TitanFuzz | | 8 | 571 | 243,288 | 59.0 |
| | NNSmith | | 7 | **4,666** | 117,381 | 6.8 |
| TF-XLA | WhiteFox | 49 | 20 | 12,990 | 49,000 | 59.7 |
| | WhiteFox-Mini | | 19 | 1,307 | 4,900 | 5.3 |
| | TitanFuzz | | **22** | 45,762 | 243,288 | 63.2 |
| | NNSmith | | 16 | **117,006** | 117,381 | 6.0 |

Table 3. Comparison with baselines over a 24-hour period.

| | | # Optim. | # Triggered optim. | # Triggering tests | # Tests | Coverage |
|---|---|---|---|---|---|---|
| PyTorch Inductor | WhiteFox | 61 | **41** | **12,127** | 35,380 | **54,819** |
| | TitanFuzz | | 4 | 1,697 | 167,521 | 53,592 |
| | NNSmith | | 5 | 233 | 57,664 | 49,910 |
| TF Lite | WhiteFox | 13 | **12** | 3,369 | 16,900 | 52,483 |
| | TitanFuzz | | 7 | 248 | 126,364 | **55,606** |
| | NNSmith | | 8 | **19,747** | 450,197 | 28,108 |
| TF-XLA | WhiteFox | 49 | **19** | 5,183 | 19,600 | **66,224** |
| | TitanFuzz | | **19** | 21,323 | 115,351 | 55,223 |
| | NNSmith | | 16 | **460,453** | 460,970 | 28,108 |

To further show the effectiveness of every component, we also use *code coverage* [17, 21, 37, 45] as a metric. Specifically, we report line coverage in the source languages where the optimizations are implemented: Python for PyTorch Inductor, and C++ for both TensorFlow Lite and TensorFlow-XLA. Following previous work [27, 83, 87], we measure line coverage using `Coverage.py` [2] for Python and GCOV [3] for C++.

## 6 Result Analysis

### 6.1 Comparison with Prior Work

Table 2 compares WhiteFox against the baselines on the three target compilers under their default settings. Because NNSmith has a shorter execution time than our default setting, for fair comparison, we also present results from **WhiteFox-Mini**, which produces 100 tests for each optimization, as opposed to the default 1000 tests. Notably, Column *Time* in Table 2 encompasses both the generation time of requirements/tests (including LLM invocations) and the test-execution time.

In terms of optimization triggering, we observe that WhiteFox outperforms the baselines significantly in PyTorch Inductor and TensorFlow Lite. Overall, among the tested compilers, WhiteFox outperforms existing testers by up to 8.2x in terms of the number of triggered optimizations. For

Table 4. Impact of requirement description formats on PyTorch Inductor.

| | # Triggered optim. (% Total) | # Triggering tests (% Total) |
|---|---|---|
| WF-Mix | **39 (60.9%)** | **1,113 (17.4%)** |
| WF-NL | 37 (57.8%) | 940 (14.7%) |
| WF-Code | 32 (50.0%) | 1,055 (16.5%) |
| WF-Impl | 32 (50.0%) | 638 (10.0%) |
| WF-SC | 32 (50.0%) | 745 (11.6%) |

example, out of the 61 optimizations in PyTorch Inductor, WhiteFox is able to trigger 41 optimizations, while the baseline approaches can trigger at most 5 optimizations, which is a subset of optimizations covered by WhiteFox. Regarding the time cost, WhiteFox consumes less time than all other techniques except NNSmith. Meanwhile, given the inferior performance of NNSmith, WhiteFox-Mini can still trigger more optimizations than NNSmith using less time.

WhiteFox outperforms all baselines on compilers except for TensorFlow-XLA, with two fewer optimizations being triggered compared to TitanFuzz. One possible reason is that the targeted optimizations in TensorFlow-XLA are relatively simple per our optimization filtering for fair comparison with baselines (§ 5.2). Upon inspection, many of these optimizations represent common model patterns that are widely used in practice. Therefore, they can be effectively triggered by TitanFuzz since TitanFuzz leverages LLMs to generate human-like programs by resembling the distribution of training data. Nevertheless, despite generating slightly fewer total tests compared to TitanFuzz, WhiteFox demonstrates its own edge by triggering four unique optimizations which TitanFuzz cannot. In addition, we note that NNSmith has much more triggering tests than WhiteFox and TitanFuzz over TensorFlow-XLA. This is largely due to the implementation choice of NNSmith, which always outputs the model with redundant reshapes. Thus, the vast majority of test inputs from NNSmith can trigger the `IdentityReshapeRemoving` optimization (117,006/117,381).

Regarding unique optimizations triggered by each approach, the baselines trigger 7 optimizations for PyTorch Inductor, while WhiteFox covers these 7 and an additional 34 unique optimizations. For TensorFlow Lite, the baselines trigger 9 optimizations, which are all covered by WhiteFox, plus 3 more unique optimizations. For TensorFlow-XLA, the baselines trigger 26 optimizations, including the 20 covered by WhiteFox.

Additionally, Table 3 compares WhiteFox with the baselines over a 24-hour testing period, a common setting for fuzzing approaches [37]. WhiteFox performs the best on all three subjects, substantially outperforming others on PyTorch Inductor and TensorFlow Lite. In terms of code coverage, WhiteFox covers more lines than the baselines in PyTorch Inductor and TensorFlow-XLA by up to 19.9%. For TensorFlow Lite, WhiteFox performs slightly worse than TitanFuzz (5.9%). This may be attributed to the limited number of optimizations (*13*) in TensorFlow Lite, which inherently restricts WhiteFox's code coverage exploration, as WhiteFox does not have much white-box information (i.e., optimization implementation) to guide the generation. Meanwhile, please kindly note that code coverage is just a proxy indicator and does not always correlate strongly with bug finding abilities for complicated systems [33, 70]. Despite slightly lower code coverage on TensorFlow Lite, WhiteFox still performs better on the ultimate goal, bug finding (detailed in Section 6.3). Overall, these results demonstrate the effectiveness of WhiteFox in generating test cases to cover not only optimizations but also various compiler behaviors.

Table 5. Statistics of the feedback loop on PyTorch Inductor.

| | # Triggering tests | Coverage |
|---|---|---|
| WhiteFox | **21,469** | **55,857** |
| WF-Naive | 17,004 | 54,602 |
| WF-No-Feedback | 8,152 | 52,838 |

### 6.2 Ablation Study

Given that PyTorch Inductor has the highest number of optimizations, our ablation study is solely focused on PyTorch Inductor.

*6.2.1 Requirement Generation & Test Generation.* We first study the effectiveness of the requirement description and the multiple choices for the format (shown in Table 4). The goal of the requirement generation is to assist the generation LLM in producing more tests that can trigger additional optimizations within the compiler. Thus, our main points of comparison are the number of triggered optimizations (Column "# Triggered optim.") and the number of tests that can trigger the optimization (Column "# Triggered tests").

**Effectiveness of requirement description.** Compared with WF-Impl, which feeds the implementation source code directly with the generation LLM, all three variants that use requirements (WF-Mix, WF-NL, and WF-Code) demonstrate superior performance in generating triggering tests. Notably, our default setting WF-Mix is able to generate *1.74x* more triggering tests than directly using implementation code. Besides, WF-Mix can also trigger *7* more optimizations than WF-Impl, emphasizing the importance of using requirement description. This aligns with our statement in the Approach section that optimization source code is not the most effective guide for the generation LLM due to its redundant, unrelated information, and low-level format.

**Effectiveness of mixed format.** As shown in Table 4, WF-Mix achieves the best number of triggered optimizations and triggering tests, underlining the effectiveness of combining NL and pseudo-code for requirement description. Concurrently, while WF-NL triggers more optimizations than WF-Code, it results in fewer triggering tests. This is because NL usually contains additional information than pseudo-code, ensuring vital triggering requirements are not missed during conversion from the implementation source code. Conversely, it is more straightforward for the generation LLM to correlate requirements formatted in pseudo-code with the respective test programs, leading to a higher number of triggering tests.

**Analysis LLM.** When employing requirement descriptions generated by StarCoder, WF-SC not only results in fewer triggered optimizations but also a reduced number of triggering tests compared to our default setting, which utilizes GPT4 to summarize the implementation source code. This discrepancy is anticipated, given that GPT4 is recognized as the cutting-edge LLM in tasks related to code comprehension and natural language generation [6]. Essentially, GPT4 exhibits a superior ability in translating intricate source code details into high-level input requirements compared to StarCoder. This observation underscores our rationale for choosing GPT4 as the analytical LLM. Interestingly, even WF-SC generates a higher number of triggering tests than WF-Impl, which creates the input straight from the implementation source code. Such a discovery confirms that a dual-model infrastructure might be better aligned for white-box compiler fuzzing than directly utilizing the implementation source code, emphasizing the value of having a distinct phase dedicated to requirement generation.

*6.2.2 Feedback Loop.* Next, we examine the effectiveness of our feedback loop and the Thompson Sampling algorithm. The primary aim of the feedback loop is to enhance the likelihood of generating

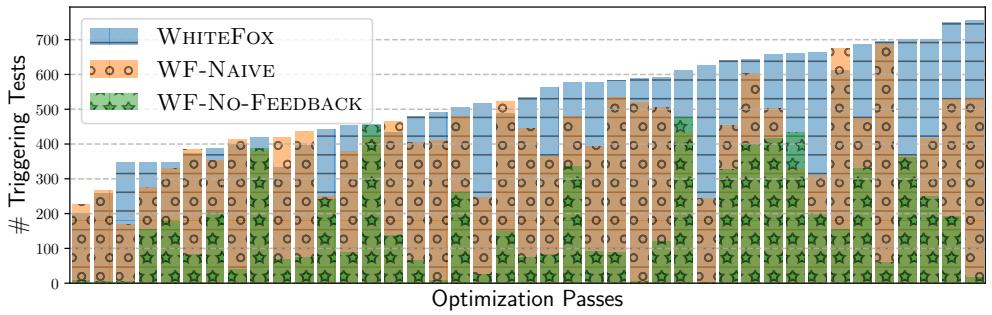



Fig. 6. Impact of the feedback loop and Thompson Sampling on PyTorch Inductor.

Table 6. Overview of WhiteFox-detected bugs.

| | Total | Confirmed | New | Fixed | Won't fix |
|---|---|---|---|---|---|
| PyTorch | 79 | 76 | 74 | 68 | 3 |
| TensorFlow Lite | 11 | 8 | 8 | 0 | 3 |
| TensorFlow-XLA | 11 | 11 | 10 | 2 | 0 |
| Total | 101 | 95 | 92 | 70 | 6 |

additional test inputs that activate already-triggered optimizations. Therefore, in this ablation study, we focus on comparing the number of triggering tests. Figure 6 showcases a bar chart that details the number of triggering tests for each triggered optimization, spanning the range of variants explored in this ablation study. Besides, we also collect the coverage results for these three variants, shown in Table 5.

**Effectiveness of feedback loop.** WhiteFox and WF-Naive, incorporating the feedback loop, produce significantly more triggering tests than the variant without it (WF-No-Feedback), with respectively *2.6x* and *2.1x* increases. This improvement not only emphasizes the effectiveness of the feedback loop but indicates that its guidance is more attuned to triggering the target optimization than relying on few-shot examples for other different optimizations. Furthermore, both approaches outperform WF-No-Feedback in terms of code coverage, demonstrating that the feedback loop can guide LLMs to generate more diverse test cases.

**Effectiveness of Thompson Sampling.** In our default configuration, WhiteFox leverages the Thompson Sampling algorithm for selecting triggering examples and achieves a remarkable *1.3x* more triggering tests compared to WF-Naive, which uses uniform sampling to select the examples. As shown in Figure 6, WhiteFox outperforms the rest over 32 out of the 41 triggered optimizations in terms of the number of trigger tests. In the meanwhile, the code coverage of WhiteFox is higher than WF-Naive. Overall, the experimental results show the effectiveness of our MAB-based triggering example selection.

### 6.3 Bug Finding

Table 6 presents the bug-finding results of WhiteFox over PyTorch, TensorFlow-XLA, and TensorFlow Lite. To date, WhiteFox has detected *101* bugs for them. Of these, 92 are confirmed as previously unknown, and *70 are already fixed*. For PyTorch, we conducted bug detection in both the evaluation and the latest versions upon the request of the PyTorch official team (will be discussed in

Table 7. Characteristics of WhiteFox-detected bugs

| | Crash | Mis-compilation | Failed optim. | Incorrectly passed optim. (OOB) |
|---|---|---|---|---|
| PyTorch Inductor | 6 | 25 | 41 | 7 (3) |
| TensorFlow-XLA | 0 | 4 | 4 | 3 (2) |
| TensorFlow Lite | 0 | 8 | 3 | 0 (0) |
| Total | 6 | 37 | 48 | 10 (5) |

§ 7.1). Thus, among the 79 identified bugs in the PyTorch Inductor, 14 are found in its most recent version. Remarkably, *10 (12.7%)* of the PyTorch bugs have been labeled with *high priority*.

Of the 79 unique WhiteFox-detected bugs, 68 are undetectable by the baselines. For TensorFlow-XLA and TensorFlow Lite, the baselines could find only 1 of the 22 bugs discovered by WhiteFox.

*6.3.1 Bug Analysis.* We next comprehensively analyze the PyTorch Inductor bugs as it is the main source of WhiteFox-detected bugs (79/101, *78.2 %*), and most of them have been fixed (68/79, *86.1 %*). Out of these 79 bugs, only 11 (13.9%) can be covered by the state-of-the-art, with 10 detectable by TitanFuzz and 3 by NNSmith.

Regarding the 68 fixed bugs, we further explore the root cause of the bugs by inspecting their corresponding developer fixes. Impressively, 47 (**69.1%**) of the fixed bugs are repaired in the optimization code of PyTorch Inductor. This demonstrates the effectiveness of WhiteFox for finding optimization bugs, which is the primary goal of our approach. Specifically, only **3** of these 47 optimization bugs can be covered by TitanFuzz and NNSmith, highlighting the significant edge of WhiteFox in testing compiler optimizations. One interesting observation is that certain optimizations appear to be more erroneous than the others; however, such erroneous optimizations instead turn out to be harder to discover. For example, *WhiteFox detects* 5 *bugs in the optimization for the important attention modules [80], which are the fundamental building blocks to LLMs*. The developer-crafted tests may seem surprising in their oversight of multiple critical bugs, but this is due to the challenge of creating precise model patterns to reveal deeply hidden issues. By exposing such critical bugs, WhiteFox demonstrates the power of white-box fuzzing with LLMs.

*6.3.2 Bug Characteristics.* We further study the detailed characteristics of the WhiteFox-detected bugs, which include *crashes*, *mis-compilations*, *failed optimizations*, *incorrectly passed optimizations*, and *vulnerabilities*, shown in Table 7. A *mis-compilation* occurs when the optimized program returns different outputs than the non-optimized one. *Failed optimizations* refer to cases where compilation with optimization fails, while it is valid without optimization. *Incorrectly passed optimizations* occur when the optimization compiles invalid models successfully. Regarding the *vulnerabilities*, in addition to the 6 crash bugs that could be used for DoS attacks, there are another 5 out-of-bound read vulnerabilities detected within the *incorrectly passed optimizations*.

*6.3.3 Won't Fix Bugs.* For the *won't fix bugs*, in PyTorch Inductor, one is due to the compiler not supporting quantized APIs, another is from undefined behavior in operators, and the third is because developers considered our input invalid, despite the optimization compiling the model and returning different results. In TensorFlow Lite, two bugs stem from its feature that doesn't guarantee input-output order, and another is the optimized output having different shapes, which is rare and not expected in both PyTorch Inductor and TensorFlow-XLA.

*6.3.4 Bug Examples.* We demonstrate representative bugs detected by WhiteFox and discuss their exploitation or security implications. Figure 7(a) illustrates a misoptimization of PyTorch


(a) A bug of *attention* optimization in Inductor.

(b) Two bugs of *binary_unary_fusion* optimization in Inductor.

(c) A bug of *FuseUnpackAndConcatToReshape* in TensorflowLite.

(d) An *out-of-bound read* bug in TensorFlow-XLA.

(e) A crash bug in LLVM.

Fig. 7. Example bugs detected by WhiteFox.

Inductor, manifested when compiling attention modules [80]. The faulty optimization through pattern matching identifies self-attentions and fuses their sub-operators into a compact and efficient implementation. However, the optimized attention module, by rearranging the tensor layout to a channel-last format and rendering the last dimension non-contiguous, results in an accuracy issue. This leads to incorrect outputs when compared to those from the unoptimized module. Given the prevalence and impact of attention modules and LLMs, this bug is labeled with *high-priority* and subsequently fixed. The developers highlighted the importance of the issue, stating, *"raising priority due to being an accuracy problem on an important operator"*.

Figure 7(b) presents two bugs detected in PyTorch Inductor for the `binary_unary_fusion` optimization, which fuses the `Linear` (*i.e.,* binary) and `ReLU` (*i.e.,* unary) operators into a compact and thus efficient operation. However, after compilation, such exemplified module returns impermissible negative outputs since its final layer is `ReLU` whose output is always non-negative [4]. Because such `Linear-ReLU` structures are incorporated in many fundamental architectures such as ResNet (Residual Network) [29], this bug, uniquely found by WhiteFox, is labeled as *high-priority* and was fixed immediately after our report. Furthermore, configuring the `ReLU` operation with `inplace=True` results in a crash of the optimized model for inputs of `bfloat16` data type, which can be leveraged for DoS attacks by requesting data in `bfloat16` format. Given the severity of this potential security issue, the developers promptly patched the vulnerability within two days.

Figure 7(c) depicts a bug manifested in the `FuseUnpackAndConcatToReshape` optimization in TensorFlow Lite. Specifically, this optimization aims to streamline `unpack-concat` operation pairs into a single reshape operation, provided that the two operations are semantically inverse to each other. In this optimization, the unpacked dimension should match the concatenated dimension in the original input of the `unpack-concat` operation pair. The example listed in Figure 7(c) violates the assumption and by theory cannot be simplified to a reshaping logic. However, the

Table 8. Comparison with baselines for LLVM optimizations

| | | # Optim. | # Triggered optim. | # Triggering tests | # Tests | Time/hour |
|---|---|---|---|---|---|---|
| LLVM | WhiteFox | 52 | **26** | 25,322 | 52,000 | 30.9 |
| | YARPGen | | 4 | **76,171** | 199,302 | 32.2 |
| | GrayC | | 4 | 8,353 | 107,234 | 30.6 |

`FuseUnpackAndConcatToReshape` function still erroneously transforms it into a wrong reshaping operation. Notably, this bug is exclusively detected by WhiteFox as it hinges on generating valid tests to trigger this particular optimization, which other techniques consistently struggle to accomplish.

Figure 7(d) shows a TensorFlow-XLA bug exclusively detected by WhiteFox. The bug-triggering model contains an embedding layer with a vocabulary size of 64 tokens, followed by one multiplication operation. When the first input to the model is 64, exceeding the embedding layer's maximum token index (63), the unoptimized model raises an `InvalidArgumentError` as expected. However, the model optimized by XLA omits index validation, leading to an out-of-bounds read vulnerability. Given the prevalence/impact of the important embedding layers, developers have swiftly addressed and fixed this issue after seeing our report.

## 7 Discussion

### 7.1 Real-World Impact

Notably, we received the acknowledgment from the PyTorch team along with the request for integrating WhiteFox into the development pipeline of PyTorch Inductor compiler.

> *"Thanks for your contributions to surfacing TorchInductor issues with Whitefox and sharing details. It will be great to figure out the next steps (for integration)." — PyTorch Team*

Consequently, we further extend WhiteFox to accommodate the most recent version of PyTorch Inductor, incorporating support for an additional 38 newly introduced optimizations. This underscores the distinct dynamic of DL compilers, which diverge from traditional compilers due to the brisk pace of DL model architecture evolution and the pressing need to optimize for nascent architectures. For context, PyTorch Inductor has experienced *1,846 commits* in the last year alone. Therefore, the principal focus of WhiteFox on DL compilers is driven by the necessity for an approach that can evolve in tandem with the rapid development of new optimizations. As described in § 6.3, WhiteFox helped detect 14 new bugs for the newly introduced optimizations, all of which have been confirmed by the developers. This underscores WhiteFox's effectiveness and ability to adapt to evolving optimizations, showing the value and significance of incorporating WhiteFox into the development workflow.

### 7.2 Generality: Case Study on LLVM

Although our main focus is on DL compilers in this work, WhiteFox is general to the compilers from various domains that contain pattern-based optimization pipelines. To show the generality of WhiteFox, we implemented a prototype of WhiteFox for testing LLVM [38], which is one of the most popular C/C++ compilers. We tried our best to collect all *middle-end* optimizations in LLVM since they are general to any architecture and are well-documented [51]. In terms of baselines, we include YARPGen [50], a recent fuzzing tool to generate C/C++ test inputs with strategies to trigger different optimizations in compilers, and GrayC [21], state-of-the-art grey-box fuzzer using

coverage feedback to generate test programs in C. Specifically, the LLVM version under test is LLVM-18-20230818-nightly, and the experimental environment matches the setup described in § 5.2.

**Optimization trigger.** Table 8 presents the optimization triggering results for LLVM optimizations against the baselines. Similar to the results on DL compilers, we observe that WhiteFox can trigger *6.5x* more optimizations than baselines while incurring less time cost. This demonstrates the effectiveness and potential of WhiteFox on different compilers.

**Bug detection.** WhiteFox detects 6 bugs for LLVM, with 2 confirmed as previously unknown, 3 pending, and 1 won't fix. Figure 7(e) presents a confirmed LLVM bug, which is only revealed when a test program references a huge array through a large enough index, crashing the LLVM post-optimization. Attackers can exploit this vulnerability for DoS by crafting specific input programs to crash Just-in-Time-enabled systems that use this optimization.

### 7.3 Limitations and Future Work

While this paper focuses on compiler optimization, WhiteFox could be potentially adapted for white-box fuzzing of other compiler code and even other complex, real-world software systems. For example, for regression bugs, WhiteFox can be deployed by setting the changed branches as targets and letting LLMs analyze their triggering conditions. However, one limitation or challenge is that, for optimization, the high-level system inputs usually have a relatively clear mapping to the low-level optimization implementation. In contrast, for arbitrary functions, this mapping may be unclear. To mitigate this, one possibility is to further leverage LLMs to infer such mappings (*e.g.*, leveraging auxiliary information, including documentation) along with the triggering conditions.

Another possible future direction is to use traditional fuzzing approaches as external tools for efficient test generation. Given the higher computational cost of invoking smaller LLMs compared to traditional techniques, this strategy could improve performance. For instance, we could utilize WhiteFox to summarize optimization triggering generation or mutation rules, which could then guide input generation for a traditional fuzzing framework such as NNSmith [45].

## 8 Conclusion

We present WhiteFox, the first practical white-box compiler fuzzer to test compiler optimizations. WhiteFox adopts a multi-agent design: an analysis LLM reads through the implementation code of compiler optimizations and summarizes desired patterns of test programs, with which a generation LLM is then prompted to efficiently and continuously synthesize meaningful test programs to exercise corresponding optimizations. Our evaluation shows that WhiteFox is effective in testing the emerging DL compilers and is also adaptable to the conventional C/C++ compilers. To date, WhiteFox has found in total 101 bugs for DL compilers, with 92 confirmed as previously unknown and 70 already fixed.

### Data-Availability Statement

The artifact of WhiteFox is available at https://github.com/ise-uiuc/WhiteFox.

### Acknowledgments

We thank Steven Chunqiu Xia for providing the help and resources to run some experiments. This work was partially supported by NSF grant CCF-2131943 and Kwai Inc. This project is supported, in part, by funding from Two Sigma Investments, LP. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of Two Sigma Investments, LP.

## References


[1] 2021. News. https://www.vice.com/en_us/article/9kga85/uber-is-giving-up-on-self-driving-cars-in-california-after-deadly-crash.

[2] 2022. Coverage.py. https://github.com/nedbat/coveragepy.

[3] 2022. GCOV. https://gcc.gnu.org/onlinedocs/gcc/Gcov.html.

[4] Abien Fred Agarap. 2018. Deep learning using rectified linear units (relu). *arXiv preprint arXiv:1803.08375* (2018).

[5] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *Advances in neural information processing systems* 33 (2020), 1877–1901.

[6] Sébastien Bubeck, Varun Chandrasekaran, Ronen Eldan, Johannes Gehrke, Eric Horvitz, Ece Kamar, Peter Lee, Yin Tat Lee, Yuanzhi Li, Scott Lundberg, et al. 2023. Sparks of artificial general intelligence: Early experiments with gpt-4. *arXiv preprint arXiv:2303.12712* (2023).

[7] 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.

[8] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374* (2021).

[9] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. 2018. {TVM}: An automated {End-to-End} optimizing compiler for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. 578–594.

[10] Yinghao Chen, Zehao Hu, Chen Zhi, Junxiao Han, Shuiguang Deng, and Jianwei Yin. 2024. ChatUniTest: A Framework for LLM-Based Test Generation. In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*. 572–576.

[11] Yongheng Chen, Rui Zhong, Hong Hu, Hangfan Zhang, Yupeng Yang, Dinghao Wu, and Wenke Lee. 2021. One engine to fuzz'em all: Generic language processor testing with semantic validation. In *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, 642–658.

[12] Mingi Cho, Seoyoung Kim, and Taekyoung Kwon. 2019. Intriguer: Field-level constraint solving for hybrid fuzzing. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. 515–530.

[13] Jaeseung Choi, Joonun Jang, Choongwoo Han, and Sang Kil Cha. 2019. Grey-box concolic testing on binary code. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 736–747.

[14] Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, et al. 2022. Palm: Scaling language modeling with pathways. *arXiv preprint arXiv:2204.02311* (2022).

[15] Neophytos Christou, Di Jin, Vaggelis Atlidakis, Baishakhi Ray, and Vasileios P Kemerlis. 2023. {IvySyn}: Automated Vulnerability Discovery in Deep Learning Frameworks. In *32nd USENIX Security Symposium (USENIX Security 23)*. 2383–2400.

[16] Arghavan Moradi Dakhel, Amin Nikanjam, Vahid Majdinasab, Foutse Khomh, and Michel C Desmarais. 2024. Effective test generation using pre-trained large language models and mutation testing. *Information and Software Technology* 171 (2024), 107468.

[17] Yinlin Deng, Chunqiu Steven Xia, Haoran Peng, Chenyuan Yang, and Lingming Zhang. 2023. Large Language Models are Zero-Shot Fuzzers: Fuzzing Deep-Learning Libraries via Large Language Models. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2023)*.

[18] Yinlin Deng, Chunqiu Steven Xia, Chenyuan Yang, Shizhuo Dylan Zhang, Shujing Yang, and Lingming Zhang. 2023. Large language models are edge-case fuzzers: Testing deep learning libraries via fuzzgpt. *arXiv preprint arXiv:2304.02014* (2023).

[19] Alastair F Donaldson, Hugues Evrard, Andrei Lascu, and Paul Thomson. 2017. Automated testing of graphics shader compilers. *Proceedings of the ACM on Programming Languages* 1, OOPSLA (2017), 1–29.

[20] Alastair F Donaldson, Paul Thomson, Vasyl Teliman, Stefano Milizia, André Perez Maselco, and Antoni Karpiński. 2021. Test-case reduction and deduplication almost for free with transformation-based compiler testing. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 1017–1032.

[21] Karine Even-Mendoza, Arindam Sharma, Alastair F Donaldson, and Cristian Cadar. 2023. GrayC: Greybox Fuzzing of Compilers and Analysers for C. (2023).

[22] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. 2020. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155* (2020).

[23] Gordon Fraser and Andrea Arcuri. 2011. Evosuite: automatic test suite generation for object-oriented software. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software*

*engineering*. 416–419.
[24] GCC 2023. GCC. https://gcc.gnu.org/.
[25] Patrice Godefroid, Nils Klarlund, and Koushik Sen. 2005. DART: Directed Automated Random Testing. 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, 213–223. https://doi.org/10.1145/1065010.1065036
[26] Rahul Gopinath, Bachir Bendrissou, Björn Mathis, and Andreas Zeller. 2020. Fuzzing with fast failure feedback. *arXiv preprint arXiv:2012.13516* (2020).
[27] J. Gu, X. Luo, Y. Zhou, and X. Wang. 2022. Muffin: Testing Deep Learning Libraries via Neural Architecture Fuzzing. In *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)*. IEEE Computer Society, Los Alamitos, CA, USA, 1418–1430. https://doi.org/10.1145/3510003.3510092
[28] Qianyu Guo, Xiaofei Xie, Yi Li, Xiaoyu Zhang, Yang Liu, Xiaohong Li, and Chao Shen. 2020. Audee: Automated testing for deep learning frameworks. In *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 486–498.
[29] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Identity mappings in deep residual networks. In *Computer Vision–ECCV 2016: 14th European Conference, Amsterdam, The Netherlands, October 11–14, 2016, Proceedings, Part IV 14*. Springer, 630–645.
[30] Christian Holler, Kim Herzig, and Andreas Zeller. 2012. Fuzzing with code fragments. In *21st USENIX Security Symposium (USENIX Security 12)*. 445–458.
[31] Heqing Huang, Peisen Yao, Rongxin Wu, Qingkai Shi, and Charles Zhang. 2020. Pangolin: Incremental hybrid fuzzing with polyhedral path abstraction. In *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1613–1627.
[32] HuggingFace 2023. Hugging Face. https://huggingface.co.
[33] Laura Inozemtseva and Reid Holmes. 2014. Coverage is not strongly correlated with test suite effectiveness. In *Proceedings of the 36th international conference on software engineering*. 435–445.
[34] Ling Jiang, Hengchen Yuan, Mingyuan Wu, Lingming Zhang, and Yuqun Zhang. 2023. Evaluating and improving hybrid fuzzing. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 410–422.
[35] Kyungtae Kim, Dae R Jeong, Chung Hwan Kim, Yeongjin Jang, Insik Shin, and Byoungyoung Lee. 2020. HFL: Hybrid Fuzzing on the Linux Kernel.. In *NDSS*.
[36] James C King. 1976. Symbolic execution and program testing. *Commun. ACM* 19, 7 (1976), 385–394.
[37] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. 2018. Evaluating fuzz testing. In *Proceedings of the 2018 ACM SIGSAC conference on computer and communications security*. 2123–2138.
[38] Chris Lattner and Vikram Adve. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *International symposium on code generation and optimization, 2004. CGO 2004.* IEEE, 75–86.
[39] Vu Le, Mehrdad Afshari, and Zhendong Su. 2014. Compiler validation via equivalence modulo inputs. *ACM Sigplan Notices* 49, 6 (2014), 216–226.
[40] Vu Le, Chengnian Sun, and Zhendong Su. 2015. Finding deep compiler bugs via guided stochastic program mutation. *ACM SIGPLAN Notices* 50, 10 (2015), 386–399.
[41] Caroline Lemieux, Jeevana Priya Inala, Shuvendu K Lahiri, and Siddhartha Sen. 2023. CODAMOSA: Escaping coverage plateaus in test generation with pre-trained large language models. In *International conference on software engineering (ICSE)*.
[42] Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, et al. 2023. StarCoder: may the source be with you! *arXiv preprint arXiv:2305.06161* (2023).
[43] Wen Li, Haoran Yang, Xiapu Luo, Long Cheng, and Haipeng Cai. 2023. PyRTFuzz: Detecting Bugs in Python Runtimes via Two-Level Collaborative Fuzzing. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*. 1645–1659.
[44] libFuzzer 2023. libFuzzer – a library for coverage-guided fuzz testing. https://llvm.org/docs/LibFuzzer.html.
[45] Jiawei Liu, Jinkun Lin, Fabian Ruffy, Cheng Tan, Jinyang Li, Aurojit Panda, and Lingming Zhang. 2023. NNSmith: Generating Diverse and Valid Test Cases for Deep Learning Compilers. In *ASPLOS*. 530–543.
[46] Jiawei Liu, Jinjun Peng, Yuyao Wang, and Lingming Zhang. 2023. NeuRI: Diversifying DNN Generation via Inductive Rule Inference. 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, 657–669. https://doi.org/10.1145/3611643.3616337
[47] Jiawei Liu, Yuxiang Wei, Sen Yang, Yinlin Deng, and Lingming Zhang. 2022. Coverage-Guided Tensor Compiler Fuzzing with Joint IR-Pass Mutation. *Proc. ACM Program. Lang.* 6, OOPSLA1, Article 73 (apr 2022), 26 pages. https://doi.org/10.1145/3527317
[48] Nelson F Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, and Percy Liang. 2023. Lost in the middle: How language models use long contexts. *arXiv preprint arXiv:2307.03172* (2023).

[49] Yinxi Liu and Wei Meng. 2023. DSFuzz: Detecting Deep State Bugs with Dependent State Exploration. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*. 1242–1256.

[50] Vsevolod Livinskii, Dmitry Babokin, and John Regehr. 2020. Random testing for C and C++ compilers with YARPGen. *Proceedings of the ACM on Programming Languages* 4, OOPSLA (2020), 1–25.

[51] LLVM 2023. LLVM's Analysis and Transform Passes. https://llvm.org/docs/Passes.html.

[52] Björn Mathis, Rahul Gopinath, and Andreas Zeller. 2020. Learning input tokens for effective fuzzing. In *Proceedings of the 29th ACM SIGSOFT international symposium on software testing and analysis*. 27–37.

[53] James B McDonald and Yexiao J Xu. 1995. A generalization of the beta distribution with applications. *Journal of Econometrics* 66, 1-2 (1995), 133–152.

[54] William M McKeeman. 1998. Differential testing for software. *Digital Technical Journal* 10, 1 (1998), 100–107.

[55] MKLDNN 2024. MKL-DNN. https://github.com/rsdubtso/mkl-dnn.

[56] Pengyu Nie, Rahul Banerjee, Junyi Jessy Li, Raymond J Mooney, and Milos Gligoric. 2023. Learning Deep Semantics for Test Completion. *arXiv preprint arXiv:2302.10166* (2023).

[57] oneDNN 2024. oneDNN. https://github.com/oneapi-src/oneDNN.

[58] OpenAI. 2023. ChatGPT. (2023). https://openai.com/blog/chatgpt.

[59] OpenAI. 2023. GPT-4 Technical Report. arXiv:2303.08774 [cs.CL]

[60] Kexin Pei, Yinzhi Cao, Junfeng Yang, and Suman Jana. 2017. Deepxplore: Automated whitebox testing of deep learning systems. In *proceedings of the 26th Symposium on Operating Systems Principles*. 1–18.

[61] Hung Viet Pham, Thibaud Lutellier, Weizhen Qi, and Lin Tan. 2019. CRADLE: Cross-Backend Validation to Detect and Localize Bugs in Deep Learning Libraries. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. 1027–1038. https://doi.org/10.1109/ICSE.2019.00107

[62] PyTorch 2023. PyTorch. http://pytorch.org.

[63] PyTorch 2023. PyTorch 2.0. https://pytorch.org/get-started/pytorch-2.0.

[64] Max Schäfer, Sarah Nadi, Aryaz Eghbali, and Frank Tip. 2023. Adaptive test generation using a large language model. *arXiv preprint arXiv:2302.06527* (2023).

[65] Mozilla Security. 2007. jsfunfuzz. https://github.com/MozillaSecurity/funfuzz.

[66] Koushik Sen. 2007. Concolic testing. In *Proceedings of the 22nd IEEE/ACM international conference on Automated software engineering*. 571–572.

[67] Koushik Sen, Darko Marinov, and Gul Agha. 2005. CUTE: A Concolic Unit Testing Engine for C. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Lisbon, Portugal) *(ESEC/FSE-13)*. Association for Computing Machinery, New York, NY, USA, 263–272. https://doi.org/10.1145/1081706.1081750

[68] Weijie Shao, Yuyang Gao, Fu Song, Sen Chen, and Lingling Fan. 2023. An Empirical Study of Bugs in Open-Source Federated Learning Framework. *ArXiv* abs/2308.05014 (2023). https://api.semanticscholar.org/CorpusID:265221980

[69] Dave Shreiner et al. 2009. *OpenGL programming guide: the official guide to learning OpenGL, versions 3.0 and 3.1*. Pearson Education.

[70] Ting Su, Jue Wang, and Zhendong Su. 2021. Benchmarking automated GUI testing for Android against real-world bugs. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 119–130.

[71] Chengnian Sun, Vu Le, Qirun Zhang, and Zhendong Su. 2016. Toward understanding compiler bugs in GCC and LLVM. In *Proceedings of the 25th international symposium on software testing and analysis*. 294–305.

[72] Maolin Sun, Yibiao Yang, Yang Wang, Ming Wen, Haoxiang Jia, and Yuming Zhou. 2023. SMT Solver Validation Empowered by Large Pre-trained Language Models. In *ASE*.

[73] Michael Sutton, Adam Greene, and Pedram Amini. 2007. *Fuzzing: brute force vulnerability discovery*. Pearson Education.

[74] Yutian Tang, Zhijie Liu, Zhichao Zhou, and Xiapu Luo. 2024. Chatgpt vs sbst: A comparative assessment of unit test suite generation. *IEEE Transactions on Software Engineering* (2024).

[75] TensorFlow 2023. TensorFlow. https://www.tensorflow.org.

[76] TensorFlowLite 2023. TensorFlow Lite. https://www.tensorflow.org/lite.

[77] TensorFlowXLA 2023. TensorFlow XLA. https://www.tensorflow.org/xla.

[78] William R Thompson. 1933. On the likelihood that one unknown probability exceeds another in view of the evidence of two samples. *Biometrika* 25, 3-4 (1933), 285–294.

[79] Triton 2024. Triton. https://github.com/openai/triton.

[80] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *Advances in neural information processing systems* 30 (2017).

[81] Jiannan Wang, Thibaud Lutellier, Shangshu Qian, Hung Viet Pham, and Lin Tan. 2022. EAGLE: Creating Equivalent Graphs to Test Deep Learning Libraries. (2022).

[82] Zan Wang, Ming Yan, Junjie Chen, Shuang Liu, and Dongdi Zhang. 2020. Deep learning library testing via effective model generation. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 788–799.

[83] Anjiang Wei, Yinlin Deng, Chenyuan Yang, and Lingming Zhang. 2022. Free Lunch for Testing: Fuzzing Deep-Learning Libraries from Open Source. In *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)*. 995–1007. https://doi.org/10.1145/3510003.3510041

[84] Chunqiu Steven Xia, Matteo Paltenghi, Jia Le Tian, Michael Pradel, and Lingming Zhang. 2023. Universal fuzzing via large language models. *arXiv preprint arXiv:2308.04748* (2023).

[85] Danning Xie, Yitong Li, Mijung Kim, Hung Viet Pham, Lin Tan, Xiangyu Zhang, and Michael W Godfrey. 2022. DocTer: documentation-guided fuzzing for testing deep learning API functions. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*. 176–188.

[86] 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.

[87] Chenyuan Yang, Yinlin Deng, Jiayi Yao, Yuxing Tu, Hanchi Li, and Lingming Zhang. 2023. Fuzzing Automatic Differentiation in Deep-Learning Libraries. In *Proceedings of the 45th International Conference on Software Engineering* (Melbourne, Victoria, Australia) *(ICSE '23)*. IEEE Press, 1174–1186. https://doi.org/10.1109/ICSE48619.2023.00105

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

[89] Zhaomo Yang, Brian Johannesmeyer, Anders Trier Olesen, Sorin Lerner, and Kirill Levchenko. 2017. Dead store elimination (still) considered harmful. In *26th USENIX Security Symposium (USENIX Security 17)*. 1025–1040.

[90] Shafiq Joty Yue Wang, Weishi Wang and Steven C.H. Hoi. 2021. CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation. In *EMNLP 2021*.

[91] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. 2018. {QSYM}: A practical concolic execution engine tailored for hybrid fuzzing. In *27th USENIX Security Symposium (USENIX Security 18)*. 745–761.

[92] Andreas Zeller, Rahul Gopinath, Marcel Böhme, Gordon Fraser, and Christian Holler. 2019. The fuzzing book.

[93] Quanjun Zhang, Chunrong Fang, Yang Xie, Yaxin Zhang, Yun Yang, Weisong Sun, Shengcheng Yu, and Zhenyu Chen. 2023. A survey on large language models for software engineering. *arXiv preprint arXiv:2312.15223* (2023).

[94] Qirun Zhang, Chengnian Sun, and Zhendong Su. 2017. Skeletal program enumeration for rigorous compiler testing. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 347–361.

Received 2024-04-06; accepted 2024-08-18