TRIGFUZZ: Triggering Conditions Guided Directed Fuzzing

Yiyang Chen*, Nuoqi Gui*, Long Wang*✉, Longfei Chen*, Xuanqing Shi*, Xi Cao†, Chao Zhang*†✉
*Tsinghua University
†JCSS, Tsinghua University (INSC) - Science City (Guangzhou) Digital Technology Group Co., Ltd.
chenyy23@mails.tsinghua.edu.cn, ✉{longwang, chaoz}@tsinghua.edu.cn

***Abstract*—Directed fuzzing aims to trigger specific vulnerabilities by steering execution towards predefined target code. However, state-of-the-art directed fuzzers predominantly focus on reaching the target code quickly, often lacking effective follow-up strategies to satisfy the vulnerability constraints required to trigger them. We find that this can be a key factor limiting their performance in directed fuzzing tasks such as crash reproduction. The main challenge is that existing directed fuzzers cannot accurately identify the triggering conditions of target vulnerabilities and effectively exploit them to guide fuzzing.**

To address this challenge, we propose TRIGFUZZ, a directed fuzzing solution guided by triggering conditions. Our approach leverages pre-trained large language models (LLMs) to automatically generate the triggering conditions of target vulnerabilities. We design a formalized representation for generated triggering conditions, along with a novel dynamic triggering validation technique to verify their correctness. The verified conditions are further transformed into "triggering distance" metrics that serve as fuzzing runtime feedback to guide seed scheduling and mutation strategies, enabling directed fuzzing to effectively generate vulnerability-triggering test cases. Our evaluations demonstrate that TRIGFUZZ can generate high-quality triggering conditions for 96.67% of target vulnerabilities and outperform state-of-the-art directed fuzzers with over a 1.72x speedup in reproducing target vulnerabilities on the benchmark Magma. Lastly, we detected 7 previously unknown vulnerabilities with 2 CVE IDs assigned in well-tested real-world software using TRIGFUZZ.

1. Introduction

Directed fuzzing represents a significant branch of fuzzing [1] and has been widely applied to critical software engineering tasks such as bug report verification [2], [3] and vulnerability reproduction [4], [5]. Unlike general coverage-guided fuzzing that aims to comprehensively explore the program under test (PUT), directed fuzzing focuses on testing predetermined target code sites within the PUT [6]. For instance, crash points identified from bug reports can be designated as target sites, and directed fuzzing can be used to focus testing on these locations in order to generate proof-of-concept (PoC) inputs, thereby assisting developers in efficiently debugging and patching the vulnerabilities. In typical application scenarios, triggering the target vulnerabilities is the ultimate goal of directed fuzzing.

In recent years, numerous directed fuzzing approaches have been proposed, often focusing on reaching the target code more quickly—a prerequisite for triggering the vulnerabilities that reside in it. Works exemplified by AFLGo [6], [7], [8] employ fitness metrics to quantify the distance between seed inputs and target sites and prioritize closer seeds during seed scheduling. SelectFuzz [9] and DAFL [3] narrow down the fuzzing search space by selectively collecting execution feedback from code regions relevant to the targets. Some other works such as Beacon [10] and Halo [11] leverage static or dynamic analysis of target code reachability conditions to prune inputs deemed unreachable or strategically generate path-constrained inputs.

Although these approaches have successfully reduced the time required to reach the target code [8], [9], [10], we find that they lack effective follow-up strategies to actually trigger the vulnerabilities. When applied to reachable inputs, their fuzzing strategies often degenerate into coverage-guided fuzzing, which neither prioritizes seed inputs based on their potential to trigger vulnerabilities nor employs dedicated mutation strategies for generating vulnerability-triggering inputs. As a result, these directed fuzzers often struggle to trigger the vulnerabilities, even when they reach the target code early. According to prior findings [11], the average time-to-trigger of directed fuzzers is more than 100x the time-to-reach on the Magma benchmark [12].

Two directed fuzzers, CAFL [13] and SDFuzz [2], have devised heuristic strategies aimed at accelerating the triggering of target vulnerabilities. CAFL utilizes manually crafted constraint templates of certain vulnerability types to approximate their triggering conditions, prioritizing seed inputs that better satisfy these constraints. SDFuzz treats reaching a specific call trace as a prerequisite for triggering vulnerabilities and dynamically terminates executions that are deemed unable to satisfy the required call trace. However, we argue that they are also insufficient, as evidenced by our evaluation results in §5.1. The primary reason is that triggering conditions are typically closely tied to both code semantics and vulnerability types [14], making them highly target-specific. CAFL's predefined templates cannot support many vulnerability types (*e.g.*, floating-point exceptions). Meanwhile, since CAFL overlooks code semantics, it also fails to handle many special cases within the supported

types. SDFuzz employs a generic call trace-based strategy, representing only a coarse and conservative approximation that may neglect specific code semantics and crucial intra-procedural information, thus limiting its effectiveness.

Large language models (LLMs) have demonstrated remarkable capabilities in code comprehension and insight generation [15], [16]. Their training corpora typically incorporate high-quality security-related resources, such as diverse bug reports, enabling them to enhance or potentially replace human expertise across a wide range of security-related tasks [17]. It appears promising that LLMs, when provided with the source code along with corresponding bug reports of target vulnerabilities, can effectively analyze and generate precise triggering conditions to guide directed fuzzing.

However, there remains a significant gap in leveraging LLMs for effective vulnerability-triggering guidance. We summarize two primary challenges as follows:

C1: How can LLMs be applied to generate correct and informative triggering conditions? Triggering conditions can be a broad concept. For example, CVE profiles sometimes provide coarse-grained natural language descriptions of how the corresponding vulnerabilities can be triggered. However, such "triggering conditions" are neither sufficient nor practical for guiding directed fuzzing. To address this, we need a formal representation of triggering conditions that incorporates rich, vulnerability-specific knowledge extracted by LLMs while providing *actionable guidance* for dedicated fuzzing strategies. Moreover, real-world programs often have large codebases that contain substantial noise. Trivial outputs and hallucinations may be generated by LLMs if improper code context or poorly constructed prompts are provided. Thus, it is also crucial to prompt LLMs carefully to ensure the *correctness* of the analysis.

C2: How to leverage generated triggering conditions to guide directed fuzzing? Even if correct and informative triggering conditions can be generated, they often vary dramatically across different target vulnerabilities. While designing fuzzing strategies based on triggering conditions to rapidly trigger vulnerabilities is the core objective of this work, reconstructing an entirely new fuzzer for each target vulnerability is impractical. Therefore, designing a robust fuzzing strategy that can automatically adapt to different targets remains a significant challenge.

Our Approach. To address these challenges, we propose an LLM-based directed fuzzing approach that automatically analyzes and effectively exploits triggering conditions to generate vulnerability-triggering test cases.

For *C1*, we begin by utilizing a pre-trained LLM to analyze the target program's source code and the associated bug report to extract essential knowledge about the target vulnerability. We then strategically prompt the LLM to produce *a series of formal 5-tuple representations of the triggering conditions* that encapsulate critical elements such as the required data constraints and execution order. These formal representations enable the accurate modeling of complex real-world triggering conditions. To ensure correctness, we introduce a *dynamic triggering validation* mechanism that leverages fuzzing-generated reachable inputs to verify the generated triggering conditions in an online manner.

For *C2*, we design a novel directed fuzzing framework that dynamically adapts the fuzzing strategy to different triggering conditions. We first develop an algorithm that computes a novel fitness metric, *triggering distance*, based on the triggering conditions of the target vulnerability to evaluate the potential of different reachable inputs in triggering the vulnerability. We then employ lightweight source-level instrumentation to collect the triggering distance for each input as runtime feedback during fuzzing. Leveraging the triggering distance of inputs, we refine both the seed creation rule and the seed scheduling strategy to effectively preserve and prioritize inputs with greater potential to trigger the target vulnerability. Moreover, we develop a *triggering byte-aware mutation strategy* based on a key observation: the input bytes influencing the triggering distance are fixed. In other words, input bytes whose mutations can reduce the triggering distance often have the potential to be further mutated to continuously reduce the triggering distance, ultimately leading to the generation of vulnerability-triggering test cases. This is because the triggering conditions are fixed, and thus the associated variables and their corresponding input bytes remain stable. Thus, we dynamically monitor input mutations that cause changes in the triggering distance during fuzzing and leverage this information to narrow down the range of input bytes to mutate. We then thoroughly mutate these bytes to trigger the target vulnerability.

We implemented our approach in a prototype named **TRIGFUZZ**. In our evaluation, we used the Magma benchmark [12] and included four state-of-the-art directed fuzzers: AFLGo [6], SelectFuzz [9], CAFL [13], and SDFuzz [2], as baselines. We also integrated TRIGFUZZ into IDFuzz [18], one of the most advanced directed fuzzers optimized for target code reachability, and tested it on the Google Fuzzer Test Suite [19] used in IDFuzz's original paper. The evaluation results demonstrate that TRIGFUZZ generates high-quality triggering conditions for 96.67% of targets in Magma while significantly outperforming baseline directed fuzzers in reproducing target vulnerabilities with over 1.72x speedup. Moreover, TRIGFUZZ's strategy for accelerating vulnerability triggering integrates well with IDFuzz, achieving 3.14x and 2.73x speedup over IDFuzz and TRIGFUZZ alone, respectively, in reproducing target vulnerabilities. Additionally, we discovered 7 new vulnerabilities using TRIGFUZZ.

In summary, our contributions are as follows:

- We identified the lack of vulnerability-triggering optimization in existing directed fuzzers and proposed leveraging LLMs to generate and exploit triggering conditions to enhance directed fuzzing.
- We introduced a *5-tuple formal representation of triggering conditions* and a *dynamic triggering validation* mechanism, enabling pre-trained LLMs to accurately generate the triggering conditions for diverse real-world target vulnerabilities.
- We define a new metric, *triggering distance*, based on the generated triggering conditions to measure the potential of inputs in triggering the vulnerability. Lever-

aging this *triggering distance*, we develop a novel directed fuzzing framework incorporating dedicated seed scheduling and mutation strategies to accelerate vulnerability triggering.
- We implemented a prototype directed fuzzer TRIGFUZZ. Our experiments showed that TRIGFUZZ significantly outperformed state-of-the-art directed fuzzers while being complementary to them.
- We discovered 7 new vulnerabilities using TRIGFUZZ.

2. Background and Motivation

2.1. Directed Grey-box Fuzzing

Directed fuzzing aims to focus testing efforts on specified critical regions in the PUT and has garnered significant research interest in recent years as a promising software testing technique [1], [11]. Compared to constraint-solving techniques such as symbolic execution, directed fuzzing avoids scalability issues like path explosion and expensive constraint solving [6], [20], [21], offering significant advantages in efficiently reproducing specific vulnerabilities.

Typical application scenarios for directed fuzzing include crash reproduction, candidate vulnerability confirmation, and patch testing. In these scenarios, information about the target vulnerability is often available, whereas a proof-of-concept (PoC) input is absent. For instance, vulnerability reporters may be unable to share crash inputs due to privacy concerns (*e.g.*, inputs containing sensitive user data), or they can only provide environment-specific inputs that are difficult to reproduce in other settings (*e.g.*, inputs tied to outdated software versions). In either case, they resort to providing detailed vulnerability descriptions instead. Similarly, LLM-based static analysis tools may report potential vulnerabilities with inferred contextual details, yet cannot provide actual crash inputs. To confirm these vulnerabilities, security analysts identify potential vulnerability locations from the available reports and employ directed fuzzing to generate PoC inputs.

Beyond target code locations, some directed fuzzers further leverage additional information in the reports. For example, CAFL [13] exploits vulnerability types and crash dump data provided in bug reports to set constraint templates, and SDFuzz [2] uses the target call traces from bug reports to guide fuzzing.

Motivated by the setting established in previous research, we similarly consider bug reports as valid inputs for our system. This enables the LLM to more accurately generate the triggering conditions for the target vulnerabilities.

2.2. LLM-Enhanced Fuzzing

LLMs have demonstrated remarkable capabilities in understanding and generating natural language and code [17]. Harnessing the power of LLMs to enhance fuzzing has become a growing interest among researchers [22]. One line of research leverages LLMs to assist in generating text-based inputs for fuzzing. Prior works, such as TitanFuzz [23] and Fuzz4All [24], utilize LLMs directly as fuzzing engines to generate inputs. ChatAFL [25] employs LLMs to analyze protocol specifications and subsequently constructs protocol grammars to guide input mutations. Magneto [26] leverages LLMs to facilitate the generation of initial seed corpora. Another line of research focuses on utilizing LLMs to generate fuzz drivers. PromptFuzz [27] is a coverage-guided fuzzer designed specifically for prompt fuzzing, iteratively generating fuzz drivers to explore previously undiscovered library code. Zhang et al. [28] have systematically investigated the effectiveness of employing LLMs for generating effective fuzz drivers.

While these prior works leverage LLMs to generate text-based fuzz inputs or fuzz drivers, there is still limited exploration of applying LLMs to fuzzing tasks that involve binary-format inputs [22]. In this work, we leverage the code comprehension capabilities of LLMs to analyze the triggering conditions of target vulnerabilities and use them to inform dedicated seed scheduling and mutation strategies that accelerate vulnerability triggering. Unlike previous approaches that are limited to text-based fuzz inputs, our optimization method does not directly utilize LLMs to generate fuzz inputs. Instead, LLMs are employed to assist at a higher-level fuzzing strategy, enabling generalization across various input formats.

2.3. A Motivating Example

Listing 1 presents a simplified example of a use-of-uninitialized-variable vulnerability caused by improper use of the `sscanf` function. Such vulnerabilities are common in real-world software, as demonstrated by CVE-2019-12730 [29] and CVE-2021-22925 [30].

In this example, line 5 uses `sscanf` to parse `input.field` into variables `a`, `b`, `c`, and `d` using the format string `"%d.%d.%d.%d"`. If `input.field` does not strictly conform to this format, `sscanf` may partially initialize the variables from left to right. For example, if the string is `"192.168"`, only $a = 192$ and $b = 168$ are initialized, while `c` and `d` remain uninitialized. The return value `v` indicates the number of successfully assigned variables. When $v < 2$, a use-of-uninitialized-variable vulnerability involving `b` may be triggered at line 6.

Limitations of Existing Directed Fuzzers. Most existing directed fuzzing tools [6], [7], [8], [10], [11] focus on reaching the target code (*i.e.*, line 5 and line 6) quickly. However, once the target is reached, they are unaware of the triggering conditions for the use-of-uninitialized-variable vulnerability. Consequently, they are unable to distinguish among reachable inputs based on their potential to trigger the vulnerability, nor can they perform fine-grained mutations on the input fields corresponding to `input.field`.

CAFL employs a template for use-of-uninitialized-variable vulnerabilities that prioritizes inputs which execute the "use" site but not the "initialization" site [13]. However, this is not applicable to this example, as "initialization" (line 5) and "use" (line 6) are always executed together.

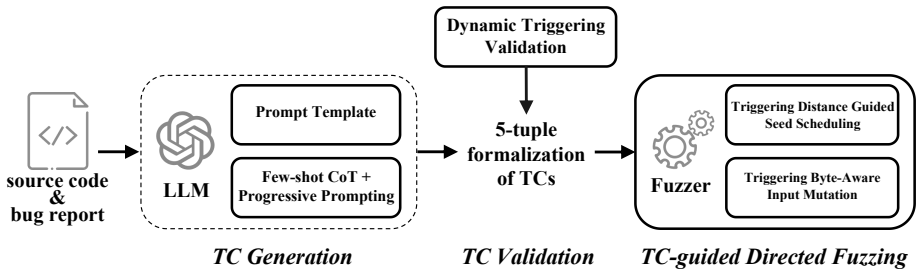

Figure 1: Workflow of TRIGFUZZ.

```
1 #include <stdio.h>
2
3 void foo() {
4     int a, b, c, d;
5     int v = sscanf(input.field,"%d.%d.%d.%d",&a,&b,&c,&d);
6     use(b); // use-of-uninitialized-variable
7 }
```

Listing 1: A motivating example.

SDFuzz guides fuzzing towards target program states by early terminating executions that are deemed unlikely to satisfy the required call trace [2]. However, this strategy only applies to inter-procedural scenarios and is ineffective in this example, where the vulnerability occurs entirely within the `foo` function and thus remains undetected.

Consequently, in this example, existing directed fuzzers consistently fail to effectively trigger the use-of-uninitialized-variable vulnerability.

Towards Triggering Conditions Guided Directed Fuzzing. When provided with the source code and the information that lines 5–6 may contain a use-of-uninitialized-variable vulnerability, a human expert can easily infer that the vulnerability is triggered when a poorly formatted `input.field` string (*e.g.*, `"192."`) fails to initialize at least two variables. In such cases, the value of `v` at line 5 is less than 2, indicating that the triggering condition of the vulnerability is $v < 2$. This presents an opportunity to leverage advanced LLMs to replicate this reasoning process performed by human experts.

Based on the inferred triggering condition, we can design a directed fuzzing strategy that prioritizes seeds yielding smaller `v` values during execution, as such seeds have more poorly formatted `input.field` strings, and are therefore more likely to trigger the vulnerability with fewer mutations. Moreover, the triggering condition can be further leveraged to identify the seed offsets corresponding to `input.field`. This is because only mutations applied to the relevant seed field can affect the value of `v`, whereas changes to other parts of the seed cannot. By monitoring variations in `v` before and after each mutation, we can infer the positions of the corresponding seed fields. Subsequent mutations can then focus on these positions to further reduce `v`, ultimately increasing the likelihood of triggering the target vulnerability.

3. Design

In this work, we leverage the code reasoning capabilities of LLMs to design a directed fuzzing approach guided by *triggering conditions* (hereinafter referred to as "TC"), and implement a prototype named TRIGFUZZ. Here, TC refers specifically to the vulnerability-triggering constraints that must be satisfied *after* the target code is reached, and does not encompass path constraints (*i.e.*, branch conditions along the execution path leading to the target). We present an overview of TRIGFUZZ in Figure 1. The workflow of TRIGFUZZ consists of three stages: *TC Generation*, *TC Validation*, and *TC-guided Directed Fuzzing*.

In the *TC Generation* stage, we first collect the source code and vulnerability reports associated with the target vulnerability. These artifacts serve as inputs to the LLM, guided by carefully crafted prompt templates. The LLM generates candidate TCs in a formalized *5-tuple* representation. In the *TC Validation* stage, we introduce a novel *dynamic triggering validation* technique to systematically filter and retain high-quality TCs from the LLM-generated candidates. In the final *TC-guided Directed Fuzzing* stage, we instrument the target program based on the validated TCs to monitor the *triggering distance* of inputs during fuzzing. The fuzzer leverages this feedback to perform vulnerability-specific seed scheduling and mutation, thereby accelerating the discovery of vulnerability-triggering inputs. We elaborate on the details of these three stages below.

3.1. TC Generation

To make the generated TCs informative, our design in this stage targets two objectives. First, we define a formal representation for TCs that can precisely capture various real-world triggering conditions. This formal representation enables us to conveniently quantify how close a given test input is to satisfying these TCs. Second, we design corresponding prompting strategies that enable the LLM to effectively generate TCs under this representation.

TC Formalization. The modeling of vulnerability triggering conditions has been widely studied. Prior work [13], [31] shows that, except for certain classes of vulnerabilities such as race conditions, the triggering conditions of most software memory vulnerabilities can be represented as logical constraints over program state (e.g., equalities, inequalities, and set memberships defined over program variables).

TABLE 1: Mapping Between Conditional Statements and Triggering Distance Expressions.

Conditional Statement	Triggering Distance Expression
$a < b$	$a - b$
$a <= b$	$a - b$
$a == b$	$\|a - b\|$
$a! = b$	0 for true, 1 for false
`ptr == NULL`	0 for true, 1 for false

Building upon this theoretical foundation and our empirical observations, we formalize the triggering conditions of target vulnerabilities as combinations of *Triggering Condition Units* (TCUs). Each TCU initially consists of three elements: a conditional statement (*cond*), its code location (*loc*), and its execution order (*seq*). For example, in the motivating example in Listing 1, a TCU can be represented as $\langle \mathrm{v} < 2$, line 6 in filename.c, executed first(*seq*=0)$\rangle$.

Next, we derive a triggering distance expression for each TCU based on its conditional statement to quantify how close the actual variable values are to satisfying the condition. The principle is that the triggering distance yields a positive value if the condition is not met and ≤ 0 if satisfied (except for strict inequality conditions, *e.g.*, $a < b$, where zero distance does not constitute satisfaction). The corresponding construction rules are summarized in TABLE 1, which we developed with reference to the work of Korel [32] and CAFL [13]. We further discuss the work of Korel in §7.

Certain special condition types may not be amenable to the rules in TABLE 1. In such cases, we leverage the code comprehension capabilities of LLMs to handle these conditions appropriately. For example, when a TC involves pointer comparisons (*e.g.*, `a.addr == b`), the corresponding rule in TABLE 1 yields a triggering distance equal to the raw numerical difference between the two pointer values. However, since distinct memory addresses correspond to distinct objects, a small pointer difference does not imply that the program state is closer to triggering the vulnerability. Thus, the triggering distance computed this way is semantically meaningless. This limitation has also been noted by CAFL [13]. The same issue arises for type checks (*e.g.*, `a.type == type1`, which admits only two states: matched or unmatched). To address these cases, we instruct the LLM to classify the type of each condition prior to distance computation, and for conditions involving pointer comparisons or type checks, the LLM assigns a binary distance of 0 or 1 in place of a raw numerical difference. Note that we try to avoid directly using conditions that correspond to binary distances and instead identify and utilize the preconditions that affect the relevant variables. We discuss this in detail in §5.1 and §6.

However, we recognize that some conditional statements combine multiple equalities and inequalities over program variables using logical conjunctions (AND) or disjunctions (OR). Such composite conditions cannot directly translate into a single triggering distance expression using the aforementioned rules. To resolve this, we decompose each complex conditional statement into multiple TCUs, each containing a single primitive condition (equality or inequality). We then introduce a conjunct element (*conj*) to annotate their logical relationships (AND/OR).

Definition 1. A *Triggering Condition Unit* (TCU) is defined as a 5-tuple:

$$\text{TCU} = \langle cond, loc, seq, conj, w \rangle$$

where:

- *cond*: conditional statement,
- *loc*: code location,
- *seq*: execution order,
- *conj*: conjunct identifier,
- *w*: weighting factor.

Specifically, we first convert the target conditional statement into its *Disjunctive Normal Form* (DNF) [33], formalized in Expression 1. Such a transformation is mathematically guaranteed. The resulting DNF consists of n conjuncts, denoted as $(\wedge c_j)_i$, each representing a set of AND operations between primitive sub-conditions c_j. Within this framework, the element *conj* serves as an identifier specifying the index of the conjunct i for each primitive condition c_j. For example, the expression $(a < b \;\&\&\; c == d)||(e + f < 1)$ can be transformed into three primitive conditions: $\langle a < b, conj = 0\rangle$, $\langle c == d, conj = 0\rangle$, and $\langle e + f < 1, conj = 1\rangle$.

$$\vee_{i=1}^{n}\left(\wedge c_j\right)_i \quad \begin{cases} n : \text{number of conjuncts} \\ c_j : \text{primitive conditional statement} \end{cases} \tag{1}$$

When computing the triggering distance for a complex conditional statement, we first compute it for each TCU separately. We then group and sum the expressions of TCUs sharing the same *conj*, aligning with the logical AND semantics. Finally, we take the minimum across all conjunct expressions, consistent with logical OR semantics.

We also observe that the values of TCU expressions may vary significantly, leading certain expressions to dominate the summed or minimal values and overshadow the contributions of others. Thus, we introduce a weight element (*w*) within each TCU, balancing the relative contributions of different expressions to the final triggering distance.

Ultimately, the triggering condition for a vulnerability is formalized as combinations of TCUs, represented as 5-tuples, as detailed in Definition 1.

Prompting Strategies. To derive the triggering conditions in the aforementioned representation, we prompt a pre-trained large language model (*e.g.*, GPT-5 [34]) using vulnerability reports and program source code. We employ progressive prompting [15] and the Few-shot Chain-of-Thought (CoT) [35], [36] technique to mitigate issues related to token limitations and hallucinations in LLMs.

We illustrate our prompt template in Figure 2. First, we embed a vulnerability report into the prompt. A typical

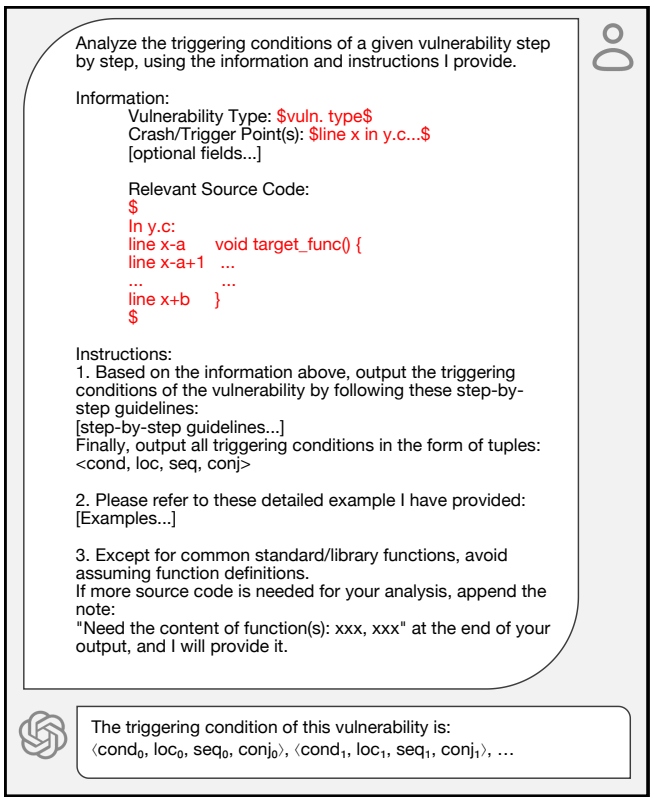

Figure 2: Prompt Template. Red placeholders $...$ are to be replaced with target-specific content, while [...] represents abbreviated content for brevity.

vulnerability report includes several mandatory fields and optional fields. The mandatory fields consist of the *vulnerability type* and the *code locations at which the vulnerability is triggered*. For vulnerabilities involving multiple trigger points, all relevant locations must be specified (*e.g.*, we require the code locations of both the free and use operations for a use-after-free vulnerability). In scenarios such as crash reproduction and candidate vulnerability confirmation, these mandatory fields are readily obtainable and constitute the basic information required by most existing directed fuzzers [6], [9], [13]. The optional fields include:

- A complete vulnerability call trace, typically obtained from ASan reports [37] and necessary for SDFuzz [2].
- A patch corresponding to the vulnerability, which is often available in patch testing scenarios.

Our design can work with only the mandatory fields, while the inclusion of optional fields may enhance the quality of the LLM's outputs, which we will evaluate in §5.3.

Next, we embed the source code of the program into the prompt. To reduce prompt noise [38] and accommodate LLMs' limited context windows [15], [39], we adopt a code slicing approach that selectively includes only the code segments relevant to the target vulnerability in the prompt, based on progressive prompting [15].

Specifically, we first embed the function(s) containing the vulnerability (hereafter referred to as the *vulnerable function(s)*) into the prompt. For multi-point vulnerabilities such as use-after-free, all involved functions are included. However, this is often insufficient, as many real-world vulnerabilities span multiple functions, where the semantics and value ranges of critical variables are determined by upstream functions or key information resides in callee functions.

Li et al. found that LLMs can progressively complete relevant code by being instructed within the prompt to request function definitions when needed, a method referred to as progressive prompting [15]. We apply this method in our task. Specifically, we build a dictionary containing all functions from the call trace observed during vulnerability triggering, which may be obtained from vulnerability reports or static analysis. Each entry maps a function name to its definition. We also include all callee functions of the vulnerable function(s) in the same way. During prompting, we instruct the model: "Except for widely used external library functions, do not assume the definitions of any other functions. If the current information is insufficient for your analysis, please list the required function names in the format: 'xxxx'." We build an agent that automatically retrieves the requested functions from the dictionary and provides them to the LLM in an interactive manner.

After embedding the vulnerability report and relevant program source code into the prompt, we further apply few-shot CoT prompting to guide the LLM in producing outputs in the desired format. Specifically, we provide two examples demonstrating a step-by-step reasoning process. The detailed examples used in this step are presented in Appendix A. The LLM is expected to follow this reasoning pattern and produce its final output in the same TCU-based format. At this stage, we instruct the LLM to generate all TCU components except the w element, which will be determined based on actual runtime observations during fuzzing.

3.2. TC Validation

In this step, we validate the TCs generated by the LLM and determine the w elements in the TCUs. We develop a *dynamic triggering validation* method for this purpose.

Although we observed that advanced models such as GPT-5 generally achieve high accuracy in generating TCs, they produce errors in individual attempts, as evidenced by our evaluation results in §5.3. Thus, we query the LLM three times and select the optimal TC through validation.

Dynamic Triggering Validation. All possible TCs can be categorized into three classes based on their relationship to vulnerability triggering: (1) the vulnerability may not be triggered even if the TC is satisfied (*i.e.*, the TC fails to constrain all relevant program variables, or the constrained value ranges are insufficient); (2) the vulnerability will be triggered if the TC is satisfied, but the corresponding program state is unreachable (*e.g.*, the TC satisfying values outside the feasible variable range); (3) the vulnerability will be triggered if the TC is satisfied, and the corresponding program state is reachable, which is the desirable result.

To constrain the generated TCs to fall within Class 3, static analysis–based validation is impractical. Even stat-

ically determining the reachability of target code, which is a necessary precondition for vulnerability triggering, is computationally expensive and lacks scalability [6], [11]. Moreover, statically determining whether a vulnerability is triggered, without actual execution, requires the correct vulnerability triggering conditions in the first place.

Therefore, we develop a dynamic validation approach that leverages concrete program inputs to verify the correctness of TCs. We instrument the PUT in two ways, run the instrumented binaries with reachable inputs produced by fuzzing, and monitor the program behavior.

The first instrumentation approach (referred to as *v1*) records each TCU's triggering distance at its corresponding loc and computes the final triggering distance for the input, as introduced in §3.3. This instrumented binary serves dual purposes: it is used for the first validation step and as the actual program binary that TRIGFUZZ fuzzes. If the triggering distance is observed to be ≤ 0 (< 0 for strict inequality conditions) when executing reachable inputs, this indicates that the TC can be satisfied by merely reachable inputs rather than vulnerability-triggering inputs (*i.e.*, the TC falls within Class 1), and thus the TC fails the first validation step. Meanwhile, we assign the maximum observed value for each TCU's output to its corresponding w element.

The second instrumentation approach (referred to as *v2*) directly modifies the PUT to forcibly satisfy the TC. The resulting instrumented binary is used solely for the second validation step. For each TCU, we insert an assignment statement at its corresponding loc that satisfies its corresponding $cond$ (*e.g.*, `a = b-1;` for $a < b$). We then replay a random reachable input and monitor whether the vulnerability manifestation consistent with the report is observed. If so, we consider the TC highly unlikely to belong to Class 1 and treat it as successfully validated. We employ necessary sanitizers (*e.g.*, ASan [37]) to assist in detecting vulnerability behavior.

The first validation step relies on actual fuzz inputs that may not cover all boundary cases. TCs passing this step may still belong to Class 1. However, TCs failing this step definitely belong to Class 1. Thus, the first validation step serves as an initial filter to eliminate clearly incorrect TCs. The second validation step modifies the program's original logic. Its results may be unreliable. TCs failing this step may still be correct. Thus, the second validation step primarily serves to identify TCs of demonstrably higher quality.

Three-Phase Directed Fuzzing. While TC validation requires reachable inputs produced by fuzzing, we do not follow a stop-validate-resume pattern where fuzzing pauses after obtaining reachable inputs for validation. Instead, we employ a *Three-Phase Directed Fuzzing* approach that enables continuous fuzzing with online TC validation.

This process is illustrated in Figure 3. Our three candidate TCs are TC1, TC2, and TC3. Before fuzzing begins, we build three instrumented binaries. The first is a *v1*-instrumented binary that monitors all three candidate TCs simultaneously, incorporating their instrumentation to avoid repeated compilation. The other two are *v2*-instrumented binaries corresponding to TC1 and TC2, respectively. We

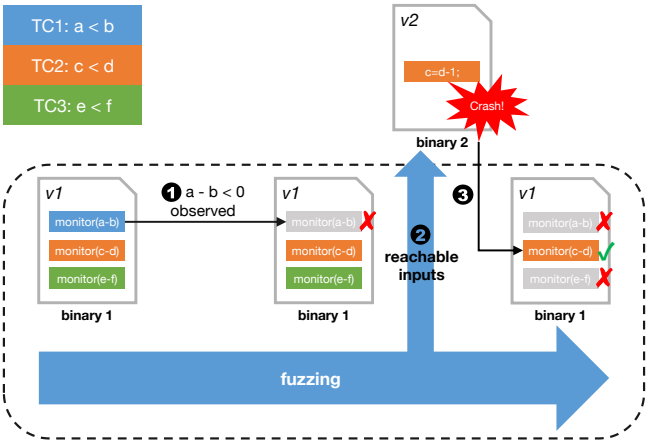

Figure 3: Three-phase directed fuzzing.

begin by fuzzing the *v1* binary. In Phase 1 (*pioneering phase*), we focus on reaching the target code and generating reachable inputs. Once reachable inputs are produced, we enter Phase 2 (*validation phase*), where we continuously monitor runtime feedback to perform the first validation step. Concurrently, we randomly select reachable inputs and replay them on the two *v2* binaries to perform the second validation step. We then select the TC that passes the most validation checks and notify the fuzzer via shared memory. The fuzzer enters Phase 3 (*exploitation phase*), disabling monitoring for the remaining TCs and continuing to fuzz exclusively with the validated TC. When multiple TCs pass an equal number of validation checks, we randomly select one. This tie-breaking mechanism explains why we only need two *v2*-instrumented binaries instead of three.

We do not further verify whether the TC can actually be satisfied (*i.e.*, whether it belongs to Class 2). This is because dynamically verifying this essentially reduces to generating a triggerable and satisfiable input (*i.e.*, a PoC). Since producing such a PoC is the ultimate goal of TRIGFUZZ, performing this verification would be logically circular.

3.3. TC-guided Directed Fuzzing

After obtaining the LLM-generated and validated TC, we leverage it to guide directed fuzzing. Specifically, we fuzz the *v1* binary and collect the *triggering distance* as runtime feedback during fuzzing. Based on this feedback, we develop a novel seed scheduling strategy that prioritizes inputs more likely to satisfy the TC. Furthermore, during fuzzing, we monitor variations in triggering distance across inputs to identify input bytes that are likely to influence the satisfaction of the TC and selectively mutate them.

Triggering Distance as Runtime Feedback. At fuzzing runtime, for each TCU, we collect the value of the expression corresponding to $cond$ when the program execution reaches the associated code location loc. Since the same loc may be reached multiple times during the execution of a single input, we retain only the minimum observed value for each TCU. This design choice is motivated by our

Algorithm 1: Triggering distance calculation.

Input: TCU Runtime Feedback Set T, Sequence Set S, Conjunct Set C
Output: Final Triggering Distance d

$d \leftarrow 0$;
foreach $s \in S, c \in C$ **do**
  // Initialize with predefined maximum distance
  $d(s, c) \leftarrow$ *DISTANCE_MAX*;
foreach $tcu \in T$ **do**
  $s \leftarrow tcu.seq$;
  $c \leftarrow tcu.conj$;
  $d\_norm \leftarrow tcu.d/tcu.w$;
  if $d(s, c) =$ *DISTANCE_MAX* **then**
    $d(s, c) \leftarrow d\_norm$;
  else
    $d(s, c) \leftarrow d(s, c) + d\_norm$;
foreach $s \in S$ **do**
  // Traverse s in order
  $d(s) \leftarrow \min_c d(s, c)$;
  // Aggregate by OR: take minimum over all conjuncts
  $d \leftarrow d + d(s)$;
  if $d(s) =$ *DISTANCE_MAX* **then**
    // If any s is not reached, skip all remaining ones
    $n \leftarrow$ number of s left in S;
    $d \leftarrow d + n \times$ *DISTANCE_MAX*;
    break;
return d;

observation that, in vulnerabilities such as buffer overflows, repeated accesses to the same buffer are common (*e.g.*, within a `for` loop). In such cases, only the access closest to the buffer boundary is relevant, which corresponds to the smallest triggering distance recorded for that TCU.

We then hierarchically aggregate all TCU triggering distances based on their *seq* and *conj* attributes to compute the final triggering distance for the given input, as described in Algorithm 1. We begin by normalizing the expression value of each collected TCU using its corresponding w element. We then group the TCUs by their seq field, such that all TCUs within a group share the same seq value. For each group, we compute the triggering distance based on the $conj$ values of its TCUs, following the method described earlier in §3.1. Finally, we compute the overall triggering distance by cumulatively summing the triggering distances of all groups in sequential order, starting from index 0 (*i.e.*, the point that must be reached first, such as the "free" site in a use-after-free vulnerability). If the target corresponding to a given sequence index is not reached, we assign a predefined maximum distance to all remaining indices, regardless of whether they are actually reached, to indicate that their preconditions have not been satisfied.

We extend the fuzzer's shared memory [40], which collects runtime feedback like coverage information, by adding a small number of state bits for tracking execution order. When a TCU target is reached, the corresponding state bit for its sequence index is updated only if all preceding sequence bits have already been set. This ensures that the tracked order accurately reflects the real execution sequence. For instance, in a use-after-free vulnerability, even if both the free and use sites are covered, our mechanism can distinguish which one was reached first, while CAFL cannot.

Seed Scheduling Guided by Triggering Distance. We introduce an additional criterion for seed creation. In addition to the traditional coverage-guided heuristic of retaining inputs that discover new execution paths [40], we also preserve inputs with smaller triggering distances. This design choice is motivated by our observation that, while inputs with smaller triggering distances are often more likely to produce vulnerability-triggering variants through mutation, they may not increase code coverage and would therefore be discarded by prior fuzzers.

We further allocate more energy during power scheduling to seeds with smaller triggering distances, following the annealing-based power schedule [6] used in AFLGo. Our implementation builds on the AFLGo framework and inherits its control-flow-based distance metric. We compute the energy contribution from each distance metric independently and sum the results to determine the total energy assigned to the input.

Triggering Byte-Aware Input Mutation. A smaller triggering distance does not always indicate a higher potential for generating vulnerability-triggering inputs. For example, consider a vulnerability that is triggered only when a specific input field satisfies a condition such as `input.field == 0`. Suppose we have two inputs, $\texttt{input}_1\texttt{.field = 1}$ and $\texttt{input}_2\texttt{.field = 2}$, whose triggering distances would typically be computed as $\text{TD}_1 = 1$ and $\text{TD}_2 = 2$, respectively. In this case, both inputs require the same mutation, changing the field value to 0, to trigger the vulnerability. The effort required is therefore equivalent, and the triggering distance does not accurately reflect the actual difficulty of reaching the vulnerable state. Thus, seed scheduling guided by triggering distance alone is not sufficient.

We observe that, unlike the control-flow distance introduced in AFLGo, where the influencing input bytes may vary depending on which program constraints the execution is blocked by, the input bytes that affect the triggering distance tend to remain stable. These bytes typically correspond to input fields associated with variables in the $cond$ elements of TCUs. As a result, when the triggering distance of an input changes after a mutation, it often indicates that the mutation has successfully modified the critical input bytes influencing the triggering distance. This observation naturally suggests a strategy in which we record such mutations and systematically apply similar mutations at the same positions, until we obtain an input whose triggering distance is zero and that successfully triggers the vulnerability.

Building on this insight, we develop a technique called *triggering byte-aware input mutation* to identify and mutate critical input bytes for triggering vulnerabilities. Specifically, for mutations in the deterministic stage [40], we directly record the position of the mutation. In the havoc stage [40], where a single iteration may involve multiple atomic mutations, we identify mutation-relevant positions by comparing the byte-level differences between the original

```
--- filename.c
+++ filename.c
@@ ... @@
+ #include "distance.h"
+
+ distance_instrument(
+     double distance = 0.0 + buf_len - a,
+     uint8_t exec_sequence = 0,
+     uint8_t conjunct = 0,
+     double weight = 1.0
+ );
  visit(buf[a]);
```

Listing 2: Example of an LLM-generated unified edit patch.

and mutated inputs.

We then apply a probing mutation at each identified position, which includes one value-change operation and two length-change operations (*i.e.*, inserting a random byte and deleting one byte). If the triggering distance changes after a probing mutation, we proceed to systematically mutate that position according to the type of mutation that caused the change. For value-changing mutations, we further explore the full byte range (0–255) and also apply value mutations to nearby multi-byte fields (*e.g.*, 2-byte and 4-byte values). For length-changing mutations, we progressively increase the length of insertions and deletions.

4. Implementation

We use GPT-5 [34] as the primary pre-trained LLM in our system. In §5.3, we also evaluate the effectiveness of alternative models on our task. We automate TC generation and validation by building an agent to interact with the LLM. The agent is implemented in Python and shell scripts, while the fuzzing components are implemented in C. Our fuzzing strategies are built on top of the AFLGo framework [6]. We will maintain our code at https://github.com/vul337/TrigFuzz.

TC Instrumentation. We instrument the PUT in two ways (*v1* and *v2*, as described in §3.2). Both ways of instrumentation are implemented by directly inserting code into the PUT's source code. To automate this process, we adopt the `udiff` edit format from the AI pair programming tool *aider* [41], which enables an LLM to modify source code to insert the required instrumentation logic. Listing 2 provides a simple example. In our experiments, all instrumented binaries modified this way compiled and executed correctly.

5. Evaluation

In this section, we evaluate TRIGFUZZ and answer the following research questions:

- **RQ1:** How efficiently can TRIGFUZZ trigger known vulnerabilities compared to existing directed fuzzers? (§5.1)
- **RQ2:** How does each component contribute to the performance of TRIGFUZZ? (§5.2)
- **RQ3:** How effectively can TRIGFUZZ generate triggering conditions? (§5.3)
- **RQ4:** Is TRIGFUZZ's approach complementary to existing directed fuzzers that focus on reaching target code? (§5.4)

Baselines. We compare TRIGFUZZ with four state-of-the-art directed fuzzers: AFLGo [6], SelectFuzz [9], CAFL [13], and SDFuzz [2]. TRIGFUZZ builds upon AFLGo. SelectFuzz represents an advanced approach focused on reaching target code locations, while CAFL and SDFuzz specifically aim to trigger known vulnerabilities, making them the most relevant baselines for our evaluation. Due to the unavailability of CAFL's prototype, we implemented it ourselves following the paper's description. CAFL uses three templates supporting seven bug types [13], all of which can be perfectly expressed within our triggering condition framework. We faithfully replicated CAFL's other components in the AFL [40] framework as specified in the original paper. All other baselines are open-source. We do not include other directed fuzzers in our comparison because they focus solely on reaching target code locations without subsequent strategies for vulnerability triggering, and most of them are not open-source (*e.g.*, Halo [11]). We have selected SelectFuzz, one of the most advanced fuzzers in this category, as a representative baseline. The "-d" option was enabled for all fuzzing campaigns.

Benchmark. We evaluate our approach using Magma [12], a state-of-the-art fuzzing benchmark widely adopted by previous directed fuzzing studies [11], [42], [43]. Magma consists of open-source libraries with widespread real-world usage and well-documented security-critical bugs. For each bug, Magma provides inline source-code-level instrumentation to collect ground-truth information on two metrics: bug reachability (whether buggy code is executed) and bug triggering (whether the fault condition is satisfied by the input). This instrumentation enables real-time monitoring of fuzzer progress. Based on Magma's triggering detection instrumentation, we establish ground-truth triggering conditions for each target bug.

We use the seeds provided in Magma as the initial seeds for fuzzing. All experiments were conducted 10 times with a time budget of 24 hours on a 64-bit Ubuntu 22.04 LTS server equipped with Intel(R) Xeon(R) Silver 4214R CPU @ 2.40GHz*48 and 128 GB of RAM.

5.1. Efficiency in Triggering Known Vulnerabilities

Vulnerability reproduction is a typical application scenario for directed fuzzing. We answer RQ1 by comparing the time-to-reach and time-to-trigger of TRIGFUZZ and baseline directed fuzzers on the Magma benchmark. We calculated the average time-to-reach and time-to-trigger for each vulnerability and applied the Mann-Whitney U test [44] and the Vargha-Delaney $\hat{A}_{12}$ effect size measure [45] to compare the results between baseline fuzzers and TRIGFUZZ. When computing improvement ratios, p-values, and $\hat{A}_{12}$ effect sizes for cases involving a timeout, we treat the timeout as 86,400 seconds (24 hours) and append a ">" symbol to the corresponding $Ratio$ value to indicate that the actual improvement may be even greater.

TABLE 2: Reproduction time for each target in Magma across different directed fuzzers. R and T indicate the average time-to-reach and time-to-trigger (in seconds), respectively. "T.O." indicates that the fuzzer cannot reproduce the target within 24 hours. $Ratio$, p, and $\hat{A}_{12}$ indicate the improvement ratio, p-value, and Vargha-Delaney effect size compared with TRIGFUZZ. "-" in the p column indicates that TRIGFUZZ underperforms the baseline fuzzer on the corresponding target. "N/A" in the Avg. p column indicates that p-values are not averaged across targets, as "-"s cannot be meaningfully combined. $\emptyset$ indicates that CAFL's templates do not support this vulnerability type or that SDFuzz fails to build this target. "#T" denotes the number of targets successfully triggered by each fuzzer.

Bug ID	TRIGFUZZ		AFLGo				SelectFuzz				CAFL				SDFuzz			
	R	T	R	T	$Ratio$	p / $\hat{A}_{12}$	R	T	$Ratio$	p / $\hat{A}_{12}$	R	T	$Ratio$	p / $\hat{A}_{12}$	R	T	$Ratio$	p / $\hat{A}_{12}$
PNG001	3	33610	3	T.O.	>2.57x	<0.01/ 0.99	3	T.O.	>2.57x	<0.01/ 0.99	2	T.O.	>2.57x	<0.01/ 0.99	3	T.O.	>2.57x	<0.01/ 0.99
PNG003	2	3	2	3	1.00x	1.00 / 0.50	2	3	1.00x	1.00 / 0.50	2	3	1.00x	1.00 / 0.50	2	3	1.00x	1.00 / 0.50
PNG007	3	901	3	765	0.85x	- / 0.46	3	695	0.77x	- / 0.43			$\emptyset$		3	9032	10.02x	<0.01/ 0.99
SND001	3	187	3	171	0.91x	- / 0.49	3	204	1.09x	0.16 / 0.83	3	281	1.50x	0.05 / 0.90	3	32	0.17x	- / 0.17
SND005	2	19	2	40	2.11x	0.02 / 0.97	2	37	1.95x	0.02 / 0.96	2	25	1.32x	0.09 / 0.89	2	97	5.11x	0.02 / 0.98
SND006	3	308	4	417	1.35x	0.04 / 0.96	4	321	1.04x	0.17 / 0.84	3	350	1.14x	0.15 / 0.83	T.O.	T.O.	>280.52x	<0.01/ 1.00
SND017	834	898	717	1081	1.20x	0.10 / 0.89	801	1212	1.35x	0.07 / 0.88	908	1304	1.45x	0.06 / 0.90	1072	1373	1.53x	0.08 / 0.85
SND020	2316	2320	2676	2735	1.18x	0.11 / 0.88	2675	2700	1.16x	0.12 / 0.85			$\emptyset$		2857	2875	1.24x	0.14 / 0.71
TIF002	13807	13822	19432	19483	1.41x	0.04 / 0.92	12976	16850	1.22x	0.10 / 0.88	32205	32235	2.33x	<0.01/ 0.99	T.O.	T.O.	>6.25x	<0.01/ 0.99
TIF006	18272	18272	39175	39175	2.14x	<0.01/ 0.99	13260	13260	0.73x	- / 0.35	42114	42114	2.30x	<0.01/ 0.99	29239	29239	1.60x	0.04 / 0.85
TIF007	8	28	9	38	1.36x	0.06 / 0.90	5	47	1.68x	0.02 / 0.96	6	25	0.89x	- / 0.47			$\emptyset$	
TIF008	66626	66656	17060	17161	0.26x	- / 0.30	23378	23425	0.35x	- / 0.21	12306	12316	0.18x	- / 0.13	T.O.	T.O.	>1.30x	0.09 / 0.80
TIF009	35922	35939	58250	58277	1.62x	0.03 / 0.94	19323	19400	0.54x	- / 0.36			$\emptyset$		13902	13977	0.39x	- / 0.36
TIF012	2	119	2	551	4.63x	<0.01/ 0.99	2	732	6.15x	<0.01/ 1.00	1	415	3.49x	<0.01/ 0.98	2	923	7.76x	<0.01/ 0.99
TIF014	8	113	8	123	1.09x	0.15 / 0.88	7	115	1.02x	0.28 / 0.80			$\emptyset$		4	151	1.34x	0.23 / 0.71
XML003	3	10454	3	2426	0.23x	- / 0.29	3	2001	0.19x	- / 0.18			$\emptyset$		3	6975	0.67x	- / 0.43
XML009	3	348	3	609	1.75x	0.02 / 0.96	3	417	1.20x	0.11 / 0.86	3	481	1.38x	0.07 / 0.90	3	1570	4.51x	0.13 / 0.98
XML014	2	9	2	11	1.22x	0.11 / 0.86	2	11	1.22x	0.11 / 0.87	1	15	1.67x	0.03 / 0.94	2	11	1.22x	0.12 / 0.70
XML017	3	11	3	11	1.00x	1.00 / 0.50	3	11	1.00x	1.00 / 0.50	2	15	1.36x	0.08 / 0.90	3	8	0.73x	- / 0.49
PDF003	10	4014	18	7786	1.94x	0.01 / 0.97	13	7086	1.77x	0.02 / 0.98	23	7615	1.90x	0.02 / 0.95			$\emptyset$	
PDF010	4	434	5	332	0.76x	- / 0.48	3	295	0.68x	- / 0.44			$\emptyset$				$\emptyset$	
PDF016	2	47	2	154	3.28x	<0.01/ 0.98	2	188	4.00x	<0.01/ 0.99	2	105	2.23x	<0.01/ 1.00	2	63	1.34x	<0.01/ 0.84
SSL001	80	3678	85	T.O.	>23.49x	<0.01/ 0.99	77	T.O.	>23.49x	<0.01/ 0.99	84	65302	17.75x	<0.01/ 0.98			$\emptyset$	
SSL002	3	342	3	291	0.85x	- / 0.47	3	290	0.85x	- / 0.46	2	293	0.86x	- / 0.44	3	200	0.58x	- / 0.40
SSL009	3	1367	3	41712	30.51x	<0.01/ 1.00	3	29692	21.72x	<0.01/ 0.99	2	20605	15.07x	<0.01/ 1.00	3	9801	7.17x	<0.01/ 0.98
SSL020	3	35811	3	T.O.	>2.41x	<0.01/ 0.99	3	80571	2.25x	<0.01/ 0.97	6	43699	1.22x	0.12 / 0.86	4	13973	0.39x	- / 0.37
SQL013	14798	31589	8690	18827	0.60x	- / 0.43	7774	21530	0.68x	- / 0.49	14563	T.O.	>2.74x	<0.01/ 1.00	T.O.	T.O.	>2.74x	<0.01/ 1.00
PHP004	3	593	3	461	0.78x	- / 0.49	3	410	0.69x	- / 0.41	2	1213	2.05x	0.01 / 0.97			$\emptyset$	
PHP009	3	184	3	4753	25.83x	<0.01/ 1.00	3	4004	21.76x	<0.01/ 0.99	2	467	2.54x	<0.01/ 0.98			$\emptyset$	
PHP011	2	11	2	11	1.00x	1.00 / 0.50	2	11	1.00x	1.00 / 0.50	1	30	2.73x	<0.01/ 0.97			$\emptyset$	
Avg.					**>1.64x**	**N/A / 0.77**			**>1.49x**	**N/A / 0.72**			**>1.88x**	**N/A / 0.85**			**>2.02x**	**N/A / 0.74**
#T		**30**		**27**				**28**				**22**				**18**		

We present the results for all targets triggered by at least one fuzzer in TABLE 2.

Overall, TRIGFUZZ achieves average speedups of 1.64x, 1.49x, 1.88x, and 2.02x over AFLGo, SelectFuzz, CAFL, and SDFuzz, respectively, in terms of time-to-trigger. Moreover, the average $\hat{A}_{12}$ values across all comparisons exceed 0.71, indicating a large effect size according to the threshold established by Vargha and Delaney [45], which further confirms the practical significance of TRIGFUZZ's advantage. We observe that TRIGFUZZ demonstrates no significant advantage in reaching the target code, with speedups of 1.03x, 0.91x, and 0.94x compared to AFLGo, SelectFuzz, and CAFL, respectively. This aligns with the strategic focus of these directed fuzzers, as TRIGFUZZ and CAFL employ identical strategies for reaching target code as AFLGo, while SelectFuzz is a more advanced fuzzer specifically designed to prioritize reaching target code. However, TRIGFUZZ demonstrates significant advantages in reducing the gap between time-to-reach and time-to-trigger, achieving speedups of 2.08x, 2.31x, and 2.06x compared to AFLGo, SelectFuzz, and CAFL, respectively. We did not include SDFuzz in these comparisons because SDFuzz fails to reach the target code even on several easy targets (*e.g.*, SND006), which would yield disproportionately large and unreliable ratio values for SDFuzz. This is discussed further below.

We also observe that compared to directed fuzzers that focus on reaching target code (*i.e.*, AFLGo and SelectFuzz), TRIGFUZZ does not accelerate vulnerability triggering on some targets (*i.e.*, PNG007, PDF010, SSL002, SQL013, PHP004). We find that this is generally attributed to two reasons. First, the generated triggering conditions are incorrect, naturally failing to guide vulnerability triggering, as exemplified by SQL013. Second, some triggering conditions exhibit a binary nature (0 or 1) instead of continuous variation. For instance, the triggering condition for PNG007 is `png_ptr->palette == NULL`. TRIGFUZZ correctly generates this condition, but the triggering distance derived from such a condition remains constant at 1, where any reduction immediately triggers the vulnerability. This fails to provide effective guidance for seed scheduling and mutation.

Further comparing the results of TRIGFUZZ and CAFL, we find that except for the 6 targets where CAFL's templates do not support the vulnerability types, TRIGFUZZ demonstrates superior performance on most targets. This is because CAFL's templates, designed based on vulnerability types, cannot account for code semantics and often fail to accurately describe the actual triggering conditions of vulnerabilities. For example, PNG001 is a divide-by-zero vulnerability whose actual triggering condition involves the divisor `row_factor` overflowing due to excessive magnitude: `row_factor == ((size_t)1 <<(sizeof(png_uint_32)*8))`. This results in a value of zero. Therefore, larger values of `row_factor` are more conducive to triggering the vulnerability. However, CAFL's template for divide-by-zero vulnerabilities assumes that the divisor should approach zero, which is contrary

TABLE 3: Reproduction time for each target in Magma across different variants of TRIGFUZZ. The symbols in the table have the same meaning as those in TABLE 2.

Bug ID	TRIGFUZZ-s				TRIGFUZZ-m			
	R	T	$Ratio$	p / $\hat{A}_{12}$	R	T	$Ratio$	p / $\hat{A}_{12}$
PNG001	3	T.O.	>2.57x	<0.01/0.99	3	T.O.	>2.57x	<0.01/0.99
PNG003	2	3	1.00x	1.00 / 0.50	2	3	1.00x	1.00 / 0.50
PNG007	3	873	0.97x	- / 0.44	3	747	0.83x	- / 0.43
SND001	3	189	1.01x	0.98 / 0.59	3	203	1.09x	0.88 / 0.60
SND005	2	20	1.05x	0.92 / 0.60	2	25	1.32x	0.61 / 0.65
SND006	3	351	1.14x	0.81 / 0.61	3	357	1.16x	0.78 / 0.62
SND017	900	995	1.11x	0.85 / 0.61	1011	1077	1.20x	0.73 / 0.63
SND020	2355	2369	1.02x	0.97 / 0.59	2316	2410	1.04x	0.94 / 0.59
TIF002	13103	14338	1.04x	0.94 / 0.59	18298	18358	1.33x	0.59 / 0.65
TIF006	24817	24817	1.36x	0.57 / 0.65	30594	30594	1.67x	0.34 / 0.70
TIF007	7	33	1.18x	0.76 / 0.62	6	22	0.79x	- / 0.44
TIF008	43203	43291	0.65x	- / 0.48	T.O.	T.O.	>1.30x	0.09 / 0.80
TIF009	23609	23679	0.66x	- / 0.35	26630	26684	0.74x	- / 0.41
TIF012	2	158	1.33x	0.59 / 0.65	2	333	2.80x	0.07 / 0.79
TIF014	7	189	1.67x	0.34 / 0.70	7	845	7.48x	<0.01/0.87
XML003	3	18136	1.73x	0.31 / 0.70	3	8162	0.78x	- / 0.47
XML009	3	411	1.18x	0.75 / 0.62	3	410	1.18x	0.76 / 0.62
XML014	2	9	1.00x	1.00 / 0.50	2	13	1.44x	0.49 / 0.67
XML017	3	13	1.18x	0.75 / 0.62	3	13	1.18x	0.75 / 0.62
PDF003	9	7109	1.77x	0.29 / 0.71	10	7082	1.76x	0.30 / 0.71
PDF010	6	601	1.38x	0.54 / 0.66	5	435	1.00x	1.00 / 0.59
PDF016	2	96	2.04x	0.20 / 0.74	2	98	2.09x	0.18 / 0.74
SSL001	82	62500	16.99x	<0.01/0.87	80	25889	7.04x	<0.01/0.87
SSL002	3	321	0.94x	- / 0.47	3	317	0.93x	- / 0.49
SSL009	3	2928	2.14x	0.17 / 0.74	3	7807	5.71x	<0.01/0.87
SSL020	3	64072	1.79x	0.28 / 0.71	3	T.O.	>2.41x	<0.01/0.99
SQL013	20681	64381	2.04x	0.20 / 0.74	15204	T.O.	>2.74x	<0.01/1.00
PHP004	3	212	0.36x	- / 0.19	3	868	1.46x	0.48 / 0.67
PHP009	3	444	2.41x	0.12 / 0.77	3	495	2.69x	0.08 / 0.79
PHP011	2	12	1.09x	0.87 / 0.60	2	11	1.00x	1.00 / 0.50
Avg.			**>1.35x**	**N/A / 0.62**			**>1.58x**	**N/A / 0.67**
#T		29				26		

to the actual triggering condition. As another example, the target PDF016 is an out-of-bounds read vulnerability. This vulnerability is triggered when creating an indirect object reference with invalid `num` or `gen` values and then parsing this indirect reference. Therefore, its core triggering condition is that the `num` or `gen` values are invalid: `num <= 0 || gen < 0`, rather than the distance between the read position and the bound, which is what CAFL's template focuses on. In addition, whereas CAFL only optimizes seed scheduling strategies, we design dedicated seed mutation strategies for TRIGFUZZ to effectively leverage the triggering conditions.

Comparing the results of TRIGFUZZ and SDFuzz, we find that TRIGFUZZ demonstrates superior overall performance and has broader applicability in implementation. In terms of implementation, SDFuzz checks the call stack after functions in the vulnerability call trace are invoked, executing `exit(0)` to save runtime when it determines that reaching the target call trace is impossible. This strategy is effective on some targets, such as SSL020. However, in rare cases, SDFuzz may erroneously terminate reachable call stacks, making the target vulnerability more difficult to trigger and even preventing the target code from being reached, as exemplified by SND006.

5.2. Ablation Study

TRIGFUZZ leverages LLM-generated triggering conditions to guide fuzzing. The strategies that directly impact TRIGFUZZ's performance include *seed scheduling guided by triggering distance* and *triggering byte-aware input mutation*. Therefore, we conducted ablation experiments on these two techniques to investigate their contributions to TRIGFUZZ's effectiveness in answering RQ2.

We evaluated different variants of TRIGFUZZ on the Magma benchmark, including: the full version of TRIGFUZZ, TRIGFUZZ without *triggering byte-aware input mutation* (TRIGFUZZ-s), TRIGFUZZ without *seed scheduling guided by triggering distance* (TRIGFUZZ-m), and TRIGFUZZ without both components, which is equivalent to AFLGo. The results are presented in TABLE 2 and TABLE 3. Comparing the average ratios, we observe that both TRIGFUZZ-s and TRIGFUZZ-m perform worse than TRIGFUZZ but better than the baseline AFLGo, which indicates that although these two strategies are individually effective, they need to work together to achieve TRIGFUZZ's optimal performance.

We also observed that in a few cases, applying a single strategy individually yielded better results than combining them, as seen in TIF007 and PHP004. This can be attributed to two reasons. First, our seed scheduling strategy allocates more energy to seeds with smaller triggering distances; however, a smaller triggering distance does not necessarily imply that fewer mutation operations are required to trigger a vulnerability, as explained in §3.3. Second, our mutation strategy incorporates systematic mutations on critical fields in addition to the conventional mutations of the AFLGo framework, which introduces a trade-off between mutation effectiveness and computational overhead.

5.3. Quality of Generated Triggering Conditions

The quality of generated triggering conditions significantly impacts TRIGFUZZ's performance. To address RQ3, we evaluate how design choices in TC generation and validation influence the quality of generated triggering conditions.

Specifically, we evaluate the impact on triggering condition quality of: (1) removing optional fields (*i.e.*, vulnerability call traces and patches) from the prompt, which is necessary because this information may not be available in practice, (2) excluding TC validation from the workflow, and (3) using different base LLMs. The base LLMs we include are three of the most advanced and representative models currently available: GPT-5 [34], Claude Sonnet 4.5 [46], and DeepSeek-R1 [47]. We also include a less advanced model, GPT-4 [48], to more intuitively evaluate the impact of model capability on triggering condition quality. The prompting techniques employed in TC generation (*e.g.*, progressive prompting, few-shot CoT) are not evaluated, as these are well-established techniques validated in prior work [17] rather than contributions of this paper.

We assess the quality of generated triggering conditions using the following criteria:

- Perfect (●): The generated triggering conditions fully cover the ground-truth triggering conditions recorded in Magma, and any additional generated conditions do not contradict the ground truth. This is scored as 1 point.
- Partial (◐): The generated triggering conditions either (1) partially cover the ground-truth triggering conditions, or (2) fully cover the ground truth but include

TABLE 4: Quality assessment of generated triggering conditions.

Bug ID	TRIGFUZZ	*wo*	*wv*	*claude*	*deepseek*	*gpt4*
PNG001	◐	◐	◐	◐	○	○
PNG003	●	●	○	●	●	●
PNG007	●	◐	◐	●	●	●
SND001	◐	◐	◐	◐	◐	○
SND005	●	●	●	●	●	●
SND006	●	◐	◐	●	●	○
SND017	●	●	◐	●	●	●
SND020	●	●	●	●	●	◐
TIF002	●	●	●	●	●	◐
TIF006	◐	○	◐	○	○	○
TIF007	◐	◐	◐	◐	◐	◐
TIF008	◐	○	◐	◐	●	○
TIF009	◐	◐	◐	●	●	●
TIF012	◐	◐	◐	◐	○	○
TIF014	◐	◐	◐	●	○	●
XML003	●	◐	●	●	●	●
XML009	●	○	●	●	●	●
XML014	●	●	●	●	●	◐
XML017	●	○	●	●	●	●
PDF003	●	●	●	●	●	○
PDF010	◐	◐	◐	◐	◐	●
PDF016	●	○	●	●	●	○
SSL001	◐	◐	◐	◐	◐	◐
SSL002	◐	◐	◐	◐	◐	◐
SSL009	●	●	◐	●	●	◐
SSL020	●	●	○	●	●	◐
SQL013	○	○	○	○	○	○
PHP004	●	○	●	●	◐	○
PHP009	●	●	◐	●	◐	○
PHP011	◐	○	◐	○	◐	○
Total	23	16	18.5	23	21	14
Cost	$5.74	$5.81	$4.98	$8.55	$1.13	$36.52
Time	134m	116m	97m	34m	210m	37m

extraneous conditions that may contradict it. This is scored as 0.5 points.

- Insufficient (○): The generated triggering conditions are incorrect, either syntactically or semantically. This is scored as 0 points.

Triggering conditions generated by the model may be semantically equivalent to the ground truth despite differences in syntactic representation, complicating accurate evaluation. To address this challenge, we employ manual expert assessment. Two security researchers with extensive experience in vulnerability analysis independently examine and score each generated triggering condition, followed by cross-validation to resolve discrepancies. We also record the cost and time of LLM queries.

We evaluate the triggering conditions generated by six variants of TRIGFUZZ: (1) the full version of TRIGFUZZ, (2) TRIGFUZZ with prompts excluding vulnerability call traces and patches ("*wo*"), (3) TRIGFUZZ without TC validation ("*wv*"), (4) TRIGFUZZ based on Claude Sonnet 4.5 ("*claude*"), (5) TRIGFUZZ based on DeepSeek-R1 ("*deepseek*"), and (6) TRIGFUZZ based on GPT-4 ("*gpt4*").

The results are shown in TABLE 4. Overall, TRIGFUZZ successfully generates effective triggering conditions for 29 out of 30 targets, achieving a 96.67% success rate, among which 17 targets achieve complete coverage of the ground truth conditions recorded in Magma. These findings provide compelling evidence in support of the aforementioned experimental results demonstrating TRIGFUZZ's capability to rapidly trigger target vulnerabilities. We also find that the quality of generated triggering conditions is not strongly associated with vulnerability types, but rather exhibits a strong correlation with the program itself. For example, `libtiff` has complex program states and contains numerous intricate variables. Thus, targets with the TIF prefix exhibit generally poor triggering condition generation quality.

Comparing TRIGFUZZ with TRIGFUZZ-*wo* reveals that incorporating vulnerability call traces and patches into prompts significantly improves triggering condition quality for 10 targets. For PNG007, SND006, and XML003, prompts containing only vulnerability type and code location can generate partially valid triggering conditions. In these cases, we observe that the LLM has already acquired the necessary variable semantics and function definitions, but lacks understanding of the vulnerability. In such situations, adding call traces has minimal impact, while including patches enables the LLM to effectively capture critical triggering conditions. For the remaining 7 affected targets, incorporating the vulnerability call trace and patch enables the LLM to generate valid triggering conditions from scratch. We find that this situation typically arises when the LLM has not obtained sufficient variable semantics and function definitions. In such cases, the call trace and patch are equally important: the former effectively guides the LLM to locate critical function definitions through progressive prompting, while the latter helps the LLM identify overlooked variables and their conditions.

> **Finding 1.** While the call trace and patch of a vulnerability can offer significant guidance for LLMs to generate triggering conditions, the vulnerability type and code location information alone can guarantee a baseline quality of the generated triggering conditions.

Comparing TRIGFUZZ with TRIGFUZZ-*wv*, we find that TC validation improves the quality of generated triggering conditions across 7 targets. This occurs because single-query LLM outputs may be unstable, exhibiting syntax errors or missing critical conditions. TC validation detects such errors and filters out high-quality TCs from multiple LLM outputs. Our results show that TRIGFUZZ requires an average of 1.4 queries to generate the finally adopted TC.

> **Finding 2.** The quality of triggering conditions generated by a single LLM query may be unstable. TC validation can help filter out high-quality triggering conditions from multiple queries.

Comparison across base LLMs shows that advanced models generate significantly better triggering conditions than older ones. For example, GPT-5 scores 23 versus GPT-

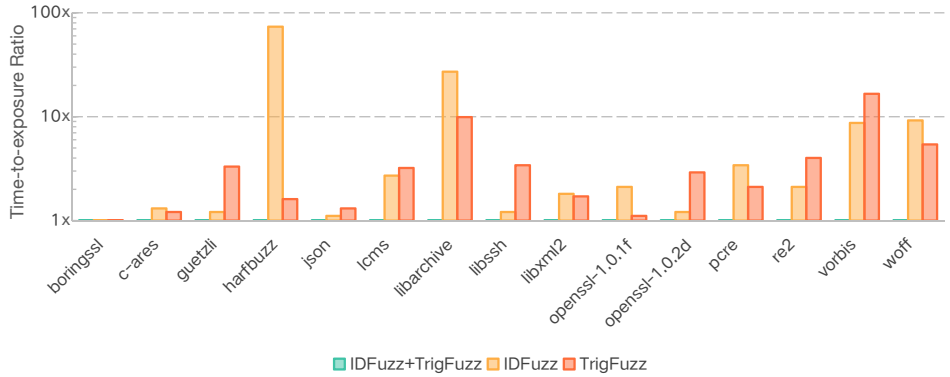

Figure 4: Comparison of the integration. The y-axis is the ratio of the time-to-exposure compared with IDFuzz+TRIGFUZZ (logarithmic scale).

4's 14. GPT-4-based TRIGFUZZ frequently misses critical conditions or generates inverted ones (*e.g.*, $bit \geq 2$ instead of $bit < 2$ for SND020). The three advanced models have distinct strengths: DeepSeek-R1 offers lowest cost but longest query time; Claude Sonnet 4.5 provides fastest analysis (˜1 min/target, shorter than the static analysis time required by most directed fuzzing approaches); GPT-5 achieves optimal balance of quality, cost, and speed.

Finding 3. The triggering condition generation task benefits substantially from the reasoning capabilities of advanced LLMs.

5.4. Complementarity with Existing Directed Fuzzers

TRIGFUZZ's strategy focuses on reducing the gap between reaching the target code and triggering the target vulnerability. In this section, we explore whether this strategy can be combined with existing state-of-the-art directed fuzzers that focus on reaching the target code to simultaneously achieve both fast reachability and fast triggering.

We integrate TRIGFUZZ with IDFuzz [18], one of the most advanced directed fuzzers that focus on reaching target code, which leverages machine learning methods to identify critical seed fields and implement target-oriented intelligent seed mutation. Since IDFuzz is currently unable to run on Magma, we conduct experiments on the Google Fuzzer Test Suite [19] benchmark used in IDFuzz's original paper. We compare the results of IDFuzz (AFLGo-based version), TRIGFUZZ, and IDFuzz+TRIGFUZZ. In IDFuzz+TRIGFUZZ, for seeds that have not reached the target code, we adopt IDFuzz's seed mutation strategy, while for reachable seeds, we employ TRIGFUZZ's power scheduling and mutation strategies. We follow the experimental setup in IDFuzz's original paper. The results are shown in Figure 4. TRIGFUZZ can be integrated with IDFuzz and improve its performance by 3.14x and 2.73x compared to using IDFuzz or TRIGFUZZ alone, respectively. This improvement indicates that TRIGFUZZ's ability to accelerate vulnerability triggering and state-of-the-art directed fuzzers' ability to accelerate code reachability are orthogonal and complementary.

5.5. New Vulnerabilities Discovered

We also evaluated TRIGFUZZ's capability in detecting new vulnerabilities. We selected the latest versions of popular projects previously tested in fuzzing research [19].

First, we perform code review on the target projects using the static analysis tool SVF [49] and an LLM (GPT-5), prompting the LLM to generate inferred vulnerability reports for identified suspicious locations. We then provide these potential vulnerability locations along with the generated reports as input to TRIGFUZZ to conduct directed fuzzing and produce PoC inputs. To date, we have discovered 7 new vulnerabilities in libarchive [50] and gpac [51]. A continuously updated list of discovered vulnerabilities is maintained here.

As a representative example, we discovered a heap-based out-of-bounds write in libarchive during the generation of Joliet identifiers for ISO 9660 images. The root cause lies in the identifier generation routine `isoent_gen_joliet_identifier()`, where libarchive computes an offset `noff` indicating the position at which a numeric suffix should be inserted to avoid duplicate Joliet filenames. Due to the absence of a bounds check, `noff` can take a negative value, causing the pointer `p` to reference a location before the beginning of the allocated identifier buffer and resulting in an out-of-bounds write. The triggering condition for this vulnerability is `noff < 0`, which TRIGFUZZ readily inferred and leveraged to guide fuzzing, successfully triggering the vulnerability within 24 hours, whereas AFLGo failed to trigger it within 48 hours.

6. Limitations

Incomplete Information Obtained Through Static Analysis. We extract code relevant to a vulnerability by slicing along the vulnerability call chain and supplying this code

to LLM for TC generation, as introduced in §3.1. When the call chain cannot be obtained from the bug report, we derive potential call paths by constructing a call graph through static analysis. However, indirect calls can cause these call chains to be incomplete, leading to: (1) we cannot provide the LLM with the caller function, preventing it from inferring the semantics of variables defined earlier in the execution context; and (2) we cannot precisely identify the callee functions corresponding to indirect call sites queried by the LLM, limiting the model's ability to reason about the effects of those calls. In fact, incompleteness in call graph inference is an open challenge faced by static analysis [9], [52]. To address this, we plan to explore dynamic tracing or AI-based techniques such as CALLEE [53] to complement call graphs in future work.

Instability of TC Validation. We dynamically validate the correctness of a triggering condition by examining the program state when reachable inputs execute the code location associated with that condition. If such inputs produce a triggering distance ≤ 0, or if forcibly satisfying the condition does not result in the expected manifestation of the vulnerability, we conclude that satisfying the condition cannot trigger the target bug. However, this validation either relies on reachable inputs generated through fuzzing, which may fail to cover all boundary cases, or modifies the original program logic, which may introduce unreliability. Consequently, a triggering condition that passes validation is not guaranteed to truly trigger the vulnerability. On the other hand, TCs that fail validation may still effectively guide fuzzing. For example, if the ground truth triggering condition is $a < b \wedge c < d$ but the LLM generates $a < b$, the TC may fail validation due to the missing condition $c < d$. However, guiding fuzzing toward satisfying $a < b$ alone can still facilitate vulnerability triggering. Therefore, improving TC validation reliability is crucial. Future research directions may include enabling LLM self-verification or employing cross-validation across multiple models.

Fuzzing-Uninformative Triggering Conditions. We perform seed scheduling based on the triggering distances of different seeds and identify critical seed fields by examining mutations that change the triggering distances. Therefore, a key prerequisite for the effectiveness of our approach is that the triggering distance derived from the triggering condition assumes multiple informative values. However, as discussed in §5.1, some triggering conditions yield only two distances: 1 (not triggered) and 0 (triggered), which is uninformative for guiding fuzzing. For example, the triggering condition for PNG007 is `png_ptr->palette == NULL`. We have attempted to guide the LLM, when encountering such cases, to infer their prerequisite conditions as auxiliary triggering conditions to better inform fuzzing. However, this limitation cannot be fully eliminated. In such cases, the performance of TrigFuzz degrades to that of its baseline, AFLGo.

Potential Data Contamination in Benchmark Evaluation. We evaluate TRIGFUZZ using the publicly available Magma benchmark, which provides manually annotated triggering conditions for each target vulnerability, authored by the benchmark maintainers [12]. While these annotations serve as the ground truth for assessing the correctness of LLM-generated triggering conditions, they also introduce a potential concern: the LLMs evaluated in this work may have been exposed to Magma and its associated ground-truth annotations during pre-training. This could introduce bias into our TC quality evaluation in §5.3, as a model may reproduce correct TCs through memorization rather than genuine code reasoning, a phenomenon commonly referred to as data contamination [54].

We clarify that this concern applies solely to our evaluation of LLM-generated triggering condition quality (§5.3), and does not affect our broader conclusion that accurate triggering conditions effectively guide vulnerability triggering. Furthermore, while direct querying revealed that all models evaluated in this work are aware of the Magma benchmark (which is expected given that Magma has been publicly available on GitHub since 2020), none of the models could recall the annotated triggering conditions for any specific target when prompted. Moreover, as reported in Finding 3, advanced models consistently produced higher-quality triggering conditions than weaker ones, suggesting that performance differences reflect genuine reasoning capabilities rather than memorization. We also applied TRIGFUZZ to the GFTS benchmark, which lacks ground-truth triggering conditions, in §5.4, and to real-world 0-day vulnerability discovery in §5.5, both of which demonstrate consistently strong performance free from data contamination concerns. Nonetheless, a rigorous investigation of data contamination remains an open question for future work.

7. Related Work

Large Language Model for Vulnerability Detection. Benefiting from the increasingly powerful code reasoning capabilities of LLMs, recent years have witnessed significant advancement in applying LLMs to vulnerability detection. Researchers have explored various adaptation techniques including fine-tuning [55], [56], [57], prompt engineering [15], [58], and retrieval augmentation [59], [60]. However, despite some promising results, directly leveraging LLMs for vulnerability detection still faces severe challenges, including the inability to effectively detect inter-procedural vulnerabilities [57] and high noise in training datasets with incorrect heuristic-based labels [61], [62]. Current methods achieve only ˜67.6% detection accuracy while suffering from high false positive rates [17].

In this paper, rather than directly employing LLMs to determine whether code contains vulnerabilities, we provide LLMs with precise target vulnerability information and leverage them to guide directed fuzzing, effectively mitigating the issues of insufficient accuracy and high false positive rates prevalent in LLM-based vulnerability detection.

Constraint Solving. Constraint solving is a fundamental technique in program analysis. Its representative approach, concolic execution, leverages constraint solvers to resolve path constraints and generate constraint-satisfying test cases. For instance, KLEE [63] systematically explores program

paths through constraint solving to achieve high code coverage, while QSYM [64] employs selective instruction-level symbolic execution to solve partial path constraints for a seed input, subsequently modifying specific bytes to satisfy these constraints and explore new execution paths. ConcoLLMic [65] further mitigates the scalability limitations of traditional concolic execution by employing LLM agents and operating at higher levels of semantic abstraction. However, it still reasons over entire execution paths and requires numerous LLM invocations.

Our work generates vulnerability-triggering constraints (*i.e.*, TCs). Rather than explicitly solving these constraints, we compute a fitness metric (*i.e.*, triggering distance) based on them to guide directed fuzzing, progressively minimizing the triggering distance through evolutionary search to generate a PoC input. Compared to concolic execution, our approach focuses exclusively on vulnerability-triggering constraints without needing to maintain constraints over entire execution paths, and incurs lower overhead owing to less frequent LLM invocations and the inherent efficiency of fuzzing.

Branch Condition Distance. Korel [32] introduced the concept of *branch functions*, which map branch conditions in a program to real-valued distances that quantify how far the current program state is from satisfying a given condition. The *constraint distance* employed by CAFL [13] and the triggering distance calculation rules for individual TCUs in TABLE 1 are informed by this approach. Nevertheless, we emphasize that our work differs fundamentally from that of Korel in three respects. First, we leverage LLMs to extract vulnerability-triggering conditions, which, unlike branch conditions, do not appear explicitly in the program source code and cannot be derived through static syntactic analysis alone. Second, we formalize a comprehensive TC framework that extends prior work by incorporating additional critical elements, such as condition location and dynamically collected normalized values (w), to more precisely characterize the requirements for vulnerability triggering. Third, we exploit the code comprehension capabilities of LLMs to handle special condition types that resist mechanical rule application.

8. Conclusion

In this work, we present TRIGFUZZ, a directed fuzzing solution leveraging LLMs to generate vulnerability triggering conditions for test guidance. Through a novel formal representation and validation technique, TRIGFUZZ produces high-quality triggering conditions that guide seed scheduling and mutation strategies. Our evaluation shows TRIGFUZZ generates effective conditions for 96.67% of targets, achieving over 1.72x speedup over state-of-the-art directed fuzzers on Magma while being complementary to existing fuzzing strategies that focus on reaching target code locations. So far, we discovered 7 new vulnerabilities using TRIGFUZZ.

9. Ethics Considerations

As mentioned in §5.5, TRIGFUZZ discovered 7 new vulnerabilities. We reported each vulnerability to the respective developers immediately upon discovery, following the security policies specified in their GitHub repositories. To date, 5 of the reported vulnerabilities have been confirmed by the developers, 2 of which have been patched. We will continue to follow up on the confirmation and patching status of all reported vulnerabilities. We did not disclose these vulnerabilities to any third parties other than the developers.

10. LLM Usage Considerations

- LLMs were used for editorial purposes in this manuscript, and all outputs were inspected by the authors to ensure accuracy and originality.
- In our work, LLMs are used to analyze vulnerable code and vulnerability reports to generate triggering conditions. All models are accessed via publicly available commercial APIs, making our experiments independently replicable. However, LLM outputs may exhibit non-determinism, as we adopt the default non-zero temperature settings of each model in our experiments.

Acknowledgments

We would like to thank the anonymous reviewers and our shepherd for their constructive comments and efforts to help improve this paper. This work is supported by the National Natural Science Foundation of China under grant 62132009 and U24A20337, and Joint Research Center for System Security, Tsinghua University (Institute for Network Sciences and Cyberspace) - Science City (Guangzhou) Digital Technology Group Co., Ltd.

References

[1] J. Li, B. Zhao, and C. Zhang, "Fuzzing: a survey," *Cybersecurity*, vol. 1, no. 1, pp. 1–13, 2018.

[2] P. Li, W. Meng, and C. Zhang, "{SDFuzz}: Target states driven directed fuzzing," in *33rd USENIX Security Symposium (USENIX Security 24)*, 2024, pp. 2441–2457.

[3] T. E. Kim, J. Choi, K. Heo, and S. K. Cha, "{DAFL}: Directed grey-box fuzzing guided by data dependency," in *32nd USENIX Security Symposium (USENIX Security 23)*, 2023, pp. 4931–4948.

[4] J. Xuan, X. Xie, and M. Monperrus, "Crash reproduction via test case mutation: Let existing test cases help," in *Proceedings of the 2015 10th joint meeting on foundations of software engineering*, 2015, pp. 910–913.

[5] M. Payer, "The fuzzing hype-train: How random testing triggers thousands of crashes," *IEEE Security & Privacy*, vol. 17, no. 1, pp. 78–82, 2019.

[6] M. Böhme, V.-T. Pham, M.-D. Nguyen, and A. Roychoudhury, "Directed greybox fuzzing," in *Proceedings of the 2017 ACM SIGSAC conference on computer and communications security*, 2017, pp. 2329–2344.

[7] H. Chen, Y. Xue, Y. Li, B. Chen, X. Xie, X. Wu, and Y. Liu, "Hawkeye: Towards a desired directed grey-box fuzzer," in *Proceedings of the 2018 ACM SIGSAC conference on computer and communications security*, 2018, pp. 2095–2108.

[8] Z. Du, Y. Li, Y. Liu, and B. Mao, "WindRanger: a directed greybox fuzzer driven by deviation basic blocks," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 2440–2451.

[9] C. Luo, W. Meng, and P. Li, "Selectfuzz: Efficient directed fuzzing with selective path exploration," in *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2023, pp. 2693–2707.

[10] H. Huang, Y. Guo, Q. Shi, P. Yao, R. Wu, and C. Zhang, "Beacon: Directed grey-box fuzzing with provable path pruning," in *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2022, pp. 36–50.

[11] H. Huang, A. Zhou, M. Payer, and C. Zhang, "Everything is good for something: Counterexample-guided directed fuzzing via likely invariant inference," in *2024 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, 2024, pp. 142–142.

[12] A. Hazimeh, A. Herrera, and M. Payer, "Magma: A ground-truth fuzzing benchmark," *Proc. ACM Meas. Anal. Comput. Syst.*, vol. 4, no. 3, Dec. 2020. [Online]. Available: https://doi.org/10.1145/3428334

[13] G. Lee, W. Shim, and B. Lee, "Constraint-guided directed greybox fuzzing," in *30th USENIX Security Symposium (USENIX Security 21)*, 2021, pp. 3559–3576.

[14] H. Hanif and S. Maffeis, "Vulberta: Simplified source code pre-training for vulnerability detection," in *2022 International joint conference on neural networks (IJCNN)*. IEEE, 2022, pp. 1–8.

[15] H. Li, Y. Hao, Y. Zhai, and Z. Qian, "Enhancing static analysis for practical bug detection: An llm-integrated approach," *Proceedings of the ACM on Programming Languages*, vol. 8, no. OOPSLA1, pp. 474–499, 2024.

[16] K. Wang, L. Huang, W. Fang, and W. Wang, "Clarc: C/c++ benchmark for robust code search," *arXiv preprint arXiv:2603.04484*, 2026.

[17] X. Zhou, S. Cao, X. Sun, and D. Lo, "Large language model for vulnerability detection and repair: Literature review and the road ahead," *ACM Transactions on Software Engineering and Methodology*, vol. 34, no. 5, pp. 1–31, 2025.

[18] Y. Chen, C. Zhang, L. Wang, W. Zhu, C. Luo, N. Gui, Z. Ma, X. Zhang, and B. Su, "{IDFuzz}: Intelligent directed grey-box fuzzing," in *34th USENIX Security Symposium (USENIX Security 25)*, 2025, pp. 6219–6238.

[19] "Google Fuzzer Test Suite," 2025, https://github.com/google/fuzzer-test-suite.

[20] M. Wang, J. Liang, C. Zhou, Z. Wu, J. Fu, Z. Su, Q. Liao, B. Gu, B. Wu, and Y. Jiang, "Data coverage for guided fuzzing," in *33rd USENIX Security Symposium (USENIX Security 24)*, 2024, pp. 2511–2526.

[21] P. Chen and H. Chen, "Angora: Efficient fuzzing by principled search," in *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2018, pp. 711–725.

[22] Y. Jiang, J. Liang, F. Ma, Y. Chen, C. Zhou, Y. Shen, Z. Wu, J. Fu, M. Wang, S. Li *et al.*, "When fuzzing meets llms: Challenges and opportunities," in *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*, 2024, pp. 492–496.

[23] Y. Deng, C. S. Xia, H. Peng, C. Yang, and L. Zhang, "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*, 2023, pp. 423–435.

[24] C. S. Xia, M. Paltenghi, J. L. Tian, M. Pradel, and L. Zhang, "Fuzz4all: Universal fuzzing with large language models," in *Proceedings of the 46th International Conference on Software Engineering*, ser. ICSE '24, 2024.

[25] R. Meng, M. Mirchev, M. Böhme, and A. Roychoudhury, "Large language model guided protocol fuzzing," in *Proceedings of the 31st Annual Network and Distributed System Security Symposium (NDSS)*, vol. 2024, 2024.

[26] Z. Zhou, Y. Yang, S. Wu, Y. Huang, B. Chen, and X. Peng, "Magneto: A step-wise approach to exploit vulnerabilities in dependent libraries via llm-empowered directed fuzzing," in *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, 2024, pp. 1633–1644.

[27] Y. Lyu, Y. Xie, P. Chen, and H. Chen, "Prompt fuzzing for fuzz driver generation," in *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, 2024, pp. 3793–3807.

[28] C. Zhang, Y. Zheng, M. Bai, Y. Li, W. Ma, X. Xie, Y. Li, L. Sun, and Y. Liu, "How effective are they? exploring large language model based fuzz driver generation," in *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2024, pp. 1223–1235.

[29] "CVE-2019-12730," 2025, https://nvd.nist.gov/vuln/detail/CVE-2019-12730.

[30] "CVE-2021-22925," 2025, https://nvd.nist.gov/vuln/detail/CVE-2021-22925.

[31] W. L. Fithen, S. V. Hernan, P. F. O'Rourke, and D. A. Shinberg, "Formal modeling of vulnerability," *Bell Labs technical journal*, vol. 8, no. 4, pp. 173–186, 2004.

[32] B. Korel, "Automated software test data generation," *IEEE Transactions on software engineering*, vol. 16, no. 8, pp. 870–879, 1990.

[33] F. M. Brown, *Boolean reasoning: the logic of Boolean equations*. Courier Corporation, 2012.

[34] OpenAI, "Introducing gpt-5," 2025, https://openai.com/index/introducing-gpt-5/.

[35] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell *et al.*, "Language models are few-shot learners," *Advances in neural information processing systems*, vol. 33, pp. 1877–1901, 2020.

[36] J. Wei, X. Wang, D. Schuurmans, M. Bosma, F. Xia, E. Chi, Q. V. Le, D. Zhou *et al.*, "Chain-of-thought prompting elicits reasoning in large language models," *Advances in neural information processing systems*, vol. 35, pp. 24 824–24 837, 2022.

[37] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "{AddressSanitizer}: A fast address sanity checker," in *2012 USENIX annual technical conference (USENIX ATC 12)*, 2012, pp. 309–318.

[38] N. F. Liu, K. Lin, J. Hewitt, A. Paranjape, M. Bevilacqua, F. Petroni, and P. Liang, "Lost in the middle: How language models use long contexts," *arXiv preprint arXiv:2307.03172*, 2023.

[39] A. Hurst, A. Lerer, A. P. Goucher, A. Perelman, A. Ramesh, A. Clark, A. Ostrow, A. Welihinda, A. Hayes, A. Radford *et al.*, "Gpt-4o system card," *arXiv preprint arXiv:2410.21276*, 2024.

[40] M. Zalewski, "American Fuzzy Lop," 2025, https://lcamtuf.coredump.cx/afl/.

[41] "aider, AI Pair Programming in Your Terminal," 2025, https://github.com/Aider-AI/aider.

[42] A. Shah, D. She, S. Sadhu, K. Singal, P. Coffman, and S. Jana, "MC2: Rigorous and Efficient Directed Greybox Fuzzing," in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, 2022, pp. 2595–2609.

[43] H. Huang, P. Yao, C. Hung-Chun, Y. Guo, and C. Zhang, "Titan: Efficient multi-target directed greybox fuzzing," in *2024 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, 2023, pp. 59–59.

[44] P. E. McKnight and J. Najab, "Mann-whitney u test," *The Corsini encyclopedia of psychology*, pp. 1–1, 2010.

[45] A. Vargha and H. D. Delaney, "A critique and improvement of the cl common language effect size statistics of mcgraw and wong," *Journal of Educational and Behavioral Statistics*, vol. 25, no. 2, pp. 101–132, 2000.

[46] Anthropic, "Introducing claude sonnet 4.5," 2025, https://www.anthropic.com/news/claude-sonnet-4-5.

[47] D. Guo, D. Yang, H. Zhang, J. Song, R. Zhang, R. Xu, Q. Zhu, S. Ma, P. Wang, X. Bi *et al.*, "Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning," *arXiv preprint arXiv:2501.12948*, 2025.

[48] J. Achiam, S. Adler, S. Agarwal, L. Ahmad, I. Akkaya, F. L. Aleman, D. Almeida, J. Altenschmidt, S. Altman, S. Anadkat *et al.*, "Gpt-4 technical report," *arXiv preprint arXiv:2303.08774*, 2023.

[49] Y. Sui and J. Xue, "Svf: interprocedural static value-flow analysis in llvm," in *Proceedings of the 25th international conference on compiler construction*, 2016, pp. 265–266.

[50] "libarchive," 2025, https://github.com/libarchive/libarchive.

[51] "gpac," 2025, https://github.com/gpac/gpac.

[52] K. Lu and H. Hu, "Where does it go? refining indirect-call targets with multi-layer type analysis," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019, pp. 1867–1881.

[53] W. Zhu, Z. Feng, Z. Zhang, J. Chen, Z. Ou, M. Yang, and C. Zhang, "Callee: Recovering call graphs for binaries with transfer and contrastive learning," in *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2023, pp. 2357–2374.

[54] O. Sainz, J. Campos, I. García-Ferrero, J. Etxaniz, O. L. de Lacalle, and E. Agirre, "Nlp evaluation in trouble: On the need to measure llm data contamination for each benchmark," in *Findings of the Association for Computational Linguistics: EMNLP 2023*, 2023, pp. 10 776–10 787.

[55] C. Thapa, S. I. Jang, M. E. Ahmed, S. Camtepe, J. Pieprzyk, and S. Nepal, "Transformer-based language models for software vulnerability detection," in *Proceedings of the 38th annual computer security applications conference*, 2022, pp. 481–496.

[56] C. Ni, X. Yin, K. Yang, D. Zhao, Z. Xing, and X. Xia, "Distinguishing look-alike innocent and vulnerable code by subtle semantic representation learning and explanation," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2023, pp. 1611–1622.

[57] A. Sejfia, S. Das, S. Shafiq, and N. Medvidović, "Toward improved deep learning-based vulnerability detection," in *Proceedings of the 46th IEEE/ACM international conference on software engineering*, 2024, pp. 1–12.

[58] M. D. Purba, A. Ghosh, B. J. Radford, and B. Chu, "Software vulnerability detection using large language models," in *2023 IEEE 34th International Symposium on Software Reliability Engineering Workshops (ISSREW)*. IEEE, 2023, pp. 112–119.

[59] X. Du, G. Zheng, K. Wang, Y. Zou, Y. Wang, W. Deng, J. Feng, M. Liu, B. Chen, X. Peng *et al.*, "Vul-rag: Enhancing llm-based vulnerability detection via knowledge-level rag," *arXiv preprint arXiv:2406.11147*, 2024.

[60] X. Zhou, T. Zhang, and D. Lo, "Large language model for vulnerability detection: Emerging results and future directions," in *Proceedings of the 2024 ACM/IEEE 44th International Conference on Software Engineering: New Ideas and Emerging Results*, 2024, pp. 47–51.

[61] R. Croft, M. A. Babar, and M. M. Kholoosi, "Data quality for software vulnerability datasets," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2023, pp. 121–133.

[62] Y. Xiong, A. Liu, X. Gao, and A. Yauhen, "Doa estimation using deep neural network with regression," in *2022 5th International Conference on Information Communication and Signal Processing (ICICSP)*. IEEE, 2022, pp. 1–5.

[63] C. Cadar, D. Dunbar, D. R. Engler *et al.*, "Klee: unassisted and automatic generation of high-coverage tests for complex systems programs." in *OSDI*, vol. 8, 2008, pp. 209–224.

[64] I. Yun, S. Lee, M. Xu, Y. Jang, and T. Kim, "QSYM: A practical concolic execution engine tailored for hybrid fuzzing," in *27th USENIX Security Symposium (USENIX Security 18)*, 2018, pp. 745–761.

[65] Z. Luo, H. Zhao, D. Wolff, C. Cadar, and A. Roychoudhury, "Agentic concolic execution," in *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2026, pp. 1–19.

Appendix A.
Detailed Few-Shot Examples Provided in LLM Prompts

A.1. Example 1

```
Information:
    Vulnerability Type: use-of-uninitialized-variable
    Crash/Trigger Point(s): line 6 in example.c

    Relevant Source Code:
    In example.c:
    1    #include <stdio.h>
    2
    3    void foo() {
    4        int a, b, c, d;
    5        int v = sscanf(input.field,"%d.%d.%d.%d",&a
         ,&b,&c,&d);
    6        use(b);
    7    }

Steps for triggering condition generation:
    1. Identify and output each conditional statement
       along with its code location.
        v < 2, line 6 in example.c (line 6 means the
            conditional statement should already be
            satisfied when executing line 6)
    2. If multiple conditional statements across
       different locations are required, assign an
       execution order to each group.
        v < 2, line 6 in example.c, 0 (0 means executed 1
            st)
    3. For any complex conditional statements, decompose
       them into atomic conditional statements, and
       assign a conjunct identifier to each one for
       later reconstruction.
        v < 2, line 6 in example.c, 0, 0 (we first
            convert the target conditional statement into
             its Disjunctive Normal Form. The resulting
            DNF consists of n conjuncts, each
            representing a set of AND operations between
            primitive sub-conditions. Within this
            framework, the element conj serves as an
            identifier specifying the originating
            conjunct i for each primitive condition, 0
            means the first conjunct)
    4. Finally, output all triggering conditions in the
       form of tuples: <cond, loc, seq, conj>
        <v < 2, line 6 in example.c, 0, 0>
```

Code After Instrumentation:

```
1 #include <stdio.h>
2
3 void foo() {
4     int a, b, c, d;
5     int v = sscanf(input.field,"%d.%d.%d.%d",&a,&b,&c,&d);
6     distance_instrument(0.0 + v - 2, 0, 0, 2.0);
7     use(b); // use-of-uninitialized-variable
8 }
```

A.2. Example 2

Information:

Vulnerability Type: Null Pointer Dereference
Crash/Trigger Point(s): line 62 in b.c

Relevant Source Code:

```
In a.c:
29  for (i = 0; i < ctx->num_components; i++) {
30      if ((ctx->comp_flags[i] & COMP_FLAG_SKIP) &&
31          (ctx->components[i]->id >= ctx->
     skip_threshold)) {
32          free(ctx->components[i]);
33          ctx->components[i] = NULL;
34      }
35  }

In b.c:
40  void context_render(DecodeContext *ctx, FILE *out
     ) {
42      uint32_t offset;
43      uint32_t error_code = 0;
44
45      fprintf(out, "=== render: num_components=%u
     ===\n", ctx->num_components);
46
47      for (offset = 0; offset < ctx->num_components
     ; offset++) {
48
49          /* skip components not participating in
     rendering */
50          if (!(ctx->comp_flags[offset] &
     COMP_FLAG_ACTIVE))
51              continue;
52
53          /* validate render state before accessing
      the component */
54          error_code = check_render_state(ctx,
     offset);
55
56          /* abort the loop if any invariant is
     violated */
57          if (error_code != 0) {
58              fprintf(out, "[render] abort at
     offset=%u\n", offset);
59              break;
60          }
61
62          print_component(out, offset, ctx->
     components[offset]);
63
64          /* propagate write errors to the next
     iteration's check */
65          int rc = fprintf(out, "");
66          if (rc < 0)
67              error_code = (uint32_t)(-rc);
68      }
69  }
```

Steps for triggering condition generation:

1. Identify and output each conditional statement along with its code location.

```
ctx->comp_flags[i] & COMP_FLAG_SKIP != 0 && ctx->
     components[i]->id >= ctx->skip_threshold,
     line 30 in a.c
offset line 62 in b.c == i line 33 in a.c
ctx->comp_flags[offset] & COMP_FLAG_ACTIVE != 0,
     line 62 in b.c
error_code == 0, line 62 in b.c
```

2. If multiple conditional statements across different locations are required, assign an execution order to each group.

```
ctx->comp_flags[i] & COMP_FLAG_SKIP != 0 && ctx->
     components[i]->id >= ctx->skip_threshold,
     line 30 in a.c, 0
offset line 62 in b.c == i line 33 in a.c, 1
ctx->comp_flags[offset] & COMP_FLAG_ACTIVE != 0,
     line 62 in b.c, 1
error_code == 0, line 62 in b.c, 1
```

3. For any complex conditional statements, decompose them into atomic conditional statements, and assign a conjunct identifier to each one for later reconstruction.

```
ctx->comp_flags[i] & COMP_FLAG_SKIP != 0, line 30
     in a.c, 0, 0
ctx->components[i]->id >= ctx->skip_threshold,
     line 30 in a.c, 0, 0
record(i), line 33 in a.c, 1, 0
offset == record(i), line 62 in b.c, 1, 0
ctx->comp_flags[offset] & COMP_FLAG_ACTIVE != 0,
     line 62 in b.c, 1, 0
error_code == 0, line 62 in b.c, 1, 0
```

4. Finally, output all triggering conditions in the form of tuples: <cond, loc, seq, conj>

```
<ctx->comp_flags[i] & COMP_FLAG_SKIP != 0, line
     30 in a.c, 0, 0>
<ctx->components[i]->id >= ctx->skip_threshold,
     line 30 in a.c, 0, 0>
<record(i), line 33 in a.c, 1, 0>
<offset == record(i), line 62 in b.c, 1, 0>
<ctx->comp_flags[offset] & COMP_FLAG_ACTIVE != 0,
     line 62 in b.c, 1, 0>
<error_code == 0, line 62 in b.c, 1, 0>
```

Code After Instrumentation:

In a.c:

```
 29  for (i = 0; i < ctx->num_components; i++) {
+30      distance_instrument(0.0 + ctx->
     skip_threshold - ctx->components[i]->id, 0, 0,
     w0);
 31      if ((ctx->comp_flags[i] & COMP_FLAG_SKIP)
      &&
 32          (ctx->components[i]->id >= ctx->
     skip_threshold)) {
+33          distance_instrument(0.0 + i, 1, 0, w1)
     ;
 34          free(ctx->components[i]);
 35          ctx->components[i] = NULL;
 36      }
 37  }
```

In b.c:

```
 59              break;
 60          }
 61
+62          distance_instrument(0.0 - offset, 1,
     0, w1);
 63          print_component(out, offset, ctx->
     components[offset]);
```

Since we have <ctx->components[i]->id >= ctx->skip_threshold, line 30 in a.c, 0, 0> and <offset == record(i), line 62 in b.c, 1, 0>, we exclude all remaining conditions mapped to binary distances. Note that error_code are not intrinsically binary-valued. However, the LLM infers from code semantics that the underlying predicate <error_code == 0, line 62 in b.c, 1, 0> reduces to a boolean judgment (*i.e.*, whether an error has occurred), and therefore assigns it a binary distance.

Appendix B.
Meta-Review

The following meta-review was prepared by the program committee for the 2026 IEEE Symposium on Security and Privacy (S&P) as part of the review process as detailed in the call for papers.

B.1. Summary

The paper presents TRIGFUZZ, which is a directed grey-box fuzzer. However, rather than focusing on how to reach a target quickly, TRIGFUZZ focuses on strategies to satisfy the constraints required to actually trigger the vulnerability at the target. TRIGFUZZ leverages Large Language Models to generate triggering conditions and uses a distance-based metric. This results in TRIGFUZZ being considerably faster than existing tools.

B.2. Scientific Contributions

- Creates a New Tool to Enable Future Science
- Provides a Valuable Step Forward in an Established Field

B.3. Reasons for Acceptance

1) This paper provides a valuable step forward in an established field. Existing directed fuzzers focus on reaching target code quickly but lack effective strategies for satisfying the conditions that actually trigger vulnerabilities. The key insight is that LLMs can extract these conditions from source code and bug reports.
2) The paper creates a new tool to enable future science. TRIGFUZZ achieves a considerable speed-up in triggering known vulnerabilities compared to existing tools.

B.4. Noteworthy Concerns

None.