LLMBisect: Breaking Barriers in Bug Bisection with A Comparative Analysis Pipeline

Zheng Zhang *, Haonan Li *, Xingyu Li*, Hang Zhang†, Zhiyun Qian*,
* University of California, Riverside † Indiana University Bloomington
* {zzhan173, hli333, xli399, zhiyun.qian,krish}@ucr.edu
† hz64@iu.edu

***Abstract*—Bug bisection has been an important security task that aims to understand the range of software versions impacted by a bug, i.e., identifying the commit that introduced the bug. However, traditional patch-based bisection methods are faced with several significant barriers: For example, they assume that the bug-inducing commit (BIC) and the patch commit modify the same functions, which is not always true. They often rely solely on code changes, while the commit message frequently contains a wealth of vulnerability-related information. They are also based on simple heuristics (e.g., assuming the BIC initializes lines deleted in the patch) and lack any logical analysis of the vulnerability.**

In this paper, we make the observation that Large Language Models (LLMs) are well-positioned to break the barriers of existing solutions, e.g., comprehend both textual data and code in patches and commits. Unlike previous BIC identification approaches, which yield poor results, we propose a comprehensive multi-stage pipeline that leverages LLMs to: (1) fully utilize patch information, (2) compare multiple candidate commits in context, and (3) progressively narrow down the candidates through a series of down-selection steps. In our evaluation, we demonstrate that our approach achieves significantly better accuracy than the state-of-the-art solution by more than 38%. Our results further confirm that the comprehensive multi-stage pipeline is essential, as it improves accuracy by 60% over a baseline LLM-based bisection method.

I. Introduction

N-day vulnerabilities are known security flaws that are often not fixed in a timely manner due to complex dependency chains and limited maintenance resources [18]. The widespread reuse of open-source projects exacerbates this problem, as there are usually multiple downstream distributions maintained by different parties in the ecosystem [53], making it difficult to apply upstream security patches on time across all distributions. Research has shown the popularity and severity of this problem in critical open-source projects such as Linux [32] and Android [53], potentially affecting billions of users.

The information of affected software versions of a specific vulnerability is crucial for N-day vulnerability mitigation. To obtain such information, it is necessary to locate the commit introducing the vulnerability (*i.e.,* bug-inducing commit, or *BIC*) — an essential task known as *bug bisection*. In this paper, we align with previous studies [52] and define a *BIC* as a commit that introduces a software bug into a program. It is possible for multiple commits to contribute to the bug, with the final commit making the bug triggerable. In such cases, we consider the final commit as the *BIC*, as it marks the point when the vulnerability is considered to exist.

Automated bug bisection can significantly speed up the bug-fixing process in downstreams (*e.g.,* 2.23x on average for Google's codebase, according to a previous study [7]), however, achieving a high accuracy remains challenging. Consequently, public information of vulnerable versions (*e.g.,* in NVD database [5]) is usually incomplete or inaccurate, as shown in previous studies [12], [49], [24].

Existing automatic bug bisection approaches can be classified into several categories, each with its own significant limitations:

(1) PoC-Based. Directly or symbolically execute the PoC (Proof-of-Concept) against each software version to test whether the vulnerability can be triggered. [21], [52] Though straightforward, this approach suffers from limited availability of vulnerability PoCs. Furthermore, direct PoC execution [21] often fails due to subtle variations across software versions, resulting in low accuracy [6], while symbolic analysis [52] is known to be expensive, and only supports limited bug types (*e.g.,* use-after-free, out-of-bounds memory access).

(2) Bug Report-Based. These approaches first collect available bug reports and then identify possible BICs by their "relevance to the bug" [11], [46], [14], with simplified assumptions such as "a BIC should touch the code where the failure happens". Similar to PoCs, detailed bug reports are often not available. Moreover, the simplified assumptions/heuristics may not hold in reality, reducing the accuracy, *e.g.,* Fonte's [11] accuracy drops to 36% when N=1 in its top-N ranking algorithm.

(3) Patch-Based. Being the most widely used, these approaches statically analyze the bug-fix commit (usually available for vulnerabilities in open-source projects) to "infer" the bug-inducing commit in the commit history. Existing techniques in this category [39], [13], [26], [48], [54], [47] generally rely on manually developed, hardcoded, and thus inherently imprecise heuristics. For example, one common one is to treat the commit that introduces one or more lines deleted by the bug-fix as the bug-inducing commit; however, there are many

Network and Distributed System Security (NDSS) Symposium 2026
23-27 February 2026, San Diego, CA, USA
ISBN 979-8-9919276-8-0
https://dx.doi.org/10.14722/ndss.2026.230990
www.ndss-symposium.org

situations where the bug-inducing commit and the fix commit do not intersect. For example, a patch can add an additional security check before the original vulnerable code (without removing any existing lines of code), potentially in a different function, or the deleted lines in the bug-fix are irrelevant. Another significant shortcoming is that existing approaches usually only analyze the structured code changes in the bug-fix patches, while unable to take any advantage of the commit messages in the unstructured natural language form. However, commit messages often contain rich and valuable information that can boost the bug bisection performance (*e.g.,* hints on the vulnerability root causes).

In this paper, we target patch-based bisection because it is the most widely applicable scenario — not all bugs come with PoCs or crash reports. We specifically have three goals: i) support all types of patches and vulnerabilities, ii) utilize full patch information including both code changes and commit messages, and iii) go beyond the simple hardcoded heuristics and make accurate decisions based on analysis of the vulnerability logic. To achieve these goals, we propose LLMBISECT, an LLM-powered highly accurate bug-bisection solution. Our core insight is that LLMs are capable of understanding both code and natural languages, extracting useful information for bug-bisection. Recent LLM models (*e.g.,* OpenAI o1) also show impressive abilities in code reasoning.

Though promising, we find several obstacles to the direct application of LLMs. First, LLMs tend to produce *excessive false positives*, aggressively and incorrectly labeling commits as bug-inducing. Second, LLMs suffer from *self-consistency* issues, yielding conflicting decisions across multiple runs. Finally, the *cost* of using LLMs on large-scale software is prohibitive: modern projects often contain tens of thousands of commits, and processing each one naively can lead to substantial token consumption and increased time and costs.

On the positive side, we observe that LLMs excel at comparative assessment, accurately selecting the true BIC from a small pool of candidates. To exploit this strength, we design a multi-step filtering pipeline that leverages LLMs' comparative reasoning ability. First, we perform coarse-grained filtering to extract candidate BICs at scale (§III-C), using lightweight heuristics such as code changes and commit-message keywords. This step is inexpensive and efficient, yet it substantially narrows the search space for subsequent stages. Next, we apply fine-grained filtering in a multi-round, comparative fashion, ensuring that the LLM evaluates all promising candidates side by side. This design markedly improves accuracy. Finally, we incorporate majority voting at selected key points to mitigate self-consistency issues while avoiding significant performance overhead.

We extensively evaluate LLMBISECT on the Linux kernel, one of the most complex and important open-source software. The results show that LLMBISECT achieves a remarkable accuracy of 91%, significantly outperforming state-of-the-art bug-bisection approaches.

We summarize our contributions as follows:

(1) We analyzed and articulated the limitations of existing bug-inducing commit (BIC) identification methods and proposed a novel and fundamentally different solution: an automated bisection tool based on LLMs, called LLMBISECT. Our approach incorporates previously overlooked information, i.e., commit messages, which often contain valuable cues about relationships to bug-inducing commits.

(2) We identified key challenges in directly applying LLMs to the bisection task and proposed a new multi-step filtering framework to address them. This design hinges on generating large numbers of candidate bug-inducing commits and leveraging the comparative power of LLMs to later prune them. We open-source our solution to facilitate further research [4].

(3) We evaluated LLMBISECT against state-of-the-art methods and demonstrated that it significantly outperforms both existing tools and a baseline LLM-based solution. We also analyzed the root causes that limited the accuracy of earlier approaches and explained how our design overcomes these challenges. Through a comprehensive ablation study, we validated the effectiveness of our design across all stages.

II. MOTIVATION

A. Motivating example

The Patch:

tty: n_gsm: fix race condition in status line change on dead connections

gsm_cleanup_mux() cleans up the gsm by closing all DLCIs, stopping all timers, removing the virtual tty devices and clearing the data queues. This procedure, however, may cause subsequent changes of the virtual modem status lines of a DLCI. More data is being added the outgoing data queue and the **deleted kick timer** is restarted to handle this. At this point many resources have already been removed by the cleanup procedure. Thus, a kernel panic occurs. Fix this by proving in gsm_modem_update() that the cleanup procedure has not been started and the mux is still alive.

```
static int gsm_modem_update(...)
+   if (dlci->gsm->dead)
        return -EL2HLT;
```

The Bug-inducing Commit(partly):

```
static void __gsm_data_queue(...)
+   mod_timer(&gsm->kick_timer,...);

+static void gsm_kick_timer(...)

static int gsm_cleanup_mux(...)
    /* Finish outstanding timers, making sure
they are done */
+   del_timer_sync(&gsm->kick_timer);
```

Figure 1: Motivating example

Figure 1 illustrates a race-condition-induced use-after-free vulnerability. However, in this example, the true BIC that

The incorrect BIC(partly):

```
+ static int gsm_modem_update(struct gsm_dlci
*dlci, u8 brk)
+{
+   if (dlci->adaption == 2) {
+       /* Send convergence layer type 2 empty
data frame. */
+       gsm_modem_upd_via_data(dlci, brk);
+       return 0;
+   } else if (dlci->gsm->encoding == 0) {
+       /* Send as MSC control message. */
+       return gsm_modem_upd_via_msc(dlci, brk);
+   }
+
+   /* Modem status lines are not supported. */
+   return -EPROTONOSUPPORT;
+}
```

Figure 2: Motivating example FP

introduced the vulnerability modified a completely different function from the one targeted by the patch. Specifically, the **bug-inducing commit** introduced a new thread that runs the newly introduced function `gsm_kick_timer()`, while the **patch commit** adds a check in `gsm_modem_update()` to ensure that the cleanup process has not been initiated before it it allowed to create the timer thread (it will eventually call `__gsm_data_queue()`), and hence eliminating the possibility of a race.

SymBisect, the state-of-the-art PoC-based Bisection method, cannot accurately extract the BIC in this case because: 1. There is no existing Proof of Concept (PoC). 2. SymBisect only supports two specific types of bugs: Out-of-Bounds (OOB) and Use-After-Free (UAF). Specifically, it does not support race condition cases.

VSZZ, the state-of-the-art method in the SZZ family [39], [27], [16], [34], [13], fails to handle such cases because its fundamental assumption is that the BIC modifies the same functions as the patch (specifically, that the BIC initializes the lines deleted in the patch). However, in this case, the BIC and the patch modify completely different functions.

V0Finder, an advanced vulnerable code clone detection method, identifies vulnerable versions by comparing the patch functions of the target version and the patch functions before the patch, after normalization and abstraction. If they are identical, the target version is considered vulnerable. However, this method fails in the illustrated case because the BIC does not modify the patch functions at all.

This case motivates us to think of a better approach for extracting candidate commits, one that goes beyond merely tracking patch functions. In fact, in this example, although the BIC modifies a different function than the patch, we note that `gsm_cleanup_mux()` is explicitly mentioned in the patch description. This provides an important hint that we can expand our focus to not only analyze code changes but also extract valuable information from commit messages.

The Patch

```
thermal_zone_device_register_with_trips(...)
release_device:
    put_device(&tz->device);
-   tz = NULL;
remove_id:
    ida_free(&thermal_tz_ida, id);
free_tzp:
    kfree(tz->tzp);
```

The correct BIC

```
thermal_zone_device_register_with_trips(...)
remove_id:
    ida_free(&thermal_tz_ida, id);
+free_tzp:
+   kfree(tz->tzp);
```

The incorrect BIC (VSZZ)

```
thermal_zone_device_register_with_trips(...)
+release_device:
+   put_device(&tz->device);
+   tz = NULL;
```

The incorrect BIC (V0Finder)

```
thermal_zone_device_register_with_trips(...)
-  if (!ops) {
+  if (!ops || !ops->get_temp) {
       pr_err("Thermal zone device . . .");
```

Figure 3: Motivating example #2

Figure 3 illustrates another motivating example, which represents a NULL pointer deference bug. In this example, although the BIC modifies the same function as the patch and is thus included among the candidates, the flawed heuristics of traditional methods prevent them from accurately identifying the correct BIC.

Specifically, the buggy code incorrectly sets `tz` to NULL under certain conditions, which causes the NULL pointer to be dereferenced subsequently in `kfree(tz->tzp)`. The patch fixes this vulnerability by removing the assignment that sets tz to NULL.

The commit introducing this vulnerability added a `kfree()` function call where the null dereference occurs. Before the BIC, the `kfree` function call did not exist, so naturally, the null dereference was not an issue.

VSZZ does not try to understand the logic of the vulnerability. Instead, it tracks the lines deleted in the patch, leading back to the commit that initialized the line (in this case, an earlier commit that first created the line of `tz = NULL`). This completely overlooks the actual BIC.

V0Finder's flawed heuristics, comparing hash values (essentially string matching after normalization and abstraction) of the whole patch function, take a different approach, focusing

on all commits that modified the patched function. Specifically, it identifies a commit that modified the patch function (the latest one before the patch) as the BIC, but this modification is unrelated to the vulnerability. V0Finder does not determine whether the modification is logically connected to the vulnerability; it simply assumes that, before the modification, the function was different, and therefore, the vulnerability did not exist.

B. Limitations of previous methods

Based on the motivating examples, we summarize the key weaknesses in patch-based methods, including *SZZ algorithm and its variants* [39], [27], [16], [34], [13] and most vulnerable code clone detection solutions [26], [48], [54], [47]. They suffer from the following limitations:

1) They often only consider code changes, ignoring commit messages, which frequently contain crucial information about vulnerabilities.

2) They fail to account for cases where a Bug-Introducing Commit (BIC) does not change the functions affected by the patch.

3) Many of them focus on deleted lines in the patch, making them ineffective when patches only include added lines or when the deleted lines are not critical to the vulnerability.

4) They tend to treat all code changes (such as deleted lines) equally. In reality, not all changes are of equal significance to the vulnerability.

5) Their judgments are often based on simple heuristics rather than logical reasoning. For example, VSZZ, the state-of-the-art SZZ method, traces back commit history to the earliest commit (instead of the most recent) that introduces the deleted lines of a patch. Such heuristics are often not accurate.

C. Insights

Revisiting the motivating example, we propose three design goals for an improved solution:

1) Leverage Full Patch Context: The solution should utilize the complete patch context, including both the patch code diff and commit messages, as these provide critical clues about the bug-inducing commit.

2) Minimize Assumptions and Requirements: Unlike approaches such as VSZZ, the solution should support patches that only add lines. It should also handle all types of bugs rather than being restricted to specific categories (e.g., SymBisect). Additionally, it must accommodate patches that do not modify functions, a limitation seen in V0Finder.

3) Incorporate Logical Reasoning: The approach should analyze the logic of the vulnerability to make a decision on the bug-inducing commit, rather than depend on simplistic and hardcoded heuristics like those used in VSZZ and V0Finder.

To achieve these goals, we propose leveraging large language models (LLMs) for the task of bug bisection. LLMs are well-suited for this purpose due to their ability to comprehend both code and patch descriptions. Moreover, they are trained on all types of bugs and patches and thus not limited to reasoning about specific types of bugs/patches. LLMs have demonstrated effectiveness in various bug analysis tasks [51], [44], [29], [43] and have been improving one generation after another.

III. Design

A. Design Motivation

Though LLMs show great potential in enhancing existing bug bisection techniques, it remains unclear how to best leverage their capabilities for optimal performance. To explore this, we began by reproducing the typical workflow of previous bug-inducing commit (BIC) identification work with LLM's drop-in help, which serves as the baseline for our design.

A baseline LLM-based bisection method. The method is inspired by heuristics employed in classic bug bisection methods (*e.g.,* SZZ). It operates as follows: 1) lists the commits that modified the patch function – this is an extended set of candidate BIC commits compared to prior methods like SZZ, and 2) queries the LLM for each commit in the list in reverse chronological order, stopping at the first one identified as the BIC. Through this baseline and its subsequent variants, we encountered several challenges that led to a notably low accuracy — the baselin achieves only 30%, as will be shown in §V. These challenges revealed key limitations of applying LLMs without structural guidance in the bug bisection task. They also led to new opportunities for improvement, enabled by a better understanding of BIC characteristics and the strengths and weaknesses of LLMs in the bug bisection task. These insights directly informed our design choices and gradually shaped the final version of our design through multiple iterations.

Intuitively, bug bisection is the process of identifying the BIC from a list of candidate commits related to a given patch and its associated vulnerability. Thus, we can divide the process into two steps: 1) extracting candidate commits from the commit history, and 2) selecting the exact bug-inducing commit from the candidate commits.

1) Collection of BIC candidates: **Baseline.** Our aforementioned baseline method generalizes the state-of-the-art method, i.e., VSZZ and V0Finder, which considered only the commits that changed patch function(s). Specifically, the baseline method collects *all historical commits that modify the patched function(s)*. The intuition is that this represents a superset of commits encompassing the BICs identifiable by previous methods. It can also overcome their limitation of not supporting patches with added lines only (no deleted lines).

Challenge #1. The total number of commits that modified the patch function is often quite large in the commit history. Too many candidates can reduce the accuracy of the LLM (as observed in our preliminary experiments). Moreover, some real-world BICs do not modify the patch functions at all, which will be missed by the above solution.

Observation #1. Not all functions or lines modified in the bug-fix commit are equally important or relevant to the vulnerability. For example, some code changes are merely for refactoring purposes without changing the semantics of the code. Previous methods also attempt to identify irrelevant code

changes. However, their methods are limited to only simple patterns such as adding or removing comments [48], [26].

Solution #1. We change the simple, non-distinguishing function-based candidate selection to a fine-grained, critical-line-based selection. Specifically, we first identify the most relevant changed lines to the vulnerability from the bug-fix commit, with LLM's help, and then include only historical commits that touch these lines in the candidate list. Note that this method is no longer limited to changes within the patched functions. Instead, it also considers changes to global variable definitions and struct definitions as potentially critical. As a result, this approach not only significantly reduces the number of candidates to inspect (by 81% on average in our evaluation) but also enables the identification of new, previously overlooked candidates.

It is worth noting that we also considered further expanding the scope for critical line selection (e.g., callers of the patched functions). However, this will substantially bloat the number of BIC candidates, with little benefit. As will be shown in §V-A, there are 8 cases where the BIC modified files are completely different from those in the patch, causing the BIC to be excluded from our candidate set. None of these cases could be resolved by including the callers of the patched functions in the analysis.

Challenge #2. While this improves the accuracy if the BIC indeed modified the critical lines, it still does not solve an aforementioned problem — the code change made in the bug-inducing and bug-fix commits can be disjoint (*e.g.,* in different functions or files).

Observation #2. The patch commit messages often contain useful clues hinting at the vulnerability's root cause and connecting it to the bug-fix (*e.g.,* the commit message of a bug-fix in `foo()` may mention that the vulnerability originates from `bar()`). The motivating example illustrated this point.

Solution #2. Going beyond the function- and critical-line-based candidate selection, we can leverage LLMs to select additional BIC candidates using hints extracted from the commit messages (*e.g.,* commits that modified a function mentioned in the commit message). Because we look for functions or variables outside of the patched function, it is complementary to the previous two methods by design.

2) Selection of BIC from candidates: **Baseline.** As mentioned in the aforementioned solution, we follow SZZ-style bisection, which simply inspects the BIC candidates in reverse chronological order; the first one recognized as BIC will be the selected one. This is a reasonable choice because we define a BIC as the last commit contributing to the vulnerability. While plausible, we identified multiple challenges during our preliminary experiments.

Challenge #3. High false positive rate of LLMs. During reverse chronological traversal, the LLM tends to identify BICs too eagerly, causing it to stop prematurely and miss the actual bug-inducing commits, which leads to low accuracy.

Observation #3. Despite having FPs, LLMs perform well in discerning the real BIC when it is presented with multiple candidates. The LLM's strength in comparative reasoning eliminates the need to have them make definitive decisions about individual BIC candidates in isolation.

Solution #3. We adopt a two-round BIC selection: 1) let the LLM inspect *all* candidates and identify *all* potential BICs, without early termination, and 2) let the LLM *compare* all the identified BICs and select a final one.

Challenge #4. False negatives are incurred in using any single method to collect BICs. Vulnerability-relevant lines can still sometimes be missed by LLMs in some cases, resulting in false negatives in BIC recognition.

Observation #4. The three methods of generating BIC candidates can complement each other in terms of the covered BIC candidates.

Solution #4. To avoid missing the correct BICs, we feed all three sets of BIC candidates (generated by three different methods) to the LLM, using the three methods described earlier. Although this approach increases token consumption compared to using a single set, it helps improve coverage. To further improve accuracy, we make a final selection from the results generated by different methods (*e.g.,* function-based and critical-line-based), rather than merging the candidate commits at the beginning. This is because the accuracy of the LLM tends to drop when we feed a large set of BIC candidates. In other words, we will feed only three candidates to the LLM, when it is making the final verdict. In most cases, even if a method produces an incorrect candidate, it is unlikely to be selected among the final three. However, when it does generate the correct candidate, that candidate is very likely to be selected in the final stage. This is again taking advantage of the comparative reasoning power of the LLM.

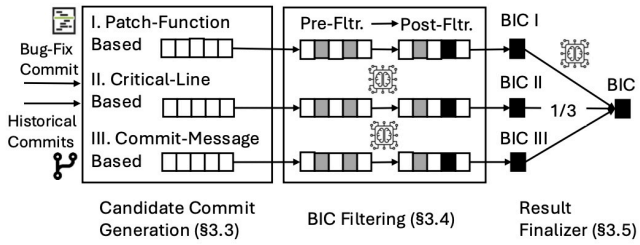

Figure 4: Workflow of LLMBISECT

B. Workflow

Motivated by the above design explorations, we present the workflow of our final design of LLMBISECT in Fig. 4. As we can see, there are three overall stages: 1) Candidate Commit Generation. 2) BIC Filtering. 3) Result Finalizer.

Candidate Commit Generation. Given a bug-fix commit, this stage's goal is to list all historical commits that could potentially be the BIC for future investigation (*i.e.,* candidate generation). As described previously, we have three candidate commit generation methods, based on patch functions, critical lines, and commit messages, respectively. These methods can complement each other (Solution #1 and Solution #2 in Section III-A).

BIC Filtering. At this stage, we aim to select the most likely Bug-Inducing Commit (BIC) from each list generated in the first stage, resulting in up to three final BIC candidates. This process is divided into two phases: the pre-filtering phase, which identifies possible BICs, and the post-filtering phase, which selects the most likely BIC (Solution #3 in Section III-A).

Result Finalization. At this stage, we finalize our decision by selecting one final Bug-Inducing Commit (BIC) from the potential BICs (up to three) identified during the BIC filtering stage. (Solution #4 in Section III-A)

Majority voting. LLMs sometimes make different decisions regarding BIC selection in multiple runs (*i.e.,* self-consistency). We observed that there usually exists a "dominating" decision occurring in most runs. Thus we adopt a "majority voting" mechanism in our design, where we run LLMs multiple times (defaulted at 7) for BIC identification during the Result Finalization phase.

C. Candidate Commit Generation

The quality of the candidate lists can significantly impact the accuracy of the final BIC identification. On the one hand, too many commits in the list will simultaneously increase the likelihood of errors and the cost. On the other hand, missing relevant commits leads to false negatives. As a result, we would ideally like the list to (1) contain the true BIC, and (2) be small enough. In practice, these two goals are hard to achieve simultaneously. We will present our key design below to strike a good balance.

Function-based Generator. As used in the baseline method (§III-A), the most commonly used generator in existing work is based on patched function(s) in the bug-fix commit, where all historical commits modifying the same function(s) are selected as candidates. This strategy is effective as bug-inducing and -fix commits frequently modify the same function(s).

Critical-line-based Generator. First, it recognizes lines that are truly relevant to the vulnerability logic (*i.e.,* critical lines). To achieve this, we utilize LLM's ability to comprehend both code and natural language to recognize critical lines, which are far more accurate than heuristic-based approaches used in existing work. We also provide the LLM with the full definitions of the patched functions, as part of prompt engineering, to better facilitate its understanding of vulnerability logic. Second, we will only treat historical commits that modify critical lines as BIC candidates.

Conceptually, we would like an LLM to focus on particular parts of the code that pertain to the vulnerability, whether they are part of the patched functions or changes to a global variable definition (if it is included in the code diff). It turns out that it is a non-trivial task. As mentioned, prior work often relies on overly simplistic heuristics to define critical lines. For example, all deleted lines within a function are considered critical [13], or every line in the patched functions is defined as critical [48]. We would like to generalize it and improve it, with the help of LLMs.

In particular, we divide patches into three types and apply tailored strategies using LLMs to identify critical lines:

(1) Patches with deleted lines. Deleted lines in a bug-fix commit are often related to the vulnerability, so in this case, we narrow our scope of critical line identification to the deleted lines (excluding trivial ones like comments) to improve efficiency. However, if LLMs recognize no critical lines among those deleted, we expand our scope to the whole patched function.

(2) Patches with only added lines. If a patch has only added lines, previous solutions, such as VSZZ simply give up. However, we would extract critical lines from the entire modified function/struct. Specifically, we would feed the whole patch, including the code diff and commit message, as well as the complete definitions of the affected functions. For example, if the patch merely adds a range check for a variable (such as an array index), the LLM can analyze the commit message and the function's surrounding code to identify critical statements related to this variable (e.g., an array access with the index, where the out-of-bounds (OOB) error occurs). These critical statements are often modified in the BIC.

(3) Patches with only reordered lines. These patches merely change the line positions (*e.g.,* adjust the critical section length by moving the lock/unlock statements). Here vulnerabilities are usually caused by improper relative positioning of two lines, one being the line modified by the patch and the other whose relative order to the modified line has changed. Therefore, merely focusing on the presence of modified lines is insufficient to determine whether a vulnerability exists. The introduction of a vulnerability is often closely related to the other line. Therefore, for such patches, we extract critical lines from the modified lines and the affected context statements (the statements whose relative position to the modified line has been altered after applying the patch). For example, if a patch moves a `lock()` call to an earlier position in the function, thereby extending the scope of the lock to include more statements, the statements newly encompassed by the lock after the patch are considered affected context statements. These often include critical statement related to the vulnerability.

Commit-message-based Generator. As discussed in §III-A, neither of the above generators can correctly include the BIC candidate if it has no overlap (regarding the modified code) with the bug-fix commit. To address this issue, we design the third generator based on commit messages, from which we extract valuable information regarding the vulnerability. More specifically, we try to extract the following information from the commit message with regular expression matching:

(1) Function/struct/variable names. They could indicate the actual location of the vulnerable code or global variables.

(2) Commit hashes. Some commit messages directly reference earlier (BIC) commits by their hashes.

We also include names of modified functions by the bug-fix as keywords, though technically they are not extracted from the bug-fix commit messages. They are useful because even the BIC may not modify the same functions, it might still

modify their callers which contain their names in the code (e.g., adding a call to the patched function).

To avoid redundant execution, we disregard all functions or structs modified by the Bug-fix commit (which have already been tracked by the first two generators).

After running the three above candidate generators independently, we obtain *three* candidate lists at the end of this stage, which will be fed as input to the next stage (§III-D).

D. BIC Filtering

In §III-C, we generate three lists of candidates, at this stage, we try to pick *one* most likely BIC from *each* list, resulting in *up to three* final BIC candidates (it is posible that no BIC is selected from a certain list) for the next stage (§III-E). One straightforward way to pick the BIC from a candidate list, as mentioned in §III-A, is to inspect each commit in reverse chronological order and stop when one is recognized as the BIC. However, this leads to a high positive rate because the "most likely" BIC infers that it can only be reliably identified from a comparison of multiple potential ones. Therefore, we design our BIC filtering process to be composed of two sub-phases: the pre- and post-filtering.

(1) Pre-Filtering. For *every* commit in a candidate list, we prompt it with the original bug-fix commit to LLM for a decision regarding whether it *could* be the BIC. This will result in multiple potential BICs selected by the LLM.

(2) Post-Filtering. The LLM is then instructed to perform a *comparative assessment* of all selected BICs in the pre-filtering phase, to finally pick *one* most likely BIC per candidate list.

This design gives LLMs sufficient opportunities to review all candidate commits and carefully compare them for better-informed decisions, significantly boosting the BIC identification accuracy, compared to baseline early stop solution.

E. Result Finalization

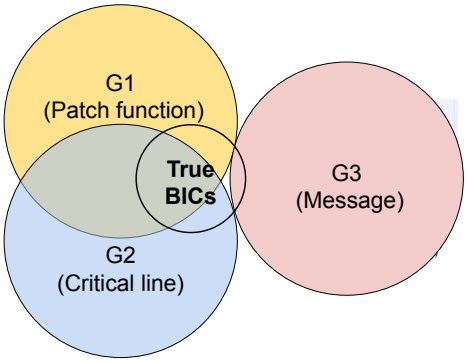

Figure 5: Candidate Generators

The last BIC filtering stage (§III-D) outputs up to three potential BICs selected from multiple candidate lists, while we still need to finalize our decision by picking one final BIC. To achieve this, our procedure is similar to the last stage (§III-D). Specifically, we present all BIC candidates (up to three) to the LLM for a comparative evaluation, in order to reach a final BIC decision as LLMBISECT's output. Note that though rarely the case, it is possible that LLMBISECT eventually fails to output any BIC (*e.g.,* zero candidates were selected in the previous BIC filtering or this result finalization process).

As mentioned in §III-A, we have three methods of generating BIC candidates. They can complement each other. They each have their trade-offs regarding the two goals listed above. Conceptually, we can see Fig. 5 which illustrates the different candidate sets produced by different generators. As a result, LLMBISECT adopts all three of the aforementioned BIC candidate generators. Our design is to have them work independently initially. Later on, we will attempt to pick the final result with Result Finalization. Note that we do not want to merge all the candidates into the same set initially and then have LLM pick one. This is because such a set will be too large which will hurt the accuracy. As mentioned in §III-A, the number of candidates produced by the function-based generator is already large, limiting the LLM's accuracy in picking the right BIC. It is therefore beneficial to keep the set of candidates produced by critical-line-based and commit-message-based methods separate. This way, if the correct BIC is located in either of the two sets, it will likely be correctly identified by the LLM. Again, the function-base generator can be viewed as a backup option. In case the correct BIC is present in only its result, then at least we would still have a chance to identify it.

IV. IMPLEMENTATION

We implement a prototype of LLMBISECT with 5,331 LoC in Python. In this section, we discuss some noteworthy implementation details.

Function-Based Candidate Generation. One can use a git command to track all commits that modify a specific function: `git log <commit_hash> -L:<funcname>:<filename>`. However, it can miss some commits when the function has been renamed or the file that contains the function has been renamed. To address this limitation, we developed a script to track all commits that modify the given file/function more comprehensively, correctly handling the renaming issues. For each commit, we then extract the functions modified by it, enabling us to obtain the complete list of commits modifying the specific function.

Patch Type Classification. We implement a Python script to first determine the type of patch (e.g., those with only added lines). This is relatively straightforward. We first extract and ignore all changes relating to comments, and then can easily classify patches into those with only added lines and those with deleted lines (we do not differentiate the patches with only deleted lines). Among the patches with deleted lines, if there are also added lines, we then use a simple string-match-based heuristic to identify reordered statements. Specifically, we consider a patch as reordering changes if and only if all the changes are related to reordering. In other words, all the removed lines must show up as added lines in another location verbatim.

After collecting this information, we first obtain historical commits that modify the same files as the bug-fix commit, then for each commit, we check whether it matches any of the extracted information with the commit message (*e.g.,* has the same hash, change/call the mentioned function, *etc.*). If so, we also consider it as a BIC candidate, which can be missed by function-based or critical-line-based generators.

LLM Models. In our implementation, we primarily use OpenAI o1 (`o1-preview-2024-09-12`) as the main LLM. We also evaluate other models, including GPT-4o (`gpt-4o-2024-08-06`) and the open-source, Llama 3 (`nvidia/llama-3.1-nemotron-70b-instruct`). The results of these evaluations are presented in Section V-D. The specific prompts used in each step are included in the appendix.

V. Evaluation

In this section, we evaluate LLMBisect to answer the following research questions:

- RQ1: How accurately does LLMBisect identify BICs?
- RQ2: How does LLMBisect compare against other state-of-the-art BIC identification methods?
- RQ3: How does each component and phase of the pipeline of LLMBisect contribute to its final performance?
- RQ4: How costly is the solution?

Dataset. We evaluate LLMBisect against Linux kernel CVEs. Several key considerations inform this choice: (1) Linux kernel is one of the most important and widely used software, its ecosystem contains numerous downstream distributions potentially impacted by N-day vulnerabilities, highlighting the importance of an accurate bug bisection, (2) The kernel also has one of the most complex codebases, containing a wide range of vulnerabilities reported daily by security practitioners. We believe the diversity and complexity of Linux kernel CVEs can rigorously test LLMBisect's accuracy and reliability. Note that despite our choice, LLMBisect by design is agnostic to the target software or vulnerability types.

Given the sheer number of Linux kernel CVEs and the high cost of advanced LLM tokens (e.g., o-1), we randomly sampled 100 CVEs in each of 2023 and 2024 (200 in total). We specifically include CVEs in 2024 as they are published after the LLM knowledge cut-off date, validating whether LLMBisect's result is influenced by the LLM's pre-existing knowledge about the CVEs (As shows later, there is nearly no difference. LLMBisect demonstrated similar accuracy on CVEs from 2023 and 2024). We also included CVEs in 2023 because the CVE assignment criteria became more relaxed starting from 2024 (*e.g.,* many non-security issues also had CVEs assigned) [3], [7], [1]. Testing these CVEs demonstrates LLMBisect's accuracy on security vulnerabilities more reliably.

Ground Truth. To get the ground truth (*i.e.,* the correct BIC for a specific vulnerability), we intentionally include in our dataset only those CVEs whose fix commit has a *fixes tag* [2], which points to the BIC(s) given by kernel developers. We then manually verify them according to our BIC definition and assemble the ground truth. After manual verification, we identified and corrected 11 cases with inaccurate fix tags. It is important to note that fix tags are merely used to provide the ground truth that is otherwise difficult to obtain – *we remove the fixes tags from bug-fix commits during the experiments*.

Dataset	Tools	Correct	Incorrect	Accuracy
LLMBisect (200 CVEs)	LLMBisect	182	18	91%
	V0Finder	66	134	33%
	VSZZ	102	98	51%
SymBisect (32 syzbot bugs)	LLMBisect	29	3	90.6%
	SymBisect	24	8	75%

Table I: **The results of BIC identification**

Threats to Validity. In our sampling process, we followed the same approach used in prior studies (e.g., SymBisect), selecting commits that include "Fixes" tags and manually verifying them. One potential bias in this approach is that commits with "Fixes" tags may contain more detailed and descriptive commit messages, which can make them easier for LLMs to process and understand. Unfortunately, there is a lack of dataset without "Fixes" tags to validate or invalidate the hypothesis. As a result, we choose to follow the best available approach used by prior methods. Nonetheless, we acknowledge the potential biases introduced by this approach and leave the task of building a reliable benchmark of bugs without "Fixes" tags as future work.

Comparison Targets. We extensively compare LLMBisect with three state-of-the-art tools covering different bug bisection methodologies:

(1) PoC-based bisection. SymBisect[52] is a state-of-the-art PoC-based bisection tool. It generates various guidance from PoC execution traces and uses principally guided under-constrained symbolic execution to confirm the bug's existence. However, it only supports limited vulnerability types and relies on PoCs — unavailable for most Linux kernel CVEs. Even then, we would like to see how LLMBisect compares to SymBisect using its evaluation dataset, which SymBisect supports very well. This is an interesting experiment that can showcase the performance differences between the symbolic reasoning (in SymBisect) and LLM's reasoning (in LLMBisect).

(2) Patch-based bisection with SZZ-style algorithms. As mentioned in §II, SZZ-style algorithms generally rely on the assumption that BIC will initialize lines deleted in the bug-fix commits. We select VSZZ [13] — the state-of-the-art open-source tool in this domain — as a comparison target, we configure it with default options specified in its tutorial [9].

(3) Patch-based bisection with vulnerable code clone detection. These methods are based on code similarity comparison between the vulnerable pre-fix version and a specific target version to probe the first vulnerable version (§II). V0Finder [48] is a latest tool in this direction, we configure it with its default options [8] in our comparison.

Tools	TP	FP	FN	Precision	Recall	F-1 Score
LLMBISECT	4121	151	146	96.5%	96.6%	96.5%
V0Finder	1594	56	2748	96.6%	36.7%	53.2%
VSZZ	4140	1660	85	71.4%	98.0%	82.6%

Table II: **The results of vulnerable versions detection**

Year	Inaccurate Cases	Accuracy
2023	8	92%
2024	10	90%

Table III: **The accuracy with cases in different years**

A. Accuracy of LLMBISECT (RQ1)

Accuracy of BIC Identification. Table I shows the results of BIC identification with different tools, LLMBISECT consistently achieves the highest accuracy of more than 90%, outperforming other state-of-the-art tools by significant margins (*i.e.,* 25.6% - 58%). Specifically, LLMBISECT accurately identified the correct BICs for the two motivating examples mentioned in Section II. Note that the comparison with SymBisect is based on SymBisect's own dataset due to its reliance on PoCs and specific vulnerability types, as mentioned previously in §V. These results show LLMBISECT's superior accuracy, even on dataset originally designed for other tools. We will describe the comparison results in detail in §V-C. Besides, Table III shows that there is nearly no difference in accuracy between cases from 2023 and those from 2024. This rules out the potential influence of the LLM's pre-existing knowledge about the CVEs on the results, demonstrating the general applicability of our method.

Inaccuracy Analysis. As shown in Table I, LLMBISECT has 18 inaccurate bisection cases out of the 200 CVEs, after inspecting each, we summarize 4 underlying reasons arising in 3 different phases of LLMBISECT (Fig. 4), as listed in Table IV. We now detail these reasons by phase.

Phase I: Candidate Generation. LLMBISECT will miss the correct BIC (*i.e.,* false negative) if it is not included in the initial candidate list, 10 failure cases belong to this category. Specifically, for 8 of them, the BIC and bug-fix commits modify completely different files, making it difficult to recognize the correct BIC candidates without incurring a high cost (*e.g.,* we need to enumerate virtually *all* commits for all files in the codebase.). In the remaining 2 cases, the BIC and bug-fix commit modify different functions, structs, or variables within the same file, however, our candidate generator fails to correlate them based on the bug-fix commit message, which does not contain enough hints (*e.g.,* the vulnerable function name) to locate the remotely related BIC.

Phase II: BIC Filtering. In this phase, LLMs first try to identify (multiple) potential BICs from a specific generator's candidate list, then select *one* BIC from multiple by comparing them. We have 4 failure cases where the true BIC does not survive this filtering process. Upon further investigation, we found that the failure is mainly because of the excessive number of potential BICs to filter (*e.g.,* 84.25 on average for these 4 cases vs. 36.5 for all). This confirms our design consideration (§III-A) that more candidate commits can decrease the accuracy, besides increasing the costs. We also observed that LLM's self-consistency issue contributes to 3 of these failure cases, where the correct BIC can be selected in some LLM runs but not in others.

Phase	Reason	Num
Candidate commit Generation	BIC changed different files	8
	Insufficient info in commit messages	2
BIC Filtering	Not Pick groundtruth as final BIC	4
Result Finalization	Not Pick groundtruth as final BIC	4

Table IV: **The reasons of LLMBISECT's inaccuracy**

It is worth noting that we do not have any inaccurate cases in the pre-filtering phase (*e.g.,* LLMs fail to pick the correct BIC from the generator's candidate list at beginning), this confirms our observation (§III-A) that LLM is less likely to make FNs when deciding whether an individual commit is a potential BIC.

Phase III: Result Finalization. Phase II selects one BIC from each of three generators' candidate lists, resulting in three final BIC candidates. Then, the result finalizer further selects one BIC from these three. 5 failure cases are due to that the correct BIC does not survive this final "1/3" selection process. We observed that the failure here is again related to LLMs' self-consistency (*e.g.,* correct BICs can survive in *some* runs).

B. Accuracy of Vulnerable Version Detection

One common application of bug bisection is to determine the software versions affected by a vulnerability, informing downstream developers for timely patch porting [53]. From this perspective, solely evaluating the accuracy of BIC identification has its limitations. For example, if a vulnerability is fixed in version 6.0 but introduced in version 5.0, a tool that identifies the introduction of the bug in version 5.1 or 5.19 would both be considered inaccurate from the perspective of BIC identification accuracy. However, the impact of such inaccuracies on downstream users can vary significantly.

Therefore, in addition to verifying whether our tool accurately identifies bug-inducing commits, we also evaluate the accuracy of identifying vulnerable versions. Specifically, once the BIC is determined, we can identify all vulnerable versions on the Linux mainline branch, i.e., versions between the BIC and the patch, considering only major releases such as v5.0, v5.1. By comparing the vulnerable versions derived from the true BIC with those derived from the BIC identified by our tool, we calculate the tool's false positives (FP), false negatives (FN), and true positives (TP) for this task. As Fig. 7 shows, once we identify the BIC, we can determine the numbers of TP, FN, and FP based on its relative position to the true BIC and the patch commit. However, the number of TNs depends on the manually selected starting point (e.g., whether we start counting from v2.6 or v4.0) and is not a fixed value. Therefore, TNs are not included in our statistics.

As shown in Table II, LLMBISECT achieves an overall F-1 score of 96.5%, much higher than all existing tools. Note that this evaluation is performed on a per-bug-version-pair basis.

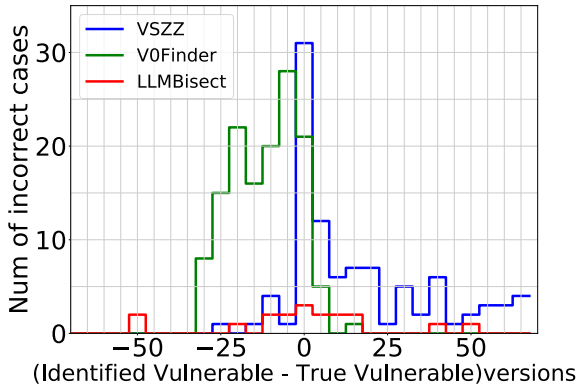

Figure 6: **Distribution of inaccurate cases over version distances**

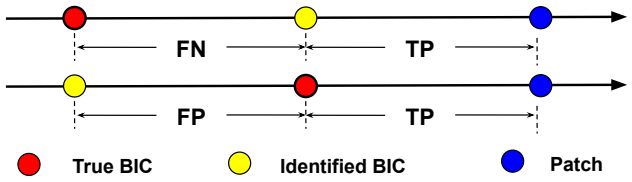

Figure 7: **Explanation of TP/FP/FN**

Fig. 6 shows the distribution of inaccurate cases for different methods in terms of FP/FN versions. The X-axis represents the number of FP or FN vulnerable versions for each case (e.g., 10 indicates a case where the method produced 10 false positive vulnerable versions, and -5 indicates a case where the method produced 5 false negative vulnerable versions). Note that, as shown in Fig. 7, a single method cannot produce both FP and FN for the same case.

We group the inaccurate cases into intervals of 5 based on their FP/FN counts and plot the number of cases in each group on the Y-axis. From the figure, we can observe that VSZZ produces a large number of false positive versions, V0Finder generates many false negative versions, whereas LLMBISECT significantly reduces both false positives and false negatives.

C. Comparison against SOTA Tools (RQ2)

As shown in Table I and Fig. 6, LLMBISECT significantly outperforms other state-of-the-art tools regarding accuracy. It achieves higher accuracy (91% compared to the 41.5% average of preceding tools in our dataset) in BIC identification and superior F1 scores (96.5% as opposed to 67.9%) compared to all prior tools. Remarkably, LLMBISECT even demonstrates much better performance than SymBisect on SymBisect's own evaluation dataset. In this section, we provide an in-depth analysis of these tools' inaccuracies and how LLMBISECT improves over them.

V0Finder. V0Finder treats the pre-patched version of functions modified in the bug-fix commit as vulnerable, it then compares it to all previous versions syntactically, by essentially a *whole-function* strict string match with certain abstraction and normalization. All identical historical versions will also be treated as vulnerable, while the BIC is the commit turning a non-vulnerable version into vulnerable. We detail V0Finder's weaknesses as follows.

Flaw 1. Similar to *Flaw 2* of VSZZ, the BIC may not make changes to the same functions as in the bug-fix commit (*e.g.,* 19 such cases in our dataset), rendering V0Finder's patch-function-based BIC probing invalid.

Flaw 2. V0Finder's syntactical similarity calculation is unaware of semantics and vulnerability logic. Consequently, it will likely identify a historical commit as the BIC wrongly, as long as it makes *any* changes (that cannot be normalized or abstracted away by V0Finder's string matching algorithm) in the function patched by the bug fix, These changes may not relate to the vulnerability at all (*e.g.,* not on the critical lines of the vulnerability) — 84 of V0Finder's inaccurate cases are due to this, or relate to but not introduce the vulnerability — 31 failure cases are due to this.

As mentioned before, LLMBISECT addresses these shortcomings by making decisions based on the understanding of the vulnerability logic with the help of LLMs and its comprehensive consideration of the patch contexts. As a result, LLMBISECT resolves 117 out of 134 V0Finder's inaccurate cases.

VSZZ. VSZZ identifies the BIC as the earliest commit that initializes the lines deleted by the bug-fix commit. If the bug-fix does not delete any lines, the commit initializing the file modified by the bug-fix will be treated as the BIC. We group VSZZ's inaccurate cases based on flaws in this heuristic algorithm and discuss how LLMBISECT addresses them.

Flaw 1. VSZZ fundamentally assumes that deleted lines in the bug-fix commit are related to the vulnerability's root cause, so the BIC must introduce these lines. However, the BIC can actually be within completely different functions (*e.g., 19* such cases in our dataset) or irrelevant to those deleted lines (*6* such

Reason	Inaccurate Cases	Solved in LLMBISECT
BIC changed different functions	19	9
Not identified critical lines	84	80
Flawed Heuristic	31	28
Total	134	117

Table V: **The reasons of V0Finder method failed**

Reason	Inaccurate Cases	Solved in LLMBISECT
BIC changed different functions	19	9
Only focus on deleted lines	28	27
Not identified critical lines	6	5
Flawed Heuristic	45	40
Total	98	81

Table VI: **The reasons of VSZZ method failed**

Reason	Inaccurate Cases	Solved in LLMBISECT
Under-constrained Symbolization	5	4
Scalability	3	3
Total	8	7

Table VII: **The reasons of SymBisect method failed**

cases in our dataset).

Flaw 2. The bug-fix commit can have no deleted lines, in this case, the heuristic of "treating the line-initialization commit as BIC" is oversimplified and highly inaccurate. 28 of bug-fix commits in our dataset have no deleted lines.

Flaw 3. The BIC may *modify* but *not initialize* the deleted lines in the bug-fix commit, violating VSZZ's heuristic. We observed *45* such cases in our dataset.

The above flaws stem from VSZZ's reliance on hardcoded, simplified, and code-oriented heuristics. LLMBISECT, on the other hand, utilizes LLM's deep and flexible understanding of vulnerability logic (*e.g.,* recognize critical lines) to identify BICs, with minimal assumptions, *e.g.,* the presence (*Flaw 2*) and significance (*Flaw 1*) of deleted lines and BIC's operation (*Flaw 3*.). Furthermore, LLMBISECT takes advantage of full patch context, including the commit messages, to extract valuable information for BIC locating, significantly addressing *Flaw 1*. As a result, LLMBISECT resolves 81 out of 98 VSZZ's inaccurate cases.

SymBisect. SymBisect decides whether a specific vulnerability affects a software version with under-constrained symbolic execution, guided by hints extracted from PoC execution traces for better scalability. Despite its reliance on PoC and limited support for vulnerability types, we identify issues impacting its accuracy on its own evaluation dataset (that we use for our comparison).

Flaw 1. Under-constrained symbolic execution assumes overly relaxed constraints (and often infeasible) of program variables unknown in its analysis scope, *e.g.,* global variables initialized outside of the local analyzed function(s). This results in over-approximation of program behaviors, for instance, a software version can wrongly be recognized as vulnerable. SymBisect fails in 5 cases in our dataset due to this reason.

Flaw 2. Symbolic execution is known to be expensive. To address the scalability issue, SymBisect utilizes information (*e.g.,* promising paths) extracted from PoC execution traces to guide its symbolic execution. However, this guide may be incomplete or inaccurate, leading to missed vulnerable paths and/or conditions, eventually causing inaccuracies in BIC identification. We observed 3 such inaccurate cases in the SymBisect evaluation dataset.

LLMBISECT, unlike SymBisect, does not rely on the expensive symbolic execution for BIC identification. Instead, its decision is based on LLM's profound understanding of the vulnerability logic, from both code changes and commit messages, avoiding the above difficulties.

D. Ablation Study (RQ3)

1) Effectiveness of Design Points: As discussed in §III-A, our final design results from multiple iterations and refinements of a baseline workflow. During this process, we adopt different effective design points that *all* improve LLMBISECT's accuracy. To demonstrate it, we start with the baseline method and gradually integrate each of our design points, observing the change in BIC detection accuracy. We show the results in Fig. 8, as can be seen, the accuracy steadily improves as more design points are adopted (*e.g.,* from 30.5% to 91%). In the remainder of this section, we detail the reasons behind these improvements by analyzing each intermediate configuration in Fig. 8.

(0) The Baseline Method. As described in §III-A, the most straightforward baseline method inspired by existing work is to let LLM inspect each commit (reverse chronologically) that touches the same function(s), i.e., candidates are generated using the patch-function-based generator alone. The first identified BIC will be output as the final result. As analyzed in §III-A, this approach has a low accuracy (30.5% in Fig. 8) mainly due to LLM's high false positive rate in single-commit BIC decision and missing true BIC with single generator.

(1) A Second Baseline Method. We also consider an alternative baseline method where we pick BIC candidates from the N most recent relevant commits that modify the patched function(s), where N is chosen such that we do not overfill the context window of the LLM. We will have the LLM pick a single commit from them as the BIC. While a plausible design, we note that it has two major conceptual limitations: 1. The most recent N relevant patches may not include the true BIC if it resides early in the long commit history. 2. A long input (near the context window limit) is known to degrade LLM's performance [22], [31], [33]. It is especially challenging given the many candidate BIC commits that are interconnected (modifying the same function). As shown in Fig. 8, this method (Baseline2) achieved an overall accuracy of 58% (116/200), outperforming the previous baseline but still falling short of our proposed approach. In 51 cases, the true BIC was absent from the LLM's candidate set, either because it was not identified as relevant or because it was excluded due to the context window limit (the first limitation noted above). In another 33 cases, the true BIC was present among the candidates, but the LLM selected incorrectly (the second limitation).

(2) Added: BIC Filtering. We then adopt the BIC comparative filtering process (§III-D), where all potential BICs are identified and then compared by the LLM to determine the most likely one. As shown in Fig. 8, this significantly improve the accuracy compared to the strawman workflow (30.5% $\rightarrow$ 77.5%).

(3) Replaced: C1 $\rightarrow$ C2. Patch-function-based candidate generation (*i.e., C1* in Fig. 8) can result in too many candidates, confusing the LLM and eventually reducing accuracy. We show that a more fine-grained critical-line-based strategy (*C2* in Fig. 8 to replace *C1*, detailed in §III-C) increases the accuracy from 77.5% to 81.5%.

(4) Added: Result Finalizer. As discussed in §III-A, critical-line-based candidate generation (*C2* in Fig. 8) is more precise, however, it can also miss true BICs if some critical lines are missed. Our solution is to combine *C2* and *C1* with the *result finalizer* (§III-E), this design further improves the accuracy (81.5% $\rightarrow$ 84% in Fig. 8).

(5) Replaced: C1+C2 $\rightarrow$ C1+C3. We also tried C1+C3 and got similar results (83.5%) compared to C1+C2.

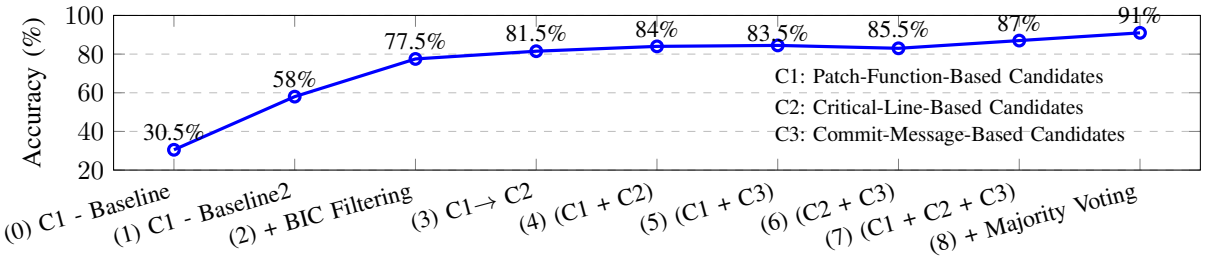

Figure 8: **Ablation Study with Different Design Points**

(6) Replaced: C1+C2 → C2+C3. C2+C3 and got similar results (85.5%) compared to C1+C2.

(7) Added: Commit-Message-Based Candidate Generation. Neither *C1* nor *C2* captures BICs having no code overlaps with the corresponding bug-fix commits, as mentioned in §III-A. We thus develop another strategy that seeks implicitly connected BICs from the commit messages of the bug-fix (*C3* in Fig. 8, detailed in §III-C). As shown in Fig. 8, this improves the accuracy to 87% from the previous configuration.

8) Added: Majority Voting. As mentioned before (§V-A), the well-known self-consistency issue of LLMs can negatively impact our accuracy, when the correct decision is not yielded in the first run. To address this, we incorporate the *majority voting* mechanism which selects the most frequent answer among multiple LLM runs in the result finalizer. This further improves LLMBISECT's accuracy compared to the previous configuration (*i.e.,* 87% → 91% in Fig. 8).

Importantly, our approach is not a simple application of LLMs. As shown above, an intuitive LLM-based method yields low accuracy (30.5%). We identified the shortcomings of such baseline approaches (outlined as four challenges) and addressed them through a structured workflow. Some design components, such as the three complementary candidate commit generators, are not LLM-specific (though one uses an LLM). We carefully assigned tasks to LLMs only where they are most effective. The ablation study confirms the advantages of this design.

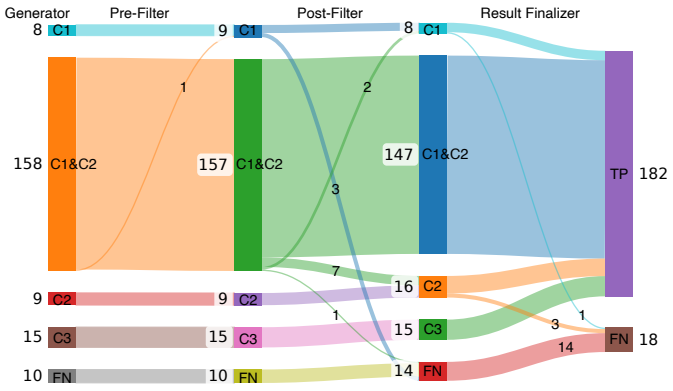

Figure 9: **Flow of Ground-Truth Bug-Inducing Commits**

2) Breakdown by Components and Phases: To more effectively illustrate the relationships between different candidate

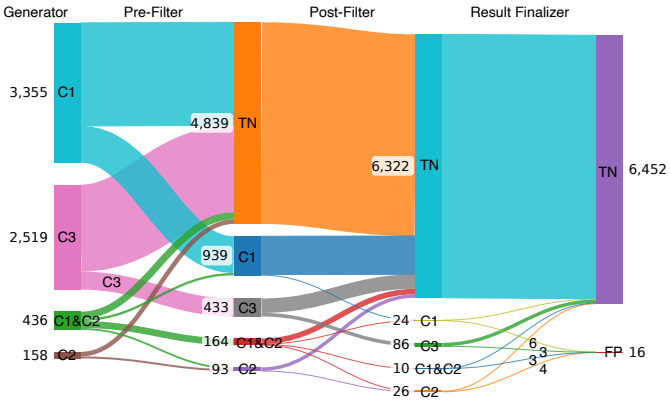

Figure 10: **Flow of False Bug-Inducing Commits**

generators and understand their individual importance, we present two Sankey diagrams in Fig. 9 and Fig. 10. Fig. 9 visualizes recall losses, showing how the correct BICs are wrongly filtered out or overlooked (*i.e.,* false negatives) by different methods at each phase. In contrast, Fig. 10 tracks how incorrect BICs (*i.e.,* false positives) are progressively pruned across multiple pipeline stages. As previously termed in Fig. 8, C1, C2 and C3 refer to the candidates produced by our three generators (*i.e.,* function-, critical-line-, and commit-message-based), respectively. The label "C1 & C2" indicates the *intersection* of C1 and C2 (*e.g.,* in Fig. 9, 158 true BICs were initially identified by both the function- and critical-line-based generators). Note that C3 by design has no overlaps with either C1 or C2, as C3 intends to identify commits that are otherwise missed by C1 and C2 (see Challenge #2 in the §III-A). It is worth mentioning that each bug has exactly one true BIC per our definition. Thus, TPs and FNs add up to 200 (*i.e.,* total number of our evaluated bugs) in Fig. 9. However, since each generator may initially identify multiple candidate BICs for a given bug, FPs in the early stages significantly exceed 200 in Fig. 10. Note that the final numbers of FN (in Fig. 9) and FP (in Fig. 10) are not equal, i.e., 18 vs. 16. This is because in two bugs, LLMBISECT failed to produce any result (none of the candidate commits were identified as the BIC). These cases are counted as FNs but not as FPs.

We have the following observations:

• C1, C2, and C3 effectively **complement each other** by covering the true BICs that the others miss, collectively

contributing to the high overall accuracy of our approach (Observation #4 in Design Section). This is shown in Fig. 9. Using only C1, the accuracy is 77.5% (155 out of 200), and using only C2, the accuracy is 81.5% (163 out of 200). For instance, 15 true BICs are *exclusively* identified by only C3 in the very first phase and retained throughout the remainder of the pipeline, contributing to a 7.5% improvement in overall accuracy. Without C3, we would have 25 instead of currently 10 missed true BICs in Phase I. This result confirms Observation #2 and highlights the effectiveness of Solution #2 described in §III-A. Similarly, both C1 and C2 have their own indispensable contributions across the pipeline, supporting the design rationale behind Solution #1.

• Strong True BIC Retention. Our pipeline design demonstrates a strong ability to retain the true BICs at every stage, achieving a high end-to-end TP rate. As shown in Fig. 9, the combined filtering across all three stages, *i.e.,* Pre-Filter, Post-Filter, and Result Finalizer, mistakenly discarded only 8 true BICs, yielding a high recall of 95.8% (182/190). Particularly noteworthy is the performance of the LLM-based method in the Pre-Filter stage: while eliminating 4,839 FP candidates (Fig. 10), it did not discard a single correct BIC initially identified by generators. Such a high recall (or low false negative) is likely due to its capacity to align high-level natural language intent (from commit messages) with low-level code modifications, a capability vital for tracing causal relationships between BICs and patches. In addition, the Post-Filter and Result Finalizer stages each missed only 4 true BICs, highlighting the powerful comparative reasoning ability of LLMs and providing strong empirical support for Observation #3.

• Effective False Positive Filtering. Fig. 10 shows that there are many incorrect BICs (*i.e.,* false positives) generated at the beginning. Specifically, the total number of FP candidates initially exceeds 6,400 across all three strategies, and the majority of FPs are contributed by C1, consistent with our Challenge/Observation/Solution #1. More than 1,600 false positives remain after Pre-Filtering — a clear reflection of Challenge #3. It is likely due to the lack of a well-defined notion of "bug-inducing commits" to LLMs — any commit that appears related to the vulnerability is considered bug-inducing. Nevertheless, our design effectively filtered out false positives (FPs) throughout the pipeline. In the Pre-Filter stage, 75% of initial FPs (*i.e.,* 4,839 candidates) are successfully filtered out. The Post-Filter stage additionally removes 1,483 FPs, and eventually, only 16 FP candidates remain after the Result Finalizer stage. We can see the contribution of false positives from C1, C2, C3 is as follows: C1-only: 6, C2-only: 4, C1&C2: 3, C3-only: 3. This result again confirms LLM's powerful comparative reasoning (Observation #3), which we effectively leverage in our pipeline design (Solution #3).)

• Internal consistency. Fig. 9 illustrates that in 147 cases, both C1 and C2 include the correct top candidate, and in all such cases, the Result Finalizer selects them correctly. Additionally, there are 8 cases where only C1 contains the correct BIC, 16 cases where only C2 does, and 15 cases where only C3 does. Overall, we observe that the correct top candidate is selected in the vast majority of cases. The only exceptions are 1 case where the top candidate appears only in C1, and 3 cases where it appears only in C2 — these candidates are correct BICs, but the Result Finalizer does not select them.

Patch Information	Inaccurate Cases	Accuracy
Commit Message+ Code Change	19	91.0%
Code Change	58	71.0%

Table VIII: **The accuracy with/without commit message**

3) The Role of Commit Messages: One of LLMBISECT's major advantages is its utilization of the full patch information, including both code changes and natural language commit messages. Besides *C3* in Fig. 8 for candidate initialization, commit messages also help LLMs make more informed decisions when inspecting each commit for BIC identification. To quantitatively understand the commit message's impact, we strip the commit messages of all commits and re-run our evaluation. Note that the impact is multi-front: (1) the commit-message-based generator basically no longer works, (2) the critical-line-based generator is substantially weaker because the LLM can no longer benefit from the commit messages to understand the logic of the bug, and (3) the selection of the BICs is also weaker because the LLM can no longer benefit from the description of the purpose of the candidate commits. As shown in Table VIII, the gap in accuracy is significant: 71% vs 91%.

4) Consistency in Result Finalizer's majority voting: Table IX presents the statistics of the majority voting after repeating the queries seven times. Note that Result Finalizer receives up to three candidate inputs, each from a different candidate generator. For 95 cases where we had three candidates, the LLM produced a vastly consistent 7:0:0 vote in 80% (76 cases). An additional 9 cases resulted in a 6:1 vote. When the number of candidates is two, the LLM produced an overwhelming 7:0 vote in 95.8% (92/96) of the cases. As the number of candidates decreases, the stability of the LLM's output increases — an observation consistent with what we discussed in Challenge #1.

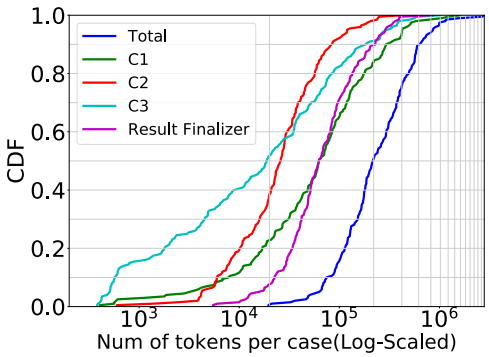

Figure 11: **Distribution of token cost per case**

E. Different LLMs

LLMBISECT's design is agnostic to the underlying LLM; Nonetheless, we conduct a comparative evaluation by swap-

Candidates Size	3	3	3	3	3	3	2	2	2	2	1	0
Count of each BIC	(7, 0, 0)	(6,1,0)	(5,2,0)	(4,3,0)	(3,2,2)	(4,2,1)	(7, 0)	(6,1)	(5,2)	(4,3)	(7)	()
Num of Cases	76	9	6	2	1	1	92	2	1	1	7	2

Table IX: **Majority voting inter-consistency**

Model	Inaccurate Cases	Accuracy
OpenAI o1	19	91.0%
GPT-4o	69	65.5%
LLama3.2	58	71.0%

Table X: **The accuracy with different LLM models**

ping between three widely used LLMs: OpenAI o1, GPT-4o, and LLama 3, covering both commercial and open-source models. The evaluation results are summarized in Table X. As can be seen, OpenAI o1 achieves the highest accuracy (91%) likely due to its enhanced reasoning capability, followed by LLama 3 (71%) and GPT-4o (65.5%). We found that the majority of inaccuracies occur during the process of comparing multiple suspected BICs and selecting the final result (specifically, during the Post-Filtering and Result Finalizer stages). For example, GPT-4o produced a total of 49 inaccurate cases across these two steps. This suggests a gap for different models in tasks which requires extensive reasoning on multiple code snippets and commit message.

F. Token cost (RQ4)

For each case using the O1 model, the minimum token usage was 19,557 tokens (approximately $0.30 at the time of writing), and the maximum was 2,772,509 tokens (approximately $41). The median was 219,523 tokens (approximately $3.3 with the o1 model), and the average was 330,426 tokens (approximately $4.9). Overall, this cost is acceptable compared to the expense of hiring professionals for manual analysis and the potential risks posed by N-day vulnerabilities.

We further break down the cost by methods, which is shown in Fig. 11. As expected, the patch-function-based method (that generates C1) consumes significantly more tokens than the critical-line-based method and the commit-message-based method. The Result Finalizer stage also consumes a significant number of tokens because of the majority voting that repeats the experiment seven times.

G. Case Study.

We revisit the example in Fig.1 to illustrate how LLMBISECT identifies the correct bug-inducing commit (BIC), highlighting key features, the strengths and limitations of LLMs, and insights into their behavior.

Initially, generators C1, C2, and C3 produce 3, 2, and 35 BIC candidates, respectively. Only C3 includes the correct BIC, as it appears in the commit message but modifies different functions/structs than the patch, making it inaccessible to C1 and C2.

High False Positives. After the Pre-Filter step, 1, 1, and 20 candidates remain from C1, C2, and C3, respectively. The LLM flags all 22 as potential BICs, reflecting its tendency toward high false positives (Design Challenge #3). For example, commit c19ffe00fed6 is incorrectly identified as a candidate BIC because it significantly refactored the patched function `gsm_modem_update()` (see Appendix Fig.2).

Comparative Reasoning Ability. In the Post-Filter stage, LLMBISECT selects the most likely BIC among candidates. The LLM correctly picks the true BIC from the 20 candidates in C3, demonstrating strong comparative reasoning. Rather than making isolated binary decisions, it evaluates relative differences, showing a deeper understanding of vulnerability logic in context.

In the Result Finalizer stage, LLMBISECT selects the correct BIC from the three final candidates using majority voting — all seven runs agree. The LLM's explanation aligns closely with the actual root cause:

"The patch description clearly indicates that the crash occurs because data can still be queued and the 'kick_timer' restarted after gsm_cleanup_mux() has already begun tearing down resources. ..... c568f7086c6e, is where the problematic timer-based re-queuing mechanism is added, but does not check if `gsm->dead` *is set. This allowed updates (and modem status line changes) to be triggered after teardown, hence causing the race condition that the final patch fixes."*

Importance of Both Commit Messages and Code Changes. A core strength of LLMBISECT is its ability to integrate information from both commit messages and code. Following the ablation study on the importance of commit messages, we find that removing either commit messages or code changes as input will lead the LLM to misidentify c19ffe00fed6 as the BIC in all seven runs. Below is an example LLM response that incorrectly summarizes the behavior of the true BIC.

"c568f7086c6e focuses on adding a kick_timer to handle data transmission logic and does not appear to introduce the risk of calling gsm_modem_update after the gsm mux is marked dead."

VI. LIMITATIONS AND DISCUSSION

Incomplete BIC candidate generation. As reported in Table IV, there are 10 failure cases where our solution simply failed in the candidate generation phase. For 8 of them, the file containing the true BIC is not explicitly mentioned in the patch. Our current design is unable to handle such cases. One potential strategy to solve this is to expand the scope of code context based on a dependency analysis, e.g., slicing, to identify more relevant functions beyond the patched ones. One can also extract less precise hints in the commit message (e.g., mentioning a module name) to constrain the search space. Retrieval-Augmented Generation (RAG) is one potential solution to extract additional relevant code context automatically. We leave addressing these corner cases as future work.

Non-determinism. As with most LLM-based solutions, the results are inherently non-deterministic due to sampling during generation. To mitigate this, common prompt strategies such as majority voting are often used, and we adopt this strategy in our work as well. In cases where the LLM exposes confidence scores or output probabilities, an alternative is to select the response with the highest likelihood, though this depends on the interfaces provided by specific models.

Dependency on the quality of commit messages. As we show in §V-D3, commit messages indeed provide significant benefits to the overall accuracy of the solution. On the flip side, it also means that our solution depends on the quality of the commit messages. As a result, we expect to see degraded performance when our solution is applied to projects where the commit messages are not as informative as those in the Linux kernel. Some possible mitigations are (1) perform better prompt engineering to extract more description about the bug before conducting bug bisection, and (2) leverage RAG to obtain more information about the patches and the bugs, e.g., from mailing lists or other sources.

Advanced LLM Post-training & Prompt Techniques. Several proven techniques can further boost the performance and cost-efficiency of LLMBISECT. Fine-tuning and *Reinforcement learning from human feedback* (RLHF) can align responses to our intent [35]. As mentioned above, *Retrieval-Augmented Generation* (RAG) [28] can fetch the most relevant context (e.g., functions, relevant documents, mailing list threads) to provide additional code context or bug-related information, thereby improving performance. Advanced prompt and context design, such as Chain-of-Thought [45] and *Multi-agent LLM architectures* [40] improve LLM in reasoning without extra training. These techniques could help the migration of LLMBISECT to weaker models and save costs. We leave the exploration of these directions to future work.

VII. Related Work

The Application of LLMs in Program Analysis. Recent research has explored the integration of LLMs into static analysis to enhance its effectiveness in code comprehension and bug findings [44], [42], [29], [43]. LLMs have also been employed to understand and generate code comments, documentation, and system logs, improving code readability and maintainability [25], [30], [20]. The integration of LLMs in program analysis represents a significant advancement in software engineering, offering tools that enhance productivity, code quality, and security.

PoC-based vulnerable version identification. SymBisect [52] leverages under-constrained symbolic execution to determine whether a specific software version contains a given vulnerability, enabling the identification of BICs. However, SymBisect supports only specific types of functions and requires an existing PoC. Dai et al.[17] proposed a PoC migration approach that takes an initial PoC as input and adapts it to identify other affected versions; however, it is specifically designed for user-space programs.

SZZ Methods. SZZ (short for Śliwerski, Zimmermann, and Zeller) [39] is an algorithm designed to identify bug-inducing commits in version control systems, also called B-SZZ. It identifies earlier changes at the location of a bug fix as bug-inducing commits. However, its straightforward approach struggles to handle complex bugs effectively. To address this limitation, AG-SZZ [27] incorporates an annotation graph to exclude non-semantic changes, such as whitespace, comments, and formatting adjustments, thereby reducing false positives. MA-SZZ [16] further improves on this by filtering out meta-changes like branch modifications and file attribute updates, ensuring that only source code changes are analyzed. V-SZZ [13] expands the algorithm's scope by targeting vulnerabilities introduced in earlier software versions. NEURAL-SZZ [41] leverages a Heterogeneous Graph Attention Network (HAN) to capture semantic relationships between lines of code, enhancing precision in tracing bug origins. However, it is limited to Java and exhibits a relatively high false positive rate. Combining advanced techniques like NEURAL-SZZ and V-SZZ can significantly improve bug-tracing accuracy, while AG-SZZ and MA-SZZ remain practical solutions for simpler scenarios.

Vulnerable code clone detection. Vulnerable code clone detection is a specialized type of code clone detection [10], [36], [37], [38], [19]. It involves identifying pieces of source code in software systems that are similar to or identical to code fragments known to have security vulnerabilities. They usually perform similarity comparisons on what they define as vulnerability-related code (usually a few lines within the patch function or the entire function) [26], [23], [15], [50], [54], [15], [48]. However, rule-based code extraction and similarity-based solutions often fail to identify vulnerability-relevant code or confirm the presence of vulnerabilities, as they lack vulnerability comparison based on logical structures. Our evaluation demonstrates that such methods perform poorly on complex programs such as the Linux kernel.

VIII. Conclusion

In conclusion, we introduced LLMBISECT, a novel, LLM-driven bug bisection pipeline that effectively pinpoints bug-inducing commits in Linux kernel. By combining both code changes and commit-message insights, LLMBISECT overcomes the limitations of traditional patch-based methods, which often fail to capture the true scope and context of a vulnerability. Our results underscore the potential of large language models to streamline vulnerability detection, reducing the window in which attacks can occur.

IX. Ethics Considerations

This research was conducted in alignment with recognized ethical guidelines, ensuring responsible practices in methodology, data handling, and reporting. Our work aims to enhance bug bisection and vulnerability identification in open-source software, ultimately helping developers and maintainers address security threats more effectively.

Our research is based on N-day vulnerabilities—specifically, bugs that have already been patched in the Linux mainline. Furthermore, we do not anticipate any adverse impact on individuals or groups, as our analysis is strictly limited to publicly available codebases and does not involve any personal or sensitive data.

In developing LLMBISECT, we utilized both closed-source and open-source large language models without incorporating any copyrighted or sensitive material.

ACKNOWLEDGMENT

We thank the anonymous reviewers for their insightful comments and valuable suggestions. This material is based upon work supported by the National Science Foundation under Grant No. #2155213 & #2247881 and the Defense Advanced Research Projects Agency (DARPA) under Agreement No. HR00112590041.

REFERENCES

[1] Data for May 2024 CVE. https://github.com/jgamblin/monthlyCVEStats/blob/main/2024/May/May2024.ipynb.
[2] Fixes Tag. https://docs.kernel.org/process/submitting-patches.html.
[3] kernel.org Added as CVE Numbering Authority (CNA). https://www.cve.org/Media/News/item/news/2024/02/13/kernel-org-Added-as-CNA.
[4] LLMBisect repo. https://github.com/seclab-ucr/LLMBisect.
[5] National Vulnerability Database. https://nvd.nist.gov/.
[6] Syzbot Bisection Motivation. https://lore.kernel.org/all/CACT4Y+Y3nN=nLEkHXLFcX7vxp_vs1JrD=8auJ3cX9we6TQHO+w@mail.gmail.com/T/#u.
[7] The kernel becomes its own CNA. https://lwn.net/Articles/961961/.
[8] V0Finder Source Code. https://github.com/WOOSEUNGHOON/V0Finderpublic.
[9] VSZZ Source Code. https://figshare.com/ndownloader/files/31748777.
[10] Q. U. Ain, W. H. Butt, M. W. Anwar, F. Azam, and B. Maqbool. A systematic review on code clone detection. *IEEE access*, 7:86121–86144, 2019.
[11] G. An, J. Hong, N. Kim, and S. Yoo. Fonte: Finding bug inducing commits from failures. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 589–601. IEEE, 2023.
[12] A. Anwar, A. Abusnaina, S. Chen, F. Li, and D. Mohaisen. Cleaning the nvd: Comprehensive quality assessment, improvements, and analyses. *IEEE Transactions on Dependable and Secure Computing*, 19(6):4255–4269, 2021.
[13] L. Bao, X. Xia, A. E. Hassan, and X. Yang. V-szz: automatic identification of version ranges affected by cve vulnerabilities. In *Proceedings of the 44th International Conference on Software Engineering*, pages 2352–2364, 2022.
[14] R. Bhagwan, R. Kumar, C. S. Maddila, and A. A. Philip. Orca: Differential bug localization in {Large-Scale} services. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 493–509, 2018.
[15] B. Bowman and H. H. Huang. Vgraph: A robust vulnerable code clone detection system using code property triplets. In *2020 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 53–69. IEEE, 2020.
[16] D. A. da Costa, S. McIntosh, W. Shang, U. Kulesza, R. Coelho, and A. E. Hassan. A framework for evaluating the results of the szz approach for identifying bug-introducing changes. *IEEE Trans. Softw. Eng.*, 2017.
[17] J. Dai, Y. Zhang, H. Xu, H. Lyu, Z. Wu, X. Xing, and M. Yang. Facilitating vulnerability assessment through poc migration. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, pages 3300–3317, 2021.
[18] C. Elbaz, L. Rilling, and C. Morin. Fighting n-day vulnerabilities with automated cvss vector prediction at disclosure. In *Proceedings of the 15th International Conference on Availability, Reliability and Security*, pages 1–10, 2020.
[19] C. Fang, Z. Liu, Y. Shi, J. Huang, and Q. Shi. Functional code clone detection with syntax and semantics fusion learning. In *Proceedings of the 29th ACM SIGSOFT international symposium on software testing and analysis*, pages 516–527, 2020.
[20] M. Geng, S. Wang, D. Dong, H. Wang, G. Li, Z. Jin, X. Mao, and X. Liao. Large Language Models are Few-Shot Summarizers: Multi-Intent Comment Generation via In-Context Learning. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ICSE '24, pages 1–13, New York, NY, USA, February 2024. Association for Computing Machinery.
[21] Google. Google syzbot. https://syzkaller.appspot.com/upstream/.
[22] C.-P. Hsieh, S. Sun, S. Kriman, S. Acharya, D. Rekesh, F. Jia, Y. Zhang, and B. Ginsburg. Ruler: What's the real context size of your long-context language models? *arXiv preprint arXiv:2404.06654*, 2024.
[23] J. Jang, A. Agrawal, and D. Brumley. Redebug: finding unpatched code clones in entire os distributions. Oakland'12.
[24] Y. Jiang, M. Jeusfeld, and J. Ding. Evaluating the data inconsistency of open-source vulnerability repositories. In *Proceedings of the 16th International Conference on Availability, Reliability and Security*, pages 1–10, 2021.
[25] Z. Jiang, J. Liu, Z. Chen, Y. Li, J. Huang, Y. Huo, P. He, J. Gu, and M. R. Lyu. LILAC: Log Parsing using LLMs with Adaptive Parsing Cache. *Proceedings of the ACM on Software Engineering*, 1(FSE):137–160, July 2024.
[26] S. Kim, S. Woo, H. Lee, and H. Oh. Vuddy: A scalable approach for vulnerable code clone discovery. Oakland'17.
[27] S. Kim, T. Zimmermann, K. Pan, and E. J. Jr. Whitehead. Automatic identification of bug-introducing changes. In *21st IEEE/ACM International Conference on Automated Software Engineering (ASE'06)*, 2006.
[28] P. Lewis, E. Perez, A. Piktus, F. Petroni, V. Karpukhin, N. Goyal, H. Küttler, M. Lewis, W.-t. Yih, T. Rocktäschel, S. Riedel, and D. Kiela. Retrieval-augmented generation for knowledge-intensive nlp tasks. *arXiv preprint arXiv:2005.11401*, 2020.
[29] 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 (PACMPL), Volume 8, Issue OOPSLA1*, 8(OOPSLA1), 2024.
[30] J. Li, D. Faragó, C. Petrov, and I. Ahmed. Only diff Is Not Enough: Generating Commit Messages Leveraging Reasoning and Action of Large Language Model. *Proceedings of the ACM on Software Engineering*, 1(FSE):745–766, July 2024.
[31] T. Li, G. Zhang, Q. D. Do, X. Yue, and W. Chen. Long-context llms struggle with long in-context learning, 2024. *URL https://arxiv.org/abs/2404.02060*.
[32] X. Li, Z. Zhang, Z. Qian, T. Jaeger, and C. Song. An investigation of patch porting practices of the linux kernel ecosystem. In *2024 IEEE/ACM 21st International Conference on Mining Software Repositories (MSR)*, pages 63–74. IEEE, 2024.
[33] 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.
[34] E. C. Neto, D. A. da Costa, and U. Kulesza. The impact of refactoring changes on the szz algorithm: An empirical study. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2018.
[35] L. Ouyang, J. Wu, X. Jiang, D. Almeida, C. L. Wainwright, P. Mishkin, C. Zhang, S. Agarwal, K. Slama, A. Ray, J. Schulman, J. Hilton, F. Kelton, L. Miller, M. Simens, A. Askell, P. Welinder, P. F. Christiano, J. Leike, and R. Lowe. Training language models to follow instructions with human feedback. *Advances in Neural Information Processing Systems*, 35:27730–27744, 2022.
[36] C. K. Roy, J. R. Cordy, and R. Koschke. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Science of computer programming*, 74(7):470–495, 2009.
[37] H. Sajnani, V. Saini, J. Svajlenko, C. K. Roy, and C. V. Lopes. Sourcerercc: Scaling code clone detection to big-code. In *Proceedings of the 38th International Conference on Software Engineering*, pages 1157–1168, 2016.
[38] G. Shobha, A. Rana, V. Kansal, and S. Tanwar. Code clone detection—a systematic review. *Emerging Technologies in Data Mining and Information Security: Proceedings of IEMIS 2020, Volume 2*, pages 645–655, 2021.
[39] J. Śliwerski, T. Zimmermann, and A. Zeller. When do changes induce fixes? *ACM sigsoft software engineering notes*, 30(4):1–5, 2005.

[40] SuperAnnotate. Llm agents: The ultimate guide 2025. *SuperAnnotate Blog*, March 2025. 8min read (approx.).
[41] L. Tang, L. Bao, X. Xia, and Z. Huang. Neural szz algorithm. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1024–1035, 2023.
[42] N. Wadhwa, J. Pradhan, A. Sonwane, S. P. Sahu, N. Natarajan, A. Kanade, S. Parthasarathy, and S. Rajamani. CORE: Resolving Code Quality Issues using LLMs. *Proceedings of the ACM on Software Engineering*, 1(FSE):789–811, July 2024.
[43] C. Wang, Y. Gao, W. Zhang, X. Liu, Q. Shi, and X. Zhang. LLMSA: A Compositional Neuro-Symbolic Approach to Compilation-free and Customizable Static Analysis, December 2024. arXiv:2412.14399 [cs].
[44] C. Wang, W. Zhang, Z. Su, X. Xu, and X. Zhang. Sanitizing Large Language Models in Bug Detection with Data-Flow. In Y. Al-Onaizan, M. Bansal, and Y.-N. Chen, editors, *Findings of the Association for Computational Linguistics: EMNLP 2024*, pages 3790–3805, Miami, Florida, USA, November 2024. Association for Computational Linguistics.
[45] J. Wei, X. Wang, D. Schuurmans, M. Bosma, B. Ichter, F. Xia, E. H. Chi, Q. V. Le, and D. Zhou. Chain-of-thought prompting elicits reasoning in large language models. *arXiv preprint arXiv:2201.11903*, 2022.
[46] M. Wen, R. Wu, and S.-C. Cheung. Locus: Locating bugs from software changes. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, pages 262–273, 2016.
[47] S. Woo, H. Hong, E. Choi, and H. Lee. {MOVERY}: A precise approach for modified vulnerable code clone discovery from modified {Open-Source} software components. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 3037–3053, 2022.
[48] S. Woo, D. Lee, S. Park, H. Lee, and S. Dietrich. V0finder: Discovering the correct origin of publicly reported software vulnerabilities. In *USENIX Security Symposium*, pages 3041–3058, 2021.
[49] J. Wunder, A. Corona, A. Hammer, and Z. Benenson. On nvd users' attitudes, experiences, hopes, and hurdles. *Digital Threats: Research and Practice*, 5(3):1–19, 2024.
[50] Y. Xiao, B. Chen, C. Yu, Z. Xu, Z. Yuan, F. Li, B. Liu, Y. Liu, W. Huo, W. Zou, et al. {MVP}: Detecting vulnerabilities using {Patch-Enhanced} vulnerability signatures. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 1165–1182, 2020.
[51] H. Xu, S. Wang, N. Li, K. Wang, Y. Zhao, K. Chen, T. Yu, Y. Liu, and H. Wang. Large Language Models for Cyber Security: A Systematic Literature Review, July 2024. arXiv:2405.04760 [cs].
[52] Z. Zhang, Y. Hao, W. Chen, X. Zou, X. Li, H. Li, Y. Zhai, and B. Lau. {SymBisect}: Accurate bisection for {Fuzzer-Exposed} vulnerabilities. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 2493–2510, 2024.
[53] Z. Zhang, H. Zhang, Z. Qian, and B. Lau. An investigation of the android kernel patch ecosystem. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 3649–3666, 2021.
[54] D. Zou, H. Qi, Z. Li, S. Wu, H. Jin, G. Sun, S. Wang, and Y. Zhong. Scvd: A new semantics-based approach for cloned vulnerable code detection. In *DIMVA*, pages 325–344. Springer, 2017.

APPENDIX: DETAILED PROMPT IMPLEMENTATION

1) **Functionality:** `Identify critical lines from patches with deleted lines.`
Role: You are an experienced Linux program analysis expert. I am working on analyzing the Linux kernel patches that fix vulnerabilities. I will give you the content of the patch. You should return the most important and representative lines which are deleted in the patch.
Parameters:
   - The patch is: `${patch_content}`
   - The complete functions before the patch are: `${function_content}`

   Instruction: Considering the purpose of the patch, from the lines which are deleted in the patch, pick the `${num_lines}` important and representative lines which are closely related to the logic of the vulnerability.

   Output Format: Output the above most important and representative lines in the format of a Python list `[]`, each element in the list is a tuple `(filename, functionname, linenum, line)`. The `linenum` is the line number inside the corresponding function. When printing the elements, each element is printed in one line instead of multiple lines.
2) **Functionality:** `Identify critical lines from patches with only added lines.`
Role: You are an experienced Linux program analysis expert. I am working on analyzing the Linux kernel patches that fix vulnerabilities. I will give you the content of the patch and You should return the most important and representative lines.
Parameters:
   - The patch is: `${patch_content}`
   - The complete functions before the patch are: `${function_content}`

   Instruction: Considering the purpose of the patch, list important and representative lines with the corresponding functions before the patch is applied. These lines must have data dependency with the added lines in the patch.
   Output Format: Output the above most important and representative lines that exist before the patch in the format of a Python list `[]`, each element in the list is a tuple `(filename, functionname, linenum, line)`. The `linenum` is the line number inside the corresponding function. When printing the elements, each element is printed in one line instead of multiple lines.
3) **Functionality:** `Identify critical lines from patches with only reordered lines.`
Role: You are an experienced Linux program analysis expert. I am working on analyzing the Linux kernel patches that fix vulnerabilities. I will give you the content of the patch and You should return the most important and representative lines.
Parameters:
   - The patch is: `${patch_content}`
   - The complete functions before the patch are: `${function_content}`
   - The identified reordered lines before the patch are: `${reorder lines}`

   Instruction: Considering the purpose of the patch, list important and representative lines with the corresponding functions before the patch is applied. These lines must have data dependency with the reordered lines in the patch.
   Output Format: Output the above most important and representative lines that exist before the patch in the format of a Python list `[]`, each element in the list is a tuple `(filename, functionname, linenum,`

`line)`. The `linenum` is the line number inside the corresponding function. When printing the elements, each element is printed in one line instead of multiple lines.

4) **Functionality:** `Pre-Filtering`
Role: You are an experienced Linux program analysis expert. I am working on analyzing the Linux kernel patches that fix vulnerabilities. I will give you the content of the patch and the content of a previous commit. You should analyze the patch and understand the logic of the corresponding vulnerability, then determine whether the given commit introduced the vulnerability.
Parameters:
   - The patch is: `${patch_content}`
   - The content of a previous commit: `${commit_content}`

   Instruction: Analyzing the logic of the patch, determine whether the given commit introduced the vulnerability.
   Output Format:
   - If so, return `True`, otherwise return `False`.
   - If you return `True`, please also explain the reason why you think the commit introduced the vulnerability.

5) **Functionality:** `Post-Filtering and Result-Finalization`
Role: You are an experienced Linux program analysis expert. I am working on analyzing the Linux kernel patches that fix vulnerabilities. I will give you the content of the patch (and the corresponding complete function definitions before the patch), also I will provide a list of previous commits (and the corresponding complete function definitions before each commit). You should analyze the patch and understand the logic of the corresponding vulnerability, then determine which commit among the list introduced the vulnerability.
Parameters:
   - The patch is: `${patch_content}`
   - The below are the lists of previous commits: `${commit_content}`

   Instruction: Analyzing the logic of the patch, determine which commit among the list introduced the vulnerability.