BugScope: Learn to Find Bugs Like Human

Jinyao Guo[1], Chengpeng Wang[1], Dominic Deluca[1], Jinjie Liu[2], Zhuo Zhang[3], Xiangyu Zhang[1]
[1]Purdue University [2]University of Southern California [3]Columbia University
{guo846, wang6590, djdeluca, xyzhang}@purdue.edu
jinjie.liu@usc.edu, zz3474@columbia.edu

Abstract

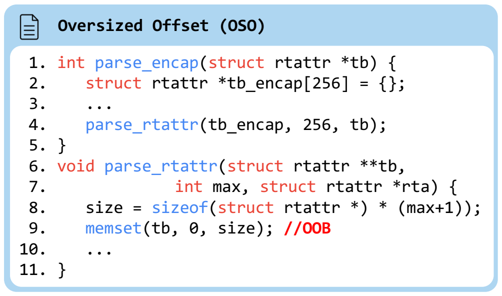
Detecting software bugs remains a fundamental challenge due to the extensive diversity of real-world bugs. Traditional static analysis tools typically rely on symbolic workflows, which limits their coverage and makes it difficult to adapt to customized bugs with diverse anti-patterns. Although recent advancements leverage large language models (LLMs) to improve bug detection, these methods still face significant difficulties in addressing sophisticated bugs and often operate within restricted analysis contexts. To overcome these limitations, we propose BugScope, an LLM-driven multi-agent that mirrors how human auditors learn new bug patterns from representative examples and apply this knowledge during code auditing. Given a set of examples that demonstrate buggy and non-buggy behaviors, BugScope first synthesizes a retrieval strategy to effectively gather relevant detection contexts through slicing, and then generates a tailored detection prompt to facilitate accurate reasoning by the LLM. Our evaluation, conducted on a curated dataset comprising 40 real-world bugs collected from 21 widely-used open-source projects, shows that BugScope achieves 87.04% precision and 90.00% recall, outperforming state-of-the-art industrial tools by 0.44 in F1 score. Further evaluation on large-scale open-source projects, including the Linux kernel, uncovered 141 previously unknown bugs, of which 78 have been fixed and 7 have been confirmed by developers, underscoring its significant practical impact.

1 Introduction

Security bugs remain a critical threat to modern software systems, leading to severe failures such as memory exhaustion from leaks or system crashes due to null pointer dereferences. The Common Weakness Enumeration (CWE), a widely adopted industry standard, categorizes software weaknesses into over 900 distinct types MITRE (2025a), each reflecting a violation of fundamental software properties. Detecting this vast spectrum of bugs typically demands highly specialized analyzers, resulting in significant and ongoing engineering costs Johnson et al. (2013).

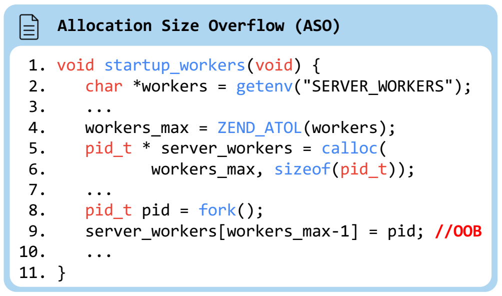
More critically, the complexity of bug detection is not limited to categorical diversity. Within a single class, bugs often arise in diverse semantic contexts and manifest through varied *anti-patterns*, further complicating detection. For instance, as shown in Figure 1, an out-of-bounds (OOB) bug may stem from an oversized offset (OSO) exceeding buffer size (line 9 in Figure 1(a)) or from an allocation size overflow (ASO) caused by integer overflow (lines 5–6 in Figure 1(b)). Detecting both cases would either require a general-purpose OOB analyzer capable of sophisticated program reasoning to relate offsets to buffer bounds, or necessitate manually crafting ad-hoc detectors for each anti-pattern. However, the former approach suffers from the trade-off between precision, recall, and scalability due to the undecidability of program behaviors Reps (2000), while the latter, though more tractable, incurs unsustainable engineering overhead given the thousands of possible anti-patterns.

The challenge further deepens with the emergence of system-specific anti-patterns that deviate from well-studied canonical forms. Figure 1(d) presents such a case in the Linux kernel, where missing validation of a user-controlled data structure, `i2c_data`, allows a high or zero value in `data->block[0]` to trigger an OOB bug (line 6) or a divide-by-zero (DBZ) bug (line 8), respectively. Therefore, the three-layered diversity, from bug classes to their generic and system-specific anti-patterns, poses a fundamental challenge for automated bug detection, underscoring the urgent need for detection techniques that can generalize across various anti-patterns in real-world scenarios.



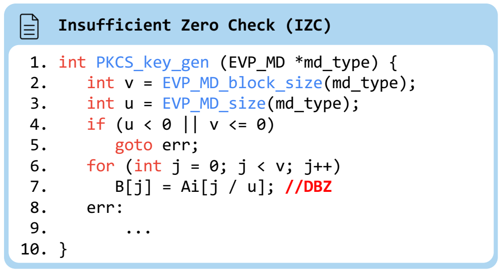
```c
int parse_encap(struct rtattr *tb) {
   struct rtattr *tb_encap[256] = {};
   ...
   parse_rtattr(tb_encap, 256, tb);
}
void parse_rtattr(struct rtattr **tb,
                  int max, struct rtattr *rta) {
   size = sizeof(struct rtattr *) * (max+1));
   memset(tb, 0, size); //OOB
   ...
}
```

(a) An example of the OSO anti-pattern



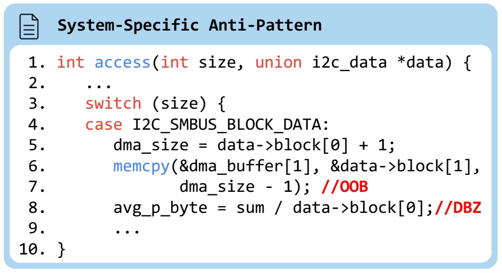
```c
void startup_workers(void) {
   char *workers = getenv("SERVER_WORKERS");
   ...
   workers_max = ZEND_ATOL(workers);
   pid_t * server_workers = calloc(
          workers_max, sizeof(pid_t));
   ...
   pid_t pid = fork();
   server_workers[workers_max-1] = pid; //OOB
   ...
}
```

(b) An example of the ASO anti-pattern



```c
int PKCS_key_gen (EVP_MD *md_type) {
   int v = EVP_MD_block_size(md_type);
   int u = EVP_MD_size(md_type);
   if (u < 0 || v <= 0)
      goto err;
   for (int j = 0; j < v; j++)
      B[j] = Ai[j / u]; //DBZ
   err:
      ...
}
```

(c) An example of the IZC anti-pattern



```c
int access(int size, union i2c_data *data) {
   ...
   switch (size) {
   case I2C_SMBUS_BLOCK_DATA:
      dma_size = data->block[0] + 1;
      memcpy(&dma_buffer[1], &data->block[1],
             dma_size - 1); //OOB
      avg_p_byte = sum / data->block[0];//DBZ
      ...
}
```

(d) An example of the system-specific anti-pattern

Figure 1: The examples of anti-patterns causing various types of bugs

Limitations of Previous Studies. Despite significant advancements in bug detection techniques, existing approaches still face critical limitations in addressing diverse software bugs. First, traditional static analyzers, such as Meta Infer Meta (2025) and CodeQL GitHub (2025), heavily rely on handcrafted symbolic rules, which are effective for detecting specific, well-known anti-patterns but lack flexibility for novel or system-specific ones, such as the example shown in Figure 1(d). Second, large language model (LLM)-based detectors like Cursor BugBot Cursor (2025) and CodeRabbit CodeRabbit (2025) demonstrate strong semantic reasoning capabilities but struggle with uncommon anti-patterns and restricted detection contexts. As shown in Section 4.3, their detection accuracy for anti-patterns like oversized offset and insufficient zero check (IZC) (Figure 1(a) and (c)) remains low, with recall not exceeding 20.00%. Third, neuro-symbolic approaches that integrate LLMs with traditional static analysis techniques Li et al. (2025b); Yang et al. (2025); Li et al. (2025a); Wang et al. (2024); Guo et al. (2025) partially mitigate these challenges but are still constrained by rigid, pipeline-based symbolic analysis, primarily focusing on anti-patterns related to data-flows. Hence, it is necessary to develop a more flexible, general-purpose bug detection technique capable of effectively identifying diverse and complex anti-patterns in real-world software.

Our Insight. While the code reasoning abilities of LLMs have been actively explored in recent static analysis research, their *learn-by-example* capability Brown et al. (2020); Dong et al. (2022) remains largely underutilized. We argue that fully leveraging this capability can yield a detection strategy akin to that used by human experts when confronting the diversity of software bugs. In practice, human code auditors study historical bug reports and patches, internalizing recurring anti-patterns (both generic and system-specific) through repeated exposure. Informed by this accumulated knowledge, they scan new code for suspicious anti-patterns, retrieve the related context when needed, and engage in deep semantic reasoning to verify the presence of bugs. Hence, our key insight is to *replicate this expert workflow by harnessing both the learning and reasoning capabilities of LLMs*. We hypothesize that, given a small number of well-curated bug examples, an LLM can generalize the underlying anti-pattern and effectively detect similar bugs in previously unseen code, even when those bugs appear in varied forms or complex contexts.

Our Approach. Based on the above insight, we propose BugScope, an autonomous LLM-powered multi-agent that emulates human auditors' mindset of learning anti-patterns from examples and applying that knowledge during post-learning code audits. Technically, BugScope comprises two collaborative components, namely the *context retrieval agent* and the *bug detection agent*, which together automate the end-to-end auditing process. Instead of relying on handcrafted rules, both agents automatically derive the analysis logic from code examples labeled with and without specific anti-patterns. Specifically, the context retrieval agent infers a *retrieval strategy* that identifies code fragments likely to exhibit the subject anti-patterns, and the bug detection agent constructs a *detection prompt* to evaluate whether retrieved snippets conform to these anti-patterns. During auditing, the context retrieval agent first localizes suspicious code snippets, collecting relevant surrounding context through forward or backward retrieval according to the synthesized retrieval strategy. The bug detection agent then applies the detection prompt to verify whether the anti-pattern is actually present, where an LLM-based validation step is applied to mitigate hallucinations.

Dataset. We curate a dataset consisting of 40 real-world bugs collected from 21 widely-used open-source projects, including prominent repositories such as OpenSSL, Git, and the Linux kernel, averaging approximately 29K GitHub stars and 81K lines of code each. Each case in the dataset provides the historical commit and exact bug location. The selected bugs span three representative categories: memory leak (MLK), divide-by-zero (DBZ), and out-of-bounds (OOB) bugs, covering seven distinct anti-patterns, respectively. Notably, these categories encompass a broad spectrum of bug semantics over various program constructs, including numeric variables, pointers, and buffers. This semantic diversity establishes our dataset as a robust benchmark for assessing the performance and adaptability of bug detectors.

Results. We evaluate BUGSCOPE using three recent reasoning models, namely Claude 3.7 Sonnet Thinking, OpenAI o4-mini, and DeepSeek-R1. Experimental results demonstrate that BUGSCOPE consistently outperforms the state-of-the-art LLM-driven code auditor RepoAudit as well as commercial static analysis tools, including Meta Infer, Cursor Bugbot, and CodeRabbit. When powered by Claude 3.7 Sonnet Thinking, BUGSCOPE achieves 87.04% precision, 90.00% recall, and an F1 score of 0.88. In contrast, the highest F1 score among the baselines is only 0.40. For specific anti-patterns, such as negative offset and insufficient zero check, the baselines detect at most one true positive and several of them even fail to identify any buggy cases at all. When deployed on large-scale, real-world codebases, BUGSCOPE uncovers 141 previously unknown bugs, 78 of which have already been fixed and 7 confirmed by developers. Notably, twelve bugs in the Linux kernel discovered by BUGSCOPE have been either fixed or acknowledged by maintainers. These results highlight the practical effectiveness and wide applicability of BUGSCOPE in real-world software auditing[1].

2 MOTIVATION

This section highlights the diversity of software bugs, discusses key limitations of current approaches, and outlines our vision for learning to find bugs like human.

2.1 DIVERSITY OF SOFTWARE BUGS

Real-world software bugs exhibit substantial diversity. Continuous reports from bug discovery platforms such as OSS-Fuzz Google (2025) and CVE MITRE (2025b) indicate that modern software systems suffer from a broad spectrum of bugs. Even within a single type of bug, its manifestation can vary considerably and even be system-specific. Such diversity poses a fundamental challenge to bug detectors: No single detection logic or template can generalize across such varied anti-patterns. In what follows, we present four representative anti-patterns drawn from open-source projects.

Figures 1(a) and (b) illustrate two OOB bugs caused by different anti-patterns. In Figure 1(a), an incorrectly calculated and thus oversized offset (`size` at line 8) in a `memset` causes zeroes to be written beyond the bounds of an array. Figure 1(b) presents an OOB bug due to an allocation size overflow. When the external input value `workers_max` (from `workers`) causes `workers_max * sizeof(pid_t)` in `calloc` to exceed the maximum value of `usize`, an integer overflow wraps the result to a small value. This results in under-allocation, as well as an OOB access at line 9. Although both cases result in OOB bugs, they stem from fundamentally different anti-patterns, underscoring the need for specialized modeling to detect each anti-pattern effectively.

Figure 1(c) shows a DBZ bug caused by insufficient zero check (IZC). The variable `u` depends on an external input and is used as the divisor at line 7. Although the code includes a conditional check at line 4 to ensure that `u` is non-negative, this check still allows `u` to be zero. Hence, using `u` directly in a division operation at line 7 without sufficient validation poses a divide-by-zero risk. Identifying this issue requires inferring the possible value range of `u` based on semantic constraints.

[1]A complete list of the detected bugs is publicly available on our bug gallary.

Figure 1(d) presents an example of system-specific anti-pattern in the Linux kernel. The data structure `i2c_data` is defined internally and contains an array named `block`, with its size stored in the first element (i.e., `data->block[0]`). At runtime, the `data` variable is user-controlled, meaning `data->block[0]` can be improperly set to zero or an excessively large value. Without proper validation, subsequent buffer manipulation or division operations may trigger OOB at lines 6–7 (when `data->block[0]` exceeds the size of `data->block`) and DBZ at line 8 (when `data->block[0]` is zero). Due to its relevance to system-specific semantics, this particular bug defies straightforward categorization into a single standard bug type. Consequently, effective detection requires auditors to possess deep understanding of the underlying system-specific behaviors.

To sum up, the above four examples underscore the considerable semantic diversity among software bugs, ranging from low-level bug categories (e.g., OOB and DBZ) to high-level anti-patterns such as oversized offset and allocation size overflow. Furthermore, the bugs introduced by generic anti-patterns represent merely the tip of the iceberg. Beneath the surface lie numerous system-specific anti-patterns that remain hidden and difficult to identify with general detection techniques. Given their tight coupling with system characteristics, such system-specific anti-patterns often pose even greater challenges for effective bug detection.

2.2 Limitations of Existing Techniques

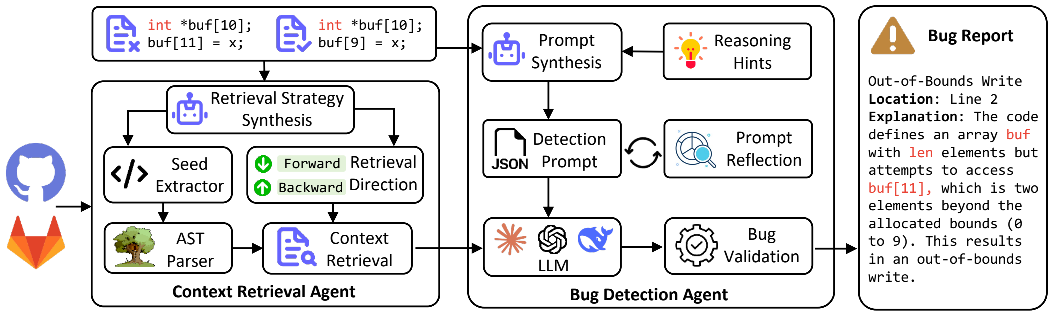
During the past several decades, researchers and practitioners have developed a variety of static analysis tools Sui & Xue (2016); Arzt et al. (2014); GitHub (2025); Meta (2025) that target specific, well-known anti-patterns for bug detection, such as those resulting from abnormal data flows. Based on handcrafted symbolic rules, the detectors can effectively detect bugs caused by common anti-patterns, while they typically lack the flexibility to generalize to novel or previously unseen anti-patterns. At one end of the spectrum, many commercial tools are closed and non-extensible, restricted to a fixed set of bug classes. For example, Meta Infer focuses primarily on concurrency bugs and memory safety violations Meta (2025). At the other end, tools like GitHub CodeQL offer extensibility, but with a steep learning curve. Crafting custom queries requires deep familiarity with its declarative language, which contains over 2,000 interfaces GitHub (2025), making it difficult for most security analysts to implement new detection logic Naik et al. (2021). Hence, the limited adaptability of these tools has become a significant practical barrier to real-world deployment.

Recent advances in LLMs have opened promising avenues for static bug detection. Particularly, reasoning models such as Claude 3.7 Sonnet Thinking and DeepSeek-R1 have showcased impressive capabilities in multi-step reasoning, significantly enhancing semantic code analysis. Over recent years, both academic researchers and industry practitioners have actively explored leveraging LLMs for bug detection tasks. One prominent approach utilizes the inherent knowledge of LLMs acquired during their pre-training phase to directly identify software bugs. Commercial tools exemplifying this method include Cursor BugBot Cursor (2025) and CodeRabbit CodeRabbit (2025). Nevertheless, due to the extensive variety of real-world bugs and anti-patterns (as previously discussed), LLMs frequently encounter difficulties recognizing uncommon or system-specific anti-patterns. Also, the lack of detection context causes several LLM-driven bug detection techniques to fail in identifying bugs across functions or files, thereby limiting their coverage of real-world bugs.

A complementary research direction integrates LLMs into traditional static analysis pipelines rather than relying on them in isolation Wang et al. (2024); Guo et al. (2025); Naik et al. (2021); Li et al. (2025a); Yang et al. (2025). For example, IRIS Li et al. (2025b) uses LLMs to infer taint specifications, reducing reliance on manually crafted rules. Similarly, KNighter Yang et al. (2025) synthesizes symbolic checkers from historical patches for customized detection. Although these approaches enhance traditional analyzers with LLM-supplied pre-knowledge, they remain limited by the inherent scope of static analyzers. Besides, recent efforts like LLMDFA Wang et al. (2024) and RepoAudit Guo et al. (2025) enable LLM-driven customization for data-flow bug detection, yet their pipeline designs are still confined to data-flow related anti-patterns, leaving many anti-patterns unaddressed, such as those in Figure 1.

2.3 Our Vision

Our vision is inspired by how human experts identify software bugs. Unlike traditional analyzers that rely on fixed rules, human auditors can adaptively detect previously unseen anti-patterns by



```
int *buf[10];
buf[11] = x;
```

```
int *buf[10];
buf[9] = x;
```

```
Out-of-Bounds Write
Location: Line 2
Explanation: The code
defines an array buf
with len elements but
attempts to access
buf[11], which is two
elements beyond the
allocated bounds (0
to 9). This results
in an out-of-bounds
write.
```

Figure 2: The overview of BUGSCOPE

learning from a few representative examples. Specifically, they generalize from past cases, retrieve relevant code context on demand, and apply holistic reasoning to assess whether a bug is present. This flexibility allows them to adapt their strategy to different anti-patterns, including those adhering to subtle or system-specific behaviors. Motivated by this insight, we introduce BUGSCOPE, an autonomous multi-agent that mimics the expert workflow of human auditors. Given a small set of labeled examples, BUGSCOPE learns the underlying anti-pattern, derives a retrieval strategy to extract relevant context, and synthesizes a detection prompt to guide its analysis. By harnessing both the learning and reasoning capabilities of LLMs, BUGSCOPE enables highly customizable bug detection across a wide spectrum of bugs. In the next section, we present the technical design and core components of BUGSCOPE.

3 APPROACH

This section presents BUGSCOPE, an autonomous LLM-based multi-agent that mimics how human auditors learn anti-patterns from examples and apply this knowledge to detect bugs.

3.1 OVERVIEW

When auditing code for potential bugs, human auditors typically begin by asking two fundamental questions: (1) ***Which code fragments are relevant to a potential bug's anti-pattern?*** (2) ***Is a bug truly present in a given code fragment?*** To answer the first question, auditors identify *faulty values*, which are values that may trigger bugs when propagated through execution (e.g., null values for NPD), as well as *dangerous operands*, which are operands that manifest bugs when processing such values (e.g., pointer dereferences). These elements serve as starting points for collecting relevant code along the program execution path, either forwardly or backwardly, to construct a context for bug detection. To answer the second question, auditors summarize the low-level semantics of the anti-patterns observed in the examples and verify the presence of bugs within the retrieved context.

We automate manual code auditing through BUGSCOPE, an LLM-powered multi-agent system. As illustrated in Figure 2, BUGSCOPE replicates the manual code auditing process by introducing two agents, namely the *context retrieval agent* and the *bug detection agent*. Given a set of buggy and non-buggy examples (of a specific anti-pattern), the context retrieval agent first synthesizes a retrieval strategy consisting of a seed extractor, which identifies program constructs as either faulty values or dangerous operands, and a retrieval direction (either forward from a faulty value or backward from a dangerous operand) to collect semantically relevant code, forming the candidate snippet for inspection. Simultaneously, the bug detection agent synthesizes a detection prompt that encodes the low-level semantics of the anti-pattern and uses it to assess whether the extracted snippet conforms to the targeted anti-pattern. In the following subsections, we will present more details.

3.2 CONTEXT RETRIEVAL AGENT

The goal of the context retrieval agent is to emulate how human auditors focus on relevant code fragments based on their understanding of the anti-pattern. According to our observations, this process consists of two phases: (1) Determining the retrieval strategy from the anti-pattern and (2) Performing slicing-based context retrieval by reasoning about program dependencies.

Retrieval Strategy Synthesis. In the first phase, the agent determines the retrieval strategy, including both the starting point for context retrieval and the appropriate retrieval direction. This is accomplished via the retrieval strategy synthesis, during which an extractor is synthesized to identify either faulty values or dangerous operands as the seeds. These seeds serve as slicing criteria for one of two retrieval directions:

- **Forward Slicing:** Collect the statements that are affected by the faulty value, which requires tracing program value propagation forwardly. When detecting the system-specific anti-pattern in Figure 1, for instance, we extract the parameter `data` as the seed and collect all the statements that depend on it in the forward manner.
- **Backward Slicing:** Collect the statements that influence the identified dangerous operand. For example, when detecting the anti-pattern of oversized offset shown in Figure 1(a), we can trace how the size of the buffer and the offset are computed by following program dependencies in reverse.

The result of this stage is a seed extractor that identifies the retrieval seeds with the corresponding retrieval direction. These seeds, including faulty values or dangerous operands, are then passed to the next stage for context retrieval.

Slicing-Based Context Retrieval. In the second stage, the context retrieval agent performs slicing-based context retrieval guided by the synthesized retrieval strategy. Starting from each seed, it reasons about program dependencies, including both data and control dependencies, to collect and aggregate relevant context along the specified retrieval direction.

During the retrieval, the slicing may involve inter-procedural analysis. For example, an allocated memory object might propagate beyond the current function, or a buffer offset might depend on the parameters of the current function. To address such cases, the context retrieval agent employs a parser (e.g., Tree-sitter Brunsfeld (2018)) to construct a call graph and trace inter-procedural dependencies along it. To balance precision and scalability, the agent limits the analysis to a fixed call depth, collecting slices that cross function boundaries at most K times.

To reduce hallucinations during bug detection, the retrieved slices are aggregated and inlined into a single, simplified function, which serves as the output of the context retrieval agent. This compact, self-contained snippet minimizes irrelevant code, thereby reducing LLM hallucinations and enhancing the reliability of detection results generated by the bug detection agent (Section 3.3).

3.3 Bug Detection Agent

The goal of the bug detection agent is to understand the detection logic according to the anti-pattern examples and then detect the anti-pattern upon the retrieved code snippet. Technically, the process begins by synthesizing a detection prompt from the examples. Based on the detection prompt, the LLM-driven detector subsequently analyzes code snippets retrieved by the context retrieval agent, determining whether each snippet contains the targeted anti-pattern. In what follows, we present the details of the two phases.

Detection Prompt Synthesis. The detection prompt synthesis ensures that BugScope generalizes effectively across diverse anti-patterns. Observations from manual code audits show that human auditors typically detect various anti-patterns by applying generic yet critical reasoning strategies related to different kinds of program values. Specifically, these reasoning strategies involve:

- **Primitive Values**: These include integers, floats, strings, and other fundamental data types supported by the programming language. Human auditors commonly reason about properties such as numeric ranges and ordering. Such semantic relationships are essential when primitive values are involved in bug detection contexts.
- **Pointer Values**: Pointers refer to memory locations of other program values. Auditors primarily reason about aliasing relationships involving pointers. For instance, if a dangerous operation involves a pointer p, auditors identify and track all its aliases, i.e., other pointers referencing the same memory as p, to ensure that all relevant memory operations are considered during the audit.
- **Buffer Values**: Buffers aggregate multiple program values of the same type within contiguous memory regions, combining characteristics of both primitive and pointer values. Human auditors

typically reason about the range of offsets (primitive values) and base pointer aliases (pointer values) when examining buffers within detection contexts.

Inspired by human auditing practices, we explicitly incorporate the above strategies as reasoning hints into the detection prompt synthesis process. Specifically, the examples integrated into the synthesized detection prompt facilitate few-shot chain-of-thought prompting. Additionally, the reasoning hints help LLMs align the reasoning process more closely with that of human auditors, enabling the LLMs to effectively capture critical program properties and further determine whether the detection context contains any instance of the targeted anti-pattern.

To mitigate LLM hallucinations during the detection prompt synthesis, we instantiate the mechanism of the prompt reflection mechanism. Specifically, we leverage LLMs to examine and refine the initially synthesized detection prompt based on the provided examples, which can improve the quality of the detection prompt and provide better guidance to the LLMs in the detection.

Detection with Validation. In the detection phase, BugScope uses the synthesized detection prompt to guide the LLM-based bug detection. When analyzing a code snippet obtained by the context retrieval agent, the detector assesses whether the logic matches the learned anti-pattern. If a bug is detected, the detector generates a structured bug candidate that includes the buggy program path and a natural-language explanation.

Given that bug detection often involves intricate control flows (e.g., branches, loops) and complex data flows, LLMs can sometimes produce false positives or unsupported conclusions. To address this, we introduce a bug-type-agnostic, LLM-based validator that independently verifies each bug candidate, filtering out erroneous detections and enhancing reliability. Ultimately, the bug detection agent produces validated bug reports, each containing a detailed description of the buggy program path and a clear, human-readable explanation generated by the LLM.

4 Evaluation

Implementation. We implement BugScope as an LLM-powered multi-agent for detecting bugs in C/C++ programs. Following prior work Wang et al. (2024); Guo et al. (2025), we adopt the `tree-sitter` parsing library Brunsfeld (2018) to support the context retrieval agent, enabling accurate and efficient syntactic analysis across large-scale codebases. In line with existing studies Shi et al. (2018); Heo et al. (2017), we bound the call stack depth to $K = 3$, focusing on bugs that stem from the faulty values and dangerous operands within call chains across function boundaries at most three times. For the underlying reasoning engine, we evaluate three state-of-the-art reasoning models, namely Claude 3.7 Sonnet Thinking, OpenAI o4-mini, and DeepSeek-R1. For brevity, we refer to Claude 3.7 Sonnet Thinking as Claude 3.7 in the reminder of the paper. We configure Claude 3.7 with an output token limit of 4,096 and a reasoning token limit of 2,048. OpenAI o4-mini does not support customization of token limits or temperature, so we adopt its default configuration. For DeepSeek-R1, which allows only the output token budget to be set, we configure it with a 4,096-token limit as well.

We present our evaluation details in four subsections. Section 4.1 provides details of our curated evaluation dataset. Section 4.2 evaluates the performance of BugScope on the dataset with different LLM engines. Section 4.3 benchmarks the effectiveness of BugScope against existing LLM-based detectors and commercial tools, respectively. Finally, Section 4.4 demonstrates the practical utility of BugScope by applying it to a range of real-world, large-scale open-source projects.

4.1 Dataset

We first survey recent works published in top venues in computer security and software engineering, and manually collect the corresponding bug reports released by the authors as a dataset Guo et al. (2022; 2024); Huang et al. (2024); Guo et al. (2025); Shi et al. (2021; 2018). As shown in Table 2, we focus on three categories of bugs, namely Out-of-Bounds (OOB), Divide-by-Zero (DBZ), and Memory Leak (MLK). The three bug types are representative as they cover a wide spectrum of program properties, including pointer-related properties, numeric properties, and both. To support the customization, we target each anti-pattern of the selected buggy cases. Concretely, we obtain

Table 1: The details of the evaluation dataset

Project	Bug Type	Anti-pattern	Commit	Target File	Report
zstd	OOB	OSO	e5db7c9	programs/util.c	link
systemd	OOB	OSO	fa2ba7a	src/basic/time-util.c	link
frr	OOB	OSO	26b2fbf	zebra/kernel_netlink.c	link
redis	OOB	OSO	93dda65	src/t_zset.c	link
qemu	OOB	OSO	232e925	contrib/elf2dmp/qemu_elf.c	link
curl	OOB	NOF	e0c68f02	lib/sendf.c	link
curl	OOB	NOF	e0c68f02	lib/sendf.c	link
zstd	OOB	NOF	e5db7c9	programs/util.c	link
php-src	OOB	NOF	492f9c6	ext/opcache/zend_accelerator_blacklist.c	link
openssl	OOB	NOF	2837b19	crypto/bf/bf_ofb64.c	link
php-src	OOB	ASO	492f9c6	sapi/cli/php_cli_server.c	link
systemd	OOB	ASO	fa2ba7a	src/libsystemd-network/disc-router.c	link
frr	OOB	ASO	26b2fbf	bfdd/control.c	link
binutils-gdb	OOB	ASO	4bb461e	ld/libdep_plugin.c	link
gcc	OOB	ASO	9715f10	libcpp/files.cc	link
git	DBZ	IZC	49ac1d3	git/builtin/pack-objects.c	link
linux	DBZ	IZC	9e9b451	block/blk-mq-cpumap.c	link
binutils-gdb	DBZ	IZC	2005aa02	gdb/amd64-tdep.c	link
openssl	DBZ	IZC	bc8c3627	crypto/pkcs12/p12_key.c	link
vim	DBZ	IZC	ccfb7c67	vim/src/misc2.c	link
systemd	DBZ	IZC	f6e40037	src/shared/creds-util.c	link
ImageMagick	DBZ	IZC	442c87b9	MagickCore/cache.c	link
ImageMagick	DBZ	IZC	442c87b9	MagickCore/cache.c	link
libuv	DBZ	IZC	af1a79cf	src/unix/linux-core.c	link
linux	DBZ	LZD	9e9b451	drivers/video/logo/pnmtologo.c	link
linux	DBZ	LZD	9e9b451	lib/math/rational.c	link
linux	DBZ	LZD	9e9b451	drivers/char/agp/isoch.c	link
goaccess	DBZ	LZD	0abddd5f	src/gholder.c	link
goaccess	DBZ	LZD	0abddd5f	src/gholder.c	link
libsass	MLK	UEC	4da7c4b	src/permutate.hpp	link
memcached	MLK	UEC	e15e1d6	memcached.c	link
memcached	MLK	UEC	6fb5ef7	restart.c	link
h3	MLK	UEC	3a02395	src/apps/filters/h3.c	link
TrinityEmulator	MLK	UEC	ad25460	contrib/elf2dmp/main.c	link
linux	MLK	MSC	4cd8371	drivers/net/nfpcore/nfp_cppcore.c	link
linux	MLK	MSC	73b73bac	mm/damon/reclaim.c	link
rtl_433	MLK	MSC	474feb5	rtl_433/src/sdr.c	link
libuv	MLK	MSC	98a4bab	docs/code/plugin/main.c	link
TrinityEmulator	MLK	MSC	ad25460	contrib/elf2dmp/main.c	link
binutils-gdb	MLK	MSC	0ebc886	binutils/bucomm.c	link

seven distinct anti-patterns in total. For OOB bugs, we include the following patterns: (1) oversized offset (OSO), where the offset exceeds the buffer length; (2) negative offset (NOF), where the offset used in the buffer read or write operation is negative; and (3) allocation size overflow (ASO), where an integer overflow during memory allocation leads to a smaller-than-expected buffer. For DBZ bugs, we include (1) insufficient zero check (IZC), where a conditional check on the unconstrained divisor (e.g., a user-specified input) is insufficient to rule out zero; and (2) literal zero division (LZD), where a literal zero value is used as a divisor without validation. For MLK bugs, we cover (1) unexecuted cleanup (UEC), where the cleanup code exists but is not executed due to early returns or exceptions; and (2) missing cleanup (MSC), where the program does not include any code that frees allocated memory.

Table 2: The overall statistics of the dataset. **#BE/#NE** denote the numbers of buggy/non-buggy examples.

Bug	Anti-Pattern	Cases	#BE	#NE
OOB	OSO	5	2	3
	NOF	5	1	1
	ASO	5	1	1
DBZ	IZC	9	1	2
	LZD	5	1	1
MLK	UEC	5	1	2
	MSC	6	1	1

In total, the dataset consists of 40 cases collected from 21 open-source projects, which have 29K GitHub stars on average. Each case includes the corresponding historical bug-inducing commit and precise bug location, providing a realistic basis for evaluating detection capability. In our evaluation,

Table 3: The statistics of BugScope upon real-world benchmark programs. In the column **Model**, ✳ represents Claude 3.7 Sonnet Thinking, 🐋 represents DeepSeek-R1, and ֍ represents OpenAI o4-mini. **AP** indicates **Anti-Pattern**. **#C** denotes the number of cases for each anti-pattern. **#Seed** denotes the number of seeds extracted from a specific file. **#R** denotes the number of reproduced bugs. **#N** denotes the number of new bugs found. **P(%)**, **R(%)**, and **F1** denote precision, recall, and F1 score, respectively. **Time** and **Cost** indicates the total time and financial cost of analyzing all the seeds when analyzing a specific anti-pattern, respectively.

Model	AP	#C	#Seed	#R	#N	#TP	#FP	P(%)	R(%)	F1	Time (s)	Cost ($)
✳	OSO	5	293	4	2	6	3	66.67	80.00	0.73	6,318	36.42
	NOF	5	276	5	2	7	2	77.78	100.00	0.88	4,818	27.45
	ASO	5	30	5	0	5	0	100.00	100.00	1.00	1,920	4.99
	IZC	9	33	6	1	7	1	87.50	66.67	0.76	2,254	4.87
	LZD	5	17	5	5	10	0	100.00	100.00	1.00	829	1.40
	UEC	5	36	5	1	6	1	85.71	100.00	0.92	1,895	6.11
	MSC	6	18	6	0	6	0	100.00	100.00	1.00	2,120	2.76
	Total	40	703	36	11	47	7	**87.04**	**90.00**	**0.88**	20,154	84.00
🐋	OSO	5	293	3	1	4	2	66.67	60.00	0.63	12,642	6.90
	NOF	5	276	5	1	6	2	75.00	100.00	0.86	3,420	2.76
	ASO	5	30	4	0	4	0	100.00	80.00	0.89	1,458	0.87
	IZC	9	33	5	1	6	0	100.00	55.56	0.71	2,115	0.68
	LZD	5	17	5	3	8	1	88.89	100.00	0.94	951	0.35
	UEC	5	36	5	1	6	1	85.71	100.00	0.92	1,330	0.87
	MSC	6	18	5	0	5	0	100.00	83.33	0.91	2,476	0.98
	Total	40	703	32	7	39	6	**86.67**	**80.00**	**0.83**	24,392	13.40
֍	OSO	5	293	3	0	3	2	60.00	60.00	0.60	9,976	9.04
	NOF	5	276	5	1	6	2	75.00	100.00	0.86	2,598	4.87
	ASO	5	30	5	0	5	0	100.00	100.00	1.00	909	1.11
	IZC	9	33	5	0	5	0	100.00	55.56	0.71	1,538	2.06
	LZD	5	17	5	2	7	1	87.50	100.00	0.93	455	0.55
	UEC	5	36	5	1	6	2	75.00	100.00	0.86	822	1.05
	MSC	6	18	5	1	6	0	100.00	83.33	0.91	827	0.77
	Total	40	703	33	5	38	7	**84.44**	**82.50**	**0.83**	17,125	19.45

we target the historical versions of the projects immediately preceding the bug-fixing commits. More details of the dataset are provided in Table 1.

4.2 Performance of BugScope

Setup and Metrics. We evaluate BugScope on the dataset to assess its effectiveness. For each anti-pattern, we configure BugScope with a small set of buggy and non-buggy examples, as shown in Table 2. We then task BugScope with automatically synthesizing a context retrieval strategy and a detection prompt using the LLMs, which are subsequently used to detect the targeted anti-pattern. Given the large scale of modern codebases, analyzing all functions is computationally prohibitive. Therefore, we constrain seed extraction to the same file as the original bug report, which ensures relevant context while reducing overhead. Notably, such experimental setting mirrors a realistic usage scenario, particularly in commit-level code reviews, where many very recent commercial AI coding auditing tools such as Cursor BugBot Cursor (2025) and CodeRabbit CodeRabbit (2025) are commonly applied. When analyzing the seeds in a given source file, we measure the precision, recall, and F1 score of the bug detection. To quantify the computation resource consumption of BugScope, we measure its time cost and financial cost according to the pricing policy of the reasoning model APIs.

Result. Table 3 presents the detailed statistics of BugScope upon our dataset. According to Table 3, BugScope powered by Claude 3.7 achieves the best performance. Specifically, it successfully reproduces 36 out of 40 bugs with a recall of 90.00%. It also discovers 11 new bugs in historical code versions, 10 of which have already been fixed in the latest commits. In total, BugScope reports 47 true positives and 7 false positives, resulting in a precision of 87.04% and a F1 score of 0.88

with Claude 3.7. When powered by other reasoning models, BUGSCOPE also demonstrates strong and consistent performance. Specifically, it detects 39 and 38 true positives with DeepSeek-R1 and OpenAI o4-mini, respectively, achieving the F1 score of 0.83 in both cases. Compared to Claude 3.7, the recall of the other two LLMs drops by 10% and 7.5%, particularly on two anti-patterns, namely oversized offset (OSO) and insufficient zero check (IZC), which demand more sophisticated semantic reasoning. This performance gap can be attributed to the stronger reasoning capability of Claude 3.7 in code reasoning tasks, as evidenced by its top performance on the SWE-Bench benchmark among the three models Jimenez et al. (2024).

In terms of cost, BUGSCOPE requires an average of 34 seconds and \$0.02 to synthesize the retrieval strategy, and 73 seconds and \$0.04 to synthesize the detection prompt for each anti-pattern using Claude 3.7. In the detection phase, Claude 3.7 incurs the highest expense per seed, which is \$0.12 (= 84/703), whereas Deepseek-R1 and o4-mini achieve comparable precision and recall (both over 84% precision and over 80% recall in total) with significantly lower cost (\$0.02 and \$0.03 per seed, respectively), demonstrating the cost-effectiveness of BUGSCOPE across different LLMs.

4.3 COMPARISON WITH BASELINES

Setup and Metrics. We compare BUGSCOPE against the state-of-the-art LLM-driven bug detection tool RepoAudit Guo et al. (2025), as well as three commercial tools, namely Cursor BugBot Cursor (2025), CodeRabbit CodeRabbit (2025), and Meta Infer Meta (2025). RepoAudit is an LLM agent that detects data-flow bugs in the repository. BugBot and CodeRabbit are commercial agents that can be integrated within GitHub repositories for automatically scanning pull requests. Infer is a production-grade static analyzer that supports a wide range of built-in checkers for different bug types. We evaluate all tools upon the dataset introduced in Section 4.1.

We use Claude 3.7, which is the most effective model according to Section 4.2, to power RepoAudit. For fairness, we extend RepoAudit with source and sink extractors for DBZ and OOB and limit their scope to the target file containing the original bug. For BugBot and CodeRabbit, we simulate a GitHub pull request containing the target file and collect reported issues that match the relevant bug type. Since Infer is a compiler-based static analysis tool, we run it from the project root and only consider the reports relevant to the target bug type within the corresponding file. We measure the precision, recall, and F1 score of the baselines and track the number of newly discovered bugs.

Result. As shown in Table 4, BUGSCOPE significantly outperforms all baselines across all three bug categories. The precision and recall of RepoAudit are only 32.14% and 42.50%, which are much lower than the ones of BUGSCOPE. Although BugBot and CodeRabbit attain comparable precision, their overall effectiveness is limited by poor recall, resulting in F1 scores of only 0.40 and 0.29, respectively. RepoAudit demonstrates strong recall on MLK bugs but suffers from low precision on DBZ and OOB categories, yielding an overall F1 score of 0.37. Infer performs the worst, detecting only a single OOB bug with an F1 score of 0.04. Upon examining the results of baselines, we identify several key drawbacks in their analysis capabilities. RepoAudit adopt a source–sink analysis paradigm primarily tailored for data-flow vulnerabilities. Although it can support the customization of sources and sinks in the data-flow reachability analysis, many bug instances, such as the ones in the categories of DBZ and OOB, can not be formulated as a data-flow reachability problem, which requires reasoning about numerical relationships and multiple domains of program properties. As a result, although RepoAudit achieves strong F1 scores of 0.92 and 0.91 on two anti-patterns, namely unexecuted cleanup (UEC) and missing cleanup (MSC), respectively, its performance drops significantly on others, such as negative offset and insufficient zero check.

BugBot and CodeRabbit struggle with complex anti-patterns. While these tools can leverage the prior knowledge of the LLMs to identify common anti-patterns, they fail to reason about less common or more intricate ones, such as negative offset (NOF), allocation size overflow (ASO), and insufficient zero check (IZC). Besides, the two tools also suffer from the lack of necessary contexts. BugBot, for example, supports limited inter-procedural reasoning within a single file but can not trace bug propagation paths across multiple files upon our subjects, leading to the low quality of bug reports and especially introducing numerous false negatives. CodeRabbit encounters similar challenges, with even weaker context retrieval capabilities. For example, among the only two true positives of literal zero division (LZD) detected by CodeRabbit, the provided explanations are overly generic, merely stating that "if the divisor is zero, a divide-by-zero error may occur" without

Table 4: The comparison results between BUGSCOPE and baselines. **AP** indicates **Anti-Pattern**. **#C** denotes the number of cases for each anti-pattern. **#R** denotes the number of reproduced bugs. **#N** denotes the number of new bugs found. **#TP** and **#FP** are the number of true and false positives, respectively. **P(%)**, **R(%)**, and **F1** denote precision, recall, and F1 score, respectively.

Tool Name	**AP**	**#C**	**#R**	**#N**	**#TP**	**#FP**	**P(%)**	**R(%)**	**F1**
BugScope	OSO	5	4	2	6	3	66.67	80.00	0.73
	NOF	5	5	2	7	2	77.78	100.00	0.88
	ASO	5	5	0	5	0	100.00	100.00	1.00
	IZC	9	6	1	7	1	87.50	66.67	0.76
	LZD	5	5	5	10	0	100.00	100.00	1.00
	UEC	5	5	1	6	1	85.71	100.00	0.92
	MSC	6	6	0	6	0	100.00	100.00	1.00
	Total	40	36	11	47	7	**87.04**	**90.00**	**0.88**
RepoAudit	OSO	5	1	0	1	4	20.00	20.00	0.20
	NOF	5	0	0	0	9	0.00	0.00	0.00
	ASO	5	2	0	2	2	50.00	40.00	0.44
	IZC	9	1	0	1	15	6.25	11.11	0.08
	LZD	5	3	0	3	7	30.00	60.00	0.40
	UEC	5	5	1	6	1	85.71	100.00	0.92
	MSC	6	5	0	5	0	100.00	83.33	0.91
	Total	40	17	1	18	38	**32.14**	**42.50**	**0.37**
BugBot	OSO	5	1	1	2	1	66.67	20.00	0.31
	NOF	5	1	0	1	2	33.33	20.00	0.25
	ASO	5	0	1	1	1	50.00	0.00	0.00
	IZC	9	1	0	1	0	100.00	11.11	0.20
	LZD	5	3	1	4	0	100.00	60.00	0.75
	UEC	5	1	0	1	2	33.33	20.00	0.25
	MSC	6	4	1	5	0	100.00	66.67	0.80
	Total	40	11	4	15	6	**71.43**	**27.50**	**0.40**
CodeRabbit	OSO	5	0	0	0	3	0.00	0.00	0.00
	NOF	5	0	0	0	0	0.00	0.00	0.00
	ASO	5	1	0	1	0	100.00	20.00	0.33
	IZC	9	0	0	0	0	0.00	0.00	0.00
	LZD	5	1	1	2	0	100.00	20.00	0.33
	UEC	5	1	1	2	0	100.00	20.00	0.33
	MSC	6	4	1	5	0	100.00	66.67	0.80
	Total	40	7	3	10	3	**76.92**	**17.50**	**0.29**
Infer	OSO	5	0	0	0	11	0.00	0.00	0.00
	NOF	5	1	0	1	1	50.00	20.00	0.29
	ASO	5	0	0	0	0	0.00	0.00	0.00
	IZC	9	0	0	0	0	0.00	0.00	0.00
	LZD	5	0	0	0	0	0.00	0.00	0.00
	UEC	5	0	0	0	0	0.00	0.00	0.00
	MSC	6	0	0	0	0	0.00	0.00	0.00
	Total	40	1	0	1	12	**7.69**	**2.50**	**0.04**

providing any justification of why the divisor could be zero. In contrast, BUGSCOPE successfully identifies the complete bug propagation path across multiple functions, delivering clear and precise explanations. Several detailed case studies of BugBot and CodeRabbit are provided in Appendix A.

As noted by prior work Guo et al. (2025), Infer relies heavily on build interception and supports only a fixed set of built-in checkers. We were able to evaluate only 22 out of 40 cases; the rest failed due to incompatibilities with Infer's build process. Specifically, Infer uses `infer capture` to intercept builds, assuming Clang-style commands and standard toolchains. Projects using GCC-specific flags, custom Makefiles, or non-standard systems (e.g., LINUX) often fail during capture, resulting in build or analysis failures. Even among the successfully compiled projects, Infer exhibits poor performance, detecting only one true positive and producing 12 false positives in the OOB

Table 5: The statistics of BUGSCOPE upon six real-world projects. **TP** and **FP** indicates the numbers of true positives and false positives, respectively. **Co** and **Fx** denote the numbers of bugs that are confirmed and fixed, respectively. **P(%)** denotes precision.

Project	**OOB**					**DBZ**					**MLK**				
	TP	**FP**	**Co**	**Fx**	**P (%)**	**TP**	**FP**	**Co**	**Fx**	**P (%)**	**TP**	**FP**	**Co**	**Fx**	**P (%)**
vim	4	2	0	4	66.67	0	1	0	0	0.00	2	1	0	2	66.67
systemd	0	0	0	0	0.00	2	1	0	2	66.67	0	2	0	0	0.00
dynamips	2	0	0	2	100.00	0	0	0	0	0.00	21	2	0	19	91.30
zstd	1	1	0	0	50.00	4	1	0	1	80.00	18	4	0	18	81.82
openldap	0	2	0	0	0.00	0	1	0	0	0.00	18	5	0	17	78.26
git	1	0	1	0	100.00	2	0	2	0	100.00	5	1	2	3	83.33
Total	8	5	1	6	61.54	8	4	2	3	66.67	64	15	2	59	81.01

category. Crucially, if no internal checker exists for a given bug type, Infer cannot detect it at all. For example, upon our investigation, DBZ bugs are entirely unsupported, resulting in zero relevant reports. Furthermore, even for broader categories like OOB, where relevant checkers exist, its effectiveness is constrained by an inability to differentiate between distinct anti-patterns (e.g., oversized offset (OSO) vs. negative offset (NOF)).

In contrast, BUGSCOPE replicates the human process of learning anti-patterns. Benefiting from the generalization ability of LLM-based reasoning and the synthesized detection prompt, BUGSCOPE can be adapt to diverse bug types and anti-patterns. Our technical designs enables BUGSCOPE to consistently deliver high precision and recall along with informative bug explanations, even in cases where traditional or commercial tools fall short.

4.4 REAL-WORLD IMPACT

Setup and Metrics. To evaluate the effectiveness of BUGSCOPE in real-world scenarios, we conduct two additional experiments. In the first experiment, we apply the previously synthesized seed extractors and detection prompts to six open-source GitHub projects. These projects span diverse application domains, with codebase sizes ranging from 106K to over 1 million lines of code (LoC). On average, the selected repositories contain 654K LoC and have accumulated 22.3K GitHub stars, highlighting both their scale and high profile. To manage computational overhead, we set the upper bound on the call depth K to 2 and cap the number of seed statements at 100 per repository. If the number of the extracted seeds exceeds this threshold, we randomly sample 100 for further analysis.

In the second experiment, we move beyond well-defined bug types and evaluate the ability of BUGSCOPE to detect system-specific anti-patterns derived from real-world patches. Specifically, we select three Linux patches from prior work Chen et al. (2025), each of which represents a specific anti-pattern in Linux kernel. We then use BUGSCOPE to generate the retrieval strategy and detection prompt, facilitating the detection of similar bugs.

Result. Table 5 summarizes the results on six real-world projects. BUGSCOPE detects 8 divide-by-zero bugs, 8 out-of-bounds bugs, and 64 memory leak bugs, achieving the overall precision of 76.92%. Compared to other bug types, BUGSCOPE achieves the best performance in detecting memory leaks, with a precision of 81.01%. In contrast, the precision for detecting divide-by-zero and out-of-bounds bugs is lower, at 66.67% and 61.54%, respectively. This is primarily because we adopt a backward analysis approach for these bug types, where the analysis proceeds from the bug trigger point backward along the control flow. Due to the limited call depth and the presence of external inputs (e.g., user input), the analysis may fail to capture sufficient pre-context. Meanwhile, since we employ general detection prompts to cover a wider variety of anti-patterns, the model is allowed to infer missing pre-context during detection, which can lead to false positives. These false positives can be mitigated by customizing detection prompts with more restrictive bug patterns to enforce stricter matching criteria.

In the second experiment, BUGSCOPE extracts a distinct anti-pattern from each patch, which contains one buggy example and one non-buggy example. Eventually, BUGSCOPE detects 39 null pointer dereference (NPD) bugs, 15 DBZ bugs, and 7 OOB bugs, achieving an average precision of 91.04%. Overall, BUGSCOPE detects 141 previously unknown bugs with a precision of 82.46%.

Notably, 78 of these bugs have already been fixed, and 7 have been confirmed by developers. Among the 61 true bugs discovered in the Linux kernel, ten have been fixed and two have been confirmed by Linux developers. The above statistics on real-world bug detection highlight the significant impact of BUGSCOPE in the open-source community, demonstrating its effectiveness in precisely identifying real-world software bugs.

5 RELATED WORK

LLM-driven Bug Detection. Recent studies have adopted two complementary paradigms for leveraging LLMs in bug detection: LLM-augmented program analysis and LLM-based autonomous agents. In the former, LLMs enhance traditional static analyzers by generating analysis queries or domain specifications. For example, the neuro-symbolic framework IRIS Li et al. (2025b) exploits the contextual reasoning of the LLMs to infer taint specifications that integrate seamlessly into existing analysis pipelines, while LLift Li et al. (2024) couples an LLM with a static analyzer to detect use-before-initialization bugs in OS code. In the latter paradigm, LLMs serve as the primary detection engine: RepoAudit Guo et al. (2025) uses iterative prompting, long-term memory, and validation loops to autonomously traverse repositories and uncover diverse vulnerabilities; LLMDFA Wang et al. (2024) decomposes data-flow analysis into LLM-generated subtasks for AST extraction and theorem-prover integration; and RFCScan Zheng et al. (2025) cross-references implementations against RFC specifications to identify functional protocol bugs. While these approaches have demonstrated strong performance, they often focus on narrow bug classes or require substantial domain expertise (e.g., writing CodeQL rules or formal specs). In contrast, our work explore the paradigm of *learn-from-example*: By providing buggy and non-buggy code samples, we enable the LLM to understand the underlying anti-pattern, achieving broad vulnerability coverage without complex domain knowledge or specialized analysis infrastructure.

Customizable Static Analysis. As highlighted by prior work Johnson et al. (2013), customization support has long been an overlooked aspect of static analysis. Mainstream platforms, such as FlowDroid Arzt et al. (2014), SVF Sui & Xue (2016), Meta Infer Calcagno et al. (2009), and KLEE Cadar et al. (2008), are typically tailored to specific categories of bugs. Adapting these tools to detect other types of vulnerabilities often requires substantial modifications to their underlying reasoning engines. GitHub CodeQL GitHub (2025) enables the customization by writing specific queries, but mastering the CodeQL language presents a steep learning curve. To address this limitation, recent studies have explored automatic synthesis of bug checkers. For instance, MoCQ Li et al. (2025a) and KNighter Yang et al. (2025) utilize predefined checker templates upon existing static analysis platforms to synthesize checkers from examples. Sporq Naik et al. (2021) further enhances checker quality through iterative user interaction, allowing additional examples to clarify user intent. In contrast to these approaches, our work does not depend on any symbolic reasoning-based static analysis infrastructure. Instead, we leverage the generalization capabilities of LLMs, enabling them to reason in a human-like manner by following synthesized detection prompts. This yields significantly more flexible customization for bug detection. We believe our novel analysis paradigm offers a brand-new perspective on customizable static analysis and can inspire broader applications at the intersection of generative AI and software engineering.

6 CONCLUSION

This paper presents BUGSCOPE, an LLM-based multi-agent that emulates the human auditing process for learning and detecting bugs from examples. Given user-specified code examples, BUGSCOPE identifies relevant contexts through LLM-driven program slicing and synthesizes a detection prompt that encode the corresponding detection logic. Our empirical evaluation on real-world benchmark programs demonstrates that BUGSCOPE achieves 87.04% precision and 90.00% recall across multiple bug categories, including memory leaks, divide-by-zero, and out-of-bounds errors. Across real-world open-source projects, BUGSCOPE successfully uncovers 141 true bugs, 78 of which have been fixed and 7 confirmed by developers, highlighting its substantial practical impact on the security and reliability of real-world software systems.

REFERENCES

Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick D. McDaniel. Flowdroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In Michael F. P. O'Boyle and Keshav Pingali (eds.), *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, pp. 259–269. ACM, 2014. doi: 10.1145/2594291.2594299.

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

Max Brunsfeld. Tree-sitter-a new parsing system for programming tools. In *Strange Loop Conference,. Accessed–. URL: https://www. thestrangeloop. com//tree-sitter—a-new-parsing-system-for-programming-tools. html*, 2018.

Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In Richard Draves and Robbert van Renesse (eds.), *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA, Proceedings*, pp. 209–224. USENIX Association, 2008.

Cristiano Calcagno, Dino Distefano, Peter W. O'Hearn, and Hongseok Yang. Compositional shape analysis by means of bi-abduction. In Zhong Shao and Benjamin C. Pierce (eds.), *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009, Savannah, GA, USA, January 21-23, 2009*, pp. 289–300. ACM, 2009. doi: 10.1145/1480881.1480917.

Wei Chen, Bowen Zhang, Chengpeng Wang, Wensheng Tang, and Charles Zhang. Seal: Towards diverse specification inference for linux interfaces from security patches. In *Proceedings of the Twentieth European Conference on Computer Systems*, pp. 1246–1262, 2025.

CodeRabbit. Coderabbit: Ai-powered code review for github. `https://www.coderabbit.ai/`, 2025. Accessed: 2025-07-07.

Cursor. Bugbot documentation. `https://docs.cursor.com/bugbot`, 2025. Accessed: 2025-06-26.

Qingxiu Dong, Lei Li, Damai Dai, Ce Zheng, Jingyuan Ma, Rui Li, Heming Xia, Jingjing Xu, Zhiyong Wu, Tianyu Liu, et al. A survey on in-context learning. *arXiv preprint arXiv:2301.00234*, 2022.

GitHub. CodeQL for Java. `https://codeql.github.com/`, 2025. [Online; accessed 13-July-2025].

GitHub. Codeql documentation. `https://codeql.github.com/docs/`, 2025. Accessed: 2025-07-10.

Google. Oss-fuzz: Continuous fuzzing for open source software. `https://google.github.io/oss-fuzz/`, 2025. Accessed: 2025-07-10.

Jinyao Guo, Chengpeng Wang, Xiangzhe Xu, Zian Su, and Xiangyu Zhang. Repoaudit: An autonomous llm-agent for repository-level code auditing, 2025. URL `https://arxiv.org/abs/2501.18160`.

Yiyuan Guo, Jinguo Zhou, Peisen Yao, Qingkai Shi, and Charles Zhang. Precise divide-by-zero detection with affirmative evidence. In *Proceedings of the 44th International Conference on Software Engineering*, pp. 1718–1729, 2022.

Yiyuan Guo, Peisen Yao, and Charles Zhang. Precise compositional buffer overflow detection via heap disjointness. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pp. 63–75, 2024.

Kihong Heo, Hakjoo Oh, and Kwangkeun Yi. Machine-learning-guided selectively unsound static analysis. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pp. 519–529. IEEE, 2017.

Jianjun Huang, Jianglei Nie, Yuanjun Gong, Wei You, Bin Liang, and Pan Bian. Raisin: Identifying rare sensitive functions for bug detection. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, pp. 1–12, 2024.

Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R Narasimhan. SWE-bench: Can language models resolve real-world github issues? In *The Twelfth International Conference on Learning Representations*, 2024. URL `https://openreview.net/forum?id=VTF8yNQM66`.

Brittany Johnson, Yoonki Song, Emerson Murphy-Hill, and Robert Bowdidge. Why don't software developers use static analysis tools to find bugs? In *2013 35th International Conference on Software Engineering (ICSE)*, pp. 672–681. IEEE, 2013.

Haonan Li, Yu Hao, Yizhuo Zhai, and Zhiyun Qian. Enhancing static analysis for practical bug detection: An llm-integrated approach. *Proceedings of the ACM on Programming Languages*, 8 (OOPSLA1):474–499, 2024.

Penghui Li, Songchen Yao, Josef Sarfati Korich, Changhua Luo, Jianjia Yu, Yinzhi Cao, and Junfeng Yang. Automated static vulnerability detection via a holistic neuro-symbolic approach. *CoRR*, abs/2504.16057, 2025a. doi: 10.48550/ARXIV.2504.16057. URL `https://doi.org/10.48550/arXiv.2504.16057`.

Ziyang Li, Saikat Dutta, and Mayur Naik. Iris: Llm-assisted static analysis for detecting security vulnerabilities. In *The Thirteenth International Conference on Learning Representations*, 2025b.

Meta. Infer Static Analyzer. `https://fbinfer.com/`, 2025. [Online; accessed 13-July-2025].

MITRE. Common Weakness Enumeration (CWE). `https://cwe.mitre.org/index.html`, 2025a. URL `https://cwe.mitre.org/index.html`. Accessed: 2025-07-11.

MITRE. Cve - common vulnerabilities and exposures. `https://www.cve.org/`, 2025b. Accessed: 2025-07-10.

Aaditya Naik, Jonathan Mendelson, Nathaniel Sands, Yuepeng Wang, Mayur Naik, and Mukund Raghothaman. Sporq: An interactive environment for exploring code using query-by-example. In *The 34th Annual ACM Symposium on User Interface Software and Technology*, pp. 84–99, 2021.

Thomas Reps. Undecidability of context-sensitive data-dependence analysis. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 22(1):162–186, 2000.

Qingkai Shi, Xiao Xiao, Rongxin Wu, Jinguo Zhou, Gang Fan, and Charles Zhang. Pinpoint: Fast and precise sparse value flow analysis for million lines of code. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 693–706, 2018.

Qingkai Shi, Peisen Yao, Rongxin Wu, and Charles Zhang. Path-sensitive sparse analysis without path conditions. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pp. 930–943, 2021.

Yulei Sui and Jingling Xue. SVF: interprocedural static value-flow analysis in LLVM. In Ayal Zaks and Manuel V. Hermenegildo (eds.), *Proceedings of the 25th International Conference on Compiler Construction, CC 2016, Barcelona, Spain, March 12-18, 2016*, pp. 265–266. ACM, 2016. doi: 10.1145/2892208.2892235.

Chengpeng Wang, Wuqi Zhang, Zian Su, Xiangzhe Xu, Xiaoheng Xie, and Xiangyu Zhang. LLMDFA: Analyzing dataflow in code with large language models. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*, 2024.

Chenyuan Yang, Zijie Zhao, Zichen Xie, Haoyu Li, and Lingming Zhang. Knighter: Transforming static analysis with llm-synthesized checkers. *CoRR*, abs/2503.09002, 2025. doi: 10.48550/ARXIV.2503.09002. URL `https://doi.org/10.48550/arXiv.2503.09002`.

Mingwei Zheng, Chengpeng Wang, Xuwei Liu, Jinyao Guo, Shiwei Feng, and Xiangyu Zhang. An llm agent for functional bug detection in network protocols, 2025. URL `https://arxiv.org/abs/2506.00714`.

Table 6: The detailed evaluation results of Cursor Bugbot and CodeRabbit. **BT** indicates **Bug Type**. **AP** indicates **Anti-Pattern**. **R** denotes the number of reproduced bugs. **N** denotes the number of new bugs found. **TP** and **FP** indicate the numbers of true positives and false positives, respectively.

Project	**BT**	**AP**	**Cursor Bugbot**					**CodeRabbit**				
			R	**N**	**TP**	**FP**	**Report**	**R**	**N**	**TP**	**FP**	**Report**
zstd	OOB	OSO	0	1	1	0	link	0	0	0	1	link
systemd	OOB	OSO	0	0	0	0	link	0	0	0	1	link
frr	OOB	OSO	0	0	0	0	link	0	0	0	0	link
redis	OOB	OSO	1	0	1	0	link	0	0	0	0	link
qemu	OOB	OSO	0	0	0	1	link	0	0	0	1	link
curl	OOB	NOF	0	0	0	0	link	0	0	0	0	link
curl	OOB	NOF	0	0	0	0	link	0	0	0	0	link
zstd	OOB	NOF	0	0	0	0	link	0	0	0	0	link
php-src	OOB	NOF	1	0	1	2	link	0	0	0	0	link
openssl	OOB	NOF	0	0	0	0	link	0	0	0	0	link
php-src	OOB	ASO	0	1	1	0	link	0	0	0	0	link
systemd	OOB	ASO	0	0	0	0	link	0	0	0	0	link
frr	OOB	ASO	0	0	0	0	link	1	0	1	0	link
binutils-gdb	OOB	ASO	0	0	0	0	link	0	0	0	0	link
gcc	OOB	ASO	0	0	0	1	link	0	0	0	0	link
git	DBZ	IZC	0	0	0	0	link	0	0	0	0	link
linux	DBZ	IZC	0	0	0	0	link	0	0	0	0	link
binutils-gdb	DBZ	IZC	0	0	0	0	link	0	0	0	0	link
openssl	DBZ	IZC	0	0	0	0	link	0	0	0	0	link
vim	DBZ	IZC	0	0	0	0	link	0	0	0	0	link
systemd	DBZ	IZC	1	0	1	0	link	0	0	0	0	link
ImageMagick	DBZ	IZC	0	0	0	0	link	0	0	0	0	link
ImageMagick	DBZ	IZC	0	0	0	0	link	0	0	0	0	link
libuv	DBZ	IZC	0	0	0	0	link	0	0	0	0	link
linux	DBZ	LZD	0	0	0	0	link	0	0	0	0	link
linux	DBZ	LZD	1	0	1	0	link	0	0	0	0	link
linux	DBZ	LZD	0	0	0	0	link	0	0	0	0	link
goaccess	DBZ	LZD	1	1	2	0	link	1	1	2	0	link
goaccess	DBZ	LZD	1	0	1	0	link	0	0	0	0	link
libsass	MLK	UEC	1	0	1	0	link	1	1	2	0	link
memcached	MLK	UEC	0	0	0	2	link	0	0	0	0	link
memcached	MLK	UEC	0	0	0	0	link	0	0	0	0	link
h3	MLK	UEC	0	0	0	0	link	0	0	0	0	link
TrinityEmulator	MLK	UEC	0	0	0	0	link	0	0	0	0	link
linux	MLK	MSC	1	0	1	0	link	1	0	1	0	link
linux	MLK	MSC	1	0	1	0	link	1	0	1	0	link
rtl_433	MLK	MSC	0	0	0	0	link	0	1	1	0	link
libuv	MLK	MSC	1	0	1	0	link	1	0	1	0	link
TrinityEmulator	MLK	MSC	1	1	2	0	link	1	0	1	0	link
binutils-gdb	MLK	MSC	0	0	0	0	link	0	0	0	0	link
Total			11	4	15	6		7	3	10	3	

A Analysis of BugBot and CodeRabbit

We evaluate two industrial LLM-based bug detectors, namely Cursor BugBot and CodeRabbit, upon our curated dataset. Both tools are commercial agents that can be integrated within GitHub repositories for automatically scanning pull requests. In our experiments, we simulate a GitHub pull request containing the target file associated with the original bug and collect only those reported issues that match the intended bug type for reproduction. We present detailed evaluation results, including links to the corresponding issues identified by each tool, along with representative examples of false positives and false negatives.

Listing 1: An OOB false negative example missed by Cursor BugBot and CodeRabbit.

```
static int parse_encap_seg6(struct rtattr *tb, struct in6_addr *segs){
  struct rtattr *tb_encap[256] = {};
netlink_parse_rtattr_nested(tb_encap, 256, tb);
  ...
  return 0;
}

void netlink_parse_rtattr_nested(struct rtattr **tb, int max, struct rtattr *rta){
  netlink_parse_rtattr(tb, max, RTA_DATA(rta), RTA_PAYLOAD(rta));
}

void netlink_parse_rtattr(struct rtattr **tb, int max, struct rtattr *rta, int len){
  memset(tb, 0, sizeof(struct rtattr *) * (max + 1));
  ...
}
```

Listing 2: A DBZ false negative example missed by Cursor BugBot and CodeRabbit.

```
static unsigned int get_number(FILE *fp) {
  int c, val;
  c = fgetc(fp);
  ...
  val = 0;
  while (isdigit(c)) {
    val = 10*val+c-'0';
    if (is_plain_pbm)
      break;
    c = fgetc(fp);
    if (c == EOF)
      die("%s: end of file\n", filename);
  }
  return val;
}

static unsigned int get_number255(FILE *fp, unsigned int maxval) {
  unsigned int val = get_number(fp);
  return (255*val+maxval/2)/maxval;
}

static void read_image(void) {
  ...
  case '2':
     Plain PGM
    maxval = get_number(fp);
    for (i = 0; i < logo_height; i++)
      for (j = 0; j < logo_width; j++)
        logo_data[i][j].red = logo_data[i][j].green =
       logo_data[i][j].blue = get_number255(fp, maxval);
    break;
    ...
}
```

A.1 Evaluation Result

The detailed evaluation results for BugBot and CodeRabbit are presented in Table 6. Overall, BugBot successfully reproduces 11 known cases and 4 new bugs, including 4 out-of-bounds bugs, 5 divide-by-zero bugs, and 6 memory leak bugs. In comparison, CodeRabbit reproduces 7 cases and discovers 3 new bugs, most of which are memory leak bugs. Notably, CodeRabbit detects only 1 out-of-bounds bug and 2 divide-by-zero bugs, showing its difficulty in handling complex bug patterns. In general, both detectors demonstrate relatively high precision, but suffer from low recall.

A.2 Examples of False Negatives

In Listing 1, the function `parse_encap_seg6` defines an array with 256 elements and passes it to `netlink_parse_rtattr_nested`, along with the parameter `max` set to 256. This function then forwards both arguments to `netlink_parse_rtattr`, which eventually calls `memset(tb, 0, sizeof(struct rtattr *) * (max + 1))`, resulting in an out-of-bounds write. This bug is successfully detected by BugScope but missed by both BugBot and CodeRabbit. In practice, the functions involved in this case span multiple files, each containing thousands of lines of code. Due to limited context length, both BugBot and CodeRabbit fail to recover the inter-procedural dependencies required to detect this vulnerability.

Listing 3: The OOB false positive example reported by Cursor BugBot.

```
bool _cpp_stack_file (cpp_reader *pfile, _cpp_file *file, include_type type,
    location_t loc) {
  char *buf = nullptr;
  ...
  if (!file->header_unit && type < IT_HEADER_HWM
     && type != IT_INCLUDE_NEXT
     && pfile->cb.translate_include)
  buf = (pfile->cb.translate_include(pfile, pfile->line_table, loc, file->path));
  ...
  size_t len = strlen (buf);
  buf[len] = '\n';
  cpp_buffer *buffer =
    cpp_push_buffer (pfile, reinterpret_cast<unsigned char *> (buf), len, true);
  buffer->to_free = buffer->buf;
  ...
}
```

Listing 4: The OOB false positive example reported by CodeRabbit.

```
FileNamesTable* UTIL_createExpandedFNT(const char* const* inputNames, size_t nbIfns,
    int followLinks) {
  unsigned nbFiles;
  char* buf = (char*)malloc(LIST_SIZE_INCREASE);
  char* bufend = buf + LIST_SIZE_INCREASE;
  ...
  size_t ifnNb, pos;
  for (ifnNb = 0, pos = 0; ifnNb < nbFiles; ifnNb++) {
    fileNamesTable[ifnNb] = buf + pos;
    if (buf + pos > bufend) { free(buf); free((void*)fileNamesTable); return NULL; }
    pos += strlen(fileNamesTable[ifnNb]) + 1;
  }
  ...
}
```

In Listing 2, the function `get_number` sets `val` to 0 if `fgetc` reads the character `'0'` from the input file. This value is then returned and propagated to the function `read_image`, where it is assigned to the variable `maxval`. Subsequently, `maxval` is passed as the second argument to `get_number255`, where it is used as a divisor without any prior validation. While this bug is correctly identified by BugScope, both BugBot and CodeRabbit fail to detect it. Detecting this issue requires interprocedural data flow tracking between the variables `val` in `get_number` and `maxval` in `get_number255`, along with reasoning about the value range of `val` under different input conditions. Both BugBot and CodeRabbit struggle with such complex, context-dependent vulnerability scenarios.

A.3 Examples of False Positives

In Listing 3, the variable `buf` is assigned the return value of the function `pfile->cb.translate_include`. The function then calculates the length of the content in `buf` using `strlen(buf)`, and sets the next position to `'\n'`. BugBot reports a buffer overflow for this operation. However, `strlen(buf)` only describes the length of valid characters in `buf` up to the null terminator, and does not necessarily reflect the actual allocated size of the buffer. Without enough context regarding the allocation of `buf`, we cannot conclusively determine that `strlen(buf)` represents the actual length of the buffer.

In Listing 4, the variable `buf` is allocated a fixed memory region of size `LIST_SIZE_INCREASE`, and `bufend` is set to point to the end of this buffer. In the loop, each element of `fileNamesTable` is assigned an offset within `buf` using `fileNamesTable[ifnNb] = buf + pos`. Before accessing the memory, the program checks `if (buf + pos > bufend)` to ensure the access stays within bounds. CodeRabbit flags this comparison as a potential undefined behavior due to possible pointer overflow in `buf + pos`. However, `pos` is incremented gradually based on the length of the input filenames using `pos += strlen(fileNamesTable[ifnNb]) + 1`. Triggering an overflow in `buf + pos` would require `pos` to approach the maximum value of `size_t`, which in practice would require an unrealistically large number of large input strings (e.g., more than 10^{18}). Therefore, this report is a false positive, as such extreme conditions are virtually impossible to reach in realistic scenarios.