ARTIFACT EVALUATED
USENIX
AVAILABLE

BugAuditor: Detecting Bugs via Inconsistent Defensive Code Auditing

Miaoqian Lin
The University of Hong Kong

Kai Chen
Institute of Information Engineering, Chinese Academy of Sciences

Hao Chen
The University of Hong Kong

Abstract

Modern software systems contain complex behaviors that are prone to bugs when handled incorrectly. Detecting such bugs requires reliable oracles, which are difficult to construct as they require project-specific knowledge. Prior studies mainly obtain oracles from comparable code deviations, documentation, or historical bugs. However, these sources are inherently limited, leaving many project-specific bugs undetected.

In this paper, we propose inconsistent defensive handling as a new bug oracle for LLM-driven bug auditing. Specifically, we observe that real-world systems contain abundant defensive code, where developers proactively apply defensive handling in security-sensitive contexts to prevent bugs, and inconsistencies in such handling can indicate bugs. To realize this idea, we design BugAuditor, a framework that performs bug auditing via inconsistent defensive handling. BugAuditor first locates defensive code snippets across the codebase. It then reasons about the code to infer its security intent and underlying defensive patterns, characterizing the associated security-sensitive behaviors and defensive handling. Finally, BugAuditor applies the inferred patterns to audit the codebase and reports inconsistent defensive handling of the same security-sensitive behaviors across different contexts.

We evaluate BugAuditor on the Linux kernel. The results show that BugAuditor effectively mines project-specific knowledge embedded in defensive code that existing methods miss. Using the inferred defensive patterns, BugAuditor detects 54 long-latent bugs, including resource leaks, information leaks, and invalid pointer dereferences. To date, 20 bugs have been confirmed and fixed in the latest version, and two have been assigned CVE identifiers.

1 Introduction

As modern software systems continue to grow in scale and complexity, they increasingly incorporate security-sensitive behaviors that are prone to security bugs. Detecting such bugs at scale remains challenging, as they often depend on project-specific knowledge and lack clear oracles for determining buggy behavior. To address this problem, existing methods obtain detection oracles from various sources, including code deviations [14, 21, 51], documentation [33, 42], and historical bug patches [12, 22, 27, 44]. However, these oracles are inherently limited in coverage, leaving many project-specific bugs undetected. Specifically, code deviation-based methods detect bugs by finding deviations from similar code snippets in the same codebase [21, 51]. However, these methods focus on syntactic co-occurrence statistics rather than code semantics. They can easily miss rare but critical patterns while introducing security-irrelevant noisy patterns. Documentation-based methods detect bugs by finding violations of documented specifications [33], but are limited by the availability and quality of documentation. Similarly, historical bug-based methods [27, 47] are limited to disclosed bug patterns. Table 1 summarizes these oracles and their characteristics.

Recently, large language models (LLMs) have been widely applied to bug detection [17, 24, 38]. However, the lack of effective oracles remains a key problem for LLM-driven bug detection. Directly applying LLMs to bug auditing relies heavily on the pretrained knowledge of LLMs. As a result, they can easily miss project-specific bugs and generate numerous hallucinated results that are difficult to analyze. Alternatively, combining LLMs within existing oracles, such as historical bug-based [16, 26, 41, 49] or documentation-based oracles [36, 50, 54], is still constrained by inherent limitations of these oracles. Given this, we are motivated to explore a new bug oracle for the LLM era, one that leverages LLMs' strengths and goes beyond existing oracles.

In this paper, we propose inconsistent defensive handling as a new bug detection oracle for LLM-driven bug auditing. Specifically, we observe that real-world systems contain rich *defensive code*, where developers proactively introduce defensive handling around security-sensitive behaviors to prevent bugs. These security-sensitive behaviors involve the same underlying security risks across different contexts and therefore should be protected consistently. As a result, inconsistent defensive handling of the same security-sensitive behavior may indicate bugs and serve as a bug oracle. In this way, un-

Table 1: Comparison of existing detection oracles and ours

Oracle	Characteristics
Historical bug-based	Detect bugs that are syntactically or semantically similar to known ones, but are limited to known bug patterns.
Code deviation-based	Detect bugs that deviate from comparable code, but rely on syntactic co-occurrence statistics or human heuristics, easily miss critical patterns and include irrelevant ones.
Documentation-based	Detect bugs that violate explicit specifications in documentation, limited by the quality of documentation.
Ours	Detect bugs based on inconsistent defensive code, focusing on semantically meaningful defensive behaviors across the codebase and covering diverse security-relevant patterns.

like historical bug-based methods that are limited to security-sensitive behaviors in patches, we can capture codebase-wide behaviors. Meanwhile, although some code deviation-based methods such as Engler et al. [14] tried to detect similar code inconsistencies to some extent, they were limited by traditional analysis. They either rely on syntactic and statistical analysis or human-defined heuristics. In contrast, benefiting from advances in LLMs, we propose to directly target semantically meaningful defensive behaviors, reason about their underlying security intent to infer defensive patterns, going beyond traditional syntactic code mining. Table 1 also compares our idea with existing oracles.

Based on the above idea, we present BugAuditor, a framework for LLM-driven bug auditing based on inconsistent defensive handling. BugAuditor operates in three stages: collecting defensive code across the codebase, reasoning defensive patterns from the collected code, and auditing bugs based on inconsistent defensive handling. Specifically, BugAuditor uses defensive operations as entry points to locate defensive code. It collects defensive operations from bug patches and identifies their usages across the codebase. For each usage, BugAuditor performs intra-procedural analysis to confirm the presence of the corresponding security-sensitive context and retain valid usages. The resulting functions are treated as defensive code. After collecting defensive code, BugAuditor infers the defensive patterns through reverse reasoning. Static analysis constrains the analysis scope using control- and data-flow relations. LLMs then perform semantic reasoning to understand risks guarded by the defensive operations and infer the security-sensitive behaviors. Finally, BugAuditor applies the inferred defensive patterns to audit the codebase for inconsistent defensive handling. It uses AST-based search to locate comparable functions and leverages LLMs to assess whether the security-sensitive behavior and its defensive handling are consistent with the pattern. When necessary, BugAuditor provide additional context for reasoning. Finally, BugAuditor reports detected inconsistencies with detailed reports.

We implement BugAuditor and evaluate it on the Linux kernel, a large-scale and widely-used program with complex and diverse code. Given six bug patches as seeds for defensive operation collection, BugAuditor locates 62 432 defensive code snippets and infers 16 508 unique defensive patterns. These patterns capture project-specific knowledge that goes beyond existing methods. Using the inferred defensive patterns, BugAuditor detects 54 new long-latent bugs, with an average lifetime over five years. The detected bugs span multiple types, including resource leaks, information leaks, and invalid pointer dereferences. Most of the bugs were missed by existing methods that construct oracles from documentation, historical bugs, or code deviations, as well as by current LLM-driven bug detection methods, showing that BugAuditor can detect bugs beyond the reach of existing methods. To date, 20 of these bugs have been confirmed and fixed in the latest version, with two assigned CVE identifiers. Our results show that explicit oracles enable LLMs to better leverage their semantic reasoning capabilities for bug auditing.

Contributions. Our contributions are summarized as below:

- **New idea.** We propose a new bug oracle based on inconsistent defensive handling. Leveraging code understanding capabilities of LLMs, this oracle focuses on semantically meaningful defensive behaviors across the codebase and detects inconsistent defensive handling as potential bugs.
- **New framework.** We design BugAuditor, a framework for large-scale LLM-driven bug auditing. BugAuditor combines static program analysis and LLM-based reasoning to locate defensive code, infer implicit defensive patterns, and detect bugs via inconsistent defensive handling.
- **New discoveries.** Our evaluation on the Linux kernel shows that BugAuditor effectively mines project-specific knowledge in defensive code and detects 54 long-latent bugs spanning diverse operations, 20 of which have been confirmed and fixed, and two have been assigned CVE identifiers.

2 Motivating Example

In this section, we use a case study to illustrate why existing methods fail and to motivate the need for a new detection oracle. Figure 1 shows a new bug in the Linux kernel. In this example, the function invokes `of_clk_get_by_name` to acquire a clock resource, but fails to release the acquired object by calling `clk_put` along the execution path. This bug relies on project-specific knowledge that is not explicitly documented. As a result, existing detection methods lack an effective oracle and cannot detect such bugs in practice. Next, we analyze why existing methods are ineffective in this case.

- **Documentation-based bug detection.** These methods extract detection rules from documentation and detect bugs by checking rule violations. However, documentation is often incomplete, causing many bugs to be missed. For example, the documentation of `of_clk_get_by_name` fails to require that the returned clock be released after use or to mention

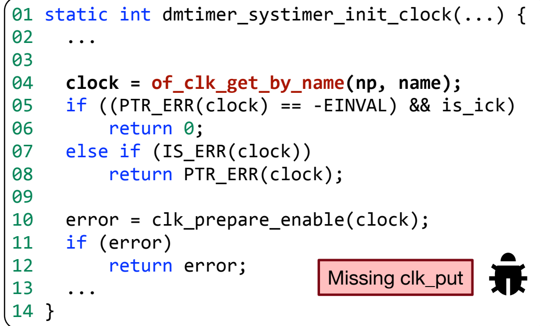

Figure 1: A newly detected bug in the Linux kernel

```
// Patch of CVE-2023-52661
01 int tegra_dc_rgb_probe(struct tegra_dc *dc) {
02  rgb->pll_d_out0 = clk_get_sys(NULL, "pll_d_out0");
03  ...
04  rgb->pll_d2_out0 = clk_get_sys(NULL, "pll_d2_out0");
05  if (IS_ERR(rgb->pll_d2_out0)) {
06  +  clk_put(rgb->pll_d_out0);
07     return err;
08  }
09  ...
10 }
```

Figure 2: A bug patch related to `clk_get_sys`

`clk_put`. As a result, documentation-based tools lack the necessary knowledge to detect this bug.

• **Code deviation-based bug detection.** These methods detect bugs by finding deviations from comparable code. They mainly fall into two types. The first is frequent code pattern-based detection, which mines frequent code patterns and treats deviations as bugs [21, 51]. However, these methods rely on syntax-level statistical co-occurrence and may miss rare but critical patterns while introducing irrelevant patterns. For example, APP-Miner [21] infers API usage patterns by extracting the maximum frequent subgraphs of API path graphs. In the motivating case, callers of `of_clk_get_by_name` almost always perform error checking after the call, while `clk_put` appears in fewer callers because resource release may occur in different functions, making it much less frequent from an intra-function perspective. As a result, the mined pattern drops the critical `clk_put`, leading to the missed bug.

The second type is one-to-one inconsistency-based bug detection [11, 30]. These methods compare two related code snippets or paths to detect inconsistency. However, they often rely on human-defined heuristics to locate the code for checking, making it difficult to scale to diverse behaviors. For example, IPPO [30] lacks the knowledge that `of_clk_get_by_name` is critical by default and therefore fails to locate the relevant code. Moreover, the buggy function contains no comparable paths for inconsistency checking, as `clk_put` is missing on all relevant paths. As a result, the bug remains undetectable even if the critical operation was manually provided. Other methods such as FICS [11] do not scale to large systems like the Linux kernel due to heavy analysis.

• **Historical bug-based bug detection.** These methods extract bug patterns from historical bug patches and detect similar bugs in the codebase, using either traditional analysis techniques [12, 27] or LLM-assisted methods [16, 49]. However, their effectiveness is fundamentally limited by the coverage of existing bug patches. Figure 2 shows a patch adding a missing `clk_put` after `clk_get_sys`. Existing historical bug-based methods can only capture patterns tied to the security-sensitive operation `clk_get_sys` explicitly exposed in patches. Thus, they can obtain the `clk_get_sys` pattern but miss bugs involving `of_clk_get_by_name`, because no historical patch captures that `of_clk_get_by_name` typically requires a corresponding `clk_put` after resource usage.

• **LLM-driven bug detection.** These methods apply LLMs to audit bugs [17, 25]. In practice, such methods rely either on LLMs' internal knowledge or on manually provided prior knowledge to guide LLM analysis. As a result, they struggle to detect bugs that depend on project-specific semantics. For example, RepoAudit [17] relies on predefined heuristics such as standard allocation functions, but lacks support for project-specific functions. Consequently, call sites of `of_clk_get_by_name` fall outside its analysis scope, preventing it from detecting the bug. Moreover, directly applying LLMs to audit bugs often produces numerous reports that are difficult to interpret and validate.

In summary, existing detection oracles rely on human-defined heuristics, documentation, historical bugs, or code deviations, all of which provide incomplete project-specific knowledge. As a result, both traditional and LLM-based methods may miss bugs when the required knowledge is absent from the oracle. Given recent advances in LLMs, we are motivated to explore a new detection oracle that leverages LLMs' semantic reasoning to capture project-specific knowledge, thereby going beyond traditional bug oracle construction.

3 From Defensive Code to Bug Oracle

In this paper, we design an effective oracle for LLM-driven bug auditing. Specifically, we observe that real-world systems contain abundant *defensive code*. Different from conventional defense mechanisms, we use the term *defensive code* broadly to refer to code that developers proactively introduce in security-sensitive contexts to prevent bugs or unsafe states. Missing or incorrectly applying defensive handling can either directly introduce bugs (e.g., missing resource release after use) or may leave the system in unsafe states in rare conditions (e.g., missing guards against risky behaviors).

Defensive code acts as latent patches embedded in the codebase, reflecting developers' experience in anticipating and mitigating error-prone behaviors. Similar to bug patches, defensive code encodes project-specific knowledge and serves as a valuable source of implicit specifications. At the same time, defensive code differs from bug patches in several key

aspects, as summarized in Table 2. Unlike bug patches, defensive code is introduced proactively to prevent bugs rather than in response to discovered bugs. It is widely scattered across the codebase and leaves no explicit trace in version history. More importantly, defensive code is far richer than bug patches, as most security-sensitive contexts are protected proactively and never evolve into publicly reported bugs.

Consequently, defensive code provides a valuable source of project-specific knowledge for bug detection by revealing which operations and contexts are security-sensitive. If we can effectively leverage this implicit knowledge, we can expand the coverage of bug detection beyond existing methods.

Table 2: Differences between bug patches and defensive code

Aspect	Patch code	Defensive code
Timing	Added after bugs are discovered	Written proactively to prevent bugs
Traceability	Easy to locate via commits	Hard to locate from large codebases
Amount	Limited to disclosed bugs	Many instances hidden in codebase

Defensive pattern. To leverage the knowledge embedded in defensive code, we first use *defensive patterns* to capture its key semantics. A defensive pattern describes which code contexts are security-sensitive and what behaviors are introduced to prevent bugs. Accordingly, it consists of two components: ❶ *Security-sensitive behaviors*: behaviors that are inherently error-prone and require proper handling, as improper or missing handling may lead to security bugs. ❷ *Defensive behaviors*: behaviors introduced to prevent bugs in code contexts involving security-sensitive behaviors. In this paper, we use the term *defensive behavior* to refer to any code explicitly used to prevent security bugs. This includes behaviors that are mandatory in every scenario of security-sensitive behaviors as well as behaviors that prevent bugs only in rare conditions.

Security-sensitive behaviors and their defensive behaviors have a clear causal relationship: defensive behaviors are introduced to mitigate the risks associated with specific security-sensitive behaviors. We manually summarize representative kinds of security-sensitive behaviors and their defensive behaviors, as shown in Table 3. As shown, security-sensitive behaviors include operations such as resource allocation, data copying, pointer dereference, and array access. Correspondingly, defensive behaviors include operations such as resource release, initialization, NULL checks, bounds checks, and permission validation. Missing or incorrect defensive handling of security-sensitive behaviors may result in security bugs.

Bug oracle. Based on these observations, we propose a new detection oracle based on inconsistent defensive handling. The rationale is that the same security-sensitive behaviors share the same underlying risks and therefore should have consistent defensive handling. Inconsistent defensive handling of the same security-sensitive behavior across different contexts can indicate bugs. With this oracle, we can leverage rich knowledge embedded in defensive code, going beyond

Table 3: Mapping between security-sensitive behaviors and corresponding defensive behaviors

Security-sensitive behaviors	Defensive behaviors
Resource allocation and use	Release resource after usage
Copying kernel data to user space	Initialize data before copying
Pointer dereference	NULL check before access
Array or buffer access	Bounds check before access
String copy or manipulation	NULL termination for strings
Access to privileged functions	Permission check
Error-returning function call	Error check before use

known bugs, and provide explicit guidance for bug auditing. This oracle relies on inconsistency analysis without assuming the correctness of any single instance. Moreover, this oracle differs from prior code deviation-based methods [14], which capture limited syntax-level patterns. It targets semantically meaningful defensive behaviors and infers their underlying security intent without requiring predefined templates.

For example, Figure 3 shows a defensive code snippet in the function `lpc32xx_clocksource_init`, where `clk_put` is used to release a previously acquired clock, preventing clock resource leaks. This defensive logic is required because `of_clk_get_by_name` acquires a clock resource and therefore requires proper cleanup. Such defensive code provides a concrete reference for bug auditing. By extracting the underlying defensive pattern, we can audit other code that performs the same security-sensitive operation and check whether cleanup is handled consistently. Inconsistent defensive handling for `of_clk_get_by_name`, such as missing `clk_put`, indicates potential bugs. For example, the function in Figure 1 invokes `of_clk_get_by_name` but applies inconsistent defensive handling, indicating a bug.

```
01 static int __init lpc32xx_clocksource_init(...) {
02   ...
03   clk = of_clk_get_by_name(np, "timerclk");
04   if (IS_ERR(clk)) {
05       ...
06   }
07   ret = clk_prepare_enable(clk);
08   if (ret) {
09       pr_err("clock enable failed (%d)\n", ret);
10       goto err_clk_enable;
11   }
12   ...
13  err_clk_enable:
14     clk_put(clk);
15     return ret;
16 }
```

Figure 3: A defensive code snippet in the Linux kernel

4 Overview

To realize the above idea, we design BugAuditor, a framework that performs bug auditing via inconsistent defensive handling. As shown in Figure 4, BugAuditor consists of three key stages: ① *Defensive code collecting*: collecting defensive

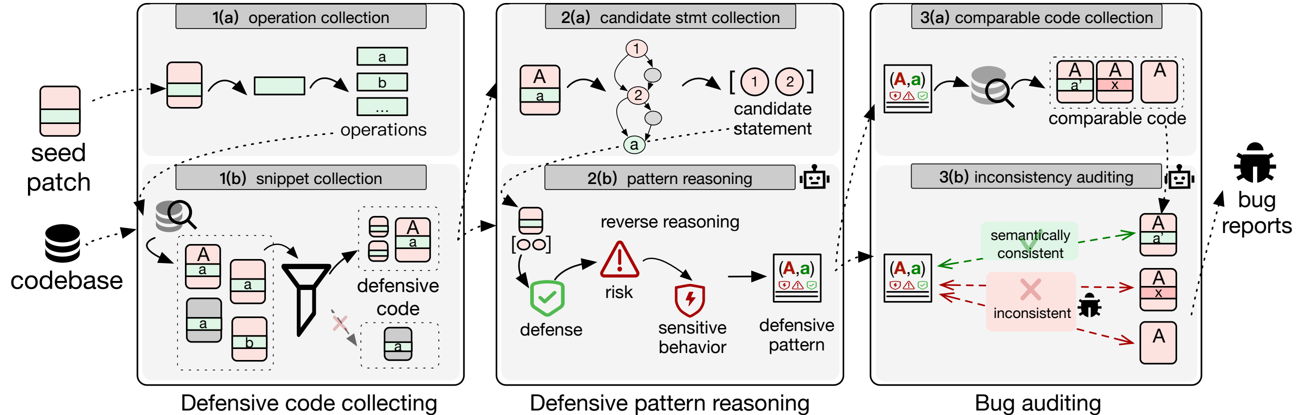

Figure 4: The overall workflow of BugAuditor

code snippets across the codebase, which serve as auditing references. ② *Defensive pattern reasoning*: inferring defensive patterns implied by the located snippets. ③ *Bug auditing*: applying the inferred defensive patterns to detect bugs.

Specifically, BugAuditor first locates defensive code in the codebase to mine project-specific knowledge. Instead of enumerating security-sensitive operations, which are highly diverse and difficult to cover exhaustively, BugAuditor uses defensive operations as entry points. It collects defensive operations from historical bug patches as seeds and locates their usages across the codebase. BugAuditor collects defensive code at the function level. To ensure that the collected code includes the corresponding security-sensitive behavior and provides sufficient context for pattern reasoning, BugAuditor analyzes how the variable protected by the defensive operation is introduced and propagated within each function.

After collecting defensive code snippets, BugAuditor adopts a hybrid strategy that combines static program analysis and LLM-based semantic reasoning to infer the defensive pattern. Specifically, BugAuditor first applies dominator-based and data dependency analysis to collect candidate statements for security-sensitive behaviors, and groups equivalent contexts based on these candidates. For each group, BugAuditor provides the candidate statements together with the defensive snippets to guide LLM-based reverse reasoning. Starting from the observed defensive operation, the LLMs reason about the risk it mitigates and identify the underlying security-sensitive behavior, thereby inferring the defensive pattern.

Finally, BugAuditor uses the inferred defensive patterns to audit the codebase for inconsistent defensive handling. BugAuditor first employs AST-based search to locate comparable functions that may exhibit the same security-sensitive behavior. It then uses LLMs to check whether the behavior matches the pattern and whether defensive handling is consistent. When necessary, LLMs can request additional function-level context, which BugAuditor provides to support better reasoning. Detected inconsistencies are reported as potential bugs, together with detailed explanations.

Example. Next, we walk through the motivating example to illustrate how BugAuditor leverages the defensive code in Figure 3 to detect the buggy code in Figure 1. Specifically, BugAuditor first collects `clk_put` from a bug patch (shown in Figure 2) and treats it as a defensive operation, as it is introduced to fix a disclosed bug. Using `clk_put` as an analysis entry point, BugAuditor examines its usage across the codebase and collects defensive code snippets, including the one shown in Figure 3. Given such a defensive snippet, BugAuditor then infers the defensive pattern by analyzing which security-sensitive behaviors necessitate the use of `clk_put`. By combining control-flow and data-flow analysis with LLM-based semantic reasoning, BugAuditor infers the underlying security-sensitive behavior. Specifically, it determines that the call to `of_clk_get_by_name` acquires a clock resource that should be released after use. Based on this inferred intent, BugAuditor derives the corresponding defensive pattern. Based on this pattern, BugAuditor locates other code regions that involve the same security-sensitive behavior, such as the function shown in Figure 1. Further auditing reveals inconsistent defensive handling between the two code snippets. Finally, BugAuditor reports this inconsistency as a potential bug, which is confirmed to be a clock resource leak.

5 Design

In this section, we present the design of BugAuditor. As shown in Figure 4, BugAuditor proceeds through three stages: defensive code collecting, defensive pattern reasoning, and bug auditing. We describe each stage in detail below.

5.1 Defensive Code Collecting

To analyze defensive code, the first challenge is to collect defensive code snippets in the codebase. Unlike bug patches, which can be directly obtained from commit history, defensive code has no explicit annotations and is scattered throughout the codebase. Existing methods [30] often rely on predefined

security-sensitive operations to locate relevant code. However, such operations are highly diverse and system-specific, making manually defined heuristics inherently incomplete.

To address this problem, we propose using defensive operations as entry points to locate defensive code. This design is motivated by two key observations. First, defensive operations have simple and explicit semantics, as they are introduced to mitigate specific security risks, making them easier to identify than security-sensitive operations with diverse functionality. Second, a single defensive operation can cover many security-sensitive operations. For example, a NULL check can appear in different contexts but consistently prevents NULL pointer dereference. By locating where defensive operations occur, we can identify security-sensitive code contexts without enumerating all such behaviors in advance.

Defensive operation collection. BugAuditor begins by collecting defensive operations as seeds for locating defensive code. Defensive operations are code constructs introduced to guard against potential security risks, such as condition checks and resource releases. Rather than manually specifying such operations, BugAuditor collects them from historical bug patches. Since bug patches are created to fix missing or incorrect protections, the newly added code naturally contains concrete defensive operations. Specifically, given a bug patch, BugAuditor analyzes its behaviors to identify the operations that were introduced to fix the bug and collects them as defensive operations. Variable names are then abstracted to enable generalization across different code contexts. Importantly, bug patches are used only to collect initial defensive operations, rather than to extract bug patterns. To expand the defensive operation set, BugAuditor identifies wrapper functions that encapsulate known defensive operations, as such operations may be helper functions to support different usage contexts. To identify such wrappers, BugAuditor first identifies the critical variable on which the defensive operation is performed. If a function simply forwards this critical variable to a known defensive operation without modifying it, BugAuditor includes it in the defensive operation set.

For example, Figure 2 shows CVE-2023-52661, where a clock resource acquired by `clk_get_sys` was not released when necessary. The patch fixes the bug by adding a missing `clk_put`. From this, BugAuditor identifies the newly added `clk_put` as a defensive operation. Although §2 describes a scenario in which no historical patch is available for `of_clk_get_by_name`, this example shows that BugAuditor extracts `clk_put` from a patch involving a different operation `clk_get_sys`. BugAuditor then expands the defensive operation set by identifying related wrapper functions, such as `clk_bulk_put_all`, `dss_put_clocks`, and `clk_bulk_put`.

Defensive code snippet identification. After identifying defensive operations, BugAuditor uses them as anchors to locate defensive code snippets. A code snippet is considered valid defensive code only if it includes the security-sensitive behavior, which is essential for inferring defensive patterns. This makes inter-procedural defensive code analysis challenging, as security-sensitive behavior may span multiple functions and complex call chains, while overly broad contexts often introduce substantial noise and reduce reasoning precision.

To balance analyzability and precision, we analyze defensive code at the function level, which provides clear boundaries for control flow and local variable scope. At the same time, a function containing a defensive operation does not necessarily constitute valid defensive code, as it may not include the corresponding security-sensitive behavior. To determine whether such behavior is present, BugAuditor analyzes how the variable protected by the defensive operation is introduced and propagated within the function. Specifically, if the variable is defined or reassigned within a function, the risk is likely introduced locally, and the function is considered to involve security-sensitive behavior and is collected.

Therefore, given a defensive operation, BugAuditor analyzes its usage across the codebase and collects the functions that use it. It then examines each function to determine whether it constitutes valid defensive code. Specifically, for each function, BugAuditor locates the use site of the defensive operation and identifies the variable it manipulates as the critical variable. BugAuditor then performs backward data-dependency analysis to trace the origin of this variable. If the critical variable is defined or reassigned within the function, the function is retained as a defensive code snippet. If the variable is only passed as a parameter, the function is excluded. After this, BugAuditor retains defensive code instances, each providing the necessary context for pattern reasoning.

For example, `clk_put` is a previously collected defensive operation. By analyzing its usage across the codebase, BugAuditor collects function-level defensive code, such as `samsung_pwm_alloc`, `s3c24xx_serial_enable_baudclk`, and `omap_dmic_select_fclk`, as shown in Figure 5. In each case, the variable released by `clk_put` is defined or assigned within the same function, indicating that the corresponding security-sensitive behavior is introduced locally. BugAuditor therefore retains these functions as defensive code.

5.2 Defensive Pattern Reasoning

After locating defensive code, BugAuditor infers the risks guarded by defensive operations and the security-sensitive behaviors that introduce them, forming the defensive patterns. However, directly applying LLMs for analysis is ineffective for two reasons. First, defensive snippets often contain unrelated logic. Pure program slicing lacks semantic understanding and may introduce noise, while full code context can overwhelm LLMs with irrelevant information. Second, many snippets share the same underlying security-sensitive behavior, thus repeatedly querying LLMs for semantically equivalent instances incurs unnecessary token cost.

To address this, we adopt a hybrid strategy that combines the strengths of static program analysis and LLMs. Static

```
01 int samsung_pwm_alloc(device_node *np,...) {
02  ...
03  pwm.timerclk = of_clk_get_by_name(np, "timers");
04  if (IS_ERR(pwm.timerclk)) {
05      ...
06  }
07  ret = _samsung_pwm_clocksource_init();
08  if (ret)
09      goto err_clocksource;
10
11 err_clocksource:
12  clk_put(pwm.timerclk);
13  return ret;
14 }
```

```
01 int s3c24xx_serial_enable_baudclk(...){
02  ...
03  clk = clk_get(dev, clk_name);
04  if (IS_ERR(clk))
05      continue;
06
07  ret = clk_prepare_enable(clk);
08  if (ret) {
09      clk_put(clk);
10      continue;
11  }
12  ...
13 }
```

```
01 int omap_dmic_select_fclk(struct omap_dmic *dmic){
02  ...
03  mux = clk_get_parent(dmic->fclk);
04  if (IS_ERR(mux)) {
05      ...
06  }
07  ret = clk_set_parent(dmic->fclk, parent_clk);
08  if (ret < 0)
09      goto err_busy;
10
11 err_busy:
12  clk_put(mux);
13  return ret;
14 }
```

Figure 5: Three defensive code snippets involving different security-sensitive operations that are all related to `clk_put`

analysis allows us to collect statements related to defensive operations through control-flow and data-flow relations. In contrast, LLMs can reason about code semantics and infer the security risks that defensive operations are intended to mitigate. In this way, BugAuditor can first groups equivalent instances through static analysis and then effectively infer defensive patterns for each group. Next, we describe how BugAuditor infers defensive patterns in two steps: candidate statement collection and pattern reasoning.

Candidate statement collection. BugAuditor first performs static analysis to collect candidate statements for security-sensitive operations. To do this, we introduce a dominator-based analysis that leverages the causal relation between security-sensitive behaviors and defensive operations. As the execution of defensive operations is conditioned on the presence of security-sensitive behaviors, the two typically exhibit a dominance relationship in the control-flow graph. Given a defensive code snippet and the known defensive operation, BugAuditor first constructs the control-flow graph of the function and locates the defensive statement. It then performs dominator analysis to identify statements that dominate or post-dominate the defensive statement, that is, statements appearing on all control-flow paths leading to or from the defensive statement. To further narrow the candidates, BugAuditor incorporates data dependency analysis. Considering that defensive operations and the behaviors they protect typically operate on the same variable, BugAuditor first extracts the variable manipulated by the defensive operation and then retains only statements related to this variable. It also includes assignments that define the variable.

Pattern reasoning. After collecting candidate statements, BugAuditor provides them to LLMs for semantic reasoning, since not all candidates correspond to security-sensitive behaviors and some may only serve functional purposes. BugAuditor treats two defensive contexts as equivalent if their candidate statements are identical or one contains the other. It then groups equivalent contexts and performs LLM reasoning only once per group, avoiding redundant invocations. Specifically, BugAuditor prompts LLMs to perform reverse reasoning starting from the defensive operation. As shown in Table 4, LLMs first interpret the behavior of the defensive operation, and then reason over the given code to identify the corresponding security-sensitive behavior. During this process, the candidate statements guide LLMs to critical code locations, and the full code provides surrounding semantics and execution conditions for better understanding. Based on the reasoning, BugAuditor obtains corresponding defensive patterns.

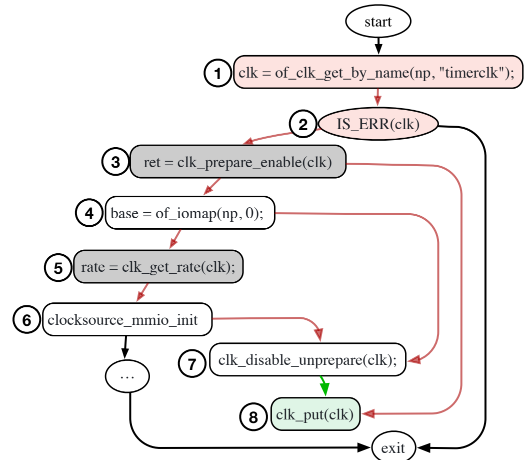

Figure 6: Simplified CFG of `lpc32xx_clocksource_init`

For example, consider the defensive code snippet in the function `lpc32xx_clocksource_init` shown in Figure 3, which involves the defensive operation `clk_put`. BugAuditor first constructs the control-flow graph (CFG) of the function. Figure 6 shows a simplified CFG. BugAuditor locates the node corresponding to the defensive operation `clk_put` and applies dominator analysis to collect statements that appear on all control-flow paths reaching this node. As a result, the collected candidate statements include `of_clk_get_by_name`, `IS_ERR`, and `clk_prepare_enable`. In contrast, although `clk_get_rate` operates on the same clock object, it does not appear on all paths leading to `clk_put` and is therefore excluded from the candidate set. In addition, statements unrelated to the clock variable are discarded by data dependency analysis. BugAuditor then provides the candidate statements together with the complete function body to the LLM for semantic reasoning. During this step, the LLM analyzes the security-sensitive behavior protected by the defensive operation and derives the corresponding defensive pattern. In this

Table 4: Prompt for defensive pattern reasoning

Task. You are given a code snippet extracted from a single function, along with one defensive operation already used in the code. Your task is to analyze the defensive pattern via reverse reasoning:
- Analyze what the defensive operation protects against. Examine where and when `{defensive_operation}` is executed in the code.
- Reverse reason to find security-sensitive behaviors. Based on what `{defensive_operation}` defends against, reason backwards to identify the core operation that creates the need for this defense.

Input. (1) Defensive code snippet: `{code_snippet}`
(2) Defensive operation: `{defensive_operation}`
(3) Related statement: `{candidate_statements}`

Output. Output the result in JSON format:
{ "security_sensitive_behavior": "...", "defensive_behaviors": "..." }

example, `clk_prepare_enable` is identified as a non-core operation and is excluded from the inferred defensive pattern. Finally, the LLM identifies `of_clk_get_by_name` as the security-sensitive operation and generates the patterns.

In addition, `clk_put` is used across multiple code contexts, and the associated security-sensitive operations may differ. As a result, the inferred defensive pattern can be summarized as a template, in which the specific security-sensitive operations vary. Table 5 summarizes the defensive pattern inferred by BugAuditor starting from the defensive operation `clk_put`. Besides `of_clk_get_by_name`, other contexts involve security-sensitive operations such as `clk_get` and `of_clk_get`. Although these APIs differ in functionality and usage context, they all involve acquiring clock resources whose lifetime must be explicitly managed. Failing to release these resources can lead to leaks, and thus the same defensive operation `clk_put` is required.

Table 5: Inferred defensive pattern related to `clk_put`

Defensive Pattern. *Security-sensitive behaviors*: When `{security-sensitive func}` is called, it creates a resource leak risk because failing to release the acquired clock reference with `clk_put` leads to unbalanced resource management.
Defensive behaviors: The `clk_put` is executed in cleanup sections to prevent resource leaks by releasing the clock reference acquired by `{security-sensitive func}`.

Security-sensitive functions. `clk_get`, `of_clk_get_by_name`, `clk_hw_get_clk`, `of_clk_get`, `clk_get_sys`, `clk_get_cpu`, `clk_get_optional`, `clk_get_parent`, `of_clk_get_from_provider`

5.3 Bug Auditing

After inferring defensive patterns, BugAuditor leverages them to detect bugs by auditing inconsistent defensive handling across the codebase. The key insight is that the same security-sensitive behavior carries the same underlying risk and should therefore be protected consistently. As a result, when a code location involves the same security-sensitive behavior but applies different or missing defensive handling compared to the pattern, this inconsistency may indicate bugs.

Table 6: Prompt for extracting AST query operations

Task. Your task is to extract key syntactic operations from the behavior. Only list operations explicitly mentioned in the behavior text.

Input. Behaviors: `{behaviors}`

Output. Output the result in JSON format:
{ "key_ops": ["<callee_fn>", ...] }

Given a defensive pattern, BugAuditor first locates comparable code locations that involve the same security-sensitive behavior. It then audits whether these locations apply consistent defensive handling as specified by the pattern, and generates bug reports when inconsistencies are detected.

Comparable function collection. BugAuditor begins by locating functions that exhibit the security-sensitive behavior specified in a defensive pattern. To identify such functions at scale, BugAuditor adopts AST-based search, which leverages key syntactic characteristics of the behavior to locate relevant code across the codebase. Specifically, we predefines a set of AST query templates based on the templates supported by the underlying AST search tool. Given a natural-language description of a security-sensitive behavior, BugAuditor asks the LLM to extract key syntactic elements, such as function names. It then instantiates the AST query templates with these extracted elements and runs the queries to collect relevant code snippets. The prompt for extracting key operations is shown in Table 6. These queries are not intended to precisely capture the full behavior, as some execution conditions may be difficult to express using AST alone. Instead, this step aims to roughly locate relevant code regions for subsequent auditing. For example, for the defensive pattern shown in Table 5, BugAuditor generates AST queries targeting the corresponding security-sensitive APIs and retrieves functions that invoke these operations. During this stage, BugAuditor excludes the reference functions used to infer the defensive pattern, avoiding trivial self-comparisons. Finally, BugAuditor obtains functions that perform the same security-sensitive behavior for subsequent inconsistency auditing.

Inconsistency auditing. After collecting comparable functions, BugAuditor further examines whether their defensive handling is consistent with the reference defensive pattern. Inconsistent handling for the same security-sensitive behavior indicates bugs. Rather than checking defensive operations syntactically, BugAuditor uses LLMs for code semantic reasoning, since defensive code varies widely in form.

Specifically, given a defensive pattern, its reference function, and a comparable function. BugAuditor prompts the LLMs to reason about potential inconsistencies. First, the LLMs check whether the security-sensitive behavior specified by the defensive pattern actually exists in the comparable function, since comparable locations are obtained through coarse-grained syntactic matching. If the behavior is confirmed, the LLMs then examines whether the function applies defensive handling consistent with the pattern.

Table 7: Prompt for bug detection via inconsistency auditing

Task. You are performing bug auditing based on defensive-handling consistency. Your goal is to determine whether an inconsistency leads to a real bug. Follow the reasoning steps below and avoid false positives. - Locate the security-sensitive behavior in the candidate function. - Determine whether defensive handling is required. - Determine whether the required defensive behavior is present. Only report a bug if: the security-sensitive behavior exists, defensive handling is required, and the defense is missing or semantically inconsistent. - If you are not sure, set judgment=uncertain and request more context. Context requests: - To request source code for a function: "SOURCE_OF:<func_name>" - To request caller code: "CALLER_OF:<func_name>"
Input. (1) Reference defensive pattern: `{defensive_pattern}` (2) Audited function code: `{function_code}` (3) Additional context (if any): `{extra_context}`
Output. Output the result in JSON format: { "judgment": "<consistent | inconsistent | uncertain>", "bug_explanation": "...", "needs_more_context": <true | false>, "requested_context": ["..."] }

Since the same behavior may have different syntactic forms, BugAuditor reasons about inconsistencies at the semantic level rather than relying on syntax. For example, for functions where a return value of zero indicates success and a negative value indicates an error, both a negative-value check (`if (ret < 0)`) and a non-zero check (`if (ret)`) express the same defensive logic and should be considered consistent. To reduce misjudgment caused by insufficient context, BugAuditor allows the LLM to request additional context, such as the source code of relevant caller or callee functions. Finally, BugAuditor detects inconsistencies, which occur when the defensive operation specified by the pattern is missing or when the applied operation is not equivalent to the pattern.

For example, with the defensive pattern in Table 5, BugAuditor detects the new bug in Figure 1. Specifically, BugAuditor first leverages the syntactic details of the security-sensitive behavior, such as the function `of_clk_get_by_name`, to locate comparable functions, including `dmtimer_systimer_init_clock`. It then checks whether these functions exhibit the same security-sensitive behavior and whether the corresponding defensive handling is applied consistently. Since `dmtimer_systimer_init_clock` contains no call to `clk_put` or any semantically equivalent operation, BugAuditor reports this inconsistency as a bug. To facilitate manual validation, BugAuditor generates bug reports that explain the detected inconsistency and provide the corresponding defensive code and inferred patterns, as shown in Figure 7. Note that `clk_put` is required on all paths after the clock resource is used. On the normal path, the acquired clock escapes the current function through assignment to a parameter and is expected to be released elsewhere during driver cleanup. Therefore, in this case, we focus only on the definite bug in the error paths of the current function.

```
"buggy_function": "dmtimer_systimer_init_clock",
"bug_explanation": "The function calls of_clk_get_by_name which increments the reference count of the clock. On error paths after successful clock acquisition (...), the function returns without calling clk_put to release the reference. This creates a reference count leak. "
"pattern_source_func": ...
"defensive_pattern": ...
```

Figure 7: Bug report for the motivating example

6 Implementation

We describe the key implementation details below. For initial defensive operation collection, BugAuditor uses PyDriller [37] to parse bug patches and extract code changes to identify defensive operations. tree-sitter [40] is used to parse code and capture syntactic elements such as function and variable names. For code retrieval, BugAuditor employs Weggli [43], an AST-based querying tool built on tree-sitter, to efficiently locate target snippets. Weggli is used during defensive code collection to gather operation usages and during bug detection to locate comparable functions and relevant context. For pattern reasoning, BugAuditor analyzes functions with Joern [48], exports CFG, and performs additional control-flow analysis with NetworkX [18] to obtain candidate statements. BugAuditor excludes snippets when the candidate statements involve function pointer calls because accurate indirect call resolution requires additional analysis beyond the current scope. For LLMs, we deploy DeepSeek-V3.2 [29] locally via vLLM and access it through an API endpoint. The temperature is set to 0 to reduce randomness, all other parameters follow the default settings.

7 Evaluation

We evaluate BugAuditor to answer the following questions.

- **RQ1.** How effective is BugAuditor at reasoning defensive patterns? (Please refer to §7.1)
- **RQ2.** Can BugAuditor detect new bugs in the real-world large-scale codebase? (Please refer to §7.2)
- **RQ3.** How effective are the key components of BugAuditor? (Please refer to §7.3)
- **RQ4.** How does BugAuditor compare with state-of-the-art bug detection tools? (Please refer to §7.4)
- **RQ5.** What is the generalizability of BugAuditor across programs? (Please refer to §7.5)
- **RQ6.** What are the performance and cost of BugAuditor? (Please refer to §7.6)

Dataset. We evaluate BugAuditor on the widely used open-source Linux kernel (v6.10-rc4). The codebase contains over 27 million lines of code across more than 70 000 files. Its scale and complexity introduce many error-prone operations, and bugs in the kernel can affect a broad range of downstream software, making it a critical target for analysis. To initiate the analysis, we use six patches for CVEs to collect defensive

operations as entry points (Table 8). These include three operations related to complex resource management and three check-related operations. For NULL checks, we also include common variants as shown.

Platform. We deploy the open-source model DeepSeek-V3.2 utilizing 8× H200 GPUs. The model is accessed through an API endpoint for evaluation. Other experiments were conducted on a 64-bit Ubuntu server equipped with 212 GB of memory and 2 TB of storage, powered by an Intel Xeon Gold 5218 CPU with 64 cores.

Table 8: The defensive operations used in the evaluation

Type	Operation	Code
Resource releases	Memory release	`kfree()` [6]
	Device reference release	`of_node_put()` [1]
	Clock reference release	`clk_put()` [2]
Security checks	Error pointer check	`if (IS_ERR(val))` [4]
	Negative value check	`if (val < 0)` [3]
	NULL pointer check	`if (!val)`, `if (val != NULL)` [5]

7.1 Effectiveness of Defensive Pattern Analysis

With six defensive operations, BugAuditor identifies 123 536 occurrences, collects 62 432 snippets, and infers 16 508 unique patterns. Table 9 shows statistics of defensive patterns. Patterns are considered identical within the same defensive operation if they involve the same security-sensitive operation. The inferred security-sensitive behaviors typically involve function-call operations that are project-specific and semantically diverse. The number of inferred patterns also varies across defensive operations, depending on their usage scope. For example, NULL pointer checks yield 6175 patterns because many functions return `NULL` to signal errors and require handling. In contrast, `clk_put` targets a specific resource type (clock resources) and yields only eight patterns.

Examples of inferred patterns. Table 18 presents cases of inferred defensive patterns. For each defensive operation, the table shows its main defensive pattern template and up to the top 30 corresponding security-sensitive functions. A defensive operation mitigates the same risk across contexts, yielding a shared pattern template while the associated security-sensitive operations vary. Results show that BugAuditor captures diverse security-sensitive operations that are difficult to identify via heuristic-based methods. For example, starting from the defensive operation `kfree`, BugAuditor obtains various resource allocations with non-typical characteristics. These include functions whose names do not explicitly convey allocation intent (e.g., `lg_wmab`), as well as functions that allocate resources via output parameters rather than return values (e.g., `device_get_devnode`). Moreover, BugAuditor reasons from usage code, rather than starting from low-level primitives and tracking lengthy data-flow, which reduces analysis overhead.

Occurrences of inferred patterns. We count the occurrences

Table 9: Statistics of defensive patterns for six seed operations

Operation	#Usage	#Collected snippet	#Pattern
`kfree`	27 749	11 921	922
`of_node_put`	2224	1664	102
`clk_put`	359	181	8
NULL pointer check	62 240	27 344	6175
Negative value check	20 086	16 114	7013
Error pointer check	10 878	5208	2288
Total	123 536	62 432	**16 508**

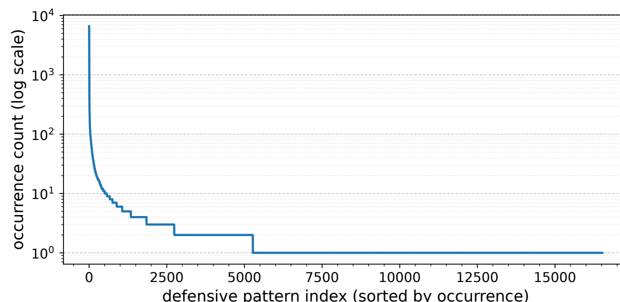

Figure 8: Long-tail occurrences of inferred defensive patterns

of each defensive pattern across the codebase. As shown in Figure 8, we sort defensive patterns by occurrence count and plot them by index, with occurrence counts shown on a logarithmic scale. The results show a clear long-tail trend: a small number of defensive patterns occur frequently, while most appear in only a few locations. In particular, 96.97% of defensive patterns occur fewer than 10 times. Only 0.22% of patterns occur more than 100 times. Frequent-pattern mining methods require patterns to occur often (e.g., more than 10 times) for mining [21], thus miss low-frequency patterns.

Accuracy of inferred patterns. We evaluate the accuracy of inferred defensive patterns by manual validation. Given the large number of inferred patterns, we randomly sample up to 100 patterns per defensive operation, yielding 508 samples in total. Each defensive pattern describes security-sensitive behaviors. We assess only whether the described operation is correct, as it captures the core behavior. Other contextual details, such as bug type descriptions, do not affect inconsistency auditing and are excluded. We start from the defensive operation and examines related variables and surrounding code to identify the security-sensitive behavior. When necessary, we inspect the source code of related functions.

Among sampled defensive patterns, 469 (92.3%) are correctly inferred. Results show that the accuracy of varies across defensive operations. Patterns related to `kfree` and `of_node_put` show the lowest accuracy, mainly due to complex resource-management scenarios in the Linux kernel. In contrast, `clk_put` achieves 100% accuracy. Its security-sensitive operations all include clear semantic cues, such as subwords such as `clk` and `get`, making them easier for LLMs to reason about. Defensive checks (e.g., NULL pointer checks) also achieve high accuracy, as they often involve

Table 10: Pattern accuracy per defensive operation

Defensive operation	#Sample	Accuracy
kfree	100	88%
of_node_put	100	75%
clk_put	8	100%
NULL pointer check	100	100%
Negative value check	100	99%
Error pointer check	100	99%
Total	508	**92.3%**

immediate return-value checks with simple control- and data-flow relations. Further analysis shows that the 39 incorrect cases stem from two causes. *(1) Incorrect security-sensitive operation reasoning.* BugAuditor misidentifies the security-sensitive operation in complex data-flow contexts. Although the correct behavior is present, multiple related operations appear simultaneously, increasing reasoning difficulty and leading to incorrect results. *(2) Incorrect defensive code snippet collection.* One major cause is incomplete defensive code collection. The snippet lacks sufficient security-sensitive context: the variable originates from external input but is misidentified as locally defined, leading to incorrect reasoning. We observe one misclassification where a functional negative check is mistaken for a defensive operation, leading to inaccuracies.

Extensibility analysis. To demonstrate the extensibility of BugAuditor, we use function-related seed defensive operations to automatically collect additional operations and mine defensive patterns. We add `kref_put` as a seed because it is a core Linux kernel primitive for releasing reference counters, closely related to a family of reference-management routines, similar to `kfree` for memory release. From these seeds, BugAuditor identifies many related defensive operations, largely due to diverse kernel scenarios where higher-level wrappers encapsulate resource-release logic. Here, we evaluate whether extending defensive operations uncovers additional patterns. To ensure accuracy under complex data flows, we adopt conservative constraints: we consider only wrappers whose names contain subwords such as `free` or `put`, and limit the wrapper call depth to five levels. More complete and precise identification requires further investigation. The results are shown in Table 11, showing that a single seed can expand to multiple operations and corresponding patterns, broadening the knowledge coverage of BugAuditor. BugAuditor confines defensive pattern reasoning to a single function, causing operations with cross-function handling to yield no patterns. Addressing these cases requires interprocedural analysis, discussed in §8.

Table 11: Statistics of wrappers and patterns from seeds

Seed operation	#Wrapper of seed	#Inferred pattern
kfree	6301	334
of_node_put	28	1
clk_put	31	12
kref_put	1583	388
Total	7943	735

7.2 Effectiveness of New Bug Detection

BugAuditor uses the mined defensive patterns for bug auditing. To balance analysis cost and manual validation effort, we configure BugAuditor to randomly audit up to 100 comparable functions for each defensive pattern. As a fully automated detector, BugAuditor generates 675 reports in total, with an example shown in Figure 7. Each report includes the explanation, the inferred defensive pattern, and the referenced defensive code for checking.

Specifically, we manually audited the inconsistency reports generated by BugAuditor. Manual validation takes about 10 person-hours, about one minute per report. For each case, we examined the report and inspected the involved operations and function implementations to determine whether the reported inconsistency indicates a real bug. The validation process is straightforward and manageable because each report provides sufficient information, and validation mainly checks whether the pattern is valid and whether it is violated. Recurring patterns can be checked quickly. During this process, we conservatively exclude 253 cases where defensive responsibility cannot be determined within the audited function. In these cases, the security-sensitive operation appears unprotected locally, but the protected variable propagates to the caller, where the handling may be performed. Confirming inconsistency requires whole call-chain reasoning. To avoid misclassification, we conservatively exclude these cases.

Found bugs. Among the remaining 422 reports, we confirm 54 true bugs, including resource leaks, improper resource management, and incorrect checks. We count each buggy function as one bug. These bugs involve 38 distinct defensive patterns, covering 38 security-sensitive operations and 20 defensive operations. Table 12 presents an overview of the detected bugs, and Table 21 presents example bugs and their defensive patterns. Security impacts closely correlate with the type of defensive operation. When defensive operations are used for resource cleanup, missing or incorrect handling leads to resource leaks and, in some cases, information leaks if sensitive data is not cleared. Among the 51 resource leaks, the resources span multiple types, such as memory and reference counters. Incorrect checks typically result in NULL or invalid pointer dereference. We present two case studies in Appendix A. Beyond the confirmed bugs, BugAuditor identifies 104 cases of missing checks. While adding such checks could improve robustness, their absence may be reasonable in practice when the called functions are unlikely to fail. Therefore, we do not count them as bugs. We will further examine these cases and prioritize high-impact ones for patching. We also observe one benign inconsistency arising from an open-coded helper routine. It introduces no security impact but adopting the standard helper would improve maintainability.

Bug lifetime. We measure bug lifetime from the timestamp of the introducing commit to the time of detection. The average lifetime is 5.3 years, with a median of 5.1 years and

Table 12: Overview of newly detected bugs

Bug type	#Bug	CWE
Resource Leak	51	CWE-404 [7]
NULL/Invalid Pointer Dereference	2	CWE-476 [8]
Information Leak	1	CWE-497 [9]
Total	**54**	-

a maximum of 18.4 years. These results indicate that bugs can remain latent in the kernel for many years. These bugs evade detection because existing approaches lack comprehensive oracles for diverse project-specific operations. Starting from defensive semantics, BugAuditor leverages LLM-based reasoning to infer defensive patterns, enabling it to uncover knowledge beyond prior methods and detect bugs they miss.

Bug disclosure. We reported the detected new bugs to developers. We analyzed each bug and submitted corresponding patches to assist with fixes. To date, 20 bugs have been fixed, and two have been assigned CVE identifiers. We are currently working with developers to resolve the remaining cases.

False positives. We identify 264 false reports and analyze their root causes. Among them, 21% come from incorrect pattern reasoning, while 79% come from LLM misjudgments during bug detection. Specifically, during pattern reasoning, security-sensitive operations may be misidentified, leading to incorrect defensive patterns that directly result in false reports. The remaining false reports come from LLM misjudgments during detection and fall into two types: *(1) Unrecognized defensive handling*. Some code employs semantically equivalent protections that are not normalized during pattern reasoning. For example, resources may be released via wrapper macros (e.g., `__free` attribute) or other project-specific mechanisms. *(2) Infeasible bugs*. Certain warnings correspond to execution conditions that cannot be satisfied. For example, although a NULL check appears missing, the variable is guaranteed to be non-NULL under the program constraints. We further discuss mitigation strategies for false positives in §8.

7.3 Ablation Study

To evaluate the contribution of key components in BugAuditor, we conduct a set of ablation studies. Specifically, we examine the effectiveness of defensive snippet filtering, candidate statement collection, and defensive patterns.

Defensive snippet filtering. To evaluate defensive snippet filtering, we randomly sample 500 function-level snippets before and after filtering and manually check their validity. A valid snippet must contain the corresponding security-sensitive behavior within the same function, providing sufficient local context for pattern reasoning. Filtering increases the percentage of valid snippets from 68% to 92%. We further sample 200 snippets removed by the filter and manually inspect them. All of them are correctly filtered out because they lack the security-sensitive behavior within the same function.

Table 13: LLM-only bug detection on the 54 detected bugs

Model	Function-level	File-level
DeepSeek-V3.2 (default)	7/54 (13.0%)	4/54 (7.4%)
GPT-5.4	11/54 (20.4%)	10/54 (18.5%)

Candidate statement collection. BugAuditor collects candidate statements from defensive code and provides them to the LLM for reasoning. To evaluate its effectiveness, we randomly sample 200 defensive snippets and compare the accuracy of pattern reasoning with and without candidate statements. The results show that candidate statements improve pattern accuracy from 89% to 94%. More importantly, this step greatly reduces LLM cost by grouping equivalent defensive snippets before pattern reasoning. As shown in Table 9, this grouping reduces 62 432 snippets to 16 508 groups and infers one pattern for each group, reducing LLM calls by 74%.

Defensive pattern for bug auditing. To isolate the contribution of defensive patterns to bug auditing, we compare BugAuditor with an LLM-only bug detection baseline. The baseline directly asks the LLM to detect bugs from a given code snippet using a standard prompt from prior work [25]. For the 54 bugs detected by BugAuditor, we provide the LLM with either the buggy function or the buggy file, and ask it to determine whether a bug exists and explain its reasoning. We evaluate the baseline with DeepSeek-V3.2 [29], the model used by BugAuditor in our evaluation. We also test a stronger model GPT-5.4 [35]. As shown in Table 13, DeepSeek-V3.2 recalls only 7 bugs with function-level input and 4 bugs with file-level input. GPT-5.4 recalls 11 and 10 bugs, respectively. These results show that BugAuditor detects bugs missed by LLM-only detection, even when the LLM-only detection uses a stronger model. This demonstrates the contribution of defensive patterns for LLM-driven bug auditing.

7.4 Comparison with Existing Methods

BugAuditor aims to mine project-specific knowledge beyond existing work to detect new bugs. Table 15 lists the compared tools and their categories, each representing a state-of-the-art category. These include: (1) APP-Miner [21], which mines frequent API usage patterns and reports deviations. (2) IPPO [30], which detects bugs via one-to-one inconsistency checking. We exclude FICS [11] because it does not scale to large codebases such as the Linux kernel. We also include (3) KNighter [49], a historical bug-based method and (4) RepoAudit [17], which uses LLMs to audit bugs. To compare the bug coverage of BugAuditor and existing methods, we construct a Linux bug benchmark by combining bugs detected by BugAuditor and those reported by the compared tools. For each compared tool, we collect as many reported bugs as possible from public sources. The resulting benchmark contains 396 bugs in total, with the distribution across tools shown in Table 14. Because different tools disclose different num-

Table 14: Distribution of bugs collected from different tools

Category	Tool	#Collected bugs
Frequent pattern-based	APP-Miner [21]	116
One-to-one inconsistency-based	IPPO [30]	158
Historical bug-based	KNighter [49]	64
LLM-driven	RepoAudit [17]	4
Our method	BugAuditor	54
Total		**396**

Table 15: Compared methods' detected bug coverage

Tool	Original share	Recall of full benchmark	Recall of BugAuditor-detected bugs
APP-Miner [21]	116/396 (29.3%)	120/396 (30.3%)	1/54 (1.8%)
IPPO [30]	158/396 (39.9%)	167/396 (42.2%)	4/54 (7.4%)
KNighter [49]	64/396 (16.2%)	69/396 (17.4%)	0/54 (0.0%)
RepoAudit [17]	4/396 (1.0%)	33/396 (8.3%)	5/54 (9.2%)
Documentation	–	–	18/54 (33.3%)
BugAuditor	54/396 (13.6%)	**257/396 (64.8%)**	54/54 (100.0%)

bers of bugs, this benchmark is not designed to be balanced across tools. We use it only to compare their bug coverage and analyze the types of bugs each method can or cannot detect.

For each compared tool, we report its recall on the full benchmark and on the bugs detected by BugAuditor. The results are shown in Table 15. Overall, BugAuditor achieves the highest coverage on the full benchmark. It also covers many bugs detected by prior tools, although it still misses some of them. We also assess documentation availability to evaluate the potential coverage of documentation-based methods. Below, we analyze the results in detail.

Comparison with frequent pattern mining method. Results show that APP-Miner detects only one of the bugs identified by BugAuditor. APP-Miner infers correct API usage from frequent intra-function syntax patterns. As a result, it struggles to obtain low-frequency patterns and may ignore less frequent but valid usage patterns when an API has multiple correct usages. Bugs violating these ignored patterns are therefore missed. In contrast, BugAuditor detects bugs by reasoning about semantic inconsistencies in defensive patterns, without relying on usage frequency or surface-level syntax patterns. Conversely, most bugs detected by APP-Miner are missing return-value checks for risky APIs, which can also be naturally captured by BugAuditor as inconsistent defensive patterns.

Comparison with one-to-one inconsistency-based method. IPPO detects intra-function path inconsistencies. By default, IPPO cannot detect any bugs found by BugAuditor, because it requires predefined security-related operations. When given the security operations related to BugAuditor's bugs, IPPO covers at most four bugs. This is mainly because most buggy functions do not contain similar intra-function paths for inconsistency checking. In contrast, BugAuditor detects codebase-level defensive inconsistencies and is not limited to similar paths within a single function, enabling it to detect bugs missed by IPPO. When given the related defensive operations as seeds, BugAuditor also covers most of IPPO's detected bugs, as they also manifest as inconsistent defensive handling.

Comparison with historical bug-based method. We provide BugAuditor and KNighter with the same patches related to the detected bugs. KNighter fails to detect any of the bugs identified by BugAuditor, mainly because its generated checkers are tied to the specific security-sensitive operations explicitly exposed in the patches. When a patch does not expose certain operations, KNighter cannot generate the corresponding checkers and thus misses related bugs. With six patch seeds, KNighter can cover at most six patterns. In contrast, BugAuditor uses the defensive operations in patches only as seeds, and further analyzes the entire codebase to mine codebase-wide security-sensitive behaviors and defensive handling patterns. As shown in §7.1, BugAuditor discovers thousands of patterns with high accuracy, enabling it to detect bugs missed by KNighter. Conversely, using the same patches, BugAuditor can cover most bugs detected by KNighter, because these patched bugs are also embedded in the codebase and can be inferred as defensive patterns. At the same time, BugAuditor misses some bugs detected by KNighter due to the limited expressiveness of defensive patterns.

Comparison with LLM-driven method. We evaluate RepoAudit with its default configuration on the same modules containing the detected bugs. RepoAudit detects 5 bugs identified by BugAuditor. RepoAudit's bug coverage is limited by predefined sources For example, its memory leak detection focuses on standard allocation APIs. Although RepoAudit performs inter-procedural tracking, its analysis can be incomplete when the relevant data flow spans many functions and complex call chains. In contrast, BugAuditor mines implicit defensive knowledge from the codebase, enabling it to detect bugs beyond predefined sources. Yet, BugAuditor still misses bugs detected by RepoAudit when the corresponding defensive patterns are not covered.

Comparison with documentation-based method. Methods such as Advance [33] and ChatDetector [50] obtain rules from documentation and detect violations of these rules as bugs. However, because they are not open-sourced and do not support Linux kernel analysis, direct comparison is infeasible. As an alternative, we evaluate how many of the rules violated by BugAuditor's detected bugs are explicitly documented. For each detected inconsistency, we use the correctly handled side as the reference and obtain the corresponding defensive rule. In total, the 54 detected bugs violate 38 unique defensive rules. We then manually check whether each rule is explicitly described in official documentation or comments for the relevant functions. The remaining 23 rules are implicit

project-specific knowledge, covering 36 detected bugs. Therefore, even under an ideal setting where all documented rules are perfectly extracted, documentation-based methods could cover at most 33% of the bugs detected by BugAuditor.

False negatives of BugAuditor. BugAuditor fails to detect 139 (35%) of the bugs in the full benchmark, mainly for three reasons: (1) *Limited pattern expressiveness*. BugAuditor mainly models defensive behaviors that should appear along with security-sensitive operations. However, some bugs are caused by behaviors that should not occur, such as use-after-free or double-free after a pointer has been released. BugAuditor cannot express these bugs with the current defensive pattern representation and therefore misses them. (2) *Missing defensive patterns*. BugAuditor relies on mined defensive patterns for bug auditing. If the relevant defensive operation is not covered, BugAuditor cannot mine the corresponding pattern and thus misses bugs that depend on it. This can be mitigated by adding more seeds. (3) *Imprecise LLM reasoning*. Even when a relevant defensive pattern is available, the LLM may still misjudge the code due to imprecise reasoning, causing BugAuditor to miss the bugs.

7.5 Generalizability Study

To evaluate the generalizability of BugAuditor, we conduct experiments on two widely used open-source programs, OpenSSL and FFmpeg. For each program, we collected two common defensive operations as seeds: NULL pointer checks and resource release functions (`OPENSSL_free` for OpenSSL and `av_free` for FFmpeg). Table 16 summarizes the statistics of defensive patterns. For OpenSSL, BugAuditor identified 2922 usages of the selected defensive operations, collected 1688 defensive snippets, and mined 382 defensive patterns. For FFmpeg, BugAuditor identified 3782 usages, collected 2883 defensive snippets, and mined 371 defensive patterns. To assess pattern quality, we randomly sampled 100 mined patterns from each project for manual inspection. The accuracies are 95% for OpenSSL and 88% for FFmpeg. Using these mined patterns, BugAuditor further detected two previously unknown bugs in OpenSSL and four in FFmpeg. All of these bugs have been fixed in the latest version. In the appendix, Table 19 and Table 20 present examples of inferred defensive patterns in OpenSSL and FFmpeg, respectively. For each defensive operation, the tables report the main defensive pattern template and up to 30 corresponding security-sensitive functions. These results show that BugAuditor generalizes beyond the Linux kernel and remains effective on other programs.

7.6 Performance and Cost

To evaluate whether BugAuditor scales to large codebases with reasonable cost, we measure its runtime and LLM token consumption. Table 17 summarizes the costs across the three stages. When applied to the Linux kernel (27.34 MLOC), BugAuditor completes the entire pipeline within 18 hours, consuming 58.5M input tokens and 2.6M output tokens. Based on the official API pricing [10] for DeepSeek-V3.2 ($0.28 per million input tokens and $0.42 per million output tokens), the total cost is $17.47. The code collection stage performs multi-threaded retrieval and analysis at the AST level with low overhead. The most time-consuming stage is pattern reasoning, which requires CFG construction and graph analysis. The remaining stages mainly involve LLM requests, and the overhead largely comes from network calls. In evaluations, BugAuditor audits up to 100 functions per pattern, so the reported auditing cost is measured on this sampled subset. We estimate the full-scale auditing and end-to-end costs by scaling to the full candidate set.

Table 16: Defensive patterns in OpenSSL and FFmpeg

Program	Operation	#Usage	#Snippet	#Pattern
OpenSSL	NULL pointer check	1559	764	258
	`OPENSSL_free`	1363	924	124
	Total	2922	1688	**382 (95% Acc.)***
FFmpeg	NULL pointer check	3038	2330	276
	`av_free`	744	553	95
	Total	3782	2883	**371 (88% Acc.)***

* Accuracy is based on 100 manually checked samples per program.

Table 17: Runtime and cost of BugAuditor

Stage	Runtime	Token (In/Out)	Cost
Defensive code collecting	40 min	–	–
Pattern reasoning	13 h 36 min	17.7M / 1.4M	$5.54
Bug auditing*	3 h 41 min	40.8M / 1.2M	$11.93
Measured total	**17 h 57 min**	**58.5M / 2.6M**	**$17.47**
Estimated full-scale cost			
Bug auditing	11 h 13 min	124.3M / 3.6M	$36.32
Total	25 h 29 min	142.0M / 5.0M	$41.88

*Measured on the sampled auditing subset. Full-scale cost is estimated by scaling the sampled auditing cost to all comparable functions.

8 Discussion

With continued advances in LLMs, we hope BugAuditor can inspire future research on using inconsistent defensive handling as a bug signal for deeper analysis of complex software systems. In particular, we outline the limitations of each stage in BugAuditor and discuss future directions.

Defensive code collecting. We can improve this stage by collecting cross-function defensive code and extending defensive behavior coverage. In particular, a key limitation of BugAuditor is that it collects defensive code only within intra-procedural contexts and excludes cases involving function pointers and indirect calls. Yet, defensive handling in complex systems can span multiple functions and involve indirect calls, causing BugAuditor to miss such complex defensive patterns. To support inter-procedural defensive handling, we

need to design proper strategies for defining valid context boundaries. The collected code should preserve the context needed to reason about security-sensitive behaviors while minimizing irrelevant noise that may mislead later reasoning. In addition, although BugAuditor is evaluated with only a few seeds, it can naturally incorporate more seeds to cover richer defensive behaviors, including the remaining behaviors in Table 3 and behaviors spanning multiple statements. To extend this, BugAuditor requires only a few defensive operations as seeds, which can be easily obtained from various sources, such as patches, human expertise, or LLMs.

Defensive pattern reasoning. In the current design, BugAuditor uses intra-procedural dominator-based analysis to collect relevant statements and help the LLM focus on relevant code. However, when defensive logic involves inter-procedural control flow or complex state machines, this lightweight analysis may fail to capture sufficient critical context. Therefore, future work needs new strategies for selecting proper context in complex scenarios, which also requires stronger code reasoning capabilities from LLMs. In addition, BugAuditor allows flexible adjustment of static analysis during pattern reasoning to handle special cases. Some defensive logic, such as permission checks, relies only on control flow without explicit data-flow dependencies. In such cases, candidate statement collection can operate purely at the control-flow level.

Bug auditing. We can improve this stage by enhancing comparable function collection and reducing false positives. Specifically, BugAuditor collects comparable functions using coarse-grained AST-based matching. This strategy may miss semantically equivalent security-sensitive behaviors implemented with different syntax. To address this limitation, future work can adopt semantic code search to retrieve semantically similar code snippets for auditing. To reduce false positives, we can introduce a verification stage after inconsistency detection. Once recurring false positive patterns are identified, targeted pruning can be performed using LLMs to re-check the involved operations and function code and filter out benign inconsistencies. Adopting stronger LLMs with improved code understanding and reasoning can further reduce false positives, with an increased cost of analysis.

9 Related Work

In this section, we discuss related work on bug detection.

Historical bug-based bug detection. This line of work [19, 20, 22, 23, 27, 44, 47] uses historical bugs as oracles to detect recurring bugs in new code. Early methods achieve scalability through syntactic code clone detection. ReDeBug and VUDDY [20, 23] extract vulnerable code snippets as oracles and match them against target codebases, but they struggle to detect structurally different bug variants. To address this, more recent methods extract fine-grained bug signatures. MVP [47], Movery [44], and VMuD [19] derive signatures from bug-fixing patches. Movery focuses on core vulnerable and patched lines, while VMuD selects critical fixing functions across multiple fixing functions to improve robustness. TRACER [22] extracts signatures from data-flow traces generated by bug detectors. In contrast, BugAuditor leverages inconsistencies in defensive code as detection oracles.

Code deviation-based bug detection. This line of work treats deviations from comparable code in the same codebase (e.g., inconsistencies) as potential bugs [11, 14, 21, 30, 32, 51, 55]. Some studies rely on frequent pattern mining, treating commonly observed usage patterns as correct and flagging deviations as potential bugs. For example, APISan [51] and APPMiner [21] mine frequent API usage patterns, while Crix [32] detects missing security checks through cross-checking similar code paths. Other studies avoid relying on frequent similar instances by detecting one-to-one inconsistencies. FICS [11] compares functionally similar but differently implemented code snippets, while IPPO [30] and NDI [55] identify inconsistencies via differential analysis within or across functions. Different from these methods, BugAuditor detects bugs by identifying semantic inconsistencies in defensive handling.

LLM-driven bug detection. With the rapid advancement of LLMs, an increasing number of studies have explored their use for bug detection [16, 17, 38, 46, 49, 50, 53]. To effectively leverage LLMs, prior work relies on external knowledge sources to guide the analysis. These sources include human-defined heuristics, historical bug patterns, and documentation. For example, RepoAudit [17] audits code by prompting LLMs with predefined locations. KNighter [49] leverages bug patches to guide LLMs in generating static checkers, while BugScope [16] and BugStone [46] directly use bug patches to detect similar bugs with LLMs. Other methods, such as RFCAudit [53] and ChatDetector [50], utilize LLMs to extract security rules from documentation. In contrast, BugAuditor leverages defensive code to detect inconsistent handling.

Classic and specialized bug detection. Classic static analyzers use well-defined checkers to detect bugs [15, 31]. For example, tools such as Clang Static Analyzer [31] use predefined checkers to detect common bugs such as NULL pointer dereferences. While effective for common bug types, they struggle to cover diverse project-specific knowledge and rely on low-level data-flow or state tracking. In contrast, BugAuditor infers defensive patterns that encode project-specific knowledge, avoiding redundant low-level tracking. Beyond general static analyzers, another line of studies designs specialized detectors for certain bug types [13, 34, 39, 45, 52]. They focus on specific security-sensitive operations to detect related bugs. For example, CID [39] analyzes increment–decrement consistency to detect reference leaks, while Pex [52] focuses on permission checks. Goshawk [34] targets memory-related operations. APISpecGen [28] generates three-tuple API specifications. These methods rely on predefined heuristics and target specific modeled bug patterns. In contrast, BugAuditor detects inconsistent defensive handling across diverse security-sensitive operations without specific modeling.

10 Conclusion

In this paper, we propose a new oracle for bug detection and design BugAuditor, which detects bugs by auditing inconsistent defensive handling. By locating defensive code and mining the embedded project-specific knowledge, BugAuditor infers defensive patterns and performs inconsistency-based auditing. This method performs semantic reasoning over defensive behavior and detects previously unknown project-specific bugs beyond the reach of existing methods. Our evaluation on the Linux kernel shows that BugAuditor detects 54 previously unknown bugs. To date, 20 have been confirmed and fixed by maintainers, including two assigned CVE identifiers.

Acknowledgments

We would like to thank the anonymous reviewers for their constructive comments. We are grateful to the Linux kernel maintainers for their valuable feedback and collaboration during bug fixing. This work is partially supported by the JC STEM Lab Initiative. The IIE author is supported in part by NSFC (U24A20236), CAS Project for Young Scientists in Basic Research (Grant No. YSBR-118).

Ethical Considerations

This work analyzes publicly available Linux kernel source code to identify security bugs. It does not involve human subjects, private user data, or testing on production systems. Given the critical role of the Linux kernel in real-world systems, unaddressed bugs may lead to security risks. Our goal is to identify such bugs and facilitate timely remediation to improve system security.

- **Ethical Principles.** We recognize that vulnerability research introduces potential risks if findings are misused. We therefore follow the ethical principles outlined in the Menlo Report, including Beneficence, Respect for Persons, Justice, and Respect for Law and Public Interest. We carefully consider these principles throughout the research process and implement mitigation measures accordingly.

- **Stakeholders.** The primary stakeholders include Linux kernel developers and maintainers, end users of Linux-based systems, the broader security community, and researchers. Throughout the research process, we maintained respect for individuals and avoided using discovered vulnerabilities to interfere with end users. We complied with relevant laws and upheld the public interest. No testing was conducted on live production systems.

- **Potential Impacts.** Our findings help improve software security by enabling developers to identify and fix bugs. Timely patching reduces the risk of exploitation and benefits end users. At the same time, vulnerability discovery has dual-use implications: if mishandled, it could expose systems to risk. We did not publicly disclose any bug details prior to fix availability or coordinated disclosure timelines, protecting users from undue risk.

- **Mitigation Measures.** We responsibly reported detected bugs to the Linux kernel community through established disclosure channels, which helps reduce the remediation burden for developers and maintainers. For each detected bug, we followed Linux community guidelines and submitted a corresponding patch, improving ecosystem-wide security and benefiting the broader community. Each report includes (1) a security patch that fixes the bug and (2) a detailed explanation describing the root cause, potential risks, and the applied fix. To date, 20 bugs have been confirmed and fixed in the latest version. We actively participated in follow-up discussions with maintainers to assist with subsequent fixing.

- **Decision to Publish.** We determined that the benefits of this work outweigh the potential risks. Our study provides actionable insights for improving system security while adhering to responsible disclosure practices. By controlling the release of sensitive information and prioritizing remediation, we ensure that the work advances ecosystem security without exposing users to undue risk.

Open Science

To facilitate future research, we have released the code of BugAuditor at https://zenodo.org/records/20267685 for permanent archival. We also actively maintain BugAuditor at https://github.com/Yuuoniy/BugAuditor.

References

[1] CVE-2022-50199. https://nvd.nist.gov/vuln/detail/CVE-2022-50199, 2022.

[2] CVE-2023-52661. https://nvd.nist.gov/vuln/detail/CVE-2023-52661, 2023.

[3] CVE-2023-52692. https://nvd.nist.gov/vuln/detail/CVE-2023-52692, 2023.

[4] CVE-2024-53078. https://nvd.nist.gov/vuln/detail/CVE-2024-53078, 2024.

[5] CVE-2025-38059. https://nvd.nist.gov/vuln/detail/CVE-2025-38059, 2025.

[6] CVE-2025-39830. https://nvd.nist.gov/vuln/detail/CVE-2025-39830, 2025.

[7] CWE-404: Improper Resource Shutdown or Release. https://cwe.mitre.org/data/definitions/404.html, 2026.

[8] CWE-476: NULL Pointer Dereference. https://cwe.mitre.org/data/definitions/476.html, 2026.

[9] CWE-497: Exposure of Sensitive System Information to an Unauthorized Control Sphere. https://cwe.mitre.org/data/definitions/497.html, 2026.

[10] DeepSeek API Documentation - Models & Pricing. https://api-docs.deepseek.com/quick_start/pricing, 2026.

[11] Mansour Ahmadi, Reza Mirzazade Farkhani, Richard Ryan Williams, and Long Lu. Finding Bugs Using Your Own Code: Detecting Functionally-similar yet Inconsistent Code. In *USENIX Security Symposium*, 2021.

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

[13] Navid Emamdoost. Detecting Kernel Memory Leaks in Specialized Modules with Ownership Reasoning. *Proceedings 2021 Network and Distributed System Security Symposium*, 2021.

[14] Dawson Engler, David Yu Chen, Seth Hallem, Andy Chou, and Benjamin Chelf. Bugs as deviant behavior: A general approach to inferring errors in systems code. *ACM SIGOPS Operating Systems Review*, 35(5):57–72, 2001.

[15] GitHub, Inc. CodeQL. https://codeql.github.com/, 2026.

[16] Jinyao Guo, Chengpeng Wang, Dominic Deluca, Jinjie Liu, Zhuo Zhang, and Xiangyu Zhang. BugScope: Learn to Find Bugs Like Human. *ArXiv*, abs/2507.15671, 2025.

[17] Jinyao Guo, Chengpeng Wang, Xiangzhe Xu, Zian Su, and Xiangyu Zhang. RepoAudit: An Autonomous LLM-Agent for Repository-Level Code Auditing. *ArXiv*, abs/2501.18160, 2025.

[18] Aric A. Hagberg, Daniel A. Schult, Pieter Swart, and JM Hagberg. Exploring network structure, dynamics, and function using networkx. *Proceedings of the Python in Science Conference*, 2008.

[19] Kaifeng Huang, Chenhao Lu, Yiheng Cao, Bihuan Chen, and Xin Peng. VMud: Detecting Recurring Vulnerabilities with Multiple Fixing Functions via Function Selection and Semantic Equivalent Statement Matching. *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, 2024.

[20] Jiyong Jang, Abeer Agrawal, and David Brumley. ReDeBug: Finding Unpatched Code Clones in Entire OS Distributions. In *Proceedings of the 33rd IEEE Symposium on Security and Privacy (S&P)*, 2012.

[21] J. Jiang, J. Wu, X. Ling, T. Luo, S. Qu, and Y. Wu. APP-Miner: Detecting API Misuses via Automatically Mining API Path Patterns. In *2024 IEEE Symposium on Security and Privacy (SP)*, pages 43–43, Los Alamitos, CA, USA, may 2024. IEEE Computer Society.

[22] Wooseok Kang, Byoungho Son, and Kihong Heo. TRACER: Signature-based Static Analysis for Detecting Recurring Vulnerabilities. *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, 2022.

[23] Seulbae Kim, Seunghoon Woo, Heejo Lee, and Hakjoo Oh. VUDDY: A Scalable Approach for Vulnerable Code Clone Discovery. In *Proceedings of the 38th IEEE Symposium on Security and Privacy (S&P)*, 2017.

[24] Fengjie Li, Jiajun Jiang, Dong Chen, and Ying Xiong. LLM-based Vulnerability Detection at Project Scale: An Empirical Study. *ArXiv*, abs/2601.19239, 2026.

[25] Jie Lin and David Mohaisen. From Large to Mammoth: A Comparative Evaluation of Large Language Models in Vulnerability Detection. In *Network and Distributed System Security Symposium*, 2025.

[26] Miaoqian Lin and Hao Chen. SpecAuditor: Generating Audit Specifications for LLM-Driven Bug Detection. In *IEEE Symposium on Security & Privacy*, San Francisco, CA, USA, 2026.

[27] Miaoqian Lin, Kai Chen, and Yang Xiao. Detecting API Post-Handling Bugs Using Code and Description in Patches. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 3709–3726, Anaheim, CA, August 2023. USENIX Association.

[28] Miaoqian Lin, Kai Chen, Yi Yang, and Jinghua Liu. Uncovering the iceberg from the tip: Generating API Specifications for Bug Detection via Specification Propagation Analysis. In *Network and Distributed System Security Symposium*, 2025.

[29] Aixin Liu, Aoxue Mei, Bangcai Lin, Bing Xue, Bingxuan Wang, Bingzheng Xu, Bochao Wu, Bowei Zhang, Chaofan Lin, Chen Dong, et al. Deepseek-v3.2: Pushing the frontier of open large language models. *arXiv preprint arXiv:2512.02556*, 2025.

[30] Dinghao Liu, Qiushi Wu, Shouling Ji, Kangjie Lu, Zhenguang Liu, Jianhai Chen, and Qinming He. Detecting Missed Security Operations Through Differential Checking of Object-based Similar Paths. *In Proceedings of the 27th ACM Conference on Computer and Communications Security (CCS)*, 2021.

[31] LLVM Project. Clang Static Analyzer. https://clang-analyzer.llvm.org/, 2026.

[32] Kangjie Lu, Aditya Pakki, and Qiushi Wu. Detecting Missing-Check Bugs via Semantic- and Context-Aware Criticalness and Constraints Inferences. In *Proceedings of the 28th USENIX Security Symposium (Security)*, 2019.

[33] Tao Lv, Ruishi Li, Yi Yang, Kai Chen, Xiaojing Liao, Xiaofeng Wang, Peiwei Hu, and Luyi Xing. RTFM! Automatic Assumption Discovery and Verification Derivation from Library Document for API Misuse Detection. In *Proceedings of the 27th ACM Conference on Computer and Communications Security (CCS)*, 2020.

[34] Yunlong Lyu, Yi Fang, Yiwei Zhang, Qibin Sun, Siqi Ma, Elisa Bertino, Kangjie Lu, and Juanru Li. Goshawk: Hunting Memory Corruptions via Structure-Aware and Object-Centric Memory Operation Synopsis. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 2096–2113, 2022.

[35] OpenAI. Introducing GPT-5.4. https://openai.com/index/introducing-gpt-5-4/, 2026.

[36] Xiangpu Song, Longjia Pei, Jianliang Wu, Yingpei Zeng, Gaoshuo He, Chaoshun Zuo, Xiaofeng Liu, Qingchuan Zhao, and S Guo. ProtocolGuard: Detecting protocol non-compliance bugs via LLM-guided static analysis and dynamic verification. In *Proceedings of the 33rd ISOC Network and Distributed System Security Symposium (NDSS)*, 2026.

[37] Davide Spadini, Maurício Finavaro Aniche, and Alberto Bacchelli. PyDriller: Python framework for mining software repositories. *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2018.

[38] Yuqiang Sun, Daoyuan Wu, Yue Xue, Han Liu, Haijun Wang, Zhengzi Xu, Xiaofei Xie, and Yang Liu. Gptscan: Detecting logic vulnerabilities in smart contracts by combining gpt with program analysis. In *Proceedings of the IEEE/ACM 46th international conference on software engineering*, pages 1–13, 2024.

[39] Xin Tan, Yuan Zhang, Xiyu Yang, Kangjie Lu, and Min Yang. Detecting Kernel Refcount Bugs with Two-Dimensional Consistency Checking. In *Proceedings of the 30th USENIX Security Symposium (Security)*, 2021.

[40] tree-sitter. Tree-sitter. https://tree-sitter.github.io/tree-sitter/, 2026.

[41] Claire Wang, Ziyang Li, Saikat Dutta, and Mayur Naik. Qlcoder: A query synthesizer for static analysis of security vulnerabilities. *ArXiv*, abs/2511.08462, 2025.

[42] Xiaoke Wang and Lei Zhao. APICAD: Augmenting API Misuse Detection through Specifications from Code and Documents. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 245–256, 2023.

[43] weggli-rs. weggli. https://github.com/weggli-rs/weggli, 2026.

[44] Seunghoon Woo, Hyunji Hong, Eunjin Choi, and Heejo 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, Boston, MA, August 2022. USENIX Association.

[45] Qiushi Wu, Aditya Pakki, Navid Emamdoost, Stephen McCamant, and Kangjie Lu. Understanding and Detecting Disordered Error Handling with Precise Function Pairing. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 2041–2058. USENIX Association, August 2021.

[46] Qiushi Wu, Yue Xiao, Dhilung Kirat, Kevin Eykholt, Jiyong Jang, and Douglas Lee Schales. One Bug, Hundreds Behind: LLMs for Large-Scale Bug Discovery. *ArXiv*, abs/2510.14036, 2025.

[47] Yang Xiao, Bihuan Chen, Chendong Yu, Zhengzi Xu, Zimu Yuan, Feng Li, Binghong Liu, Yang Liu, Wei Huo, Wei Zou, and Wenchang Shi. MVP: Detecting Vulnerabilities using Patch-Enhanced Vulnerability Signatures. In *Proceedings of the 29th USENIX Security Symposium (Security)*, 2020.

[48] Fabian Yamaguchi, Nico Golde, Daniel Arp, and Konrad Rieck. Modeling and discovering vulnerabilities with code property graphs. *2014 IEEE Symposium on Security and Privacy*, pages 590–604, 2014.

[49] Chenyuan Yang, Zijie Zhao, Zichen Xie, Haoyu Li, and Lingming Zhang. KNighter: Transforming Static Analysis with LLM-Synthesized Checkers. *Proceedings of the ACM SIGOPS 31st Symposium on Operating Systems Principles*, 2025.

[50] Yi Yang, Jinghua Liu, Kai Chen, and Miaoqian Lin. The Midas Touch: Triggering the Capability of LLMs for RM-API Misuse Detection. *ArXiv*, abs/2409.09380, 2024.

[51] Insu Yun, Changwoo Min, Xujie Si, Yeongjin Jang, Taesoo Kim, and M. Naik. APISan: Sanitizing API Usages through Semantic Cross-Checking. In *Proceedings of the 25th USENIX Security Symposium (Security)*, 2016.

[52] Tong Zhang, Wenbo Shen, Dongyoon Lee, Changhee Jung, Ahmed M Azab, and Ruowen Wang. Pex: A permission check analysis framework for linux kernel. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1205–1220, 2019.

[53] Mingwei Zheng, Chengpeng Wang, Xuwei Liu, Jinyao Guo, Shiwei Feng, and Xiangyu Zhang. An LLM Agent for Functional Bug Detection in Network Protocols. *ArXiv*, abs/2506.00714, 2025.

[54] Mingwei Zheng, Chengpeng Wang, Xuwei Liu, Jinyao Guo, Shiwei Feng, and Xiangyu Zhang. RFCAudit: AI Agent for Auditing Protocol Implementations Against RFC Specifications. *2025 40th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1221–1233, 2025.

[55] Qingyang Zhou, Qiushi Wu, Dinghao Liu, Shouling Ji, and Kangjie Lu. Non-Distinguishable Inconsistencies as a Deterministic Oracle for Detecting Security Bugs. *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, 2022.

A Case Study

In this section, we present two case studies illustrating bugs detected by BugAuditor. The bugs cause sensitive information disclosure and NULL pointer dereference, respectively. Both bugs involve project-specific operations lacking documented correct specifications and have been assigned CVE identifiers.

① **Inconsistent resource cleanup that may lead to sensitive data disclosure. (CVE-2025-38575)** Figure 9 shows a bug exposed by inconsistent resource cleanup. Specifically, BugAuditor first collects resource release operations similar to `kfree`, including `aead_request_free`. As shown in Figure 9, BugAuditor collects the defensive code snippets, such as the function `sec_aead_soft_crypto`. After analysis, it determines that the security-sensitive behavior in this function is `aead_request_alloc`, and infers a defensive pattern in which the allocated object is released via `aead_request_free`. Based on this pattern, BugAuditor locates the comparable function `ksmbd_crypt_message`, and performs auditing. It detects that this function releases the allocated object using `kfree` instead of `aead_request_free`. These two cleanup operations are not equivalent. In particular, `aead_request_alloc` returns a crypto request object that embeds sensitive cryptographic state. Its corresponding release function, `aead_request_free`, clears sensitive data before releasing the object, whereas `kfree` does not zero out the underlying memory. In `ksmbd_crypt_message`, the allocated request is released using `kfree`. As a result, the request is freed without proper sanitization, leading to potential sensitive data disclosure. Note that this bug is detected via inconsistent defensive handling, regardless of whether the analysis starts from `kfree` or from `aead_request_free`.

```
01 int sec_aead_soft_crypto(struct sec_ctx *ctx,...) {
02
03  subreq = aead_request_alloc(a_ctx->fallback_aead_tfm,...);
04
05  aead_request_set_tfm(subreq, a_ctx->fallback_aead_tfm);
06  aead_request_set_callback(subreq, aead_req->base.flags,
07                ...);
08  ...
09  aead_request_free(subreq);
10  return ret;
11 }
```
inconsistency
```
01 int ksmbd_crypt_message(struct ksmbd_work *work, ...) {
02
03  req = aead_request_alloc(tfm, KSMBD_DEFAULT_GFP);
04
05  aead_request_set_crypt(req, sg, sg, crypt_len, iv);
06  aead_request_set_ad(req, assoc_data_len);
07  ...
08  kfree(req);
09  return rc;
10 }
```

Figure 9: An information leak bug detected by BugAuditor

```
1 int bnxt_alloc_rx_page_pool(struct bnxt *bp,...) {
2   pool = page_pool_create(&pp);
3   if (IS_ERR(pool))
4           return PTR_ERR(pool);
5   rxr->page_pool = pool;
6   ...
7 }
```
inconsistency
```
1 int mlx4_en_create_rx_ring(struct mlx4_en_priv *priv, ...) {
2   ring->pp = page_pool_create(&pp);
3   if (!ring->pp) {
4         err = PTR_ERR(ring->pp);
5         goto err_ring;
6   }
7   ...
8 }
```

Figure 10: An incorrect error check detected by BugAuditor

② **Inconsistent error checking that may lead to invalid pointer dereference. (CVE-2025-39858)** Figure 10 shows a bug exposed by inconsistent error checking. In the Linux kernel, `IS_ERR()` is a built-in mechanism designed to catch error-encoded pointers, where failure states are represented by special pointer values. By locating its usage, BugAuditor locates the function `bnxt_alloc_rx_page_pool` and further identifies `page_pool_create` as the security–sensitive operation and obtains the defensive pattern in which its return value is validated using `IS_ERR()`. Based on this pattern, BugAuditor locates the comparable function, `mlx4_en_create_rx_ring`, which also invokes `page_pool_create` but validates its return value using a NULL check. This inconsistent defensive handling indicates a bug. Further analysis shows that `page_pool_create` returns an error-encoded pointer on failure rather than `NULL`. While such errors are meant to be detected using `IS_ERR`, `mlx4_en_create_rx_ring` instead applies a NULL check. This incorrect error handling may cause an invalid pointer dereference and a potential system crash.

Table 18: Cases of the defensive patterns and up to 30 related security-sensitive functions in the Linux kernel

Defensive operation	Defensive pattern (template)	Security-sensitive functions
`kfree`	***Security-sensitive behaviors***: When `{security-sensitive func}` allocates memory, it creates memory leak risk because failure to free leads to resource exhaustion. ***Defensive behaviors***: The `kfree` is executed in error paths to prevent memory leaks by releasing memory before function exit.	kcalloc; kmalloc; kzalloc; kmemdup; kasprintf; kmalloc_array; kzalloc_node; kstrdup; kstrndup; nvmem_cell_read; ath12k_wmi_tlv_parse_alloc; ath11k_wmi_tlv_parse_alloc; rtw_zmalloc; memdup_user; ocrdma_init_emb_mqe; ath10k_wmi_tlv_parse_alloc; kmalloc_node; kmemdup_nul; qed_dcbnl_get_dcbx; wfx_alloc_hif; xenbus_read; memdup_array_user; pinconf_generic_parse_dt_config; tomoyo_realpath_from_path; ieee802_11_parse_elems; hws_action_create_generic; iwl_acpi_get_object; kobject_get_path; lg_wmab; pmc_data_allocate
`of_node_put`	***Security-sensitive behaviors***: When `{security-sensitive func}` is called, it creates resource leak risk because the reference count of the device node may not be properly managed. ***Defensive behaviors***: The `of_node_put` is executed after the use of node to prevent leaks by decrementing the reference count of the device node.	of_get_child_by_name; of_find_compatible_node; of_find_node_by_path; of_parse_phandle; of_get_parent; of_graph_get_endpoint_by_regs; of_find_matching_node; of_find_node_by_name; of_graph_get_remote_node; of_get_cpu_node; of_get_next_child; of_find_matching_node_and_match; of_find_node_by_type; of_irq_find_parent; of_graph_get_remote_port_parent; of_node_get; of_graph_get_port_by_id; of_find_node_by_phandle; of_get_compatible_child; of_cpu_device_node_get; of_get_available_child_by_name; for_each_matching_node_and_match; of_find_device_by_node; of_get_next_available_child; for_each_matching_node; tegra_xusb_find_port_node; for_each_endpoint_of_node; of_find_next_cache_node; of_graph_get_remote_port; of_nvmem_layout_get_container
`clk_put`	***Security-sensitive behaviors***: When `{security-sensitive func}` is called, it creates a resource leak risk because failing to release the acquired clock reference leads to unbalanced resource management. ***Defensive behaviors***: The `clk_put` is executed in cleanup sections to prevent resource leaks by releasing the clock reference.	of_clk_get_by_name; clk_hw_get_clk; of_clk_get; clk_get; of_clk_get_from_provider; clk_get_sys; clk_get_optional; clk_get_parent
Negative value check	***Security-sensitive behaviors***: When `{security-sensitive func}` is called, it creates error handling risk because the call may return a negative value indicating a failure. ***Defensive behaviors***: The negative-check is executed after the security-sensitive API call to prevent error propagation by returning the error code if ret is negative.	regmap_read; pm_runtime_resume_and_get; i2c_smbus_read_byte_data; platform_get_irq; phy_read; regmap_write; phy_read_mmd; regmap_update_bits; i2c_smbus_write_byte_data; kstrtoul; nla_parse_nested_deprecated; snd_ctl_add; snd_pcm_new; open; platform_get_irq_byname; pcim_enable_device; i2c_transfer; i2c_master_send; fs_parse; regmap_bulk_read; regulator_bulk_enable; i2c_smbus_read_word_data; kstrtobool; rvu_get_blkaddr; btrfs_search_slot; ida_alloc; kstrtouint; kstrtoint; nla_parse_nested; smb_init;
NULL pointer check	***Security-sensitive behaviors***: When `{security-sensitive func}` is called, it creates null pointer dereference risk because the call may return NULL if it fails. ***Defensive behaviors***: The NULL pointer check is executed after the call to prevent null pointer dereference by returning early if it fails.	kmalloc; kvzalloc; kzalloc; alloc_skb; kmalloc_array; kmemdup; kcalloc; bpf_map_lookup_elem; devres_alloc; nlmsg_new; of_device_get_match_data; btrfs_alloc_path; kmem_cache_zalloc; ath10k_wmi_alloc_skb; dma_alloc_coherent; skb_header_pointer; malloc; calloc; platform_get_resource; device_get_match_data; zalloc; ath11k_wmi_alloc_skb; nla_nest_start_noflag; kstrdup; vzalloc; kasprintf; alloc_etherdev; fopen; alloc_page; kzalloc_node;
Error pointer check	***Security-sensitive behaviors***: When `{security-sensitive func}` is called, it creates null pointer dereference risk because the call may return an error pointer if it fails. ***Defensive behaviors***: The error pointer check is executed after the call to `{security-sensitive func}` to prevent null pointer dereference by returning an error code if the return value is an error pointer.	intel_ring_begin; qcom_cc_map; ar; drm_atomic_get_crtc_state; nvkm_gsp_rm_ctrl_get; mlx4_alloc_cmd_mailbox; nct6775_update_device; i915_gem_object_create_internal; tty_alloc_driver; cifs_sb_tlink; dasd_device_from_cdev; intel_context_create; platform_device_register_simple; sdhci_pltfm_init; tape_alloc_request; __filemap_get_folio; clk_get; drm_dev_alloc; drm_property_create_blob; memdup_user; nand_get_sdr_timings; debugfs_create_dir; nxp_c45_find_secy; vfio_alloc_device; __hci_cmd_sync; ext4_journal_start; mock_file; mthca_alloc_mailbox; rpc_run_task; syscon_regmap_lookup_by_phandle;

Table 19: Cases of the defensive patterns and up to 30 related security-sensitive functions in OpenSSL

Defensive operation	Defensive pattern (template)	Security-sensitive functions
NULL pointer check	***Security-sensitive behaviors***: When `{security-sensitive func}` returns a pointer, it creates NULL-pointer-dereference risk because the call may fail and return NULL. ***Defensive behaviors***: The NULL pointer check is executed before the returned pointer is used to prevent NULL-pointer dereference.	OPENSSL_zalloc; OPENSSL_malloc; BIO_next; OPENSSL_strdup; BIO_new; OSSL_PARAM_BLD_new; ossl_bio_new_from_core_bio; ossl_lib_ctx_get_data; BN_CTX_new_ex; EVP_MD_CTX_new; ossl_d2i_PUBKEY_legacy; rand_get_global; ENGINE_new; OPENSSL_realloc; CMS_RecipientInfo_get0_pkey_ctx; BN_new; X509_get0_pubkey; CMS_get0_content; OPENSSL_memdup; EVP_MD_fetch; BIO_get_data; param_push; BN_dup; M_ASN1_new_of; BIO_new_fp; OSSL_FUNC_core_get_libctx; ossl_lib_ctx_get_ex_data_global; DH_new; EC_KEY_get0_group; CMS_dataInit
`OPENSSL_free`	***Security-sensitive behaviors***: When `{security-sensitive func}` allocates memory, it creates memory leak risk because failure to free leads to resource exhaustion. ***Defensive behaviors***: The `OPENSSL_free` is executed in error paths to prevent memory leaks by releasing memory before function exit.	OPENSSL_malloc; OPENSSL_zalloc; app_malloc; OPENSSL_strdup; i2s_ASN1_INTEGER; test_mk_file_path; ASN1_item_i2d; OPENSSL_hexstr2buf; multihexstr2buf; construct_pbkdf2_params; pkey_get_bn_bytes; construct_tls1_prf_params; OPENSSL_strndup; OPENSSL_memdup; DSO_convert_filename; construct_hkdf_params; construct_kbkdf_params; X509_NAME_oneline; EVP_PKEY_get1_encoded_public_key; i2d_ASN1_INTEGER; ossl_dh_key2buf; EC_KEY_key2buf; OPENSSL_buf2hexstr; i2d_ASN1_TYPE; glue2bio; i2d_ECDSA_SIG; ossl_sk_ASN1_UTF8STRING2text; ct_base64_decode; DSO_merge; asn1_d2i_read_bio

Table 20: Cases of the defensive patterns and up to 30 related security-sensitive functions in FFmpeg

Defensive operation	Defensive pattern (template)	Security-sensitive functions
NULL pointer check	***Security-sensitive behaviors***: When `{security-sensitive func}` returns a pointer, it creates NULL-pointer-dereference risk because the operation may fail and return NULL. ***Defensive behaviors***: The NULL pointer check is executed before the returned pointer is used to prevent NULL-pointer dereference.	av_mallocz; avformat_new_stream; av_malloc; av_calloc; av_malloc_array; av_buffer_ref; av_strdup; av_frame_alloc; av_realloc_array; av_fast_realloc; av_pix_fmt_desc_get; ff_get_video_buffer; av_packet_alloc; av_realloc; av_packet_new_side_data; av_asprintf; av_buffer_alloc; av_refstruct_alloc_ext; ff_get_audio_buffer; av_frame_new_side_data; av_fifo_alloc2; av_memdup; av_frame_clone; av_packet_side_data_new; av_refstruct_allocz; allocate_array_elem; ff_framesync_dualinput_get; av_frame_get_side_data; av_realloc_f; av_hwframe_ctx_alloc
`av_free`	***Security-sensitive behaviors***: When `{security-sensitive func}` allocates memory, it creates memory leak risk because failure to free leads to resource exhaustion. ***Defensive behaviors***: The `av_free` is executed in error paths to prevent memory leaks by releasing memory before func- tion exit.	av_malloc; av_mallocz; av_strdup; av_calloc; av_malloc_array; avio_close_dyn_buf; av_asprintf; av_fast_padded_malloc; av_bprint_finalize; ff_nal_parse_units_buf; av_memdup; av_tree_insert; avpriv_ac3_parse_header; decode_str; av_encryption_info_add_side_data; opencl_get_platform_string; opencl_get_device_string; av_append_path_component; av_exif_clone_ifd; av_dovi_alloc; av_dict_get_string; vlc_common_init; av_fast_malloc; ff_alloc_a53_sei; av_escape; get_codecs_sorted; av_dynamic_hdr_plus_alloc; av_des_alloc; av_iamf_param_definition_alloc; ff_nal_unit_extract_rbsp

Table 21: Examples of detected bugs, showing the buggy functions, reference functions, and inferred defensive patterns

Buggy function	Reference function	Security-sensitive behaviors	Defensive behavior
fuse_create_open	fuse_file_open	When `fuse_file_alloc` allocates memory for `ff`, it creates a memory leak risk because failure to free allocated resources leads to resource exhaustion.	`fuse_file_free` is executed in error paths to prevent memory leaks by releasing `ff` before function exit.
ksmbd_crypt_message	libipw_ccmp_decrypt	When `aead_request_alloc` allocates memory for `req`, it creates a memory leak risk because failure to free leads to resource exhaustion.	`aead_request_free` is executed after `crypto_aead_decrypt` to prevent memory leaks by releasing `req` after its use.
pl111_amba_probe	lima_pdev_probe	When `drm_dev_alloc` allocates `ddev`, it creates a memory leak risk because failure to free allocated resources leads to resource exhaustion.	`drm_dev_put` is executed in error paths to prevent memory leaks before function exit.
tps6594_regulator_probe	rpi_exp_gpio_probe	When `of_get_parent` assigns to `fw_node`, it creates a use-after-free risk because `fw_node` may reference a device node that is not properly managed.	`of_node_put` is executed after using `fw_node` to prevent use-after-free bugs by decrementing the reference count of the device node.
sti_gdp_init	v3d_platform_drm_probe	When `dma_alloc_wc` allocates memory for `v3d->mmu_scratch`, it creates memory leak risk because failure to free the DMA-coherent memory leads to resource exhaustion.	The `dma_free_wc` is executed in error cleanup paths to prevent memory leaks by releasing `v3d->mmu_scratch` before function exit.
oaktrail_hdmi_setup	cavium_ptp_get	When `pci_get_device` retrieves a PCI device reference, it creates a memory leak risk because failing to release the reference can lead to unfreed memory.	`pci_dev_put` is executed after using dn to prevent memory leaks by releasing the PCI device reference.
gvp11_probe	ibmvscsi_probe	When `scsi_host_alloc` is called, it creates a memory leak risk because failure to free allocated resources leads to resource exhaustion.	`scsi_host_put` is executed in error paths to prevent memory leaks before function exit.
dwc3_of_simple_probe	devm_clk_bulk_get_all_enabled	When `clk_bulk_get_all` acquires clock resources for `devres->clks`, it creates a resource leak risk because failure to release clocks would break resource management.	`clk_bulk_put_all` is executed in error paths after `clk_bulk_prepare_enable` failure to prevent resource leaks by releasing all acquired clock resources.
mtk_pericfg_init	clk_mt8196_apmixed_probe	When `mtk_alloc_clk_data` allocates memory for `clk_data`, it creates a memory leak risk because failure to free allocated resources leads to resource exhaustion.	`mtk_free_clk_data` is executed in error paths to prevent memory leaks by releasing `clk_data` before function exit.
ras_process_init	jbd2_journal_start_thread	When `kthread_run` creates a new thread for `t`, it creates a NULL/invalid pointer dereference risk because `kthread_run` may return an error pointer if thread creation fails.	An error-pointer check is executed after thread creation to prevent NULL/invalid pointer dereference by checking whether `t` is an error pointer before proceeding.
snow_probe	lpc32xx_clocksource_init	When `of_clk_get_by_name` acquires a reference to `clk`, it creates a reference leak risk because failure to release the reference leads to clock resource exhaustion.	`clk_put` is executed in error paths and cleanup sections to prevent reference leaks by releasing the acquired clock reference before function exit.
mtk_dp_dt_parse	chipone_dsi_host_attach	When `of_graph_get_endpoint_by_regs` is called on `dev->of_node`, it creates a reference leak risk because the returned endpoint must be released to avoid resource exhaustion.	`of_node_put` is executed after obtaining the endpoint to prevent reference leaks by releasing the reference to the device node.