Make a Feint to the East While Attacking in the West: Blinding LLM-Based Code Auditors with Flashboom Attacks

Xiao Li[1], Yue Li[1], Hao Wu[1], Yue Zhang[2], Kaidi Xu[2], Xiuzhen Cheng[3], Sheng Zhong[1], Fengyuan Xu[1*]
[1]*National Key Lab for Novel Software Technology, Nanjing University*
[2]*Drexel University*
[3]*Shandong University*

***Abstract*—LLM-based vulnerability auditors (e.g., GitHub Copilot) represent a significant advancement in automated code analysis, offering precise detection of security vulnerabilities. This paper explores the potential to circumvent LLM-based vulnerability auditors by diverting their focus, decided by the LLM attention mechanism, away from real vulnerable code segments. In these LLM-based vulnerability auditors, the attention mechanism is supposed to focus on potentially vulnerable code sections to identify security issues. Our approach introduces high-attention code snippets (code fragments designed to draw focus) into the codebase under review. By strategically diverting the model's focus away from actual vulnerabilities, this technique effectively "blinds" the LLM, resulting in missed detections. To scale this approach, we present Crazy-Ivan[1], an automated system that identifies and seamlessly integrates high-attention code snippets, shifting focus away from genuine vulnerabilities to decoy functions. Through systematic function-level prioritization and refinement, Crazy-Ivan optimizes the blinding effect, producing the Flashboom that can reduce the model's capacity to detect true security risks. Our evaluation underscores the effectiveness of Flashboom, achieving blinding success rates of up to 96.3% on CodeLlama and 83.05% on Gemma, with notable cross-model transferability and applicability across multiple programming languages. In a case study with GitHub Copilot, Flashboom led the tool to overlook a critical blockchain vulnerability, underscoring the security implications of such attention-diverting attacks and the risks inherent in relying solely on LLM-based automated auditing systems. We have reported our findings to the respective LLM-based code auditor vendors, who have acknowledged the issues and are currently working on fixes.**

1. Introduction

Large Language Models (LLMs) are transforming software security by powering automated tools capable of not only writing and testing code but also detecting complex vulnerabilities with unprecedented precision [55], [53], [13], [38], [18], [42], [9], [31], [35], [40], [41]. Trained on vast datasets that encompass a wide range of security knowledge and diverse code examples—both secure and vulnerable—these models can generalize across coding scenarios to detect subtle patterns and anomalies that may indicate security risks. Once trained, these models are integrated into Integrated Development Environments (IDEs) pipelines, where they offer real-time guidance, proactively suggesting security improvements and catching issues in developers' code [10], [12], [44], [3], [34], [4], [11]. For example, GitHub Copilot [12] is an AI-powered code assistant developed by GitHub in collaboration with OpenAI and Microsoft, which supports code security inspection. As of early 2024, GitHub Copilot had over 1.3 million paid subscribers, and more than 50,000 organizations have integrated GitHub Copilot Business to enhance developer productivity [32].

While LLM-based code vulnerability auditors, such as GitHub Copilot, offer powerful tools for detecting security risks, they also create new attack surfaces. This paper explores one such vulnerability by targeting LLM-based code auditors specifically. Consider the following scenario: an attacker injects malicious code into a public GitHub repository or a Stack Overflow thread. This may involve creating a new open-source project or contributing to an existing one. A developer later downloads and integrates this compromised code—potentially with modifications—into their project, thereby posing a security risk. For instance, existing efforts have demonstrated that malicious contributors can stealthily introduce vulnerabilities, potentially exposing users of the open-source ecosystem to various attacks [49]. To prevent such attacks, developers often take proactive measures and may employ an LLM-based vulnerability auditing tool (e.g., GitHub Copilot) for code auditing. As a result, the vulnerable code will be swiftly identified and reported, preventing the attack from succeeding.

We now pose a key question: *is it possible to bypass an LLM-based code vulnerability auditor?* Our examination of the attention mechanism offers valuable insights into this possibility. As a fundamental component of LLMs, the attention mechanism enables these models to selectively focus on specific parts of the input data when generating predictions or output. In an ideal scenario, an LLM tasked with auditing code will allocate more attention weights to potentially vulnerable sections, increasing the likelihood of detection and thus enabling it to identify security risks effectively. Building

∗. Corresponding author: Fengyuan Xu (fengyuan.xu@nju.edu.cn)

1. Source code, dataset and attack results are available at https://github.com/oxygen-hunter/Flashboom.

on this principle, our approach seeks to exploit the attention mechanism by identifying code snippets that naturally draw high levels of attention weights (potentially surpassing the attention weights given to genuine vulnerabilities within the code under review). By inserting these attention-attracting snippets, we can redirect the focus of the LLM-based auditor away from the actual vulnerabilities, effectively "blinding" the model to security risks. We designate this attack as the Flashboom attack.

While manually created Flashboom attacks can be effective, they lack scalability due to the challenging selection of attention-diverting code snippets, each requiring manual verification. To address the limitations, we introduce Crazy-Ivan, an automated tool designed to optimize and scale the creation of Flashboom attacks. Our key insight is that, since the vulnerability lies within attention mechanisms, the most effective approach is to identify code fragments that naturally capture high levels of attention weights, thereby maximizing the "blinding" effect on the LLM-based code auditors. In particular, Crazy-Ivan operates at the function level. It begins by tokenizing the codebase and calculating function-level attention scores to identify code functions that naturally draw significant attention weights. This allows it to prioritize functions that can effectively divert the LLM's focus. Next, Crazy-Ivan selects the highest-attention functions and refines them, ensuring they are complete and can compile independently (with all their dependencies in the context). Finally, Crazy-Ivan verifies the blinding effectiveness by inserting these attention-attracting functions into target code and measuring attention weights distribution. If the LLM's focus shifts away from actual vulnerabilities, the tool confirms successful blinding.

We implemented Crazy-Ivan and conducted a comprehensive evaluation of its effectiveness and the Flashboom attacks. Key findings include: (i) The success rate of Flashboom (generated by Crazy-Ivan) significantly surpassed traditional methods like obfuscation, achieving blinding success rates of up to 96.3% on CodeLlama and 83.05% on Gemma. (ii) Flashboom also demonstrated strong transferability across models, with Gemma reaching over 87% blinding success when attacks generated on one model were applied to others, highlighting robust cross-model efficacy. (iii) In terms of scalability, Flashboom excelled across programming languages: on the Big-Vul C/C++ dataset, it achieved 100% blinding success on Phi models, while on the CVE-Fixes Python dataset, it maintained high effectiveness with 100% success rates on CodeLlama. (iv) Notably, Flashboom maintained high semantic similarity with the original code, making it less detectable by developers, with a mean similarity score of 0.92 or higher on CodeLlama across datasets.

We conducted an end-to-end evaluation of the Flashboom attack on GitHub Copilot to demonstrate its security implications. Our test used code with a front-running vulnerability [20], a frequent issue in blockchain applications where attackers exploit transaction timing for gain. Before the introduction of Flashboom, Copilot successfully identified seven vulnerabilities in the smart contract, including the primary front-running issue. However, after injecting Flashboom, Copilot failed to detect this critical vulnerability. Instead, it flagged ten unrelated issues, such as outdated Solidity versions and function visibility concerns. This security case study illustrates that attackers could potentially exploit this weakness to introduce undetected vulnerabilities into codebases. As reliance on automated auditing tools grows, this weakness poses threats not just to individual projects but to the broader ecosystem.

Moreover, it is worth noting that Flashboom introduces a groundbreaking approach to exploiting LLM vulnerabilities. Rather than merely identifying a flaw within an LLM-based code auditor, we have uncovered a fundamental weakness in the design of the attention mechanism itself—one that is inherently susceptible to manipulation. Unlike existing attack methods, such as gradient-based adversarial examples [57], [52] or malicious prompt injection [19], [29], our approach specifically targets the core properties of the attention mechanism. This allows for a more sophisticated and powerful form of attack, enabling attention to be selectively diverted to misleading elements during code vulnerability detection or other detection tasks, such as malicious code identification, and even extending beyond code-specific tasks. Our attack holds significant potential as a foundational advancement in the landscape of LLM attack strategies.

Our contributions can be summarized as follows:

- We introduce the Flashboom attack, an attack designed to exploit attention mechanisms in LLM-based code auditors, effectively "blinding" these models to genuine vulnerabilities by diverting attention to high-attention code snippets. Our Flashboom highlights a new attack surface of LLM—the vulnerability of their fundamental attention mechanism.
- We develop Crazy-Ivan, an automated tool that optimizes the Flashboom attack by identifying and inserting high-attention code segments, ensuring scalability and robustness. Crazy-Ivan operates at the function level to prioritize, refine, and insert attention-diverting functions, allowing systematic and efficient deployment of the attack.
- We conducted a thorough evaluation of Flashboom across several dimensions. Results indicate that Flashboom achieves a high blinding success rate, demonstrating strong cross-model transferability and impressive scalability, with perfect success rates across various datasets and programming languages. Our end-to-end evaluation on GitHub Copilot further underscores the critical security implications of this attack.

2. Background

2.1. LLM and Attention Mechanism

LLM. LLM is a type of artificial intelligence system designed to generate human-like text based on the input it receives. These models are built using a deep learning technique (particularly, transformer architecture [47]), which has proven

highly effective for understanding and generating natural language. LLMs are trained on a vast dataset of text collected from books, websites, articles, and other written materials. The training process involves teaching the model to predict the next word in a sentence given the words that came before it. Once trained, LLMs can generate text by being prompted with an initial input or question. The model uses its training to predict a probable sequence of words following the prompt. It can generate answers to questions, complete sentences, or even entire paragraphs.

Attention Mechanism. At the heart of LLMs is the decoder-only transformer architecture [47], which uses mechanisms called attention and self-attention. It was originally introduced to address the limitations of earlier sequence processing models, such as Recurrent Neural Networks (RNNs) and Long Short-Term Memory networks (LSTMs), which struggled with long sequences due to issues like vanishing gradients and difficulty in capturing long-range dependencies. The fundamental idea behind the attention mechanism is to enable models to focus selectively on parts of the input sequence that are relevant to performing a specific task, much like how human attention works when we focus on specific aspects of our environment. In the context of language processing, this means the model can pay "attention" to different words or phrases in a sentence or document when generating each word of its output. The self-attention mechanism calculates the attention weights between every pair of tokens in a prompt. For example, in a prompt with a total of T tokens, given a token (referred to as the query token), the self-attention mechanism computes the influence (i.e., attention scores) of all preceding tokens (referred to as key tokens) on this token. This results in a $T \times T$ matrix to store the attention score between each pair of tokens. Additionally, transformers use a multi-head attention mechanism. For an LLM consisting of L transformer layers, each layer contains H attention heads to calculate attention. This ultimately forms a (T, T, L, H) tensor to store the complete attention scores [26].

2.2. LLM-based Code Auditing

The emergence of LLMs has revolutionized various domains, including natural language processing, content generation, and more recently, software development and security. Prominent models like OpenAI's GPT series [2] have showcased exceptional proficiency in not only understanding and producing human-like text but also in comprehending and generating code. This breakthrough has set the stage for leveraging LLMs as powerful tools in code auditing, particularly in identifying vulnerabilities. Typically, the development of such tools involves training LLMs on extensive datasets composed of both secure and vulnerable code snippets. Through this training, LLMs develop the ability to detect patterns and anomalies commonly associated with security vulnerabilities. These trained models are then integrated into IDEs or incorporated into Continuous Integration/Continuous Deployment (CI/CD) pipelines. Here, they offer real-time feedback and recommendations, enhancing the mitigation of potential security risks and bolstering overall code safety.

TABLE 1: Comparison of LLM-Based Tools for Code Auditing. OSS stands for open-source software.

Tool	Year	Country	Company	IDE	Real-Time	OSS
Tabnine [44]	2020	Israel	Codota	✔	✔	×
DeepCodeAI [3]	2020	Switzerland	Snyk	✔	✔	×
Codex [10]	2021	USA	OpenAI	✔	✔	×
Copilot [12]	2021	USA	GitHub	✔	✔	×
Security Copilot [34]	2023	USA	Microsoft	×	✔	×
CodeRabbit [11]	2023	USA	CodeRabbit	×	✔	×
Q Developer [4]	2024	USA	Amazon	✔	✔	×

As shown in Table 1, we provide a comparative overview of several LLM-based tools that are designed for vulnerability detection or related functionalities in coding environments. It can be observed that most tools offer integration with IDEs. This feature is crucial for developers as it allows them to use these tools seamlessly within their regular coding environments, enhancing productivity and ensuring immediate application of the tool's capabilities. For example, GitHub Copilot is an AI-powered code assistant developed by GitHub in collaboration with OpenAI and Microsoft Azure AI. While GitHub Copilot itself is not a dedicated security tool, it integrates with GitHub and can be used in conjunction with GitHub's native tools like GitHub Actions or GitHub Advanced Security for pre-upload checks. GitHub Copilot supports code security inspection by suggesting improvements based on best practices it has learned.

LLM-based tools are widely-used in real-world applications. GitHub Copilot has over 1.3 million paid users and 50,000+ organizations, while DeepCodeAI serves over 4 million users and 100,000+ organizations; Amazon Q is also adopted by major firms like Deriv and Bolttech. In the research domain, LLMs are increasingly utilized for vulnerability detection, as demonstrated by studies in S&P 2024 [46] and ICSE 2024 [43]. Notably, a USENIX Security 2024 study [50] leveraged GPT to analyze vulnerabilities in obfuscated malicious code.

3. Flashboom Attack

3.1. Threat Model

Scope and Scenario. We consider the following scope and scenario: A developer, aiming to build a project, often relies on online resources such as those from GitHub or Stack Overflow—a common practice in programming. An attacker takes advantage of this by strategically injecting vulnerable code into a public GitHub repository or a Stack Overflow thread, either by creating a new open-source project or contributing to an existing one (the injected code should be syntactically correct and can be compiled to avoid being noticed by the developers). Aware of such threats, the developer uses an LLM-based vulnerability auditing tool (e.g., Github Copilot), integrated within their local IDE, to review the code. Alternatively, they may issue custom prompts to query the LLM directly, asking it to verify the

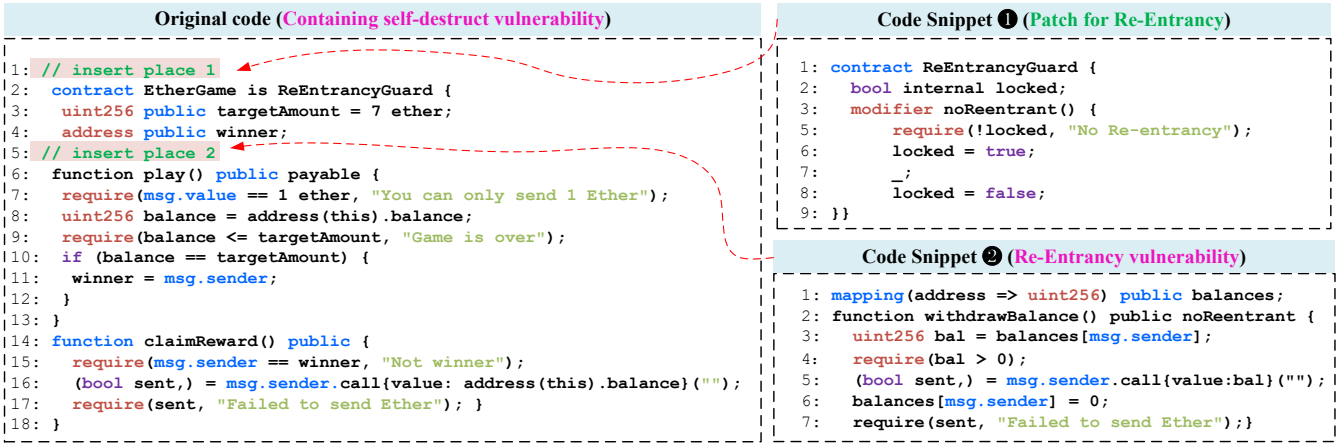

Figure 1: Example of Flashboom attack. The original code `EtherGame` contains a *Self Destruct Vulnerability*. The attack payloads `ReEntrancyGuard` and `withdrawBalance` will be inserted into the comments area of `EtherGame`.

security of specific code segments or explain the behavior of particular functions. The attacker's strategy involves crafting the vulnerable code in a way that exploits the limitations of the LLM-based auditor's detection capabilities. The goal is to bypass the auditing mechanism, ensuring the vulnerable code is not flagged, thereby compromising the developer's project.

Attack Assumptions. We make the following assumptions. First, the attacker's control is strictly limited to the content of the vulnerable code. Specifically, while the attacker may have a general understanding of the potential LLMs that a developer might use, they do not know which exact model will be selected. The attacker cannot influence the specific prompts that the developer crafts to query the LLM. Additionally, the attacker has no control over the internal operations of the LLM-based auditor, including configuration settings, network connectivity, or other operational parameters. Second, it is assumed that developers primarily rely on the LLM-based auditor for vulnerability detection, without using non-LLM-based scanning tools. However, they may employ techniques such as chain-of-thought reasoning (CoT) [48], [40], [31], in-context learning (ICL) [6], [40], or retrieval-augmented generation (RAG) [24], [41] to enhance the auditing process.

3.2. Key Idea and Preliminary Validation

As discussed in §2.2, LLM-based code auditors have gained significant traction today due to their ability to quickly analyze vast amounts of code for vulnerabilities and optimization opportunities. The goal is to bypass the auditing mechanism to ensure that the vulnerable code goes undetected, ultimately compromising the developer's project. In our threat model, the attacker has control solely over the code itself, making code modification the primary approach to evade detection by the LLM-based auditors. However, our preliminary experiments reveal that this approach poses considerable challenges. Straightforward obfuscation techniques (e.g., renaming vulnerable functions with benign identifiers, or embedding vulnerable code within multiple function layers) proved ineffective. Most LLMs we tested (e.g., Mistral [21], MixtralExpert [22], Phi [1], Gemma [45], CodeLlama [39], and GPT-4o [36]) exhibited robust resistance to these methods.

Key Idea. The setbacks encountered in these experiments led us to shift our focus toward the core mechanism of large models: the attention mechanism. The attention mechanism provides a critical lens for understanding, refining, and innovating how LLMs interact with complex, real-world code fragments. When an LLM audits code, it distributes its attention weights across different lines. Ideally, vulnerable sections should receive more attention weights to improve the chances of detecting security risks. However, in the absence of external guidance (e.g., prompt adjustments that direct the LLM to prioritize specific lines), attention weights is distributed unevenly rather than uniformly. This observation led to our core idea: identify certain code snippets (whether vulnerable or non-vulnerable) that tend to attract a disproportionately high level of attention weights, we can use those to "blind" LLM. Specifically, when these attention-attracting snippets coexist with vulnerable segments, they can significantly reduce the attention weights allocated to the vulnerable code, effectively diminishing the LLM's ability to detect vulnerabilities.

Preliminary Validation. Motivated by this idea, we conducted an experiment with a straightforward design. Our experiment begins by selecting a vulnerable code segment, followed by inserting an additional fragment aimed at capturing heightened attention weights from the LLM. This raises the question: what type of code can attract even greater focus than the initial vulnerable code? The answer is another vulnerable code segment.

This strategy is reasonable, as LLM developers may have fine-tuned attention weights for vulnerable code during training, making these segments inherently more detectable. However, if the LLM detects this newly inserted vulnerability, there's a risk that developers may discard the entire fragment, rendering the attack ineffective. To mitigate this, we propose

an additional strategy: embedding a patch to neutralize the newly introduced vulnerability. Ultimately, the attack code comprises three elements: the primary target vulnerability, a secondary vulnerability crafted to attract attention weights, and a patch for this secondary vulnerability, designed to assure the LLM that the code is free of vulnerabilities.

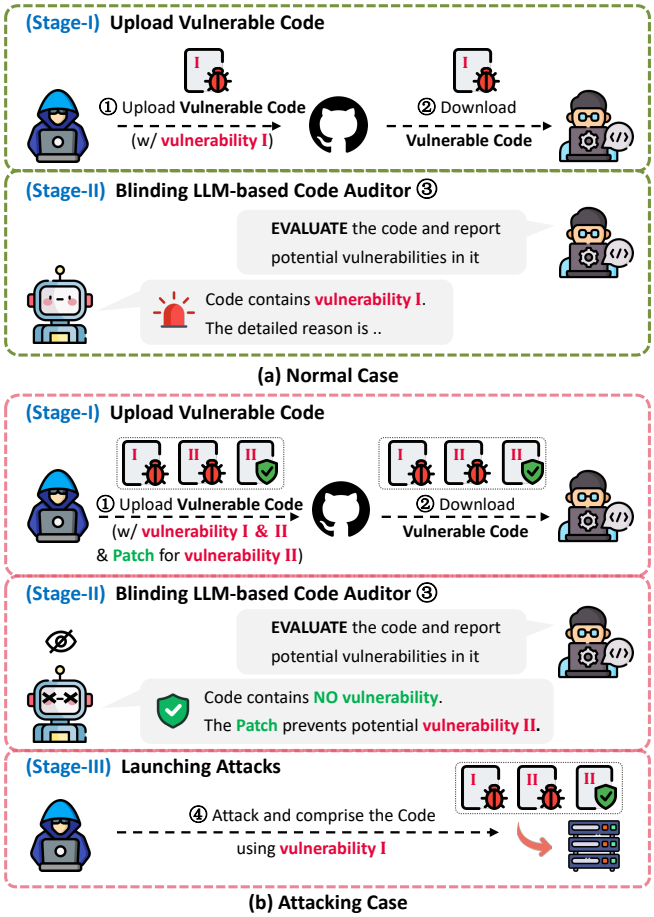

Figure 2: Example of Flashboom on LLM-based auditor

As shown in Figure 1, the provided `Solidity` code contains three key fragments: the original code, which includes the `EtherGame` contract vulnerable to a self-destruct attack [7] (details discussed below); code fragment ❷, which contains the `withdrawBalance` function susceptible to a re-entrancy attack [20] (details also discussed below), as well as a patch ❶ to address the re-entrancy vulnerability.

To elaborate, `EtherGame` operates as a multi-player game where (i) each player can deposit exactly one Ether per turn using the `play` function, (ii) the 7th player to deposit one Ether wins, and (iii) the winner can withdraw the accumulated Ether using the `claimReward` function. However, since the `play` function uses `address(this).balance` directly for balance management instead of a dedicated variable, it is exposed to a self-destruct attack: first, six players each deposit one Ether via `play`, setting `address(EtherGame).balance` to 6 Ether. Next, an attacker deploys a contract that self-destructs, transferring an additional 2 Ether to `EtherGame` and bypassing the game's rules by setting `address(EtherGame).balance` to 8 Ether. This breaks the game, preventing further deposits or a new winner. Fragment ❷ is vulnerable to a re-entrancy attack, which occurs when an external contract recursively calls back into the vulnerable contract before the initial function invocation completes. This allows an attacker to perform multiple actions before state changes, such as balance updates, are finalized, potentially enabling unauthorized fund access or other unintended behaviors.

We utilize the LLMs Mistral and GPT-4o to evaluate the code. In the standard scenario, where the original `EtherGame` contains a self-destruct vulnerability, both LLMs quickly detect it. Given the high similarity between the original and modified code (88.46%), with only an 11.54% difference, it would be reasonable to expect that both LLMs could readily identify the same vulnerabilities in the new version. However, both models failed to detect the self-destruct vulnerability. Specifically, the Mistral model mistakenly flagged a re-entrancy issue in the new code segment, while GPT-4o identified no vulnerabilities at all, even asserting that "*the newly added code element ❷ has a re-entrancy issue, but it is mitigated by code fragment ❶*." The LLMs appear to be "blind" in this context.

3.3. Workflow of Flashboom Attacks

We now outline the attack workflow, illustrated in the accompanying Figure 2, where an attacker seeks to introduce vulnerable code into a project via GitHub. Under normal circumstances (i.e., Figure 2-a), LLM-based auditors would promptly identify and flag any vulnerabilities, preventing their integration into a production environment. However, the attack scenario diverges significantly from this standard process (i.e., Figure 2-b). The steps of the attack are as follows:

(**I**) **Upload Vulnerable Code**. The attacker uploads a code segment with a primary vulnerability (**Step** ①), which developers will later download and integrate into their project (**Step** ②). To strategically divert the LLM's focus, an additional code segment (which may be an additional code segment containing both the vulnerability and its corresponding patch) is embedded. The code is designed to attract the LLM's focus, thereby deflecting scrutiny from the primary vulnerability.

(**II**) **Blinding LLM-based Code Auditor**. When a developer downloads the code, they may integrate it within an existing project using a local IDE that incorporates LLM-based auditing tools. As the developer performs a security assessment, the LLM auditor, distracted by the additional code segment, fails to detect the primary flaw and consequently reports the code as secure (**Step** ③).

(**III**) **Launching Attacks**. With the code now integrated into a project (e.g., a web server, or a blockchain financial game), the undetected primary vulnerability remains

active. This provides the attacker with an entry point to compromise the system (**Step** ④).

4. Crazy-Ivan: A Tool for Making Flashboom

While the manually crafted Flashboom attack proves effective, it has a notable limitation. In the provided code, the newly introduced segment responsible for "*blinding*" the LLM (known as the "Flashboom") was selected randomly, making this approach unscalable. Moreover, each instance (or exploit) requires manual verification to confirm the success of the blinding process. To address these limitations, we need an improved exploit generation approach to assist in automatically producing potential Flashboom. Therefore, in the remainder of this section, we discuss the details of our tool, named Crazy-Ivan. We begin by presenting the key idea of Crazy-Ivan, followed by its concrete design.

4.1. Key Idea

Our key insight is that, since this vulnerability arises from attention mechanisms, our approach targets the code fragment that captures the highest level of attention weights from a set of candidates, thereby maximizing the chances of blinding the LLM. For instance, Figure 3 illustrates the attention weights distribution across three versions of `EtherGame` in §3.2, with darker shades representing higher attention levels. As anticipated, the Flashboom receives significantly higher attention weights compared to the surrounding code. Please note that in §3.2, we intentionally include a vulnerable code fragment to shift the LLM's foucs. However, it is not essential for all Flashboom to involve vulnerable code. Instead, our primary requirement is to select a code fragment that effectively captures attention weights.

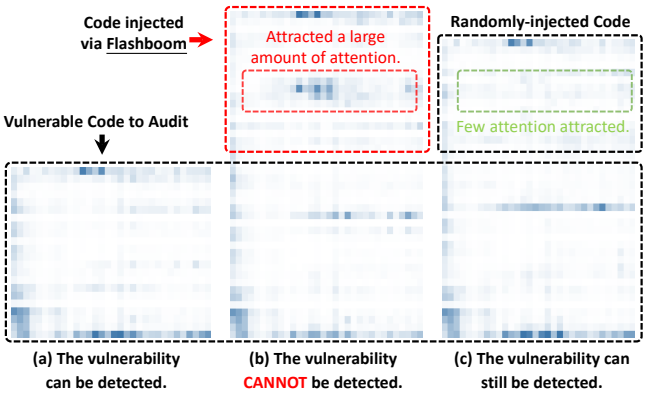

Figure 3: Attention heat maps of three versions of EtherGame: (a) original, (b) inserted an effective blind exploit, and (c) inserted a non-effective blind exploit.

4.2. Concrete Design of Crazy-Ivan

As shown in Figure 4, our Crazy-Ivan operates in three key phases. In the first phase (**Phase I**), we tokenize the codebase, compute attention values (**Step ❶**), and reduce the dimensionality of the attention data on a function-by-function basis (**Step ❷-❹**). This dimensionality reduction not only enhances computational efficiency but is also essential for subsequent steps, as we aim to insert selected high-attention code into the target code (e.g., code with a vulnerability) at the function level. In the second phase (**Phase II**), we identify the functions with the highest attention scores (**Step ❺**) and leverage LLMs to automatically complete and refine these functions (**Step ❻**). This ensures that each function is self-contained and can be compiled independently. The third phase (**Phase III**) involves a rigorous final selection of successfully compiled functions to confirm their effectiveness. At this stage, we conduct a validation test to check whether inserting these Flashboom functions into the vulnerable code (**Step ❼**) effectively reduces the attention scores on the original vulnerable functions (**Step ❽**), thus achieving the intended "blinding" effect.

Phase (I) - Function-Wise Attention Calculation. In the initial phase, our Crazy-Ivan calculates the function-wise attention scores, which measure how much focus the model assigns to specific functions in each file in the codebase[2]. This step helps us identify functions that the LLM is likely to focus on, making them as primary candidates for Flashboom. The most straightforward approach is to use an LLM to generate an attention matrix for all functions in the codebase, allowing us to pinpoint the most significant ones. However, the attention matrix generated by LLMs can become excessively large, especially when handling extensive codebases with hundreds or thousands of functions. For extensive or complex code structures, the size and computational demands of the attention matrix can quickly escalate, making the calculation of attention scores prohibitively expensive. This computational overhead poses a major challenge, as it can lead to substantial delays and require significant processing power.

To address this, we employ dimensionality reduction techniques to streamline computation. Specifically, we introduce a novel concept called function-wise attention scores, which capture the attention weights of entire functions rather than individual tokens. By treating functions as discrete units, we significantly reduce computational overhead. To compute function-wise attention scores, we start with line-level attention scores, aggregating them at the line level and then summing these values for each function. As shown in algorithm 1 in the Appendix A, the methodology proceeds as follows:

(1) Tokenization and Attention Extraction: The codebase is tokenized, and the attention values output, i.e., $\mathbf{A}$(§2.1), is generated for each token across all layers and attention heads of the model (line 4)(**Step ❶**). This attention values output is structured as a high-dimensional tensor with dimensions (T, T, L, H), where

2. Note that this codebase does not necessarily overlap with the target code intended for upload to GitHub. For instance, in §3.2, the code to be uploaded is `EtherGame`, which includes a *Self-Destruct Vulnerability*. The attack payloads `ReEntrancyGuard` and `withdrawBalance` are drawn from a much larger codebase that has no overlap with `EtherGame`.

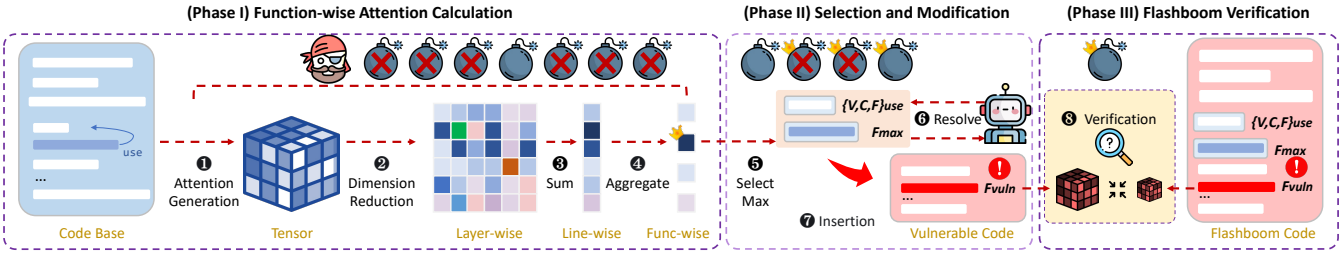

Figure 4: Overview of Crazy-Ivan. The pipeline follows the order of the circled numbers.

T represents the number of tokens in the code, L denotes the number of layers in the model, and H indicates the number of attention heads per layer. The first dimension of $\mathbf{A}$ denotes query tokens, and its second dimension denotes key tokens.

(2) Line-wise Attention Array Calculation: To streamline the analysis, the tensor is reduced, focusing on essential aspects of attention while maintaining interpretability. This reduction includes the following steps: (i) The attention values are summed across all heads, i.e., $\mathbf{A}_{\text{head}}$, resulting in a tensor of shape (T, T, L) (line 5). This aggregation merges the perspectives from different attention heads, producing a unified view of token interactions across layers. (ii) Then we only take the attention score of the last query token of $\mathbf{A}_{\text{head}}$ (line 6). This is due to that in the decoder-only LLMs, the last query token carries semantical information of all preceding tokens. (iii) Attention values are aggregated at the line level, summing the attention scores for all tokens within each line. This produces the matrix $\mathbf{A}_{\text{layer}}$ with dimensions L for each line (line 9). Each entry of $\mathbf{A}_{\text{layer}}$ captures the cumulative attention weight a line receives across all layers, providing a focused view on attention weights distribution at the line level (**Step ❷**). (iv) To obtain an overall line-wise attention score, we sum the attention values across all layers for each line, creating a one-dimensional array with a length equal to the number of lines. Each entry in this array, i.e., a_{line}, represents the total attention weights allocated to a specific line (line 10) (**Step ❸**).

(3) Function-Wise Attention Calculation: After obtaining line-level attention scores, the next step is to aggregate these scores for each function in the code. To that end, (i) we first identify the start and end lines for each function within the codebase, and (ii) for each function f, we calculate the cumulative attention weights by summing the values in a_{line} that correspond to the lines within the function's boundaries (line 12) (**Step ❹**). This function-wise attention score, FuncwiseAtt, represents the total attention weights that the model allocates to each function. Functions with elevated attention scores are prioritized, as they represent critical regions of the code that may be promising candidates for blind exploitation.

Our optimization reduces computational load while preserving the effectiveness of identifying critical functions.

TABLE 2: Function-wise attention scores for key functions in the `BaseToken` file

Function Name	# Line of Code	FuncwiseAtt
`BaseToken._transfer`	9	18.38
`BaseToken.transfer`	3	13.80
`BaseToken.transferFrom`	5	13.92
`BaseToken.approve`	5	17.84
`CustomToken.CustomToken`	9	133.37

Now we provide an example to illustrate what the function-wise attention score looks like. When calculating attention for Solidity code using the Mistral, which comprises 32 multi-heads across 32 layers, the total sum of all values reaches $32 \times 32 = 1024$. However, the cumulative function-wise attention scores across all functions are typically less than 1024, as a portion of attention weights are allocated to non-functional elements within the code (e.g., `#pragma solidity ^0.4.19`, `import "../Own.sol"`, and isolated curly braces marking the end of contracts). In a given source file, the highest function-wise attention score for individual functions can range from 100 to 200, while functions with lower significance may have scores in the single digits or low tens. For example, in this Solidity source file (https://pastebin.com/swG2utx4), the function-wise attention scores for several functions are summarized in Table 2, with `CustomToken.CustomToken` receiving the highest attention score.

Phase (II) - Functions Selection and Modification. Once the function-wise attention scores are computed, we proceed by selecting the functions with the highest attention scores, denoted as F_{max}, from each file in the codebase (**Step ❺**). These high-attention functions serve as initial candidates for Flashboom, which are portions of code that we intend to transplant into a target program (e.g., the code contains the vulnerable code). However, simply transferring this function without further modification may not work as expected, leading to our next challenge. For the Flashboom to integrate seamlessly and compile successfully when inserted into the target code, it must be self-contained. This means it should include all the elements it relies on, such as dependent variables and auxiliary functions, to avoid compilation errors.

To address this, we leverage GPT-4o to automate the dependency resolution process, ensuring that the Flashboom is complete and self-sufficient. This involves using GPT-4o's

language and code generation capabilities to analyze F_{max} and identify all external variables and functions it relies on (**Step ❻**). The prompt template we used is reported in the Appendix B. Specifically, we identified three categories of rules to manage these dependencies effectively:

- **Rules for F_{max} and Its Dependencies**: Beyond the function with the highest attention score F_{max}, we also identify and gather member variables V_{use} and functions F_{use} within the same contract/class that F_{max} depends on. Additionally, we collect any external functions or classes outside the immediate contract/class C_{use} that F_{max} relies upon. These member variables, functions, and classes will also be incorporated into the target code to ensure completeness and functionality.
- **Rules for Vulnerable Code and Its Dependencies.** The vulnerable code needs to be introduced into the target code. In addition to the vulnerable code itself F_{vuln}, we also need to gather member variables $V_{\text{vuln\_use}}$ and functions $F_{\text{vuln\_use}}$, within the same contract/class, as well as any external classes $C_{\text{vuln\_use}}$ outside the immediate contract/class that F_{vuln} depends on.
- **Rules for Insertion Location.** $V_{\text{vuln\_use}}$ and $F_{\text{vuln\_use}}$ should be placed before F_{vuln}. Similarly, V_{use} and F_{use} should be positioned before F_{max}. This ordering ensures that the functions can be properly invoked (**Step ❼**).

It is essential that all code inserted as part of the Flashboom maintains unique naming conventions to avoid conflicts with existing functions in the target code. For example, if the Flashboom includes a function named `processData` and the target code already has a function with this name, it should be renamed in the Flashboom (e.g., to `handleData`) to prevent naming conflicts.

Phase (III) - Flashboom Verification. Simply inserting code may result in incomplete or ineffective blinding. Therefore, rigorous validation of the effectiveness of the selected functions is essential. One of the most straightforward ways to assess an Flashboom attack's success is by measuring the performance of vulnerability detection tools before and after inserting the Flashboom functions. A significant drop in detection accuracy suggests that the Flashboom attack has been effective. However, while a decrease in accuracy indicates an impact on the detection tool's performance, it does not clearly explain why this change occurred. The drop may result from factors unrelated to the Flashboom attack's effectiveness, such as shifts in the tool's sensitivity or inherent biases. Consequently, this does not guarantee the robustness of the Flashboom attack. Additionally, various vulnerability detection tools may apply different metrics or thresholds for evaluating accuracy, making it challenging to reproduce consistent results across tools.

To achieve this, we insert the selected Flashboom functions into the target code and measure the resulting attention weights distribution. If the Flashboom functions are effective, they should divert a significant portion of the model's focus away from the original vulnerabilities, spreading attention weights across the newly added code (**Step ❽**). This focus shift would suggest that the Flashboom functions are successfully "camouflaging" the vulnerable areas, making them less prominent within the model's attention landscape. This systematic verification ensures that the inserted functions not only coexist with the vulnerable code but also actively reduce detectable vulnerability-related signals within the model's attention, enabling the attack to be applied across different LLMs.

5. Evaluation

In this section, we conducted comprehensive experiments to evaluate the performance of Crazy-Ivan and Flashboom attacks. Specifically, we aim to address the following five research questions:

- **RQ1:** How does the success rate of Flashboom (generated by Crazy-Ivan) on different models compare to traditional methods such as obfuscation and renaming? **(Effectiveness)**
- **RQ2:** What is the success rate of Flashboom generated by Crazy-Ivan on one model when transferred to other models? **(Transferability)**
- **RQ3:** How effective is Crazy-Ivan across different programming languages? **(Scalability)**
- **RQ4:** How similar is the target code before and after inserting Flashboom to ensure that developers do not notice the ongoing attack? **(Stealthiness)**
- **RQ5:** Does the decrease in attention scores significantly influence the results (i.e., lower the focus of the original vulnerable function) throughout the experiment? **(Attribution Analysis)**

5.1. Experiment Setup

Datasets: Our dataset consists of two main parts: a sample dataset and a vulnerability benchmark dataset. We generate Flashboom from the sample dataset and apply the Flashboom on the vulnerability benchmark dataset. The sample dataset includes Solidity smart contracts (*Messiq*), C++ and Python projects (*Leetcode*). The vulnerability benchmark dataset includes Solidity smart contracts (*Smart-bugs*), C++ projects (*Big-Vul*), and Python projects (*CVE-Fixes*). To avoid exceeding the LLM's inference ability, we filter out the code files whose number of context tokens exceeds 4096. In total 6 datasets are created, as shown in Table 3:

- **Sample dataset.** The sample dataset serves as the codebase from which our Crazy-Ivan processes and generates the Flashboom. Specifically, two datasets are used as the sample dataset: The *Messiq* dataset [30] consists of over 40,000 smart contracts collected from real-world Ethereum applications, while the *Leetcode* dataset [23] includes over 3,000 C++ and Python solutions to various LeetCode problems.
- **Vulnerability Benchmark.** We apply the Flashboom to the vulnerability benchmark dataset. Since the benchmark dataset originally contains known vulnerabilities, the ideal goal is to trick the LLM into perceiving it as a dataset without vulnerabilities. The vulnerability

benchmark dataset comprises three curated datasets: *Smart-bugs* [15] is a dataset of 140 Solidity smart contracts with vulnerabilities tagged according to the DASP taxonomy [20], covering nine common vulnerability types, including Reentrancy, Access Control, Denial of Services, and Arithmetic issues. *Big-Vul* [16] is a C/C++ vulnerability dataset from 348 GitHub projects, containing 3,754 vulnerabilities along with code changes and CVE summaries. *CVE-Fixes* [5] is a multi-language dataset sourced from the U.S. National Vulnerability Database, including 5,365 commits and 50,322 methods with CVE/CWE summaries and code changes.

TABLE 3: All datasets used in our evaluation. Avg LOC stands for the average line of code. Avg Func stands for the average number of functions.

	Sample datasets			**Vulnerability benchmark**		
	Messiq	*Leetcode-cpp*	*Leetcode-py*	*Smart-bugs*	*Big-Vul*	*CVE-Fixes*
Count	10,000	3,000	3,000	128	100	100
Avg LOC	118	40	33	35	57	26
Avg Func	13	2	3	4	1	1

LLM-based Code Auditors. We gather six LLMs specialized in code understanding to investigate the weaknesses of LLM-based auditing tools, as shown in Table 4. We test those LLM-based auditors using a combination of methods, including chain-of-thought (CoT) reasoning and in-context learning prompts. First, we provide the LLM with a step-by-step guide for detecting vulnerabilities in the specific programming language, followed by a code sample identical to the original target code (a code fragment from the vulnerability benchmark without the Flashboom inserted). This sample includes detailed information, such as vulnerability type, CVE/CWE ID, and an explanation of the vulnerability. After this preparation, we submit the code to the LLM and request an audit report. Later, once the Flashboom is generated by Crazy-Ivan, we insert it into the target code and resubmit it to the LLM-based auditors. This approach simulates a worst-case scenario for attackers, as the auditor has access to the original code. If our Flashboom succeeds in this rigorous setting, where the LLM auditor has every advantage to detect vulnerabilities, it strongly demonstrates the module's robustness under typical, less ideal conditions.

TABLE 4: 6 victim models, HumanEval *Pass@1* is reported by PapersWithCode [37] or model source papers.

Model	**Parameters**	**Context Length**	**HumanEval**
GPT-4o [36]	>1T	128k	90.2
Phi3 [1]	14B	4k	62.2
MixtralExpert [22]	46.7B	32k	40.2
CodeLlama [39]	7B	16k	38.4
Mistral [21]	7B	32k	30.5
Gemma [45]	2.6B	8k	17.7

Evaluation Metrics. The metric we use to evaluate the effectiveness of a Flashboom is the *Blinding Success Rate* (BSR). For a given vulnerable code segment, a successful blinding occurs when the LLM auditor successfully detects the vulnerability before applying the Flashboom but fails to detect it afterward. Specifically, we rate the LLM's responses in the vulnerability detection task on a scale from 1 to 4: Score 1: No vulnerabilities detected. Score 2: Multiple vulnerabilities are detected, but they do not include the ground truth. Score 3: Only one vulnerability is detected, and it is the ground truth. Score 4: Multiple vulnerabilities are detected, including the ground truth as well as other vulnerabilities.

We define three levels of criterion for determining the success and failure of LLM auditors.

- **Existence Level:** The criterion for success is that, the LLM's audit report correctly answers *yes* on the presence of vulnerability (score 2, 3, or 4). The vulnerability type is allowed to be incorrectly classified. For failure, no vulnerability could be detected (score 1). The existence level focuses on whether the LLM under attack generates false positives.
- **Type Level:** The criterion for success is that, the LLM's audit report not only identifies the presence of a vulnerability but also classifies the vulnerability type correctly, aligning with the ground truth (score 3 or 4). The failure criterion is that the LLM finds zero or several vulnerabilities, excluding the true positive (score 1 or 2). The type level focuses on whether the LLM under attack ignores the true positive.
- **Strict Level:** The criterion for success is that, the LLM's audit report classifies the ground truth vulnerability with its type (score 3 or 4). The failure criterion is that the LLM finds no vulnerability (score 1). The strict level focuses on whether the LLM under attack reports nothing suspicious.

We define three specific metrics *BSR@exist*, *BSR@type* and *BSR@strict* to calculate the blinding success rate under the three criterion levels. ***BSR@exist***: The ratio of instances where a score initially rated as 2, 3, or 4 is downgraded to 1 after Flashboom attack, over the total instances with an initial score of 2, 3, or 4. ***BSR@type***: The ratio of instances where a score initially rated as 3 or 4 is reduced to 1 or 2 after Flashboom attack, over the total instances with an initial score of 3 or 4. ***BSR@strict***: The ratio of instances where a score initially rated as 3 or 4 is lowered to 1 after Flashboom attack, over the total instances with an initial score of 3 or 4. Each metric captures the proportion of successful blinding cases where the LLM auditor detects the vulnerability before applying the Flashboom but fails to detect it afterward, under different levels of criterion.

Environment. The experiments are conducted on a high-performance server with 8 NVIDIA GeForce RTX 4090 24 GB PCIe GPUs with CUDA version 12.4 and an Intel Xeon Gold 6426Y CPU.

5.2. Experiment Results

Effectiveness (RQ1): In this experiment, we evaluated the effectiveness of the top-ranked Flashboom identified by Crazy-Ivan by measuring its success rates in obscuring the vulnerability detection capabilities of LLM-based auditors on the *Smart-bugs* dataset. We conducted the experiment using five distinct LLM-based auditors—Mistral, MixtralExpert, Gemma, CodeLlama, and Phi—each tested in separate trials. To establish a comparative baseline, we applied two methods: (i) randomly select functions inserted into the code to observe whether random insertions could obscure LLM detection success rate, and (ii) renaming vulnerable functions and variables to "`Benign_function_xx`" and "`Benign_var_xx`" where "`xx`" represents the original name of the respective functions or variables. BSR were assessed through three metrics: *BSR@exist*, indicating the general presence of blinding effects, *BSR@type*, which measured success by specific vulnerability types, and *BSR@strict*, considering blinding successful only if no vulnerabilities are reported. Before our attack, four out of five LLMs (all except CodeLlama) detected vulnerabilities in over 118 of the 128 cases in the *Smart-bugs* dataset.

TABLE 5: Attack performance across different methods and victim auditors. Dataset: *Smart-bugs*. **RA** stands for Random Adding. **BR** stands for Benign Renaming.

Method	*Smart-bugs* (% #)				
	Mistral	MixtralExpert	Gemma	CodeLlama	Phi
BSR@exist					
Ours	**0.4531** (58)	**0.2734** (35)	**0.8305** (98)	**0.9630** (26)	**0.1732** (22)
RA	0.0391 (5)	0.0312 (4)	0.2373 (28)	0.3333 (9)	0.0000 (0)
BR	0.0000 (0)	0.0547 (7)	0.0169 (2)	0.7037 (19)	0.0236 (3)
BSR@type					
Ours	**0.5840** (73)	**0.4215** (51)	**0.8273** (91)	**0.9583** (23)	**0.3607** (44)
RA	0.0560 (7)	0.0909 (11)	0.3636 (40)	0.5417 (13)	0.0082 (1)
BR	0.0080 (1)	0.0909 (11)	0.0273 (3)	0.7500 (18)	0.0246 (3)
BSR@strict					
Ours	**0.4480** (56)	**0.2645** (32)	**0.8182** (90)	**0.9583** (23)	**0.1803** (22)
RA	0.0400 (5)	0.0331 (4)	0.2364 (26)	0.2917 (7)	0.0000 (0)
BR	0.0000 (0)	0.0578 (7)	0.0182 (2)	0.7083 (17)	0.0246 (3)

As illustrated in Table 5, we show the Flashboom performance comparisons across different LLM-based auditors. Each column reflects the success rate of the top Flashboom compared to baseline methods on the Smart-bugs dataset. Under all the metrics, Flashboom outperformed other baseline methods across all five models, demonstrating consistent efficacy. Specifically: with *BSR@exist* as the metric, Flashboom improved the blinding rate from 0 to 17.32% on the Phi model compared to random. Against the benign renaming method, Flashboom achieved a blinding rate increase from 0 to 45.31% on Mistral, with improvements across other models ranging from 1.36x (CodeLlama) to 49.14x (Gemma). When *BSR@type* was used as the standard, Flashboom consistently outperformed random, with improvements from 1.77x (CodeLlama) up to 43.99x (Phi). In comparison to benign renaming method, Flashboom 's success rates ranged from a 1.27x increase (CodeLlama) to a maximum of 73x (Mistral). While evaluated with the rigorous BSR@strict metric, Flashboom still surpassed the baseline methods. Compared to the random method, the success rates improved from 0 to 18.03% (Phi). Against the benign renaming method, it increased from 0 to 44.80% on the Mistral model and achieved a 1.35x (CodeLlama) to 44.9x (Gemma) improvement on other models.

Transferability (RQ2): In our second experiment, we examined the transferability of top-performing Flashboom from one LLM-based auditor to others. Specifically, using Crazy-Ivan, we selected the top three Flashboom for each LLM auditor based on their blinding success rates (*BSR@exist*, *BSR@type* and *BSR@strict*). These Flashboom were then tested for their effectiveness in performing blinding attacks across other LLM-based auditors, using the *Smart-bugs* dataset as the evaluation benchmark. Table 6 summarize the success rates of the top three Flashboom from each LLM auditor when applied to other LLM-based auditors. Our findings indicate strong transferability of Flashboom across models. Specifically, for the *BSR@exist* metric, Gemma's top one Flashboom demonstrated the highest transferability, achieving 87.5% blinding success rate on Mistral and 18.11% on Phi. Similarly, under the *BSR@type* metric, Gemma's top one Flashboom maintained a high success rate, exceeding 90% on Mistral and 40% on Phi. Even with the most rigorous *BSR@strict* metric, Gemma's top one Flashboom still achieved a 88% success rate on Mistral and 19.67% on Phi. While top Flashboom modules of some auditors, such as CodeLlama, exhibited relatively low transferability-suggesting that their BSR may rely on model-specific characteristics-the overall BSR of CodeLlama's top one dropped from over 90% (on CodeLlama itself) to less than 10% (across other models such as Mistral, MixtralExpert and Phi). It is worth noting that the second strongest Flashboom module on the Mistral model significantly outperforms all the top three Flashboom modules on other models, in terms of its blinding rate against GPT-4o, a black-box model, achieving 54.76%. These results highlight that certain modules exhibit robust cross-model transferability, proving effective on both open-source and closed-source models.

Scalability (RQ3): As of now, our experiment focuses solely on Solidity language. To confirm the scalability on different language, we plan to test effectiveness on additional dataset with other languages (C, C++, and Python) using the same experimental setup as in RQ1. The selected LLM-based auditors for this extended evaluation are Mistral, CodeLlama, and Phi. Similarly, *BSR@exist*, *BSR@type* and *BSR@strict* are selected as the metrics. Flashboom demonstrated high attack success rates, especially on the *Big-Vul* dataset, where it achieved perfect scores (*BSR@exist*, *BSR@type* and *BSR@strict*) with the Phi model across all metrics. In contrast, *RA* and *BR* showed lower overall performance on the Phi model, with success rates falling below 50% in several cases. Interestingly, while our Flashboom achieved the highest BSR overall, we observed that for Python, traditional methods

TABLE 6: The performance of BSR top3 Flashboom of one model on other victim models. Each column represents the performance of a Flashboom of one model on other models. The highest values in each row have been bold and hightlighted with blue cell. Dataset: *Smart-bugs*.

Models	*Smart-bugs*														
	Mistral			MixtralExpert			Gemma			CodeLlama			Phi		
	1st	2nd	3rd	1st	2nd	3rd	1st	2nd	3rd	1st	2nd	3rd	1st	2nd	3rd
BSR@exist															
Mistral	0.4531	0.3203	0.1562	0.4531	0.0781	0.3203	**0.875**	0.6406	0.3672	0.0156	0.4453	0.1016	0.5234	0.4766	0.2031
MixtralExpert	0.2734	0.1797	0.1328	0.2734	0.2109	0.1797	0.2578	0.4141	0.2734	0.0859	0.1875	0.125	0.3594	**0.4219**	0.1094
Gemma	0.3475	0	0.1186	0.3475	0.6102	0	**0.8305**	0.6017	0.3898	0.2542	0.5763	0.2712	0.4492	0.1271	0.2458
CodeLlama	0.6667	0.1481	0.5556	0.6667	0.5926	0.1481	0.4074	0.037	0.2222	**0.963**	0.9259	0.9259	0.6667	0.1111	0.1852
Phi	0.0315	0	0	0.0315	0	0	**0.1811**	0.1732	0.0472	0	0.0315	0.0236	0.1732	0.1181	0.0787
GPT-4o	0	**0.5476**	0	0	0.0397	**0.5476**	0.3016	0.0317	0	0.0079	0	0.0317	0.0159	0.0079	0.1111
BSR@type															
Mistral	0.584	0.496	0.24	0.584	0.08	0.136	**0.912**	0.704	0.264	0.576	0.184	0.184	0.56	0.6	0.32
MixtralExpert	0.4215	0.2893	0.0579	0.4215	0.314	0.2893	0.2975	0.5124	0.3306	0.4876	0.1653	0.1653	0.5124	0.5124	**0.6198**
Gemma	0.5182	0.3091	0.3455	0.5182	0.6455	0.2636	**0.8273**	0.6909	0.6	0.5909	0.4182	0.4091	0.4545	0.4818	0.4182
CodeLlama	0.7083	0.7083	0.6667	0.7083	0.9167	0.5833	0.7083	0.25	0.4167	**0.9583**	**0.9583**	**0.9583**	0.0833	0.625	0
Phi	0.0328	0	0	0.0328	0	0.0328	**0.459**	0.1803	0.1475	0.0246	0.0164	0.0164	0.3607	0.2213	0.1475
GPT-4o	0.0238	**0.5476**	0.0238	0	0.0397	0.1825	0.2937	0.0159	0.2381	0	0.0397	0.0079	0.0238	0.0397	0
BSR@strict															
Mistral	0.448	0.32	0.16	0.448	0.064	0.32	**0.88**	0.68	0.312	0.088	0.112	0.016	0.52	0.472	0.192
MixtralExpert	0.2645	0.1818	0.0744	0.2645	0.2149	0.1818	0.2562	0.4132	0.2893	0.124	0.1157	0.0661	0.3388	**0.4215**	0.0992
Gemma	0.3273	0	0.0364	0.3273	0.6091	0	**0.8182**	0.5818	0.3818	0.2545	0.2545	0.2455	0.4364	0.1364	0.2364
CodeLlama	0.6667	0.0833	0.7083	0.6667	0.5833	0.0833	0.375	0.0417	0.5833	**0.9583**	**0.9583**	**0.9583**	0.625	0.0833	0.0833
Phi	0.0246	0	0.0164	0.0246	0	0	**0.1967**	0.1803	0.0328	0.0164	0.0164	0	0.1803	0.123	0.082
GPT-4o	0.0238	**0.5476**	0.0079	0.0238	0.0397	**0.5476**	0.2857	0.0159	0	0.0317	0.0079	0.0079	0.0397	0.0238	0.1111

TABLE 7: Attack performance across different programming languages and victim models. Dataset: *Big-Vul*, *CVE-Fixes*.

Method	*Big-Vul* (C/C++)			*CVE-Fixes* (Python)		
	Mistral	CodeLlama	Phi	Mistral	CodeLlama	Phi
BSR@exist						
Ours	**0.7816**	**0.8356**	**1**	**0.9375**	0.9756	**0.7742**
RA	0.4598	0.0822	0.4156	0.625	0.878	0.4839
BR	0.7701	0.7945	0.4935	0.4375	**0.9878**	0.5161
BSR@type						
Ours	**0.9778**	**0.8491**	**1**	**0.9286**	**1**	**0.7931**
RA	0.6222	0.2642	0.4394	0.6905	0.8974	0.4483
BR	0.7778	0.7547	0.4848	0.4524	**1**	0.5172
BSR@strict						
Ours	**0.7778**	**0.7925**	**1.0000**	**0.9286**	**0.9872**	**0.7931**
RA	0.3333	0.0566	0.3485	0.6429	0.8846	0.4483
BR	0.7111	0.7547	0.4394	0.3810	**0.9872**	0.5172

like benign renaming also reached relatively high BSR scores. This suggests potential inherent weaknesses in LLM-based auditors when evaluating those python datasets.

Stealthiness (RQ4): To assess the stealthiness of Flashboom, firstly we measured the semantic similarity between the target code before and after the insertion of Flashboom across multiple datasets. The top-performing Flashboom (top1) on each of the victim open-source models was used to perform insertions on code from different datasets, including *Smart-bugs* (Solidity), *Big-Vul* (C/C++), and *CVE-Fixes* (Python). We used a frozen language model (FLM), specifically Alibaba-NLP/gte-large-en-v1.5 [54], [27], to compute

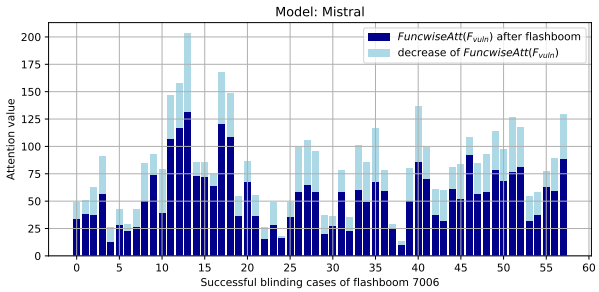

Figure 5: Decrease of FuncwiseAtt(F_{vuln}) after Flashboom 7006 applied on the successful blinding cases. Model: Mistral. Dataset: *Smart-bugs*.

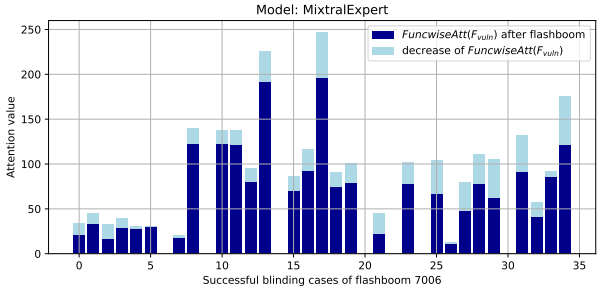

Figure 6: Decrease of FuncwiseAtt(F_{vuln}) after Flashboom 7006 applied on the successful blinding cases. Model: MixtralExpert. Dataset: *Smart-bugs*.

TABLE 8: The semantic similarity between code before and after the insertion of Flashboom.

Auditors	*Smart-bugs* (Solidity)			*Big-Vul* (C/C++)			*CVE-Fixes* (Python)		
	min	max	mean	min	max	mean	min	max	mean
Mistral	0.7594	0.9865	0.8934	0.831	0.996	0.942	0.6857	0.9964	0.9146
CodeLlama	0.7418	0.9853	0.9202	0.7967	0.9978	0.9567	0.7559	0.996	0.9234
Phi	0.65	0.9264	0.8179	0.7167	0.995	0.9513	0.6814	0.9929	0.8803
MixtralExpert	0.7594	0.9865	0.8934	-	-	-	-	-	-
Gemma	0.6447	0.9478	0.7867	-	-	-	-	-	-

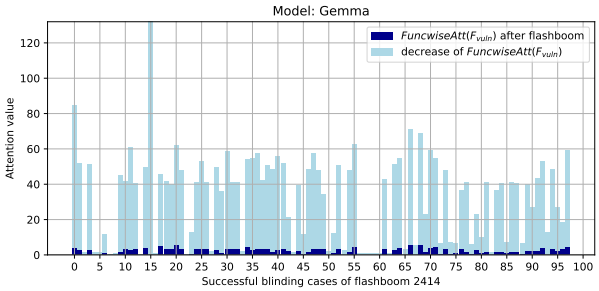

Figure 7: Decrease of FuncwiseAtt(F_{vuln}) after Flashboom 2414 applied on the successful blinding cases. Model: Gemma. Dataset: *Smart-bugs*.

cosine similarity of the embedding vectors, which allowed us to quantify semantic changes. The results, summarized in Table 8, reveal the semantic similarity values (min, max, mean) for each model across the datasets. Overall, Flashboom demonstrated high stealthiness, as evidenced by high mean similarity scores across all datasets, suggesting that code modifications are subtle enough to avoid detection by developers. For instance, CodeLlama consistently achieved the highest mean similarity, with values around 0.92 or higher across datasets, indicating minimal semantic deviation post-insertion. While Phi showed slightly lower similarity scores, especially on Smart-bugs (mean = 0.8179), it still maintained a reasonably high similarity level.

We also incorporated a user study to assess how developers perceive and respond to Flashboom-modified code. Conducted under IRB supervision, our study with 36 developers (1–8 years of experience) tested GitHub Copilot's

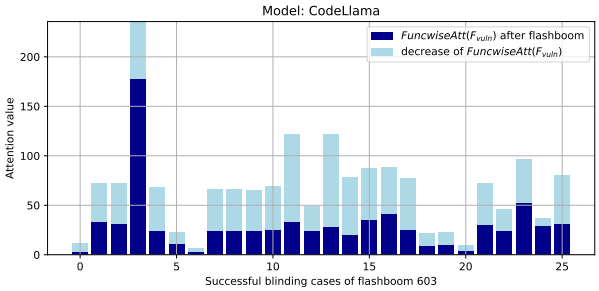

Figure 8: Decrease of FuncwiseAtt(F_{vuln}) after Flashboom 603 applied on the successful blinding cases. Model: CodeLlama. Dataset: *Smart-bugs*.

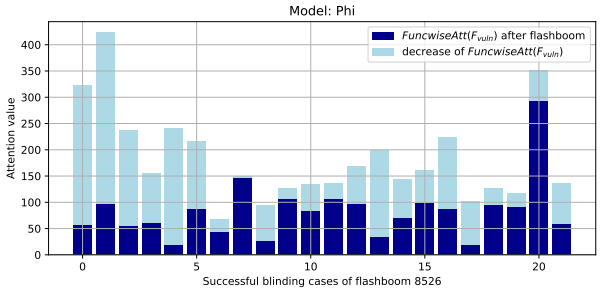

Figure 9: Decrease of FuncwiseAtt(F_{vuln}) after Flashboom 8526 applied on the successful blinding cases. Model: Phi. Dataset: *Smart-bugs*.

vulnerability detection against Flashboom and baseline methods, such as misleading comments (e.g., falsely stating vulnerable code is secure). None detected Flashboom, even with Copilot's assistance. In contrast, 75% (28/36) correctly identified deception in baseline methods like misleading comments or renamed variables. Details of our user study are illustrated in the Appendix C. These findings suggest that Flashboom effectively preserves the original code semantics and remains difficult for developers to notice, thus achieving a high level of stealthiness.

TABLE 9: The min/max/mean of the decrease amount and decrease rate of FuncwiseAtt(F_{vuln}) after top Flashboom of each model applied on the successful blinding cases. **FID** stands for Flashboom ID. Dataset: *Smart-bugs*.

Models	FID	Decrease Amount			Decrease Rate		
		min	max	mean	min	max	mean
Mistral	7006	1.20	71.71	25.86	0.07	0.51	0.32
MixtralExpert	7006	1.24	54.10	22.04	0.06	0.51	0.24
Gemma	2414	0.13	80.47	36.77	0.70	0.97	0.92
CodeLlama	603	3.38	93.89	36.96	0.20	0.78	0.58
Phi	8526	19.91	325.89	108.42	0.16	0.92	0.54

Attribution Analysis (RQ5): In response to the question on whether the decrease in attention scores significantly influences the results, we analyzed the function-wise attention (FuncwiseAtt) for vulnerable functions (F_{vuln}) after applying the top attacks across each model on the *Smart-bugs* dataset, as illustrated in Figure 5 to Figure 9. For all five models, the top Flashboom significantly decreased FuncwiseAtt(F_{vuln}) in most of the cases where attack was successful. As shown in Table 9, the greatest decrease in FuncwiseAtt(F_{vuln}) was caused by the Flashboom 8526 on the Phi model, reaching as high as 325.89, while the largest decrease rate was observed in the Flashboom 2414 of the Gemma model, which reached 0.92. The Flashboom 7006 of the MixtralExpert model caused both the smallest mean of decrease amount and mean of decrease rate, but it was sufficient to launch a successful attack.

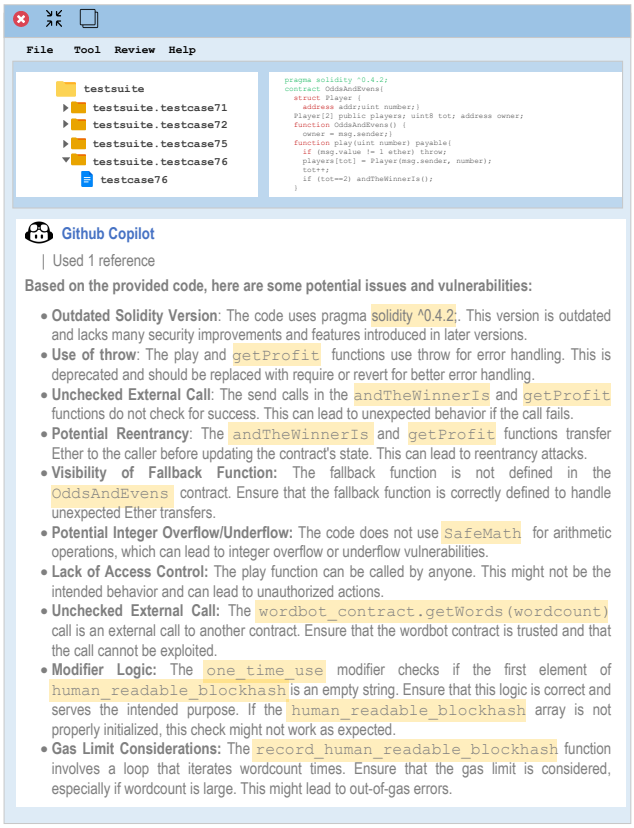

Figure 10: Output of GitHub Copilot

6. Security Case Study on Commercial Auditors

Previous experiments were primarily conducted in a chatbot-based environment, without assessing the effectiveness of our attack within an integrated IDE setting. To address this gap, our case study evaluates the impact of Flashboom on GitHub Copilot, an AI-powered code reviewer utilizing the GPT-4o model, with a specific focus on its ability to detect vulnerabilities in smart contract code. We assessed Copilot's performance using the *Smart-bugs* dataset. Specifically, we used Flashboom 3129[3] to launch our attack, one of the most effective exploits generated by Crazy-Ivan.

When Flashboom 3129 was applied to the *Smart-bugs* dataset and analyzed by Copilot, it led to missed detections of the actual vulnerabilities. The smart contract[4] under testing is a two-player game with a front-running vulnerability (Front-running is a type of vulnerability in blockchain and decentralized applications where an attacker can exploit knowledge of an upcoming transaction to gain an unfair advantage.) on line 8 (`players[tot] = Player(msg.sender, number);`). An attacker could exploit this flaw by observing the second player's input and front-running with the same number, thereby ensuring an unfair win. Before introducing Flashboom 3129, Copilot successfully identified seven vulnerabilities in the contract, including the main front-running issue, along with six unrelated warnings. However, after injecting Flashboom 3129, Copilot failed to detect the front-running vulnerability, instead incorrectly reporting ten different vulnerabilities. Interestingly, as shown in Figure 10 although Copilot failed to identify the actual vulnerability, it still flagged several other potential issues, such as outdated Solidity versions, function visibility concerns, risks of uninitialized state variables, and gas limit considerations.

3. Available at https://github.com/oxygen-hunter/Flashboom/blob/main/case_study/Flashboom_3129.sol

4. Available at https://github.com/oxygen-hunter/Flashboom/blob/main/case_study/front_running.sol

7. Discussion

Ethics Concerns. During our experiments, we adhered strictly to community guidelines to avoid any real-world attacks or causing harm to developers.

- **Controlled Environment**: All testing was conducted in a controlled environment, and interactions with LLMs were limited to our authorized personal accounts. We did not upload any malicious (or vulnerable) code to open-source community or hack any third-party developer.
- **Responsible Disclosure**: We reported our findings to the GitHub Copilot security team and other relevant vendors, providing a detailed explanation of the attack, reproduction steps, and a curated dataset for further analysis. As of the time of writing, some vendors have acknowledged our report. For instance, when we first reported the issue to the Copilot security team in November 2024, they recognized our findings within four days and assigned a Senior Director for Threat Hunting and a Senior Manager for Product Security to address the issue. After our paper was accepted in March 2025, we reached out to them again, but they had no updates and indicated that they are still working on fixing this problem.

Mitigation. Our attack is difficult to defend due to its stealthiness. Here we discuss some possible mitigations, including their strengths and weaknesses.

- **Attack-Aware Fine-Tuning**: By leveraging public genuine Flashboom modules, the developer of LLM-based auditors could fine-tune the model ahead, or guide the model to reference these attack samples during the inference phase. This strategy directs LLMs to purposefully disregard the injected code, reducing the impact of interference. However, considering that it is difficult to enumerate all top-attention modules in the wild, especially when addressing tasks beyond the code level, the defense may lack generalizability.
- **Sequential Masking Defense**: In situations where the actual Flashboom samples are largely unknown, an alternative strategy is to block each suspect individually. We could treat all the functions as attack suspects, segmenting the code in the prompt before inference and masking each segment sequentially. By ignoring the current segment and auditing the rest, this strategy

temporarily removes the influences of potential Flashboom. This mitigation could take effect without the knowledge of attack samples. However, this mitigation comes with two main issues. First, Flashboom elements could be scattered throughout the code, making it difficult to mask all of them once a time. Second, LLMs may depend on contextual information when detecting complex vulnerabilities. Masking the context while isolating the primary vulnerability function may result in insufficient information for LLMs to perform accurate reasoning.

Limitations. As the Flashboom attack performs inconsistently across models, particularly on GPT-4o, further analysis on why certain models resist the attack would be valuable. One possible explanation is that different models have minimal overlap in their training datasets and employ different training methods. As a result, elements that are highly sensitive to attention mechanisms in Model A may not necessarily be effective against Model B. To address this, a more targeted Flashboom attack could be developed using model distillation. Specifically, if the target model is a black-box model, a smaller student model could first be distilled from it through knowledge distillation. The Flashboom search would then be conducted on the student model, and the top-ranked Flashboom would ultimately be used to attack the teacher model.

Broader Impacts of Our Work. We believe we have uncovered a novel attack surface in LLMs, where the core mechanism—attention—can be manipulated by attackers. This manipulation disrupts the LLM's ability to focus on critical information. Although our attack was initially designed for vulnerability detection, it has the potential to generalize to other detection-based tasks, such as malicious code detection (*code level*), abnormal traffic analysis (*formatted-text level*), and phishing email identification (*natural-language level*). To validate this, we applied the approach to malicious code detection and achieved promising results. Specifically, we manually crafted language-specific Flashbooms based on observations and experience, which proved effective against certain ransomware and viruses, successfully evading the malware detection capabilities of the Mistral model. The code snippets used in these experiments are provided in our shared repository.

8. Related Work

LLMs demonstrate significant capabilities in vulnerability detection [33], [41], [43], [14], [38] due to their powerful pattern recognition and reasoning abilities. The advantage of LLM-based vulnerability detection lies in the generalization across multiple tasks and the understanding of logical vulnerabilities. Unlike traditional ML-based vulnerability detection methods that focus only on vulnerability-related features of the code [17], [25], [56], [28], [8], LLMs take source code as input to perform end-to-end detection. Recently, works on evading vulnerability detection of LLMs have gradually gained attention [46], which mainly focus on obfuscating the vulnerable code to cheat LLMs. CODEBREAKER [50] obfuscates payloads to perform evasion attacks on LLM-based detectors. ALERT [51] proposes a subtler perturbation method using variable renaming. By employing a greedy algorithm or genetic algorithm, it replaces tokens in variable names that have the most significant impact on detection accuracy, ensuring semantically similar substitutions using cosine similarity. However, obfuscation has two weaknesses: **(i)** Prompt-specific and target-model-specific perturbations are required, which lacks generalizability and transferability. **(ii)** Obfuscation reduces the stealthiness of attacks, necessitating a careful trade-off between obfuscation concealment and attack success rate. In the contrast, Our Flashboom attack crafts a self-contained top-attention code segment, subtly diverting the focus of LLMs and leading to missed detections of critical vulnerabilities, which exhibits better generalizability, transferability and stealthiness. Additionally, we provide a new perspective, directly exploiting the attention, for launching attacks on LLMs.

9. Conclusion

We have presented a novel attack Flashboom targeting LLM-based auditors by manipulating attention mechanisms. Specifically, we injected high-attention code segments into vulnerable code areas, which effectively diverted the LLM's focus away from vulnerable segments, allowing critical vulnerabilities to evade detection. An automatic tool Crazy-Ivan has been developed to generate attack payloads. Thorough experiments have validated the effectiveness of our tool, achieving a blinding success rate up to 96.3% on CodeLlama and 83.05% on Gemma, with impressive cross-model transferability and applicability across multiple programming languages. Our end-to-end case study on GitHub Copilot further demonstrates the practicality of our attack on real-world tools. In addition to the responsible disclosure of our attack and a curated dataset for reproduction to the GitHub Copilot security team, we have also discussed possible mitigations, for both developers and users of LLM-based auditors. So far, GitHub Copilot security team has acknowledged our findings and is actively working on fixes. In general, our work highlights a new attack surface of LLM-the vulnerability of their fundamental attention mechanism, contributing to the security development of the LLM community.

Acknowledgement

We sincerely thank the anonymous shepherd and reviewers for their constructive feedback and insightful suggestions. This work was supported by the National Key R&D Program of China under Grant 2022YFF0604503; in part by NSFC under Grant 62272224, Grant 62432004, Grant 62302207, and Grant 62272215; in part by the Leading Edge Technology Program of Jiangsu Natural Science Foundation under Grant BK20202001; and in part by the Science Foundation for Youths of Jiangsu Province under Grant BK20220772.

References

[1] Marah Abdin, Jyoti Aneja, Hany Awadalla, Ahmed Awadallah, Ammar Ahmad Awan, Nguyen Bach, Amit Bahree, Arash Bakhtiari, Jianmin Bao, Harkirat Behl, et al. Phi-3 technical report: A highly capable language model locally on your phone. *arXiv preprint arXiv:2404.14219*, 2024.

[2] Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*, 2023.

[3] DeepCode AI. Deepcode ai | ai code review | ai security for sast | snyk ai. https://snyk.io/platform/deepcode-ai/, 2020.

[4] Amazon. Q developer user guide. https://docs.aws.amazon.com/amazonq/latest/qdeveloper-ug/security-scans.html, 2024.

[5] Guru Bhandari, Amara Naseer, and Leon Moonen. Cvefixes: automated collection of vulnerabilities and their fixes from open-source software. In *Proceedings of the 17th International Conference on Predictive Models and Data Analytics in Software Engineering*, pages 30–39, 2021.

[6] Tom B Brown. Language models are few-shot learners. *arXiv preprint arXiv:2005.14165*, 2020.

[7] Solidity by Example. Self destruct attack. https://solidity-by-example.org/hacks/self-destruct/, 2024.

[8] Saikat Chakraborty, Rahul Krishna, Yangruibo Ding, and Baishakhi Ray. Deep learning based vulnerability detection: Are we there yet? *IEEE Transactions on Software Engineering*, 48(9):3280–3296, 2021.

[9] Chong Chen, Jianzhong Su, Jiachi Chen, Yanlin Wang, Tingting Bi, Yanli Wang, Xingwei Lin, Ting Chen, and Zibin Zheng. When chatgpt meets smart contract vulnerability detection: How far are we? *arXiv preprint arXiv:2309.05520*, 2023.

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

[11] CodeRabbit. Cut code review time & bugs in half. https://www.coderabbit.ai/, 2023.

[12] GitHub Copilot. Github copilot. https://github.com/features/copilot, 2024.

[13] Isaac David, Liyi Zhou, Kaihua Qin, Dawn Song, Lorenzo Cavallaro, and Arthur Gervais. Do you still need a manual smart contract audit? *arXiv preprint arXiv:2306.12338*, 2023.

[14] Xueying Du, Geng Zheng, Kaixin Wang, Jiayi Feng, Wentai Deng, Mingwei Liu, Bihuan Chen, Xin Peng, Tao Ma, and Yiling Lou. Vul-rag: Enhancing llm-based vulnerability detection via knowledge-level rag. *arXiv preprint arXiv:2406.11147*, 2024.

[15] Thomas Durieux, João F Ferreira, Rui Abreu, and Pedro Cruz. Empirical review of automated analysis tools on 47,587 ethereum smart contracts. In *Proceedings of the ACM/IEEE 42nd International conference on software engineering*, pages 530–541, 2020.

[16] Jiahao Fan, Yi Li, Shaohua Wang, and Tien N Nguyen. A c/c++ code vulnerability dataset with code changes and cve summaries. In *Proceedings of the 17th International Conference on Mining Software Repositories*, pages 508–512, 2020.

[17] Michael Fu and Chakkrit Tantithamthavorn. Linevul: A transformer-based line-level vulnerability prediction. In *Proceedings of the 19th International Conference on Mining Software Repositories*, pages 608–620, 2022.

[18] Zeyu Gao, Hao Wang, Yuchen Zhou, Wenyu Zhu, and Chao Zhang. How far have we gone in vulnerability detection using large language models. *arXiv preprint arXiv:2311.12420*, 2023.

[19] Kai Greshake, Sahar Abdelnabi, Shailesh Mishra, Christoph Endres, Thorsten Holz, and Mario Fritz. Not what you've signed up for: Compromising real-world llm-integrated applications with indirect prompt injection. In *Proceedings of the 16th ACM Workshop on Artificial Intelligence and Security*, pages 79–90, 2023.

[20] NCC Group. Decentralized application security project top10. https://dasp.co/#item-7, 2018.

[21] Albert Q Jiang, Alexandre Sablayrolles, Arthur Mensch, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Florian Bressand, Gianna Lengyel, Guillaume Lample, Lucile Saulnier, et al. Mistral 7b. *arXiv preprint arXiv:2310.06825*, 2023.

[22] Albert Q Jiang, Alexandre Sablayrolles, Antoine Roux, Arthur Mensch, Blanche Savary, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Emma Bou Hanna, Florian Bressand, et al. Mixtral of experts. *arXiv preprint arXiv:2401.04088*, 2024.

[23] Kamyu104. Leetcode-solutions. https://github.com/kamyu104/LeetCode-Solutions, 2024.

[24] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, et al. Retrieval-augmented generation for knowledge-intensive nlp tasks. *Advances in Neural Information Processing Systems*, 33:9459–9474, 2020.

[25] Yi Li, Shaohua Wang, and Tien N Nguyen. Vulnerability detection with fine-grained interpretations. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 292–303, 2021.

[26] Yue Li, Xiao Li, Hao Wu, Yue Zhang, Xiuzhen Cheng, Sheng Zhong, and Fengyuan Xu. Attention is all you need for llm-based code vulnerability localization. *arXiv preprint arXiv:2410.15288*, 2024.

[27] Zehan Li, Xin Zhang, Yanzhao Zhang, Dingkun Long, Pengjun Xie, and Meishan Zhang. Towards general text embeddings with multi-stage contrastive learning. *arXiv preprint arXiv:2308.03281*, 2023.

[28] Zhen Li, Deqing Zou, Shouhuai Xu, Xinyu Ou, Hai Jin, Sujuan Wang, Zhijun Deng, and Yuyi Zhong. Vuldeepecker: A deep learning-based system for vulnerability detection. *arXiv preprint arXiv:1801.01681*, 2018.

[29] Yupei Liu, Yuqi Jia, Runpeng Geng, Jinyuan Jia, and Neil Zhenqiang Gong. Formalizing and benchmarking prompt injection attacks and defenses. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 1831–1847, 2024.

[30] Zhenguang Liu, Peng Qian, Xiaoyang Wang, Yuan Zhuang, Lin Qiu, and Xun Wang. Combining graph neural networks with expert knowledge for smart contract vulnerability detection. *IEEE Transactions on Knowledge and Data Engineering*, 2021.

[31] Wei Ma, Daoyuan Wu, Yuqiang Sun, Tianwen Wang, Shangqing Liu, Jian Zhang, Yue Xue, and Yang Liu. Combining fine-tuning and llm-based agents for intuitive smart contract auditing with justifications. *arXiv preprint arXiv:2403.16073*, 2024.

[32] Visual Studio Magazine. Github copilot user growth and organizational impact. https://visualstudiomagazine.com/Articles/2024/02/05/copilot-numbers.aspx, 2024.

[33] Noble Saji Mathews, Yelizaveta Brus, Yousra Aafer, Meiyappan Nagappan, and Shane McIntosh. Llbezpeky: Leveraging large language models for vulnerability detection. *arXiv preprint arXiv:2401.01269*, 2024.

[34] Microsoft. Microsoft copilot for security | microsoft security. https://www.microsoft.com/en-us/security/business/ai-machine-learning/microsoft-copilot-security, 2023.

[35] Yu Nong, Mohammed Aldeen, Long Cheng, Hongxin Hu, Feng Chen, and Haipeng Cai. Chain-of-thought prompting of large language models for discovering and fixing software vulnerabilities. *arXiv preprint arXiv:2402.17230*, 2024.

[36] OpenAI. Hello gpt-4o. https://openai.com/index/hello-gpt-4o/, 2024.

[37] PapersWithcode. Human eval benchmark. https://paperswithcode.com/sota/code-generation-on-humaneval, 2023.

[38] Moumita Das Purba, Arpita Ghosh, Benjamin J Radford, and Bill Chu. Software vulnerability detection using large language models. In *2023 IEEE 34th International Symposium on Software Reliability Engineering Workshops (ISSREW)*, pages 112–119. IEEE, 2023.

[39] Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, et al. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*, 2023.

[40] Benjamin Steenhoek, Md Mahbubur Rahman, Monoshi Kumar Roy, Mirza Sanjida Alam, Earl T Barr, and Wei Le. A comprehensive study of the capabilities of large language models for vulnerability detection. *arXiv preprint arXiv:2403.17218*, 2024.

[41] Yuqiang Sun, Daoyuan Wu, Yue Xue, Han Liu, Wei Ma, Lyuye Zhang, Miaolei Shi, and Yang Liu. Llm4vuln: A unified evaluation framework for decoupling and enhancing llms' vulnerability reasoning. *arXiv preprint arXiv:2401.16185*, 2024.

[42] Yuqiang Sun, Daoyuan Wu, Yue Xue, Han Liu, Haijun Wang, Zhengzi Xu, Xiaofei Xie, and Yang Liu. When gpt meets program analysis: Towards intelligent detection of smart contract logic vulnerabilities in gptscan. *arXiv preprint arXiv:2308.03314*, 2023.

[43] 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.

[44] Tabnine. Tabnine ai code assistant | private, personalized, protected. https://www.tabnine.com/, 2020.

[45] Gemma Team, Thomas Mesnard, Cassidy Hardin, Robert Dadashi, Surya Bhupatiraju, Shreya Pathak, Laurent Sifre, Morgane Rivière, Mihir Sanjay Kale, Juliette Love, et al. Gemma: Open models based on gemini research and technology. *arXiv preprint arXiv:2403.08295*, 2024.

[46] Saad Ullah, Mingji Han, Saurabh Pujar, Hammond Pearce, Ayse Coskun, and Gianluca Stringhini. Llms cannot reliably identify and reason about security vulnerabilities (yet?): A comprehensive evaluation, framework, and benchmarks. In *2024 IEEE Symposium on Security and Privacy (SP)*, pages 862–880. IEEE, 2024.

[47] A Vaswani. Attention is all you need. *Advances in Neural Information Processing Systems*, 2017.

[48] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems*, 35:24824–24837, 2022.

[49] Qiushi Wu and Kangjie Lu. On the feasibility of stealthily introducing vulnerabilities in open-source software via hypocrite commits. *Proc. Oakland*, page 17, 2021.

[50] Shenao Yan, Shen Wang, Yue Duan, Hanbin Hong, Kiho Lee, Doowon Kim, and Yuan Hong. An {LLM-Assisted}{Easy-to-Trigger} backdoor attack on code completion models: Injecting disguised vulnerabilities against strong detection. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 1795–1812, 2024.

[51] Zhou Yang, Jieke Shi, Junda He, and David Lo. Natural attack for pre-trained models of code. In *Proceedings of the 44th International Conference on Software Engineering*, pages 1482–1493, 2022.

[52] Jia-Yu Yao, Kun-Peng Ning, Zhen-Hui Liu, Mu-Nan Ning, Yu-Yang Liu, and Li Yuan. Llm lies: Hallucinations are not bugs, but features as adversarial examples. *arXiv preprint arXiv:2310.01469*, 2023.

[53] Yifan Yao, Jinhao Duan, Kaidi Xu, Yuanfang Cai, Zhibo Sun, and Yue Zhang. A survey on large language model (llm) security and privacy: The good, the bad, and the ugly. *High-Confidence Computing*, page 100211, 2024.

[54] Xin Zhang, Yanzhao Zhang, Dingkun Long, Wen Xie, Ziqi Dai, Jialong Tang, Huan Lin, Baosong Yang, Pengjun Xie, Fei Huang, et al. mgte: Generalized long-context text representation and reranking models for multilingual text retrieval. *arXiv preprint arXiv:2407.19669*, 2024.

[55] Xin Zhou, Sicong Cao, Xiaobing Sun, and David Lo. Large language model for vulnerability detection and repair: Literature review and roadmap. *arXiv preprint arXiv:2404.02525*, 2024.

[56] Yaqin Zhou, Shangqing Liu, Jingkai Siow, Xiaoning Du, and Yang Liu. Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. *Advances in neural information processing systems*, 32, 2019.

[57] Andy Zou, Zifan Wang, Nicholas Carlini, Milad Nasr, J Zico Kolter, and Matt Fredrikson. Universal and transferable adversarial attacks on aligned language models. *arXiv preprint arXiv:2307.15043*, 2023.

Appendix A.
Calculation of Function-wise Attention

In §4.2, we design an algorithm to calculate function-wise attention score for the codebase, for the purpose of selecting top-attention functions as Flashboom candidates. As illustrated in algorithm 1, with a code file consisting of multiple functions (codebase) as input, we start with the initial attention tensor of the codebase. After summarizing attention values across multiple attention heads, taking the last token as the query token to keep the semantic information, and summarizing attention values across different decoder layers, we get the line-wise attention array. The final step is aggregating line-wise attention score for each function in the code. FuncwiseAtt, represents the total attention weights that a model allocates to each function.

Algorithm 1: Function-wise Attention Calculation

Data: Codebase, a code file with multiple functions: C.
Result: Function-wise attention score of the codebase C: $\text{FuncwiseAtt} = \{a_{f_1}, \ldots, a_{f_N}\}$.

begin
- $\{f_1, \ldots, f_N\} \leftarrow$ all functions in C
- $\{\lambda_1, \ldots, \lambda_M\} \leftarrow$ all lines of C
- $\mathbf{A}^{(T,T,L,H)} \leftarrow \texttt{Gen}(C)$
  `// generate initial attention tensor` $\mathbf{A}$ `with a shape of` (T, T, L, H)
- $\mathbf{A}^{(T,T,L)}_{\text{head}} \leftarrow \sum_H \mathbf{A}^{(T,T,L,H)}$
  `// sum attention across heads`
- $\mathbf{A}^{(T,L)}_{\text{head}} \leftarrow \mathbf{A}^{(T,T,L)}_{\text{head}}[-1, :, :]$
  `// take the attention score of the last query token`
- **for** $\lambda_i \leftarrow \lambda_1$ **to** λ_M **do**
  - $I_{\lambda_i} \leftarrow$ indices of line i's tokens
  - $\mathbf{A}^{(L)}_{\text{layer}} \leftarrow \sum \mathbf{A}^{(T,L)}_{\text{head}}[I_{\lambda_i}, :]$
    `// calculate the layer-wise attention of line` i
  - $a_{\text{line}_i} \leftarrow \sum_L \mathbf{A}^{(L)}_{\text{layer}}$
    `// calculate the line-wise attention`
- **for** $f_i \leftarrow f_1$ **to** f_N **do**
  - $a_{f_i} \leftarrow \sum_{l \in f_i} a_{\text{line}_l}$
    `// sum layer-wise attention score of lines in function` f_i
- $\text{FuncwiseAtt} \leftarrow \{a_{f_1}, \ldots, a_{f_N}\}$

Appendix B.
Prompt Used In Our Design

We applied a few of prompts in our design and experiments. Here, we provide a selection of them to facilitate reproducibility. As shown in Figure 11, we list the prompts used in §4.2, where we leverage GPT-4o's reasoning ability to resolve the dependency of F_{max}. We require the LLM to resolve the dependency of a given function in the specific source code, collecting all the referenced contents of that function, such as the variables, functions, and some language specific elements. Figure 12 illustrates the prompt we use in §5 to audit target code. First we build the prompt with a step-by-step guide for detecting vulnerabilities in the specific programming language. Then we provide a code example with a detailed vulnerability explanation, such as vulnerability type and CVE/CWE ID. Finally, we submit the code to the LLM and request an audit report.

System prompt

You are an intelligent code assistant designed for extracting some specific function from a complete Solidity code.
I will give you a piece of complete solidity code, which contains several contracts, interfaces, and libraries.
Your task is to extract the content of a specific function, along with all of its referenced contents. Here is how you can accomplish the task:

##INSTRUCTIONS:
- Target the specific function, with contract name and function name.
- View function code line by line, and record all symbols it used.
- Find the definition of these referenced symbols in other parts of Solidity code, including variables, functions, modifiers, events, and so on.
- Extract the specific function and its directly referenced contents.
- If any referenced state variable is initialized in the constructor function, do not include the constructor.
- Discard the pragma statement

User prompt

Please extract the following function and its referenced contents from the following Solidity code. You should discard all the irrelevant content and just give me the intercepted code.
Function: {function_name}
Solidity code: {source_code}
Please generate the response in Solidity code format.
DO NOT PROVIDE ANY OTHER OUTPUT TEXT OR EXPLANATION. DO NOT WRAP CODE WITH ```Solidity```. Only provide the intercepted code.

Figure 11: Prompt template for dependency resolution. We take Solidity language as an example. C/C++ and Python are similar.

System prompt

Auditing a smart contract is a critical process to ensure its security, reliability, and efficiency. Below is a step-by-step guide to audit a smart contract:
1. Understand the Smart Contract: The first step is to understand the smart contract's functionality. ...
2. Identify the Contract's Functions: After understanding the contract's purpose, identify all the functions within the contract. ...
3. Analyze Function Modifiers: Function modifiers can change the behavior of functions in a smart contract. ...
4. Check for Reentrancy Attacks: Reentrancy attacks are a common vulnerability in smart contracts. This occurs when ...
5. Check for Arithmetic Overflows and Underflows: Solidity, the language most commonly used for writing smart contracts, does not handle arithmetic overflows and underflows well. ...
6. Check for Unchecked Return Values: Many functions in Solidity return a boolean value indicating success or failure. ...
7. Check for Access Control Vulnerabilities: Ensure that all functions that should be restricted to certain addresses are properly protected. ...
8. Check for Randomness: If the contract uses randomness, ensure it is generated in a secure way. ...
9. Check for Gas Limit and Loops: Loops that run for an indeterminate number of iterations can cause a contract to run out of gas and fail. ...
10. Check for Timestamp Dependence: If the contract uses the block timestamp for critical functionality, it can be manipulated by miners. ...
Remember, this is a basic guide and may not cover all potential vulnerabilities. Smart contract auditing is a complex task that requires a deep understanding of blockchain technology and the Solidity programming language.
There are still a lot of vulnerabilities not listed here, don't miss them.

User prompt

First review the following smart contract code. I am sure that it is vulnerable. The vulnerability type is {vuln_type}.
code: {example_code}
I will explain why it is vulnerable: {explanation}
Based on the given code, evaluate whether the following smart contract code is vulnerable.
code: {code_to_audit}
Your answer should be in such format: first tell me 'YES' or 'NO', if 'YES', tell me why it is vulnerable, if 'NO', don't tell me anything more.

Figure 12: Prompt for code auditing. We take Solidity language as an example. C/C++ and Python are similar.

Appendix C. User Study of Attack Stealthiness

To evaluate how developers perceive and respond to Flashboom-modified code, under IRB supervision, we conducted a user study among 36 developers (1-8 years of experience in programming). We curated 8 pieces of code, including 4 types: 2 vulnerable-free code, 2 vulnerable code without modification, 2 vulnerable code with Flashboom injected, and 2 vulnerable code with baseline methods modified like misleading comments and renaming. As illustrated in Figure 13, the users were required to detect the existence of vulnerability in the code (**RQ1** and **RQ2**), with the assistance of GitHub Copilot, and answering whether any suspicious contents exists that may mislead the detection process of GitHub Copilot (**RQ3**). We publicly release our user study[5], including questionnaire design, tested code and user feedback.

We measured the number of times that each code snippet was mentioned in **RQ3** answers (especially the ones which are modified by Flashboom and baseline methods). Ultimately, we found that none of the participants suspected the Flashboom-modified code of containing any intentional attempts to mislead GitHub Copilot or human reviewers. One developer with a background in smart contract security even commented after the study, *"I really couldn't tell where Flashboom was embedded—it just looked like typical on-chain code, where unrelated logic is often included."* In contrast, for baseline methods mentioned in our paper—such as carefully crafted misleading comments claiming the function is secure or renaming variables and functions to mimic standard security implementations—**75% (28/36)** of participants correctly identified these techniques as deliberate attempts to mislead LLM-based vulnerability detection. The results indicate that Flashboom demonstrates strong stealthiness.

Guide & Questions

Guide:Use GitHub Copilot to perform vulnerability detection on each piece of code by asking, for example, "Are there any vulnerabilities in the code?". After completing the detection, answer the following questions for each piece of code:
RQ1: Are there any security vulnerabilities in the code? (Yes/No)
[If the answer to **RQ1** is Yes, please answer] **RQ2**: What important security vulnerabilities did GitHub Copilot identify? (You may directly paste the detection results from GitHub Copilot.)
RQ3: Does the tested code contain any intentionally misleading content that could affect GitHub Copilot's detection (including misleading code, comments, etc.)? (Yes/No, if Yes, please list them.)

Figure 13: Guide & Questions in user study.

5. Available at https://github.com/oxygen-hunter/Flashboom_User_Study

Appendix D. Meta-Review

The following meta-review was prepared by the program committee for the 2025 IEEE Symposium on Security and Privacy (S&P) as part of the review process as detailed in the call for papers.

D.1. Summary

This paper investigates adversarial attacks on LLM-based vulnerability auditors by leveraging attention mechanisms to mislead detection. The proposed Flashboom attack manipulates attention scores to obscure vulnerabilities, reducing the model's ability to flag security risks. The authors conduct extensive evaluations across multiple datasets, programming languages, and models, with a real-world case study on GitHub Copilot.

D.2. Scientific Contributions

- Introduces an adversarial attack leveraging attention mechanisms to evade LLM-based vulnerability detection.
- Demonstrates the effectiveness of the attack across different LLMs and datasets.
- Provides a large-scale empirical evaluation, including a real-world case study.

D.3. Reasons for Acceptance

1) **Novel attack surface**: The paper introduces a unique adversarial attack that exploits attention mechanisms, a relatively underexplored attack vector in adversarial machine learning and software security.
2) **Comprehensive evaluation**: The authors perform extensive experiments, showing cross-model transferability and evaluating the attack in real-world scenarios, such as GitHub Copilot.
3) **Potential security implications**: The work highlights systemic risks in LLM-based vulnerability detection, contributing valuable insights to both adversarial ML and software security research.

D.4. Noteworthy Concerns

1) **Practical impact**: A reviewer questions the real-world relevance of attacking LLM-based auditors, given existing research suggesting their vulnerability detection capabilities are already limited. A discussion of LLM adoption rates and practical deployment scenarios would strengthen the paper.
2) **Evaluation methodology**: The effectiveness metrics used (e.g., BSR@exist and BSR@type) may not fully capture real-world vulnerability detection failures. More rigorous metrics that focus on ground-truth vulnerabilities would improve the evaluation.
3) **Stealthiness of the attack**: The injected Flashboom code could raise flags in security pipelines, prompting developers to inspect and remove it. A stronger justification for the attack's stealth properties is needed.
4) **Role of attention mechanisms**: The attribution of attack success to attention score reductions appears ad-hoc. The paper should better distinguish between heuristic intuition and technical fact when discussing attention in LLMs.
5) **Transferability limitations**: The attack performs inconsistently across models, particularly on GPT-4o, raising concerns about its generalizability. Further analysis on why certain models resist the attack would be valuable.
6) **Correct errors/misunderstandings**: Many reviewers pointed out errors and content that can lead to misunderstanding. For instance, sharing the code should be done in a timely manner (please try to do it by the revision deadline). Provide results regarding the C5 in the rebuttal (promised to assign senior security leaders to address it).

Appendix E. Response to the Meta-Review

We appreciate the reviewers' constructive feedback and the shepherd's support in refining our work. We would like to respond to the meta-review concerns as follows:

1) **Practical impact**: At the end of §2.2, we added a paragraph discussing LLM adoption rates and practical deployment scenarios.
2) **Evaluation methodology**: In §5, we introduced a new metric, ***BSR@strict***, which counts success only when the LLM-auditor produces no warnings/errors. Results only drop by 1%, meaning 99% of cases still succeed under this stricter setting.
3) **Stealthiness of the attack**: In RQ4 of §5, we added a user study on developers' perceptions of our attack. Conducted under IRB supervision, our study with 36 developers (1–8 years of experience) tested GitHub Copilot's vulnerability detection against Flashboom and baseline attacks, such as misleading comments (e.g., falsely stating vulnerable code is secure). None detected Flashboom, even with Copilot's assistance. In contrast, 75% (28/36) correctly identified baseline attacks.
4) **Role of attention mechanisms**: Throughout the paper, we clarified that attention is a set of weights, not a direct focus indicator, by using "**focus**" for intuition and "**attention weights/scores/values**" in later sections for technical accuracy.
5) **Transferability limitations**: In §7, we expanded our discussion on attack limitations, providing a further analysis on why certain models (e.g., GPT-4o) resist the attack—due to training data, and exploring potential solutions—model distillation.
6) **Correct errors/misunderstandings**:
   - Share the code: As mentioned in abstract, our source code, dataset and attack results are available now at

https://github.com/oxygen-hunter/Flashboom.

- Responsible disclosure results: In §7, we included the latest update on our disclosure to the GitHub Copilot team—as of our paper's acceptance (March 2025), they are still working on the fix.