2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)

Closing the Gap: A User Study on the Real-world Usefulness of AI-powered Vulnerability Detection & Repair in the IDE

Benjamin Steenhoek[1*], Kalpathy Sivaraman[†], Renata Saldivar Gonzalez[†], Yevhen Mohylevskyy[†], Roshanak Zilouchian Moghaddam[†], and Wei Le[*]
[*]Department of Computer Science, Iowa State University, Ames, IA, USA
[†]Microsoft Data & AI, Redmond, WA, USA

***Abstract*—Security vulnerabilities impose significant costs on users and organizations. Detecting and addressing these vulnerabilities early is crucial to avoid exploits and reduce development costs. Recent studies have shown that deep learning models can effectively detect security vulnerabilities. Yet, little research explores how to adapt these models from benchmark tests to practical applications, and whether they can be useful in practice.**

This paper presents the first empirical study of a vulnerability detection and fix tool with professional software developers on real projects that they own. We implemented DEEPVULGUARD, an IDE-integrated tool based on state-of-the-art detection and fix models, and show that it has promising performance on benchmarks of historic vulnerability data. DEEPVULGUARD scans code for vulnerabilities (including identifying the vulnerability type and vulnerable region of code), suggests fixes, provides natural-language explanations for alerts and fixes, leveraging chat interfaces. We recruited 17 professional software developers at Microsoft, observed their usage of the tool on their code, and conducted interviews to assess the tool's usefulness, speed, trust, relevance, and workflow integration. We also gathered detailed qualitative feedback on users' perceptions and their desired features. Study participants scanned a total of 24 projects, 6.9k files, and over 1.7 million lines of source code, and generated 170 alerts and 50 fix suggestions. We find that although state-of-the-art AI-powered detection and fix tools show promise, they are not yet practical for real-world use due to a high rate of false positives and non-applicable fixes. User feedback reveals several actionable pain points, ranging from incomplete context to lack of customization for the user's codebase. Additionally, we explore how AI features, including confidence scores, explanations, and chat interaction, can apply to vulnerability detection and fixing. Based on these insights, we offer practical recommendations for evaluating and deploying AI detection and fix models. Our code and data are available at this link: https://doi.org/10.6084/m9.figshare.26367139.

***Index Terms*—deep learning, vulnerability detection, vulnerability repair, IDE, user study**

I. INTRODUCTION

Security vulnerabilities impact users' safety, security, and privacy and cost organizations millions of dollars per year [25, 30], with reports of breaches exposing millions of records becoming commonplace [3]. Early detection of vulnerabilities during the development phase can greatly reduce costs and mitigate potential impacts [4, 7, 24]. In recent years, deep learning (DL) vulnerability detection models have emerged as a promising approach for scanning code during software development [11, 21, 48]. These models can identify vulnerability patterns in code snippets and offer the advantage of analyzing code during editing [12] with less configuration than traditional static analysis tools [22].

Despite promising benchmark performance [11, 21, 48], it remains unclear whether these models are actually useful in real-world development settings. In the past, Major organizations such as Microsoft [16], Google [43], Facebook [18], and Coverity [5] have reported a gap between benchmarking success and practical application with static analyzers. Recently, Fu et al. [22] conducted a preliminary controlled study with 6 developers, showing that AI tool support reduced the time to diagnose and fix a vulnerability from 10-15 minutes to 3-4 minutes and motivating further user studies of AI detection and fix tools. However, their study used a single bug from their dataset rather covering real-world code-bases and they only studied vulnerability detection and fixing.

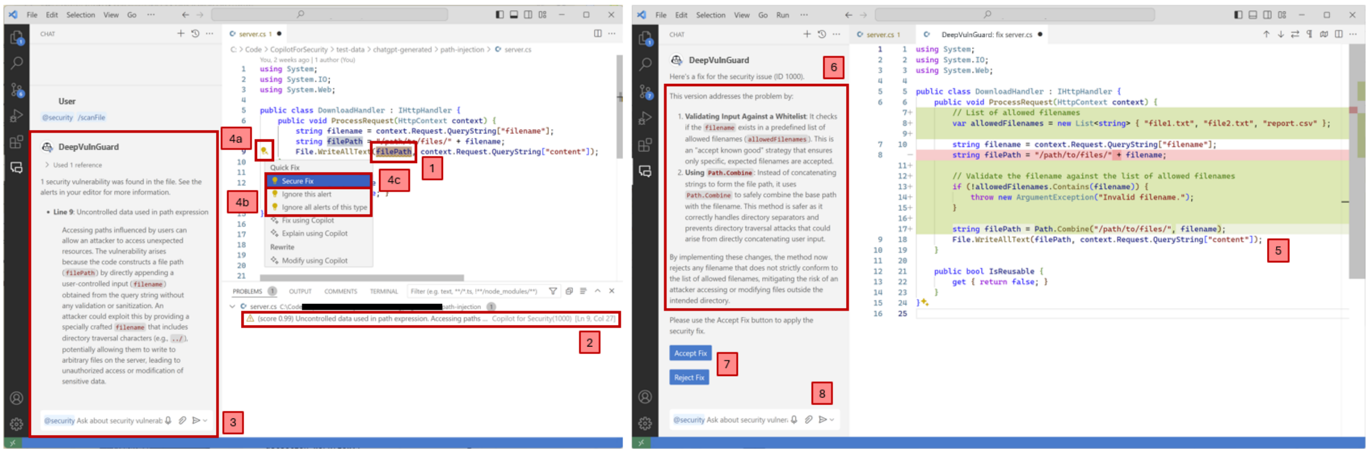
In our work, we recruited 17 professional developers at Microsoft to use our detection & repair tool in a *real-world development setting* with their own projects; beyond detection and fixing, we also built and studied AI-powered *explanation and chat interfaces*, which have recently become prominent in the integrated development environments (IDEs) [1]. Our study provides a deeper understanding of the real-world usefulness and nuances of deploying these models.

To carry out our study, we developed DEEPVULGUARD, an extension integrated with Visual Studio Code (VSCode) [35], a popular IDE with over 14 million active users. We used state-of-the-art models, CodeBERT [12, 20] and the GPT-4 large language model (LLM) [42], for detection and fix tasks. Participants scanned 24 projects, 6.9k files, and over 1.7 million lines of source code, generating 170 alerts and 50 fix suggestions. To the best of our knowledge, ours is the first study to evaluate a detection and fix tool with professional developers on their own projects.

We initially evaluated DEEPVULGUARD's potential for deployment by testing its detection and fix models on established vulnerability datasets. Our models achieved 80%

[1] Work primarily done during an internship at Microsoft.
Corresponding email: bensteenhoek@microsoft.com

1558-1225/25/$31.00 ©2025 IEEE
DOI 10.1109/ICSE55347.2025.00126



Detection workflow

Fix suggestion workflow

Fig. 1: Overview of DEEPVULGUARD's user interface on an example program. (1) An editor alert; (2) Problems menu entry; (3) The explanation of the alert; (4a) Quick fix interaction; (4b) Ignore options; (4c) Fix trigger; (5) Suggested fix; (6) Explanation of the fix suggestion; (7) Accept/Reject buttons.

precision, 32% recall, and a 46% F1 score on SVEN [23] for vulnerability detection and fixed 13% of vulnerabilities on the Vul4J [9] dataset. DEEPVULGUARD performs comparably or better than state-of-the-art models [8, 17, 52] and meets the threshold for acceptable false positives [15]. These results indicate that our models are promising for detecting and fixing security vulnerabilities and can generate meaningful results for the user study.

Our results show that 59% of participants expressed interest in future use of DEEPVULGUARD, although there are several issues that limit its usefulness. For example, one problem was an high rate of false positives in practice, caused by incorrect vulnerability pattern recognition and lack of context about code snippets (e.g., inter-procedural vulnerabilities). This highlights the need for more precise pattern recognition and better integration of environment and program context. Additionally, the requirement to trigger a manual scan significantly disrupted the users' workflow; developers prefer tools that run in the background and alert them whenever potential vulnerabilities are detected. Regarding fixes, 75% of proposed security fixes were unsuitable to apply "as-is" due to lack of customization and incorrect integration into the code. Although some fixes were functionally correct, they were not tailored to the user's codebase and could not be applied without significant modifications. An interactive chat method shows promise to allow developers to guide the generation towards more applicable fixes. Our findings offer concrete recommendations for improving these pain points found in these tools.

In this paper, we make the following research contributions:

1) We developed DEEPVULGUARD a VSCode extension for detecting, explaining and fixing vulnerabilities, incorporating insights from static analysis and AI tool research. Our tool allows customization of backend models, and we provide its code in our data package to support further user studies. DEEPVULGUARD uses a multi-task training approach for jointly predicting vulnerability classification, localization, and bug type. We also introduced a new vulnerability filtering method with LLMs which improved precision by over 20%.
2) We conducted a user study with 17 professional software engineers at Microsoft. Through interviews and surveys as they ran our tool on their own code, we quantitatively assessed multiple dimensions of usefulness for detection and fix tools and provided practical recommendations for improving deep learning-based vulnerability detection and fix tools.

II. USER STUDY INTERFACE

To study whether deep learning-based vulnerability tools can be useful in practice, we built DEEPVULGUARD, a Visual Studio Code extension that brings state-of-the-art detection + fix techniques to an IDE interface. DEEPVULGUARD allows users to (1) scan source code with CodeBERT and LLM models, (2) view the reported vulnerabilities and LLM-generated explanations directly inside the editor, and (3) generate suggestions for mitigating the vulnerability. We also implemented a telemetry module to collect user data, enabling longitudinal studies of AI-based vulnerability detection tools. As our study is the first of its kind in this area, we believe our tool will be a beneficial contribution which facilitates future user studies of vulnerability tools. We released the extension code in our data package. The code can be easily adjusted to call alternative detection and fixing solutions to be studied with developers in a real-world setting.

A. IDE Integration

Figure 1 shows an overview of DEEPVULGUARD's user interface. Users begin by requesting to scan a file or directory. If any potential vulnerabilities arise, they are shown as highlights

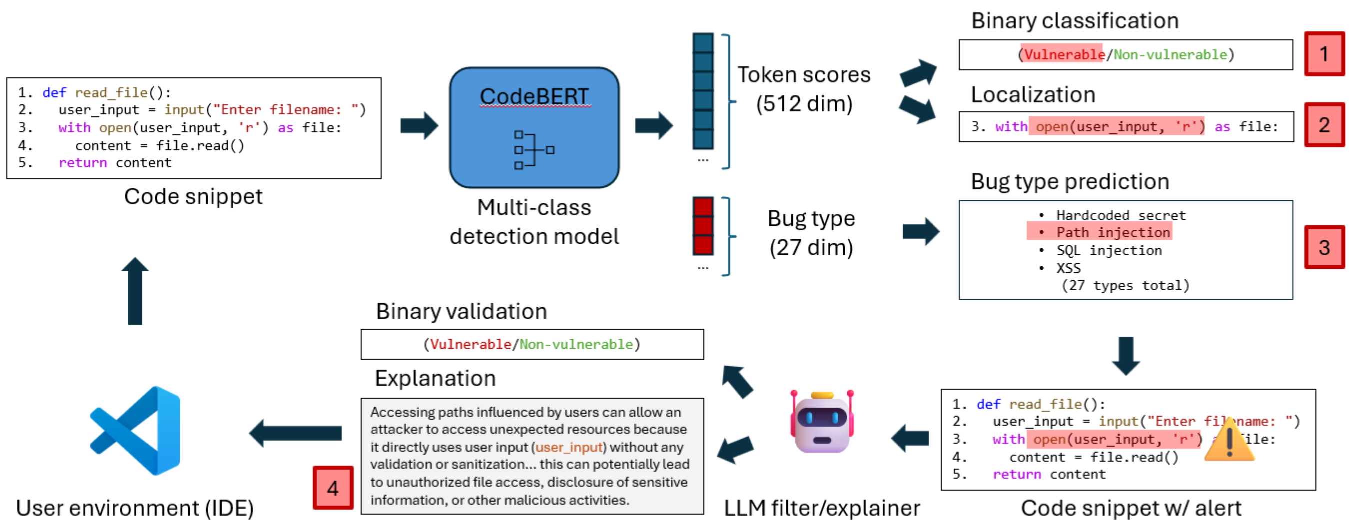


Fig. 2: An overview of DeepVulGuard's detection workflow. (1) Binary classification into vulnerable/not-vulnerable; (2) Localization; (3) Multi-class classification into one of 27 vulnerability types; (4) Alert and explanation shown to the user.

in the editor (1) and actionable entries in the Problems window (2), and a natural-language explanation of the vulnerability is shown in the chat panel (3). The user can use this information to assess the vulnerability and decide if a fix is required. They can also ask questions or make suggestions to the chatbot by sending follow-up messages. By clicking on the *quick fix* lightbulb (4a), the user can ignore the specific alert or alert types (4b), or generate a *quick fix* (4c). On requesting the quick fix, the suggested code modifications will be presented in a *diff* view (5), showing the lines to be removed and added. As well, an explanation of the fix is shown in the chat panel (6). The user can modify the fix in the editor or suggest improvements with natural-language chat messages if desired, then Accept it to apply it to their files or Reject it to revert to the original code (7). Users can enter chat messages (8), e.g. asking for clarification, information, or inputs which trigger the vulnerability, and our tool will generate a conversational response.

We drew inspiration for our tool's design from several foundational research studies on static analyzers and AI-assisted developer tools. Johnson et al. [27] showed that developers requested static analysis tools to be available in the IDE, along with quick fixes, and the ability to modify rule sets. Similarly, Christakis and Bird [15] identified bad warning messages, lack of suggested fixes, and poor visualization as pain points. Smith et al. [46] presented design guidelines, such as presenting alerts in actionable locations, integrating with their workflow by tracking progress, batch processing, allowing code editing during scans, and scalability of the interface. We incorporated all of these features into DeepVulGuard.

A recent study on AI-powered code completion Wang et al. [50] found that users in focus groups valued the ability to view a measure of the model's confidence. To study this in a practical implementation, we integrated confidence scores into our tool's alerts, shown in Figure 1 (2). Fu et al. [22] conducted a survey study and found that most participants valued localizations, CWE type prediction, and quick fixes, so we integrated these features into our tool and evaluate them in our study, shown in Figure 1 (1, 2, and 4a).

B. Model Architecture & Training

Our tool can be easily configured to leverage a wide variety of deep learning models or static analyzers. Figure 2 shows the workflow of the current design; specifically, we implemented the following techniques (please refer to our data package [2] for the implementation details, including our model training procedure, dataset statistics, and hyper-parameters).

Fine-tuning CodeBERT for multi-task vulnerability detection: CodeBERT [20] and similar models consistently perform well on various vulnerability datasets [21, 22, 48] with relatively low latency which is suitable for detection in the editor [12]. We fine-tuned CodeBERT using *multi-task learning* to (1) predict whether a code snippet contains a vulnerability, (2) localize the tokens causing it, and (3) identify the vulnerability type. We trained on a dataset of over 1.3 million alerts labeled by CodeQL in GitHub projects following Chan et al. [12]'s methodology, focusing on 27 vulnerability types related to Web security, e.g. Path Injection, SQL Injection, Hard-coded Credentials, Unvalidated URL Redirect, Cross-Site Scripting (details in data package). We generate alerts in the extension based on the predicted vulnerability type, confidence score, and localization.

Filtering and explaining alerts with GPT-4: To further filter the false positives produced by fine-tuned CodeBERT, we used GPT-4 [42] to filter the alerts and generate explanations. We annotated the code snippet with a comment describing the alert type at the localized line, e.g., for SQL injection: `// ALERT: This SQL query depends on a user-provided value` (see our data package for all

```
You are a vulnerability detector. Only respond with
↪  "Yes" or "No" and an explanation. Does the
↪  following code snippet contain a SQL Injection
↪  vulnerability at line marked by ALERT?
```

Fig. 3: DEEPVULGUARD's LLM filter prompt.

types of annotations). Then we instructed GPT-4 to confirm whether the vulnerability is present. If the answer is *Yes*, the alert and explanation are shown to the user; otherwise, the alert is not shown ((4) in Figure 2). We tried several prompts and evaluated on the SVEN benchmark [23], ultimately selecting the prompt shown in Figure 3. This prompt improved the Precision to an acceptable threshold of 80% [16] while keeping the best Recall.

```
A static analyzer has identified a {rule_id}
↪  security vulnerability in the {language} method
↪  below:

```
{method}
```

The SARIF result message is as follows: {message}

{description}

Write a fixed version of the method above and wrap
↪ it in triple backticks, then explain why your
↪ version addresses the problem.
```

Fig. 4: DEEPVULGUARD's fix model prompt.

Prompting GPT-4 for repair and explanation: We used GPT-4 with custom prompts to generate and explain code fixes. The prompt, shown in Figure 4, includes the source code, vulnerability report, and an instruction to provide a fixed version of the code and an explanation. We displayed the explanation in the chat panel and inserted the code suggestion to show a diff with the original content, shown in Figure 1 on the right. As with the LLM filter, we iterated on several prompts and chose the best performance on an internal dataset of bugs and Vul4J [9].

C. Evaluating Detection and Fix Capabilities

To ensure that DEEPVULGUARD is both effective and representative of the state-of-the-art, we tested its performance on benchmarks that resemble the real-world deployment scenario as closely as possible. To evaluate DEEPVULGUARD's detection capability, we used the SVEN dataset [23], which contains 380 high-quality vulnerability examples from open-source Python projects (93% label accuracy [17])). Our detection model supports all the security vulnerability types present in the dataset. We used a strict definition for true positives: the predicted bug type must match, and localized line number must match the lines changed in the patch. Figure 5 shows on the SVEN dataset our model achieved 80% Precision and 32% Recall, with an F1 score of 46%.

Overall, our results are better than or on par with the prediction quality of SOTA models on vulnerability detection. For example, most recently, Ding et al. [17] reported that SOTA models, including CodeBERT, attained 18-21 F1 score on their dataset of C/C++ vulnerabilities. We cannot directly compare our model with other SOTA models on our dataset as most are trained on C/C++-specific memory or pointer bugs [11, 17, 21, 33, 47, 48].

Christakis and Bird [15] found that most developers tolerate up to a 20% false-positive rate; with the LLM filter, our model meets this threshold on the SVEN dataset, with 80% precision. These results highlight DEEPVULGUARD's practical effectiveness and potential for deployment in real-world applications.

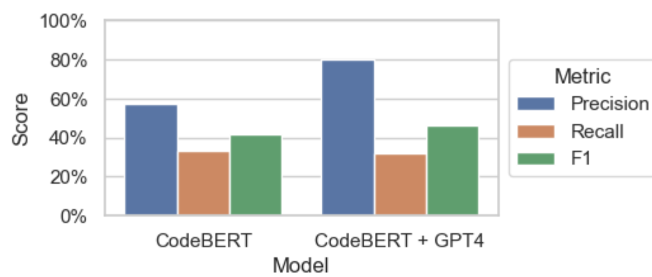


Fig. 5: Performance of DEEPVULGUARD's detection component on vulnerabilities from SVEN.

To evaluate DEEPVULGUARD's fix component, we used the Vul4J [9] dataset, which includes executable tests to reproduce security vulnerabilities. We assessed the test results, supplemented by manual validation, to verify that the suggested fixes mitigated issues without breaking other functionality. Among the 24 single-hunk bugs with vulnerability types that our tool handles, our model produced 3 (13%) correct fixes and 2 (8%) partial fixes, which resolved the issue but broke 1-3 other tests; 10 (42%) fixes had errors inserting the generated code into the file and 9 (37%) fixes could not compile. For efficiency needed for using in IDE, we chose to not run LLM multiple times. These results show that our model performs similarly to SOTA evolution-based automated program repair (APR) tools [8] (13% correct fixes, taking up to 7 minutes in the 75th percentile, intersection $n = 24$) and LLMs such as Codex [52] (15.4% plausible fixes on the first try, intersection $n = 13$).

We conducted the above performance probe to confirm that DEEPVULGUARD can be used to conduct a meaningful study; that it is practical for handling real-world vulnerabilities and offers performance comparable to state-of-the-art techniques. We did not aim for a comprehensive controlled evaluation to claim that DEEPVULGUARD outperforms the current state of the art.

III. USER STUDY DESIGN

We developed three research questions to guide our study.

RQ1: Is DEEPVULGUARD useful in practice?
RQ2: Which aspects of vulnerability detection + fix tools are most useful?
RQ3: What features do developers want from vulnerability detection + fix tools?

A. Study design

We carried out an *exploratory case study* [19] with a group of 17 professional developers at Microsoft. We asked users to run DEEPVULGUARD on projects they were actively developing or were familiar with, and answer survey questions about their perception of the tool. To the best of our knowledge, our tool is the first vulnerability detection + fix tool to be studied in a real-world setting with professional developers on projects which they own. This study enabled us to explore many open questions such as the role of explanations, the developers' tolerance for false-positives or delayed results, and what constitutes an effective fix in a secure development context. We chose an exploratory study over, e.g. a controlled study, because it elicits rich feedback from developers in a real-world setting. Our approach takes advantage of the developers' deep understanding of their own projects, leading to a more accurate assessment of potential vulnerabilities and providing more valuable insights.
Recruitment: We carried out our study with a group of 17 Microsoft developers. We recruited developers primarily using snowball sampling, with a 53% participation rate. In total, participants scanned a total of 24 projects, 6.9k files, and over 1.7 million lines of source code, and generated 170 alerts and 50 fix suggestions.

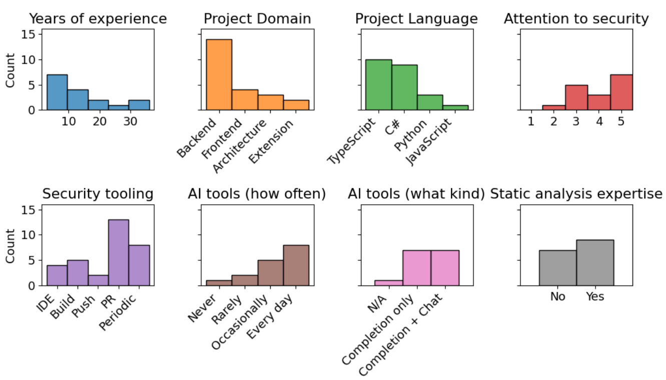


Fig. 6: Participant demographics and tool adoption. Where applicable, participants listed multiple items for the project domain, project language, and security tooling.

Figure 6 shows the participants' demographic information (with the exception of one participant who declined the demographic survey), indicating that we studied a diverse set of developers from various levels of experience and backgrounds. Participants had a median of 11 years of experience; participants worked on both front-end and back-end domains and represented a diverse set of applications such as web applications, back-end services, IDE extensions. Most projects were written in C# or TypeScript. Most participants considered themselves security-conscious (median 4 out of 5), and all had some form of security tool running in continuous integration or periodically, though more than half of the projects used tools for reasons of organizational compliance rather than individual initiative; most participants did not use security tools in the IDE. The majority of participants used AI-powered tools occasionally (a few times a week) or every day, such as code completion tools or chatbots, though three participants used AI tools rarely or not at all, stating that they did not find them useful. 56% of developers had expertise in developing static analysis tools, indicating that they were exceptionally knowledgeable about security.
Interviews: Each participant ran our tool on a code-base associated with a production application which they actively develop and shared their perspective guided by both structured and free-form questions. We first asked the participants to run our tool on a simple web server containing a known vulnerability to introduce the features of our tool: detection, fix, and chat. Then, we directed the participants to run the tool on security-critical areas of their application, such as web interfaces, API endpoints, database code, and file processing code.

We ran the study as a *think-aloud empirical study* [31, 44], meaning we asked developers to verbalized their thoughts while running the tool and processing the results. When participants explicitly asked questions, we provided help and answered questions to facilitate a smooth interview process and clarify the participant's statements, e.g., about the meaning of different UI elements, bug type descriptions, or behavior of the tool, but we refrained from explaining the results of the tool or interpreting the meaning of its outputs to avoid biasing the study. Each interview lasted approximately 50 minutes, consisting of a 10-minute setup and demographic survey, average 28 minutes usage of the tool and 12 minutes post-usage survey and discussion. We interviewed all subjects over video calls, and with their full consent, recorded field notes and demographic, audio, screen-capture, survey, and tool usage data.

After they used the tool, we asked participants about various aspects of the tool: (1) their overall perception of the usefulness of the detection alerts and suggested fixes and their satisfaction with the speed (Q1-Q3, reported on a Likert scale from 1 to 5 from "not useful/satisfied at all" (1) to "very useful/satisfied" (5)); (2) whether the tool fits their workflow, whether they trust in the tool, whether the reported alert types were relevant, and whether they would keep using the tool (Q4-Q7, reported as Yes/No). We also asked the participants what features they found especially useful and what features they would like to see in the tool. We asked the questions verbally during the interview, immediately after trying the tool, in order to collect free-form feedback on each question and ensure that the participant could recollect their experiences with the tool.

We designed the initial set of interview questions, guided by our research questions and informed/inspired by findings and open questions from previous studies of static analysis and AI tools [6, 15, 22, 27, 36, 46, 50]. Three authors tried the tool and all authors reviewed the survey questions, and we gathered feedback from outside researchers within our organization to improve the design of the planned questions.

Data Analysis Process: We analyzed the data quantitatively and qualitatively, reporting the results in Section IV.

To quantify users' perceptions of our tool, we tallied the responses to the post-interview survey, shown in Figure 7. Regarding questions Q1-Q3, we report the mean and distribution of Likert scores, and regarding Q4-Q7, we report the proportion of "Yes" responses. We also categorized each alert or fix that the participants examined during the interviews into "Useful" or one of 8 problem categories, based on the participant's explanation. We discuss the results in Section IV-A.

We conducted a *grounded-theory analysis* to analyze the study participants' rich free-form feedback [13], following the literature [15, 27, 28]. Grounded-theory analysis is a method used to analyze data by identifying recurring concepts, grouping these concepts into salient categories, and developing themes that provide an overall understanding of participants' perceptions of the tool. These concepts are derived from participants' quotations, reflecting their thoughts while using the tool and their responses to survey questions. All the resulting concepts and groups are referred to as a *codebook*.

We analyzed over 11 hours of usage and survey transcripts and identified a total of 161 codes in 12 distinct groups. Relevant codes are shown in Figure 8 and Figure 12. To create the initial codebook, the first and second authors independently analyzed two randomly selected interviews and generated lists of recurring concepts. They then met to create a unified list of concepts, create higher-level groups, and develop overall themes. Each author independently analyzed half of the remaining interviews, periodically syncing and jointly analyzing the same interviews to update the codebook and compare notes. Both raters agreed on all the classifications for alert responses. This was an iterative process [13], where we created the initial codebook after conducting the first 6 interviews and refactored/added groupings periodically as we conducted the remaining 11 interviews. We present our qualitative analysis in Section IV-B.

During the interviews, study participants suggested several features they felt would be useful, which provide useful recommendations for tool builders and directions for further research; we identify these as concepts in our grounded-theory analysis and discuss these feature requests in Section IV-C. The anonymized demographic data, interview and survey script, and codebook are in our data package [2].

IV. User study results

A. RQ1: Is DeepVulGuard useful in practice?

Detection: Figure 7 reports the results of our post-interview survey. On average, participants rated DeepVulGuard's alerts at 2.5 out of 5 for usefulness (Q1), with 2 participants giving it a rating of 4.5 or above and 3 participants giving it a rating of 1. Only 53% of participants felt that they trusted the tool's warnings about vulnerability alerts (Q5). **The biggest barrier to usefulness and trust in the tool's alerts was the amount of false positives**, with 30% of users explicitly reporting losing trust in the tool after frequently encountering false positives. The false positive rate in real-world settings was higher than in our SVEN dataset measurements (Section II-C). We attribute this difference to varying languages and vulnerability types: SVEN contains Python code, whereas most participants worked on Typescript or C# which comprise only 6% of our training data. Additionally, SVEN examples are intra-procedural, lacking information about the calling context and runtime environment, which may widen the gap between benchmark data and real-world testing.

76% of participants felt that the vulnerability types detected by the tool were relevant (Q6), with some participants expressing strong approval, for example: *"Definitely all of the all the categories of the vulnerabilities that were found here were good. They're all ones that hit these kinds of code all the time."*.

Fix suggestions: On average, participants rated DeepVulGuard's fix suggestions at 2 out of 5 for usefulness (Q1). 2 participants gave it a rating of 4, and 7 participants gave it a rating of 1. **For fixes, one of the most common issues was that the fix was not customized to the developer's codebase**, for example, creating a function to sanitize user inputs when the developer wants to reuse their existing sanitization library; this often prevented the users from directly applying the fix, requiring an overhaul to produce a fix with their intended approach. This highlights a limitation of common exact match or execution-based metrics for evaluating AI-based fixes, as these metrics do not capture the practical nuances of generating fixes for real-world codebases.

Speed: The average response time for the tool was 3.9 seconds per file. More than half of the participants were "very satisfied" with the speed of the tool, rating it at 5/5 (Q3). When asked about the tool's speed, one participant stated *"Totally satisfied.*

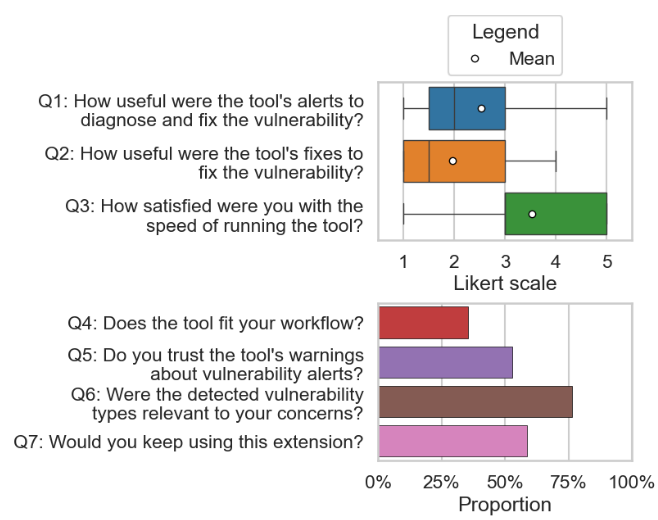


Fig. 7: Summary of participants' overall perceptions of DeepVulGuard, from our post-interview survey.

I can wait for this kind of stuff" (referring to security alerts + fixes).

Workflow integration: More than half (65%) of participants felt that the tool in its current state would not fit into their workflow (Q4). 14 out of 17 users expressed that **the tool would be more useful if it was running in the background and scanning their code while they were editing or ran along with their build or commit commands**; manually triggering the scan was a barrier to usage, since it required a stopping point in development.

Summary: Although not fully satisfactory, the tool shows promise — **59% of participants expressed that they would keep using the extension.**

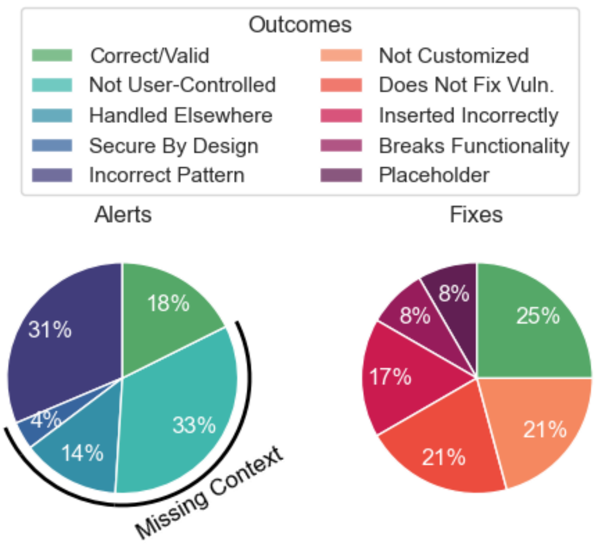


Fig. 8: Participant responses to LLM-filtered alerts and LLM-generated fixes while using DEEPVULGUARD.

Figure 8 reports the participants' responses to the 51 alerts and 24 fixes for which they provided direct feedback during the interviews, based on the categories assigned in our grounded-theory analysis. Here we display alerts from the combined CodeBERT + LLM model, excluding four participants who used the CodeBERT model. Participants considered 18% of alerts, and 25% of fixes, to be useful and without significant problems. Figure 9 shows an anonymized example of a URL redirection vulnerability which DEEPVULGUARD successfully detected and fixed by adding extra validation. In this case, DEEPVULGUARD added logic to check that the URL matches a list of approved domains and asked the user to fill the list of domains, based on their knowledge of the application's running context.

The primary causes of false positive alerts were missing context (totaling 51% of alerts) and incorrect pattern recognition (31%). Missing context involved misidentifying variables as user-controlled or overlooking vulnerabilities handled by the calling context or runtime environment. Incorrect pattern recognition involved misidentifying harmless patterns as vulnerabilities, such as constant strings mistaken for hard-coded credentials. We hypothesize that incorporating references to the calling context and runtime environment [32], along with in-context examples [53] of commonly misidentified patterns, into the LLM filter prompt shown in Figure 3 may help to address these limitations.

```
  function getUrl(string url) {
+   const allowedUrls = [
+     "example.com", // TODO: Provide allowed URLs
+   ];
    this.notificationService.notify({
    // DeepVulGuard: potentially malicious URL
    ↪  redirection.
-   action: () => { window.open(url); }
+   action: () => {
+     if (allowedUrls.includes(url))
+       window.open(url);
+     }
    )
  }
```

Fig. 9: A vulnerability that DEEPVULGUARD successfully found and fixed by adding validation logic to ensure that an attacker cannot redirect the user to a malicious third-party site.

Figure 10 shows an example of a "Not User-Controlled" outcome. This C# function's purpose is to redirect the user to a URL retrieved from a database. DEEPVULGUARD predicted that the field `resolvedPage.PageURL` could be user-controlled and therefore redirect the user to a malicious third-party website. However, in context, the developer knew that this URL is retrieved from an internally-controlled database, so the URL cannot be overridden by attackers.

```
public HttpResponse RedirectToPage(int pageId) {
  var resolvedPage = Database.LookupById(pageId);
  // 21 lines redacted...
  // DeepVulGuard: potentially malicious URL
  ↪  redirection.
  return Redirect(resolvedPage.PageURL);
}
```

Fig. 10: An example of a false-positive alert caused by missing context. The URL is non-malicious because it is retrieved from an internally-controlled database, not from user input.

21% of fixes were rejected because they were not customized to the user's codebase – they did not incorporate existing functions in the project (e.g. sanitization) or didn't comply with project style and linting rules, and thus could not be directly applied. Additionally, 21% did not address the underlying vulnerability, another 17% incorrectly inserted code generated by the LLM, resulting in syntax or indenting issues, and 8% of fixes mitigated the issue but broke existing functionality. 8% were placeholders containing instructional comments rather than functional fixes.

```
  app.delete("/room/:roomid", (req, resp) => {
    let roomId = request.params.roomid;
    // DeepVulGuard: potential SQL or script
    ↪ injection.
-   removeRoom(roomId);
+   roomId = sanitizeInput(roomId);
+   removeRoom(roomId);
  });
+ // Sanitization routine generated by DeepVulGuard
+ function sanitizeInput(input) {
+   return input.replace
+     /<script.*?>.*?<\/script>/gi, ""
+   );
+ }
```

Fig. 11: An example of a non-customized fix. In this case, the user would prefer simpler validation, such as checking that `roomId` is a number, or reusing their project's existing sanitization routines.

Figure 11 shows an example of a "Non-Customized" fix generated by DEEPVULGUARD. This TypeScript endpoint is intended to look-up a room by its numeric ID and remove it from the database. DEEPVULGUARD raised a valid concern that the user input could contain malicious values that would allow the user to perform destructive privileged actions, such as a SQL injection. However, the developer would have preferred to simply validate that `roomId` is numeric, which would catch all possible database or script injections, or reuse one of their project's existing sanitization routines; they stated that this would make the fix easier to understand and maintain.

B. RQ2: Which aspects of vulnerability detection + fix tools are most useful?

Figure 12 summarizes the participants' comments about different components of detection and fix tool components, based on the participants' think-aloud feedback while using DEEPVULGUARD, which we categorized in our grounded-theory analysis. Based on their responses, fix suggestions and confidence scores seem to be the most useful aspects of the tool, with 44% and 50% positive feedback respectively.

Fix suggestions: Fixes that have the correct initial approach allow the user to apply them with minimal changes. Out of the comments on fixes, 21% noted that the fixes could be applicable, with 7% of these noting that the fixes required minor changes such as changing variable names or error messages.

Fixes gave developers insight on vulnerabilities and best security practices. Beyond mitigating the vulnerability, 14% of comments noted that seeing the diff between 'bad' and 'good' code helped them understand the root cause of the issue. Fixes can also provide guidance on secure best practices when developers are working on unfamiliar code (7%). One developer said, *"If I knew I was starting in an area I wasn't very familiar on the most secure practices, it would be very helpful."*

However, some fixes lacked context (39%) or were more complex than the user's ideal solution (9%). Many security issues require multi-site edits, either when changing the semantics of a shared function or when encoding or decoding data; our tool is currently limited to fixing one function at a time, so the fix can lack context, which can result in breaking functionality, expressed by one participant as follows: *"This is how [the fix] should be defined, but then I would have to go find all the places where it's used and fix them all. And that might be a lot of places."* Incorporating additional context, such as the list of functions which call the function to be changed, could help provide more, contextualized fixes.

Placeholder fix suggestions were less preferred for users who were expecting functional fixes. LLMs occasionally generate placeholder code by default, including containing instructional comments rather than functional fixes. Some users did not find these useful, constituting 9% of comments on fixes, such as *"Well, that's not helpful"* and *"It's not really adding anything to the output"*. Since placeholder fixes can negatively impact users who prefer functional fixes, it's important to set expectations; a potential improvement could be to re-generate the fix when placeholders are initially generated, and if a functional fix cannot be provided, the placeholder fix should be accompanied by an explanatory message.

Chat interactions added value by allowing developers to iterate on fixes. In one case where the fix used the wrong approach at first, the developer iterated on the fix by first suggesting a different approach and then specifying their style guidelines, and arrived at a fix which they would apply without having to write the code themselves, saying afterwards, *"I like that this is conversational and I could do a few more rounds of interaction to understand what could be alternative solutions or better ways to approach this issue besides the initial suggestion"*. Before the chat feature was implemented, 50% of participants expressed the desire to ask the chatbot for more information about alerts or to suggest modifications to fixes. Recent research supports the potential usefulness of chat interactions. Nam et al. [36] found that developers completed more coding tasks within a given time when using a chatbot for code explanations compared to using a search engine.

Confidence score: **Users overwhelmingly used the confidence score to rank issues by importance.** This interaction constituted 44% of user feedback on this feature; an additional 6% noted that the confidence score was helpful for understanding the model's prediction. The confidence score can be useful to rank issues, but should be displayed to the user with full understanding of its meaning; 39% of feedback indicated that participants were unclear about the meaning of the score. We hypothesize that integrating the severity score [40] with the model confidence score will make it more useful for prioritizing vulnerabilities. Finally, a high-confidence result which is a false-positive can degrade the trust in the tool, as seen in 11% of feedback; therefore, expectations should be managed.

Vulnerability Explanations: Explaining the vulnerability and security best practices can be useful for providing compre-

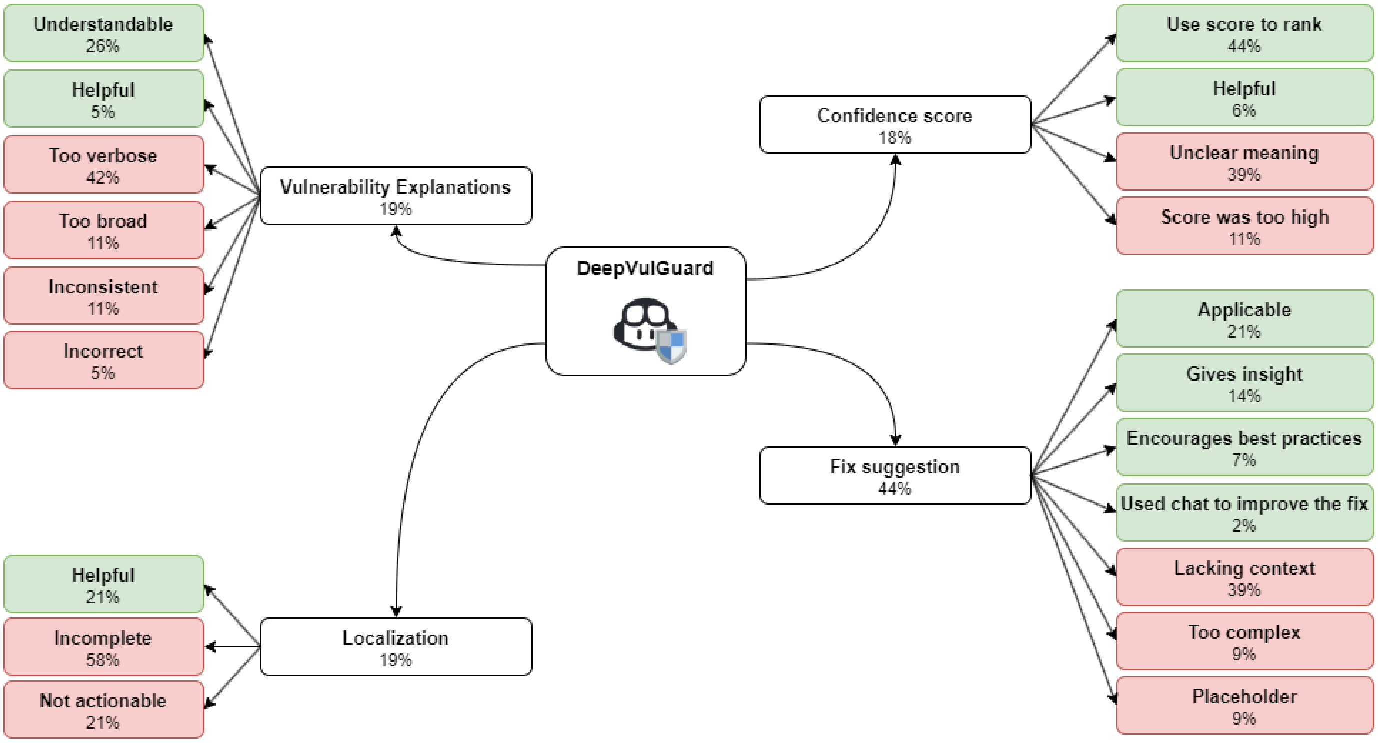


Fig. 12: Participants' in-use feedback on the aspects of DEEPVULGUARD. The aspects of the tool are organized into trees, where the leaf nodes are categories of comments from participants and the intermediate nodes are groupings of each category. Percentages show what portion of the comments on their respective aspects that each category constitutes; positive comments are green and negative comments are red.

hensive understanding of a vulnerability; 35% of feedback was positive, indicating that the explanations were understandable and helpful. One developer compared with existing static analysis tools: *"Normally with a static analysis tool, if I get an error that I'm a little unsure on, I would have to go out to a website of track down [an explanation], so providing me a diff and some text here explained it a bit."*

With explanations, brevity and adding visual annotations are important. 42% of feedback mentioned that the alert descriptions were too verbose; 11% mentioned that the verbiage was too broad to be useful. To quote one developer, *"If I see the code, it says sanitized input then OK, so I need to sanitize it... I would be more comfortable looking at the fix to know what the issue it is detecting, than read the verbose text."* Later they stated, *"I usually read one or two lines and then I stopped there. If it is too verbose, I probably don't pay too much attention."*. Another developer noted, *"Rather than giving me a wall of text, it would be great if it gave me bad and good examples."* One suggestion from this participant is to visually annotate the explanation by presenting labels with short, recognizable names, such as *Path Injection*, and allow the user to read the full explanation if they are interested.

Users expect the tool's outputs to be consistent, which introduces challenges when integrating LLM explanations and chat. 11% of feedback on explanations noted inconsistencies between the explanation and subsequent fixes. For example, one user noted that an alert's explanation specified not to use an insecure hashing function `btoa`, while the suggested fix used this function. While this specific issue could be solved by simply including the LLM-generated explanation in the fix prompt, the general issue is important for tool builders to be aware of.

Localization: **Users preferred highlights on a complete line or variable/string/function call.** DEEPVULGUARD highlights the tokens which were localized by the model, which may not necessarily align to semantic boundaries. 21% of participant feedback noted that localizations was helpful, especially the ability to zoom to a vulnerability' location. However, 58% of feedback noted that the localization seemed incomplete because it only highlighted part of the structure it was referencing. This behavior is by-design since the underlying model was specifically trained to only flag parts of the code that contributed to the vulnerability. However, this was one reason that users lost trust in our tool; to quote one user, *"The fact that the squiggle starts part way through a word made me wonder – Oh, is it just on the wrong line, or maybe got some wires crossed somewhere?"*. Our model detects vulnerable code patterns at the fault location. **In 21% of feedback, users noted that they would prefer to see an alert in a more actionable location – the root cause, or source for input validation issues, rather than at the fault location**; one user stated, *"Preferably, I'd actually do that check way before this,* *either where we download or where we extract, and that way we know that when we get here, this path is already sanitized."*.

C. RQ3: What features do developers want from vulnerability detection + fix tools?

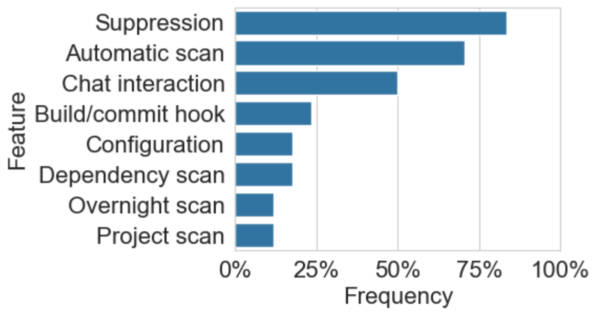


Fig. 13: The relative frequency of features suggested by the study participants.

Figure 13 displays the feature suggestions from the study, identified through our grounded-theory analysis. We implemented basic versions of some highly requested features from early interviews, specifically alert suppression and basic chat interaction. We measured the frequency of these requests from the participants who lacked these features.

Chat interaction was a frequently requested feature. Before implementing the chat feature, 3 out of 6 participants wanted a chat interface to better understand alerts, view examples of inputs that triggered them, and refine suggested fixes. Users who had chat available found it useful; for example, one user asked for alternative suggestions to address an alert and expressed, *"Yeah, these are good ideas. This would seed some ideas for me"*.

Starting the scan manually was a major disruption to developers' workflow. 12 out of 17 participants expressed that they would prefer if the tool scanned their code in the background and reported problems while they were editing, and 3 participants said it would be useful to trigger scans through a build or commit hook. One developer explained as follows: *"From a workflow perspective, I don't typically reach a point where I say, 'Oh, I'm just gonna stop now and go check for security issues'."*

Before we implemented the suppression feature, 5 out of 6 participants wanted the ability to suppress irrelevant alerts. One user specifically mentioned that the tool's outputs would be more manageable if they could filter out alerts identified as false positives or those frequently incorrect or irrelevant to their use case. Participants suggested several other enhancements, such as options for long-running overnight scans, click-to-scan UI interactions, the ability to scan entire directories or projects and their dependencies, and team-shared configurations sensitivity and rulesets.

V. Discussions

Based on our study, we find that current SOTA AI vulnerability detection and fix tools are not yet satisfactory, but developers remain eager to continue using them. This motivates us to continue researching and improving deep learning based methods and tools of vulnerability detection and fix. We have highlighted key lessons from our study in bold in Section IV. Here, we expand on several aspects of evaluating and deploying AI detection and fix tools.

Our detection results show a substantially higher false positive rate in real-world deployment compared to benchmark tests. Our study indicates it is impractical to (re)train a model for every new code base due to the lack of labeled data. While current test data for AI models often come from the same projects as the training data, in real-world scenarios, AI models are typically applied to unseen projects. Additionally, current deep learning models handle one function at a time [45], including DeepVulGuard; however, we see that many vulnerabilities in real-world code are related to multiple functions and the runtime environment. Lacking such program and environment context led to 51% of our tool's alerts being identified as false positives, as shown in Figure 8. This result highlights the need for vulnerability benchmarks that incorporate more realistic contextual information.

We also see that developers usually have specific definitions of "false positive" tailored to their own codebase and deployment scenario. For instance, one user dismissed a warning about sensitive data in an SSH key file, citing feasibility issues despite acknowledging the vulnerability: *"This is a problem, but I don't think there's anything they can do about it... I mean, it is a vulnerability – but if somebody can obtain access to the file system, then they have access to all kinds of password files."* These user-specific assumptions are difficult to incorporate into dataset labels, emphasizing the need for holistic evaluations in realistic development scenarios. To improve AI to predict likely feasible bugs, we may need to construct datasets using bugs with reproducible exploits rather than potential (but possibly infeasible) vulnerabilities from CVEs.

We found that 21% of suggested fixes, though functionally-correct (i.e. they would pass unit tests), were rejected because they were not customized to the user's code-base. Current test execution-based benchmarks [9, 29] do not capture this critical issue, highlighting the need for more realistic evaluations that consider this aspect of fix suggestions.

Based on our study, we make several recommendations for deploying AI detection and fix tools. First, when we deployed multiple models for detection, explanations, and fix, we need to ensure that the outputs of these models are consistent, so we do not confuse users. Second, LLM-generated explanations were often too verbose, suggesting a need to guide the LLM to generate concise output and code examples, and to add visual annotations. Third, users prioritize issues based on the displayed score, and this score should reflect important aspects such as severity, not just the confidence score.

VI. Threats to Validity

Since our work is an empirical study, there may be limits to the generalizability of its findings [26].

External and internal validity: Our sample of 17 developers from Microsoft may not fully represent all software developers' opinions. Research estimates that 16 users are typically sufficient to fully understand the challenges users face when using a tool [38]. This aligned with our findings, as the final two rounds of five interviews each added only 5 and 2 new codes respectively, indicating saturation. We recruited 11 more developers than a similar user study [22]. We included developers with experience ranging from 3 to over 30 years, covering projects in four programming languages and including backend, frontend, and IDE extension code.

Following the literature's recommendation to test iteratively with small user groups [37], we first tested with three participants, added key features to our tool like *directory scan*, *click-to-scan*, and *LLM filter*, then tested with three more users before adding *chat functionality* and *alert suppression*, and finally included the remaining eleven participants. We account for the developing feature set in Figure 13 and report only the LLM filter model responses in Figure 8.

We studied one set of models and 27 vulnerability types, which may limit generalization to other models and vulnerability types. The focus of our study was on understanding the practical usefulness of DL models in the IDE, so we chose to study one set of SOTA models (validated in Section II-C). Our tool supports the top 25 CWEs [49], plus the most frequent vulnerability types CodeQL detected in our dataset. Future work could study more models and vulnerability types.

Construct validity: We used think-aloud interviews, discussed in Section IV-B, which may result in users providing personal preferences rather than real system issues [41]. We addressed this by using grounded-theory analysis to assess the users' objective verdicts on root causes of alerts and fixes (Figure 8) and quantify the support for each feedback category (Figure 12).

VII. Related Work

Deep learning for vulnerability detection and fixing: Recent research has explored various DL methods for vulnerability detection, including graph neural networks (GNNs) [11, 33, 48] and transformer models [12, 21, 22, 54]. GNNs typically require complete source code to generate the necessary abstract syntax trees (ASTs) and control flow graphs (CFGs), which limits their effectiveness on incomplete code snippets. Our model is based on the state-of-the-art approach from Chan et al. [12], which optimizes for both in-IDE latency and incomplete code snippets. Compared to Fu et al. [22], which uses three separate models for localization, type, and severity prediction, we fine-tune our model to predict the presence, location, and type of vulnerabilities in a single forward pass, enhancing both simplicity and efficiency. Additionally, we introduce a novel LLM-based filtering technique that improved our model's precision by 20%; this is compatible with Fu et al. [22]'s approach. We also integrate SOTA LLMs for fix suggestions and show that they perform on par with existing DL and APR tools in Section II-C.

Benchmark studies of AI detection + fix models: Several empirical studies of DL models corroborate our results in underscoring the need for user studies in realistic scenarios. Chakraborty et al. [11] found that DL models often face issues with data duplication and unrealistic distributions of vulnerable classes. Chen et al. [14] showed that existing models have difficulty generalizing to unseen projects, but increasing the volume of training data can improve their generalization. Steenhoek et al. [47] demonstrated that while some models perform well on benchmarks matching their training data, they may struggle to generalize to new projects and bug types. Recently, Ding et al. [17] indicated that current benchmarks may overestimate the performance of deep learning models.

User studies of static analysis and AI tools: We developed our tool based on findings and recommendations from several user studies of traditional static analysis tools [15, 27, 46] (see Section II-A for more details). A controlled study by Fu et al. [22] involving six software practitioners demonstrated that DL detection & fix tools can be beneficial. They also surveyed 21 practitioners about the usefulness of features like localization, type and severity prediction, and fix suggestions, which informed our tool's design. To our knowledge, we are the first to conduct a study with professional developers on projects they own in a real-world deployment setting. There has been work on investigating whether APR tools are useful in practice. Surveys showed that most software practitioners prefer manual bug fixes over current APR tools due to unreliable and slow patch production [34, 39, 51]. Campos et al. [10] deployed APR in the IDE in a controlled study with 16 developers on a given project; APR increased developers' speed but may impact maintainability. We studied deep learning tools in a real-world development scenario where professional developers run the tool on their own projects. The tool is fast and can improve fixes via conversations with developers.

VIII. Conclusions

Recent research has introduced various deep learning vulnerability tools with promising benchmark performance. However, there has been no extensive user study on their real-world utility. To address this, we conducted a comprehensive user study with 17 professional developers, analyzing 24 projects, 6.9k files, and over 1.7 million lines of code, generating 170 alerts and 50 fix suggestions. Our study revealed that while current models show promise, they are not yet practical for everyday use due to challenges with (1) false positives caused by missing code context and incorrect pattern recognition, and (2) fixes which were not customized to the codebase. Based on user feedback, we make several recommendations for aligning model evaluations with real-world development scenarios, and for deploying models in practice.

Through our user study, we identified several areas for further research. One direction is to support automatic code scanning and address questions such as when to scan and how much code context to include. Another direction is to further develop AI powered chat interaction. Our preliminary chatbot implementation showed useful for explaining vulnerabilities and generating fixes. Future research should also resolve consistency issues among AI models' outputs.

IX. Acknowledgments

This research was partially supported by the U.S. National Science Foundation (NSF) under Awards #1816352 and #2313054. We thank Drs. Sarah Fakhoury, Christopher Bird, Denae Ford, and Thomas Zimmerman for their insightful discussions and valuable contributions to the validation of the user study design. We are also deeply appreciative of the participants of the user study for their time and input, which were essential to this research.

References

[1] GitHub Copilot. https://github.com/features/copilot.

[2] The data package for our study. https://doi.org/10.6084/m9.figshare.26367139, 2024.

[3] List of data breaches - Wikipedia. https://en.wikipedia.org/wiki/List_of_data_breaches, 2024.

[4] W. Baziuk. BNR/NORTEL: path to improve product quality, reliability and customer satisfaction. In *ISSRE*, 1995. doi: 10.1109/ISSRE.1995.497665.

[5] Al Bessey, Ken Block, Ben Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles Henri-Gros, Asya Kamsky, Scott McPeak, and Dawson Engler. A few billion lines of code later: using static analysis to find bugs in the real world. *Communications of the ACM*, 2010. doi: 10.1145/1646353.1646374.

[6] Christian Bird, Denae Ford, Thomas Zimmermann, Nicole Forsgren, Eirini Kalliamvakou, Travis Lowdermilk, and Idan Gazit. Taking Flight with Copilot: Early insights and opportunities of AI-powered pair-programming tools. *ACM Queue*, 2023. doi: 10.1145/3582083.

[7] Barry W. Boehm. *Software Engineering Economics*. Springer Berlin Heidelberg, 2002. ISBN 978-3-642-59412-0.

[8] Quang-Cuong Bui, Ranindya Paramitha, Duc-Ly Vu, Fabio Massacci, and Riccardo Scandariato. APR4Vul: an empirical study of automatic program repair techniques on real-world java vulnerabilities. *Empirical Software Engineering*. doi: 10.1007/s10664-023-10415-7.

[9] Quang-Cuong Bui, Riccardo Scandariato, and Nicolás E. Díaz Ferreyra. Vul4J: A dataset of reproducible java vulnerabilities geared towards the study of program repair techniques. In *MSR*, 2022. doi: 10.1145/3524842.3528482.

[10] Diogo Campos, André Restivo, Hugo Sereno Ferreira, and Afonso Ramos. Automatic program repair as semantic suggestions: An empirical study. In *ICST*, 2021. doi: 10.1109/ICST49551.2021.00032.

[11] Saikat Chakraborty, Rahul Krishna, Yangruibo Ding, and Baishakhi Ray. Deep learning based vulnerability detection: Are we there yet? *IEEE Transactions on Software Engineering*, 2021. doi: 10.1109/TSE.2021.3087402.

[12] Aaron Chan, Anant Kharkar, Roshanak Zilouchian Moghaddam, Yevhen Mohylevskyy, Alec Helyar, Eslam Kamal, Mohamed Elkamhawy, and Neel Sundaresan. Transformer-based vulnerability detection in code at edit-time: Zero-shot, few-shot, or fine-tuning?, 2023. URL https://arxiv.org/abs/2306.01754. arXiv: 2306.01754.

[13] Kathy Charmaz. *Constructing grounded theory: A practical guide through qualitative analysis*. Sage, 2006. ISBN 0761973532.

[14] Yizheng Chen, Zhoujie Ding, Lamya Alowain, Xinyun Chen, and David Wagner. DiverseVul: A new vulnerable source code dataset for deep learning based vulnerability detection. In *RAID*, 2023. doi: 10.1145/3607199.3607242.

[15] Maria Christakis and Christian Bird. What developers want and need from program analysis: an empirical study. In *ASE*, 2016. doi: 10.1145/2970276.2970347.

[16] Maria Christakis and Christian Bird. What developers want and need from program analysis: an empirical study. In *ASE*, 2016. doi: 10.1145/2970276.2970347.

[17] Yangruibo Ding, Yanjun Fu, Omniyyah Ibrahim, Chawin Sitawarin, Xinyun Chen, Basel Alomair, David Wagner, Baishakhi Ray, and Yizheng Chen. Vulnerability detection with code language models: How far are we?, 2024. URL https://arxiv.org/abs/2403.18624. arXiv: 2403.18624.

[18] Dino Distefano, Manuel Fähndrich, Francesco Logozzo, and Peter W. O'Hearn. Scaling static analyses at facebook. *Communications of the ACM*, 2019. doi: 10.1145/3338112.

[19] Steve Easterbrook, Janice Singer, Margaret-Anne Storey, and Daniela Damian. *Selecting Empirical Methods for Software Engineering Research*. Springer London, 2008. ISBN 978-1-84800-044-5.

[20] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. CodeBERT: A pre-trained model for programming and natural languages. In Trevor Cohn, Yulan He, and Yang Liu, editors, *EMNLP Findings 2020*, 2020. doi: 10.18653/v1/2020.findings-emnlp.139.

[21] Michael Fu and Chakkrit Tantithamthavorn. LineVul: A transformer-based line-level vulnerability prediction. In *MSR*, 2022. doi: 10.1145/3524842.3528452.

[22] Michael Fu, Chakkrit Tantithamthavorn, Trung Le, Yuki Kume, Van Nguyen, Dinh Phung, and John Grundy. AIBugHunter: A Practical Tool for Predicting, Classifying and Repairing Software Vulnerabilities, 2023. URL http://arxiv.org/abs/2305.16615. arXiv:2305.16615.

[23] Jingxuan He and Martin Vechev. Large language models for code: Security hardening and adversarial testing. In *ACM CCS*, 2023. URL https://arxiv.org/abs/2302.05319.

[24] Watts S. Humphrey. *A Discipline for Software Engineering*. Addison-Wesley Longman Publishing Co., Inc., 1995. ISBN 0201546108.

[25] IBM. Cost of a data breach 2024. https://www.ibm.com/reports/data-breach, 2024.

[26] Andreas Jedlitschka, Marcus Ciolkowski, and Dietmar Pfahl. *Reporting Experiments in Software En-*

gineering. Springer London, 2008. doi: 10.1007/978-1-84800-044-5_8.

[27] Brittany Johnson, Yoonki Song, Emerson Murphy-Hill, and Robert Bowdidge. Why don't software developers use static analysis tools to find bugs? In *ICSE*, 2013. doi: 10.1109/ICSE.2013.6606613.

[28] Brittany Johnson, Christian Bird, Denae Ford, Nicole Forsgren, and Tom Zimmermann. Make your tools sparkle with trust: The PICSE framework for trust in software tools. In *ICSE SEIP*, 2023. doi: 10.1109/ICSE-SEIP58684.2023.00043.

[29] René Just, Darioush Jalali, and Michael D. Ernst. Defects4J: a database of existing faults to enable controlled testing studies for java programs. In *ISSTA*, 2014. doi: 10.1145/2610384.2628055.

[30] Huangm Keman, Xiaoqing Wang, William Wei, and Stuart Madnick. The Devastating Business Impacts of a Cyber Breach. https://hbr.org/2023/05/the-devastating-business-impacts-of-a-cyber-breach, 2023.

[31] Clayton Lewis. *Using the "thinking-aloud" method in cognitive interface design*. IBM TJ Watson Research Center, 1982.

[32] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela. Retrieval-augmented generation for knowledge-intensive nlp tasks, 2021. URL https://arxiv.org/abs/2005.11401. arXiv: 2005.11401.

[33] Yi Li, Shaohua Wang, and Tien N. Nguyen. Vulnerability detection with fine-grained interpretations. In *FSE*, 2021. doi: 10.1145/3468264.3468597.

[34] Fairuz Nawer Meem, Justin Smith, and Brittany Johnson. Exploring experiences with automated program repair in practice. In *ICSE*, 2024. doi: 10.1145/3597503.3639182.

[35] Microsoft. Visual Studio Code - Code Editing. Redefined. https://code.visualstudio.com/, 2024.

[36] Daye Nam, Andrew Macvean, Vincent Hellendoorn, Bogdan Vasilescu, and Brad Myers. Using an LLM to help with code understanding. In *ICSE*, 2024. doi: 10.1145/3597503.3639187.

[37] Jakob Nielsen. Why you only need to test with 5 users. https://www.nngroup.com/articles/why-you-only-need-to-test-with-5-users/, 2000.

[38] Jakob Nielsen and Thomas K. Landauer. A mathematical model of the finding of usability problems. In *CHI*, 1993. doi: 10.1145/169059.169166.

[39] Yannic Noller, Ridwan Shariffdeen, Xiang Gao, and Abhik Roychoudhury. Trust enhancement issues in program repair. In *ICSE*, 2022. doi: 10.1145/3510003.3510040.

[40] NVD. NVD - Vulnerability Metrics. https://nvd.nist.gov/vuln-metrics/cvss, 2024.

[41] Liam O'Brien and Stephanie Wilson. Talking about thinking aloud: Perspectives from interactive think-aloud practitioners. *Journal of User Experience*, 2023.

[42] OpenAI, Josh Achiam, Steven Adler, et al. GPT-4 technical report, 2024. URL https://arxiv.org/abs/2303.08774. arXiv: 2303.08774.

[43] Caitlin Sadowski, Edward Aftandilian, Alex Eagle, Liam Miller-Cushon, and Ciera Jaspan. Lessons from building static analysis tools at Google. *Communications of the ACM*, 2018. doi: 10.1145/3188720.

[44] Carolyn B. Seaman. *Qualitative Methods*. Springer London, 2008. doi: 10.1007/978-1-84800-044-5_2.

[45] Adriana Sejfia, Satyaki Das, Saad Shafiq, and Nenad Medvidović. Toward improved deep learning-based vulnerability detection. In *ICSE*, ICSE '24, 2024. doi: 10.1145/3597503.3608141.

[46] Justin Smith, Lisa Nguyen Quang Do, and Emerson Murphy-Hill. Why can't Johnny fix vulnerabilities: A usability evaluation of static analysis tools for security. 2020. ISBN 978-1-939133-16-8. URL https://www.usenix.org/conference/soups2020/presentation/smith.

[47] Benjamin Steenhoek, Md Mahbubur Rahman, Richard Jiles, and Wei Le. An empirical study of deep learning models for vulnerability detection. In *ICSE*, 2023. doi: 10.1109/ICSE48619.2023.00188.

[48] Benjamin Steenhoek, Hongyang Gao, and Wei Le. Dataflow analysis-inspired deep learning for efficient vulnerability detection. In *ICSE*, 2024. doi: 10.1145/3597503.3623345.

[49] The MITRE Corporation. CWE top 25 most dangerous software weaknesses. https://cwe.mitre.org/top25/, 2024.

[50] Ruotong Wang, Ruijia Cheng, Denae Ford, and Thomas Zimmermann. Investigating and Designing for Trust in AI-powered Code Generation Tools, 2023. URL http://arxiv.org/abs/2305.11248. arXiv:2305.11248.

[51] Emily Winter, David Bowes, Steve Counsell, Tracy Hall, Sæmundur Haraldsson, Vesna Nowack, and John Woodward. How do developers really feel about bug fixing? directions for automatic program repair. *IEEE Transactions on Software Engineering*, 2023. doi: 10.1109/TSE.2022.3194188.

[52] Yi Wu, Nan Jiang, Hung Viet Pham, Thibaud Lutellier, Jordan Davis, Lin Tan, Petr Babkin, and Sameena Shah. How effective are neural networks for fixing security vulnerabilities. In *ISSTA*, 2023. doi: 10.1145/3597926.3598135.

[53] Sang Michael Xie and Sewon Min. How does in-context learning work? A framework for understanding the differences from traditional supervised learning. https://ai.stanford.edu/blog/understanding-incontext/, 2022.

[54] Aidan ZH Yang, Claire Le Goues, Ruben Martins, and Vincent Hellendoorn. Large language models for test-free fault localization. In *ICSE*, 2024. doi: 10.1145/3597503.3623342.