FLAG: Finding Line Anomalies (in code) with Generative AI

Baleegh Ahmad
New York University

Benjamin Tan
University of Calgary

Ramesh Karri
New York University

Hammond Pearce
University of New South Wales

Abstract

Code contains security and functional bugs. The process of identifying and localizing them is difficult and relies on human labor. In this work, we present a novel approach (FLAG) to assist human debuggers. FLAG is based on the lexical capabilities of generative AI, specifically, Large Language Models (LLMs). Here, we input a code file then extract and regenerate each line within that file for self-comparison. By comparing the original code with an LLM-generated alternative, we can flag notable differences as anomalies for further inspection, with features such as distance from comments and LLM confidence also aiding this classification. This reduces the inspection search space for the designer. Unlike other automated approaches in this area, FLAG is language-agnostic, can work on incomplete (and even non-compiling) code and requires no creation of security properties, functional tests or definition of rules. In this work, we explore the features that help LLMs in this classification and evaluate the performance of FLAG on known bugs. We use 121 benchmarks across C, Python and Verilog; with each benchmark containing a known security or functional weakness. We conduct the experiments using two state of the art LLMs in OpenAI's code-davinci-002 and gpt-3.5-turbo, but our approach may be used by other models. FLAG can identify 101 of the defects and helps reduce the search space to $12-17\%$ of source code.

1 Introduction

Bugs occur in code when there is a discrepancy between a developer's intent and their implementation. These bugs can introduce security vulnerabilities or functional deficiencies. Finding them is a laborious process—though assisted tools exist, they typically will only work when programs are fully completed (or at least compilable), and will focus only on a subset of languages or bug classes. Hence, during the development process, developers and their teams should regularly check their work, and so approaches will combine manual review with automated testing [4], static analysis [8], and fuzzing [20] to help identify potential problems. Code must be checked against intentions that are captured intrinsically, explicitly in the case of artifacts like tests, assertions, or rules (e.g., queries when using CodeQL [18]) or implicitly (when executing code for crashes in fuzzing).

Given that instances of buggy code are infrequent relative to correct code (average industry code has been estimated as containing 0.5 and 25 bugs per 1,000 lines [24]), we assume that developer intent is mostly captured in source code and comments with occasional lapses that need to be found and dealt with. Imagine a developer who wants to review their code manually; if a program comprises only a few lines of code and comments, it is probably feasible for both novices and experts. As projects grow, this becomes increasingly challenging by virtue of scale. Finding ways to narrow down potential problem areas in code for review can help developers, especially under a time crunch. *Can the intent of the source code and comments be used to flag problems in the code, narrowing the extent of manual review needed?* To answer this, we investigate the use of Generative AI, specifically large language models (LLMs), to FLAG anomalies in code.

LLMs such as GPT-3 [5] and Codex [7] demonstrate significant capability for number of lexical tasks including code-writing. They produce outputs based on the continuation of their inputs—a kind of 'smart autocomplete'. These inputs can be existing code and comments, and the models will produce probablistically likely matching code. This raises an interesting possibility: if (a) buggy lines of code are a minority of those present, and (b) a majority of the code aligns with the intent of the author, is it plausible to use LLMs to measure if a given line of code is an outlier? If so, this is a strong indication that the line was unusual and potentially defective.

Using this intuition, we propose a novel approach where we use LLMs to generate alternative lines of code given existing code and comments; these alternatives are compared against the developer's code to identify potential discrepancies. Recent work has motivated the exploration of LLMs, both in terms of their impact on security (e.g., in vulnerability introduction [26] and user studies [33]) and their use for im-

proving security (e.g., bug repair [27]). This work provides complementary insights into whether LLMs can be used to help identify bugs both in the form of security vulnerabilities and functional deficiencies. Our **contributions** are as follows.

- We propose FLAG, a framework for the novel use of LLMs in the detection of bugs by comparing original code with LLM-generated code. The details of the techniques used in the tool are mentioned in Section 3.
- We explore features of source code and information from LLMs to classify code as buggy or not. Effectiveness of these features is analyzed by performing experiments for major LLMs in various modes across multiple languages in Section 5, with further discussion in Section 6.
- The tool and results are open-sourced at [32].

2 Background

In this section we discuss bug detection, how LLMs work and the information available in source code that LLMs can use. This conveys the motivation for why FLAG is needed before discussing the details of its implementation in Section 3.

2.1 Large Language Models (LLMs)

LLMs (including GPT-2 [30], GPT-3 [5], and Codex [7]) are based on the Transformer [40] architecture. They can be thought of as "scalable sequence prediction models" [7] capable of a wide range of lexical tasks. When provided with a prompt consisting of a sequence of tokens, they generate the most probable set of tokens to continue or complete the sequence, similar to an intelligent autocomplete feature. In this context, tokens refer to common sequences of around four characters long and are assigned a unique identifier within a user-defined vocabulary size. This byte pair encoding (BPE) [15] enables the LLMs to process a larger amount of text within their fixed-size input window. Thus most LLMs work over tokens rather than individual characters.

Once trained over suitable examples, an LLM can be used to fill in the body of a function based on its signature and/or comment [7]. In the case of the commercial models we choose to investigate, this corpus is made up of billions of lines of open-source code scraped from the internet (e.g. GitHub).

LLMs range in size and capabilities. In this work we evaluate 2 OpenAI LLMs, code-davinci-002 and gpt-3.5-turbo. code-davinci-002 is a Codex model that is optimized for code completion tasks. We use it in two completion modes, without and with suffixes, referred to as auto-complete and insertion, respectively. gpt-3.5-turbo improves on GPT-3 and can understand as well as generate natural language or code. We use it in two modes, without any instruction or with an instruction to generate the next line of code, referred to as auto-complete and instructed-complete respectively.

2.2 Role of comments

Comments in code are often ignored for static code analysis. This is because the quality of comments is highly variable, and they do not play a role in the actual functionality of the program. They are, however, a good source of documentation for the intended behavior of the program [36]. Just like a human uses comments to reason about the code, tools like LLMs also possess the same ability.

Some efforts have utilized comments in an attempt to detect weaknesses in code. The most relevant work is icomment [37] which utilizes Natural Language Processing to automatically analyze comments and detect inconsistencies between comments and source code. They reason that these inconsistencies could contain a bug because the comment was correct but corresponding code implementation was wrong. On the other hand, the code could be correct, but the comment was bad. icomment takes comments and forms rules for source code to pass. Failure of these rules are reported as inconsistencies. Another relevant work is @tComment [38] which focuses on Javadoc comments. They use the same insight of code comment inconsistencies to take source code files for Java to infer properties for methods and then generate random tests for these methods. Failure of these tests are reported as inconsistencies. FLAG takes the insight of comment code inconsistencies but instead of generating rules or tests, generates alternate code for comparison with original code. Moreover, FLAG uses previous code plus comments for its operation.

2.3 Bug detection

Static detectors are used in various shapes and forms across many software companies. They are typically used in the development stage before deployment to catch bugs in code. Google's Error Prone [29] catches common programming mistakes in Java. Facebook's Infer [19] does the same for Java and C/C++/Objective C code. And other well-known tools include SpotBugs [35] and CodeQL [18]. They typically work by conducting analysis over the Abstract Syntax Tree (AST) and/or data-flow graph of the code [16]. These constructs are traversed with an elaborate set of checkers that are used to indicate improper behavior of code e.g., CodeQL could be used to create a query that checks whether a path exists between two nodes that should not otherwise. This improper behavior could be a pattern in the AST or some flow in the data-flow graph. Detectors may also infer rules from version histories and source code comments. While static detectors are generalizable across databases of the same language, they require the creation of a large set of patterns and known buggy flows for detecting bugs. Moreover, they are only able to find bugs in this limited knowledge base of patterns and flows.

Unit testing is the other methodology commonly used in the struggle to identify defects in code [14, 31]. The obvious challenge with unit tests is the strict requirement for knowl-

edge of the functionality of the program. The workaround employed is the development of automatically generated unit tests. Code coverage through this approach remains limited, and even when the coverage is there, the faults are sometimes not revealed [34]. Additionally, unit tests do not cater to affirming the security of the code.

2.4 Machine Learning based Detectors

Researchers have used language model-based techniques to try and detect bugs. Bugram [41] uses N-gram Language models to obtain sequences for tokens in programs. These token sequences are explored according to their probabilities in the learned model, and the ones with lower probabilities were marked as probable bugs. Hoppity [13] is a learning-based approach relying on graph transformations to detect and fix bugs. The graph model of the source code is used to make a series of predictions regarding the position of bug nodes and corresponding graph edits to produce a fix. EnSpec [6] is a method that uses code entropy (a metric devised to represent the *naturalness* of code derived from a statistical language model) for bug localization. They use the intuition that buggy code tends to be higher in entropy to facilitate localization. In another work, solutions of bugs and a language model based on long short-term memory (LSTM) networks was used for bug detection [39]. The researchers of this work trained their model over Aizu Online Judge (AOJ) [3], which contains several million lines of source code.

More recently, LLMs have been explored for this purpose. The difference between LLMs and other language models is the size of training data and complexity of the network. FuzzGPT [11] uses LLMs as edge case fuzzers by priming LLMs to produce unusual programs for fuzzing. First, they allow LLMs to directly learn from historical reported bugs and then generate similar bug-triggering code snippets to find new bugs. They use Codex and CodeGen models to detect bugs in the popular DL libraries PyTorch and TensorFlow. In another work, Li et al. [21] show how ChatGPT can be used through differential prompting to detect bugs in the Quixbugs [22] database. This involves generating reference designs for a problem using ChatGPT. For a given test input, if reference designs produce the same output, but the buggy version produces a different one, the test case is identified as a failure-inducing test case. DeepBugs [28] uses natural language elements in code to implement a name-based bug detection machine learning tool. The key idea is to convert function names and identifiers into embeddings that are learned by the network in order to preserve semantic information. They applied their approach to a corpus of JavaScript files. While these efforts are a step forward in using LLMs to detect bugs, they either target a niche subset of code or require information like test cases for a program. The ability of LLMs to fix bugs is not evaluated in a generalizable way, i.e., using no information or knowledge outside of source code.

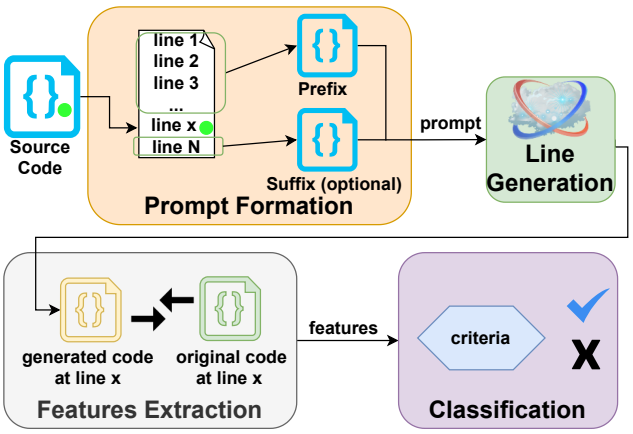


Figure 1: FLAG pipeline for Finding Line Anomalies.

2.5 Why FLAG?

The limitations mentioned for static detectors and machine learning detectors leave room for a tool that does not require huge efforts to set up security rules and tests and is applicable on many programming languages. Through our various experiments, we have demonstrated that our checking approach is agnostic to the language of code. Our insights can be used to leverage off-the-shelf LLMs that work on various languages for providing code feedback. FLAG also does not require code to be compiled or even syntactically correct, enabling it to work on incomplete code. This allows vulnerability checking at earlier stages of design when compared to traditional static checkers. Additionally, FLAG does not require creation of security rules and tests. This is a big plus because security checks are by nature non-exhaustive and require a lot of time and domain expertise to devise. This is particularly true for hardware where formal tools have been shown to fail in detecting a lot of RTL security bugs [12].

3 FLAG Approach

The FLAG consistency checker relies on the unique capabilities of LLMs for writing code. It has the approach shown in Figure 1. For each line in the source code, FLAG generates a corresponding prompt which consists of the part of code before the line (prefix) and optionally, the code after the line of code (suffix). The prompt consisting of code and comment is the input to the LLM which outputs a single line of code or comment. This generated line is compared to the original line to produce *features* that are used in the classification of the original line as buggy or not. These *features* provide either a quantitative estimate of the difference between the two lines or the degree of confidence of the line generated by the LLM. They are discussed in Section 3.3. The lines classified as potentially buggy are flagged to the designer. The FLAG flow for a given line has a 4-step process of prompt formation, line

```
1  // getValueFromArray safely returns an element from an
       array
2  int getValueFromArray(int* array,int size,int index);
3  int main(int argc, char *argv[]) {
4      int id_sequence[] = {24, 32, 10, 23, 11, 6, 42};
5      int id_index = atoi(argv[1]);
6      printf("%
7  ␣␣␣␣(id_sequence,7,id_index));
8  ␣␣␣␣return␣0;
9  }
10 int␣getValueFromArray(int*␣array,int␣size,int␣index){
11 //given␣the␣array␣of␣integers,␣return␣the␣value␣at␣the
       ␣given␣index␣and␣-1␣if␣the␣index␣is␣out␣of␣bounds
12     if(index < size) {
13         return array[index];
14     }
15 }
```

(a) Step 1: Traversing original file for LLM input. Target line:12 (pink). Prefix lines 1-11 (yellow) and optional suffix lines 13-15 (gray).

```
1  if (index >= 0 && index < size) {
```

(b) Step 2: Response by gpt-3.5-turbo in auto-complete mode. This is detected as different to the original line 12, which is thus flagged for inspection, where a defect (CWE-125, out-of-bounds read) is found.

Figure 2: LLM consistency check on line 12 of benchmark C1-10 (CWE-125) in source C1.

generation, feature extraction, and classification.

To begin the consistency checks on a file, we must give FLAG a line to start checking from. This is done to give the LLM enough context to start producing relevant code and comments. Normally, we give the starting line number after the header definitions, initial comments, and declaration of modules and internal signals. In some cases, when the defect is part of the declared signals, we make an exception to start from the beginning of the file. Additionally, the file is preprocessed to skip empty lines and to identify if a comment was present before the line to start checking from.

3.1 Prompt Formation

Prompt formation takes the source code and line to be classified as the input and produces the prompt as the output. This prompt is sent to the LLM for line generation. An example is shown in Figure 2a. The file shown is from source C1 for the defect containing CWE-125. The prompt generated for line 12 consists of the prefix covering lines 1-11 and the suffix covering lines 13-15. This process is repeated for each line in the file. As the FLAG checker progresses line by line, the prefix increases while the suffix decreases.

We limit prefix and suffix lengths to a maximum of 50 lines to stay within the token limits of the LLMs. The prompt is tweaked to elicit a better response from the LLM e.g., if the LLM generates a comment when it should be generating code, we append the first few characters of the original line of code to the prefix. This process may be repeated multiple times till a valid response is obtained and requires feedback from line generation. This is why it is discussed in Section 3.2. The suffix is used only when the model supports insertion mode.

Algorithm 1 LLM line generation

```
1:  procedure GENERATE(orig_lines, loc, num_lines )
2:  If loc > max_pre_len
3:      prefix ← orig_lines[loc-max_pre_len:loc]
4:  Else prefix ← orig_lines[:loc]
5:  If num_lines − loc > max_suf_len
6:      suffix ← orig_lines[loc+1:loc+max_suf_len]
7:  Else suffix ← orig_lines[loc+1:]
8:  max_attempts ←3, attempts ←0, generated ←False
9:  while generated==False
10:     Try
11:         If attempts > 0
12:             Append orig_lines[loc][:4] to prefix
13:             response ← Generation for given prefix and
                           suffix (optional) by LLM
14:     Except Note error
15:     Else
16:         If response==‘’ and attempts < max_attempts:
17:             Increment attempts
18:         Elif orig_lines[loc] is code and response is not
19:             Increment attempts
20:     Else generated ←True
21: end while
22: end procedure
```

3.2 Line Generation

Line generation takes the prompt as an input and outputs a line of code or comment generated by the LLM. For example, the line generated for the prompt in Figure 2a is shown in Figure 2b. In this instance, the response by LLM is different from the original line of code. It is flagged for inspection, revealing that the original line of code contained the security defect CWE-125, i.e., out-of-bounds read. The LLM used for this example was gpt-3.5-turbo in auto-complete mode. Details of experiments are in Section 5.

The LLM is guided to produce a reasonable output because sometimes the LLM may return an empty line or return a comment instead of code. This process is described in Algorithm 1. `orig_lines` is a list of the lines in the original file. Assume that the list is indexed starting 1. `loc` is the line number that we want the LLM to generate. `num_lines` is the total number of lines in the original file (after being preprocessed). We initialize the prefix and suffix as empty strings and use `max_pre_len` and `max_suf_len` as the limits of their sizes, respectively. This is done as an attempt to keep the token size of the prompt reasonable. For our experiments, we set the limits to 50. The assignment of appropriate content from the original file to prefixes and suffixes is shown in lines 2-7. To overcome occasional unusable outputs, we prompt the LLM to produce a response again, up to a maximum of

max_attempts times. This is shown in the Try and Catch block in lines 15-19. Lines 17 and 19 increment the number of attempts and return to the start of the Try block at line 10. On the first try, an LLM is prompted to produce a response with the given prefix and suffix. On the second and third attempts, we provide assistance to elicit a non-empty response. This is done by appending the first 5 characters of the line the LLM is attempting to generate to the prefix. This assist is shown in line 12. If there is any error in the try block, the error is noted in line 14 before returning to line 10. If there is no error and the conditions in lines 16 and 18 do not hold, line 20 is executed. This asserts the signal that a line has been successfully generated and the while loop is exited. For all completions, we use a temperature of 0, max_token limit of 150, top_p value of 1, and end line character as the stop token.

3.3 Feature Extraction

Feature extraction takes the original and buggy lines of code and outputs *features*. These are quantitative values that are used for the classification of the original line as potentially buggy or not. They represent the difference between the two lines or the confidence of the generated code.

3.3.1 Features

`Levenshtein Distance (ld)` is an edit distance between two strings. It considers three operations: addition, deletion, and substitution. The sum of the number of operations required to convert one string to another is the Levenshtein Distance. A perfect match results in a score of 0. We use `ld` to compare lines of code. Since the goal is to flag defects in code, we use `ld` as the primary criterion to identify defects.

`BLEU` i.e., Bilingual Evaluation Understudy Score [25], is a metric for evaluating a candidate sentence to a reference sentence. A perfect match results in a score of 1.0, whereas a perfect mismatch results in a score of 0.0. We use it for comparing comments as they resemble natural language. We collect the cumulative BLEU-1 to BLEU-4 scores but found that only BLEU-1 produced meaningful numbers. BLEU-2, BLEU-3 and BLEU-4 had minuscule quantities, which are not useful. For the rest of this paper, `BLEU` refers to BLEU-1.

`Distance from comment (dfc)` indicates how far a line of code is from the closest comment before it. If the line also contains a comment, `dfc` has the value 0. If there is no comment before the line, `dfc` has no value. We only consider comments before code as relevant to the code because that is typically how code is written.

`logprob` in the context of code generation by LLMs is the log of the probability of the token generated. If a token is more likely to be generated, it will have a higher `logprob` with its maximum possible value being 0. If a token is less likely to be generated, it will have a lower `logprob`, i.e., a negative value with higher magnitude. An LLM tends to "choose" a token with a higher probability. For a generated line, we take the average of the logprobs of the tokens generated, which we refer to as `logprob` from here on. `logprob` closer to 0 indicates more confidence in the generation, whereas a more negative value indicates lesser confidence.

3.3.2 Extraction

In order to obtain these *features* values, the original and generated lines are stripped to remove trailing white spaces. If either of the lines is a combination of code and comment, the code and comment are separated for comparison. The code of the original line is compared with the code of the generated line to calculate the `ld`. The comment of the original line is compared with the comment of the generated line to generate the `BLEU` metric. Additionally, if the original line is a comment, the location of the most recent previous comment is updated to the current line, and the `dfc` is calculated. The lines generated and *features* obtained, for example, in Figure 2, are shown in Table 1. For the example shown, LLM gpt-3.5-turbo was used in auto-complete mode. It is not possible to obtain `logprob` values for gpt-3.5-turbo through the public API so those values are not shown. `logprob` values are available for experiments with code-davinci-002.

3.4 Classification

Classification for a given file takes the *features* as inputs and selects lines to flag based on some conditions. These conditions are referred to as *criteria*, and the lines flagged are referred to as *reported_lines*. The conditions can be inclusive or exclusive. The inclusive conditions are designed to bring lines into the set of *reported_lines*. The exclusive conditions are used to eliminate lines from *reported_lines*. The inclusive conditions use two thresholds in our FLAG framework, i.e., Levenshtein Distance Upper Limit (`ld_limit`) and Distance from Comment Upper Limit (`dfc_limit`). These thresholds are chosen for reasons detailed below. The `ld_limit` and `dfc_limit` work together to yield different *criteria*.

Why `ld_limit`? `ld` indicates the difference between two pieces of code. If `ld` is 0, the pieces of code are identical. This means that the LLM does not have an alternative suggestion, so there is no reason to flag this code. If `ld` is greater than 0, the pieces of code are different, indicating that the LLM is indicating an alternative to the original line of code. This could be worthy of flagging. But if `ld` is a very high number, it could indicate that the LLM is generating something completely different. Based on the understanding that a buggy version of code is normally very similar to the fixed version, we use an upper limit to `ld` to target the generated code that is different but not too different.

Why `dfc_limit`? `dfc` indicates how far a line of code is from the closest comment before it. If there is a comment close to the code, i.e., `dfc` has a low value, we hypothesize

Table 1: Lines generated by gpt-3.5-turbo (auto-complete mode) and features extracted for benchmark C1-10(CWE-125). Line with true defect is shaded in pink. Lines flagged according to criterion C2(20,10) are marked with a check in **Flagged?** column. File shown partially for brevity.

Original	**Generated**	**dfc**	**ld**	**BLEU**	**Flagged?**
printf("%d\r\", getValueFromArray(id_sequence, 7, id_index));	getValueFromArray(id_sequence, sizeof(id_sequence)/sizeof(int),id_index));	6	67	–	
return 0;	return 0;	7	0	–	
}	}	8	0	–	
int getValueFromArray (int* array, int size, int index) {	int getValueFromArray(int* array, int size, int index){	10	1	–	
//given the array of integers return the value at the given index and -1 if the index is out of bounds	//if the index is out of bounds return -1	0		0.425	
if(index <size) {	if (index >= 0 && index <size) {	1	15	–	✓
return array[index];	return array[index];	2	0	–	
}	} else {	3	7	–	✓
}	else {	5	6	–	

that the LLM would generate code with relevant information, so we trust it more. If `dfc` is greater than `dfc_limit`, the comment is likely not relevant to the line of code, and we discard it. The use of `dfc` relaxes our criteria for selecting lines to flag as a line of code that has a `ld` greater than `ld_limit` may still be flagged if it has a `dfc` less than `dfc_limit`.

A simple criterion C0 uses the `ld_limit` as its threshold.

$$C0(ld\_limit) : 0 < ld <= ld\_limit$$

A more elaborate criterion C1 uses both thresholds,

$$C1(ld\_limit, dfc\_limit) :$$
$$0 < ld \; AND \; (ld <= ld\_limit \; OR \; 0 < dfc < dfc\_limit)$$

The final criterion C2 uses both thresholds and employs a *reduce_fp()* function to remove false positives.

$$C2(ld\_limit, dfc\_limit) :$$
$$0 < ld \; AND \; (ld <= ld\_limit \; OR \; 0 < dfc < dfc\_limit)$$
$$AND \; reduce\_fp()$$

The side-effect of detecting defects is a substantial number of false positives in *reported_lines*. We tackle this by adopting a few measures in *reduce_fp()*. The process inspects *reported_lines* to remove some of the flagged lines that are likely false positives. The first measure is recomputing `ld` after removing white spaces in generated and original lines. This removes false positives where the `ld` counted white spaces, e.g., `always(@posedge clk)` and `always (@posedge clk)`. The second is checking if the original line has just a keyword. If this is the case, that line is removed from *reported_lines* because a simple keyword can not be buggy. The third uses `logprob` values as thresholds for exclusion. If the LLM has a large negative `logprob` value for the generated line, it is removed from *reported_lines* as it indicates that the LLM is not confident in its suggestion. We use a threshold of < -0.5 for our experiments with code-davinci-002 to remove lines. Classification of the example shown in Figure 2 using C2(20,10) is shown in column `Flagged?` of Table 1. Of the 2 flagged lines, one contains the original defect (shaded in pink), while the other is a false positive.

4 Benchmark Dataset

To evaluate the feasibility of using LLMs to flag inconsistencies in the comment or code, we experiment on a set of example defects in C, Python, and Verilog from various sources, summarized in Table 2. We collect both security-related and functional defects. Sources C1, P1, and V1 consist of security defects, whereas sources C2, P2, and V2 have defects that pose a functional issue. Some of the security-related defects are described in Table 3, and the remaining security and functional defects are described in Appendix A. 19 of the 35 security defects have a weakness present in MITRE's top 25 CWEs list. Common Weakness Enumerations (CWE)s are categories of security issues that can manifest in code [9]. Each defect has a source code file and corresponding line number(s) that contains the defect. We gather 121 defects; 43 for C, 34 for Python, and 44 for Verilog.

4.1 C sources

C1 contains security vulnerabilities curated by the authors of [26, 27]. We chose 12 real-world CVEs and 5 instances of CWEs investigated in those works. Common Vulnerabilities and Exposures (CVE) is a glossary that classifies vulnerabilities. A particular exposure in the real world is documented and given an identification. We investigated cve-2018-19664, cve-2012-2806, cve-2016-5321, cve-2014-8128, cve-2014-8128, cve-2016-10094, cve-2017-7601, cve-2016-3623, cve-2017-7595, cve-2016-1838, cve-2012-5134 and cve-2017-5969. The details of each CVE could be obtained

Table 2: Sources for defects in C, Python, and Verilog. Func=Functional, Sec=Security

Source	Lang.	Type	What we used as defects/bugs	#Bugs
C1 [26, 27]	C	Sec	Insecure code instances from CVEs and CWEs that authors used to engineer repairs.	13
C2 [43]	C	Func	Defects in students' submissions for introductory programming assignments.	30
P1 [26, 27]	Python	Sec	Insecure code instances from CVEs and CWEs that authors used to engineer repairs.	12
P2 [42]	Python	Func	Bugs extrapolated from patch files for a subset of the bugs in this database.	22
V1 [1]	Verilog	Sec	Defects introduced in modules from various sources to evaluate their repair tool.	10
V2 [2]	Verilog	Func	Defects introduced in common Verilog modules by authors to evaluate their repair tool.	34

Table 3: An example selection of security-related bugs and descriptions (The full table is in Appendix A). (*) indicates that the CWE is in one of MITRE's top 25 CWEs list.

ID	CWE	Description
C1-1	125*	libjpeg-turbo 2.0.1 has a heap-based buffer over-read in the put_pixel_rows function.
C1-2	119*	Heap-based buffer overflow in libjpeg-turbo 1.2.0 allows remote attackers to cause a denial of service.
C1-3	119*	The DumpModeDecode function in libtiff 4.0.6 and earlier allows attackers to cause a denial of service.
P1-1	79*	Writing user input directly to a web page allows for a cross-site scripting vulnerability.
P1-2	20*	Security checks on the substrings of an unparsed URL are often vulnerable to bypassing.
P1-3	22*	Uncontrolled data used in path expression can allow an attacker to access unexpected resources.
V1-1	1234	Lock protection is overridden when debug mode is active.
V1-2	1271	A locked register does not have a value assigned on reset. When it is brought out of reset, the state is unknown.
V1-3	1280	An asset is allowed to be modified even before the access control check is complete.

from the NIST National Vulnerability Database [10]. For CWEs, we included cwe-119 and cwe-125, which were inspired by MITRE examples, and cwe-416, cwe-476, cwe-732, taken from CodeQL examples.

C2 contains defects in assignments submitted by students [43]. There were 74 unique assignments spread across 10 weeks. We selected 30 of these assignments with topics including Simple Expressions, Loops, Integer Arrays, Character Arrays (Strings) and Functions, Multi-dimensional Arrays, Recursion, Pointers, Algorithms (sorting, permutations, puzzles), and Structures (User-Defined data-types). For each of the chosen assignments, we picked a submission by a student which had a clearly identifiable bug. This was done by comparing the buggy version with the correct version in the repository posted by the authors of the work. Each unique assignment also contains a correct version of the code with comments at the start describing the intended functionality of the code. We use these comments and add them as prefix to the relevant source code files we analyze.

4.2 Python sources

P1 uses the work by authors to use Github Copilot to complete code based on scenarios developed for select CWEs [26, 27]. We look at completions marked as containing the weakness by the authors and selected one for each CWE for our work. We cover cwe-79, cwe-20, cwe-22, cwe-78, cwe-89, cwe-200, cwe-306, cwe-434, cwe-502, cwe-522, cwe-732 and cwe-798.

P2 has defects noted in real-world Python projects by analyzing their version control histories [42]. This was done by identifying commits on Github that intend to fix a bug. We chose one or two bugs identified in the youtube-dl, tqdm, fastapi, luigi, scrapy, black, nvbn, spacy, keras, pysnooper, cookiecutter and ansible projects. The defects cover a range of causes, including wrong regex expression, incorrect returned objects, incorrect split token used, incorrect assignment, incorrect parameter value, incorrect os command, incorrect function argument, incorrect computation, wrong error condition, missing file encoding, incorrect appended element to list, incorrect file path in os command and others. For each of these defects we isolate the source code file containing the bug and its location by analyzing the *bug_patch.txt* file for the bug.

4.3 Verilog sources

V1 contains defects in the form of security vulnerabilities gathered by the authors from 3 sources [1] . The first source is MITRE's hardware CWEs, from which they design 4 modules with the CWEs present (a locked register, a reset logic for register, an access checker, and a TrustZone peripheral). The second source is the OpenTitan SoC [23], where authors insert bugs in the RTL of 3 modules (ROM Controller, One Time Programmable Memory Controller, and interface for KMAC Key Manager). The third is bugs in Hack@DAC 2021 SoC [17] that uses the Ariane core with bugs present for the bug-hunting competition. The bugs are in the Control and Status Register Regfile, Direct Memory Access, and AES interface modules.

V2 contains functional defects in 11 Verilog projects [2]. These include a 3-to-8 decoder, 4-bit counter with overflow, t-flip flop, finite state machine, 8-bit left shift register, 4-to-1 multiplexer, i2c bidirectional serial bus, sha3 cryptographic

hash function, core for the Tate bilinear pairing algorithm for elliptic curves, Core for Reed Solomon error correction and an SDRAM controller. The defects include numeric errors, incorrect sensitivity lists, incorrect assignments, incorrect resets, blocking instead of non-blocking assignments, wrong increment of counter and skipped buffer overflow check.

5 Experiments and Results

To evaluate FLAG consistency checking approach, we design experiments using 2 OpenAI LLMs. The first LLM is code-davinci-002 which is optimized for code-completion tasks. For code-davinci-002, there is no system prompt, but it supports an insert capability if a suffix is given in addition to the prefix. We run 2 experiments with code-davinci-002, in auto-complete (without suffix) and insertion (with suffix) modes. The second LLM is gpt-3.5-turbo. It improves on GPT-3 and can understand and generate natural language or code. At the time of this manuscript, it was the "most capable GPT-3.5 model". It is an instructional model whose role is deducted based on the system prompt it is given. After the system prompt is given, the model is then given the message which specifies the task. We conduct two experiments with gpt-3.5-turbo. The first is done without any system prompt. We refer to this mode auto-complete. The second gives the following system prompt: *"You are a skilled AI programming assistant. Complete the next line of code."*. This mode is instructed-complete.

An experiment involves choosing a particular set of LLM and mode, e.g., code-davinci-002 in insertion mode, and using it to execute our consistency checking approach on all 121 benchmarks. In total, we run 4 experiments for the possible combinations of 2 LLMs and their 2 modes of completion. To evaluate the success of an experiment, we focus on 3 metrics. The first is the `Number of defects detected (DD)`. For a given set of inputs, `DD` is the sum of defects that were correctly identified. It is effectively the True Positives(TP) and has a maximum possible value of 121, i.e., total defects across all sources. The second is the `False Positive Rate (FPR)`. For a given set of inputs, `FPR` is the ratio between the number of lines incorrectly highlighted and the total number of lines. The third is the `True Positive Rate (TPR)` or `Recall` i.e., the ratio between the true positives and total number of positives. This is the `DD`/total number of defects.

5.1 Results

The results are presented in Table 4. We break down the results for each experiment by source and criteria to present a nuanced view. For C, we break down C1 into C1-CVEs and C1-CWEs because of the difference in their size and results by FLAG checker. By showing snapshots of different criteria, we illustrate how the results change based on complexity.

C1-1	C1-2 x x ✓ x	C1-3	C1-4 x x ✓ ✓	C1-5	C1-6
C1-7	C1-8	C1-9	C1-10	C1-11	C1-12 x x ✓ ✓
C1-13	P1-1 x x ✓ ✓	P1-2 ✓ ✓ ✓ x	P1-3	P1-4 x x x ✓	P1-5
P1-6	P1-7	P1-8 x x ✓ x	P1-9	P1-10 x x ✓ ✓	P1-11
P1-12 x x ✓ ✓	V1-1 x x ✓ x	V1-2	V1-3 ✓ ✓ x x	V1-4	V1-5
V1-6	V1-7 ✓ ✓ ✓ x	V1-8	V1-9 x x ✓ ✓	V1-10	

Figure 3: Security bugs detected according to modes and LLMs. A green cell indicates that LLMs in both modes were able to detect this bug. Conversely, pink indicates that no LLM in any mode was able to detect this defect. The remaining cells have a set of 4 symbols to represent whether the defect was detected by [code-davinci-002 in auto-complete, code-davinci-002 in insertion, gpt-3.5-turbo in auto-complete and gpt-3.5-turbo in instructed-complete]. A sequence of [✓ ✓ ✓ x] means that the defect was detected by code-davinci-002 in both modes and by gpt-3.5-turbo in auto-complete but not in instructed-complete e.g. P1-2.

Ultimately, we propose criterion 2 as it balances TPR and FPR and uses inclusive and exclusive *features* from Section 3.3.1.

gpt-3.5-turbo has a greater ability to detect defects, but code-davinci-002 has fewer false positives. For criterion 2, gpt-3.5-turbo performed better than code-davinci-002 in terms of TPR. It was able to detect 90 of the defects in auto-complete and 77 in instructed-insertion mode, while code-davinci-002 detected 76 in auto-complete and 78 in insertion. code-davinci-002 performs better in terms of FPR. It has a lower FPR of 0.121 in auto-complete and 0.141 in insertion compared to 0.172 and 0.181 for gpt-3.5-turbo in instructed-complete and auto-complete, respectively. The criteria take particular values for `ld_limit` and `dfc_limit`. This need not be the case as they can take a range of values which can impact the results. We discuss these relations and insights of the nature of true and false positives in Section 5.3.

5.2 Deeper Dive into Security Bugs

As we are interested in the security applications of FLAG, we breakdown the detection of security bugs in Figure 3. The breakdown for functional defects is shown in Appendix A. 16 of the 35 security bugs were detected by both LLMs in both modes, and 29 were detected by at least one LLM in one mode. gpt-3.5-turbo in auto-complete mode performed the best, detecting 26 compared to 22, 19 and 19 for gpt-3.5-turbo in instructed-complete, code-davinci-002 in auto-complete and code-davinci-002 in insertion respectively. Python security bugs were the ones best detected by the LLMs. All defects

Table 4: Results Summary. Number of defects detected (DD), False Positive Rate (FPR) and True Positive Rate (TPR) shown for each combination of LLM, Mode of completion, Language, Source and Criteria. C0 has a ld_limit of 10 i.e C0(10). C1 has ld_limit of 20 and dfc_limit of 10 i.e., C1(20,10). C2 has the same limits as C1 but employs the reduce_fp() function, i.e., C2(20,10). Numbers in bold summarize the metrics of all sources for a mode of completion of an LLM. Bold number in DD column is the sum of DD for all sources. Bold numbers in FPR and TPR columns are the FPR and TPR for all sources.

Model	Mode of Completion	Language	Source	# Defects	Criterion 0			Criterion 1			Criterion 2		
					DD	FPR	TPR	DD	FPR	TPR	DD	FPR	TPR
davinci-002	auto complete	C	C1- CVEs	8	1	0.071	0.125	2	0.113	0.250	2	0.098	0.250
			C1-CWEs	5	1	0.101	0.200	2	0.163	0.400	2	0.124	0.400
			C2	30	19	0.088	0.633	25	0.111	0.833	24	0.093	0.800
		Verilog	V1	10	1	0.081	0.100	7	0.216	0.700	7	0.206	0.700
			V2	34	20	0.089	0.588	26	0.172	0.765	25	0.149	0.735
		Python	P1	12	5	0.063	0.417	10	0.165	0.833	7	0.092	0.583
			P2	22	5	0.064	0.227	9	0.163	0.409	9	0.147	0.409
				121	**52**	**0.075**	**0.430**	**81**	**0.138**	**0.669**	**76**	**0.121**	**0.628**
	insertion	C	C1- CVEs	8	2	0.143	0.250	2	0.144	0.250	2	0.126	0.250
			C1-CWEs	5	1	0.147	0.200	2	0.163	0.400	2	0.124	0.400
			C2	30	24	0.108	0.800	25	0.111	0.833	24	0.094	0.800
		Verilog	V1	10	4	0.155	0.400	7	0.215	0.700	7	0.204	0.700
			V2	34	25	0.148	0.735	26	0.174	0.765	25	0.150	0.735
		Python	P1	12	7	0.209	0.583	9	0.296	0.750	7	0.175	0.583
			P2	22	9	0.145	0.409	11	0.196	0.500	11	0.168	0.500
				121	**72**	**0.145**	**0.595**	**82**	**0.162**	**0.678**	**78**	**0.141**	**0.645**
gpt-3.5-turbo	auto complete	C	C1- CVEs	8	0	0.089	0.000	4	0.165	0.500	4	0.137	0.500
			C1-CWEs	5	0	0.147	0.000	3	0.302	0.600	3	0.256	0.600
			C2	30	19	0.162	0.633	27	0.245	0.900	25	0.219	0.833
		Verilog	V1	10	0	0.116	0.000	8	0.352	0.800	8	0.318	0.800
			V2	34	18	0.135	0.529	30	0.294	0.882	29	0.251	0.853
		Python	P1	12	1	0.092	0.083	11	0.282	0.917	11	0.189	0.917
			P2	22	4	0.060	0.182	10	0.193	0.455	10	0.176	0.455
				121	**42**	**0.099**	**0.347**	**93**	**0.210**	**0.769**	**90**	**0.181**	**0.744**
	instructed complete	C	C1- CVEs	8	1	0.096	0.125	3	0.168	0.375	3	0.131	0.375
			C1-CWEs	5	0	0.124	0.000	3	0.279	0.600	3	0.256	0.600
			C2	30	19	0.133	0.633	24	0.224	0.800	22	0.193	0.733
		Verilog	V1	10	0	0.127	0.000	6	0.372	0.600	6	0.318	0.600
			V2	34	14	0.159	0.412	26	0.312	0.765	25	0.257	0.735
		Python	P1	12	2	0.102	0.167	10	0.282	0.833	10	0.194	0.833
			P2	22	3	0.055	0.136	8	0.160	0.364	8	0.144	0.364
				121	**39**	**0.105**	**0.322**	**80**	**0.210**	**0.661**	**77**	**0.172**	**0.636**

for Python were identified. It is followed by Verilog and C, respectively. A closer inspection allows us to develop some insights into why this might be the case. Firstly, the size of P1 source files was smaller in comparison to other sources. It had an average file size of 17 lines compared to 1426 of C1 and 249 of V1. Additionally, the small file size allowed the LLM to take in the entire source code before the bug as a part of the prompt, providing it with the complete context of the intended functionality of the files. Secondly, benchmarks in P1 were particularly designed by Pearce et al. [26, 27] for security evaluation whereas benchmarks in V1 and C1 were a combination of those explicitly designed for security and real-world examples. 6 of the benchmarks in V1 were from code in real implementations of SoCs, and 8 benchmarks in C1 are from real-world CVEs. This also explains why FLAG performed poorly on C1-CVEs, i.e., C1-1 - C1-8.

We are interested in the comparison of the performance of FLAG between functional and security bugs. Based on data in Figure 4, one may deduce that FLAG detects functional bugs better than security bugs. Considering results at C2(20,10), functional bugs have a higher average TPR of 0.689 compared to that for security bugs at 0.676. They also have a lower average FPR of 0.176 compared to that for security bugs at 0.271. A closer look reveals that Python is an exception with a higher TPR for P1 as compared to P2. For gpt-3.5-turbo the TPR for P1 is twice that of P2, indicating that gpt-3.5-turbo does well in detecting security bugs in Python. We exclude data for C1-CVEs from the analysis because the number of lines for these benchmarks is many times larger than others, skewing averages towards C1-CVEs.

5.3 Detailed Analysis

How does refining criteria from C0 to C2 impact DD and AFPR? Moving from C0 to C1, `DD` and `FPR` both increase. This is because `ld_limit` is considerably relaxed from 10 to 20, and `dfc_limit` is used to include lines that may exceed the `ld_limit`. On average, the `DD` increases from 51 to 84 while the `FPR` increases from 0.106 to 0.18. Moving from criteria C1 to C2, `DD` and `FPR` both decrease. This is because

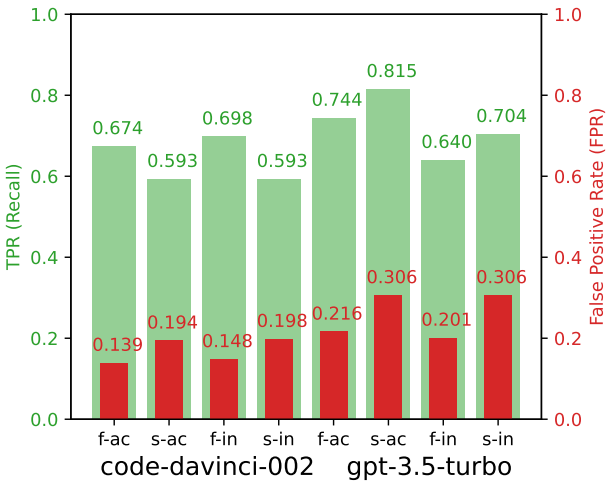


Figure 4: Performance for function and security categories using criterion 2 C2(20,10). 'f'=functional, 's'=security, 'ac'=auto-complete, 'in'=insert for code-davinci-002, 'in'=instruct-complete for gpt-3.5-turbo. C2-CVEs excluded.

the *reduce_fp()* function removes highlighted lines from *reported_lines*. Some false positives were removed, but so were some true positives. On average, the DD decreases from 85 to 80 while the FPR decreases from 0.18 to 0.154. While the DD decreases by 4.76%, the FPR decreases by 14.6%. This signifies importance of *reduce_fp()* in improving the performance of consistency checking. The trends are shown in Figure 5.

How do the different completion modes impact DD and AFPR? Instinctively, we may think that an insertion mode should do better than the auto-complete mode, as the LLM has access to more information in the form of the suffix, in addition to the prefix. Additionally, instructing gpt-3.5-turbo

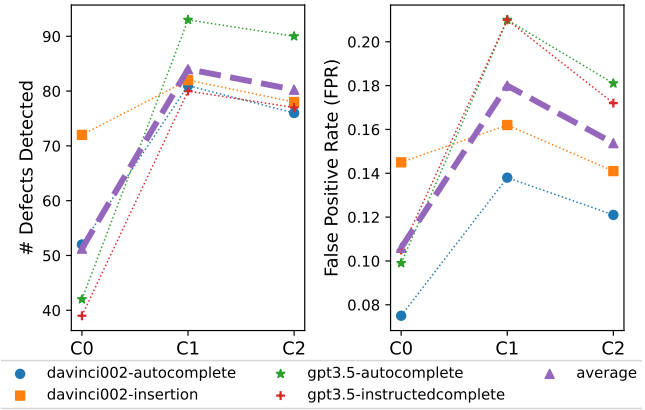
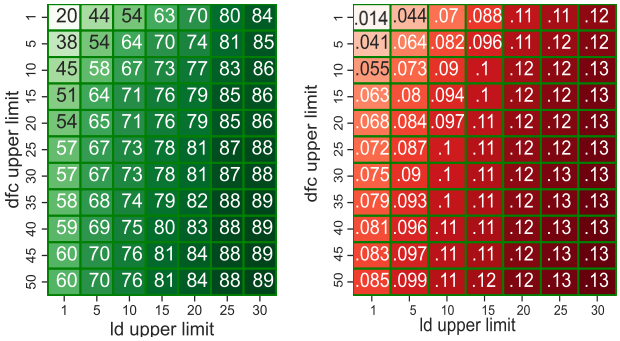


Figure 5: Trend of DD (left) and FPR (right) for C0 - C2 criteria. Combined trend for LLMs and modes is the average in purple.

Figure 6: Performance of completion modes for C2(20,10).

to produce "the next line of code" in the instructed-complete mode should perform better than the auto-complete mode as the likelihood of generating code instead of an explanation of code should be higher. Figure 6 compares the performances of different completion modes at C2(20,10). For code-davinci, the insertion mode does allow us to detect 2 additional defects but at the cost of 16.5% increase in FPR. This is probably not a valuable enough trade-off, meaning that not a lot of benefit is obtained from using the insertion mode. For gpt-3.5-turbo, the instructed-complete mode does significantly worse than its auto-complete counterpart. It detects 13 fewer defects while decreasing the FPR by only 4.97%.

How do `ld_limit` and `dfc_limit` impact DD and AFPR? The two main determinants of highlighting lines in our checking approach are `ld_limit` and `dfc_limit`. Since they can take any continuous value, there is a need for a deeper analysis other than considering the particular values discussed in criteria C0 to C2. We conduct sweeps across the two limits for the code-davinci-002 model in auto-complete mode to see their impacts on the DD and FPR. Figure 7a and Figure 7b show



(a) Defects detected (DD) (b) False Positive Rates (FPR)s

Figure 7: Defects, FPRs for (ld_limit,dfc_limit) combinations for code-davinci-002 auto-complete mode.

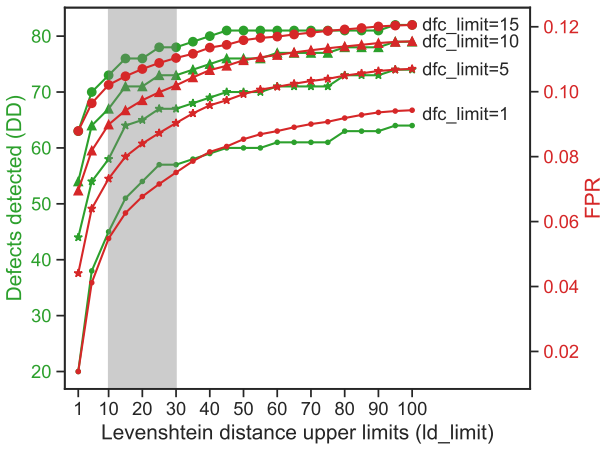
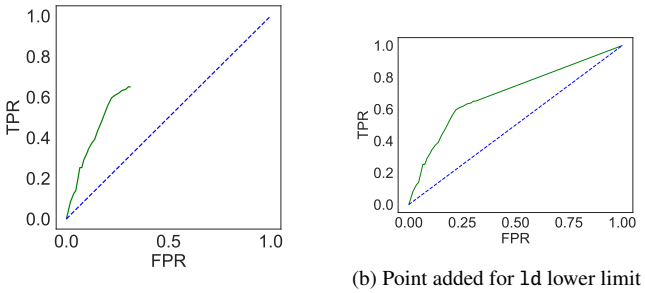


Figure 8: Change in Defects Detected and False Positive Rate with Levenshtein distance upper limit (ld_limit). Highlighted region produces decent results for `dfc_limit` values.



(a) `ld` lower limit is set to 0.

(b) Point added for `ld` lower limit at -1 and upper limit set at 1000.

Figure 9: Receiver Operating Characteristic (ROC) when `ld_limit` are threshold for code-davinci002 auto-completes.

how the DD and FPR change respectively at different combinations of `ld_limit` and `dfc_limit` in the range from 0 to 30 for `ld_limit` and 0 to 50 for `dfc_limit`. We observe that the `ld_upper_limit` is much more dominant in impacting the DD and FPR. A change in `ld_upper_limit` brings about a greater change in both DD and FPR compared to an equal change in `dfc_limit`. As a result, DD and FPR also saturate earlier while increasing `ld_limit`. DD loosely saturates at `ld_limit` of 30 as there is limited impact of changing `dfc_limit` at that value. Similarly, FPR loosely saturates at `ld_limit` of 20. These heatmaps are partial, but extending them for `ld_limit`=50 reveals that the maximum possible values for DD and FPR are 90 and 0.14, respectively. We also observe that DD and FPR are more sensitive at smaller values of `ld_limit` and `dfc_limit`. We can achieve 80% of the maximum value of DD, i.e., 72 at `ld_limit` of 15 and `dfc_limit` of 10. Similarly, we can achieve 80% of the maximum value of FPR i.e., 0.11 at `ld_limit` of 15 and `dfc_limit` of 25. This analysis helps in gaining an estimate of what are the ranges that should be considered when designing similar tools.

This can be explored further by zooming into the smaller values for `dfc_limit` and extending the values for `ld_limit` to 100 as shown in Figure 8. For a particular `dfc_limit`, as we increase the `ld_limit`, both DD and FPR increase. While the rate of increase for both decreases (shown by the decreasing gradient), there is a point after which the rate of increase in DD is significantly less than that of FPR. This is the point beyond which the benefit of obtaining more defects would be overpowered by the cost of increasing false positives. Conversely, a much smaller value of `ld_limit` would simply not find enough defects. The highlighted region in Figure 8 indicates the values for which the consistency checking approach provides "decent" results i.e., between 10 and 30. A similar exploration with keeping `ld_limit` constant and sweeping `dfc_limit` provides similar insights. The values for DD and FPR, however, are not distinct enough for different values of `ld_limit` for a graphical representation to be illustrative.

Naturally, a question may be asked of what is the optimal combination of `ld_limit` and `dfc_limit`? How many false positives a coder is willing to accept for the benefit of detecting more bugs is a matter of subjectivity. We illustrated the trade-off between the TPR and the FPR in the heatmaps. A way of deciding which limits to use could be by first looking at the number of false positives you are ready to bear. For example, if you are willing to look at 7 lines out of 100 for every bug detected, i.e., FPR of 0.07, you may choose `(ld_limit,dfc_limit)` of (5,10) as shown in Figure 7b. At these limits, you would be able to locate 58 of the 121 defects covered in our benchmarks as that is the corresponding value in Figure 7a.

How much better are LLMs in relation to random guess? In order to illustrate the success of consistency checking as a classifier, we represent the TPR and FPR in the form of a Receiver Operating Characteristic (ROC) curve at different thresholds for `ld_limit`. We used code-davinci002 in auto-complete mode for this purpose. In our experiments, we set a hard lower limit for `ld` of 0. This means that some of the lines that have a `ld` of zero will never be counted as a true positive or a false positive, resulting in an incomplete ROC curve. This situation is represented in Figure 9a, where the TPR and FPR are capped. The ROC curve, however, lies above the TPR==FPR line showing that our classification methodology is better than guesswork. For completeness' sake, if we utilize a lower limit at -1 and upper limit at 1000, we can obtain TPR and FPR of 1 because all lines will be highlighted as containing a defect. This is represented in Figure 9b.

Are bugs of one language easier to detect than the others? Based on the data in Figure 10, we can conclude that FLAG consistency checker performed the best on C and worst on Python. Although C has a slightly lower TPR than Verilog for both LLMs, it has a significantly lower FPR. While its TPR is 10.5% lower, its FPR is 48.4% less than that of Ver-

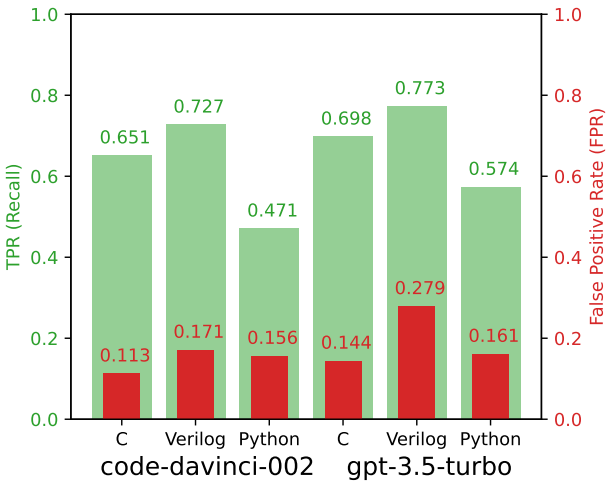


Figure 10: Performance on languages for criterion 2 i.e., C2(20,10). Values represent averages across modes for LLM.

ilog. Python has the lowest TPR and only the second lowest FPR for both LLMs. This is surprising because we expect the larger amount of open-source Python in the training data would translate to better performance than on Verilog. This is probably because 22 of the benchmarks for Python were real-world examples compared to only 6 for Verilog.

How much do comments help? We signify the role of comments by analyzing the `dfc` and BLEU scores for true and false positives. The `dfc` signifies the closeness of a line to a comment, while the BLEU score signifies the quality of the comment produced by the LLM. Figure 11 shows how the `dfc` differs between true and false positives. The data here does not include instances where `dfc` is not available, i.e., when there is no comment before the line concerned. Most of the data in both cases lies in the range for smallest values of `dfc`. This is because comments are frequently written in benchmarks we study. `dfc` for true positives has a lower average of 18.4 compared to that of false positives at 327.9. LLM does a better job at classification when the line concerned is closer to a comment. `dfc` for true positives has a much smaller standard deviation of 47.7 compared to that of false positives at 632.1. Thus false positives cover a much larger range of values.

Is the average dfc or avg bleu score correlated to the success of the checker? Figure 12 shows how the previous comment's BLEU score differs between true and false positives. The data does not include lines for which there was no comment before the line or when the LLM was not able to produce a comment. For a line of code being analyzed, FLAG tracks the most recent comment preceding it. FLAG compares the comment produced by the LLM at this line to the original comment to compute previous comment's BLEU score. A high value indicates that the LLM is on the right track following the intention of the coder. The previous comment's

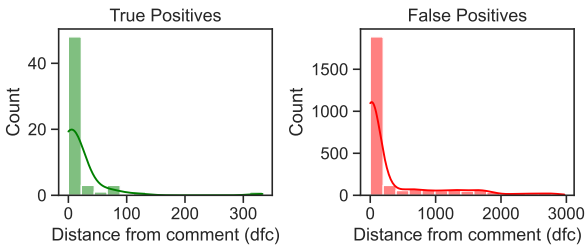


Figure 11: Distance from comment (dfc) distributions for code-davinci-002 in auto-complete mode.

BLEU score for true positives has a similar average of 0.407 compared to that of false positives at 0.473. This shows that the previous comment's BLEU score does not play a crucial role in the classification. The previous comment's BLEU score for true positives has a similar standard deviation of 0.193 compared to that of false positives at 0.236.

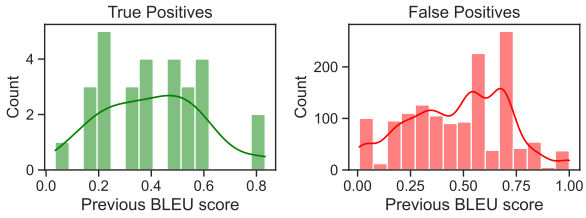


Figure 12: Previous comment's BLEU score distributions for code-davinci-002 in auto-complete mode.

6 Discussion

In the previous section, we explored whether there is a significant difference between true positives and false positives based on information like `dfc` or previous comment's BLEU scores. This analysis was done at the level of granularity of

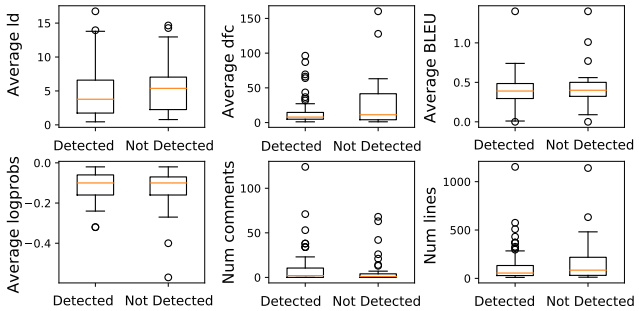


Figure 13: Metadata relation to success of checker. Data used for code-davinci-002 in auto-complete mode for C2(20,10).

lines. There is however room for such an analysis at the level of granularity of the file or benchmark. Is there a relation between properties of files, like the number of comments, size of benchmark, etc., to the likelihood of a defect being detected for that benchmark? We investigate this relation for the following properties of each benchmark: average `ld`, average `dfc`, average `BLEU`, average `logprobs`, number of comments and number of lines (size). The results are shown in Figure 13. The boxplots reveal that average `dfc`, number of comments, and number of lines have a significantly different distribution for benchmarks where defects were detected than those where they were not. A lower average `dfc`, higher number of comments, and smaller number of lines increase the likelihood of the defect being detected. Average `ld`, average `BLEU`, and average `logprobs`, however, do not pose this difference.

FLAG relies on the fundamental idea that a line containing a defect is likely to be written in an alternative way by an LLM i.e., `ld`> 0. In our experiments, however, this was not always the case. There were 11 benchmarks for which `ld` was 0 for both LLMs in both their modes for C2(20,10), as the LLMs generated the exact same code as the original code. 8 of them were from source V2, 2 from C1 and 1 from P2. Thus, LLMs sometime produce functionally buggy code for Verilog but generally generate alternate code when facing a defect.

A limitation of FLAG is that going line by line for almost every line in the source code does not scale in terms of time. While smaller files $\sim$100 lines are checked in under a minute, scanning a file with thousands of lines can take about an hour. To address this, one could prioritize generating a subset of lines. This could be done by checking security-sensitive regions of source code, e.g., checking if conditions and statements inside the if block. Another approach is to ignore lines without much substance, e.g., a line with only a keyword.

Another limitation is the high `FPR`. The `FPR` for our experiments ranges from 0.121 to 0.172, meaning that for a source code file of 100 lines with one defect, about 12-17 lines would be falsely highlighted. Future work will focus on further reducing the flagged lines. An approach is to consider *features* for regions of source code instead of lines for the criteria for classification, e.g., do not highlight defects between lines x and y for file z because average `logprobs` for this region indicates that the LLM is not confident in its suggestions.

FLAG can aid a designer by focusing the search for bugs on only $12-17\%$ of the source. Moreover, efforts to reduce false positives should be met with caution as they may eliminate some true positives. FLAG's approach to removing false positives of moving from criteria C1 to C2 reduced the number of defects detected by an average of 3.75. An example is shown in Table 5. The defect is the line highlighted in pink for which the LLM generates alternative code that has an `ld` of 10 and a `dfc` of 3. It is highlighted by FLAG as a concern when using C1(20,10) but not when using C2(20,10). This is because C2 uses *reduce_fp()*, which removes a defective line as it has a `logprob` < -0.5. FLAG is promising for bug detection as

Table 5: Lines generated and features extracted by code-davinci-002 in auto-complete mode for CWE-79 in P2. Defect in last line was detected with C1 but not C2. logp=logprob.

Original line	Generated line	dfc	ld	logp
bp = Blueprint('hello' __name__)	bp = Blueprint('hello' __name__)	10	1	-0.33
#function which says hello to the user	#function to run the command	0		-1.67
@bp.route("hello/<username>")	@bp.route('/hello /<username>')	1	3	-0.12
def hello(username):	def hello(username):	2	0	-0.14
return " Hello %s! " % username	return "Hello "+username+"!"	3	18	-0.6

LLMs improve. With larger data sizes and fine-tuning, newer LLMs can produce better code and code suggestions.

7 Conclusions and Future work

Overall, 101 out of 121 defects were detected by at least one mode of the 2 LLMs. On average, for the 4 combinations of mode and LLMs, 80 defects were detected at an `FPR` of 0.154 using criterion C2. This gives credence to consistency checks by FLAG as a bug detector and search space localizer. We find that gpt-3.5-turbo has a better ability to detect bugs but a higher `FPR` than code-davinci-002. FLAG is marginally better on functional bugs than on security-related bugs. We find that Levenshtein distance between the original and LLM-generated code is the dominant classification feature among those we explored. That said, comments play an important role in the performance of FLAG as lines of code with smaller distances from comments are classified with better success. Lastly, FLAG does best on C and worst on Python for the benchmarks we studied.

Since FLAG is novel in its implementation, several future work directions can be formulated. More features in code and LLMs can be used in classification e.g., BLEU scores for code and embedding scores for code and comments. With a bigger set of classification features and a bigger set of true and false positives in the form of benchmarks, ML classifiers could be trained to determine the criteria for classification. Future versions of FLAG could analyze code and comments in chunks rather than lines. Another interesting idea is to run multiple LLMs and flag a line based on the union of features.

Acknowledgments

This research work is supported in part by a gift from Intel Corporation. This work does not in any way constitute an

Intel endorsement of a product or supplier. We acknowledge the support of the Natural Sciences and Engineering Research Council of Canada (NSERC), RGPIN-2022-03027.

Availability

The artifacts (FLAG source code, LLM outputs) produced and presented in this study are at [32].

References

[1] Baleegh Ahmad, Shailja Thakur, Benjamin Tan, Ramesh Karri, and Hammond Pearce. Fixing Hardware Security Bugs with Large Language Models, February 2023. arXiv:2302.01215 [cs].

[2] Hammad Ahmad, Yu Huang, and Westley Weimer. CirFix: automatically repairing defects in hardware design code. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '22, pages 990–1003, New York, NY, USA, February 2022. Association for Computing Machinery.

[3] Aizu. Aizu Online Judge, 2015. Accessed on 2023-06-06. https://onlinejudge.u-aizu.ac.jp/home.

[4] Stefan Berner, Roland Weber, and Rudolf K. Keller. Observations and lessons learned from automated testing. In *Proceedings of the 27th international conference on Software engineering*, ICSE '05, pages 571–579, New York, NY, USA, May 2005. Association for Computing Machinery.

[5] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel Ziegler, Jeffrey Wu, Clemens Winter, Chris Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language Models are Few-Shot Learners. In H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan, and H. Lin, editors, *Advances in Neural Information Processing Systems*, volume 33, pages 1877–1901. Curran Associates, Inc., 2020.

[6] Saikat Chakraborty, Yujian Li, Matt Irvine, Ripon Saha, and Baishakhi Ray. Entropy Guided Spectrum Based Bug Localization Using Statistical Language Model, February 2018. arXiv:1802.06947 [cs].

[7] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating Large Language Models Trained on Code, July 2021. arXiv:2107.03374 [cs].

[8] Brian Chess and Gary McGraw. Static analysis for security. *IEEE Security & Privacy*, 2(6):76–79, November 2004.

[9] The MITRE Corporation. CWE - CWE-1194: Hardware Design (4.1), 2022. https://cwe.mitre.org/data/definitions/1194.html.

[10] National Vulnerability Database. NVD - Home, 2023. Accessed on 2023-06-06. https://nvd.nist.gov/.

[11] Yinlin Deng, Chunqiu Steven Xia, Chenyuan Yang, Shizhuo Dylan Zhang, Shujing Yang, and Lingming Zhang. Large Language Models are Edge-Case Fuzzers: Testing Deep Learning Libraries via FuzzGPT, April 2023. arXiv:2304.02014 [cs].

[12] Ghada Dessouky, David Gens, Patrick Haney, Garrett Persyn, Arun Kanuparthi, Hareesh Khattri, Jason Fung, Ahmad-Reza Sadeghi, and Jeyavijayan Rajendran. Hardfails: Insights into Software-Exploitable Hardware Bugs. In *Proceedings of the 28th USENIX Conference on Security Symposium*, SEC'19, pages 213–230, Santa Clara, CA, USA, 2019. USENIX Association.

[13] Elizabeth Dinella, Hanjun Dai, Ziyang Li, Mayur Naik, Le Song, and Ke Wang. Hoppity: Learning graph transformations to detect and fix bugs in programs. In *International Conference on Learning Representations (ICLR)*, 2020.

[14] EvoSuite. Release V1.2.0 · EvoSuite/evosuite, 2021. Accessed 2023-05-30. https://github.com/EvoSuite/evosuite/releases/tag/v1.2.0.

[15] Philip Gage. A New Algorithm for Data Compression. *C Users Journal*, 12(2):23–38, February 1994.

[16] Andrew Habib and Michael Pradel. How many of all bugs do we find? a study of static bug detectors. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ASE '18, pages 317–328, New York, NY, USA, September 2018. Association for Computing Machinery.

[17] HACK@EVENT. HACK@DAC21 – HACK@EVENT, 2022. https://hackatevent.org/hackdac21/.

[18] Github Inc. CodeQL for research, 2021. https://securitylab.github.com/tools/codeql/.

[19] Infer. Infer, 2021. Accessed 2023-06-06. https://github.com/facebook/infer.

[20] Jun Li, Bodong Zhao, and Chao Zhang. Fuzzing: a survey. *Cybersecurity*, 1(1):6, December 2018.

[21] Tsz-On Li, Wenxi Zong, Yibo Wang, Haoye Tian, Ying Wang, Shing-Chi Cheung, and Jeff Kramer. Finding Failure-Inducing Test Cases with ChatGPT, April 2023. arXiv:2304.11686 [cs].

[22] Derrick Lin, James Koppel, Angela Chen, and Armando Solar-Lezama. QuixBugs: a multi-lingual program repair benchmark set based on the quixey challenge. In *Proceedings Companion of the 2017 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity*, SPLASH Companion 2017, pages 55–56, New York, NY, USA, October 2017. Association for Computing Machinery.

[23] lowRISC contributors. Open source silicon root of trust (RoT) | OpenTitan, 2023. https://opentitan.org/.

[24] Steve McConnell. *Code complete*. Microsoft Press, Redmond, Wash, 2nd edition, 2004.

[25] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. BLEU: a method for automatic evaluation of machine translation. In *Proceedings of the 40th Annual Meeting on Association for Computational Linguistics*, ACL '02, pages 311–318, USA, July 2002. Association for Computational Linguistics.

[26] Hammond Pearce, Baleegh Ahmad, Benjamin Tan, Brendan Dolan-Gavitt, and Ramesh Karri. Asleep at the Keyboard? Assessing the Security of GitHub Copilot's Code Contributions. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 754–768, May 2022. ISSN: 2375-1207.

[27] Hammond Pearce, Benjamin Tan, Baleegh Ahmad, Ramesh Karri, and Brendan Dolan-Gavitt. Examining Zero-Shot Vulnerability Repair with Large Language Models. In *2023 IEEE Symposium on Security and Privacy (SP)*, pages 2339–2356. IEEE Computer Society, 2023.

[28] Michael Pradel and Koushik Sen. DeepBugs: a learning approach to name-based bug detection. *Proceedings of the ACM on Programming Languages*, 2(OOPSLA):147:1–147:25, October 2018.

[29] Error Prone. Error Prone, 2023. Accessed 2023-05-30. https://github.com/google/error-prone.

[30] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language Models are Unsupervised Multitask Learners. pages 1–24, 2019. https://cdn.openai.com/better-language-models/language_models_are_unsupervised_multitask_learners.pdf.

[31] Randoop. Randoop: Automatic unit test generation for Java, 2023. Accessed 2023-05-30. https://randoop.github.io/randoop/.

[32] Anonymized for review. Artifacts for "FLAG: Finding Line Anomalies (in code) with Generative AI", June 2023. https://zenodo.org/record/8012211.

[33] Gustavo Sandoval, Hammond Pearce, Teo Nys, Ramesh Karri, Siddharth Garg, and Brendan Dolan-Gavitt. Lost at C: A User Study on the Security Implications of Large Language Model Code Assistants. *USENIX Security Symposium*, 2023.

[34] Sina Shamshiri, René Just, José Miguel Rojas, Gordon Fraser, Phil McMinn, and Andrea Arcuri. Do Automatically Generated Unit Tests Find Real Faults? An Empirical Study of Effectiveness and Challenges (T). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 201–211, November 2015.

[35] SpotBugs. SpotBugs, 2022. Accessed 2023-05-30. https://spotbugs.github.io/.

[36] Daniela Steidl, Benjamin Hummel, and Elmar Juergens. Quality analysis of source code comments. In *2013 21st International Conference on Program Comprehension (ICPC)*, pages 83–92, May 2013. ISSN: 1092-8138.

[37] Lin Tan, Ding Yuan, Gopal Krishna, and Yuanyuan Zhou. /*icomment: bugs or bad comments?*/. *ACM SIGOPS Operating Systems Review*, 41(6):145–158, October 2007.

[38] Shin Hwei Tan, Darko Marinov, Lin Tan, and Gary T. Leavens. @tComment: Testing Javadoc Comments to Detect Comment-Code Inconsistencies. In *Verification*

and Validation 2012 IEEE Fifth International Conference on Software Testing, pages 260–269, April 2012. ISSN: 2159-4848.

[39] Yunosuke Teshima and Yutaka Watanobe. Bug Detection Based on LSTM Networks and Solution Codes. In *2018 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*, pages 3541–3546, October 2018. ISSN: 2577-1655.

[40] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is All you Need. In *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017.

[41] Song Wang, Devin Chollak, Dana Movshovitz-Attias, and Lin Tan. Bugram: bug detection with n-gram language models. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ASE '16, pages 708–719, New York, NY, USA, August 2016. Association for Computing Machinery.

[42] Ratnadira Widyasari, Sheng Qin Sim, Camellia Lok, Haodi Qi, Jack Phan, Qijin Tay, Constance Tan, Fiona Wee, Jodie Ethelda Tan, Yuheng Yieh, Brian Goh, Ferdian Thung, Hong Jin Kang, Thong Hoang, David Lo, and Eng Lieh Ouh. BugsInPy: a database of existing bugs in Python programs to enable controlled testing and debugging studies. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2020, pages 1556–1560, New York, NY, USA, November 2020. Association for Computing Machinery.

[43] Jooyong Yi, Umair Z. Ahmed, Amey Karkare, Shin Hwei Tan, and Abhik Roychoudhury. A feasibility study of using automated program repair for introductory programming assignments. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, pages 740–751, New York, NY, USA, August 2017. Association for Computing Machinery.

A Appendix

Table 6 is the complete table for security-related bugs examined in this work. The description for the functional defect benchmarks used and their corresponding IDs are presented in Table 7. In Figure 14, we present the breakdown of the combinations of LLM and their modes that were able to detect each of the collected defects.

Table 6: Complete descriptions for security-related bugs. (*) indicates that the CWE is in one of MITRE's top 25 CWEs list.

ID	CWE	Description
C1-1	125*	libjpeg-turbo 2.0.1 has a heap-based buffer over-read in the put_pixel_rows function.
C1-2	119*	Heap-based buffer overflow in libjpeg-turbo 1.2.0 allows remote attackers to cause a denial of service.
C1-3	119*	The DumpModeDecode function in libtiff 4.0.6 and earlier allows attackers to cause a denial of service.
C1-4	787*	Segmentation fault due to an out-of-bounds write can be triggered with a malformed TIFF file.
C1-5	787*	Out of bounds write in function used to set pixel which may lead to corruption of data, a crash, or code execution.
C1-6	189	Off-by-one error in LibTIFF 4.0.7 allows remote attackers to have unspecified impact via a crafted image
C1-7	20*	Signed integer overflow in tif_jpeg.c might allow remote attackers to cause a denial of service.
C1-8	369	rgb2ycbcr tool in LibTIFF 4.0.6 and earlier allows remote attackers to cause a denial of service (divide-by-zero).
C1-9	119*	Improper restriction of bounds when accessing the memory array. No check for array accessed.
C1-10	125*	Potential buffer under-read. No check for lower bond of array index.
C1-11	416*	An allocated memory block is used after it has been freed. This can cause memory corruption.
C1-12	476*	Dereferencing an untested value from a function that can return null may lead to undefined behavior.
C1-13	732	Creating a file that is world-writable can allow an attacker to write to the file.
P1-1	79*	Writing user input directly to a web page allows for a cross-site scripting vulnerability.
P1-2	20*	Security checks on the substrings of an unparsed URL are often vulnerable to bypassing.
P1-3	22*	Uncontrolled data used in path expression can allow an attacker to access unexpected resources.
P1-4	78*	Externally controlled strings in command line may allow malicious user to change the meaning of the command.
P1-5	89*	Building a SQL query from user-controlled sources is vulnerable to insertion of malicious SQL code by the user.
P1-6	200	Different messages for an incorrect username, versus a correct username is correct but wrong password.
P1-7	306*	No authentication for functionality that requires a user identity or consumes a significant amount of resources.
P1-8	434*	Allows the attacker to upload or transfer files of dangerous types that can be automatically processed.
P1-9	502*	The product deserializes untrusted data without sufficiently verifying that the resulting data will be valid.
P1-10	552	Insufficiently protected credentials make files or directories accessible to unauthorized actors.
P1-11	732	Overly permissive file permissions allow files to be readable or writable by users other than the owner.
P1-12	798*	Password comparison with a string literal allows an attacker to bypass the authentication that has been configured.
V1-1	1234	Lock protection is overridden when debug mode is active.
V1-2	1271	A locked register does not have a value assigned on reset. When it is brought out of reset, the state is unknown.
V1-3	1280	An asset is allowed to be modified even before the access control check is complete.
V1-4	1276	The signal depicting the security level for a peripheral instantiated in a SoC is incorrectly grounded.
V1-5	1245	An alert is triggered in the FSM when start signal is high in a state other than Waiting.
V1-6	1245	The error signal for an FSM moving into invalid state upon global escalation is not asserted.
V1-7	1245	The done for an FSM is asserted outside of expected window, i.e., during a transmission state.
V1-8	1234	The core is incorrectly unstalled if there is a request to enter debug mode.
V1-9	1271	The register controlling whether the Physical Memory Protection register is writable is not assigned a reset value.
V1-10	1245	A 4 bit FSM has only 15 states defined and no default statement, resulting in an incomplete case statement.

Table 7: Descriptions for bugs in source C2, P2, and V2

ID	Description	ID	Description	ID	Description
C2-1	Rounding error	C2-30	Decrement instead of increment	V2-7	Negated if-condition
C2-2	Incorrect formula	P2-1	wrong regex expression	V2-8	True and false branches of if-statement swapped
C2-3	compares character instead of integer	P2-2	wrong regex expression	V2-9	Default in case statement omitted
C2-4	Incorrect calculation	P2-3	incorrect returned objects	V2-10	Assignment to next state omitted, default cases in case statement omitted
C2-5	Using wrong variable	P2-4	incorrect regex expression	V2-11	state omitted from senslist
C2-6	Using wrong variable	P2-5	sudo mode not considered	V2-12	Blocking instead of nonblocking assignments
C2-7	Incorrect condition for prime	P2-6	incorrect split token used	V2-13	Blocking instead of nonblocking assignments
C2-8	Incorrect condition for prime	P2-7	incorrect assignment	V2-14	Negated if-condition
C2-9	incorrect for loop condition	P2-8	incorrect implementation of adding error codes to string messages	V2-15	Incorrect sens
C2-10	incorrect for loop initializer	P2-9	incrrect buffer assignment	V2-16	Three numeric errors
C2-11	Incorrect assignment	P2-10	incorrect parameter value	V2-17	Hex instead of binary numbers
C2-12	Incorrect array index	P2-11	incorrect os command	V2-18	1 bit instead of 4 bit output wire
C2-13	Incorrect swap	P2-12	incomplete return	V2-19	Incorrect sens
C2-14	Incorrect string shift amount	P2-13	incorrect function argument	V2-20	Incorrect assignment
C2-15	Incorrect input scan	P2-14	incorrect computation	V2-21	Removed cmd_ack
C2-16	Compare w/ incorrect variable in condition	P2-15	wrong error condition	V2-22	For-loop going too long
C2-17	No space in print	P2-16	incorrect assignment	V2-23	Incorrect assignment to out_ready
C2-18	Incorrect input scanning	P2-17	missing file encoding	V2-24	Not checking for buffer overflow during assignment
C2-19	Recursion not returned	P2-18	incomplete function arguments	V2-25	Logical instead of bitwise negation
C2-20	Incorrect comparison	P2-19	incorrect implementation of identifying column	V2-26	Incorrect bitshifting
C2-21	Incorrect condition	P2-20	incorrect appended element to list	V2-27	Incorrect instantiation of module
C2-22	Missing statement	P2-21	incorrect condition	V2-28	\textgreater instead of \textgreater{ }\textgreater for bitshifting
C2-23	Incorrect increment	P2-22	incorrect file path in os command	V2-29	Insufficient bits for numeric value
C2-24	Program not returned	V2-1	Numeric error	V2-30	Removed @posedge reset from senslist (sync vs async reset)
C2-25	Incorrect while loop condition	V2-2	Numeric error	V2-31	wr_data_r not reset correctly
C2-26	Incorrect sorting condition	V2-3	Incorrect assignment	V2-32	rd_data_r assigned incorrectly
C2-27	Statements wrongly excluded from if condition	V2-4	Incorrect sens	V2-33	Numeric error in parameter
C2-28	Incorrect assignments to object attributes	V2-5	Else-if instead of if	V2-34	Default in case statement omitted
C2-29	Wrong formula	V2-6	Counter never reset		

C2-1	C2-2	C2-3	C2-4 x x ✓ x	C2-5	C2-6	C2-7	C2-8 x x ✓ ✓	C2-9	C2-10	C2-11
C2-12	C2-13 x x ✓ ✓	C2-14	C2-15	C2-16 ✓ ✓ ✓ x	C2-17	C2-18	C2-19	C2-20	C2-21	C2-22
C2-23	C2-24 ✓ ✓ x x	C2-25	C2-26	C2-27	C2-28 ✓ ✓ x x	C2-29	C2-30 ✓ ✓ ✓ x	P2-1 x ✓ ✓ ✓	P2-2 x ✓ ✓ ✓	P2-3
P2-4	P2-5	P2-6	P2-7	P2-8	P2-9	P2-10 x x ✓ ✓	P2-11	P2-12 ✓ ✓ x ✓	P2-13 ✓ x ✓ x	P2-14 ✓ x x x
P2-15 x ✓ ✓ ✓	P2-16	P2-17 ✓ ✓ ✓ x	P2-18	P2-19 x ✓ ✓ x	P2-20	P2-21 x ✓ x x	P2-22 ✓ ✓ x x	V2-1	V2-2	V2-3 ✓ ✓ ✓ x
V2-4	V2-5	V2-6	V2-7	V2-8	V2-9	V2-10	V2-11	V2-12 x x ✓ ✓	V2-13	V2-14
V2-15	V2-16 x x ✓ ✓	V2-17 x x ✓ ✓	V2-18	V2-19 x x ✓ ✓	V2-20	V2-21 x x ✓ x	V2-22 ✓ ✓ ✓ x	V2-23 ✓ ✓ x x	V2-24	V2-25 ✓ ✓ x x
V2-26	V2-27	V2-28	V2-29	V2-30	V2-31	V2-32	V2-33	V2-34 x x ✓ x		

Figure 14: Functional bugs detected according to mode and LLM. A cell highlighted in green indicates that both LLMs in both modes were able to detect this defect. A cell highlighted in pink indicates that no LLM in any mode was able to detect this defect. The remaining cells present a sequence of 4 symbols to represent whether the defect was detected by code-davinci-002 in auto-complete, code-davinci-002 in insertion, gpt-3.5-turbo in auto-complete and gpt-3.5-turbo in instructed-complete respectively. A sequence of ✓ ✓ ✓ x means that the defect was detected by code-davinci-002 in both modes and by gpt-3.5-turbo in auto-complete but not in instructed-complete e.g. C2-16. x x x ✓ means that the defect was detected only by gpt-3.5-turbo in instructed-complete.