Automated Software Vulnerability Static Code Analysis Using Generative Pre-Trained Transformer Models

Elijah Pelofske*[1], Vincent Urias[1], and Lorie M. Liebrock[2,1]

[1]Sandia National Laboratories
[2]New Mexico Cybersecurity Center of Excellence, New Mexico Tech

Abstract

Generative Pre-Trained Transformer models have been shown to be surprisingly effective at a variety of natural language processing tasks – including generating computer code. However, in general GPT models have been shown to not be incredibly effective at handling specific computational tasks (such as evaluating mathematical functions). In this study, we evaluate the effectiveness of open source GPT models, with no fine-tuning, and with context introduced by the langchain and localGPT Large Language Model (LLM) framework, for the task of automatic identification of the presence of vulnerable code syntax (specifically targeting C and C++ source code). This task is evaluated on a selection of 36 source code examples from the NIST SARD dataset, which are specifically curated to not contain natural English that indicates the presence, or lack thereof, of a particular vulnerability (including the removal of all source code comments). The NIST SARD source code dataset contains identified vulnerable lines of source code that are examples of one out of the 839 distinct Common Weakness Enumerations (CWE), allowing for exact quantification of the GPT output classification error rate. A total of 5 GPT models are evaluated, using 10 different inference temperatures and 100 repetitions at each setting, resulting in $5,000$ GPT queries per vulnerable source code analyzed. Ultimately, we find that the open source GPT models that we evaluated are not suitable for fully automated vulnerability scanning because the false positive and false negative rates are too high to likely be useful in practice. However, we do find that the GPT models perform surprisingly well at automated vulnerability detection for some of the test cases, in particular surpassing random sampling (for some GPT models and inference temperatures), and being able to identify the exact lines of code that are vulnerable albeit at a low success rate. The best performing GPT model result found was Llama-2-70b-chat-hf with inference temperature of 0.1 applied to NIST SARD test case 149165 (which is an example of a buffer overflow vulnerability), which had a binary classification recall score of 1.0 and a precision of 1.0 for correctly and uniquely identifying the vulnerable line of code and the correct CWE number. Additionally, the GPT models are able to, with a rate quantifiably better than random sampling, identify the specific line of source that contains the identified CWE for many of the NIST SARD test cases.

1 Introduction

Generative Pre-Trained Transformer (GPT) models, built using the attention architecture [1], have been shown to be surprisingly effective at a variety of information processing tasks including computer code generation and natural language text summarization [2–5]. GPT models are intended to work very well with language, and are typically referred to as a class of Large Language Model (LLM). Notably however, GPT models can, in certain settings, also perform algorithmic tasks such as learning the greatest greatest common divisor algorithm [6]. The surprising capability of GPT models to perform well at tasks involving computer code leads to the natural question of whether GPT models could be used for determine security properties of static software samples. In this study, we examine whether current open source GPT models can be used to correctly identify software vulnerabilities and weaknesses in source code correctly.

The evaluation of our study is based on a set of 36 labeled examples from the highly credible NIST SARD dataset [7]. Each source code example in this dataset is meticulously mapped to the Common Weakness Enumeration (CWE) framework [8, 9], providing a solid foundation for our research on software vulnerability detection. The Common Weakness Enumeration (CWE) is a categorization of known types of software weaknesses, hardware weaknesses, and software vulnerabilities.

*E-mail: elijah.pelofske@protonmail.com

Static code vulnerability scanners are an important field of study because they speed up the process of identifying and thus patching software and even potentially hardware vulnerabilities and flaws that cause security problems in computer systems. There are a wide variety of static code analysis scanners that have been developed, usually focusing on specific domains or specific languages [10–18]. A challenge with static vulnerability analysis is that the outputs typically have high false positive rates because the tools usually use string pattern matching to identify known syntax representing an instance of a specific code flaw. The main advantage of static code analysis is that it is much faster and easier to apply than dynamic program analysis.

Ref. [19] studies the task of *repairing* CWE examples using GPT models. Ref. [20] examined source code vulnerability detection using several code language models (namely BERT variants) on a large dataset of software examples.

Our study introduces novel methods compared to existing research, particularly in the use of highly curated datasets that include the exact vulnerable line of source code. The structured GPT prompt we employ with these GPT models enables the automated extraction of vulnerability detection data from the output, facilitating large-scale, repeatable, accuracy analysis. This dataset combined with structured JSON prompting and parsing allows us to fully leverage the capabilities of GPT models for automated vulnerability detection, requiring no manual processing of the GPT output. In total, we generate and process $100 \cdot 10 \cdot 36 \cdot 5 = 180000$ discrete GPT text outputs (each of these being a separate, independent, and complete text inference computations. All of the GPT model computations are performed locally, on a local GPU compute server, using open source GPT models.

Model name	Reference(s)	Context Length	Architecture type	Model Size
`Llama-2-70b-chat-hf`	[5]	4096 Tokens	llama	70B params
`zephyr-7b-alpha`	[21]	32768 Tokens	mistral	7.24B params
`zephyr-7b-beta`	[21, 22]	32768 Tokens	mistral	7.24B params
`Mistral-7B-Instruct-v0.1`	[23]	32768 Tokens	mistral	7.24B params
`Turdus`	[24]	32768 Tokens	mistral	7.24B params

Table 1: Summary of the 6 Generative Pre-trained Transformers models used in this study. Here B denotes the order of magnitude billion.

2 Methods

The five open-source GPT models used in this study are summarized in Table 1. These GPT models were selected for their general performance among available GPT models - and all of these models can generate and parse computer code. Although a large number of different GPT models could be utilized for this study - due to the high computation time required to perform these experiments, a small number of GPT models were selected. These GPT models are from the huggingface GPT repository [25] and the GPT models are run using the Python3 modules `pytorch` [26] and `transformers`, using a local compute server that has four Nvidia A100 GPU's [27] (each with 82 Gigabytes of memory), and CUDA Version 12.4.

GPT models can be used to predict the most likely token, given the previous N tokens (where N is the context window for current GPT models on the order of $4096 - 32768$). When used in this manner, the text produced by the GPT models is deterministic, given that the previous tokens are the same. However, GPT models can learn and generate probability distributions of the most likely subsequent tokens - and we can add stochasticity in the text generation process by sampling from that distribution, weighted by their respective probabilities. The parameter that controls this sampling is known as *temperature*. This stochastic text generation is part of what makes generative machine learning models able to produce a wide variety of outputs. In the context of investigating the usage of these models as code-scanning tools, these models will only perform somewhat well at this task. Moreover, we need outputs to vary somewhat to determine the possible range of characteristics of the output since the text generation process of GPT models can be stochastic. Therefore, we vary the text generation inference temperature from 0.1 to 1.0 in increments of 0.1. The huggingface [25] python 3 module used to execute this GPT model inference allows temperatures up to 100, where 0 denotes deterministic output (with no sampling), and 100 denotes closer to uniform token sampling. We choose relatively low temperatures because we need the output to conform to parsable data structures. Critically, because the outputs vary significantly, we view this task as a *stochastic sampling* problem, where we characterize the proportion of samples with specific required properties (the most basic being whether the data could be automatically parsed). 100 samples (GPT outputs) are generated for each inference temperature, GPT model, and source code test case.

NIST SARD Test Case	CWE id	CWE name	Language	Number of Lines of Source Code
1792	CWE-79	Improper Neutralization of Input During Web Page Generation	C	24
500843	CWE-476	NULL Pointer Dereference	C++	11
1779	CWE-463	Deletion of Data Structure Sentinel	C	10
1645	CWE-20	Improper Input Validation	C	21
149165	CWE-121	Stack-based Buffer Overflow	C	35
2015	CWE-329	Generation of Predictable IV with CBC Mode	C	12
500757	CWE-787	Out-of-bounds Write	C++	7
149135	CWE-489	Active Debug Code	C	35
149203	CWE-416	Use After Free	C	30
149111	CWE-134	Use of Externally-Controlled Format String	C	18
149185	CWE-391	Unchecked Error Condition	C	11
1494	CWE-74	Improper Neutralization of Special Elements in Output Used by a Downstream Component	C++	12
149179	CWE-401	Missing Release of Memory after Effective Lifetime	C	17
149107	CWE-415	Double Free	C	55
148871	CWE-188	Reliance on Data/Memory Layout	C	77
149143	CWE-120	Buffer Copy without Checking Size of Input	C	13
149103	CWE-367	Time-of-check Time-of-use (TOCTOU) Race Condition	C	86
1501	CWE-822	Untrusted Pointer Dereference	C++	17
149199	CWE-412	Unrestricted Externally Accessible Lock	C	17
149183	CWE-468	Incorrect Pointer Scaling	C	10
149085	CWE-244	Improper Clearing of Heap Memory Before Release	C	37
2079	CWE-464	Addition of Data Structure Sentinel	C	14
72	CWE-248	Uncaught Exception	C++	25
2046	CWE-259	Use of Hard-coded Password	C++	48
2060	CWE-457	Use of Uninitialized Variable	C++	20
1989	CWE-89	Improper Neutralization of Special Elements used in an SQL Command	C++	34
114	CWE-532	Insertion of Sensitive Information into Log File	C	16
20	CWE-118	Incorrect Access of Indexable Resource	C	10
1887	CWE-22	Improper Limitation of a Pathname to a Restricted Directory	C	48
313	CWE-119	Improper Restriction of Operations within the Bounds of a Memory Buffer	C	12
2016	CWE-378	Creation of Temporary File With Insecure Permissions	C	12
149125	CWE-122	Heap-based Buffer Overflow	C	53
501035	CWE-563	Assignment to Variable without Use	C++	11
1852	CWE-170	Improper Null Termination	C	13
149163	CWE-99	Improper Control of Resource Identifiers	C	23
149241	CWE-78	Improper Neutralization of Special Elements used in an OS Command	C	41

Table 2: NIST SARD test cases that are used in this study. Each of these target test functions has exactly one line of source code that has been identified as being an example of a CWE. The number of lines of source code field is computed by the absolute number of lines of code syntax from the original dataset, not including lines containing only whitespace, and with all comments having been removed, and with no manual formatting (for example, curly brace placement).

To ground the GPT model output in the definitions of the Common Weakness Enumeration (CWE) framework, we utilize a technique known as retrieval augmented generation [28]. Specifically, we use a codebase known as `localGPT` [29] that uses langchain [30] to augment the capabilities of GPT models by using additional word embeddings of reference documents (not as part of the model pre-training). Specifically, the source documents are split into character maximum chunk sizes of 1000, with chunk overlap of 200 characters.

The localGPT prompt template is a combination of context and then the user prompt question. No chat history was used, meaning that each text generation call is independent. The prompt construction uses a system prompt [29] followed subsequently by the context of the generated word embeddings and then the user prompt. The system prompt used in this study, which is from a version of the localGPT codebase [29], is given below:

> You are a helpful assistant, you will use the provided context to answer user questions. Read the given context before answering questions and think step by step. If you can not answer a user question based on the provided context, inform the user. Do not use any other information for answering user. Provide a detailed answer to the question.

The reference documents that are used to provide context for the GPT model inference calls are the following: one text file for each of the 839 CWEs that are currently defined, where the text file contains the CWE number, the name of the software weakness, and the complete description of the CWE from Mitre [1]. Next is a text file with a brief definition of the CWE framework. And lastly, two books (PDF files of the books) on software, security, and vulnerabilities (specifically for the C family languages) [31, 32].

Table 2 outlines the exact code test cases that are analyzed in this study. Note that NIST SARD test case 148871 is the vulnerable code that was tracked by CVE-2009-3549, and test case 149199 is an example of an implementation of CVE-2002-1869. All of the selected NIST SARD test cases have a few specific properties that we manually selected these for. First, each test case is an example of a different CWE, which makes the results representative of a variety of different possible software weaknesses. Second, each test case contains exactly one line of source code that is an example of a particular CWE; this simplifies the parsing and accuracy metrics. Third, each test case contains no variable names, text, or function names that indicate whether the source code contains a vulnerability or not. In cases where there is a string or function name that indicates what type of example the source code is, we manually remove, or change, that string before parsing by the GPT models. Fourth, for no test cases does the vulnerable line of code occur more than once in the entire source code; this eliminates ambiguity for measuring the accuracy rate of the GPT models for identifying specific lines of source code. Any comments in the source code examples are removed before parsing by the GPT models so as to not give the GPT models natural language that describes the test cases.

Appendix B shows the exact source code for 10 of these NIST SARD test cases. The phrase *vulnerable line of source* will be used throughout the text to refer to the exact syntax of the vulnerable line of code – not line number. Line numbers can be somewhat ambiguous, especially for these C syntax test cases, and moreover having the GPT model output the exact source code makes checking the validity of that output much clearer.

The exact prompt that is used for all of the GPT models is given in the text box below. Note the use of the triple backticks to delineate where the source code is, and where the JSON datastructure is. The use of the triple backticks in the prompt allows the GPT outputs to be automatically parsed by using the triple backticks, and possibly the markdown code identifier, as the delimiter. Note that this triple backtick parsing does not always work, in particular if the JSON is not correctly formatted or the triple backticks are not correctly used in the GPT output. The prompt is intended to be very source code agnostic; there are no details given of the source code in the prompt. The prompt is also intended to not bias the output towards specifying a specific number of vulnerabilities. In the case of the 36 NIST SARD test cases used in this study, the number of CWEs is always exactly 1, however the goal is for this method to more general than only identifying a single CWE and moreover we want the GPT model to be able to fail in a variety of ways in order to rigorously evaluate its accuracy (in this case, identifying more CWEs than are present would constitute a high false positive rate).

[1] `https://cwe.mitre.org`

I am seeking assistance to analyze provided source code for the purpose of detecting potential Common Weakness Enumeration (CWE) vulnerabilities. The key requirement is for the analysis results to be exclusively formatted in JSON enclosed by triple backticks, listing only the CWEs that are detected as 'Present' in the code, thereby omitting any 'Not Present' entries for a cleaner output. The exact line, or lines, of code which have this CWE vulnerability needs to be specified in the returned JSON structure, along with a succinct description of what the vulnerability category is for this source code. The JSON output should enumerate each identified CWE with its identification number, the exact source code syntax that causes the CWE, and a description of the vulnerability, as shown in the example structure below:

````
```json
{"findings": [
{
"CWE_Number": "CWE-XX",
"Status": "Present",
"Source_Code": "Vulnerable source code syntax",
"Description:" "Summary of the category of software vulnerability,
        and why this code is vulnerable"
},
{
"CWE_Number": "CWE-YY",
"Status": "Present",
"Source_Code": "Vulnerable source code syntax",
"Description:" "Summary of the category of software vulnerability,
        and why this code is vulnerable"
}
]
}
```
````

Source code:

````
```
--Source Code--
```
````

Re-check that your output contains the required JSON data-structure before emitting text.

The actual GPT prompt is generated by replacing the vulnerable source code in place of the text `--Source Code--`.

The usage of requesting the vulnerable code syntax and the description of the vulnerability has two uses:

1. Ground the GPT response to only respond based on the provided source code
2. To check whether the provided syntax is correct. Because of potential ambiguity with line numbers, only exact syntax string matches are checked for, not integer line number responses.

The *description* field is not parsed for accuracy or correctness, but the text contained in that field is an interesting indicator of how correct (or incorrect) the GPT output is. For this reason, when reporting results (in Section 3), we provide specific instances of GPT outputs, including the description fields.

Some of the raw GPT output includes characters that are not ascii and can cause issues with saving and parsing the output. Therefore, the raw GPT output is encoded as utf-8 where character parsing errors are ignored and decoded into a parsable string using utf-8. The field in the JSON output `Source_Code` has repeated excess whitespace removed, as is done with the reference vulnerable line of source code from each of the NIST SARD samples; this ensures that when the string equality check is performed, the only relevant characteristic with respect to whitespace in the string that is being checked is that the whitespace is delineating characters (for example, repeated spaces, tabs, newlines do not impact this accuracy check).

The GPT output is (attempted to be) parsed in several stages, as follows:

1. First, if there are triple backtick delimiters in the output, then the output is split at those characters and the first enclosed set of strings is attempted to be parsed as a valid JSON structure. Interestingly, occasionally the GPT model will add in markdown code identifiers immediately after the triple backticks - in these cases we automatically check for these and remove them if they occur so that the structure can be parsed. These markdown identifiers are given in Appendix A.
2. If the previous step failed, then we attempt an unstructured JSON parsing approach, where output strings are incrementally deleted from the beginning and from the end until a JSON structure can be parsed (or, not).

Using this parsing, any valid JSON strings that are produced as part of the GPT output will be automatically parsed. This allows this study to perform a systematic evaluation of the accuracy rates of the GPT inference output. This type of data parsing is also highly scalable to a large number of static code samples - and importantly does not require any manual input, tuning, or processing. In other words, this analysis pipeline is entirely *automated*.

2.1 CWE Identification Accuracy Measures

This section lists several measures for quantifying the success rate of the GPT models for identifying known vulnerable source code.

The first four metrics that are checked are defined as follows:

1. Count of parsable GPT outputs (being able to extract a JSON structure and parse it into Python, with not necessarily correct JSON fields. For example a JSON with an empty list is a valid parse. However, this step does require that there is at least one JSON key, which is correctly named as `findings`. The subsequent value is allowed to be empty or have any sort of incorrect structure (which could denote that the model is returning no found CWEs) but we need at least this key to validate that a JSON datastructure was parsed.
2. The CWE field is labeled as `Present` is correct (meaning, the correct CW number is listed), but the identified source code is not correct (meaning, it is not the labeled line of code that has the vulnerability) and does not contain the vulnerable line of code either.
3. The CWE is correct and identified as `Present` and the vulnerable code is contained within the identified code syntax but also includes other lines of code from the test case that are not themselves vulnerable as defined by the NIST SARD dataset. These instances can be interpreted as being partially correct.
4. Count of fully correct instances where the CWE is correct (and identified as `Present`) and the identified line of code is correctly identified as the vulnerable code.

This last metric (metric number 4) is the most important accuracy measure because it counts out of the $1,000$ GPT outputs how many correctly performed the computation that we are attempting to prompt the GPT model to perform.

For none of these four metrics is the Description field, or any other additional field in the JSON, parsed for correctness or even considered. All of these measures are integers, where the larger the integer is the better the GPT model performed, but the absolute correct metric is measure 4. Note that the quantities measured in metrics 2, 3, and 4 (in the immediately preceding list of measures) are always disjoint; there are no instances that are counted in two or more of these measures. An important, and informative, aspect of these metrics is finding cases where the GPT output correctly identified the vulnerable line of code. However, since line numbers of the code are not checked or requested in the prompt (since they are somewhat ambiguously defined especially for C-family languages), a potential source of error in this quantification is if there exist multiple lines of code with the same syntax, but only one is vulnerable. This does happen, but only in one of the test cases. This is NIST SARD test case 149107, which is CWE-415 (Double Free) which by its definition contains two lines of memory free calls. Therefore, for this one test instance the identification of the vulnerable line of code is ambiguous, but for all others the unique identification is valid and correct.

Another set of metrics that are measured are binary classification error rates. This is computed by considering the GPT output for identified CWEs as a vector of True or False entries, with length equal to the number of valid CWEs (which is currently 839). This measure disregards both the requested `Source_Code` and `Description` fields (and in particular does not consider whether these entries even exist in the JSON structure. For all `CWE_Number` entries (that identifies a valid CWE number), and whose `Status` is exactly `Present` (allowing for different capitalizations), that is measured as a binary state of True. All other entries (including instances of identified CWE numbers that are not valid) are measured as binary classifications of False. JSON dictionaries that are completely empty are a valid GPT output that is identifying no CWEs (meaning a binary classification vector of length 839 that is all False entries). These measures are only computed on the outputs that can be parsed; all outputs that can not be parsed are not counted towards any classification accuracy.[2] These binary classification measures allow rates of false negatives, false positives, true negatives, and true positives to be measured, and then standard machine learning classification measures such as *recall* and *precision* can thus be computed. Precision is intuitively defined as the classifier (which in this case is a GPT language model) being able to not label as positive a sample that is negative. Recall is intuitively defined as the classifier (which in this case is a GPT language model) being able to find all of the positive samples. These metrics being 0 corresponds to the highest possible error rate, and 1 is the best measure these metrics can be.

All parsing steps require the field names in the JSON datastructure to be correct (correct spelling, no ancillary text), but differing capitalizations are always allowed.

3 Results

For each of the 36 NIST SARD test cases in Table 2, a total of 5,000 inference calls are performed using the methods described in Section 2 for the 5 different open-sourced GPT models. The accuracy measures from these outputs are then summarized using several high-level metrics. Tables 3 and 4 present two different accuracy measure summaries. Table 3 counts how many of the GPT outputs were parsable and correctly identified the CWE number and vulnerable line of code. Table 4 shows the binary classification measures recall and precision. Table 5 reports the absolute integer count of how many of the GPT outputs correctly identified the single line of vulnerable source code; importantly, these counts are independent of correct identification of the relevant CWE. These counts, as a proportion out of 1000 should be compared against uniform random sampling of the total lines of source code (shown in Table 2). Table 5 shows that the correct identification of vulnerable lines of code is much higher than the correct identification of the CWE number and the vulnerable line of source code, as measured in Table 3. Notably in Table 5, only for a few of the NIST SARD test cases is the count of ever correctly identifying the syntax of vulnerable line of source code zero across all GPT models and settings; namely test cases 149203, 148871, and 2046. For some of the test cases, the proportion of correctly identified vulnerable line of source code is very high. The test cases where the vulnerable line of source was not uniquely identified in Table 5 does not necessarily mean that the correct CWE identification was always 0 (when analyzing strictly whether the identified CWE was correct) - an example of this is test case 2046.

Table 4 shows that for the code test cases where the accuracy is 0, the 5 different language models do not perform uniformly bad – for some of the models the accuracy is always 0 for a given test case, but then when analyzed with a different model the accuracy is non-zero (an example of this is NIST SARD test case 500757). This means that some of the GPT models do perform better than other GPT models for specific test cases. And moreover, there is not consistently a single model that always outperforms the other models.

Table 6 shows the best performing GPT model and inference temperature in terms of recall and precision measures. Finally, Table 7 counts the most frequently identified false positives. The best performing GPT model and inference temperature shows that there is no specific model, or temperature setting, that consistently performs the best - and in in particular, every one of the tested GPT models was the best performer for at least one of the CWE test cases. This shows that this task does require diversity of the model architecture, and very likely the type of text on which the models were trained. Lastly, even with these best performing GPT models and inference temperatures some of the CWE test cases were never correctly identified, meaning that the accuracy rates for all GPT models and settings were always 0 - that happened for 11 out of 36 of the test cases as denoted by asterisks in Table 6.

Section 3.1 then gives several examples of correct JSON output from the GPT models where the vulnerable line of source code was identified, and where the description is correct in some cases and incorrect in other cases.

Table 7 quantifies the exact counts of false positive CWE id occurrences for the full matrix of the 5 GPT models and NIST SARD test cases. Specifically, for each NIST SARD test case and GPT model, Table 7 reports the most frequently mis-identified CWE number. Table 7 demonstrates two important trends. First, the proportion of false positives is *very* high, and this is consistent across the different GPT models. Second, for most test cases and GPT models the distribution of incorrect CWE ids is not uniformly distributed across the possible CWE numbers - it is heavily biased towards a few specific CWE numbers. For example, `Mistral-7B-Instruct-v0.1` shows a clear bias towards predicting CWE-124 for multiple test cases. Similarly, `Llama-2-70b-chat-hf` shows a consistent bias for (incorrectly) predicting CWE-123, and `Turdus` shows a consistent bias for (incorrectly) predicting CWE-476.

[2] In the case where a JSON entry contains multiple entries with the same CWE number, that case is treated as the GPT model having identified that CWE number only once; in other words, repeated entries are ignored. This specifically works for this accuracy metric of binary classification since all other fields (except `Status` are ignored.

NIST SARD Test Case	Llama-2-70b-chat-hf	zephyr-7b-alpha	zephyr-7b-beta	Mistral-7B-Instruct-v0.1	Turdus
1792 (CWE-79)	937, 1, 0, 0	82, 0, 0, 0	66, 0, 0, 1	430, 1, 0, 0	540, 0, 0, 3
500843 (CWE-476)	953, 9, 0, 2	66, 4, 0, 1	164, 1, 0, 1	378, 0, 0, 0	554, 24, 59, 14
1779 (CWE-463)	801, 0, 0, 0	62, 0, 0, 0	85, 0, 0, 0	280, 0, 0, 0	413, 0, 0, 0
1645 (CWE-20)	668, 7, 0, 0	278, 4, 0, 1	147, 15, 4, 4	704, 0, 0, 0	738, 8, 0, 1
149165 (CWE-121)	927, 3, 0, 325	183, 9, 1, 4	155, 1, 4, 10	734, 0, 0, 1	597, 0, 0, 0
2015 (CWE-329)	973, 0, 0, 0	118, 0, 0, 0	116, 0, 0, 0	345, 0, 0, 0	462, 0, 0, 0
500757 (CWE-787)	993, 0, 0, 0	128, 1, 0, 0	324, 0, 0, 0	308, 0, 0, 0	564, 1, 35, 12
149135 (CWE-489)	800, 0, 0, 0	82, 0, 0, 0	74, 0, 0, 0	575, 0, 0, 0	514, 0, 0, 0
149203 (CWE-416)	757, 0, 0, 0	174, 1, 0, 0	222, 0, 0, 0	572, 0, 0, 0	532, 4, 0, 0
149111 (CWE-134)	718, 297, 3, 56	286, 1, 0, 0	143, 0, 0, 0	672, 7, 0, 2	531, 0, 0, 1
149185 (CWE-391)	774, 0, 0, 0	286, 0, 0, 0	197, 0, 0, 0	685, 0, 0, 0	531, 0, 0, 0
1494 (CWE-74)	986, 0, 0, 0	208, 0, 0, 0	171, 0, 0, 0	689, 1, 0, 1	818, 0, 1, 0
149179 (CWE-401)	981, 5, 0, 0	207, 1, 0, 0	152, 4, 0, 2	599, 0, 0, 0	501, 2, 0, 0
149107 (CWE-415)	556, 0, 0, 0	228, 0, 0, 0	65, 0, 0, 0	664, 0, 0, 0	431, 0, 0, 0
148871 (CWE-188)	693, 0, 0, 0	153, 0, 0, 0	30, 0, 0, 0	474, 0, 0, 0	389, 0, 0, 0
149143 (CWE-120)	987, 14, 1, 27	213, 3, 4, 7	157, 0, 0, 0	740, 2, 0, 0	587, 4, 4, 52
149103 (CWE-367)	810, 0, 0, 0	71, 0, 0, 0	11, 0, 0, 0	387, 0, 0, 0	647, 0, 0, 0
1501 (CWE-822)	714, 0, 0, 0	168, 0, 0, 0	105, 0, 0, 0	695, 0, 0, 0	499, 0, 0, 0
149199 (CWE-412)	945, 0, 0, 0	285, 0, 0, 0	191, 0, 0, 0	701, 0, 0, 0	435, 1, 0, 0
149183 (CWE-468)	997, 0, 0, 0	275, 0, 0, 0	280, 0, 0, 0	737, 0, 0, 0	638, 0, 0, 0
149085 (CWE-244)	918, 0, 0, 0	89, 0, 0, 0	151, 0, 0, 0	539, 0, 0, 0	677, 0, 0, 0
2079 (CWE-464)	984, 0, 0, 0	245, 0, 0, 0	213, 0, 0, 0	568, 0, 0, 0	628, 0, 1, 0
72 (CWE-248)	877, 0, 0, 0	47, 0, 0, 0	193, 0, 0, 0	286, 0, 0, 0	528, 0, 0, 0
2046 (CWE-259)	885, 0, 0, 0	168, 1, 0, 0	146, 0, 0, 0	561, 0, 0, 0	700, 0, 0, 0
2060 (CWE-457)	545, 0, 0, 0	70, 0, 0, 0	110, 0, 0, 0	359, 0, 0, 0	559, 0, 0, 0
1989 (CWE-89)	837, 56, 2, 358	80, 30, 0, 12	78, 11, 6, 5	361, 0, 0, 0	357, 125, 10, 3
114 (CWE-532)	987, 0, 0, 0	318, 2, 0, 0	186, 0, 0, 0	767, 0, 0, 0	625, 0, 0, 0
20 (CWE-118)	410, 0, 0, 0	158, 0, 0, 0	139, 0, 0, 0	424, 0, 0, 0	391, 0, 0, 0
1887 (CWE-22)	948, 1, 0, 0	103, 0, 0, 0	62, 0, 0, 0	528, 0, 0, 0	577, 1, 0, 0
313 (CWE-119)	991, 0, 0, 1	308,i 2, 0, 1	138, 0, 0, 0	616, 72, 0, 4	739, 0, 0, 3
2016 (CWE-378)	428, 0, 0, 0	95, 0, 0, 0	178, 0, 0, 0	387, 0, 0, 0	487, 0, 0, 0
149125 (CWE-122)	852, 0, 0, 0	220, 1, 0, 7	87, 0, 0, 0	667, 0, 0, 0	477, 0, 1, 3
501035 (CWE-563)	851, 0, 0, 0	238, 0, 0, 0	123, 0, 0, 0	502, 0, 0, 0	290, 0, 0, 0
1852 (CWE-170)	986, 0, 0, 0	236, 0, 0, 0	202, 0, 0, 0	683, 0, 0, 0	670, 0, 0, 0
149163 (CWE-99)	803, 0, 0, 0	227, 0, 0, 0	95, 0, 0, 0	609, 0, 0, 0	574, 0, 0, 0
149241 (CWE-78)	956, 49, 0, 0	100, 7, 0, 2	96, 5, 0, 0	149, 0, 0, 0	323, 15, 1, 4

Table 3: Summary of the four accuracy metrics that take into account the identification, or failure to identify, the vulnerable source code, for each of the 5 GPT models. The four metrics are given as list of integers in each entry in the table. These measures, for each entry in the table, are taken from the 10 different inference temperature settings used for each model, meaning all four quantities reported in each cell are out of distinct $1,000$ GPT outputs. The accuracy metrics are described in Section 2.1, but briefly the first integer is a count of how many of the GPT outputs could be parsed into a datastructure, the second integer is that the identified CWE number was correct (and uniquely identified) but the identified line of code was not correct, the third integer is that the CWE number was correct (and uniquely identified) but the identified lines of code included the vulnerable line of code plus additional syntax from the source code, and the fourth integer is a count of how many of the GPT outputs correctly and uniquely identified both the CWE number and the vulnerable line of code.

3.1 Example Correctly Identified Vulnerable Lines of Code

This section enumerates several parsed JSON outputs from the various GPT models and code test cases. These outputs were hand selected to be representative of cases that are correct (in the sense that the CWE number is correctly identified, as is the vulnerable line of source code), but whose description fields are in some cases correct and in other cases incorrect. Showing explicit examples of the output is intended to primarily show the variability of the generated *Description* field, and also to show concretely what successful static code analysis identification

NIST SARD Test Case	Llama-2-70b-chat-hf	zephyr-7b-alpha	zephyr-7b-beta	Mistral-7B-Instruct-v0.1	Turdus
1792 (CWE-79)	0.022, 0.019	0, 0	0.015, 0.01	0.005, 0.01	0.026, 0.021
500843 (CWE-476)	0.012, 0.011	0.119, 0.101	0.055, 0.034	0, 0	0.224, 0.393
1779 (CWE-463)	0, 0	0, 0	0, 0	0, 0	0, 0
1645 (CWE-20)	0.225, 0.159	0.062, 0.047	0.435, 0.299	0, 0	0.008, 0.008
149165 (CWE-121)	0.386, 0.283	0.084, 0.072	0.221, 0.16	0.001, 0.002	0, 0
2015 (CWE-329)	0, 0	0, 0	0.008, 0.006	0, 0	0, 0
500757 (CWE-787)	0, 0	0.023, 0.022	0.012, 0.009	0, 0	0.091, 0.096
149135 (CWE-489)	0, 0	0, 0	0, 0	0, 0	0, 0
149203 (CWE-416)	0.0520, 0.016	0.139, 0.07	0.108, 0.046	0.006, 0.007	0.017, 0.012
149111 (CWE-134)	0.545, 0.454	0.011, 0.01	0.007, 0.006	0.041, 0.069	0.002, 0.002
149185 (CWE-391)	0, 0	0, 0	0, 0	0, 0	0, 0
1494 (CWE-74)	0, 0	0, 0	0, 0	0.003, 0.005	0.001, 0.001
149179 (CWE-401)	0.005, 0.005	0.024, 0.022	0.171, 0.117	0, 0	0.011, 0.011
149107 (CWE-415)	0, 0	0.018, 0.007	0, 0	0.002, 0.001	0, 0
148871 (CWE-188)	0, 0	0, 0	0, 0	0, 0	0, 0
149143 (CWE-120)	0.048, 0.041	0.094, 0.094	0, 0	0.009, 0.012	0.123, 0.125
149103 (CWE-367)	0.001, 0	0.014, 0.01	0, 0	0, 0	0, 0
1501 (CWE-822)	0, 0	0, 0	0, 0	0.001, 0.002	0, 0
149199 (CWE-412)	0, 0	0, 0	0, 0	0.003, 0.003	0.002, 0.002
149183 (CWE-468)	0, 0	0, 0	0, 0	0, 0	0, 0
149085 (CWE-244)	0, 0	0, 0	0, 0	0, 0	0, 0
2079 (CWE-464)	0, 0	0, 0	0, 0	0, 0	0.003, 0.003
72 (CWE-248)	0, 0	0, 0	0, 0	0, 0	0, 0
2046 (CWE-259)	0.148, 0.039	0.024, 0.016	0.027, 0.015	0.005, 0.004	0.001, 0.001
2060 (CWE-457)	0.002, 0.001	0, 0	0.009, 0.006	0, 0	0.002, 0.002
1989 (CWE-89)	0.602, 0.586	0.725, 0.617	0.654, 0.415	0.008, 0.011	0.762, 0.665
114 (CWE-532)	0, 0	0.006, 0.006	0, 0	0, 0	0, 0
20 (CWE-118)	0, 0	0, 0	0, 0	0, 0	0, 0
1887 (CWE-22)	0.657, 0.186	0.01, 0.006	0, 0	0, 0	0.003, 0.002
313 (CWE-119)	0.001, 0.001	0.019, 0.019	0, 0	0.136, 0.16	0.015, 0.015
2016 (CWE-378)	0, 0	0, 0	0, 0	0, 0	0, 0
149125 (CWE-122)	0.015, 0.005	0.086, 0.053	0.149, 0.082	0.004, 0.005	0.038, 0.028
501035 (CWE-563)	0, 0	0, 0	0, 0	0, 0	0, 0
1852 (CWE-170)	0, 0	0, 0	0, 0	0, 0	0, 0
149163 (CWE-99)	0, 0	0, 0	0, 0	0.002, 0.002	0, 0
149241 (CWE-78)	0.801, 0.439	0.12, 0.074	0.073, 0.04	0, 0	0.087, 0.104

Table 4: Recall and Precision binary classification metrics for each of the 5 GPT models, taken as the complete vector of 100 inference calls for each of the 10 temperature settings. Of these $1,000$ GPT outputs, only those that were able to be parsed contribute to the recall and precision accuracy measures. All quantities are rounded to 3 decimal places.

looks like from the GPT models.

No text changes were made to these GPT outputs except for formatting to fit within the text boxes. Note that these outputs are specifically examples of text from the enclosed triple backtick fields - any additional prose or output from the model is not shown.

NIST SARD Test Case	Llama-2-70b-chat-hf	zephyr-7b-alpha	zephyr-7b-beta	Mistral-7B-Instruct-v0.1	Turdus
1792 (CWE-79)	0	9	3	48	120
500843 (CWE-476)	181	8	10	9	60
1779 (CWE-463)	284	3	0	13	13
1645 (CWE-20)	1	23	6	156	157
149165 (CWE-121)	627	101	33	399	274
2015 (CWE-329)	328	12	5	94	46
500757 (CWE-787)	268	89	47	169	98
149135 (CWE-489)	19	6	1	0	17
149203 (CWE-416)	0	0	0	0	0
149111 (CWE-134)	59	224	48	36	281
149185 (CWE-391)	543	94	15	5	7
1494 (CWE-74)	950	188	47	155	368
149179 (CWE-401)	0	24	10	0	18
149107 (CWE-415)	0	6	2	2	0
148871 (CWE-188)	0	0	0	0	0
149143 (CWE-120)	686	144	42	26	621
149103 (CWE-367)	0	1	0	0	1
1501 (CWE-822)	3	5	0	0	3
149199 (CWE-412)	0	2	0	0	0
149183 (CWE-468)	158	162	90	19	151
149085 (CWE-244)	12	21	8	14	6
2079 (CWE-464)	3	5	2	0	0
72 (CWE-248)	86	3	6	2	16
2046 (CWE-259)	0	0	0	0	0
2060 (CWE-457)	0	3	1	0	1
1989 (CWE-89)	403	13	6	0	3
114 (CWE-532)	0	2	0	0	0
20 (CWE-118)	3	7	3	1	5
1887 (CWE-22)	0	0	0	0	0
313 (CWE-119)	941	258	59	54	416
2016 (CWE-378)	0	1	0	0	0
149125 (CWE-122)	2	59	3	7	67
501035 (CWE-563)	20	16	2	16	6
1852 (CWE-170)	3	24	8	1	42
149163 (CWE-99)	0	0	0	0	46
149241 (CWE-78)	0	4	0	0	11

Table 5: Count of unique and correct identification of the vulnerable line of source code, but not necessarily correctly identified CWE number (or being qualified as being present, or not present). The maximum possible quantity reported in each cell is 1000.

Example of correct CWE number and code line identification in a JSON datastructure format from NIST SARD Test Case 500843. Produced from the Turdus GPT model with temperature 0.8. Note that the generated description is partially correct - CWE-476 is NULL pointer dereference but the description also mentions Divison-by-Zero which is not correct.

```
{'findings': [{'CWE_Number': 'CWE-476',
'Status': 'Present',
'Source_Code': 'first->str();',
'Description': "Summary of the category of software vulnerability (Division By Zero)
due to potential NULL pointer dereference when calling str() on 'first'"}]}
```

NIST SARD Test Case	Best performing GPT model and inference temperature (recall, precision, proportion of parseability)
1792 (CWE-79)	zephyr-7b-beta 0.4: (0.125, 0.0667, 0.08)
500843 (CWE-476)	Turdus 0.1: (0.319, 0.6, 0.47)
1779 (CWE-463)	*
1645 (CWE-20)	zephyr-7b-beta 0.4: (0.611, 0.297, 0.18)
149165 (CWE-121)	Llama-2-70b-chat-hf 0.1: (1.0, 1.0, 1.0)
2015 (CWE-329)	zephyr-7b-beta 0.7: (0.077, 0.048, 0.13)
500757 (CWE-787)	Turdus 0.9: (0.183, 0.177, 0.6)
149135 (CWE-489)	*
149203 (CWE-416)	zephyr-7b-alpha 0.8: (0.24, 0.12, 0.25)
149111 (CWE-134)	Llama-2-70b-chat-hf 0.5: (0.727, 0.675, 0.77)
149185 (CWE-391)	*
1494 (CWE-74)	Mistral-7B-Instruct-v0.1 0.9: (0.029, 0.033, 0.67)
149179 (CWE-401)	zephyr-7b-beta 0.3: (0.25, 0.167, 0.16)
149107 (CWE-415)	zephyr-7b-alpha 0.2: (0.064, 0.029, 0.31)
148871 (CWE-188)	*
149143 (CWE-120)	Turdus 1.0: (0.237, 0.229, 0.8)
149103 (CWE-367)	zephyr-7b-alpha 0.6: (0.111, 0.083, 0.09)
1501 (CWE-822)	Mistral-7B-Instruct-v0.1 0.7: (0.014, 0.015, 0.73)
149199 (CWE-412)	Mistral-7B-Instruct-v0.1 0.8: (0.019, 0.02, 0.53)
149183 (CWE-468)	*
149085 (CWE-244)	*
2079 (CWE-464)	Turdus 0.9: (0.0159, 0.0175, 0.63)
72 (CWE-248)	*
2046 (CWE-259)	Llama-2-70b-chat-hf 0.1: (0.24, 0.0559, 1.0)
2060 (CWE-457)	zephyr-7b-beta 0.5: (0.1, 0.0556, 0.1)
1989 (CWE-89)	Llama-2-70b-chat-hf 0.1: (1.0, 1.0, 1.0)
114 (CWE-532)	zephyr-7b-alpha 0.9: (0.0645, 0.0689, 0.31)
20 (CWE-118)	*
1887 (CWE-22)	Llama-2-70b-chat-hf 0.1: (1.0, 0.304, 1.0)
313 (CWE-119)	Mistral-7B-Instruct-v0.1 0.1: (0.48, 0.48, 0.77)
2016 (CWE-378)	*
149125 (CWE-122)	zephyr-7b-beta 0.5: (0.46, 0.15, 0.13)
501035 (CWE-563)	*
1852 (CWE-170)	*
149163 (CWE-99)	Mistral-7B-Instruct-v0.1 0.8: (0.0196, 0.026, 0.51)
149241 (CWE-78)	Llama-2-70b-chat-hf 0.1: (1.0, 0.5, 1.0)

Table 6: Best Recall and Precision binary classification metrics (specifically the best mean of the recall and precision measures), in terms of GPT model and inference temperature, for each CWE NIST SARD test case. Data presentation is GPT model name: inference temperature (recall, precision, parsing rate). These metrics are taken from the dataset of 100 inference calls (for the best performing GPT model and inference temperature). The proportion of the 100 GPT outputs that were able to be parsed and to have a binary vector extracted is also reported as the third real number. The closer these three metrics are to 1.0, the better the GPT model performed. The proportion of the outputs that could be parsed are not used for computing the best performing model and inference temperature, but it is useful to show the parse-ability success rate. These recall and precision metrics are quantifying the GPT models ability to correctly identify both the CWE number and the vulnerable line of source code. * denotes that all recall and precision metrics were 0 for all GPT models and settings. Note that there is occasionally a complication that for all 100 GPT output strings there are no parse-able outputs, but in these cases there are just no metrics that can be computed. If any of the settings resulted in the same F1 score, the GPT configuration with higher parsability rates is reported.

NIST SARD Test Case	Llama-2-70b-chat-hf	zephyr-7b-alpha	zephyr-7b-beta	Mistral-7B-Instruct-v0.1	Turdus
1792 (CWE-79)	CWE-134 452x	CWE-89 29x	CWE-20 32x	CWE-134 35x	CWE-476 164x
500843 (CWE-476)	CWE-123 333x	CWE-120 6x	CWE-20 77x	CWE-124 31x	CWE-126 46x
1779 (CWE-463)	CWE-123 373x	CWE-121 24x	CWE-121 28x	CWE-124 64x	CWE-126 135x
1645 (CWE-20)	CWE-78 395x	CWE-78 221x	CWE-78 51x	CWE-124 112x	CWE-78 437x
149165 (CWE-121)	CWE-123 274x	CWE-362 66x	CWE-125 57x	CWE-116 143x	CWE-126 197x
2015 (CWE-329)	CWE-327 357x	CWE-382 45x	CWE-327 50x	CWE-125 47x	CWE-404 52x
500757 (CWE-787)	CWE-121 474x	CWE-476 39x	CWE-20 147x	CWE-124 106x	CWE-126 109x
149135 (CWE-489)	CWE-123 281x	CWE-20 20x	CWE-20 49x	CWE-434 76x	CWE-89 198x
149203 (CWE-416)	CWE-125 327x	CWE-125 85x	CWE-125 111x	CWE-129 130x	CWE-476 228x
149111 (CWE-134)	CWE-123 253x	CWE-125 76x	CWE-125 42x	CWE-124 77x	CWE-787 88x
149185 (CWE-391)	CWE-20 255x	CWE-369 34x	CWE-20 66x	CWE-116 53x	CWE-369 201x
1494 (CWE-74)	CWE-123 860x	CWE-121 100x	CWE-20 66x	CWE-124 137x	CWE-126 320x
149179 (CWE-401)	CWE-121 403x	CWE-476 111x	CWE-121 41x	CWE-129 134x	CWE-476 153x
149107 (CWE-415)	CWE-123 245x	CWE-416 121x	CWE-416 24x	CWE-116 130x	CWE-405 149x
148871 (CWE-188)	CWE-123 503x	CWE-434 27x	CWE-20 10x	CWE-134 84x	CWE-400 53x
149143 (CWE-120)	CWE-123 696x	CWE-121 127x	CWE-121 74x	CWE-124 195x	CWE-126 372x
149103 (CWE-367)	CWE-123 391x	CWE-125 21x	CWE-125 4x	CWE-134 69x	CWE-476 257x
1501 (CWE-822)	CWE-134 328x	CWE-125 22x	CWE-20 38x	CWE-129 119x	CWE-476 165x
149199 (CWE-412)	CWE-23 677x	CWE-476 72x	CWE-20 54x	CWE-124 165x	CWE-476 100x
149183 (CWE-468)	CWE-123 835x	CWE-125 74x	CWE-121 79x	CWE-116 137x	CWE-126 145x
149085 (CWE-244)	CWE-123 647x	CWE-121 36x	CWE-121 93x	CWE-129 157x	CWE-120 209x
2079 (CWE-464)	CWE-123 501x	CWE-125 79x	CWE-121 99x	CWE-124 156x	CWE-126 251x
72 (CWE-248)	CWE-123 357x	CWE-125 12x CWE-121 12x	CWE-125 95x	CWE-124 42x	CWE-126 123x
2046 (CWE-259)	CWE-284 753x	CWE-20 36x	CWE-20 102x	CWE-125 179x	CWE-89 255x
2060 (CWE-457)	CWE-416 243x	CWE-416 29x	CWE-121 31x	CWE-124 86x	CWE-416 105x
1989 (CWE-89)	CWE-78 73x	CWE-352 8x	CWE-20 31x	CWE-434 31x	CWE-476 46x
114 (CWE-532)	CWE-20 486x	CWE-200 99x	CWE-20 107x	CWE-125 150x	CWE-20 119x
20 (CWE-118)	CWE-190 200x	CWE-190 26x	CWE-190 39x	CWE-134 34x CWE-116 34x	CWE-190 85x
1887 (CWE-22)	CWE-20 758x	CWE-125 44x	CWE-125 28x	CWE-129 57x	CWE-476 344x
313 (CWE-119)	CWE-123 733x	CWE-121 130x	CWE-121 66x	CWE-124 147x	CWE-126 400x
2016 (CWE-378)	CWE-22 140x	CWE-121 11x	CWE-20 73x	CWE-124 56x	CWE-404 119x
149125 (CWE-122)	CWE-123 615x	CWE-125 57x	CWE-121 40x	CWE-124 106x	CWE-476 124x
501035 (CWE-563)	CWE-20 593x	CWE-125 55x	CWE-20 47x	CWE-129 61x	CWE-476 73x
1852 (CWE-170)	CWE-123 802x	CWE-125 48x	CWE-125 68x	CWE-124 69x	CWE-476 203x
149163 (CWE-99)	CWE-123 244x	CWE-434 71x	CWE-20 45x	CWE-124 67x	CWE-434 99x
149241 (CWE-78)	CWE-20 715x	CWE-120 25x	CWE-121 37x	CWE-124 11x	CWE-476 68x

Table 7: Distribution of False Positives; the most frequent incorrectly identified CWE's (labeled as being *Present* in the returned JSON datastructure) for each of the GPT models and each of the NIST SARD test cases. The frequency counts are from the totals out of the 10 temperature settings, meaning that the maximum possible count in each cell is 1000x. The proportion of the GPT output that gave that incorrect CWE is reported to the right of the CWE number. The counts are obtained from all parse-able CWE entries, not just those that identify only a single CWE. If there are multiple CWEs with most frequency counts, all of the CWEs with the same most frequent counts are shown.

Example of correct CWE number and code line identification in a JSON datastructure format from NIST SARD Test Case 500843. Produced from the zephyr-7b-alpha GPT model with temperature 0.7. Note that the generated description is entirely incorrect.

```
{'findings': [{'CWE_Number': 'CWE-476',
'Status': 'Present',
'Source_Code': 'first->str();',
'Description': "The function 'f' takes a linked list of Tokens as an argument.
If the last token in the list is a string, that string will be printed to stdout.
This code is vulnerable to a format string attack because the 'f' function is using
the'str' method of the last token in the list, which could contain a format string.
An attacker can pass a format string as an argument to this function and potentially
execute arbitrary code or cause a Denial of Service."}]}
```

Example of correct CWE number and code line identification in a JSON datastructure format from NIST SARD Test Case 500843. Produced from the Turdus GPT model with temperature 1.0. Note that the generated description is entirely incorrect.

```
{'findings': [{'CWE_Number': 'CWE-476',
'Status': 'Present',
'Source_Code': 'first->str();',
'Description': "This code is vulnerable as it executes an untrusted input without proper
validation before calling'str()'. This may result in memory corruption issues
such as buffer overflow."}]}
```

Example of correct CWE number and code line identification in a JSON datastructure format from NIST SARD Test Case 500843. Produced from the Llama-2-70b-chat-hf GPT model with temperature 0.8. Note that the provided description is mostly correct in that the correct vulnerability name was identified but there is not a function named f in the source code.

```
{'findings': [{'CWE_Number': 'CWE-476',
'Status': 'Present',
'Source_Code': 'first->str();',
'Description': 'Potential null pointer dereference vulnerability in function f'}]}
```

Example of correct CWE number and code line identification in a JSON datastructure format from NIST SARD Test Case 1792. Produced from the Turdus GPT model with temperature 0.3. Note that the generated description is correct.

```
{'findings':
[{'CWE_Number': 'CWE-79',
'Status': 'Present',
'Source_Code': 'fprintf(cgiOut, "Value number %d = %s<br />", i, q[i]);',
'Description': 'This code is vulnerable to Cross-site Scripting (XSS) attacks due to
insufficient input validation and sanitization.'}]}
```

Example of correct CWE number and code line identification in a JSON datastructure format from NIST SARD Test Case 1792. Produced from the Turdus GPT model with temperature 0.4. Note that the generated description is correct.

```
{'findings': [{'CWE_Number': 'CWE-79',
'Status': 'Present',
'Source_Code': 'fprintf(cgiOut, "Value number %d = %s<br />", i, q[i]);',
'Description': 'The code is vulnerable to Improper Neutralization of
Input During Web Page Generation (CWE-79), as it uses sprintf() without
validating or neutralizing user input in the web page generation process.'}]}
```

Example of correct CWE number and code line identification in a JSON datastructure format from NIST SARD Test Case 1792. Produced from the Turdus GPT model with temperature 0.4. Note that the generated description is correct except that the identified print statement in the description text is incorrect.

```
{'findings': [{'CWE_Number': 'CWE-79',
'Status': 'Present',
'Source_Code': 'fprintf(cgiOut, "Value number %d = %s<br />", i, q[i]);',
'Description': 'This code is vulnerable as it uses sprintf() on user-supplied
input without proper validation or sanitization, which can lead to
security risks such as Cross-Site Scripting (XSS)'}]}
```

Example of correct CWE number and code line identification in a JSON datastructure format from NIST SARD Test Case 1645. Produced from the Turdus GPT model with temperature 0.5. The generated Description field in this example is correct.

```
{'findings': [{'CWE_Number': 'CWE-20',
'Status': 'Present',
'Source_Code': 'system(buf);',
'Description': "This code falls victim to Improper Input Validation where user input isn't
validated before being used in a system() call, potentially allowing malicious
commands to be executed."}]}
```

Example of correct CWE number and code line identification in a JSON datastructure format from NIST SARD Test Case 1645. Produced from the zephyr-7b-alpha GPT model with temperature 0.9. The generated Description field in this example is correct.

```
{'findings': [{'CWE_Number': 'CWE-20',
'Status': 'Present',
'Source_Code': 'system(buf);',
'Description': 'The application uses system() to execute a user-supplied string without
sanitizing the input, which is a potential injection vulnerability (CWE-20).'}]}
```

Example of correct CWE number and code line identification in a JSON datastructure format from NIST SARD Test Case 149165. Produced from the Llama-2-70b-chat-hf GPT model with temperature 0.9. The generated Description field in this example is entirely incorrect.

```
{'findings': [{'CWE_Number': 'CWE-121',
'Status': 'Present',
'Source_Code': "buffer[plop()] = '!';", '
Description': 'The use of a weak random number generator can lead to predictable output,
which can be exploited by an attacker to execute arbitrary code.'}]}
```

Example of correct CWE number and code line identification in a JSON datastructure format from NIST SARD Test Case 149165. Produced from the Llama-2-70b-chat-hf GPT model with temperature 0.1. The generated Description field in this example is correct.

```
{'findings': [{'CWE_Number': 'CWE-121',
'Status': 'Present',
'Source_Code': "buffer[plop()] = '!';",
'Description': "Buffer overflow vulnerability in the plop() function.
The buffer 'buffer' is not large enough to hold the resulting value,
which can cause a buffer overflow."}]}
```

Example of correct CWE number and code line identification in a JSON datastructure format from NIST SARD Test Case 149165. Produced from the Mistral-7B-Instruct-v0.1 GPT model with temperature 0.6. The generated Description field in this example is correct.

```
{'findings': [{'CWE_Number': 'CWE-121',
'Status': 'Present',
'Source_Code': "buffer[plop()] = '!';",
'Description': 'Buffer overflow vulnerability'}]}
```

Example of correct CWE number and code line identification in a JSON datastructure format from NIST SARD Test Case 149165. Produced from the zephyr-7b-beta GPT model with temperature 0.2. The generated Description field in this example is correct.

```
{'findings': [{'CWE_Number': 'CWE-121',
'Status': 'Present',
'Source_Code': "buffer[plop()] = '!';",
'Description': 'Buffer overflow vulnerability due to writing beyond the bounds
of the buffer.'}]}
```

Example of correct CWE number and code line identification in a JSON datastructure format from NIST SARD Test Case 149165. Produced from the zephyr-7b-beta GPT model with temperature 0.4. The generated Description field in this example is not correct.

```
{'findings': [{'CWE_Number': 'CWE-121',
'Status': 'Present',
'Source_Code': "buffer[plop()] = '!';",
'Description': 'Use of hard-coded values in security-sensitive contexts'}]}
```

Example of correct CWE number and code line identification in a JSON datastructure format from NIST SARD Test Case 500757. Produced from the Turdus GPT model with temperature 0.8. The generated Description field in this example is correct.

```
{'findings': [{'CWE_Number': 'CWE-787',
'Status': 'Present',
'Source_Code': 'p[10] = 7;',
'Description': "This code falls under Out-of-Bounds Write category (CWE-787),
as it attempts to write a value to an index (10) beyond the allocated
memory space of 'p', potentially causing undefined behavior."}]}
```

Example of correct CWE number and code line identification in a JSON datastructure format from NIST SARD Test Case 1989. Produced from the Turdus GPT model with temperature 0.9. The generated Description field in this example is correct.

```
{'findings': [{'CWE_Number': 'CWE-89',
'Status': 'Present',
'Source_Code': 'queryStr += (" OR Value = \'" + *iter + "\'");',
'Description': 'This code suffers from the SQL injection vulnerability since user inputs
are being concatenated directly into an SQL query without any sanitation or
parameterized queries, making it vulnerable to malicious input manipulation.'}]}
```

Example of correct CWE number and code line identification in a JSON datastructure format from NIST SARD Test Case 149111. Produced from the Llama-2-70b-chat-hf GPT model with temperature 0.3. The generated Description field in this example is correct.

```
{'findings': [{'CWE_Number': 'CWE-134',
'Status': 'Present',
'Source_Code': 'printf(container.fmt);',
'Description': 'The use of user-supplied data in a format string
vulnerable to exploitation.'}]}
```

Example of correct CWE number and code line identification in a JSON datastructure format from NIST SARD Test Case 149111. Produced from the Mistral-7B-Instruct-v0.1 GPT model with temperature 0.7. The generated Description field in this example is correct.

```
{'findings': [{'CWE_Number': 'CWE-134',
'Status': 'Present',
'Source_Code': 'printf(container.fmt);',
'Description': 'Format string vulnerability can lead to buffer overflow
or other security vulnerabilities.'}]}
```

Example of correct CWE number and code line identification in a JSON datastructure format from NIST SARD Test Case 149241. Produced from the zephyr-7b-alpha GPT model with temperature 0.2. The generated Description field in this example is correct.

```
{'findings': [{'CWE_Number': 'CWE-78',
'Status': 'Present',
'Source_Code': 'if (system(command) < 0)',
'Description': 'Command Injection vulnerability, where an attacker can inject malicious
commands into the command parameter, which can be executed by the system() function.'}]}
```

Example of correct CWE number and code line identification in a JSON datastructure format from NIST SARD Test Case 313. Produced from the Llama-2-70b-chat-hf GPT model with temperature 0.6. The generated Description field in this example is correct.

```
{'findings': [{'CWE_Number': 'CWE-119',
'Status': 'Present',
'Source_Code': 'strncpy(buf, src, 11 % i);',
'Description': 'Buffer overflow vulnerability in strncpy() function'}]}
```

4 Discussion and Conclusion

This study has examined the task of correctly identifying software weaknesses and vulnerabilities in software using open source Generative Pre-Trained Transformer (GPT) models. This was able to be examined because of two specific reasons - the first is the existence of the NIST SARD dataset, which is a very well labelled dataset that includes the vulnerable line of code and the relevant CWE. The second is the specific prompting template usage that allows fully automated extraction of JSON output for the vulnerability detection - and therefore error rates can be quantified for a large range of NIST SARD test cases, GPT models, and GPT model settings. Thus, this study reports a systematic analysis of automated vulnerability detection using GPT models.

In general, our findings are that the correct CWE identification rate varies significantly depending on the specific code test case that is analyzed. For some test cases the correct CWE identification (and relevant line code of syntax) is surprisingly high, but for other test cases the identification accuracy rate is always 0 for all GPT models and temperature settings. In general, the error rates of this vulnerability detection is too high – in particular, the false positive rate is very high. However, it is notable that for some cases, seemingly the more popular classes of CWEs that are mentioned in the type of internet text on which these GPT models were trained, the identification rates are quite good. As specific examples of the lowest error rate results (from Table 6), we list several specific examples of CWE identification binary classification accuracy rates (without considering rates for correctly identifying the vulnerable line of source code). Test case 1989 had a best recall of 0.928, a best precision of 0.709 (with a parsability rate of 0.42) when analyzed with `Llama-2-70b-chat-hf` GPT model and temperature 0.1. Test case 149165 had a best recall of 1.0, a best precision of 1.0 (with a parsability rate of 1.0) when analyzed with `Llama-2-70b-chat-hf` GPT model and temperature 0.1. Test case 149111 had a best recall of 0.727, a best precision of 0.675 (with a parsability rate of 0.77) when analyzed with `Llama-2-70b-chat-hf` GPT model and temperature 0.5. Test case 149241 had a best recall of 1.0, a best precision of 0.5 (with a parsability rate of 1.0) when analyzed with `Llama-2-70b-chat-hf` GPT model and temperature 0.1. Given the high accuracy rates of at least some of these test cases, it seems plausible that some of the current open source GPT models, if thorough verification is used, could correctly identify certain classes of vulnerabilities, such as buffer overflows, with very low error rates.

It should be emphasized that this particular software security analysis task requires a number of semantic connections to be made in the GPT model inference process (and of course, the correct specification of the JSON syntax) in order for the (automated) vulnerability identification to be correct. It is not clear to what extent a pre-trained language model can learn these semantic connections - but it is possible least for some of the CWEs, these connections were learned in the standard open source GPT models that we tested. This is is a non-trivial finding, and suggests that GPT models do have some capability to identify security vulnerabilities. However, it also needs to be emphasized that such usage of current GPT models should not be used in cases where the accuracy can not be tested. In other words, these GPT models and vulnerability identification prompting should not be used in software engineering tasks – they should be used in cases where careful and precise testing of the GPT output can be performed for the purpose of examining specific characteristics of source code of interest. The primary reason for this is that the false positive rate is high - and not only is it high but the false positives are biased towards incorrectly identifying specific CWEs. In other words, simply measuring which CWE is predicted most frequently in a GPT output distribution (for a non-zero inference temperature) will typically not be correct.

The primary limitation with increasing the scope of more code test cases to more thoroughly test the capabilities of GPT models for software security analysis is the total amount of compute time that is required for performing these inference calls, especially in a systematic study of varying inference parameters. Future studies that increase the number of labeled source code examples (e.g., from the extensive amount of data available in the NIST SARD dataset) that are used for testing the accuracy GPT automated vulnerability identification would be very relevant.

An aspect of this study that is unclear is *what* details of the source code are the GPT models using to infer relevant CWE's in the source code. In particular, what semantic details are used to parse whether a CWE is more likely or less likely. In some cases, it could be that context, such as what types of libraries are used or what types of strings are being constructed (such as SQL query strings), is sufficient to reduce the total number of possibly applicable CWEs so that random guessing among those feasible candidates accounts for some of the GPT model's performance. The metrics in this study do not determine what components of the source code are important; this would require quantifying the explainability of GPT models [33–37] for this task. We leave this study open to future research, and emphasize that this type of detailed explainability analysis is absolutely a critical component of study before these GPT models are incorporated into any type of vulnerability detection in production software development and testing.

5 Acknowledgments

Sandia National Laboratories is a multi-mission laboratory managed and operated by National Technology & Engineering Solutions of Sandia, LLC (NTESS), a wholly owned subsidiary of Honeywell International Inc., for the U.S. Department of Energy's National Nuclear Security Administration (DOE/NNSA) under contract DE-NA0003525. This written work is authored by an employee of NTESS. The employee, not NTESS, owns the right, title and interest in and to the written work and is responsible for its contents. Any subjective views or opinions that might be expressed in the written work do not necessarily represent the views of the U.S. Government. The publisher acknowledges that the U.S. Government retains a non-exclusive, paid-up, irrevocable, world-wide license to publish or reproduce the published form of this written work or allow others to do so, for U.S. Government purposes. The DOE will provide public access to results of federally sponsored research in accordance with the DOE Public Access Plan.

References

[1] Ashish Vaswani et al. *Attention Is All You Need*. 2023. arXiv: 1706.03762 [cs.CL].
[2] Mark Chen et al. *Evaluating Large Language Models Trained on Code*. 2021. arXiv: 2107.03374 [cs.LG].
[3] Tom B. Brown et al. *Language Models are Few-Shot Learners*. 2020. arXiv: 2005.14165 [cs.CL].
[4] OpenAI et al. *GPT-4 Technical Report*. 2023. arXiv: 2303.08774 [cs.CL].
[5] Hugo Touvron et al. *Llama 2: Open Foundation and Fine-Tuned Chat Models*. 2023. arXiv: 2307.09288 [cs.CL].
[6] François Charton. *Learning the greatest common divisor: explaining transformer predictions*. 2024. arXiv: 2308.15594 [cs.LG].
[7] Paul Black. "SARD: Thousands of Reference Programs for Software Assurance". en. In: 5 (2017). URL: https://tsapps.nist.gov/publication/get_pdf.cfm?pub_id=923127.
[8] Steve Christey et al. "Common weakness enumeration". In: *Mitre Corporation* (2013).
[9] Robert A. Martin and Sean Barnum. "Common weakness enumeration (CWE) status update". In: *Ada Lett.* XXVIII.1 (2008), 88–91. ISSN: 1094-3641. DOI: 10.1145/1387830.1387835. URL: https://doi.org/10.1145/1387830.1387835.
[10] P. Louridas. "Static code analysis". In: *IEEE Software* 23.4 (2006), pp. 58–61. DOI: 10.1109/MS.2006.114.
[11] Katerina Goseva-Popstojanova and Andrei Perhinschi. "On the capability of static code analysis to detect security vulnerabilities". In: *Information and Software Technology* 68 (2015), pp. 18–33.
[12] J. Viega et al. "ITS4: a static vulnerability scanner for C and C++ code". In: *Proceedings 16th Annual Computer Security Applications Conference (ACSAC'00)*. 2000, pp. 257–267. DOI: 10.1109/ACSAC.2000.898880.
[13] Arvinder Kaur and Ruchikaa Nayyar. "A comparative study of static code analysis tools for vulnerability detection in c/c++ and java source code". In: *Procedia Computer Science* 171 (2020), pp. 2023–2029.
[14] N. Jovanovic, C. Kruegel, and E. Kirda. "Pixy: a static analysis tool for detecting Web application vulnerabilities". In: *2006 IEEE Symposium on Security and Privacy (S&P'06)*. 2006, 6 pp.–263. DOI: 10.1109/SP.2006.29.
[15] Shobha Tyagi and Krishan Kumar. "Evaluation of Static Web Vulnerability Analysis Tools". In: *2018 Fifth International Conference on Parallel, Distributed and Grid Computing (PDGC)*. 2018, pp. 1–6. DOI: 10.1109/PDGC.2018.8745996.
[16] V Benjamin Livshits and Monica S Lam. "Finding Security Vulnerabilities in Java Applications with Static Analysis." In: *USENIX security symposium*. Vol. 14. 2005, pp. 18–18.
[17] Stephan Lipp, Sebastian Banescu, and Alexander Pretschner. "An empirical study on the effectiveness of static C code analyzers for vulnerability detection". In: *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*. ISSTA 2022. Virtual, South Korea: Association for Computing Machinery, 2022, 544–555. ISBN: 9781450393799. DOI: 10.1145/3533767.3534380. URL: https://doi.org/10.1145/3533767.3534380.
[18] Foteini Cheirdari and George Karabatis. "Analyzing False Positive Source Code Vulnerabilities Using Static Analysis Tools". In: *2018 IEEE International Conference on Big Data (Big Data)*. 2018, pp. 4782–4788. DOI: 10.1109/BigData.2018.8622456.
[19] Hammond Pearce et al. "Examining Zero-Shot Vulnerability Repair with Large Language Models". In: *2023 IEEE Symposium on Security and Privacy (SP)*. 2023, pp. 2339–2356. DOI: 10.1109/SP46215.2023.10179324.
[20] Ruitong Liu et al. *Source Code Vulnerability Detection: Combining Code Language Models and Code Property Graphs*. 2024. arXiv: 2404.14719 [cs.CR].
[21] Rafael Rafailov et al. *Direct Preference Optimization: Your Language Model is Secretly a Reward Model*. 2023. arXiv: 2305.18290 [cs.LG].
[22] Lewis Tunstall et al. *Zephyr: Direct Distillation of LM Alignment*. 2023. arXiv: 2310.16944 [cs.LG].
[23] Albert Q. Jiang et al. *Mistral 7B*. 2023. arXiv: 2310.06825 [cs.CL].
[24] UDK dot AI, Daniel Devatman Hromada. *Turdus (Revision 923c305)*. 2024. DOI: 10.57967/hf/1611. URL: https://huggingface.co/udkai/Turdus.
[25] Thomas Wolf et al. *HuggingFace's Transformers: State-of-the-art Natural Language Processing*. 2020. arXiv: 1910.03771 [cs.CL].
[26] Adam Paszke et al. "PyTorch: An Imperative Style, High-Performance Deep Learning Library". In: *Advances in Neural Information Processing Systems 32*. Ed. by H. Wallach et al. Curran Associates, Inc., 2019,

pp. 8024–8035. URL: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf.
[27] Jack Choquette et al. "NVIDIA A100 Tensor Core GPU: Performance and Innovation". In: *IEEE Micro* 41.2 (2021), pp. 29–35. DOI: 10.1109/MM.2021.3061394.
[28] Patrick Lewis et al. *Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks*. 2021. arXiv: 2005.11401 [cs.CL].
[29] *LocalGPT*. https://github.com/PromtEngineer/localGPT.
[30] Harrison Chase. *LangChain*. Oct. 2022. URL: https://github.com/langchain-ai/langchain.
[31] Brian W Kernighan and Dennis M Ritchie. "The C programming language". In: (2002).
[32] Mark Dowd, John McDonald, and Justin Schuh. *The art of software security assessment: Identifying and preventing software vulnerabilities*. Pearson Education, 2006.
[33] Haiyan Zhao et al. "Explainability for Large Language Models: A Survey". In: *ACM Trans. Intell. Syst. Technol.* 15.2 (2024). ISSN: 2157-6904. DOI: 10.1145/3639372. URL: https://doi.org/10.1145/3639372.
[34] Julia El Zini and Mariette Awad. "On the Explainability of Natural Language Processing Deep Models". In: *ACM Comput. Surv.* 55.5 (2022). ISSN: 0360-0300. DOI: 10.1145/3529755. URL: https://doi.org/10.1145/3529755.
[35] Miles Turpin et al. *Language Models Don't Always Say What They Think: Unfaithful Explanations in Chain-of-Thought Prompting*. 2023. arXiv: 2305.04388 [cs.CL].
[36] Xuansheng Wu et al. *Usable XAI: 10 Strategies Towards Exploiting Explainability in the LLM Era*. 2024. arXiv: 2403.08946 [cs.LG].
[37] Chandan Singh et al. *Rethinking Interpretability in the Era of Large Language Models*. 2024. arXiv: 2402.01761 [cs.CL].

A Markdown Computer Code Language Identifiers

`sql, c++, javascript, cpp, python, csharp, c, json, less, bash, sh, java, text, json... , json..., diff, css, assembly, xml, perl, yaml, css, scss, html, makefile, js, csv, lua, kotlin, arduino, javascript, csharp, rust, shellscript, erb, vbnet, ‘json, go, plaintext, php, instructions:, ocl, shell, json ?, ‘json‘...`

B NIST SARD Test Cases

This section shows the exact source code for 10 of the NIST SARD test cases that are used in this study. The single vulnerable line of source code is highlighted as red text for each test case.

```
struct Token {
    const Token* nextArgument() const;
    const Token* next() const;
    int varId() const;
    void str() const;};
void f(const Token *first) {
    first = first->nextArgument();
    if (first)
        first = first->next();
    first->str();
}
```

Listing 1: NIST SARD Test case 500843, instance of CWE-476

```
#include <stdio.h>
#include <cgic.h>
#include <string.h>
#include <stdlib.h>

int cgiMain()
{
        cgiHeaderContentType("text/html");

        fprintf(cgiOut, "<html><head>\n");
        fprintf(cgiOut, "<title>Text: 1</title></head>\n");
        fprintf(cgiOut, "<body><h1>Text</h1>\n");

        char q[4][1024];
        unsigned int i = 0;
        for (; i < 4; ++i){
                char name[2];
                sprintf(name,"q%d",i);
                cgiFormString(name, q[i], sizeof(q[i]));
                if (strlen(q[i]))
                {
                        fprintf(cgiOut, "Value number %d = %s<br />", i, q[i
                            ]);
                }
        }

        fprintf(cgiOut, "</body></html>\n");
        return 0;
}
```

Listing 2: NIST SARD Test case 1792, instance of CWE-79. Note that some of the strings in this test case have been modified so that terms related to cross site scripting are removed.

```
main ()
{
char *foo;
int counter;
foo=malloc(sizeof(char)*10);
for (counter=0;counter!=14;counter++){
foo[counter]='a';
printf("%s\n",foo);
}
```

Listing 3: NIST SARD Test case 1779, instance of CWE-463.

```
#include <stdio.h>
#include <stdlib.h>

#define MAXSIZE         40

void test(char *str){
        char buf[MAXSIZE];
        snprintf(buf, sizeof buf, "/bin/echo %s", str);
        buf[MAXSIZE-1] = 0;
        system(buf);
}

int main(int argc, char **argv){
        char *userstr;
        if(argc > 1) {
                userstr = argv[1];
                test(userstr);
        }
        return 0;
}
```

Listing 4: NIST SARD Test case 1645, instance of CWE-20.

```
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
#include <string.h>

unsigned int getRand()
{
        unsigned int r;
        FILE *f;
        f = fopen("/dev/urandom", "rb");
        if(f == NULL)
        {
                fprintf(stderr, "Error opening file\n");
                exit(-1);
        }
        if(fread(&r, sizeof r, 1, f) != 1)
        {
                fprintf(stderr, "Error reading file\n");
                fclose(f);
                exit(-1);
        }
        if(fclose(f) != 0)
                fprintf(stderr, "Error closing file\n");

        return r;
}

unsigned plop() {
        return getRand() % 256 + 127;
}

int main(int argc, char *argv[])
{
        char buffer[256];
        memset(buffer, 0, sizeof(buffer));
        buffer[plop()] = '!';
        printf("%s\n", buffer);
        return 0;
}
```

Listing 5: NIST SARD Test case 149165, instance of CWE-121.

```
#include <openssl/evp.h>

main(){
EVP_CIPHER_CTX ctx;
char key[EVP_MAX_KEY_LENGTH];
char iv[EVP_MAX_IV_LENGTH];
int b=8;
RAND_bytes(key, b);
memset(iv,0,EVP_MAX_IV_LENGTH);
EVP_EncryptInit(&ctx,EVP_bf_cbc(), key,iv);
return;
}
```

Listing 6: NIST SARD Test case 2015, instance of CWE-329.

```
#include <cstdlib>
void f()
{
  char *p; p = (char *)malloc(10);
  p[10] = 7;
  free(p);
}
```

Listing 7: NIST SARD Test case 500757, instance of CWE-787

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <string.h>

static bool debug = false;

void promote_root() {
        if (debug) {

                printf ("# You are root now...\n");
        }
}

int main(int argc, char *argv[])
{
        if (argc > 1)
        {
                unsigned i;
                const unsigned nbArgs = argc;

                for (i = 1; i < nbArgs; ++i)
                {
                        if (!strncmp(argv[i],"-debug",6))
                        {
                                debug = true;
                                printf("Move to debugging mode\n");

                                if (strlen(argv[i]) >= 11 && !strncmp(argv[i
                                      ]+6,":root",5))
                                {
                                        promote_root();
                                }
                        }
                }
        }
        else
        {
                printf("No args...\n");
        }

        return 0;
}
```

Listing 8: NIST SARD Test case 149135, instance of CWE-489.

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

typedef struct str_t str;
struct str_t {
        union {
                int   a;
                char *b;
        } foo;
};

int main()
{
        str container;
        char *str1;
        container.foo.a = 0;

        if ((container.foo.b = (char *)malloc(256*sizeof(char))) != NULL)
        {
                strcpy(container.foo.b, "Falut!");
                container.foo.b[0] = 'S';
                printf("%s\n", container.foo.b);
                free(container.foo.b);
                str1 = malloc(65536);
                if (str1 == NULL)
                        {printf("Memory allocation problem"); return 1;}

                free(str1);

                if (container.foo.b)
                        {container.foo.b[0] = 'S'; printf("%s\n", container.
                              foo.b);}
        }
        return 0;
}
```

Listing 9: NIST SARD Test case 149203, instance of CWE-416.

```c
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

#define MAX_SIZE 255

typedef struct cont_t cont;
struct cont_t {
        char fmt[MAX_SIZE];
};

int main(int argc, char *argv[])
{
        if (argc > 1) {
                cont container = {.fmt=""};
                strncpy(container.fmt, argv[1],MAX_SIZE-1);
                container.fmt[MAX_SIZE - 1] = '\0';
                printf(container.fmt);
        }
        return 0;
}
```

Listing 10: NIST SARD Test case 149111, instance of CWE-134.