The FormAI Dataset: Generative AI in Software Security through the Lens of Formal Verification

Norbert Tihanyi
Technology Innovation Institute
Abu Dhabi, UAE

Tamas Bisztray
The University of Oslo
Oslo, Norway

Ridhi Jain
Technology Innovation Institute
Abu Dhabi, UAE

Mohamed Amine Ferrag
Technology Innovation Institute
Abu Dhabi, UAE

Lucas C. Cordeiro
University of Manchester
Manchester, UK

Vasileios Mavroeidis
The University of Oslo
Oslo, Norway

ABSTRACT

This paper presents the FormAI dataset, a large collection of 112, 000 AI-generated compilable and independent C programs with vulnerability classification. We introduce a dynamic zero-shot prompting technique constructed to spawn diverse programs utilizing Large Language Models (LLMs). The dataset is generated by GPT-3.5-turbo and comprises programs with varying levels of complexity. Some programs handle complicated tasks like network management, table games, or encryption, while others deal with simpler tasks like string manipulation. Every program is labeled with the vulnerabilities found within the source code, indicating the type, line number, and vulnerable function name. This is accomplished by employing a formal verification method using the Efficient SMT-based Bounded Model Checker (ESBMC), which uses model checking, abstract interpretation, constraint programming, and satisfiability modulo theories to reason over safety/security properties in programs. This approach definitively detects vulnerabilities and offers a formal model known as a counterexample, thus eliminating the possibility of generating false positive reports. We have associated the identified vulnerabilities with Common Weakness Enumeration (CWE) numbers. We make the source code available for the 112, 000 programs, accompanied by a separate file containing the vulnerabilities detected in each program, making the dataset ideal for training LLMs and machine learning algorithms. Our study unveiled that according to ESBMC, 51.24% of the programs generated by GPT-3.5 contained vulnerabilities, thereby presenting considerable risks to software safety and security.

CCS CONCEPTS

• **Security and privacy → Formal security models**; **Software security engineering**.

KEYWORDS

Dataset, Vulnerability Classification, Large Language Models, Formal Verification, Software Security, Artificial Intelligence.

PROMISE '23, December 8, 2023, San Francisco, CA, USA

ACM ISBN 979-8-4007-0375-1/23/12.
https://doi.org/10.1145/3617555.3617874

ACM Reference Format:
Norbert Tihanyi, Tamas Bisztray, Ridhi Jain, Mohamed Amine Ferrag, Lucas C. Cordeiro, and Vasileios Mavroeidis. 2023. The FormAI Dataset: Generative AI in Software Security through the Lens of Formal Verification. In *Proceedings of the 19th International Conference on Predictive Models and Data Analytics in Software Engineering (PROMISE '23), December 8, 2023, San Francisco, CA, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3617555.3617874

1 INTRODUCTION

The advent of Large Language Models (LLMs) is revolutionizing the field of computer science, heavily impacting software development and programming as developers and computer scientists enthusiastically use AI tools for code completion, generation, translation, and documentation [4, 29]. Research related to program synthesis using Generative Pre-trained Transformers (GPT) [7] is gaining significant traction, where initial studies indicate that the GPT models can generate syntactically correct yet vulnerable code [6]. A recent study conducted at Stanford University suggests that software engineers assisted by *OpenAI's codex-davinci-002 model* during development were at a higher risk of introducing security flaws into their code [27]. As the usage of AI-based tools for code generation continues to expand, understanding their potential to introduce software vulnerabilities becomes increasingly important. Considering that GPT models are trained on freely available data from the internet, which can include vulnerable code, AI tools can potentially recreate the same patterns that facilitated those vulnerabilities.

Our primary objective is to explore how proficiently LLMs can produce secure code for different coding objectives without requiring subsequent adjustments or human intervention. Additionally, we aim to uncover the most frequent vulnerabilities that LLMs tend to introduce in the code they generate, identifying common patterns in realistic examples to comprehend their behavior better. This brings forward the following research questions:

- **RQ1**: How likely is purely LLM-generated code to contain vulnerabilities on the first output when using simple zero-shot text-based prompts?
- **RQ2**: What are the most typical vulnerabilities LLMs introduce when generating code?

In particular, we explore these research questions in the context of GPT-3.5 generating C programs. GPT-3.5 is the most widely used LLM available to software developers with a free web interface. Moreover, C is one of the most popular programming languages for embedded systems, critical security systems, and Internet of Things (IoT) applications. For our purposes, simply showing through a

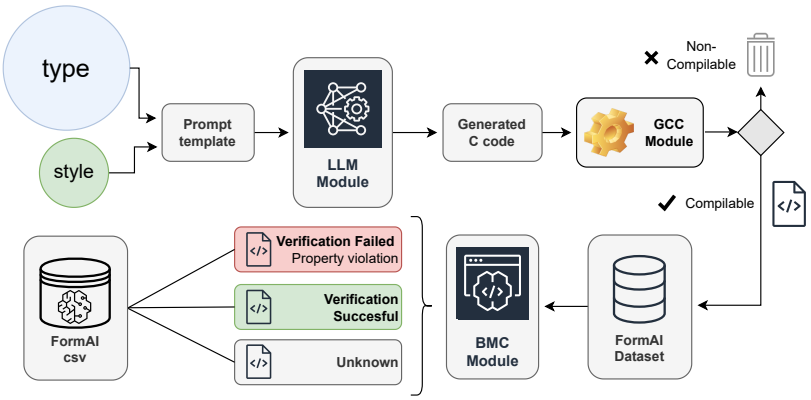

Figure 1: AI-driven Dataset Generation and Vulnerability Labeling with Program Classification by the BMC Module

handful of empirical examples that LLMs can produce vulnerable code is not gratifying and has been demonstrated before for various programming languages [6, 27, 34].

Two things are required to address the outlined research questions accurately. First, a large database containing a diverse set of C programs. Second, we need to gain insight into the variety and distribution of different vulnerabilities. At the same time, we must definitively determine whether a vulnerability is present in the code. If we label the code as vulnerable, it should not be a false positive. The latter is essential when creating datasets for machine learning purposes [28]. On that note, deep learning applications also need large datasets of vulnerable source code for training purposes [8].

Here, we developed a simple yet effective prompting method to obtain a diverse dataset, prodding the LLM to tackle a mixed bag of tasks. This resulted in 112, 000 C programs addressing various programming scenarios. Manually labeling the entire dataset is unfeasible for such a large corpus of data. Therefore, we use the Efficient SMT-based Bounded Model Checker (ESBMC) [13], which can formally falsify the existence of certain vulnerabilities. This state-of-the-art tool showcased exceptional performance in the SV-COMP 2023 [2] competition by efficiently solving many verification tasks within a limited timeframe [13]. Although it can only detect formally verifiable errors through symbolic execution, it does not produce false positives.

One limitation of this method is that it can only detect vulnerabilities within a predefined search depth bounded by the available computational capacity due to its resource-intensive nature. Suppose the complexity of the code does not allow the module to check all the nodes in the control-flow graph (CFG) [1] exhaustively under a reasonable time. In that case, we can only know the presence or absence of vulnerabilities within the predefined bound. If we do not find any vulnerabilities up to that depth, the code might still contain some. On the upside, which is why we use this method, we can definitively confirm the presence of the detected vulnerabilities up to a bound, as we can provide a *"counterexample"* as a formal model. Such databases can be useful for various research activities, especially in machine learning, which we remark on in our discussion.

Figure 1 illustrates the methodology employed in this paper. Initially, we provide instructions to GPT-3.5 to construct a C program for various tasks. This step will be elaborated thoroughly in Section 5. Next, each output is fed to the GNU C[1] compiler to check if the program is compilable. The compilable source code constitutes the FormAI dataset. These programs are input for the ESMBC module, which performs the labeling process. The labeled data is saved in a *.csv* file, which includes details such as the name of the vulnerable file, the specific line of code containing the vulnerability, the function name, and the type of vulnerability. To summarize, this paper holds the following original contributions:

- We present FormAI, the first AI-generated large-scale dataset consisting of 112, 000 independent compilable C programs that perform various computing tasks. Each of these programs is labeled based on the vulnerabilities identified by formal verification, namely, the ESBMC module;
- A comprehensive analysis on the identification and prevalence of vulnerabilities affecting the safety and security properties of C programs generated by GTP-3.5-turbo. The ESBMC module provides the detection and categorization of vulnerabilities. We connect the identified vulnerability classes with corresponding Common Weakness Enumeration (CWE) numbers.

The remaining sections are structured as follows: Section 2 discusses the inspiration for our work. Section 3 overviews the related literature. Section 4 presents a short introduction to formal verification and the ESBMC module. Section 5 outlines the approach we employed to create and categorize our dataset, where Section 6 provides an in-depth evaluation of our findings. Section 7 overviews limitations related to our work. Finally, Section 8 concludes the paper with an outlook on possible future research directions.

2 MOTIVATION

Throughout software development, it is paramount to guarantee the created programs' correctness, safety, and security. Functionally correct code produces the expected output for each given input. Safety aims to produce failure tolerant and fail-safe code, resistant against accidental or unexpected inputs that result in correct but

[1] https://gcc.gnu.org

undesired outputs, which may cause system failure or erroneous human decisions [10]. Finally, software security embodies robustness against external hazards and deliberate attacks. Our objective in this paper is to examine AI-generated source code's safety and security properties.

The term *"generated code"* signifies computer code created by an LLM, capable of using multiple forms of data as input. Textual prompts are segmented into individual units known as tokens. LLMs generate their response one token at a time, where a pre-defined token cap limits the output length. Due to this, as of today, LLMs cannot be used to spawn large applications on a single prompt. The main way developers utilize AI tools is by creating small programs or code snippets and include it into their projects. The Stanford study mentioned earlier [27] captured this important aspect. Some assignments given to students were, for example, creating:

- *two functions in Python where one encrypts and the other decrypts a given string using a given symmetric key;*
- *a function in Python that signs a given message using a given ECDSA signing key.*

We aim to prompt the LLM to produce code for tasks with similar complexity levels. Furthermore, for the ESBMC module, it is beneficial to have smaller independent programs. These allow the module to execute the verification process piece by piece, adhering to a set boundary, thus making the process manageable and more efficient. If the programs were heavily interdependent, accurately estimating the time required for the module to finish the verification process would be hardly feasible. The main area of interest in LLM-based code generation has been related to correctness. Datasets such as HumanEval provide programming challenges to assess the performance of models. For example, GPT-4 achieves a 67% success rate in solving tasks compared to 48.1% for GPT-3.5 [24]. Measuring correctness is not our goal with the FormAI dataset. For example, if the prompt says *"Create a board game using the C programming language in an artistic style"*, correctness would be difficult to verify, especially for a large dataset. The only requirements are the syntactical correctness of the program, and it must be compilable.

To restate our research objective, we aim to uncover the proportion and type of frequent coding errors in C source code generated by GPT-3.5 when prompted to perform simple tasks using natural language. The following real-life example demonstrates and underscores the necessity of this research question.

Imagine a situation where a programmer submits the following prompt to GPT-3.5: *"Provide a small C program that adds two numbers together"*. The code presented in Listing 1 is susceptible to vulnerabilities. It exhibits an integer overflow both during the addition of the variables num1 and num2, as well as in the `scanf()` functions that retrieve input numbers from the user. In 32-bit computing architectures, integers are commonly stored as 4 bytes (32 bits), which results in a maximum integer value of 2147483647, equivalent to $2^{31} - 1$. If one attempts to add 2147483647 + 1 using this small program, the result will be incorrect due to integer overflow. The incorrect result will be -2147483648 instead of the expected 2147483648. The addition exceeds the maximum representable value for a signed 32-bit integer $2^{31} - 1$, causing the integer to wrap around and become negative due to the two's complement representation.

C++ program generated by gpt-3.5-turbo

```c
#include <stdio.h>

int main() {
    int num1, num2, sum;
    printf("First number: ");
    scanf("%d", &num1);
    printf("Second number: ");
    scanf("%d", &num2);
    sum = num1 + num2;
    printf("The sum is: %d\n", sum);
    return 0;
}
```

Listing 1: Insecure code generated by gpt-3.5-turbo.

Even when GPT-3.5 is requested to write a secure version of this code –without specifying the vulnerability– it only attempts to verify against entering non-integer inputs by adding the following code snippet: `if (scanf("%d", &num1) != 1) {...}`. Clearly, the issue of integer overflow persists after sanitizing the input. When prompted to create a C program that adds two numbers, it appears that both GPT-3.5 and GPT-4 generate code with this insecure pattern. Both models implement input sanitization when seeking a more secure version, although they do not explicitly address the integer overflow issue. Applying the ESBMC module leads to the quick discovery of the integer overflow vulnerability as demonstrated in Listing 2.

ESBMC 7.2.0 module verification output (excerpt)

```
-------------------------------------------------------
Violated property:
  file test.c line 8 column 1 function main
  arithmetic overflow on add
  !overflow("+", num1, num2)
VERIFICATION FAILED
```

Listing 2: Property violation identified by ESBMC 7.2.0 in the source code presented in Listing 1.

In [6], the authors demonstrated that GPT-3.5 could efficiently fix errors if the output of the ESBMC module is provided. Given only general instruction as *"write secure code"*, or asked to find vulnerabilities, GPT-3.5 struggles to pinpoint the specific vulnerability accurately, let alone if multiple are present. While advanced models might perform better for certain vulnerabilities, this provides no guarantee that all coding mistakes will be found [26]. The main challenge is the initial detection without any prior hint or indicator. Doing this efficiently for a large corpus of C code while avoiding false positives and false negatives is still challenging for LLMs [26]. Based on this observation, we want to create an extensive and diverse dataset of properly labeled LLM-generated C programs. Such a dataset can reproduce coding errors often created by LLMs and

serve as a valuable resource and starting point in training LLMs for secure code generation.

3 RELATED WORK

This section overviews automated vulnerability detection and notable existing datasets containing vulnerable code samples for various training and benchmarking purposes.

3.1 ChatGPT in Software Engineering

In [23] Me et al. assessed the capabilities and limitations of ChatGPT for software engineering (SE), specifically in understanding code syntax and semantic structures like abstract syntax trees (AST), control flow graphs (CFG), and call graphs (CG). ChatGPT exhibits excellent syntax understanding, indicating its potential for static code analysis. They highlighted that the model also hallucinates when interpreting code semantics, creating non-existent facts. This implies a need for methods to verify ChatGPT's outputs to enhance its reliability. This study provides initial insights into why the codes generated by language models are syntactically correct but potentially vulnerable. Frameworks and techniques for turning prompts into executable code for Software Engineering are rapidly emerging, but the main focus is often functional correctness omitting important security aspects [36, 37, 39, 40]. In [22], Liu et al. questions the validity of existing code evaluation datasets, suggesting they inadequately assess the correctness of generated code.

In [19], the authors generated 21 small programs in five different languages: C, C++, Python, HTML, and Java. Combining manual verification with ChatGPT-based vulnerability detection, the study found that only 5 of the 21 generated programs were initially secure. A recent study by Microsoft [16] found that GPT models encounter difficulties when attempting to accurately solve arithmetic operations. This aligns with the findings we presented in the motivation Section. In a small study involving 50 students [32], the authors found that students using an AI coding assistant introduced vulnerabilities at the same rate as their unassisted counterparts. Still, notably, the experiment was limited by focusing only on a single programming scenario. Contrary to the previous study [32], in [25], Pearce et al. conclude that the control group, which utilized GitHub's Copilot, incorporated more vulnerabilities into their code. Instead of a single coding scenario like in [32], the authors expanded the study's comprehensiveness by choosing a diverse set of coding tasks pertinent to high-risk cybersecurity vulnerabilities, such as those featured in MITRE's "Top 25" Common Weakness Enumeration (CWE) list. The study highlights an important lesson: to accurately measure the role of AI tools in code generation or completion, it is essential to choose coding scenarios mirroring a diverse set of relevant real-world settings, thereby facilitating the occurrence of various vulnerabilities. This necessitates the creation of code bases replicating a wide range of settings, which is one of the primary goals the FormAI dataset strives to achieve. These studies indicate that AI tools, and in particular ChatGPT, as of today, can produce code containing vulnerabilities.

In a recent study, Shumailov et al. highlighted a phenomenon known as *"model collapse"* [33]. Their research demonstrated that integrating content generated by LLMs can lead to persistent flaws in subsequent models when using the generated data for training. This hints that training machine learning models only on purely AI-generated content is insufficient if one aims to prepare these models for detecting vulnerabilities in human-generated code. This is essentially due to using a dataset during the training phase, which is not diverse enough and misrepresents edge cases. We use our dynamic zero-shot prompting method to circumvent the highlighted issue to ensure diversity. Moreover, our research goal is to find and highlight what coding mistakes AI models can create, which requires a thorough investigation of AI-generated code. On the other hand, AI models were trained on human-generated content; thus, the vulnerabilities produced have roots in incorrect code created by humans. Yet, as discussed in the next section, existing datasets notoriously include synthetic data (different from AI-generated), which can be useful for benchmarking vulnerability scanners, but has questionable value for training purposes [8].

3.2 Existing databases for Vulnerable C code

We show how the FormAI dataset compares to seven widely studied datasets containing vulnerable code. The examined datasets are: Big-Vul [12], Draper [20, 30], SARD [3], Juliet [18], Devign [42], REVEAL [5], and DiverseVul[8]. Table 1 presents a comprehensive comparison of the datasets across various metrics. Some of this data is derived from review papers that evaluate these datasets [8, 17].

Big-Vul, Draper, Devign, REVEAL, and DiverseVul comprise vulnerable real-world functions from open-source applications. These five datasets do not include all dependencies of the samples; therefore, they are non-compilable. SARD and Juliet contain synthetic, compilable programs. In their general composition, the programs contain a vulnerable function, its equivalent patched function, and a main function calling these functions. All datasets indicate whether a code is vulnerable. The mentioned datasets use the following vulnerability labeling methodologies:

- `PATCH`: Functions before receiving GitHub commits for detected vulnerabilities are treated as vulnerable.
- `MAN`: Manual labeling
- `STAT`: Static analyzers
- `ML`: Machine learning-based techniques
- `BDV`: By design vulnerable

In the latter case, no vulnerability verification tool is used. Note that the size of the datasets can be misleading, as many of the datasets contain samples from other languages. For example, SARD contains C, C++, Java, PHP, and C#. Moreover, newly released sets often incorporate previous datasets or scrape the same GitHub repositories, making them redundant.

For example, Dreper contains C and C++ code from the SATE IV Juliet Test Suite, Debian Linux distribution, and public Git repositories. Since the open-source functions from Debian and GitHub were not labeled, the authors used a suite of static analysis tools: Clang, Cppcheck, and Flawfinder [30]. However, the paper does not mention if vulnerabilities were manually verified or if any confirmation has been performed to root out false positives. In [8], on top of creating DiverseVul, Chen et al. merged all datasets that were based on GitHub commits and removed duplicates, thus making the most comprehensive collection of GitHub commits containing vulnerable C and C++ code.

Table 1: Comparisons of various datasets based on their labeling classifications.

Dataset	Only C-code	Source	#Code Snippets	#Vuln. Snippets	Multiple Vulns/Snippet	Compiles/ Granularity	Vuln. Labelling	#Avg Line of Code	Labelling Method
Big-Vul	✘	Real-World	188,636	100%	✘	✘/Function	CVE/CVW	30	PATCH
Draper	✘	Synthetic+Real-World	1,274,366	5.62%	✔	✘/Function	CWE	29	STAT
SARD	✘	Synthetic+Real-World	100,883	100%	✘	✔/Program	CWE	114	BDV+STAT+MAN
Juliet	✘	Synthetic	106,075	100%	✘	✔/Program	CWE	125	BDV
Devign	✘	Real-World	27,544	46.05%	✘	✘/Function	CVE	112	ML
REVEAL	✘	Real-World	22,734	9.85%	✘	✘/Function	CVE	32	PATCH
DiverseVul	✘	Real-World	379,241	7.02%	✘	✘/Function	CWE	44	PATCH
FormAI	✔	**AI-generated**	**112,000**	51.24%	✔	✔/Program	CWE	79	ESBMC

Legend:
PATCH: GitHub Commits Patching a Vuln. **Man**: Manual Verification, **Stat**: Static Analyser, **ML**: Machine Learning Based, **BDV**: By design vulnerable

3.3 Vulnerability Scanning and Repair

Software verification is critical to ensure correctness, safety, and security. The primary techniques are manual verification, static analysis, and dynamic analysis, where a fourth emerging technique is machine learning-based detection [9, 11, 23, 35]. Manual verification techniques such as code review or manual testing rely on human effort and are not scalable. Static analysis can test the source code without running it, using techniques such as static symbolic execution, data flow analysis, control flow analysis, and style checking. On the other hand, dynamic analysis aims at observing software behavior while running the code. It involves fuzzing, automated testing, run-time verification, and profiling. The fourth technique is a promising field where LLMs can be useful in a wide range of tasks, such as code review and bug detection, vulnerability detection, test case generation, and documentation generation; however, as of today, each area has certain limitations. Research related to the application of verification tools in analyzing code specifically generated by LLMs remains rather limited. An earlier work from 2022 examined the ability of various LLMs to fix vulnerabilities, where the models showed promising results, especially when combined. Still, the authors noted that such tools are not ready to be used in a program repair framework, where further research is necessary to incorporate bug localization. They further highlighted challenges in the tool's ability to generate functionally correct code [26].

4 FORMAL VERIFICATION

This section presents the crucial foundational knowledge required to understand the technology employed in this research, specifically Bounded Model Checking (BMC). An intuitive question arises: is there a possibility that BMC could introduce false positives or false negatives to our dataset? The answer is no, and gaining insight into the underlying reasons is vital for our work. This aspect will be briefly outlined in this section.

Bounded Model Checking (BMC) is a technique used in formal verification to check the correctness of a system within a finite number of steps. It involves modeling the system as a finite state transition system and systematically exploring its state space up to a specified bound or depth. The latest BMC modules can handle various programming languages [14, 15, 31, 38, 41]. This technique first takes the program code, from which a control-flow graph (CFG) is created [1]. In CFG, each node signifies a deterministic or non-deterministic assignment or a conditional statement. Each edge represents a potential shift in the program's control position. Essentially, every node is a block representing a *"set of instructions with a singular entry and exit point"*. Edges indicate possible paths to other blocks to which the program's control location can transition. The CFG is first transformed into Static Single Assignment (SSA) and converted into a State Transition System (STS). This can be interpreted by a Satisfiability Modulo Theories (SMT) solver. This solver can determine if a set of variable assignments makes a given formula true, i.e., this formula is designed to be satisfiable if and only if there's a counterexample to the properties within a specified bound k. If there is no error state and the formula is unsatisfiable up to the bound k, there is no software vulnerability within that bound. If the solver reaches termination within a bound $\leq k$, we can definitively prove the absence of software errors.

To be more precise, let a given program $\mathcal{P}$ under verification be a finite state transition system, denoted by a triple $\mathcal{ST} = (S, R, I)$, where S represents the set of states, $R \subseteq S \times S$ represents the set of transitions and $(s_n, \cdots, s_m) \in I \subseteq S$ represents the set of initial states. In a state transition system, a state denoted as $s \in S$ consists of the program counter value, referred to as pc, and the values of all program variables. The initial state denoted as s_1, assigns the initial program location within the Control Flow Graph (CFG) to pc. Each transition $T = (s_i, s_{i+1}) \in R$ between two states, s_i and s_{i+1}, is identified with a logical formula $T(s_i, s_{i+1})$. This formula captures the constraints governing the values of the program counter and program variables relevant to the transition.

Within BMC, properties under verification are defined as follows: $\phi(s)$ represents a logical formula that encodes states satisfying a safety/security property. In contrast, $\psi(s)$ represents a logical formula that encodes states satisfying the completeness threshold, indicating states corresponding to program termination. $\psi(s)$, contains unwindings so that it does not exceed the maximum number of loop iterations in the program. It is worth noting that, in our notation, termination and error are mutually exclusive: $\phi(s) \wedge \psi(s)$ is by construction unsatisfiable. If $T(s_i, s_{i+1}) \vee \phi(s)$ is unsatisfiable, state s is considered a deadlock state. The bounded model checking problem, denoted by BMC_Φ is formulated by constructing a logical

formula, and the satisfiability of this formula determines whether $\mathcal{P}$ has a counterexample of length k or less. Specifically, the formula is satisfiable if and only if such a counterexample exists within the given length constraint, i.e.:

$$BMC_{\Phi}(k) = I(s_1) \wedge \bigwedge_{i=1}^{k-1} T(s_i, s_{i+1}) \wedge \bigvee_{i=1}^{k} \neg\phi(s_i). \quad (1)$$

In this context, I denotes the set of initial states of $\mathcal{ST}$, and $T(s_i, s_{i+1})$ represents the transition relation of $\mathcal{ST}$, between time steps i and $i+1$. Hence, the logical formula $I(s_1) \wedge \bigwedge_{i=1}^{k-1} T(s_i, s_{i+1})$ represents the executions of $\mathcal{ST}$ with a length of k and $BMC_{\Phi}(k)$ can be satisfied if and only if for some $i \leq k$ there exists a reachable state at time step i in which ϕ is violated. If $BMC_{\Phi}(k)$ is satisfiable, it implies that ϕ is violated, and an SMT solver provides a satisfying assignment from which we can extract the values of the program variables to construct a counterexample.

A counterexample, or trace, for a violated property ϕ, is defined as a finite sequence of states $s_1, \ldots, s_k$, where $s_1, \ldots, s_k \in S$ and $T(s_i, s_{i+1})$ holds for $0 \leq i < k$. If equation (1) is unsatisfiable, we can conclude that no error state is reachable within k steps or less. This valuable information leads us to conclude that no software vulnerability exists in the program within the specified bound of k. With this methodology, we aim to classify every generated C program as either vulnerable or not, within a given bound k. By searching for counterexamples within this bound, we can establish, based on mathematical proofs, whether a counterexample exists and whether our program $\mathcal{P}$ contains a security vulnerability. This approach allows us to identify security issues such as buffer overflows or access-bound violations. We note that if a program is categorized as vulnerable, this determination relies on counterexamples, effectively eliminating the chance of false positives. Conversely, in situations where no counterexample exists, we can confidently assert that the program is free from vulnerabilities up to the bound k, ensuring the absence of false negatives.

4.1 The ESBMC module

This work uses the Efficient SMT-based Context-Bounded Model Checker (ESBMC) [13] as our chosen BMC module. ESBMC is a mature, permissively licensed open-source context-bounded model checker for verifying single- and multithreaded C/C++, Kotlin, and Solidity programs. It can automatically verify both predefined safety properties and user-defined program assertions. The safety properties include out-of-bounds array access, illegal pointer dereferences (e.g., dereferencing null, performing an out-of-bounds dereference, double-free of malloced memory, misaligned memory access), integer overflows, undefined behavior on shift operations, floating-point for NaN, divide by zero, and memory leaks. In addition, ESBMC implements state-of-the-art incremental BMC and k-induction proof-rule algorithms based on SMT and Constraint Programming (CP) solvers.

5 THE FORMAI DATASET

The FormAI dataset consists of two main components: AI-generated C programs and their vulnerability labeling. In the data generation phase, we create 112,000 samples. In the classification phase, we utilize ESBMC to identify vulnerabilities in the samples. The exact methodology is thoroughly explained in this section to ensure the reproducibility of the dataset creation process.

5.1 Code generation

To generate the dataset of small C programs, we utilized the GPT-3.5-turbo model. While constructing the dataset, we opted for the OpenAI API over the web interface to automate the sample generation process. We employ GPT-3.5 to generate C code instead of GPT-4 as the latter can be up to 60 times more expensive than the former model. During the creation process, special attention was given to ensure the diversity of the FormAI dataset, which contains 112,000 compilable C samples. Requesting the model to generate a C program frequently yields similar outcomes, such as adding two numbers or performing basic string manipulation, which does not align with our objectives. Our focus lies in systematically generating a comprehensive and varied collection of small programs that emulates the code creation process undertaken by developers. Therefore, we need a methodology to circumvent the generation of redundant and repetitive C programs. To address this issue, we have developed a prompting method consisting of two distinct parts: dynamic and static parts. The static component remains consistent and unchanged, while the dynamic portion of the prompt undergoes continuous variation. An example of how a single prompt looks is shown under Listing 3.

Dynamic code generation prompt

Write a unique C **[Type]** example program in a **[Style]** style. Instructions: a. Minimum 50 lines. b. Be creative! c. Do not say I am sorry. Always come up with some code. d. Make sure the program compiles and runs without any errors. e. Please generate a code snippet that starts with "'c and ends with "'.

Listing 3: Dynamic code generation prompt.

The dynamic part of the prompt, highlighted as **[Type]** and **[Style]**, represent distinct categories within the prompt, each encompassing different elements. Each API call selects a type from a set of 200 elements for the Type category. This category contains topics such as Wi-Fi Signal Strength Analyzer, QR code reader, Image Steganography, Pixel Art Generator, Scientific Calculator Implementation, etc. In a similar fashion, during each query, a coding style is chosen from a set of 100 elements within the Style category. This helps minimize repetition, as specific coding styles such as "excited", "relaxed", or "mathematical" are combined with each Type category. By employing this method, we can generate $200 \times 100 = 20,000$ distinct combinations. As highlighted by lessons from [26, 32] we need a code base that emulates an array of settings, and tasks should be concise enough to ensure an LLM can produce outputs within its token constraints. The tasks in the Type category were selected with this goal in mind. More complex prompts like "Create a CRUD application using React for the front-end, Node.js with Express for the back-end, and MongoDB for the database", would need to be segmented into smaller tasks. A prompt where the

type is, for example "File handling" with a "thriller" Style, will be represented differently in the vector space when tokenized, compared to when the style is "happy". Although the Type-Style combination might not seem compatible in all cases, we observed, that asking the LLM to program in different styles, further increased the variability and the difference between responses for the same task in most cases. A high-temperature parameter that regulates the degree of randomness in the model's response, can further enhance diversity among the programs created. The concept of prompt creation can be seen in Figure 2.

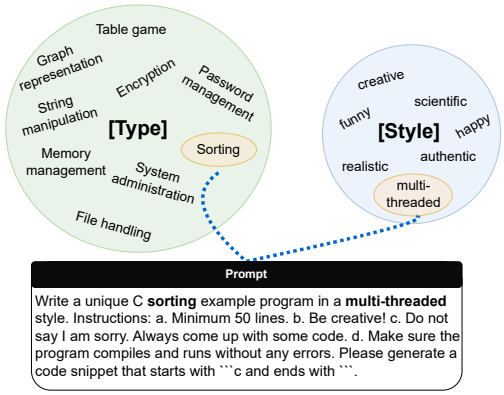

Figure 2: Dynamic prompt creation.

Decreasing the number of unsuccessful queries is an important factor from an efficiency perspective, as the price for gpt-3.5-turbo is 0.002 USD\1K token at the time of writing. Hence, refining the prompt to reduce the number of unsuccessful queries holds significant importance. To minimize the error within the generated code, we have established five instructions in each specific prompt:

a. `Minimum 50 lines:` This encourages the LLM to avoid the generation of overly simplistic code with only a few lines (which occasionally still happens);
b. `Be creative!:` The purpose of this instruction is to generate a more diverse dataset;
c. `Do not say I am sorry:` The objective of this instruction is to circumvent objections and responses such as "As an AI model, I cannot generate code", and similar statements.
d. `Make sure the program compiles:` This instruction encourages the model to include header files and create a complete and compilable program.
e. `Generate a code snippet that starts with "'c:` Enable easy extraction of the C code from the response.

Once a C code is generated, the GNU C compiler is employed to verify whether the corresponding code is compilable. During the experiment, over 90% of the generated code was compilable. The primary reason for having non-compilable code was due to the absence of necessary headers, such as `math.h`, `ctype.h`, or `stdlib.h`. During the code generation process, we ensure that the FormAI dataset exclusively consists of compilable code, while excluding any other code that does not meet this criterion. Code generation does not require high computational power, and for this task, we utilized a standard MacBook Pro 2017 with 16 GB of RAM. The generation of 112, 000 code samples was completed within 24 hours. As of the time of writing, the total cost for the creation process was approximately 200 USD.

5.2 Vulnerability classification

Following the code generation, we executed ESBMC on each individual file to classify them. Let us denote the set of all the generated C samples by Σ, such that $\Sigma = \{c_1, c_2, \ldots, c_{112,000}\}$, where each c_i represents an individual sample.

We can group all the samples into four distinct categories:

- $\mathcal{VS} \subseteq \Sigma$: the set of samples for which verification was successful (no vulnerabilities have been detected within the bound k);
- $\mathcal{VF} \subseteq \Sigma$: the set of samples for which the verification status failed (vulnerabilities detected by ESBMC based on counterexamples);
- $\mathcal{TO} \subseteq \Sigma$: the set of samples for which the verification process was not completed within the provided time frame (as a result, the status of these files remains uncertain);
- $\mathcal{ER} \subseteq \Sigma$: the set of samples for which the verification status resulted in an error.

These categories are mutually exclusive, meaning a single sample cannot belong to more than one category. For classification there are two important swithces in the ESBMC module. The `unwind` parameter controls the upper limit of loop iterations within the program, guaranteeing it remains under the program's predefined threshold. Another crucial parameter is `timeout`, which terminates the verification process after a set time interval. This is essential to prevent overly long classification times for exceptionally complex programs. Increasing the timeout parameter might improve results, but it can also reduce speed performance. Hence, it's recommended to keep both the timeout and unwind parameters low to achieve a balance between efficiency and precision.

The category denoted by *"TIMEOUT"* ($\mathcal{TO}$) includes all instances where the verification process could not be completed within the specified timeframe. It is worth noting that the $\mathcal{TO}$ category is significantly influenced by the runtime duration and the loop unwinding parameter used during ESBMC execution. For instance, if the loop unwinding parameter is set to a high number like 30, and the timeout is set to 1 second, most samples are expected to fall into this category. This occurs because only a subset of loops can be unwound up to the specified bound of 30 within the given 1-second timeframe. The category labeled as *"ERROR"* ($\mathcal{ER}$) encompasses all instances where the verification process faced errors or crashes in the core ESBMC module, GOTO converter, SMT solver, or the clang compiler module. These samples are omitted from the classification because they cannot be handled using the latest ESBMC module. The category of *"VERIFICATION SUCCESSFUL"* ($\mathcal{VS}$) indicates that using formal verification up to a certain search depth (bound), no counterexample was found in the program. However, it is important to note that increasing the verification time or the unwinding parameter can potentially lead to a change in the classification of a program as vulnerable.

"Verification failed" ($\mathcal{VF}$) represents the main focus of our interest. If a sample is classified as failed by ESBMC, it can be asserted

that there is a violation of properties in the program. Additionally, ESBMC provides a counterexample to demonstrate the specific property violation. We divided $\mathcal{VF}$ into 9 subcategories, where the first 8 are the most frequently occurring vulnerabilities, while *"Other"* encompasses the remaining vulnerabilities detected by ESBMC. Note, that in the CSV file that contains the vulnerability classification, the exact type of vulnerability is always indicated if applicable.

- $\mathcal{ARO} \subseteq \mathcal{VF}$: Arithmetic overflow
- $\mathcal{BOF} \subseteq \mathcal{VF}$: Buffer overflow on `scanf()/fscanf()`
- $\mathcal{ABV} \subseteq \mathcal{VF}$: Array bounds violated
- $\mathcal{DFN} \subseteq \mathcal{VF}$: Dereference failure : NULL pointer
- $\mathcal{DFF} \subseteq \mathcal{VF}$: Dereference failure : forgotten memory
- $\mathcal{DFI} \subseteq \mathcal{VF}$: Dereference failure : invalid pointer
- $\mathcal{DFA} \subseteq \mathcal{VF}$: Dereference failure : array bounds violated
- $\mathcal{DBZ} \subseteq \mathcal{VF}$: Division by zero
- $\mathcal{OTV} \subseteq \mathcal{VF}$: Other vulnerabilities

5.3 Experimental setup for classification

First, we must address why we chose ESBMC over other tools. For vulnerability classification within a 10-30 second time-limit per program, this verifier solves the highest amount of verification tasks according to SV-COMP 2023[2]. We ran the classification experiment using an AMD Ryzen Threadripper PRO 3995WX processor with 32 CPU cores. Beyond the `unwind` and `timeout` parameters, there are other crucial switches, such as the `k-induction` proof-rule algorithm, which is a verification technique for establishing the correctness of programs. It utilizes iterative deepening, consistently unwinding the program's execution for a set number of steps to produce verification results. Due to its incremental nature, it persistently finds the smallest property violations. The question that naturally arises is: which parameters are optimal to use, yielding the best return on computational investment?

We conducted experiments on 1,000 randomly selected samples from the FormAI, and performed various tests guiding us to determine the most optimal combination of parameters for this dataset. Table 2 presents the classification results of the 1,000 samples, showcasing the effects of different unwind (u) and time (t), coupled with the use of k-induction. Examining the detection results for parameter selection of $(u, t) = (1, 10)$, $(1, 30)$ or $(1, 100)$ without k-induction shows, that simply increasing the time threshold yields diminishing returns for the same unwind parameter.

Increasing either the timeout or unwind parameter may lead to the identification of more vulnerabilities, but at the cost of a significant increase in processing time. The same can be observed for enabling k-induction: incremental improvement, increased runtime. Considering the balance between the total run-time and the number of identified vulnerabilities, we chose the unwind parameters to be $(1, 30)$, without using the k-induction proof algorithm, for classifying the FormAI dataset. Particularly we have used the following command for each sample:

```
esbmc file.c --overflow --unwind 1 --memory-leak-check
--timeout 30 --multi-property --no-unwinding-assertions
```

[2] https://sv-comp.sosy-lab.org/2023/results/results-verified/quantilePlot-Overall.svg

Using these parameters on our 1000 sample set, 416 files were deemed non-vulnerable, while 519 files were determined to be vulnerable. Among these 519 files, a total of 2116 unique vulnerabilities were detected.

Table 2: Classification results for different parameters

(u,t)	VULN	k-ind	Running time (m:s)	$\mathcal{VS}$	$\mathcal{VF}$	$\mathcal{TO}$	$\mathcal{ER}$
(2,1000)	2438	✗	758:09	371	547	34	48
(3,1000)	2373	✗	1388:39	366	527	57	50
(2,100)	2339	✗	175:38	367	529	61	43
(2,100)	2258	✓	400:54	340	603	20	37
(1,100)	2201	✗	56:29	416	531	17	36
(1,30)	2158	✓	146:13	349	581	34	36
(3,100)	2120	✗	284:22	354	483	120	43
(1,30)	**2116**	✗	**30:57**	**416**	**519**	**30**	**35**
(1,10)	2069	✓	61:58	360	553	52	35
(1,10)	2038	✗	19:32	413	503	51	33
(3,30)	1962	✗	125:19	342	444	172	42
(1,1)	1557	✓	10:59	355	406	208	31
(1,1)	1535	✗	6:22	395	374	201	30

✓: Enabled, ✗: Disabled, (u, t) = unwind and timeout parameters

In the worst-case scenario, every program from FormAI would utilize its allocated time resulting in 30 seconds being dedicated to the verification of each sample. Using all 32 CPU threads, the entire verification process on our experimental setup would take approximately 1.2 days, calculated as $112,000 \times 30/3600/24/32$. The next section will cover the complete classification and provide a detailed distribution of the vulnerabilities in the FormAI dataset.

6 EVALUATION OF THE FORMAI DATASET

As per our methodology, we verified the compilability of every code piece by utilizing the `GNU C` compiler. Out of the complete dataset, 109,757 sample files ($\approx$ 98%) can be compiled without any dependencies solely using the simple command `gcc -lm -o <filename>`. The remaining 2% of samples pose greater complexity, including multithreaded applications, database management applications, and cryptographic applications such as AES encryption. As a result, these samples utilize ten distinct external libraries, including OpenSSL, sqlite3, pthread, and others. Upon successfully installing the following dependencies, all the files can be compiled without any issues: *libsqlite3-dev, libssl-dev, libportaudio2, portaudio19-dev, libpq-dev, libpcap-dev, libqrencode-dev, libsdl2-dev, freeglut3-dev, libcurl4-openssl-dev, libmysqlclient-dev.*

ESBMC is using the clang[3] compiler instead of *gcc* for the verification process. Among the 112,000 samples analyzed, a subset of 786 samples could not be successfully compiled using clang; therefore, these particular samples were excluded from the classification. Additionally, in a few cases, the ESBMC module crashed when attempting to handle code samples (category $\mathcal{ER}$), leading to the exclusion of those samples from the *".csv"* file containing the vulnerability labels. Despite this, we intentionally chose not to eliminate these samples from the dataset, as they hold value for further research.

We have examined over 8,848,765 lines of C code, with an average of 79 lines per sample. Programs with 47 lines are the most

[3] https://clang.llvm.org

common, with a total of 1405 samples. Among the programs in our dataset, only one surpasses a line count of 600. We employ a token-based keyword-counting mechanism to extract the cardinality of the 32 C keywords as shown in Figure 3. Tokens are the smallest elements of a programming language syntax and serve as building blocks for constructing statements, expressions, and other code constructs. The frequency is normalized to show the occurrence of keywords per million lines of code for each dataset. In this context, the frequency of if-statements, loops, and variables mimics the distribution in real-world projects. We attribute the similarity in patterns exhibited by FormAI to the fact that the training data of GPT models included actual projects from GitHub, which were written by human developers.

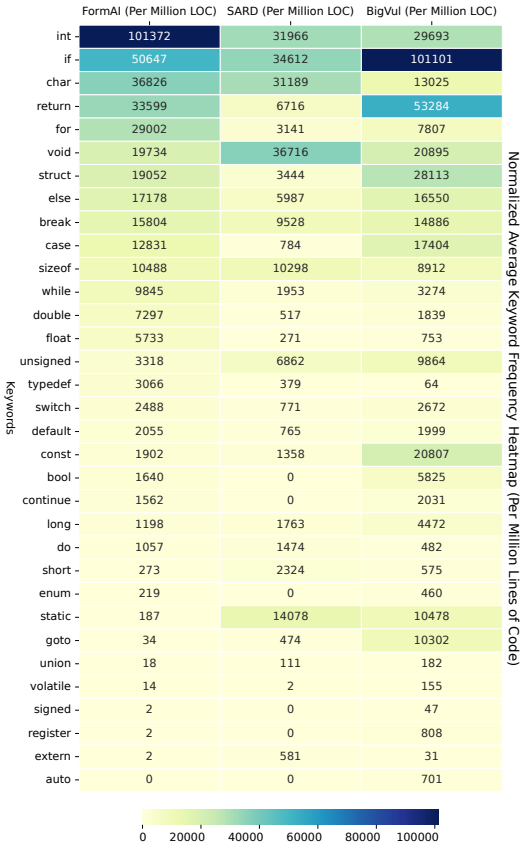

Figure 3: C Keyword frequency in FormAI, SARD, and BigVul

For the classification, in the *".csv"* files, we denoted the category $\mathcal{VS}$: as *"NOT VULNERABLE up to bound k"*. This aims to circumvent potential misinterpretations. We can only say that a program is devoid of vulnerabilities detectable by ESBMC if we configure the `–unwind` parameter to infinite, i.e. ($u = \infty$) and subsequently obtain successful verification. In total, from the 112,000 C programs, we performed the verification process on 106138 files. 5861 files were not classified due to crashes encountered by the ESBMC module. From the 106138 files, 57389 unique programs were found vulnerable. $\mathcal{VS}$ together with $\mathcal{TO}$ and $\mathcal{ER}$ constitutes 48749 C programs. The overall number of vulnerabilities detected by ESBMC is 197800.

6.1 CWE classification

Next, we connected the vulnerabilities to Common Weakness Enumeration (CWE) identifiers by manually associating the relevant CWEs with each vulnerability group. The multifaceted nature of software flaws often results in a single vulnerability being associated with multiple CWE identifiers. Table 3 shows the categorization of the most prevalent vulnerabilities and the associated 41 unique CWEs we identified across these categories. The *"Other vulnerabilities"* ($\mathcal{OTV}$) category includes Assertion failure, Same object violation, Operand of free must have zero pointer offset, function call: not enough arguments, and several types of dereference failure issues.

Table 3: The vulnerabilities identified by ESBMC, linked to Common Weakness Enumeration identifiers.

#Vulns	Vuln.	Associated CWE-numbers
88,049	$\mathcal{BOF}$	CWE-20, CWE-120, CWE-121, CWE-125, CWE-129, CWE-131, CWE-628, CWE-676, CWE-680, CWE-754, CWE-787
31,829	$\mathcal{DFN}$	CWE-391, CWE-476, CWE-690
24,702	$\mathcal{DFA}$	CWE-119, CWE-125, CWE-129, CWE-131, CWE-755, CWE-787
23,312	$\mathcal{ARO}$	CWE-190, CWE-191, CWE-754, CWE-680, CWE-681, CWE-682
11,088	$\mathcal{ABV}$	CWE-119, CWE-125, CWE-129, CWE-131, CWE-193, CWE-787, CWE-788
9823	$\mathcal{DFI}$	CWE-416, CWE-476, CWE-690, CWE-822, CWE-824, CWE-825
5810	$\mathcal{DFF}$	CWE-401, CWE-404, CWE-459
1620	$\mathcal{OTV}$	CWE-119, CWE-125, CWE-158, CWE-362, CWE-389, CWE-401, CWE-415, CWE-459, CWE-416, CWE-469, CWE-590, CWE-617, CWE-664, CWE-662, CWE-685, CWE-704, CWE-761, CWE-787, CWE-823, CWE-825, CWE-843
1567	$\mathcal{DBZ}$	CWE-369

Our research revealed 8 CWE identifiers that were included in MITRE's list of the TOP 25 CWEs for 2022. Listed in order, they are: 1. CWE-787, 4. CWE-20, 5. CWE-125, 7. CWE-416, 11. CWE-476, 13. CWE-190, 19. CWE-119, 22. CWE-362. The remaining CWEs in the top 25 list relate to web vulnerabilities like SQL injection, XSS, and authentication, which are not relevant to our C language samples. It is vital to emphasize that, in our situation, classifying the C programs based on CWE identifiers is not practical, contrary to what is done for other databases like Juliet. As shown in Table 1, most datasets contain only one vulnerability per sample. In the datasets ReVeal, BigVul, and Diversevul, a function is vulnerable if the vulnerability-fixing commit changes it, while in Juliet, a single vulnerability is introduced for each program. In FormAI, a single file often contains multiple vulnerabilities. Moreover, a single vulnerability can be associated with multiple CWEs. In most cases, multiple CWEs are required as prerequisites for a vulnerability to manifest. For example, in the case of *"CWE-120: Buffer Copy without Checking Size of Input (Classic Buffer Overflow)"*, other vulnerabilities can be facilitating the main issue. For example: *"CWE-676: Use of Potentially Dangerous Function"*, which might be the use of *scanf*, combined with *"CWE-20: Improper Input Validation"* can result in *"CWE-787: Out-of-bounds Write"*.

Labeling the vulnerable function name, line number, and vulnerability type identified by the ESBMC module provides granular information that can benefit machine learning algorithms. This level of detail can allow models to discern patterns and correlations with

higher precision, thereby improving vulnerability prediction and detection capabilities. As our programs contain several vulnerabilities and, in some cases, multiple instances of the same vulnerability, classifying each into a single CWE group, as done for Juliet, would be less optimal for training purposes. We also note that while other datasets like DiversVul and Juliet focus more on CWEs related to software security and vulnerabilities that could potentially be exploited, ESBMC also detects issues related to software safety.

6.2 Future Research Directions

The dataset containing all 112, 000 C program files, along with the two .csv files are published on GitHub[4] and IEEE Dataport[5]. The absence of false negatives and false positives makes the dataset suitable to benchmark the effectiveness of various static and dynamic analysis tools. The diverse structure of the C programs generated in the FormAI dataset made it excellent for an unexpected use case: fuzzing different applications. While running ESBMC on the dataset, we discovered and reported seven bugs in the module. After validating these issues, ESBMC developers managed to resolve them. These included errors in the goto-cc conversion and the creation of invalid SMT solver equations. Additionally, we identified bugs in the CBMC [21] and the Clang compiler, which failed to compile several programs with which GNU C had no issue. We promptly communicated these findings to the respective developers. The FormAI dataset aims to be a useful resource to train machine learning algorithms to possess the capabilities of the ESBMC module.

7 LIMITATIONS AND THREATS TO VALIDITY

While ESBMC is a robust tool for detecting many types of errors in C, as of today, it is not suited to detect design flaws, semantic errors, or performance issues. As such, more vulnerabilities might be present in the code besides the detected ones. It is crucial to highlight again that our method does not produce false negatives (as $\mathcal{VS}$ only holds up to bound k) or false positives (validated by counterexamples). A false negative result would mean that on a certain program, we claim that it is completely void of vulnerabilities, or there are no more vulnerabilities other than those detected. With more relaxed bounds or the k-induction switch enabled, as shown in Table 2, ESBMC might find slightly more vulnerabilities in a given program. The available computational capacity decides whether the verifier can finish the process under a given timeout. To find all errors detectable by ESBMC, unwind must be set to infinite, and ESMBC must complete the verification process. As we provided the original C programs and the instructions on how to run ESBMC, researchers who invest additional computational resources have the potential to enhance our findings.

In addition to the reported findings, we found several other instances of *"CWE-242: Use of Inherently Dangerous Function"*. Although ESBMC correctly reports several related functions as vulnerable, the reported line number of the vulnerability is often misleading. For instance, when the `gets()` function is invoked –which is declared in `io.c`– ESBMC marks a line number in `io.c` as the place of the vulnerability. This would be misleading for machine learning applications aiming to detect or fix vulnerabilities in the source code; thus, we exclude such reports from the two CSV files. We aim to update the CSV files as these issues are fixed in ESBMC.

[4] https://github.com/FormAI-Dataset

[5] https://ieee-dataport.org/documents/formai-dataset-large-collection-ai-generated-c-programs-and-their-vulnerability

8 CONCLUSIONS

This paper shows that GPT-3.5-turbo notoriously introduces vulnerabilities when generating C code. The broad range of programming scenarios in FormAI unveiled various coding strategies employed by GPT-3.5, which highlighted how it manages specific tasks sometimes utilizing questionable techniques resulting in a vulnerability. We created a zero-shot prompting technique to facilitate a wide range of programming scenarios and used GPT-3.5-turbo to generate C code. These programs constitute the FormAI dataset, containing 112, 000 independent compilable C programs. We used the ESBMC bounded model checker to produce formally verifiable labels for bugs and vulnerabilities. In our experiment, we allocated a verification time of 30 seconds to each program, with the unwinding parameter set to 1. In total 197800 vulnerabilities were detected by ESBMC. Some programs contain multiple different errors. The labeling is provided in a .csv file which includes: Filename, Vulnerability type, Function name, Line number, and Error type. In addition, we provide a separate .csv file containing the C code as a separate column. Finally, we connected the identified vulnerabilities to CWE identifiers. Based on these findings, we answer our research questions:

- **RQ1**: How likely is purely LLM-generated code to contain vulnerabilities on the first output when using simple zero-shot text-based prompts? **Answer**: At least 51.24% of the samples from the 112,000 C programs contain vulnerabilities. This indicates that GPT-3.5 often produces vulnerable code. Therefore, one should exercise caution when considering its output for real-world projects.
- **RQ2**: What are the most typical vulnerabilities LLMs introduce when generating code? **Answer**: For GPT-3.5: Arithmetic Overflow, Array Bounds Violation, Buffer Overflow, and various Dereference Failure issues were among the most common vulnerabilities. These vulnerabilities are pertinent to MITRE's Top 25 list of CWEs.

In addition, the FormAI dataset proved to be a valuable instrument for fuzzing different applications, as we have demonstrated by identifying multiple bugs in the ESBMC and CBMC modules and the Clang compiler.

ACKNOWLEDGEMENTS

Tamas Bisztray and Vasielios Mavroeidis have received funding from the EU Connecting Europe Facility (CEF) programme under Grant Agreement No. INEA/CEF/ICT/A2020/2373266 (JCOP project) and the Horizon Europe programme under Grant Agreement No. 101070586 (PHOENi2X project). The ESBMC development is currently funded by ARM, Intel, EPSRC grants EP/T026995/1, EP/V000497/1, EU H2020 ELEGANT 957286, and Soteria project awarded by the UK Research and Innovation for the Digital Security by Design (DSbD) Programme.

REFERENCES

[1] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. 2006. *Compilers: Principles, Techniques, And Tools* (2nd ed.). Addison-Wesley Longman Publishing Co., Inc., Boston, MA.

[2] Dirk Beyer. 2023. Competition on Software Verification and Witness Validation: SV-COMP 2023. In *Tools and Algorithms for the Construction and Analysis of*

Systems, Sriram Sankaranarayanan and Natasha Sharygina (Eds.). Springer Nature Switzerland, Cham, 495–522.

[3] Paul E. Black. 2018. A Software Assurance Reference Dataset: Thousands of Programs With Known Bugs. *Journal of Research of the National Institute of Standards and Technology* 123 (April 2018), 1–3. https://doi.org/10.6028/jres.123.005

[4] Nghi D. Q. Bui, Hung Le, Yue Wang, Junnan Li, Akhilesh Deepak Gotmare, and Steven C. H. Hoi. 2023. CodeTF: One-stop Transformer Library for State-of-the-art Code LLM. http://arxiv.org/abs/2306.00029

[5] Saikat Chakraborty, Rahul Krishna, Yangruibo Ding, and Baishakhi Ray. 2022. Deep Learning Based Vulnerability Detection: Are We There Yet? *IEEE Transactions on Software Engineering* 48, 9 (Sept. 2022), 3280–3296. https://doi.org/10.1109/TSE.2021.3087402

[6] Yiannis Charalambous, Norbert Tihanyi, Ridhi Jain, Youcheng Sun, Mohamed Amine Ferrag, and Lucas C. Cordeiro. 2023. A New Era in Software Security: Towards Self-Healing Software via Large Language Models and Formal Verification. https://doi.org/10.48550/arXiv.2305.14752

[7] Martin R. Chavez, Thomas S. Butler, Patricia Rekawek, Hye Heo, and Wendy L. Kinzler. 2023. Chat Generative Pre-trained Transformer: why we should embrace this technology. *American Journal of Obstetrics and Gynecology* 228, 6 (June 2023), 706–711. https://doi.org/10.1016/j.ajog.2023.03.010

[8] Yizheng Chen, Zhoujie Ding, Xinyun Chen, and David Wagner. 2023. DiverseVul: A New Vulnerable Source Code Dataset for Deep Learning Based Vulnerability Detection. http://arxiv.org/abs/2304.00409

[9] Lucas Cordeiro, Bernd Fischer, and Joao Marques-Silva. 2012. SMT-Based Bounded Model Checking for Embedded ANSI-C Software. *IEEE Transactions on Software Engineering* 38, 4 (July 2012), 957–974. https://doi.org/10.1109/TSE.2011.59

[10] J. Denning, Peter (Ed.). 1988. Program Verification: The Very Idea. *Commun. ACM* 31, 9 (1988), 1048–1063. https://doi.org/10.1145/48529.48530

[11] Vijay D'Silva, Daniel Kroening, and Georg Weissenbacher. 2008. A Survey of Automated Techniques for Formal Software Verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 27, 7 (July 2008), 1165–1178. https://doi.org/10.1109/TCAD.2008.923410

[12] Jiahao Fan, Yi Li, Shaohua Wang, and Tien N. Nguyen. 2020. A C/C++ Code Vulnerability Dataset with Code Changes and CVE Summaries. In *Proceedings of the 17th International Conference on Mining Software Repositories (MSR '20)*. Association for Computing Machinery, New York, NY, USA, 508–512. https://doi.org/10.1145/3379597.3387501

[13] Mikhail R Gadelha, Felipe R Monteiro, Jeremy Morse, Lucas C Cordeiro, Bernd Fischer, and Denis A Nicole. 2018. ESBMC 5.0: an industrial-strength C model checker. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. ACM, Montpellier, France, 888–891.

[14] Mikhail R. Gadelha, Felipe R. Monteiro, Jeremy Morse, Lucas C. Cordeiro, Bernd Fischer, and Denis A. Nicole. 2023. *ESBMC: 5.0: An Industrial-Strength Model Checker*. GitHub. https://github.com/esbmc/esbmc original-date: 2015-06-20T19:35:34Z.

[15] Mikhail Y. R. Gadelha, Enrico Steffinlongo, Lucas C. Cordeiro, Bernd Fischer, and Denis A. Nicole. 2019. SMT-based refutation of spurious bug reports in the clang static analyzer. In *Proceedings of the 41st International Conference on Software Engineering: Companion Proceedings, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*, Joanne M. Atlee, Tevfik Bultan, and Jon Whittle (Eds.). IEEE / ACM, Montreal, QC, Canada, 11–14. https://doi.org/10.1109/ICSE-Companion.2019.00026

[16] Shima Imani, Liang Du, and Harsh Shrivastava. 2023. Mathprompter: Mathematical reasoning using large language models. https://doi.org/10.48550/arXiv.2303.05398

[17] Ridhi Jain, Nicole Gervasoni, Mthandazo Ndhlovu, and Sanjay Rawat. 2023. A Code Centric Evaluation of C/C++ Vulnerability Datasets for Deep Learning Based Vulnerability Detection Techniques. In *Proceedings of the 16th Innovations in Software Engineering Conference*. ACM, Prayagraj, India, 1–10.

[18] Frederick E. Boland Jr and Paul E. Black. 2012. The Juliet 1.1 C/C++ and Java Test Suite. *NIST* 45, 10 (Oct. 2012), 88–90. https://www.nist.gov/publications/juliet-11-cc-and-java-test-suite Last Modified: 2021-10-12T11:10-04:00 Publisher: Frederick E. Boland Jr., Paul E. Black.

[19] Raphaël Khoury, Anderson R. Avila, Jacob Brunelle, and Baba Mamadou Camara. 2023. How Secure is Code Generated by ChatGPT? http://arxiv.org/abs/2304.09655

[20] Louis Kim and Rebecca Russell. 2018. Draper VDISC Dataset - Vulnerability Detection in Source Code. https://osf.io/d45bw/ Publisher: OSF.

[21] Daniel Kroening and Michael Tautschnig. 2014. CBMC–C Bounded Model Checker: (Competition Contribution). In *Tools and Algorithms for the Construction and Analysis of Systems: 20th International Conference, TACAS 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014. Proceedings 20*. Springer, Grenoble, France, 389–391.

[22] Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. 2023. Is Your Code Generated by ChatGPT Really Correct? Rigorous Evaluation of Large Language Models for Code Generation. https://doi.org/10.48550/arXiv.2305.01210

[23] Wei Ma, Shangqing Liu, Wenhan Wang, Qiang Hu, Ye Liu, Cen Zhang, Liming Nie, and Yang Liu. 2023. The Scope of ChatGPT in Software Engineering: A Thorough Investigation. https://doi.org/10.48550/arXiv.2305.12138

[24] OpenAI. 2023. GPT-4 Technical Report. http://arxiv.org/abs/2303.08774

[25] Hammond Pearce, Baleegh Ahmad, Benjamin Tan, Brendan Dolan-Gavitt, and Ramesh Karri. 2021. Asleep at the Keyboard? Assessing the Security of GitHub Copilot's Code Contributions. https://doi.org/10.48550/arXiv.2108.09293

[26] Hammond Pearce, Benjamin Tan, Baleegh Ahmad, Ramesh Karri, and Brendan Dolan-Gavitt. 2022. Examining Zero-Shot Vulnerability Repair with Large Language Models. http://arxiv.org/abs/2112.02125

[27] Neil Perry, Megha Srivastava, Deepak Kumar, and Dan Boneh. 2022. Do Users Write More Insecure Code with AI Assistants? http://arxiv.org/abs/2211.03622

[28] S. Picard, C. Chapdelaine, C. Cappi, L. Gardes, E. Jenn, B. Lefevre, and T. Soumarmon. 2020. Ensuring Dataset Quality for Machine Learning Certification. In *2020 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*. IEEE, Coimbra, Portugal, 275–282. https://doi.org/10.1109/ISSREW51248.2020.00085

[29] Steven I. Ross, Fernando Martinez, Stephanie Houde, Michael Muller, and Justin D. Weisz. 2023. The Programmer's Assistant: Conversational Interaction with a Large Language Model for Software Development. In *Proceedings of the 28th International Conference on Intelligent User Interfaces (IUI '23)*. Association for Computing Machinery, New York, NY, USA, 491–514. https://doi.org/10.1145/3581641.3584037

[30] Rebecca L. Russell, Louis Y. Kim, Lei H. Hamilton, Tomo Lazovich, Jacob A. Harer, Onur Ozdemir, Paul M. Ellingwood, and Marc W. McConley. 2018. Automated Vulnerability Detection in Source Code Using Deep Representation Learning. In *2018 17th IEEE International Conference on Machine Learning and Applications (ICMLA)*. IEEE, Orlando, FL, USA, 757–762. https://doi.org/10.1109/ICMLA.2018.00120

[31] Caitlin Sadowski and Jaeheon Yi. 2014. How developers use data race detection tools. In *Proceedings of the 5th Workshop on Evaluation and Usability of Programming Languages and Tools*. ACM, Portland, USA, 43–51.

[32] Gustavo Sandoval, Hammond Pearce, Teo Nys, Ramesh Karri, Siddharth Garg, and Brendan Dolan-Gavitt. 2023. Lost at C: A User Study on the Security Implications of Large Language Model Code Assistants. http://arxiv.org/abs/2208.09727

[33] Ilia Shumailov, Zakhar Shumaylov, Yiren Zhao, Yarin Gal, Nicolas Papernot, and Ross Anderson. 2023. The Curse of Recursion: Training on Generated Data Makes Models Forget. http://arxiv.org/abs/2305.17493

[34] Jovi Umawing. 2023. *ChatGPT writes insecure code.* Malwarebytes. https://www.malwarebytes.com/blog/news/2023/04/chatgpt-creates-not-so-secure-code-study-finds

[35] D.R. Wallace and R.U. Fujii. 1989. Software verification and validation: an overview. *IEEE Software* 6, 3 (May 1989), 10–17. https://doi.org/10.1109/52.28119

[36] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc Le, and Denny Zhou. 2023. Chain-of-Thought Prompting Elicits Reasoning in Large Language Models. https://doi.org/10.48550/arXiv.2201.11903

[37] Jules White, Quchen Fu, Sam Hays, Michael Sandborn, Carlos Olea, Henry Gilbert, Ashraf Elnashar, Jesse Spencer-Smith, and Douglas C. Schmidt. 2023. A Prompt Pattern Catalog to Enhance Prompt Engineering with ChatGPT. https://doi.org/10.48550/arXiv.2302.11382

[38] Martin White, Michele Tufano, Christopher Vendome, and Denys Poshyvanyk. 2016. Deep learning code fragments for code clone detection. In *Proceedings of the 31st IEEE/ACM international conference on automated software engineering*. Association for Computing Machinery, New York, USA, 87–98.

[39] Zhenchang Xing, Qing Huang, Yu Cheng, Liming Zhu, Qinghua Lu, and Xiwei Xu. 2023. Prompt Sapper: LLM-Empowered Software Engineering Infrastructure for AI-Native Services. https://doi.org/10.48550/arXiv.2306.02230

[40] Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Thomas L. Griffiths, Yuan Cao, and Karthik Narasimhan. 2023. Tree of Thoughts: Deliberate Problem Solving with Large Language Models. http://arxiv.org/abs/2305.10601

[41] Gang Zhao and Jeff Huang. 2018. Deepsim: deep learning code functional similarity. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, Lake Buena Vista, USA, 141–151.

[42] Yaqin Zhou, Shangqing Liu, Jingkai Siow, Xiaoning Du, and Yang Liu. 2019. *Devign: Effective Vulnerability Identification by Learning Comprehensive Program Semantics via Graph Neural Networks.* Curran Associates Inc., Red Hook, NY, USA, 10197–10207.

Received 2023-07-07; accepted 2023-07-28