

Short Paper: Cognitive AI-based Agents for the Creation and Verification of Software for Safety-Critical Systems

Ramesh Bharadwaj¹[0009-0008-3214-307X]

¹ US Naval Research Laboratory, 4555 Overlook Avenue SW, Washington DC, 20375 USA
ramesh.bharadwaj@nrl.navy.mil

Abstract. With the increasing reliance on collaborative and cloud-based systems, there is a drastic increase in attack surfaces on Safety-Critical Systems. Formal Methods are key for fielding and defending software systems at scale. However, the major deficiency of current approaches is their inability to model the cognitive processes of programmers and other software development professionals. They also do not scale and defy automation. What is needed is a combination of programmer behavior prediction by Cognitive Agents, with scalable code defect elimination based on Formal Methods, to develop an environment for the creation and verification of code for safety-critical systems.

Keywords: Formal Methods, Analysis, Verification, Safety-Critical Systems

1 Introduction

The term “Software Crisis” was first mentioned in the proceedings of the first NATO Software Engineering Conference in 1968 [1], and yet is still acknowledged as a crisis more than half a century later. In spite of spectacular advances in computing hardware, software remains brittle, expensive, and delivered with a number of latent flaws. This is particularly dangerous for Safety-Critical Systems, whose incorrect or deficient operation may lead to loss of lives and property. However, this “deploy and patch” attitude persists in industrial software development, even for software controlling weapons systems, with the potential for catastrophic accidents and spectacular failures. Recent advances in Machine Learning (ML), with their Data Centric, vs Human-Centric underpinnings, provided some hope of mitigating the software crisis. A. Karpathy, former Senior Director of Research of AI at Tesla, has gone so far as to call this process of code being created by computers, rather than written by humans, Software 2.0. Yet, Software 2.0 is no panacea. Systems built using this approach have unintended functions, and are sensitive to adversarial perturbations, leading to surprising failure modes when fielded.

2 Background

Generative Pre-trained Transformers [2, 3] have emerged as a general-purpose architecture for machine learning, beating the state of the art in many domains such as realistic image and natural language text creation, or automatic code generation.

Also known as Foundation Models, a term first popularized by the Stanford Institute for Human-Centered AI [4], They are trained in a self-supervised manner on a broad set of unlabeled data, and can be used for different tasks with minimal fine-tuning, using supervised transfer learning and a small corpus of human-annotated data sets. More recently, foundation models such as ChatGPT, which use self-supervised model pretraining and Reinforcement Learning with Human Feedback (RLHF) for their fine-tuning [5], have garnered wide attention in the popular press and have taken the world by storm.

More recently, in the past couple of years, models such as OpenAI’s CODEX [6], built along similar lines as Decoder-only Models (DoMs), have demonstrated the ability to write code from natural language “specifications,” providing developers the ability to create low/no-code platforms, in a process we call Software 3.0 [7]. This may indeed prove to be central to the future of software-intensive systems development and is poised to reshape the software industry. However, foundation models are also deeply flawed, leading many to speculate that software created by these tools could riddle the internet with even more bugs. This is of special concern to safety-critical systems because programmers are using these models, for example, for code creation for systems development for the US Department of Defense (DoD). Finally, these models are extremely expensive to build, given billions of weights that need optimization during their training.

3 Related Work

3.1 Automated Formal Methods

Formal Methods have never been applied to code created by Generative AI Models. No current method offers automated vulnerability detection and remediation, especially for yet-to-be encountered novel code defects. The DARPA Assured Micro-patching (AMP) program [8] comes close, but their approach is based on traditional software engineering. Also, the novel vulnerabilities AMP detected during its evaluation were shown to be previously known to AMP developers. The goal of the DARPA HACMS program [9] was to use formal methods for the elimination of exploitable bugs in an open source microkernel seL4 [10]. However, this process was extremely human-intensive, requiring more than 20 human-years of effort, and therefore not feasible or scalable for practical applications.

3.2 Generative AI Models

The process of training Generative Models falls under two distinct categories:

1. Starting with a generalized pre-trained transformer model, they are fine-tuned using code from open source repositories, creating models such as OpenAI’s CODEX.
2. A reward model (RM) is initially trained on a dataset labeled with comparisons by humans from two model outputs. Using the RM as a reward function, the pre-trained transformer’s policy (e.g., GPT-3) is fine-tuned to maximize this reward using a proximal policy optimization (PPO) algorithm.

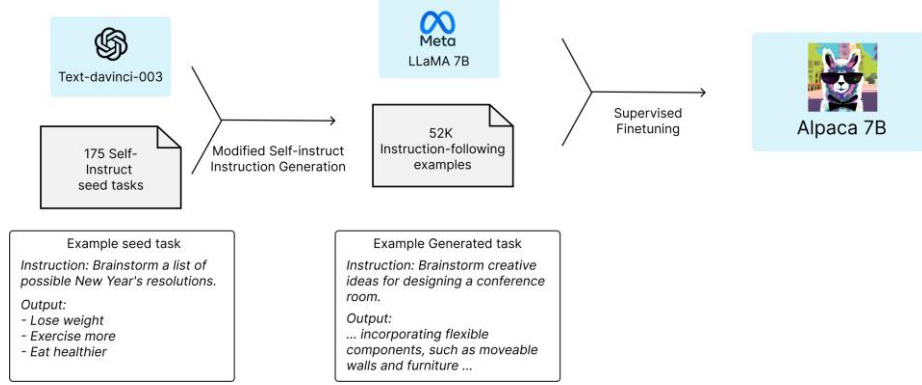


Fig. 1. Process used for training the Stanford Generative AI Model Alpaca

Figure 1 illustrates the method used to train the Stanford Generative AI Model Alpaca [11], which addresses the two important challenges for creating a high-quality instruction-following model without incurring the huge costs for training Industrial-strength models: (1) A strong pretrained language model and (2) High-quality instruction-following data. Stanford researchers addressed the first challenge by using the recently released LLaMA model from Meta. For the second challenge, they fine-tuned the model on 52K instruction-following demonstrations generated from OpenAI’s GPT-3.5, by starting with 175 human-written instruction-output pairs from the self-instruct seed set and subsequently prompting text-davinci-003 to generate more instructions using the seed set as in-context examples, resulting in 52K unique instructions and the corresponding outputs.

4 Problem Definition

Initial experimentation with commercial Generative AI Models indicates that models trained using the two approaches outlined above have comparable capabilities and generate almost identical code. For example, in response to the prompt “*Write a function in Python that opens a pdf document and returns the text*” both CODEX and ChatGPT responded with almost identical Python functions shown in Table 1.

However, due to the vintage of the data used to train the models, several run-time errors were detected by the Python interpreter when attempting to run the code: The deprecated methods `FFileReader`, `getNumPages`, `getPage` and `extractText` had to be replaced manually by their modern equivalents, by a tedious process of perusing the PyPDF2 documentation.

The resulting (hand crafted) function is shown in Table 2.

Table 1. Python function generated by ChatGPT

 Prompt: “Write a function in Python that opens a pdf document and returns the text”

```
import PyPDF2

def extract_text_from_pdf(filepath):
    with open(filepath, 'rb') as file:
        reader = PyPDF2.PdfFileReader(file)
        text = ''
        for i in range(reader.getNumPages()):
            page = reader.getPage(i)
            text += page.extractText()
        return text
```

Table 2. Hand-crafted run-time error free Python function

 Prompt: None

```
import PyPDF2

def extract_text_from_pdf(filepath):
    with open(filepath, 'rb') as file:
        reader = PyPDF2.PdfReader(file)
        text = ''
        for i in range(len(reader.pages)):
            page = reader.pages[i]
            text += page.extract_text()
        return text
```

A common error in code generated by Generative AI Models is the “off-by-one error;” such an example in code generated by CODEX is illustrated in Table 3.

Table 3. “Off-by-one” error in CODEX generated code

 Prompt: “Write a Python function for bubble sort with an unsorted array test case”

```
def bubble_sort(arr):
    for i in range(len(arr)):
        for j in range(len(arr)):
            if arr[j] > arr[j+1]:
                arr[j], arr[j+1] = arr[j+1], arr[j]
        return arr

arr = [4, 5, 7, 8, 1, 3]
print(bubble_sort(arr))
```

Table 4. Defect mitigation by ChatGPT

Prompt: “When I run the following python program:

```
def bubble_sort(arr):
    for i in range(len(arr)):
        for j in range(len(arr)):
            if arr[j] > arr[j+1]:
                arr[j], arr[j+1] = arr[j+1], arr[j]
    return arr
arr = [4, 5, 7, 8, 1, 3]
print(bubble_sort(arr))
```

I get the following error message from the python interpreter:

Traceback (most recent call last):

File "C:/Users/Ramesh/Documents/dump/BubbleSortOffbyOne.py", line 9, in <module>

```
    print(bubble_sort(arr))
```

File "C:/Users/Ramesh/Documents/dump/BubbleSortOffbyOne.py", line 4, in bubble_sort

```
    if arr[j] > arr[j+1]:
```

IndexError: list index out of range

Propose code that will fix the issue”

```
def bubble_sort(arr):
    for i in range(len(arr)):
        for j in range(len(arr)-1):
            if arr[j] > arr[j+1]:
                arr[j], arr[j+1] =
                    arr[j+1], arr[j]
    return arr

arr = [4, 5, 7, 8, 1, 3]
print(bubble_sort(arr))
```

This was remedied by harnessing ChatGPT to identify and fix the error in the CODEX generated code as illustrated in Table 4, using the prompt “Propose code that will fix the issue,” something even seasoned Python programmers were unable to identify and remedy by static analysis alone (i.e., without having to “debug” the program by running it).

5 Automated Software Factory

Our position is that researchers need to study the science behind automatic code generated by Generative AI Models, using Formal Methods tools to uncover flaws in the AI-generated code. Each such instance will serve as data for retraining the Generative AI Model to eliminate a class of vulnerabilities, enabling us to foster another research innovation: the use of Generative AI Models themselves for code rewriting, which will automate defect elimination, thereby tremendously scaling the process of defect-free software creation. The proposed approach therefore offers the intriguing possibility of moving towards the holy grail of automated software development.

5.1 Harnessing Formal Methods Tools

Static Source-code Analysis: As opposed to conventional formal methods, which attempt to prove that a system or software satisfies its requirements, i.e., application-specific properties, static code analysis tools instead attempt to establish relatively simple application-independent properties, such as lack of buffer overflow or divide by zero errors, on repositories with billions of lines of code [12]. However, their high cost to developers is tolerating false positives and dealing with false negatives. Additionally, they provide no hint as to how detected errors may be fixed, leading to further frustration among developers.

Matching and Rewriting Infrastructure using Abstract Syntax Trees (ASTs):

ASTs are data structures for representing and manipulating code within a compiler in a language-neutral manner. A parse-tree or a concrete syntax tree represents the structure of a phrase in the language in accordance with its grammar. ASTs are more compact than the corresponding parse trees for the same language construct. The process of AST manipulation within a language-independent compiler infrastructure is illustrated in Figure 2.

ASTs are amenable to analysis and rewriting using Application Program Interfaces available within an Integrated Development Environment (IDE) or a language processor such as the LLVM Compiler Infrastructure [13]. API calls for generating and printing the AST for “onePLUStwo = 1 + 2” is shown in Table 5.

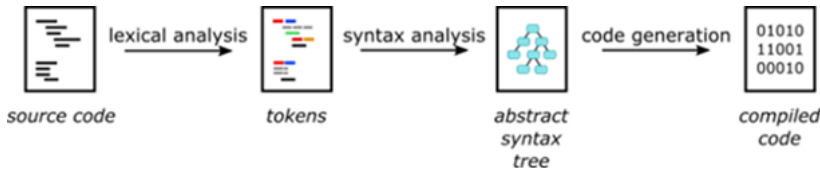


Fig. 2. The role of ASTs within a compiler

Table 5. API calls for creating and printing an AST using the LLVM infrastructure

```
import ast

code = "onePLUStwo = 1 + 2"
tree = ast.parse(code)
ast.dump(tree)
```

Table 6. AST printout generated by code in Table 5

```
"Module(
  Body [
    Assign(
      Targets [
        Name(id 'onePLUStwo', ctx Store())],
        value BinOp(
          left=Num(n=1),
          op=Add(),
          right=Num(n=2)))])"
```

5.2 Proposed Technical Approach

The process we envision for generation of correct-by-construction code is as follows: Our initial experiments will use OpenAI’s CODEX, which is pre-trained on a corpus of open-source code from Microsoft’s GitHub. After an initial exercise in understanding common errors found in the generated code, we shall create a catalog of the most frequent ones. An analysis tool based on the LLVM infrastructure will automate the detection of classes of errors in the catalog, and a scripting tool will be developed to rewrite the code to mitigate these errors. Subsequently, we will create and train a custom Generative AI Model on legacy code, which will include C++ and Python code for Machine Learning (ML) applications, including the corpus of defective-remediated code pairs generated in the previous step.

We address the two important challenges (identified by Stanford researchers when training the Alpaca model) for creating a high-quality code generation model using a similar approach by (1) utilizing a strong pretrained open-source language model and (2) by fine-tuning the model on defect mediation demonstrations generated from OpenAI’s GPT-4, by starting with the defective-remediated code pairs from the self-instruct seed set and subsequently prompting GPT-4 to generate more instances of such pairs using the seed set as in-context examples, resulting in a larger corpus of input-output pairs to fine-tune the generative model. We call this process “adversarial learning” which has never been attempted before for Generative AI Models.

6 Conclusion

Generative AI Models have the potential to mitigate the software crisis by creating an automatic code generation and code patching framework, thereby addressing most deficiencies of current bug detection and mitigation methods. If we succeed in this endeavor, this will prove to be a game-changer for software-intensive systems development and sustainment. Moreover, by constantly scanning code bases for vulnerabilities, and fixing them automatically, we also mitigate the cyber-defense problem for software systems. Lastly, an investigation into the use of Symbolic AI for defect and vulnerability detection, and Generative AI Models for their elimination, will provide software developers tools for safety-critical applications.

References

1. Randell, B.: Software Engineering: As it was in 1968. In: Proceedings 4th International Conference on Software Engineering, Munich, Germany, pp 1-10 (1969).
2. Vaswani, A. et al: Attention is all you need. In: Proceedings 31st Conference on Neural Information Processing Systems (2017).
3. Chen, M. et al: Evaluating large language models trained on code. arXiv:2107.03374v2 (2021).
4. Bommasani, R. et al: On the opportunities and risks of foundation models. arXiv:2108.07258 (2021).
5. Christiano, P. et al: Deep reinforcement learning from human preferences. arXiv:1706.03741 (2017).
6. Thompson, C.: It's Like GPT-3 but for Code - Fun, Fast, and Full of Flaws. Wired Magazine Backchannel (2022). <https://www.wired.com/story/openai-copilot-autocomplete-for-code/>, last accessed 2023/06/23.
7. Friedman, I.: Software 3.0 – the Era of Intelligent Software Development,” Medium.com (2022). https://medium.com/@itamar_f/software-3-0-the-era-of-intelligent-software-development-acd3cafe6cd7, last accessed 2023/06/23.
8. Bratus, S.: DARPA Assured Micropatching (AMP). <https://www.darpa.mil/program/assured-micropatching>, last accessed 2023/06/23.
9. Fischer, K. et al: The HACMS Program: Using Formal Methods to Eliminate Exploitable Bugs. Phil. Trans. R. Soc. A 375: 2015040 (2017).
10. VanVossen, R. et al: The seL4 Microkernel – A Robust, Resilient, and Open-Source Foundation for Ground Vehicle Electronics Architecture. NDIA Ground Vehicle Systems Engineering and Technology Symposium (2019).
11. Taori, R. et al: Alpaca: A Strong, Replicable Instruction-Following Model. <https://crfm.stanford.edu/2023/03/13/alpaca.html>, last accessed 2023/06/23.
12. Sadowski, C. et al: Lessons from Building Static Analysis Tools at Google. In: CACM, 61(4) pp 59-66 (2018).
13. The LLVM Project: The LLVM Compiler Infrastructure. <https://llvm.org/> (2007).