Combining Foundation Models and Symbolic AI for Automated Detection and Mitigation of Code Vulnerabilities

Ramesh Bharadwaj,[1] Ilya Parker[2]
US Naval Research Laboratory[1]
ramesh.bharadwaj@nrl.navy.mil[1]
Arcfield[2]
ilya.parker@arcfield.com[2]

Abstract

With the increasing reliance on collaborative and cloud-based systems, there is a drastic increase in attack surfaces and code vulnerabilities. Automation is key for fielding and defending software systems at scale. Researchers in Symbolic AI have had considerable success in finding flaws in human-created code. Also, run-time testing methods such as fuzzing do uncover numerous bugs. However, the major deficiency of both approaches is the inability of the methods to fix the discovered errors. They also do not scale and defy automation. Static analysis methods also suffer from the false positive problem – an overwhelming number of reported flaws are not real bugs. This brings up an interesting conundrum: Symbolic approaches actually have a detrimental impact on programmer productivity, and therefore do not necessarily contribute to improved code quality. What is needed is a combination of automation of code generation using large language models (LLMs), with scalable defect elimination methods using symbolic AI, to create an environment for the automated generation of defect-free code.

Background

The term "Software Crisis" was first mentioned in the proceedings of the first NATO Software Engineering Conference in 1968 (Randell, B. 1969), 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 Cyber-Physical Systems (CPSs), 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 underpinning, provide some hope of mitigating this crisis. A. Karpathy, former Senior Director of Research of AI at Tesla, has gone so far as to call this process of code being generated 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.

Introduction

Transformer Models have emerged as a general-purpose architecture for machine learning, beating the state of the art in many domains such as generative models for realistic image and natural language text creation, including automatic code generation. Initially proposed by Vaswani, A. et al. 2017, they perform sequential transformations, e.g., from natural language queries as inputs to generative answers as outputs. They may be broadly categorized as: (1) Encoder-only Models (EoMs), for tasks such as language understanding (2) Decoder-only Models (DoMs), for tasks such as text generation, and (3) Encoder-Decoder Models (EDMs), for tasks such as language translation and summarization, or for automatic code generation (Chen, M. et al. 2021).

Transformer models are also known as foundation models, a term first popularized by the Stanford Institute for Human-Centered Artificial Intelligence (Bommasani, R. et al. 2021). These models 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, transformer models such as ChatGPT, which use self-supervised model pretraining and Reinforcement Learning with Human Feedback (RLHF) for their fine-tuning (Christiano, P. et al. 2017), have garnered wide attention in the popular press and have taken the world by storm.

Early examples of foundation models, such as BERT and T5 from Google, or GPT1-3 and DALL-E (2) from OpenAI, have shown what is possible in image and natural language processing. By processing a short prompt, these models can generate an entire essay, or a complex image, even when the model was not trained on how to interpret the prompt, nor provided details of how to generate an image or sentence in a specific manner.

More recently, in the past couple of years, models such as OpenAI's CODEX (Thompson, C. 2022), 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 (Friedman, I. 2022). This may indeed prove to be central to the future of software-intensive systems development and is poised to reshape the software industry. However, transformer 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 soon programmers will use these models for their development. Finally, these models are extremely expensive to build, given billions of weights that need optimization during their training.

Related Work

Formal Methods have never been applied to code generated by Foundation Models. No current approach offers automated vulnerability detection and remediation, especially for yet-to-be encountered novel defects. The DARPA Assured Micropatching (AMP) program (Bratus, H. 2022) 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 HACMS program (Fischer, K. et al. 2017) was to use formal methods for the elimination of exploitable bugs in an open source microkernel (seL4) (VanVossen, R. et al. 2019). However, this process was extremely human-intensive, requiring more than 20 human-years of effort, and therefore not feasible or scalable for practical applications.

Problem Definition

Transformer Models, also known as Large Language Models currently fall under two distinct categories: 1. Starting with a generalized pre-trained transformer model, they are fine-tuned using code from open source repositories, in models such as 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. Nevertheless, our initial experiments indicate that the two models generate almost identical code and have comparable capabilities. 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 an almost identical Python function show in Listing 1.

Listing 1: Python function generated by ChatGPT

```
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
```

Due to the vintage of the data used to train CODEX and ChatGPT, a number of run-time errors were detected by the Python interpreter in the generated code:

1. The following methods in PyPDF2 are deprecated: `PDFFileReader`, `getNumPages`, `getPage` and `extractText`. These had to be replaced manually by their modern equivalents, after a tedious process of perusing the PyPDF2 documentation.
2. A call to extract_text_from_pdf() had to be manually inserted in order to test the function.

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

Listing 2: Hand-crafted run-time error free version

```
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

text = extract_text_from_pdf('test.pdf')
print(text)
```

Listing 3: "Off-by-one" error in CODEX created code

```
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))
```

A common error in code generated by Transformer Models is the "off-by-one error;" an example of such an error in code generated by CODEX is illustrated in Listing 3.

Listing 4: Defect-free code generated by ChatGPT

```
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 Listing 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).

Automated Software Factory

Our position is that researchers need to study the science behind automatic code generated by Foundation Models, using symbolic-AI based tools to uncover flaws in the AI-generated code. Each such instance will serve as data for retraining the Foundation Model to eliminate this class of latent vulnerabilities. This will enable us as a community to foster another research innovation: the use of Foundation 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.

Proposed Technical Approach

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 (Sadowski, C. et al. 2018). 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 is illustrated in Figure 1.

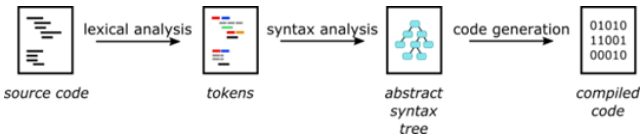

Figure 1: The role of ASTs within a compiler

ASTs are amenable to analysis and rewriting using Application Program Interfaces (APIs) available within an Integrated Development Environment (IDE) or a language processor such as the LLVM Compiler Infrastructure (The LLVM Project, 2007). An example sequence of API calls for generating and printing the AST for "`onePLUStwo = 1 + 2`" is shown in Listing 5.

Listing 5: **API calls for creating and printing AST**

```
import ast

code = "onePLUStwo = 1 + 2";
tree = ast.parse(code);
ast.dump(tree, indent 4);
```

Listing 6: **AST printout generated by code in Listing 5**

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

code = "onePLUStwo = 1 + 2";
```

The generated AST printout is illustrated in Listing 6.

Finally, the AST for the expression

"7 + 3 * (10 / (12 / (3+1) - 1))"

rendered by the program DOT is illustrated in Figure 2.

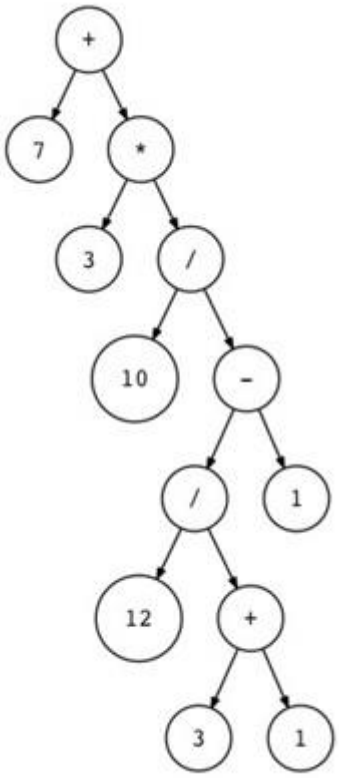

Figure 2: DOT rendered Abstract Syntax Tree

The proposed approach may be summarized as follows:
1. Understand and catalog errors introduced by extant Foundation Models such as CODEX.
2. Design and develop Symbolic AI based analysis and mitigation tool using the following techniques:
a. Generic (language independent) representation of code as Abstract Syntax Trees
b. Analysis methods for ASTs for detection of critical errors
c. Error mitigation by rewriting ASTs to fix critical errors
d. Code generation from ASTs in many popular languages
3. Experiment with software relevant error detection, analysis, and remediation, i.e., repeat steps 1 and 2 above with custom Foundation Model trained on legacy code.
4. Investigate Foundation Models' ability to find and fix critical errors in codebases (human generated as well as automated). Measure effectiveness of error-freedom by scrutinizing generated code with hand-crafted Symbolic AI-based tools, and evaluate results.

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. The "specification" of a code fragment or function to be generated will be provided as input, and the model's output will be analyzed using conventional static analysis tools – with their plethora of false positives and false negatives. After a painstaking exercise in understanding common errors found in the generated code, we shall create a catalog of the most frequent and egregious ones. A prototype tool to automate the analysis of code to detect classes of errors in the catalog, and a scripting tool to rewrite the code to mitigate these errors, will afford the possibility of writing out the corrected version of the generated code. This custom environment will find and fix errors in the code using the Application Program Interface (API) for AST manipulation and rewriting. This environment will be fine-tuned to look for specific classes of errors in the code. Subsequently, as our scientific understanding of code generation by Foundation Models progresses, we will create and train a custom Foundation Model on legacy code, which will include C++ and Python code for Machine Learning (ML) applications. Finally, we will experiment with the ability of CODEX and our custom Foundation Model to find and fix coding errors seamlessly and automatically. Their performance will be graded based on the number of latent errors detected by running the hand-crafted scalable Symbolic AI-based code analysis tool on the "correct-by-construction" code generated by our Foundation Model Factory. The innovation of the proposed research is to assess the ability of foundation models to find and fix common classes of errors and comparing their performance with hand-crafted Symbolic AI-based tool.

Conclusion

Foundation Models have the potential to mitigate the software crisis by creating an automatic code generation/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 Foundation Models for their elimination, will provide software developers tools based on Generalist System Theory for safety-critical applications.

Ethical Statement

The ability of LLMs to generate code may help individuals and organizations develop software more effectively and efficiently, saving time and resources, makes this method of code creation likely even if discouraged by organizations such as the FAA or the DoD. In this paper we have considered strategies to mitigate poor code quality and improve safety. However, code generated by LLMs also has the potential to raise concerns related to intellectual property, bias, accountability, and job displacement, among other issues. Each of these issues is worthy of study but are outside the scope of this paper. Their omittance is in no way intended to imply tacit agreement or approval, they are simply outside the scope of consideration herein. To mitigate these concerns, it is important for users to be aware of the limitations of LLMs and their responsible and ethical usage for automated code generation, especially in safety-critical systems. However, the precise meaning remains unclear; please consider this paper as a call for action for other kinds of inquiry into aspects of ethical AI beyond safety and robustness concerns. We want, as a community, to consider all the potential ethical implications of using LLMs and to help all parties take steps to ensure that benefits of AI technologies are maximized while minimizing their potential negative impacts.

Acknowledgments

Mr. Joseph Mathews deserves great credit for his unrelenting passion for improving clarity of exposition in this very important research area. Dr. David Aha, Branch Head of the Navy Center for Applied Research in Artificial Intelligence, was instrumental in attaining scientific rigor with insightful comments and constant encouragement. Dr. Peter Klein, Assistant Superintendent of NRL Information Technology Division, was instrumental in getting this research funded.

References

Randell, B. 1969. Software Engineering: As it was in 1968. In *Proc. 4th International Conference on Software Engineering*, Munich, Germany, pp 1-10.

Vaswani, A. et al. 2017. Attention is All You Need. *Proc. 31st Conference on Neural Information Processing Systems (NIPS 2017).*

Chen, M. et al. 2021. Evaluating large language models trained on code. arXiv:2107.03374v2.

Bommasani, R. et al. 2021. On the opportunities and risks of foundation models. arXiv:2108.07258.

Christiano, P. et al. 2017. Deep reinforcement learning from human preferences. arXiv:1706.03741.

Thompson, C. 2022. It's Like GPT-3 but for Code—Fun, Fast, and Full of Flaws. *Wired Magazine Backchannel*. https://www.wired.com/story/openai-copilot-autocomplete-for-code/.

Friedman, I. 2022. Software 3.0 – the Era of Intelligent Software Development. https://medium.com/@itamar_f/software-3-0-the-era-of-intelligent-software-development-acd3cafe6cd7.

Bratus, S. 2022. DARPA Assured Micropatching (AMP). https://www.darpa.mil/program/assured-micropatching.

Fischer, K. et al. 2017. The HACMS Program: Using Formal Methods to Eliminate Exploitable Bugs. *Phil. Trans. R. Soc. A 375: 2015040*.

VanVossen, R. et al. 2019. The seL4 Microkernel – A Robust, Resilient, and Open-Source Foundation for Ground Vehicle Electronics Architecture. *2019 NDIA Ground Vehicle Systems Engineering and Technology Symposium*.

Sadowski, C. et al. 2018. Lessons from Building Static Analysis Tools at Google. *CACM, 61(4) pp 59-66.*

The LLVM Project, 2007. The LLVM Compiler Infrastructure. https://llvm.org/.