UNIVERSITY OF CALIFORNIA,
IRVINE

Large Language Models for Programming Industrial Control Systems and Mitigating Real-World Software Vulnerabilities

THESIS

submitted in partial satisfaction of the requirements
for the degree of

MASTER OF SCIENCE

in Electrical and Computer Engineering

by

Rahul Dharmaji

Thesis Committee:
Professor Mohammad Al Faruque, Chair
Assistant Professor Sitao Huang
Assistant Professor Zhou Li

2024

TABLE OF CONTENTS

Page

LIST OF FIGURES

Page

LIST OF TABLES

Page

ACKNOWLEDGMENTS

I thank Professor Mohammad Al Faruque for providing significant guidance, insight, and mentorship throughout my time at UC Irvine. I also thank Assistant Professor Sitao Huang and Assistant Professor Zhou Li for serving on my Thesis Committee.

I thank my colleagues in the Advanced Integrated Cyber-Physical Systems (AICPS) Laboratory for their consistent and challenging critiques of my work, as well as for our innumerable hours of fruitful discussion and brainstorming, much of which is reflected in this manuscript.

Finally, I thank the close confidants who provided uncountable hours of support and companionship over the last 12 months. Your assistance greatly helped me adjust to this new environment! (especially 🐱 #3 :))

Portions of Chapter 1 and 3 are reprints of material as it appears in "LLM4CVE: Enabling Iterative Automated Vulnerability Repair with Large Language Models," with copyright retained by the authors. I possess the requisite rights and permissions to reprint this work up to and including in its entirety.

Portions of Chapter 2 are reprints of material as it appears in "LLM4PLC: Harnessing Large Language Models for Verifiable Programming of PLCs in Industrial Control Systems," in the 46th ACM/IEEE International Conference on Software Engineering, with copyright retained by the authors. I possess the requisite rights and permissions to reprint this work up to and including in its entirety.

Portions of Chapter 2 contain research that was supported by Siemens AG. Any opinions, findings, conclusions, or recommendations expressed in this chapter are those of the authors and do not necessarily reflect the views of the funding body.

ABSTRACT OF THE THESIS

Large Language Models for Programming Industrial Control Systems
and Mitigating Real-World Software Vulnerabilities

By

Rahul Dharmaji

Master of Science in Electrical and Computer Engineering

University of California, Irvine, 2024

Professor Mohammad Al Faruque, Chair

This manuscript is comprised of two sections – automated code generation for Programmable Logic Controllers and vulnerability repair for Common Vulnerabilities & Exposures (CVEs) with Large Language Models (LLMs).

The application of LLMs to Industrial Control Systems (ICS) is a relatively unexplored area. State-of-the-art LLMs such as GPT-4 and Code Llama fail to produce valid programs for ICS operated by Programmable Logic Controllers (PLCs). As a result, there is abundant potential to incorporate the use of Large Language Models into the PLC programming process to achieve end-to-end automation of common ICS tasks. We propose LLM4PLC, a user-guided iterative pipeline leveraging user feedback and external verification tools – including grammar checkers, compilers, SMV verifiers – as well as Parameter-Efficient Fine-Tuning and Prompt Engineering, to guide the LLM's generation. We run a complete test suite on GPT-3.5, GPT-4, Code Llama-7B, a fine-tuned Code Llama-7B model, Code Llama-34B, and a fine-tuned Code Llama-34B model. Ultimately, we demonstrate that the LLM4PLC pipeline improves the generation success rate from 47% to 72%, and the Survey-of-Experts code quality from 2.25/10 to 7.75/10.

Software vulnerabilities continue to be ubiquitous, even in the era of AI-powered code assistants, advanced static analysis tools, and the adoption of extensive testing frameworks. It has become apparent that we must not simply prevent these bugs, but also eliminate them in a quick, efficient manner. Yet, human code intervention is slow, costly, and can often lead to further security vulnerabilities, especially in legacy codebases. The advent of highly advanced Large Language Models (LLM) has opened up the possibility for many software defects to be patched automatically. We propose LLM4CVE – an LLM-based iterative pipeline that robustly fixes vulnerable functions with high accuracy. We examine our pipeline with State-of-the-Art LLMs, such as GPT-3.5, GPT-4o, Llama 3 8B, and Llama 3 70B, along with fine-tuned variants of selected models. We achieve an increase in ground-truth code similarity of 20% with Llama 3 80B.

Chapter 1

Introduction

In this chapter, a brief overview of Large Language Models, Industrial Control Systems, Programmable Logic Controllers, and Automated Vulnerability Repair is provided.

1.1 Large Language Models

The advent of highly capable Large Language Models (LLMs) has the potential to transform how Industrial Control Systems are programmed, as well as how software vulnerabilities are rectified. However, it is known that LLMs often produce flawed, uncompilable code [89]. Even then, state-of-the-art LLMs such as GPT-4o [113] and Llama 3 [132] have spurred significant changes in software engineering practices. Moreover, specialized models tuned for code generation have appeared [135], further increasing the potential for automated software augmentation and creation. Techniques such as Parameter-Efficient Fine-Tuning (PEFT) [87] and Low-Rank Adaptations (LoRAs) [60] extend the capabilities of these models, leading to an increase in performance while simultaneously streamlining the model training process [86, 179]. More recently, models incorporating a “Mixture-of-Experts” (MoE) have enabled significant gains in LLM performance [68]. Researchers have also studied "Prompt Engineering" – a method of refining LLM input to measurably improve the relevancy, accuracy, and quality of responses [32, 149, 185]. These advances in Large Language Models have created a unique opportunity for not only programming Industrial Control Systems, but also for combination with existing vulnerability repair techniques to automatically rectify common software bugs.

The popularization of Large Language Models has catalyzed significant interest in their use in many disparate fields. Specifically, the success of automated code generation has been greatly accelerated by improvements in the logical reasoning ability of these models [121]. State-of-the-art models like GPT-4o [113] have revolutionized the landscape compared to their predecessors such as CodeBERT [43]. Moreover, further advances in code synthesis have led to the refinement of methods for generating correct, understandable code from these models.

In addition to general-purpose models such as the aforementioned GPT-4o, specialized LLMs for code synthesis have emerged, such as CodeX [23], Code Llama [135], WizardCoder [95], and CodeGen [107]. For example, many of these models, including Code Llama, CodeX, and WizardCoder are trained on publicly available software repositories, which enhances their code generation abilities. Specialized systems such as CodeGen employ a multi-point synthesis scheme, where the user is periodically prompted for feedback on the generated code.

As the code generation abilities of LLMs improve, advanced evaluation metrics are needed to assess the viability of automated code synthesis methods. While existing code automation tools have studied the reliability of commercial products such as Microsoft Copilot [160], we focus on the evaluation of the models themselves. Benchmarks such as EvalPlus [89] have been created to more accurately measure the performance of LLMs on code generation tasks. These metrics build upon existing works such as HumanEval [23], and are better suited toward evaluating LLM-synthesized code. Other works have suggested using a litany of benchmarks to rank LLMs by performance [181].

1.2 Industrial Control Systems & Programmable Logic Controllers

Programmable Logic Controllers (PLCs) are an integral component of modern Industrial Control Systems (ICS). These controllers operate essential infrastructure such as oil pipelines [129], electric grids [92], and factories [174]. PLCs are real-time computers optimized for a domain-specific task, often programmed under the IEC 61131-3 standard [64]. We focus on the language defined by Part 3 of the IEC 61131-3 standard - Structured Text (ST). This language is the one that most closely resembles traditional programming languages in the standard. Then, we apply Large Language Models to generate PLC code that is efficient, safe, and verifiable from natural language specifications.

It is important to note that PLCs are Cyber-Physical Systems (CPSs), which places them in a unique risk category regarding cyberattacks. Then, it is important to follow existing best practices in securing these devices against malicious actors [41]. It has been shown that detecting security flaws in Cyber-Physical Systems as soon as possible is beneficial to the overall security and usability of a CPS product [164]. However, when Industrial Control Systems are networked – which is often the case [81] – a unique set of problems arise when faced with the challenge of security networked real-time PLCs [138]. As the rise of the Industry 4.0 paradigm marks a shift towards connected, always-on devices – especially in the manufacturing industry – new security methods are sorely needed to mitigate the potential threats enabled by these innovations [29]. Techniques aimed to preserve the *confidentiality, integrity, and security* of CPSs have been developed, with special emphasis on securing the supply chain and product lifecycle of the manufacturing system [28]. Other methods focus on the Cyber-Physical System itself, taking the approach of checking for cyber-kinetic vulnerabilities by incorporating the underlying physics of the CPS into the software analysis scheme [159]. Ultimately, these techniques all aim to secure CPSs and prevent malicious actors from extracting proprietary data or taking control of these sensitive devices.

1.3 Automated Vulnerability Repair

The automatic rectification of software vulnerabilities has remained a strong area of interest over multiple decades. Numerous methods have been developed to assist in rectifying these dangerous bugs [25, 47, 48, 50, 55, 90, 100, 102, 123, 124, 186, 192, 193, 194, 195]. One central technique to these works is their reliance on code analysis to repair code, often relying on an external compiler or static analysis tools [73]. However, these methods are unable to detect certain types of software problems. For example, bugs in the Java Reflection API are difficult to detect using these traditional techniques [78].

Other techniques have also been created to automatically fix software bugs. For example, a method using Generative Adversarial Networks (GANs) is effective for vulnerability repair [55]. This method allows for the repair of defective code without requiring a dataset of labeled training examples. However, unlike LLM4CVE, the model is evaluated on only synthetic code samples instead of real-world vulnerabilities.

Transfer Learning also been investigated for automated software repair. Researchers have demonstrated significant improvement over state-of-the-art methods, with the VRepair framework achieving almost a 50% increase in repair rate [25]. The use of the transformer architecture is also notable, even though the size of the VRepair model is significantly smaller than that of modern LLMs such as GPT-4o [113].

Similarly, Vision Transformers are also capable of rectifying code vulnerabilities. By using special queries to locate vulnerable code snippets, the model can generate more accurate and relevant repair suggestions [47]. This innovative model not only performed better than previous state-of-the-art models but also was reviewed positively by industry practitioners.

Code understanding models such as CodeT5 [166] have allowed for further improvements in the quality of generated fixes. As a precursor to modern Large Language Models such as GPT-4o and Llama 3, the CodeT5 architecture has enabled researchers to once again improve the total repair rate for software vulnerabilities [48]. The proposed framework – VulRepair – outperforms VRepair on several metrics due to extensive pre-training and the usage of Byte-Pair Encoding.

1.4 Additional Work

Appendix A contains a discussion regarding work on cross-domain security completed at the same time as the other works presented in this thesis.

Chapter 2

Large Language Models for Industrial Control Systems & Programmable Logic Controllers

Although **Large Language Models (LLMs)** have established predominance in automated code generation, they are not devoid of shortcomings. The pertinent issues primarily relate to the absence of execution guarantees for generated code, a lack of explainability, and suboptimal support for essential but niche programming languages. State-of-the-art LLMs such as GPT-4 and LLaMa2 fail to produce valid programs for **Industrial Control Systems (ICS)** operated by **Programmable Logic Controllers (PLCs)**. We propose **LLM4PLC**, a user-guided iterative pipeline leveraging user feedback and external verification tools – including grammar checkers, compilers and SMV verifiers – to guide the LLM's generation. We further enhance the generation potential of LLM by employing Prompt Engineering and model fine-tuning through the creation and usage of **LoRAs**. We validate this system using a **FischerTechnik Manufacturing TestBed (MFTB)**, illustrating how LLMs can evolve from generating structurally-flawed code to producing **verifiably correct programs** for industrial applications. We run a complete test suite on **GPT-3.5, GPT-4, Code Llama-7B, a fine-tuned Code Llama-7B model, Code Llama-34B, and a fine-tuned Code Llama-34B model**. The proposed pipeline improved the generation success rate from 47% to 72%, and the Survey-of-Experts code quality from 2.25/10 to 7.75/10.

To promote open research, we share the complete experimental setup, the LLM Fine-Tuning Weights, and the video demonstrations of the different programs on our dedicated webpage.[1]

2.1 Introduction

Programmable Logic Controllers (PLCs) are indispensable in the landscape of Industrial Automation – a market valued at $180 billion US Dollars in 2022 [109] – and these controllers drive essential infrastructure and industry such as oil pipelines [129], electric grids [92], manufacturing sites [174], and nuclear power plants [45]. PLCs are domain-specific real-time computers, integrating an *"Input-Compute-Output"* execution loop and running specialized programs created with one of five programming paradigms standardized under IEC 61131-3 [64]. Out of these five approaches, only **Structured Text (ST)** resembles conventional programming languages in regards to its syntax and structure. This property allows for automated code generation targeting the ST language using state-of-the-art techniques. Moreover, the usage of formal verification schemes for IEC 61131-3 programs [116] enables generated code to meet strict safety, complexity, and timing requirements.

Software in critical infrastructure and machinery are required to operate within a narrow safety margin and typically **necessitate extensive testing and verification**. In the typical project lifecycle, engineers and domain experts extensively analyze and design potential solutions before any programming effort is made, followed by dedicated synthesis

[1] https://sites.google.com/uci.edu/llm4plc/

and verification steps before deployment [37]. A visualization of this workflow is shown in Figure 2.1. Therefore, the governance of PLC control programs by **strict guidelines and requirements**, adds complexity to engineering tasks, resulting in hundreds of extra hours of expert-level effort, often even requiring Reverse Engineering [188] to recover the initial intent of the programmer. The primary goal of our proposed pipeline is **expediting the engineering effort** by offloading a sizeable majority of PLC-related problem-solving tasks to a dedicated LLM-Agent that assists engineers in their jobs. Our approach contrasts sharply with existing automated programming approaches, where the engineer is required to create the model design, synthesize code, and verify their solution manually.

Although some techniques exist to automate the synthesis of IEC-61131-3 PLC programs [54, 162] given a specification and synthesis paradigm – such as Linear Temporal Logic [76], or novel software-implemented frameworks such as MODI [16] – the engineering challenge of combining all parts of the PLC programming pipeline into a single unified model remains.

Recent developments in Large Language Models (LLMs) offer an alternative to legacy automation methods. However, given the irregularities in LLM code generation [143], naive use of LLMs in the Engineering Workflow – signified by inefficient prompting and blind execution of unverified output code – leads to unsafe operation [119], as showcased in Figure 2.1. Yet, foundational models such as GPT-4 [113] and LLama 2 [158] are challenging traditional approaches to automation and programming. Especially noteworthy is these models' instructional (i.e., "chat-instruct") capabilities, which allow for dynamic prompting based on a conversational input paradigm, opening the door for automated feedback mechanisms. Additionally, with the application of Parameter-Efficient Fine-Tuning (PEFT), Low-Rank Adaptations [61] have made domain-specific training easier and significantly reduced compute and data requirements for these tasks [86, 180]. Lastly, the LLM research community has pioneered "prompt-engineering" – a practice of optimizing prompts resulting in more accurate and relevant LLM responses [147, 150]. These advances offer a unique opportunity for combination with LLM-based code generation techniques for Industrial Control Systems, forming the basis of our automation pipeline.

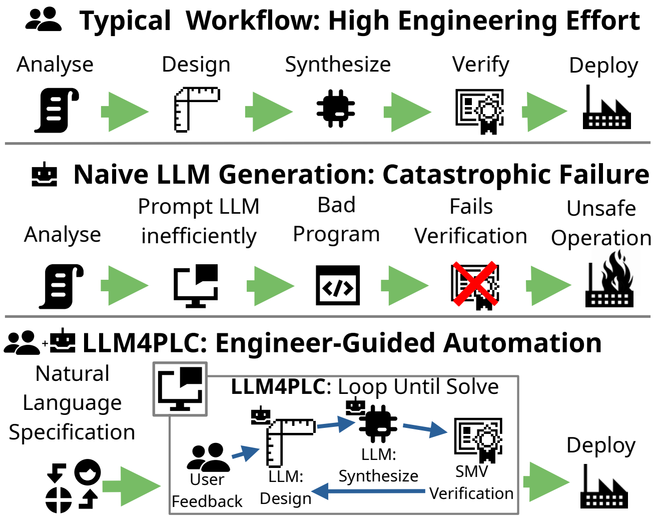
Typical Workflow: High Engineering Effort
Analyse
Design
Synthesize
Verify
Deploy
Naive LLM Generation: Catastrophic Failure
Analyse
Prompt LLM inefficiently
Bad Program
Fails Verification
Unsafe Operation
LLM4PLC: Engineer-Guided Automation
Natural Language Specification
LLM4PLC: Loop Until Solve
User Feedback
LLM: Design
LLM: Synthesize
SMV Verification
Deploy

Figure 2.1: Our proposed LLM-augmented workflow automates the high-effort stages of the PLC programming methodology

We introduce LLM4PLC, an automated pipeline that integrates Large Language Models (LLMs) with industry-standard PLC systems, employing automated verifiers and optional human feedback to ensure safe and efficient code deployment. After receiving an initial Natural Language Specification for the specified PLC system, the LLM Agent enters an automated iterative loop: generating a design schematic, synthesizing Structured Text (ST) code, and undergoing a sequential verification process that includes syntax checking [108] and model checking via the NuXmv software suite [18]. On a verification-stage success, the code can be immediately deployed; otherwise, errors from the verification stage are fed back to the LLM Agent for refinement of the erroneous ST code, with the option for human intervention during this process. This workflow is illustrated in Figure 2.1. Therefore, we summarize our key contributions as follows:

- Our work proposes and implements an automated language-driven system to verifiably program PLC devices from natural language descriptions of industrial plants.

- To the best of our knowledge, our work is the first to propose augmenting Large Language Models with automated external code verifiers to converge toward a solution iteratively.
- We present a detailed study of the generation potential of prevalent LLM models: GPT-3.5, GPT-4, Code Llama, and our fine-tuned Code Llama. We measure each configuration's generation success rate and average expert-appointed score.

The remainder of this chapter is organized as follows: Section 2.2 offers a literature review to ground our research within the existing academic landscape. Section 2.3, creates a framework for the foundations of PLC systems and identifies the issues our solution aims to address in LLM generation. This section also addresses background knowledge for other stages of our pipeline, including LoRA creation, syntax checking, and formal verification. In Section 2.4, we expand on the methodology behind LLM4PLC, detailing the design, implementation, and verification stages of our automated pipeline. Section 2.5 and Section 2.6 presents the experimental setup and results, showcasing the efficacy and reliability of our approach through quantitative evaluations. Finally, Sections 2.7, 2.8, and 2.9 discuss the broader implications of our findings, potential limitations, future directions, and conclude the chapter.

2.2 Related Works

Our work is primarily related to two fields of prior research: automated PLC programming and LLM Augmentation.

Automated PLC Code Generation has previously been explored by methods that do not utilize LLMs. In [140] the authors present a technique to translate GRAFCET – a graphical modeling language – into IEC 61131-3. Their editor binds GRAFCET elements to user-specific shapes with the ability to customize the behavior of each shape. Then, the graph is translated into an internal object model that is parsable by the GRAFCET toolchain, yielding code that might be executed in parallel [17]. The designer is required to provide a reference graphical solution, in contrast to our LLM-driven approach, which requires a set of natural-language instructions that define the desired operation of the machine.

Alternatively, previous works have developed an organization-level automation assistant. [127] proposes using a web service agent to assist production processes. Their methodology requires that each machine operating on the factory floor is accessible via a web interface implementing a well-defined API. Using OWL-S [96] and SPARQL [1], a world model is built according to the organization of web interfaces, and the internal states of the system are frequently updated due to Service Monitor invocations. OWL-S provides prescriptive commands to processes based on production goals created by SPARQL. Their tool only abstracts away the physical component of the program automation – a system design without physical access to the machines will still require that design and engineering tasks be handled by an adept engineer.

The use of LLMs in Industrial Automation has already been attempted in previous works. The team behind [33] created an intelligent assistant to assist in tasks such as process execution and troubleshooting. First, a knowledge graph based on information gleaned from user manuals is created. Experts then enhance the graph with their domain-specific knowledge. For each query, the information retrieval system extracts relevant text from the graph database, achieved in part by representing text passages as dense vectors using language models such as BERT or GPT. A limitation of this approach is that language models typically fail to return domain-specific information. As a result, the researchers trained a separate model to alleviate this shortcoming and then combined their results with a pre-trained language model. Afterward, the selected text passages are ranked by order of helpfulness, desirability, and relevancy. While this approach is helpful for maintenance and operations, it does not automate any part of the design and implementation phases.

LLM Finetuning for Code Review Automation has also been attempted by [93] through the use of LoRA-based methods to improve the Llama LLM. The authors' approach uses a

LoRA-Augmented LLM as an automated Code Review agent and forwards its feedback into the code-generation agent to enable better quality code output. Their findings show that LoRA is the preferred method for fine-tuning code generation in this context. Supplying stricter feedback in this process has also been explored in CodeRL [79], where the LLM is finetuned using Reinforcement Learning techniques by leveraging the pass rate on generated unit tests and compilers as a reward function. These two methods require a paired dataset of candidate code and automated feedback, which is difficult to acquire and may introduce biases into the framework. Rather than focusing on refinement in the training stage, our approach explores iterative refinement in the inference stage.

Dataset Preparation is also a crucial stage for code generation. Foundational Large-Language Models such as GPT-4 or Llama 2 are trained on a large and varied corpus of text [113, 158]. Recent efforts have focused on improving code generation or completion from LLMs through fine-tuning these models on specific input data. For instance, Thakur et al. focused on automatically generating hardware description language (specifically Verilog) using LLMs [154]. They fine-tuned various LLMs on a large Verilog corpus, which they assembled from project files on Github and numerous academic textbooks. Finetuning involved sharding the optimizer states across GPUs instead of using a quantized model or LoRAs. Other attempts at finetuning for specific output domains include natural language response [91, 167] and the esoteric language Hansl [153].

The authors in [53] present UniXCoder, an ML model incorporating *Abstract Syntax Trees (ASTs)* and comments into the generation scheme to create better output. Specifically, they are extracting the AST during the generation process so that the ML model can consider it while generating code. They also use comments as a source of guidance, as they are a helpful tool for understanding the underlying details and assumptions of a function, program, or algorithm. Similar to our approach, the authors create separate code completion and code generation datasets to validate their claims.

Other research groups have opted to build externally augmented language models – Tang et al. target software development languages by developing a database-equipped language model for domain adaptive code completion without fine-tuning [152]. They retrieve information from a database separate from the language model and therefore avoid excessive reliance on the weights of the language model. The next step is to use Bayesian inference for interpolating between the results of this database and the language model. Experimental results show improved performance of both CodeGPT [94] and UniXCoder [53]. Their method, while useful, does negatively impact completion speed. Ultimately, their conclusion indicated that fine-tuning was superior in terms of accuracy and code quality.

Code Verification is an important step for evaluating the LLM. There have been various methods for code verification in the literature, including control flow analysis, dynamic symbolic execution, and model checking.

[126] introduces the necessary software to perform static analysis of PLC programs. They cover rule-based approaches, syntax checking, and other techniques commonly accepted by the scientific community. [125] describes more complex techniques, such as generating an AST and performing control flow analysis to enable the creation of detailed insights regarding the logic flow of a PLC program. In our method, we build upon accepted static analysis knowledge in our verification pipeline, ultimately using this pipeline to verify the LLM-generated code.

Other works, such as [56], employ dynamic symbolic execution to generate test cases given a PLC code sample. Incorporating dynamic symbolic verification into a verification pipeline simplifies the process of proving functional correctness. Automating test generation removes human errors induced via manual test generation, such as missing test cases or redundant branch checking.

Model checking is a technique for verifying that the specifications defined for a model are met. [116] covers model checking as a verification scheme for PLC programs. The authors include SMV-based checking as a means to validate PLC software. We draw ideas from this paper regarding the usage of an SMV toolchain to perform formal model-based verification.

2.3 Background

The methodology adopted to develop our proposed pipeline builds upon the existing wisdom on PLC Software Engineering, Formal Verification Methods, and State-of-the-Art techniques in LLM prompting as well as Parameter Efficient Fine-Tuning (PEFT) using Low-Rank Adaptations (LoRAs). In this section, we provide the reader with the necessary prerequisite knowledge of each domain, while also setting up our motivation for the design choices adopted in the proposed method. First, we showcase the *prevailing approaches for efficiently querying LLMs* and the *fine-tuning methods for injecting knowledge into LLMs*, and then we delve into the *formal verification techniques used extensively in PLC programming.*

2.3.1 Large Language Models

Large language models (LLMs) leverage the attention mechanism in the Transformer architecture[161] to model sequences of increasing lengths. At the core of the Transformer model lies the self-attention mechanism, which computes attention scores for each pair of tokens in a sequence, allowing for long-range dependencies and relationships between tokens across long distances within the sequence.

All major LLMs use a Next-Token-Prediction (NTP) scheme to generate one token at a time. Formally, given a dictionary of possible tokens $\mathbb{T} = \{T_1, T_2, ..., T_N\}$, and a sequence of tokens $S = \{T_{s_1}, T_{s_2}, ...T_{s_m}\} \in \mathbb{T}^m$, an LLM model $\mathcal{M} \in R^N$ attempts to model the likelihood of the next token $T_{s_{m+1}}$ given:

$$\mathbb{P}\left(s_{m+1} = i | S\right) = \mathcal{M}\left(S\right)_i \tag{2.1}$$

Therefore, the probability of sampling a continuation sequence $\hat{S} = \left\{T_{s_{m+1}}, T_{s_{m+2}}, ..., T_{s_{m+M}}\right\}$ is expressed as:

$$\mathbb{P}\left(\hat{S}|S\right) = \prod_{i=0}^{M} \mathcal{M}\left(T_{s_1}, T_{s_2}, ..., T_{s_{m+i}}\right)_{s_{m+i+1}} \tag{2.2}$$

LLMs have gained substantial traction since the release of OpenAI's models, GPT-2 and GPT-3. At its release, GPT-2 stood out for its large size of 1.5 billion parameters [128]. Almost a year and a half later, GPT-3 came out with a staggering 175 billion parameter model. Larger models benefit from enhanced learning and the ability to store more information about complex relationships in data. While the results of OpenAI's models are remarkable, the model weights are not open-source, therefore eliminating the possibility of LLM fine-tuning[2] and creating a data privacy issue.

One of the first open-source models was Llama, released by Meta [157]. Contemporary results show that the 70B parameter Llama model performs comparably to GPT-3.5 on most benchmarks [157]. The most recent Llama release and the one most pertinent to

[2] After the writing of the original paper that comprises portions of this chapter, OpenAI introduced fine-tuning support for their language models.

our research is Code Llama [134], whose base model is Llama 2 fine-tuned on code-specific datasets [134].

Prompt Engineering

Once the LLM architecture has been selected, one can begin prompting the LLM as part of training and inference. Prompt engineering refers to creating prompts instructing the LLM to produce a desired response. Prompts are meant to provide context, thereby facilitating LLM response generation. For example, in writing code for a specific language, one could create a prompt that provides context by including an example program written in the language and then asking the LLM to fix a piece of code. This is exactly how our approach goes about prompting the LLM during training. During inference, the prompt is simply a command that asks the LLM to complete unfinished code.

Formally, a prompt $\mathcal{P}$ is a sequence of tokens $\{T_{p_1}, T_{p_2}, ..., T_{p_K}\}$ that is inserted as a prefix for any sequence S. Different prompts naturally lead to varied degrees of success, with "self-guidance" prompts achieving the best results [169]. Self-guidance prompts decompose a thought process into steps. We leverage self-guidance when we attempt compilation of the code outputted by the LLM and use any resulting errors as feedback to the LLM so that it can make the necessary corrections.

Parameter-Efficient Fine-Tuning (PEFT)

PEFT aids in adapting an LLM to a particular task. LLMs typically have billions of parameters and training them on new tasks can be computationally demanding and time-consuming [105, 137, 156]. PEFT encompasses techniques that improve model performance on a specific predefined task without compromising resource efficiency or training duration. Of the many approaches to PEFT, Low-rank Adaptation (LoRA) is a popular, performant one, and our pipeline heavily incorporates this technique.

LoRAs help address the common issue of over-fitting and catastrophic forgetting [30, 183]. The LoRA technique is applied to reduce the memory that is used up by the update weights, ΔW. This process involves low-rank decomposition of the update weight matrices [61] as can be seen in Equation 2.3, where $W_0 \in \mathbb{R}^{dxk}$, $B \in \mathbb{R}^{dxr}$, $A \in \mathbb{R}^{rxk}$, and $r \ll min(d, k)$. The process is visualized in Figure 2.2, where the tokens are passed as an input, x with dimension d.

$$W_0x + \Delta Wx = W_0x + \alpha * BAx \tag{2.3}$$

Instead of using a very large matrix, the update matrices are broken down into two smaller matrices through low-rank decomposition. Our work further exploits the benefits of this technique by training multiple LoRAs in parallel. In essence, the additional domain-specific knowledge is injected into the auxiliary network, offering augmented problem-solving capabilities without a degradation in latency. The size of these auxiliary passes is controlled by the rank r, a tunable parameter that controls the size of the LoRA. This parameter controls both the model's efficacy at incorporating new knowledge as well as its training time and required dataset size. Lastly, the LoRA injection is controlled by a strength scale α, which dictates the extent to which the auxiliary network influences the primary model. Then, α serves as a

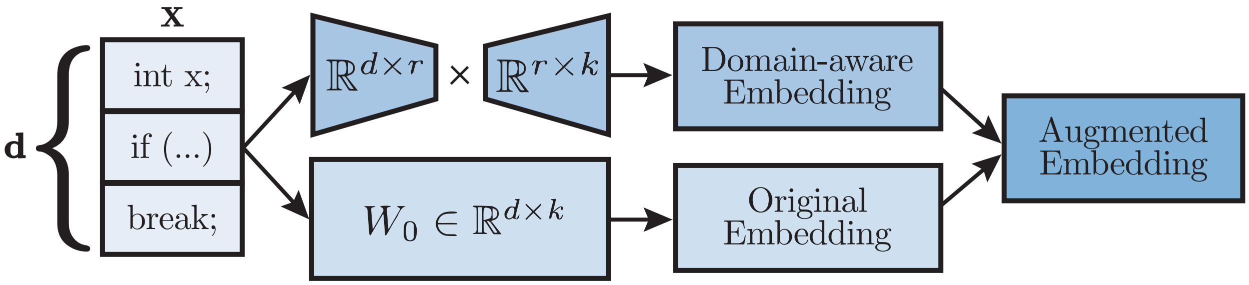

Figure 2.2: LoRAs create domain-specific embeddings that are aggregated into the original knowledge base

tunable hyperparameter, allowing for a calibrated trade-off between domain-specific expertise and generalization performance. By adjusting α, we can fine-tune the model's reliance on the LoRA, effectively balancing the incorporation of specialized knowledge against the risk of overfitting to a particular domain.

2.3.2 Model-Based Design

Model-Based Design (MBD) is an engineering paradigm that leverages mathematical and graphical modeling to facilitate the analysis, implementation, and simulation of complex systems. Originating from control engineering and systems theory [101], it has been successfully applied across diverse domains, including automotive[103], aerospace [11], cyber-physical systems [67], and industrial automation [148].

In traditional design methodologies, each development stage – requirements engineering, architecture design, implementation, and verification – is often isolated, requiring manual and often error-prone intervention to transition between stages. MBD, on the other hand, emphasizes an integrated framework where predefined models serve as a formal and comprehensive representation of the system, eliminating these transition-induced errors. These stages are standardized in [101]. An example of MBD can be seen in Figure 2.3.

Lastly, verification and validation processes are tightly integrated into the typical MBD design flow. Because the model serves as the golden reference for the system, it can be used to rigorously test the final implementation, ensuring that it meets the agreed-upon requirements and constraints.

Model-based design is a holistic approach to system development that offers significant efficiency, reliability, and maintainability benefits. Its emphasis on early-stage simulation and analysis enables proactive problem-solving, making it an increasingly essential technique for

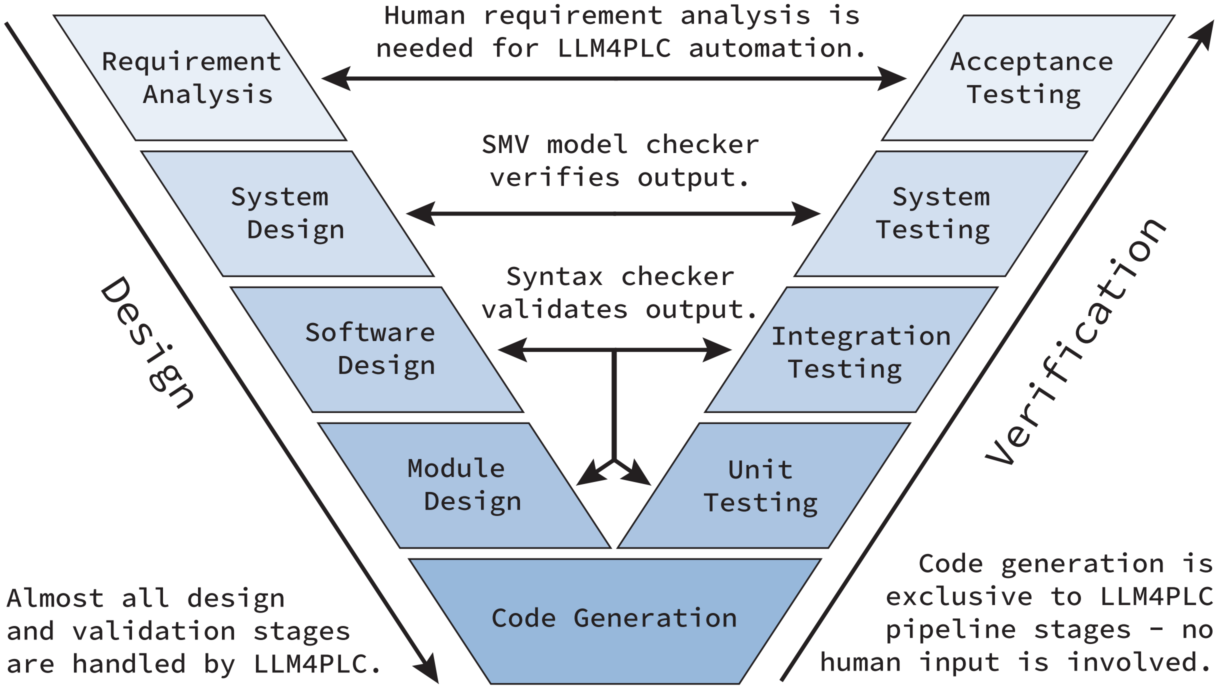
Human requirement analysis is needed for LLM4PLC automation.
Requirement Analysis
Acceptance Testing
SMV model checker verifies output.
System Design
System Testing
Syntax checker validates output.
Software Design
Integration Testing
Module Design
Unit Testing
Code Generation
Design
Verification
Almost all design and validation stages are handled by LLM4PLC.
Code generation is exclusive to LLM4PLC pipeline stages - no human input is involved.

Figure 2.3: An example of a Model-Based Design process

designing and implementing complex systems. The typical engineering workflow previously presented in Figure 2.1 adheres to this paradigm. LLM4PLC uses these successive stages as guidelines for querying the LLM as we present in section 2.4.

2.3.3 Syntax Checkers and Formal Verification

Without a methodology in which human-generated or LLM-generated code can be evaluated for safety, completeness, and accuracy, several bugs and other undesirable behavior may persist in the code. In environments such as nuclear power plants, or satellite control systems, correctness is not only desirable but wholly necessary for operation [45]. It is well known that LLMs often produce code that does not conform to language specifications [143]. There are several ways in which software can be checked for these deficiencies, many of which vary in complexity and effectiveness. We are primarily concerned with syntax checkers and compilers, which provide a first step towards ensuring the correct operation of the program.

Then, syntax checkers and compilers help alleviate this issue by providing feedback on LLM-generated code.

Syntax Checkers

The first step in transforming candidate code to a working program is to check that it conforms to the standards of the programming language used. Without this step, the code will not compile, and remedial action is required to be taken. Automated approaches to fix syntax errors exist [3], but integrating syntax checkers into an LLM code generation pipeline enables the repair of errors previously unaccounted for. Using these tools over multiple cycles of the LLM4PLC pipelines provides a deeper insight into code deficiencies, and better prepares the LLM to take corrective action.

Formal Verification

Formal verification of programs and algorithms through Symbolic Model Checking is an essential step toward deploying PLC code in hazardous environments. These tools take in the candidate code as well as strict constraints on the operation - for example, the upper temperature limit permitted. Approaches using interactive theorem provers [14], or Symbolic Model Checking [14] have become widely used for this purpose. The focus of LLM4PLC is on the usage of these tools to formally verify LLM-generated code using an SMV model generated from a plant specification document. Then, the generated PLC code is verified using the SMV model as the reference.

2.4 Methodology

LLM4PLC is a user-guided iterative pipeline designed to help LLMs generate code for Industrial PLCs. We aim to address the limitations of current state-of-the-art Large Language Models (LLMs) in this domain. Our pipeline integrates user feedback loops and incorporates a suite of external verification tools: grammar checkers and a nuXmv verifier. The pipeline is optimized via Prompt Engineering and model fine-tuning mechanisms utilizing LoRAs.

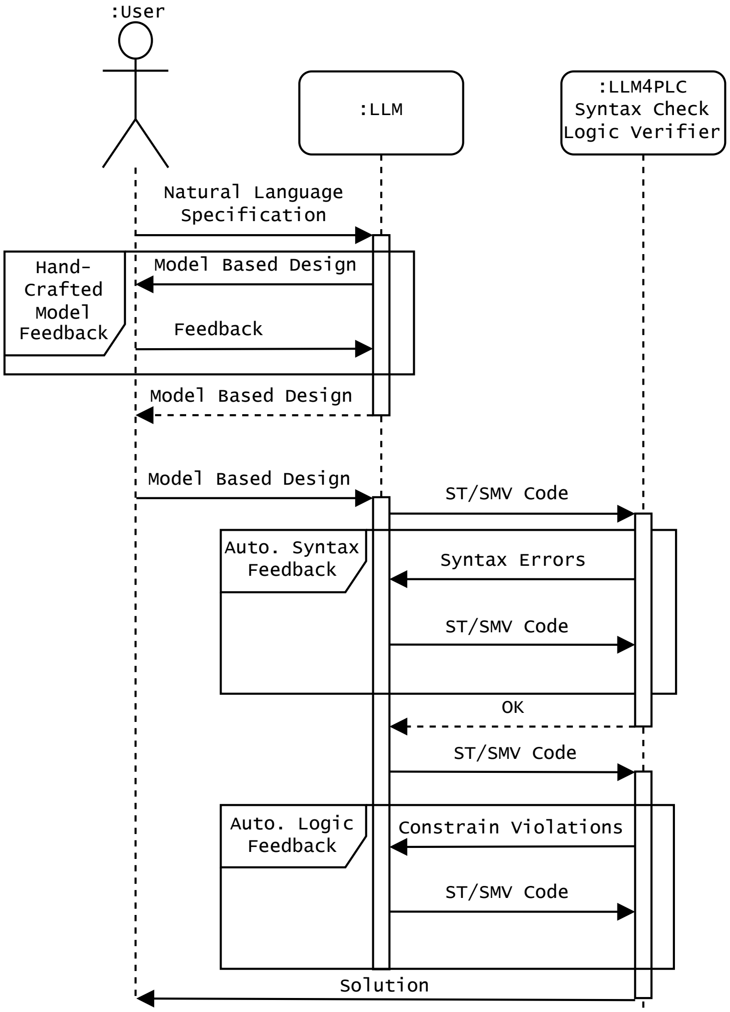

Figure 2.4: UML Representation of proposed engineering pipeline

The approach of the implementation follows a top-down view of the typical workflow in industrial settings where the engineer's effort is showcased in a representative UML diagram in Figure 2.4. Additionally, for better clarity on how blocks are interconnected, we present all modules used in LLM4PLC in Figure 2.5. We discuss the flow in the following sections.

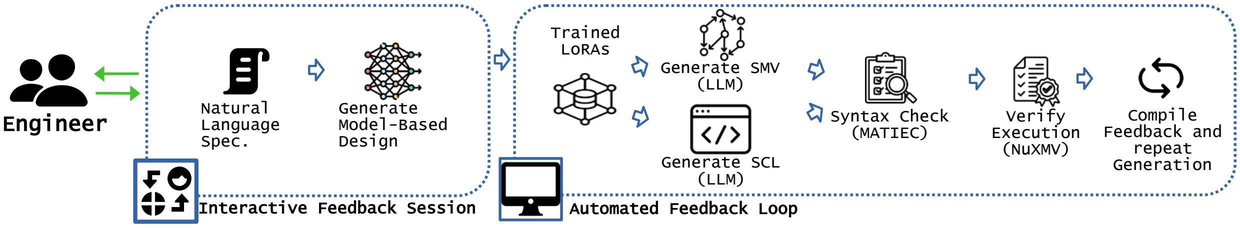

Figure 2.5: Summary of all blocks used in LLM4PLC

2.4.1 Model-Based Design

Model-Based Design (MBD) in the context of industrial PLC programming introduces a structured, systematic approach that enhances both the efficiency of the development process and the reliability of the resultant code. By following task-specific prompt guidelines for planning, the LLMs can sift through given specifications and requirements to create a comprehensive plan for the subsequent pipeline stages. The first step in LLM4PLC is to generate complete function and block declarations and their corresponding signatures, serving as an executable blueprint that ensures alignment between the development process and the defined natural language specifications. Moreover, the detailed planning phase can highlight any ambiguities in user requirements, providing an opportunity for prompt clarification and thereby minimizing the risk of deviations or errors in the subsequent code implementation phase. An example MBD plan generated by our pipeline exists on our dedicated website.[3]

Finite State Machines (FSMs) provide a significant advantage for our development methodology. Explicit planning around FSM states not only allows for a clear roadmap

[3] https://sites.google.com/uci.edu/llm4plc/home

but also optimizes the PLC scan cycle. Because PLC logic does not follow the traditional loop-based execution model, using a state variable to track the system state is crucial. Each state in the FSM is responsible for a single operation, making the execution predictable and easier to debug. In the MBD prompt used alongside the Natural Language Specification, we constrain the LLM to follow an FSM design for the design solution.

2.4.2 Syntax Checkers

Using LoRAs, we transform the MBD plan into its associated Structured Text representation in Siemens' Structured Control Language. Then we use an open-source IEC 61131-3 Structured Text compiler to perform syntax checking of LLM-generated code. Specifically, we use MATIEC [108] to search for syntax errors in the generated code. If any error is detected, the output of MATIEC is then fed into our pipeline to create a 'correction prompt' for the LLM in the next stage of the pipeline. We feed only one compiler error and its associated generated prompt per cycle for several reasons. Firstly, we want a targeted fix from the LLM for each error, rather than continuously prompting to fix multiple errors at once. Moreover, for each compiler error, subsequent errors can be dependent on the original error. Take for instance a missing semicolon which directly causes another compiler error on a subsequent line. By feeding only one error and correction prompt at a time, we work to minimize the total number of prompts and pipeline iterations needed. Note that this would also allow for multiple errors to be fixed in one iteration cycle (i.e. when one error is wholly caused as a result of a preceding error).

2.4.3 Formal Verifiers

Upon generating compilable PLC code through our pipeline, the next step requires verification using nuXmv, a symbolic model checker based on the SMV paradigm [18]. This verification process is crucial to verify that the produced code adheres not only to syntactical standards but also to functional and safety requirements intrinsic to industrial automation scenarios.

To conduct this verification, we translate the constraints from Natural Language to SMV using a feedback loop similar to the one used for ST. The provided plant specification outlines the behavior, constraints, and requirements of the industrial process the PLC code is intended to control. The SMV specification file, on the other hand, encapsulates the formal properties and conditions that the PLC code must satisfy. We use these two elements in conjunction with nuXmv as a means to perform formal verification on the PLC code. This approach ensures that the generated code is not only compilable but also reliable and safe for deployment in a real-world industrial setting. Through this verification mechanism, any discrepancies between the intended and actual behavior of the PLC code can be promptly identified and rectified before deployment, enhancing the robustness and credibility of our pipeline.

2.5 Experimental Setup

For the purposes of our study, we target GPT-3, GPT-4, Code Llama 7B, and Code Llama 34B, motivated by several considerations towards a comprehensive evaluation. GPT-3 [184] and GPT-4 [113] represent state-of-the-art general-purpose language models, serving as effective benchmarks for general text-to-code translation tasks. Code Llama 7B and Code Llama 34B [157] are specifically designed for code generation: Their respective 7-billion and 34-billion parameter counts offer a gradient of computational complexity, enabling the investigation of the trade-offs between performance and computational resources.

2.5.1 Prompting

Two prompt types, *Zero-Shot* and *One-Shot*, were employed to generate outputs from the LLMs. The One-Shot prompt incorporates a representative sampling of Structured Text syntax and code elements, thereby providing contextual cues for improved code generation. In contrast, the Zero-Shot prompt simply requests code generation without any contextual guidance. For each model we assess its performance against both prompt types.

2.5.2 LLM Fine-Tuning

We fine-tune the Code Llama 7B and Code Llama 34B through the creation of Low-Rank Adaptions (LoRAs). GPT-3.5 and GPT-4 do not provide an interface for the training of LoRAs, so we use these models in their default configurations. Then, we leverage this knowledge to train a set of four LoRAs for both code completion and code fixing tasks – two for each of Code Llama 7B and Code Llama 34B. We deploy training and inference on a compute node equipped with an Nvidia A100 as well as 128GB of main system memory.

2.5.3 Dataset

We generate a training, validation, and testing dataset from the OSCAT IEC 61131-3 Library [114]. Specifically, we run automated tests to cull deficient ST files in the OSCAT dataset, then create three separate datasets to test our pipeline's capabilities extensively.

Automated Tests

We attempted to compile each ST file in the aforementioned dataset to validate our testing data and discarded all files that did not successfully compile. MATIEC is chosen as the ST compiler for this task. This left us with 636 viable ST files to create our dataset from. We use 596 samples for Training and 40 samples for testing (95/5 Train/Test split).

Dataset Generation

We derive three datasets from the original OSCAT Dataset: (1) Generation, (2) Completion, and (3) Fixing. The generation dataset is simply the OSCAT dataset after culling all non-compilable files. We build the Completion dataset by randomly truncating files in the Generation Dataset. This allows us to simulate the code-completion abilities of the LLM and test the ability of our pipeline to aid in the code-completion task. The Fixing dataset is created by removing random lines from the Generation dataset until the resulting ST file is no longer compilable. This dataset aims to test the LLM's ability to synthesize solutions to specific syntax errors within the code. Once again, we use MATIEC in order to verify the compilation status of each file during this process.

LoRA Training

As described previously, we finetune a pair of LoRAs for each model: one for completion and one for fixing. Section 2.3 presents several parameters relevant for training LoRAs. We choose rank $r = 64$, strength scale $\alpha = 128$, and $Batch\_Size = 256$ and we run our training procedure over 5 epochs. These parameter choices were made according to prevalent wisdom in the LLM community as well as our team's experience in training and deploying such models.

2.5.4 Metrics

Our metrics involving the experimental dataset include pass rate, compiler error count, and a human evaluation of code quality. Pass rate is evaluated using the pass@k metric, as outlined in [22]. For each of the 40 test files, a single output is generated (k=1), yielding statistically significant results. Finally, we compare the amount of Engineer-Hours required for each of the configurations.

The pass rate, assessed via the pass@k metric, serves as an indicator of the method's accuracy in code generation. A high pass rate indicates the procedure's reliability in producing syntactically and semantically correct code.

The compiler error rate offers insight into the robustness of our approach. A lower rate signifies a reduced likelihood of generating syntactically flawed code, highlighting the method's precision in adhering to language-specific rules. We track the number of detected syntax errors per generated files and compute the average errors per file.

The pass rate and error rates are not individually indicative of success. Since LLM4PLC rejects all code that contains errors, an increase in error count does not necessarily imply a failure of the pipeline. This metric serves as an auxiliary measure to examine after the pass rate has been considered. For example, if a pipeline stage has a higher pass rate but also an increase in compiler error count, this is still beneficial as only the successful iterations (i.e. those with zero errors) are forwarded to the verification stage of the pipeline. Additionally, if a pipeline stage has a lower pass rate but a low compiler error count it does not indicate success, since success is firstly measured by compiler pass rate.

The human code quality evaluation provide a nuanced evaluation of the generated code's readability and maintainability. High scores in this dimension underscore the practicality of our method, emphasizing its potential for seamless integration into existing human-driven development processes.

2.5.5 Testbed

Our lab has deployed the FischerTechnik Manufacturing Testbed (MFTB)[46] – an integrated platform that simulates a miniaturized version of common manufacturing processes. The MFTB serves as a sophisticated test environment for studying and validating various aspects of automation, digital control systems, and operational efficiency in a manufacturing setting. The testbed is a complex cyber-physical system integrating various inputs and different types of outputs as follows:

- **Digital Inputs:** 22
- **Analog Inputs (0-10V DC):** 1
- **Fast Counting Inputs:** 10 (for direction detection)
- **Outputs (24V):** 35

Functional Modules in the MFTB serve four distinct purposes, and are summarized as follows:

1. Sorting Line With Color Detection
2. Multi Processing Station With Oven
3. Automated High-Bay Warehouse
4. Vacuum Gripper Robot

These modules interact in a closed-loop fashion, allowing for a self-contained material cycle. Items are retrieved from the Automated High-Bay Warehouse, undergo processing at the Multi Processing Station With Oven, sorted by color using the Sorting Line With Color Detection, and are ultimately returned to the Automated High-Bay Warehouse for storage. The setup can be seen in Figure 2.6. We write a simple natural language description of the specifications of the HighBay module and we run the corresponding prompt through our pipeline. We deploy the formally verified code on the MFTB to verify its operation on physical hardware. **The specification, results and demonstration videos are included in our website.**[4]

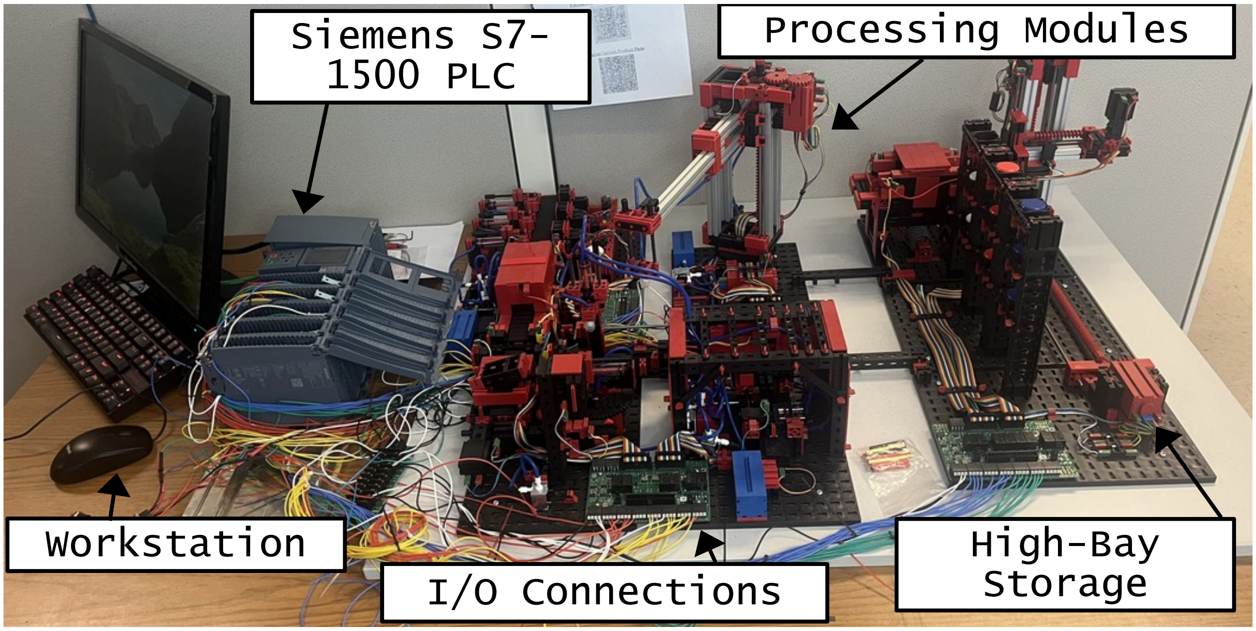

Figure 2.6: The FischerTechnik Manufacturing Testbed (MFTB) deployment in our lab

2.6 Results

In this section, we present the outcomes of our empirical evaluations. We measure the efficacy of our method – LLM4PLC – in terms of three critical metrics: pass rate, compiler error

[4] https://sites.google.com/uci.edu/llm4plc/home

count, and a human evaluation of code quality. We also compare the needed engineering-hours needed to set up each of the solutions. These metrics offer a comprehensive assessment of our approach, covering aspects such as accuracy, robustness, and practical utility in code generation processes. The experimental setup for each of these metrics is outlined in the following subsections.

2.6.1 Pass Rate

For evaluating the pass rate, we pass the set of 40 dedicated test files through various configurations of LLM4PLC pipeline stages. For each of the resulting 40 input files, we determine the pass rate as defined in Section 2.5.4. The results are presented in Figure 2.7,

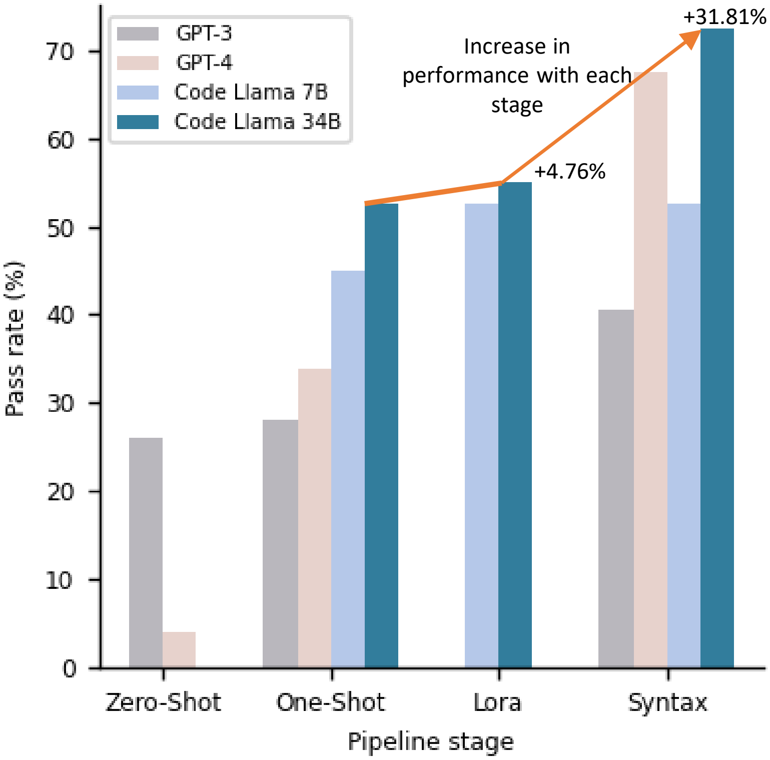

Figure 2.7: Pass rate for each model and configuration type

where *Zero-Shot* has results for the Zero-Shot prompt, *One-Shot* has results for the One-Shot prompt that includes SCL code as an example, *LoRA* has results for the LoRA-Finetuned Models and One-Shot prompt, and *syntax* incorporates the grammar checker in addition to the LoRA and the One-Shot prompt. As more components are added to our pipeline as we progress through the stages, the compilation pass rate increases, reflecting the efficacy of our proposed method, specifically in the final stage, which includes the One-Shot prompt, a LoRA, and a grammar checker. Each added component makes a positive contribution to the pass rate. The most significant increase can be seen when the grammar checker is incorporated, which when used with the optimized prompt and LoRA, results in a 72.5% pass rate for the Code Llama 34B model.

2.6.2 Compiler Error Count

Next, we compute the number of compiler errors generated for each output code file as per the procedure defined in Section 2.5.4. We then average the number of errors over the set of test files to normalize the results. The average errors per test file is then the final metric we obtain. The results are presented in Figure 2.8. The errors for *LoRA* and *syntax* are larger than that of just the *One-Shot*. However, as evidenced by the Pass Rate over each successive configuration, our pipeline can correct these errors such that the compilation rates are higher for *LoRA* and *syntax* compared to just *One-Shot*. One pattern we observed is the tendency for the LLM in this setting to expand variable definition blocks well beyond their intended size. These blocks often contained multiple syntax errors per line, and combined with the excessive generation of them in some files led to the abnormally high compiler error rate for this configuration. Note that the naive prompt generation for the Code Llama models had a 0% pass rate its error rate was omitted from the graph for that reason.

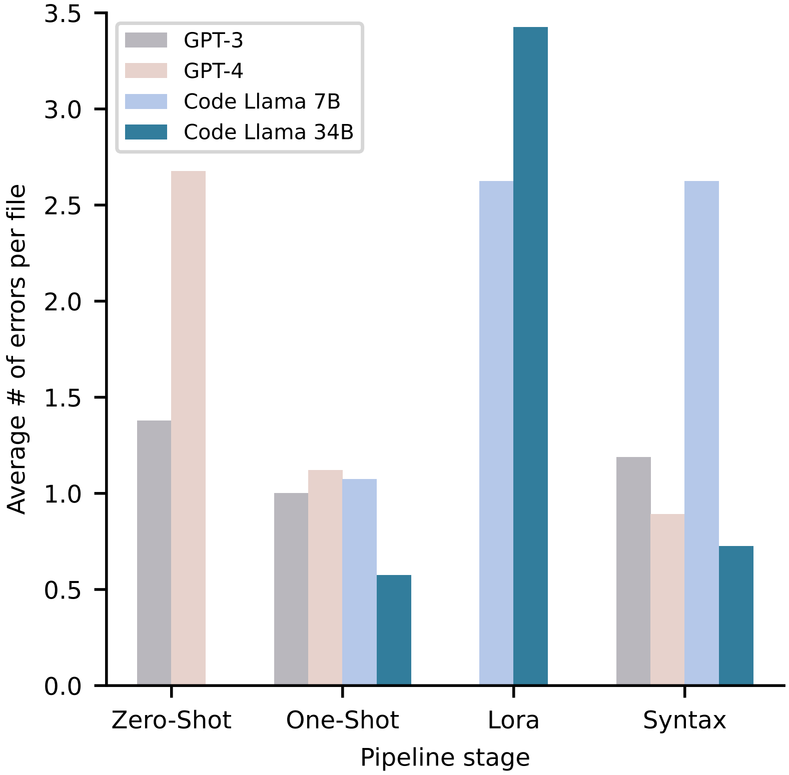

Figure 2.8: Cumulative syntax errors recorded over the dataset for each model configuration

2.6.3 Human Quality Metrics

The human quality metrics are assessed using a panel of experts of various experience with PLC programming. The panel evaluates the generated code based on predefined criteria: correctness, maintainability, and conformance to industry coding standards. The experts were informed of the purpose of this study but they were not informed which model created which code. We asked the participating individuals to score the criteria by answering the following questions:

- Correctness: On a scale of 1-10, how accurately does the generated code perform the intended function without errors?

- Maintainability: On a scale of 1-10, how easy is it to understand, modify, and extend the code?
- Conformance to Industry Coding Standards: On a scale of 1-10, how well does the code adhere to established best practices and coding standards?

This qualitative evaluation complements our quantitative metrics, providing a multi-faceted view of the system's performance. High scores in this area would signify that the generated code is not only accurate and robust but also practical for real-world applications. The results are presented in Table 2.1. Notably, the use of LLM4PLC enhanced the perceived quality of the code.

Model	Correctness	Maintainability	Best Practices
LLama-34B Naive	2.25	3.25	2.5
LLama-34B LLM4PLC	6.5	4.75	4.0
GPT-4 Naive	2.25	3.75	2.75
GPT-4 LLM4PLC	**7.75**	**6.125**	**6.0**

Table 2.1: Average Expert-Appointed score

2.6.4 Engineering Effort

The question of engineering effort is critical when choosing between GPT-based solutions, LoRA-augmented LLMs, and traditional hand-programming techniques for PLCs. Our empirical studies show marked differences in the time required for each approach, offering insights into their practical applicability.

For GPT models, the setup is remarkably efficient, often taking only a matter of minutes to integrate the API and commence code generation. In contrast, setting up a LoRA for an existing open-source model like Code Llama entails a considerable amount of time to collect data and train.

Hand-programming of PLCs, while a well-understood method, is by far the most time-consuming, often requiring orders of magnitude more time than the iterative LLM-based approaches we present. While this method provides the highest degree of control and specificity, it also demands the most significant investment in terms of time and specialized human resources. For this metric, we take the average time it took for each member of our team to set up the program for the high-bay module.

We present the table of effort needed in Table 2.2.

Approach	Setup Time	Run-Time
Hand-Coding	-	12 hours
Lora-Based	32 Hours	6 minutes
Out-Of-The-Box	-	**2 minutes**

Table 2.2: Engineering-hours required for each approach

2.7 Discussion

This section delves into the broader implications of our proposed pipeline, LLM4PLC. We explore its conceptual and practical impacts. Then, we examine how the pipeline accelerates the development iteration cycle, thereby allowing for more efficient Industrial Automation workflows. We include an excerpt from an example of the MBD Plan generated for the MFTB High Bay in Listing 2.1, an example of an erroneous generation in Listing 2.2, and its corresponding correction after the nuXmv and Compiler Checks in Listing 2.3.

```
LIST THE TRANSITIONS BETWEEN STATES:
STATE 0 -> STATE 1: IF "RED BTN" = TRUE AND NO COMPONENT IN HAND
STATE 1 -> STATE 2: IF xpos = 734 AND ypos = 405
STATE 2 -> STATE 3: ONCE ARM EXTENDED AND COMPONENT LIFTED
[...]
```

Listing 2.1: Excerpt of a generated MBD Transition Logic

```
componentHomeSlot: TUPLE OF (INT, INT);
      [...]
IF xPos < 1950 THEN
      [...]
IF "RED BTN" THEN
   IF xPos < 734 THEN
      [...]
```

Listing 2.2: Excerpt of a generated erroneous Structured Text Code

```
componentHomeSlot: ARRAY[1..2] OF INT;
    [...]
IF xPos < 1900 THEN
    [...]
IF "RED BTN" AND NOT componentInHand THEN
   IF xPos < 734 THEN
    [...]
```

Listing 2.3: Excerpt of a generated corrected Structured Text Code

2.7.1 Impact

Our proposed pipeline accelerates the iteration cycle, from model training to code verification. This acceleration improves not only development speed but also the quality and reliability of the generated code, thus having a substantial impact on the end application.

Moreover, the pipeline's speed enables real-time or near-real-time code adaptation, making it possible to apply updates or patches in time-constrained environments. This responsiveness is particularly invaluable in critical systems where failure to correct an issue promptly could result in substantial economic losses or safety risks.

2.7.2 GPT vs Code Llama

In our experiments, the fine-tuned Code Llama 34B model with the use of a grammar checker outperformed the general-purpose GPT models like GPT-3.5 and GPT-4 on several key metrics. Even with our grammar checker, the GPT models underperform compared to Code Llama 34B. This can largely be attributed to the fact that fine-tuning allows the LLM to learn the nuances of Structured Text, which GPT models have not been exposed to before. Note that due to the closed-source nature of the GPT models, fine-tuning is not a possibility. Moreover, our experiments showcased the efficacy of LoRAs for increasing model performance. It is important to note that an increase in model size (i.e. the total number of parameters) can affect the performance of a model, as a larger number of parameters allows for richer context-awareness, which improves output accuracy. We attribute the better performance of GPT-4 with the grammar checker compared to Code Llama 7B with the LoRA and grammar checker to this reason. Even so, Code Llama 34B with the LoRA and grammar checker, despite having fewer parameters than GPT-4, outperforms GPT-4 with the grammar checker, demonstrating the strength of LoRAs when it comes to improving the quality of an LLM's output on specific output domains.

Furthermore, The closed-source nature of GPT models restricts auditability and customization, creating potential barriers for applications requiring rigorous verification or specialized features. This opacity not only hinders the research community from fully understanding the limitations of these models but also inhibits collective efforts to improve upon these shortcomings. Models that allow fine-tuning can be customized, which evidently can improve performance and generalization to unseen data.

2.8 Industry Challenges

As automation and digitalization increasingly pervade industrial settings, several emergent challenges must be addressed.

The fast advancement of technology creates major challenges in the industry to maintain a sufficiently skilled workforce that can, on the one hand, leverage new technologies quickly, and on the other, preserve the industrial know-how that has been acquired from years of practical and hands-on experience. The use of LLMs to collect relevant technical knowledge and leverage it for automating engineering tasks and for guiding engineers is therefore of special interest in the industry, as it can accelerate the process of knowledge gain for new staff, reduce the negative effect of knowledge loss from staff rotation, and lessen the influence of staff skillset on engineering efficiency and quality. In addition, the overall engineering cost is expected to be reduced as a consequence of increased automation of the engineering tasks.

Another challenge is the issue of explainability and trust. Industrial applications often have stringent safety and reliability standards. Incorporating machine learning solutions that provide not just high performance but also interpretability will be crucial. The ability to understand and trust the decisions and actions of an AI system is a non-negotiable requirement in critical infrastructures where errors can lead to significant economic or human loss. Although our study shows that the code achieves a good degree of interpretability, un-regulated use of such methods will inevitably lead to unverified and unmoderated deployments in critical systems.

2.9 Conclusion

In summary, our research introduces the LLM4PLC framework, a user-guided iterative pipeline designed to improve the automated code generation capabilities of Large Language Models, specifically for Industrial Control Systems operated by Programmable Logic Controllers. By incorporating user feedback and leveraging external verification tools, we significantly enhance the model's output validity. This pipeline is further refined by Prompt Engineering techniques and model fine-tuning methods like LoRAs. Empirical validation on a FischerTechnik Manufacturing TestBed demonstrates notable improvements in both the rate of successful code generation and the quality of the generated code, as rated by experts. Our contributions pave the way for more reliable and efficient applications of LLMs in industrial settings, inching us closer to achieving verifiably correct program generation in these critical domains.

Chapter 3

Large Language Models for Few-Shot Vulnerability Repair

Software vulnerabilities continue to be ubiquitous, even in the era of AI-powered code assistants, advanced static analysis tools, and the adoption of extensive testing frameworks. It has become apparent that we must not simply prevent these bugs, but also eliminate them in a quick, efficient manner. Yet, human code intervention is slow, costly, and can often lead to further security vulnerabilities, especially in legacy codebases. The advent of highly advanced Large Language Models (LLM) has opened up the possibility for many software defects to be patched automatically. We propose **LLM4CVE** – an LLM-based iterative pipeline that robustly fixes vulnerable functions in real-world code with high accuracy. We examine our pipeline with State-of-the-Art LLMs, such as **GPT-3.5**, **GPT-4o**, **Llama 3 8B**, and **Llama 3 70B**. We achieve an increase in ground-truth code similarity of 20% with Llama 3 80B. To promote further research in the area of LLM-based vulnerability repair, we publish our testing apparatus, fine-tuned weights, and experimental data on our website.[1]

[1] https://sites.google.com/view/llm4cve

3.1 Introduction

Human developers are prone to costly mistakes when designing software systems. Often, these are not simple errors, but a fundamental misunderstanding of cybersecurity concepts [163]. These vulnerable programs are targets for criminals to steal credentials, assets, and other valuable items from end-users [20]. Moreover, the frequency of these types of attacks is steadily increasing [65]. As a result, the ability to quickly and efficiently rectify software bugs has become more critical than ever before.

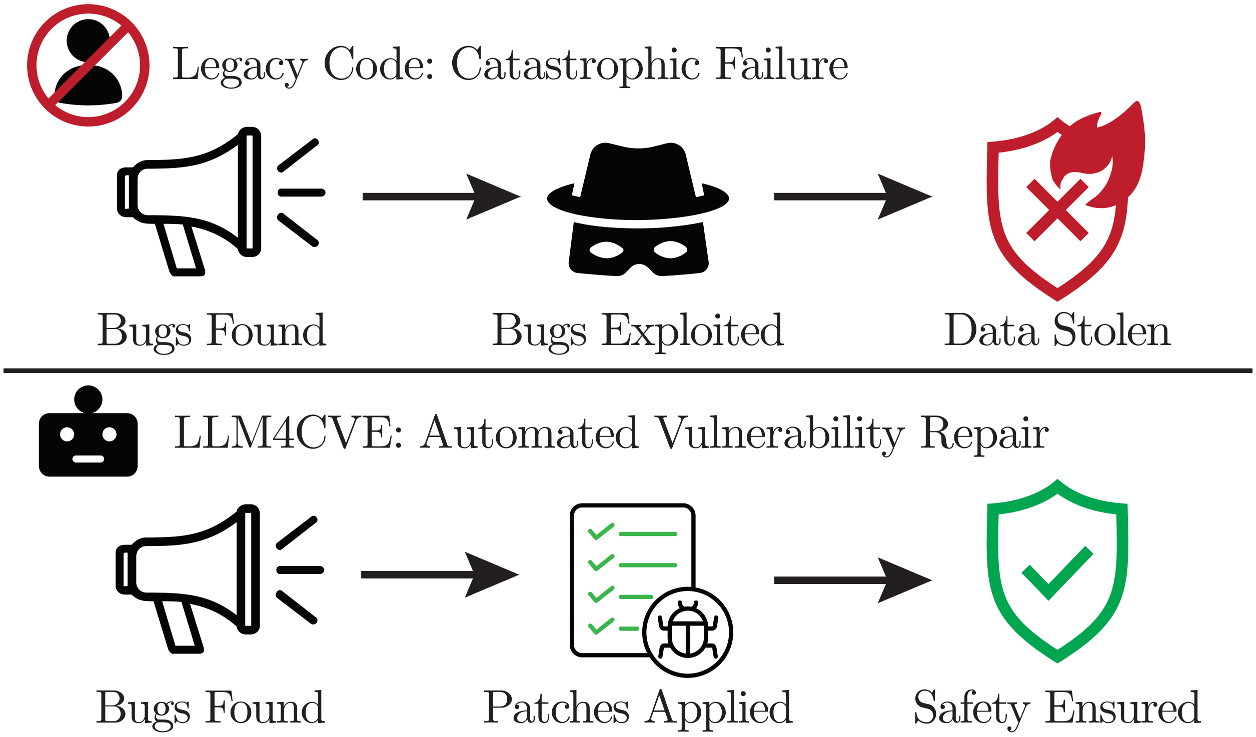

Figure 3.1: How LLM4CVE assists in preventing bug exploitation

Already, we have seen the significant impacts of these bugs – billions of dollars in lost economic value [63, 65, 110], countless man-hours spent on bug resolution [115], and the leakage of sensitive user data to malicious actors [74, 130]. Cyberattacks – like the one done to the company SolarWinds in 2020 [122] – are an example of this phenomenon. This attack infiltrated computers in both the U.S. government and private sector, leading to the leakage of millions of classified and confidential documents to foreign adversaries.

The proliferation of cyberattacks is not simply limited to one specific application domain. The Internet-of-Things (IoT) revolution has led to the adoption of embedded systems in an ever-growing variety of devices [5]. However, these systems are dangerously prone to critical security vulnerabilities [21, 31, 72]. An attacker could feasibly extract personal information or confidential credentials by exploiting these bugs. Many such attacks have been deployed in the real world, leading to the proliferation of botnets [7] and power grid disruptions [145]. As manufacturers race to incorporate new devices and features into their IoT products – often without regard for the security and privacy of the end user – these types of attacks are poised to proliferate in the future.

Even worse, autonomous vehicles are also vulnerable to many types of common security vulnerabilities [51]. These types of bugs are even more serious, as errors in control software can endanger the lives of pedestrians, passengers, or other drivers. While automatic testing suites have been created to verify the robustness of these autonomous systems [71] in the physical world, they are still prone to traditional software-driven attacks, much like any other safety-critical computer system. Therefore, these vehicles must be secured against vulnerabilities and bugs.

The shared link between the aforementioned vulnerable systems is their usage of common open-source software – often written in C. These massive projects – like the Linux kernel [85], OpenSSL [155], and FFmpeg [44] – are frequently targeted by hackers. As a result, these large software projects contain a disproportionate amount of Common Vulnerabilities and Exposures (CVEs) [136]. A CVE informs the public about a known security vulnerability and serves as a centralized repository of information regarding the exploit [99]. However, a CVE entry does not facilitate the automated repair of vulnerabilities on its own.

Several existing methods have been created to patch vulnerable software with minimal human input [35, 47, 48, 55, 124, 193]. These techniques allow for the rectification of costly bugs with a comparatively lesser human cost. Yet, even these advanced systems are prone to mistakes. While these techniques often help rectify real-world bugs, automated vulnerability repair methods are known to generate invalid code [90] or introduce additional bugs [102]. More pressingly, these techniques often require analysis from a developer experienced with the program's codebase to fully implement [192]. This presents challenges when security vulnerabilities are identified in old, unmaintained code – or where the subject-matter experts for a particular product are no longer accessible. As a result, cybercriminals would be able to exploit these bugs, potentially leading to the leakage of sensitive user data. We demonstrate in Figure 3.1 how our pipeline can mitigate the risks caused by abandoned or poorly maintained legacy code.

As an increasing amount of software governs critical real-world systems, the importance of program maintenance has grown drastically. The proportion of engineers devoted to maintaining legacy code systems has risen significantly [57]. Even then, the average time-to-fix of software vulnerabilities is only increasing [6]. This presents a growing threat to end-users, especially when these bugs may take far longer to be patched in downstream code.

The advent of highly capable Large Language Models (LLMs) has the potential to transform how software vulnerabilities are rectified, especially in older codebases. However, it is known that LLMs often produce flawed, uncompilable code [89]. Even then, state-of-the-art LLMs such as GPT-4o [113] and Llama 3 [132] have spurred significant changes in software engineering practices. Moreover, specialized models tuned for code generation have appeared [135], further increasing the potential for automated software augmentation and creation. Techniques such as Parameter-Efficient Fine-Tuning (PEFT) [87] and Low-Rank Adaptations (LoRAs) [60] extend the capabilities of these models, leading to an increase in performance while simultaneously streamlining the model training process [86, 179]. More recently, models incorporating a Mixture-of-Experts (MoE) have enabled significant gains in LLM performance [68]. Researchers have also studied Prompt Engineering – a method of refining LLM input to measurably improve the relevancy, accuracy, and quality of responses [32, 149, 185]. These advances in Large Language Models have created a unique opportunity for combination with existing vulnerability repair techniques to automatically rectify common software bugs.

We introduce LLM4CVE, an iterative pipeline that integrates Large Language Models with already-existing CVE data to fix common classes of software vulnerabilities with minimal human input. Given a snippet of code identified as faulty, our pipeline iterates until a viable candidate is obtained. LLM4CVE begins generating a candidate fix, evaluating the viability of the fix, and applying required changes to increase the viability of the synthesized code. This iterative loop is further enhanced by the incorporation of LoRA fine-tuning into our pipeline. We also use prompt engineering to create a set of optimized prompts to enhance the quality of the generated code. Ultimately, LLM4CVE is capable of synthesizing a viable replacement code snippet, automating the vulnerability repair process for real-world examples of candidate CVEs. We summarize our key contributions as the following:

- We present a novel, automated method for fixing security vulnerabilities in real-world programs that require minimal intervention from a skilled domain expert. Moreover, our approach is capable of preserving application security even in codebases with few or no maintainers.

- To the best of our knowledge, we create the first *iterative* process for a Large Language Model to automatically correct vulnerabilities in code, improving on current automated vulnerability correction tools.

- We present a detailed study of the effectiveness of our iterative pipeline for fixing various classes of CVE/CWEs across multiple foundational models. Our methodology is tested on several mainstream LLMs – including GPT-3.5, GPT-4o, Llama 3 8B, and Llama 3 70B.

The remainder of this paper is organized as follows: in Section 3.2, we examine existing approaches to vulnerability repair – including both manual and automated solutions. A discussion of the benefits of LLM4CVE over the current State-of-the-Art works follows. Next, we provide the reader with a brief background on both Large Language Models and automated vulnerability repair in Section 3.3. We discuss the foundations of LLMs and the methods for fine-tuning them, as well as general methods of repairing software vulnerabilities without human intervention. A presentation of the LLM4CVE methodology follows in Section 3.4, where each pipeline stage is thoroughly detailed. Our experimental setup and results are then displayed in Sections 3.5 and 3.6. Finally, Sections 3.7, and 3.8 discuss the potential applications of our findings, known limitations and future improvements. The paper concludes with a public release of the LLM4CVE pipeline in Section 3.9.

3.2 Related Works

Our work draws upon two bodies of existing research work: (1) automated vulnerability repair, and (2) code generation with Large Language Models. We also compare our proposed pipeline to existing works on LLM-driven vulnerability repair.

3.2.1 Automated Vulnerability Repair

Interest in automatically rectifying software vulnerabilities has been consistently strong for decades. Several methods have been created over the years to assist in fixing software bugs [25, 47, 48, 50, 55, 90, 100, 102, 123, 124, 186, 192, 193, 194, 195]. Many of these methods rely on program analysis to extract code, such as with the help of an external compiler or static analysis tool chain [73]. Yet, static analysis cannot find certain types of software problems. For example, bugs in the Java Reflection API are difficult to detect using these traditional techniques [78].

Other advanced techniques have also been developed to automatically fix software bugs. For example, a method using Generative Adversarial Networks (GANs) is effective for vulnerability repair [55]. Importantly, this method allows for the repair of problematic code without the need for labeled training examples. However, unlike LLM4CVE, the model is evaluated on only synthetic code samples instead of real-world vulnerabilities.

Transfer Learning also been investigated for automated software repair. Researchers have demonstrated significant improvement over state-of-the-art methods, with the VRepair framework achieving almost a 50% increase in repair rate [25]. The use of the transformer architecture is also notable, even though the size of their model is significantly smaller than that of modern LLMs like GPT-4o [113].

Similarly, Vision Transformers are also capable of rectifying code vulnerabilities. By using special queries to locate vulnerable code snippets, the model can generate more accurate and relevant repair suggestions [47]. This innovative model not only performed better than previous state-of-the-art models, but also was reviewed positively by industry practitioners.

Code understanding models such as CodeT5 [166] have allowed for further improvements in the quality of generated fixes. As a precursor to modern Large Language Models such as GPT-4o and Llama 3, the CodeT5 architecture has enabled researchers to once again improve the total repair rate for software vulnerabilities [48]. The proposed framework – VulRepair – outperforms VRepair on several metrics due to extensive pre-training and the usage of Byte-Pair Encoding.

3.2.2 LLM-Driven Code Generation

The popularization of Large Language Models has catalyzed significant interest in their use in many disparate fields. Specifically, the success of automated code generation has been greatly accelerated by improvements in the logical reasoning ability of these models [121]. State-of-the-art models like GPT-4o [113] have revolutionized the landscape compared to their predecessors such as CodeBERT [43]. Moreover, further advances in code synthesis have led to the refinement of methods for generating correct, understandable code from these models.

In addition to general-purpose models such as the aforementioned GPT-4o, specialized LLMs for code synthesis have emerged, such as CodeX [23], Code Llama [135], WizardCoder [95], and CodeGen [107]. For example, many of these models, including Code Llama, CodeX, and WizardCoder are trained on publicly available software repositories, which enhances their code generation abilities. Specialized systems such as CodeGen employ a multi-point synthesis scheme, where the user is periodically prompted for feedback on the generated code. However, this method requires active human intervention, as opposed to the completely automated feedback loop provided by LLM4CVE.

As the code generation abilities of LLMs improve, advanced evaluation metrics are needed to assess the viability of automated code synthesis methods. While existing code automation tools have studied the reliability of commercial products such as Microsoft Copilot [160], we focus on the evaluation of the models themselves. Benchmarks such as EvalPlus [89] have been created to more accurately measure the performance of LLMs on code generation tasks. These metrics build upon existing works such as HumanEval [23] and are better suited to evaluating LLM-synthesized code. Other works have suggested using a litany of benchmarks to rank LLMs by performance [181].

Ultimately, LLM4CVE draws from these innovations in both code synthesis and code evaluation to further the robustness and viability of our pipeline.

3.2.3 Vulnerability Repair with LLMs

Recently, Large Language Models have been identified as a key component for automatically rectifying software vulnerabilities. This is motivated by the ability of modern LLMs to generate consistently viable code snippets for many common programming languages [181]. Even for programming languages with relatively little toolchain support, augmented LLMs perform remarkably well [39]. A notable integration of these two methods is the use of LLMs to provide code suggestions for fixing potential bugs in real-time to a programmer [82, 172]. However, this novel technique requires extensive human-computer collaboration, which is not true for the LLM4CVE pipeline.

Furthermore, a significant amount of literature exists regarding the use of LLMs for automated program repair [2, 66, 69, 70, 120, 144, 173, 176, 177, 178, 182, 198]. One of the first works on this subject involves using zero-shot prompting to fix security vulnerabilities in a synthetic dataset [120]. Since this work is from 2021, the LLMs used (Codex and Jurassic J-1) are relatively out-of-date. However, the performance of these techniques is already impressive, with a significant portion of simple, synthetic bugs fixed through the authors' pipeline.

Using the Codex and GPT-3 LLMs, researchers were able to repair 76.8% of bugs in Java programs detected with static analysis tools [69]. Notably, these bugs were often security-related, falling into categories such as Null Pointer Dereferences and Thread Safety Violations. However, this tool is limited to the C# and Java programming languages, whereas a majority of critical system software is written in languages such as C and C++ [58, 133].

There is an existing precedent for providing feedback for the LLM-driven code repair process. However, existing implementations require an extensive test suite [177], which is often not available for real-world software. When given representative test cases, this framework can repair the majority of bugs automatically.

More recently, researchers have been able to incorporate advanced LLM augmentation techniques such as Low-Rank Adaptations (LoRAs) to fine-tune their code repair models [144]. These methods are similar to our proposed LLM4CVE pipeline and they are known to work effectively, as evidenced by the double-digit improvement over the non-LoRA baseline. The datasets used in this work are not security-focused, and they instead cover a wide variety of bugs present in everyday software. In comparison, we fine-tune our models on a dataset of real-world security vulnerabilities, as demonstrated in Section 3.5.1.

3.3 Background

The LLM4CVE pipeline builds upon accepted wisdom in the field of Automated Vulnerability Repair, as well as incorporating state-of-the-art LLM augmentations – including Parameter-Efficient Fine-Tuning (PEFT) and Low-Rank Adaptations (LoRAs) – while also incorporating more traditional LLM techniques such as Prompt Engineering. In this section, we provide the reader with sufficient background knowledge for each domain, while also providing context for the design choices implemented in our pipeline.

3.3.1 CVEs & CWEs

The reporting of bugs to centralized online repositories has become increasingly common. Entities such as the National Vulnerability Database [104] and MITRE's Common Vulnerabilities and Exposures [99] provide up-to-date access to bug reports and mitigation measures.

Often, when a security vulnerability is discovered in widely-used software, a description is listed on one or more of these databases [20]. Since CVEs often target a specific software problem instead of describing a broad vulnerability class, the Common Weakness Enumeration (CWE) system was created to fulfill this purpose [59]. Ultimately, both the CVE and CWE identifiers are useful in our analysis for rectifying vulnerabilities.

3.3.2 Vulnerability Analysis & Repair

The common goal of every vulnerability repair technique is to in some way rectify underlying software bugs that would otherwise lead to undefined or unsafe behavior. While formal models defining types of software deficiencies – such as faults, errors and failures – exist [9], we focus on the most simple definition – the detection and repair of problems that result in unexpected output.

Automated vulnerability repair often targets a specific class of bugs [100]. This may be due to their heightened potential for exploitation, frequency in real-world code, or their ease of detection and subsequent patching, among other factors. These classes are commonly represented in CVEs and CWEs, as explained in Section 3.3.1.

Common characterizations of vulnerability repair break the process into three sections – (1) bug detection, (2) patching, and (3) patching [124]. The detection phase involves scanning source code using static analyzers [197], machine learning [19, 83], or other methods [146]. Here, potentially faulty sections of code are identified for correction. Next, the deficient code is rectified through a variety of algorithmic [100] and deep-learning [193] methods. Before the patch is complete, a verification stage must confirm the reliability and robustness of the fix. This is often done through validation with a test suite [90, 177], static analysis [187], or by expert verification [50].

These methods are similar to industry-standard best practices regarding the software vulnerability life cycle [118]. An overview of these practices and how LLM4CVE is positioned in this cycle is shown in Figure 3.2.

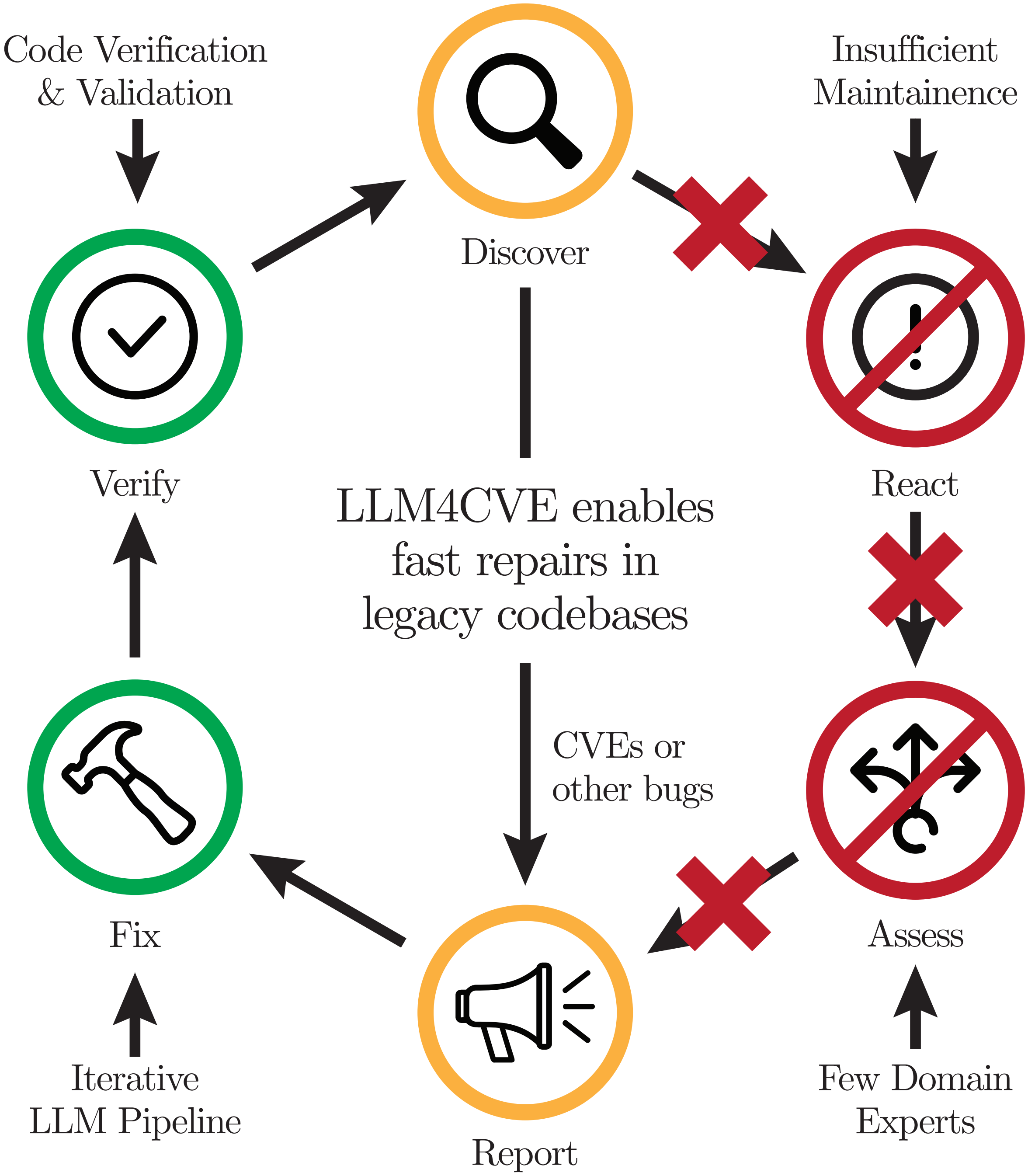
Code Verification & Validation
Discover
Insufficient Maintainence
Verify
LLM4CVE enables fast repairs in legacy codebases
React
CVEs or other bugs
Fix
Assess
Iterative LLM Pipeline
Report
Few Domain Experts

Figure 3.2: Rectification of software vulnerabilities often follows a predefined cycle

3.3.3 Large Language Models

Large Language Models are fundamentally driven by the attention mechanism [161], which allows for the comprehension of lengthy inputs with ease. Moreover, these models can capture the syntax and structure of their input, allowing for the comprehension of dependencies between tokens that are spatially distant in the input sequence, but semantically linked [141, 190]. These methods are further extended by methods such as Retrieval-Augmented Generation, which references an external knowledge base at generation-time to better provide factually correct information [80].

The rise of LLMs has been fueled by the success of OpenAI's GPT-2, GPT-3, and most recently GPT-4 and GPT-4o [113] family of highly-performant and extensible models. As these models advance, their parameter counts have grown exponentially. While GPT-3 stood at a "modest" 175 billion parameters [15], it is theorized that GPT-4 has ballooned to almost ten times its predecessor – 1.76 trillion parameters [97]. The expansion in model size has not only increased performance – it has been hypothesized that previously unknown abilities emerge from this scaling [168], although the impact of this phenomenon is still not fully understood [139].

Yet, many leading LLMs – including those from OpenAI – are not open-source, creating difficulties for fine-tuning and opening a proverbial "can of worms" regarding data privacy. Although OpenAI provides a fine-tuning API, it is licensed restrictively and monetarily expensive [112]. As a result, several open-source LLMs been publicized, including Llama 3 [132], Code Llama [135], and Mixtral (MoE) [68].

LLM Augmentation

While LLMs can provide excellent code synthesis on their own, several techniques have been developed to further improve their abilities on a diverse range of tasks. We focus on two methods – (1) Parameter-Efficient Fine-Tuning/Low-Rank Adaptation (PEFT/LoRA), and (2) Mixture-of-Experts.

The increasingly large size of modern LLMs leaves them requiring tremendous amounts of computational resources to train. Parameter-Efficient Fine-Tuning (PEFT) helps alleviate this issue by reducing the complexity of fine-tuning an LLM to a specific task [49, 88, 191]. A popular approach to PEFT is the Low-Rank Adaptation (LoRA), which freezes almost all model parameters and performs fine tuning using injected rank decomposition matrices [60]. Further performance improvements have been achieved through quantization [34], and applying LoRAs to a Mixture-of-Experts model [42]. A description of the LoRA training process is given in Figure 3.3.

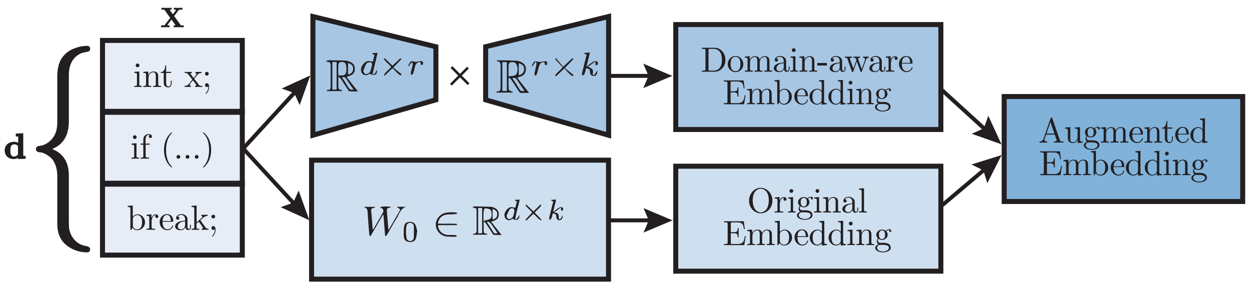

Figure 3.3: LoRAs enable the fine-tuning of LLMs with a comparatively low computational cost

Another approach to improving LLM performance is through the Mixture-of-Experts (MoE) paradigm. This methodology involves training multiple smaller "expert" models – specialized for a subset of the input domain – and using a router to select the optimal submodels to evaluate a query at inference time [199]. It has already been demonstrated that these types of LLMs scale better [36] and perform better than a compute-equivalent single-expert model [8].

Prompt Engineering

The quality of a prompt provided to an LLM directly influences the quality of the output. Refining the initial prompt to the model, known as *Prompt Engineering*, has been shown to improve performance on basic logic problems [32], object annotation [142], general reasoning tasks [175], and even the generation of prompts themselves [185]. Common Prompt Engineering techniques include Chain-of-Thought reasoning, where a complicated process is broken into individual steps for an LLM to process individually [170]. We use a similar method of few-shot prompting in the LLM4CVE pipeline, driven by automated compiler and metric-based feedback. LLMs are also capable of zero-shot reasoning simply by prompting the model to think in a step-by-step fashion [75].

CWE	Title	Count
CWE-125	Out-of-bounds Read	452
CWE-119	Improper Restriction of Operations within the Bounds of a Memory Buffer	363
CWE-20	Improper Input Validation	289
CWE-787	Out-of-bounds Write	179
CWE-476	NULL Pointer Dereference	176
CWE-190	Integer Overflow or Wraparound	156
CWE-120	Buffer Copy without Checking Size of Input ('Classic Buffer Overflow')	121
CWE-416	Use After Free	120

Table 3.1: An overview of selected CWEs for the LLM4CVE pipeline

3.4 Methodology

LLM4CVE is an iterative pipeline that intends to automatically rectify common software vulnerabilities through the use of augmented Large Language Models. In this section, we aim to describe the structural and theoretical motivations behind the implementation of the pipeline. A visualization of the LLM4CVE pipeline is given in Figure 3.4.

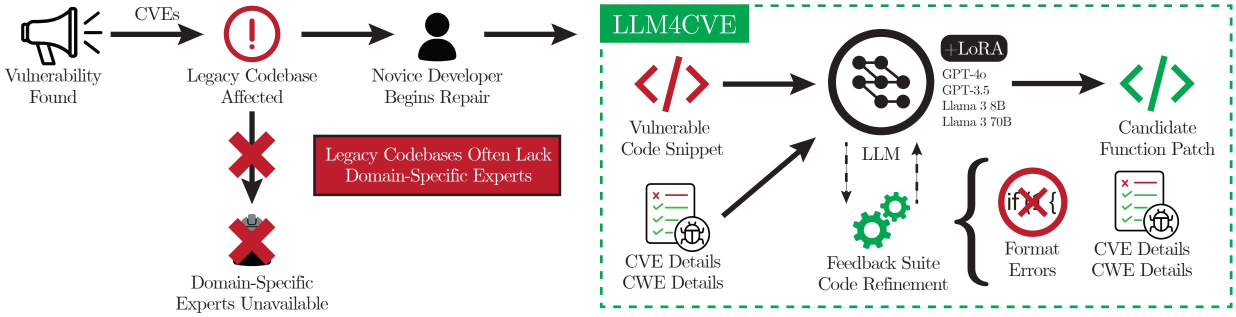

Figure 3.4: A visualization of how the LLM4CVE pipeline can automatically fix common software vulnerabilities

3.4.1 CVE Selection

We select eight of the most common CWEs for our analysis. A brief description of these CWEs along with their relative frequency in our dataset is provided in Table 3.1. Details on the filtration of vulnerability examples for our pipeline are provided in Section 3.5.2.

3.4.2 Prompting

We employ both *zero-shot* and *few-shot* prompting for generating code snippets from an LLM. In *few-shot* prompting, guidance is provided to the model over multiple rounds of code iteration, and the LLM is then able to incorporate feedback into the final code. On the other hand, in *zero-shot* prompting, no contextual guidance is provided to the model. Therefore, the LLM generates a code snippet without guidance or feedback. All LLMs in our study are tested with both types of prompting.

3.4.3 Prompt Engineering

The LLM4CVE pipeline employs two types of prompts, classified as 'guided' and 'unguided', respectively. The 'guided' prompt provides the name of the CVE and CWE, a description of both, and specific instructions to further facilitate valid repair. The 'unguided' prompt only instructs the model to repair the code – no vulnerability details are provided. We also require the LLM to output code surrounded by delimiters to further optimize the extraction of code from the model output. Moreover, we instruct the model to create compilable code, use proper syntax and make the minimal amount of changes required to fix the target bug, as well as requiring it to synthesize all modifications itself without asking for user input.

Importantly, while CVE descriptions often provide a specific reference to a problem component in a function, CWE descriptions instead categorize the issue into a broad set of vulnerabilities. As a result, the "guided" prompt offers a more thorough description of the problem, which we posit allows for the LLM to generate superior candidate patches.

3.4.4 Code Analysis

To enable the automated evaluation of the LLM code output, an automated metric is required. For this purpose, we choose CodeBLEU, a widely accepted method for calculating the semantic similarity of two pieces of code [131]. Like its predecessor BLEU [117], which is often used for determining the quality of machine-translated text, CodeBLEU allows us to determine how similar the ground truth fixed code is to the LLM-generated fix. This tool provides scores in the range of 0-100 (we use an implementation that scales these values to 0-1), with a higher score implying the candidate code snippet is similar to the reference snippet. CodeBLEU incorporates the n-gram match from BLEU, in addition to in-depth analysis of code semantics via Dataflow Graphs and Abstract Syntax Trees.

3.4.5 Iterated LLM Generation

One of the most important parts of the LLM4CVE pipeline is the automated feedback loop between analysis tools and the LLM. This allows for the LLM to generate improved code over multiple iterations, greatly increasing the quality of the final output. To implement this mechanism, we have used the difference in CodeBLEU scores between the broken input code and the LLM-generated output code.

The feedback provided conforms to two guiding principles – (1) the output code should not be dramatically different than the input code, and (2) the output code should be valid C code. Our use of CodeBLEU scoring helps the model achieve both goals, as we can safeguard against excessive semantic changes to the code. Any faults found will automatically trigger another iteration of our pipeline. Then, if any new faults are found in the generated code (perhaps due to a failure of the model to synthesize a valid snippet), this process will be repeated. Ultimately, we collect output from both of these stages, along with the previously generated code and feed it into the LLM in the same context session for each iteration. We impose a limit of two iterations on the pipeline and select the second output as the candidate patch for evaluation.

3.4.6 Evaluation Process

Unlike many other approaches to automated vulnerability repair with Large Language Models, LLM4CVE targets real-world bugs instead of synthetic vulnerabilities. As a result, the evaluation of code correctness is a significantly more complicated problem. Many vulnerable code snippets are not self-contained, which means that external context is required to improve generation performance. We tailor of evaluation suite with this fact in mind, focusing on metrics highlighting improvements in code quality, and comparing the semantic similarity of our candidate patches to real-world ground truth solutions. In addition, we perform a selection of end-to-end compilation steps using our generated patches to confirm the viability of our pipeline. We believe this methodology offers a balance between theoretical evaluation and real-world applicability.

Importantly, we note that the nature of extracting function-level vulnerabilities from real-world codebases implies that these snippets are not fully self-contained. Therefore, it is infeasible to perform traditional static analysis, which necessitates our use of alternative techniques to generate feedback for the iterate steps of our pipeline.

There are multiple valid methods in which to fix a bug. For critical code, it is often more important for a vulnerability to be fixed immediately – even if functionality is impacted. Therefore, a priority in our evaluation scheme is measuring the ability of our pipeline to fix the provided vulnerability. There are scenarios where the ground truth patch differs from our candidate patch, but we are concerned foremost with whether or not the bug is resolved. Then, both patches in this scenario would pass our evaluation method.

3.5 Experimental Setup

In this section, we discuss the design of our experimental apparatus and provide details on the preprocessing, testing, and evaluation schemes of our work. It is important to note that we use multiple compute nodes equipped with one Nvidia A100, 48 CPU cores, and 256GB of system memory throughout our study.

3.5.1 Datasets

Our primary dataset of interest is *CVEFixes* – a repository containing metadata, commit history, CVE/CWE classification, and most importantly: a before-and-after representation of vulnerable code [13]. This dataset contains over 10,000 vulnerable functions, a majority of which have labeled pairs of vulnerable (before) and non-vulnerable (after) code snippets. We use the provided SQL database to extract these labeled pairs, and we further filter them by language. We target the C programming language for the extraction of vulnerable code snippets.

3.5.2 Dataset Preparation

As provided, the CVEFixes dataset does not lend itself to easy extraction of candidate before-and-after pairs corresponding to each CVE. Therefore, we implement a preprocessing pipeline to extract this information from the dataset. First, we obtain all function-level changes for the target programming languages ordered by CVE and function name, excluding CVEs with an associated CWE with minimal information such as “NVD-CWE-noinfo” and “NVD-CWE-other”. We then filter out all CVE+name pairs that do not match our desired pattern of one vulnerable “before” code snippet, and one non-vulnerable “after” code snippet.

After obtaining a feasible set of before-and-after code snippets, we further trim these candidates by removing all pairs where at least one of the code snippets has a token count greater than 500. This ensures that we are not at risk of exceeding the context length of the Large Language Models used in our testing. After our dataset filtration stage, we are left with eight CWEs with at least 100 candidate pairs, representing 697 unique CVEs.

3.5.3 Large Language Models

For this study, we target the following Large Language Models: GPT-3.5, GPT-4o, Llama 3 8B, and Llama 3 70B. Notably, these models represent state-of-the-art performance in the categories of closed-source and open-source models. Importantly, the open-source nature of the Llama 3 family of LLMs enables the training of LoRAs to boost model performance, as explained in Section 3.5.4. The range in parameter counts also offers an opportunity to explore the performance gradient between the selected models. It is also important to note that GPT-4o is a multimodal LLM, although our pipeline uses only text-based input.

Our selection of LLMs is also motivated by the maximum context length supported by each model. A longer context length enables the LLM to incorporate more information during the generation process, which is especially important during iterative generation, as each iteration builds on top of the existing context. Therefore, a sufficiently large context window is required for our pipeline to function, which is provided by all tested LLMs. This value was derived from the OpenAI documentation for GPT models [111] and the implementation specifications for the Llama 3 [132] models. Note that the GPT-3.5-Turbo and GPT-4o models available from OpenAI represent the GPT-3.5 and GPT-4o models in our study. A table describing the context length for each model is provided in Table 3.2.

Model	Context Length (Tokens)
GPT-3.5	16,385
GPT-4o	128,000
Llama 3 8B	8,192
Llama 3 70B	8,192

Table 3.2: Context lengths of selected Large Language Models

3.5.4 LLM Augmentation

We train a Low-Rank Adaptation on the Llama 3 70B LLM using a portion of our created dataset. We employ an 90/10 train/test split to ensure sufficient data is available for our evaluation. We train on labeled “broken”/“fixed” pairs, corresponding to the pre-fix and post-fix ground truth data. Then, we evaluate the LLM+LoRA on the test set by requesting for the set of broken code samples to be rectified. A complete description of our evaluation metrics is provided in Section 3.5.8.

3.5.5 Pipeline Configurations

We use three pipeline configurations – (1) “unguided”, (2) “guided”, and (3) “guided+feedback”. The third configuration also includes the trained LoRA for the Llama 3 80B model. An explanation of the prompting scheme and the feedback mechanism is shown in Section 3.4.3 and Section 3.4.5, respectively. Across our five tested models, this results in 15 potential model/configuration combinations. Note that we use a random sample consisting of 50% of the full dataset for the “guided+feedback” configuration. We provide a full model/configuration diagram in Table 3.3.

Model	“unguided”	“guided”	“guided+feedback”
GPT-3.5	✓	✓	✓
GPT-4o	✓	✓	✓
Llama 3 8B	✓	✓	✓
Llama 3 70B	✓	✓	✓

Table 3.3: Models & pipeline configurations used in our study

3.5.6 Automated Pipeline Feedback

The LLM4CVE pipeline implements a “guided with feedback” configuration which employs both Prompt Engineering and iterative generation to synthesize higher-quality candidate vulnerability patches. We restrict the pipeline to two iterations as a means to balance performance and computing power/throughput.

One important consideration is that our iterative process must not exceed the context window of the LLM, as to prevent catastrophic forgetting. We employ two factors to mitigate this concern. Each prompt and vulnerable code snippet is roughly 500 tokens long, ensuring that even after incorporating the token length of the output as well, we can stay within the model’s context limit. It is important to consider that our feedback prompts are approximately the same size as the initial prompt, as we include the in-progress code at each iteration step.

Once we obtain an output from the LLM, if we have not reached our iteration limit as defined in Section 3.4.5, we extract the code from the response and submit it for testing. We use the change in CodeBLEU score between this code and the previous round's code, as explained in Section 3.4.4. Using these metrics, we identify issues in the candidate patch to include in the prompt for the next iteration. For example, if the CodeBLEU score has diverged significantly, it is likely an erroneous condition has been encountered, and so we inform the LLM that the candidate patch may be incorrect. Importantly, we never compare the CodeBLEU score of the candidate patch and the ground truth in this step, as our pipeline would not have the ground truth fix during real-world use.

A complete description of our pipeline's iterative generation step is provided in Figure 3.5.

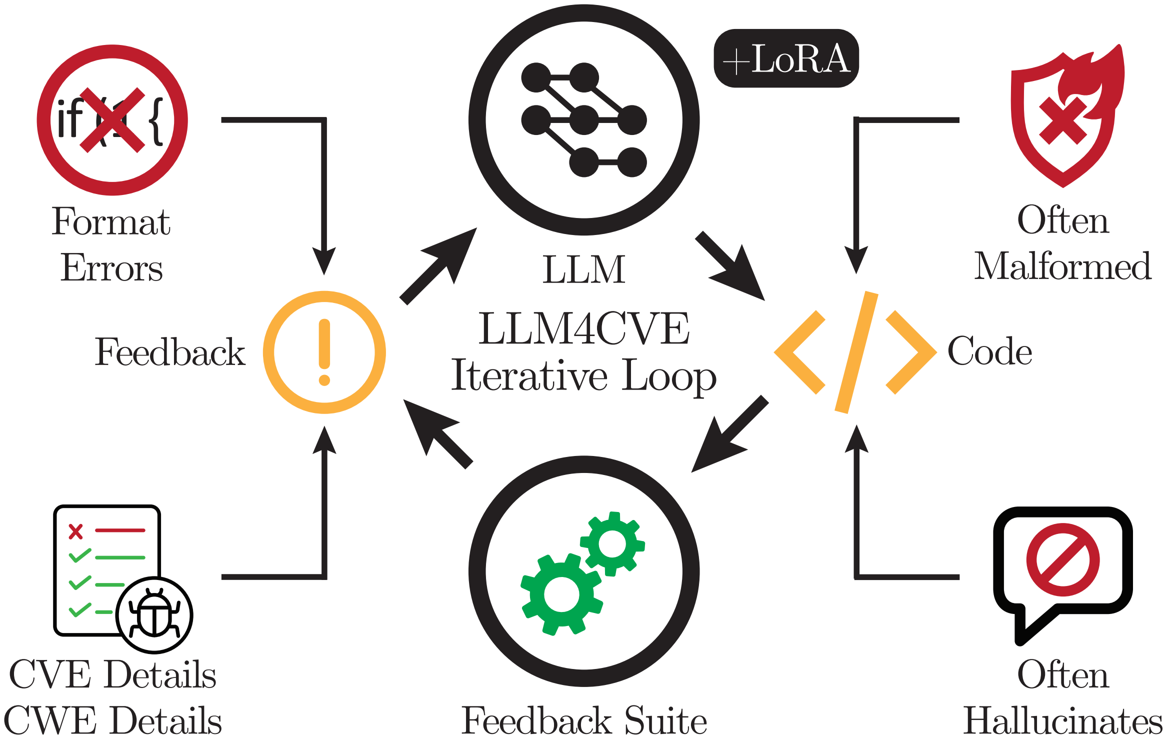

Figure 3.5: LLM4CVE uses iterative generation to improve the overall quality of patch synthesis

3.5.7 Candidate Patch Extraction

After obtaining an output from the LLM, we must extract only the code from it, discarding any delimiters or extraneous commentary. From our analysis, approximately 95% of responses are well-formed, following the code block formatting specified in the prompt. For these responses, we simply extract the candidate patch from the code block. The remaining responses require nontrivial logic to extract patches from, which we implement as well. In the case where we detect no code has been generated (<1% of samples), we output no candidate patch instead.

3.5.8 Metrics

We use three principal metrics for evaluation – CodeBLEU scores, end-to-end compilation, and required engineering effort. We employ a pass @ k scheme, with $k = 1$, as described in [23]. Detailed descriptions of our chosen metrics are provided below.

CodeBLEU Scores

We evaluate the final output code from all configurations and model types against the ground truth non-vulnerable code snippet. As a higher CodeBLEU score implies greater semantic similarity between two pieces of code, a large (i.e. near 1.00) score between the ground truth and the output code implies greater potential for valid bug correction. Importantly, a candidate patch with a CodeBLEU score less than 1.0 can still be a viable fix, as there are often multiple solutions to the given vulnerability.

End-to-End Compilation

Next, we test our pipeline's generated patches in real-world codebases. We directly apply the result of our pipeline to the codebase affected by the vulnerability. Then, we compile the entire project and determine the validity of the fix. This metric measures the ability for the LLM4CVE pipeline to fix real-world security vulnerabilities. It also ensures that the pipeline output is compliant , compiliable code.

Engineering Effort

Finally, we analyze the engineering effort required between traditional approaches and our pipeline. We compare setup times, the level of experience required, and the technical complexity of the proposed technique.

3.6 Results

In this section, we present the results of our evaluation of the LLM4CVE pipeline. We measure three key metrics – CodeBLEU Scores, End-to-End Compilation Success Rate, and required Engineering Effort. These metrics enable a multifaceted assessment of the practicality, functionality, and effectiveness of the LLM4CVE pipeline. We provide visual comparisons between the various configuration of our pipeline over the five Large Language Models described in Section 3.5.3. Moreover, we compare pipeline results between all three configurations and eight CVEs used.

3.6.1 CodeBLEU Scores

For this metric, we compare the CodeBLEU scores between the pipeline output and the ground truth fix. Our CodeBLEU software tool generates semantic similarity ratings in the range of 0.0-1.0. A score of 1.0 implies an exact match when considering n-grams, syntax, and dataflow between the two samples. An important consideration to keep in mind is that there may be multiple 'valid' fixes for a CVE, so an inexact match is not necessarily indicative of invalid code. Therefore, we treat this metric as a probabilistic estimate of the likelihood of generating a viable candidate fix. We extend this evaluation with real-world evaluation of selected candidate patches in Section 3.6.2.

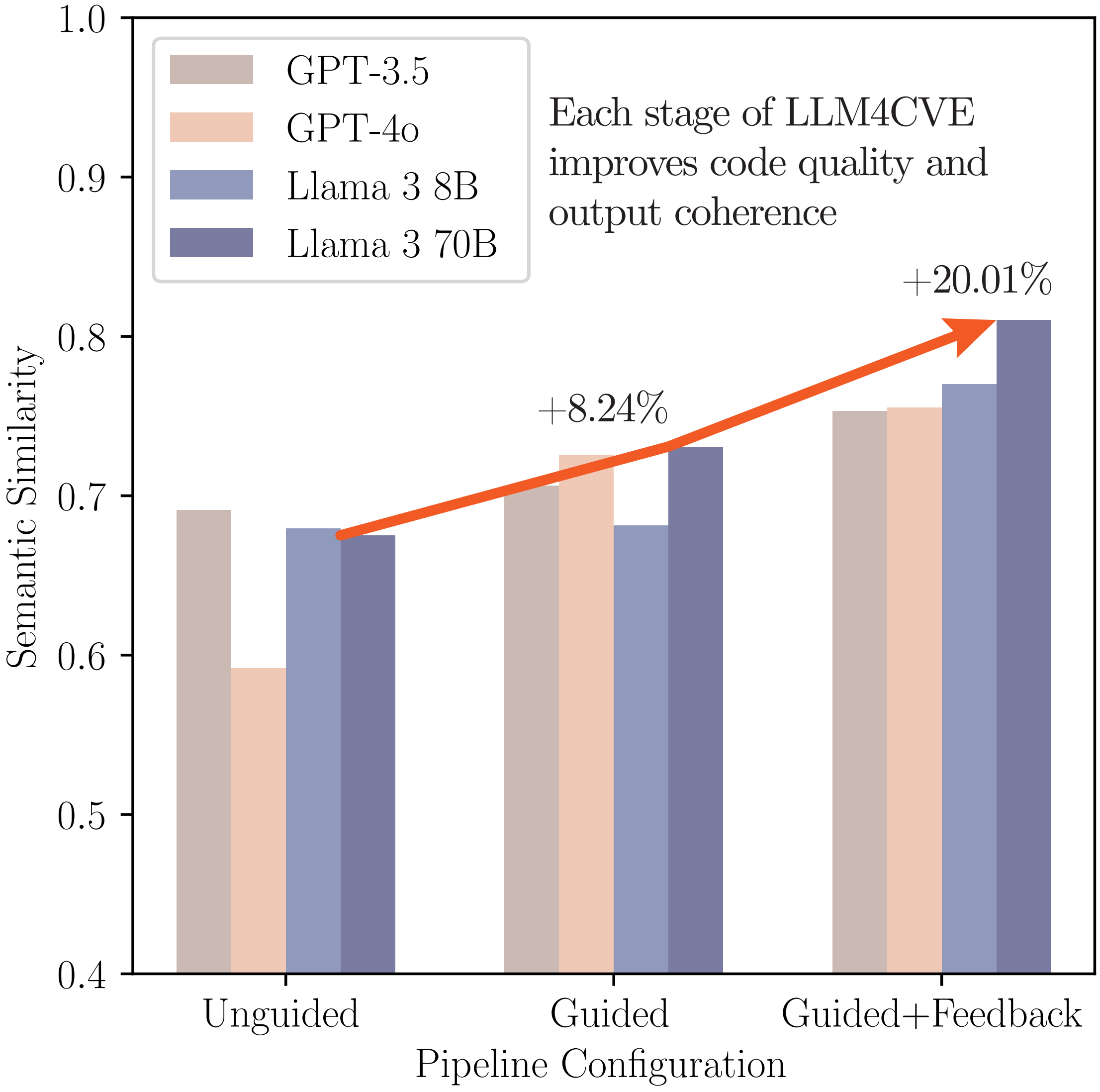

Figure 3.6: Semantic similarity scores across all pipeline configurations and model types

We include results for all three pipeline configurations – "unguided" (zero-shot), "guided" (one-shot), and "guided+feedback" (few-shot) – and the semantic similarity scores are presented in Figure 3.6. Importantly, the full configuration of the LLM4CVE pipeline – "guided+feedback" (few-shot) – demonstrates a remarkable performance improvement across all models, with the Llama 3 80B LLM peaking at a 20% increase in semantic similarity scores.

3.6.2 End-to-End Compilation

For this test, we focus on evaluating the pipeline's effectiveness in fixing real-world bugs. We begin by initializing our pipeline with a vulnerable function and associated prompt. As an example, the `cJSON_DeleteItemFromArray` function from the `iperf3` tool is chosen. This tool is a TCP, UDP, and SCTP network bandwidth measurement tool, and uses the library `cJSON`, in which this vulnerability lies. The software is affected by CVE-2016-4303 (CWE-120).

Once a candidate patch is obtained, we move to evaluation of its viability. We create a testing harness that exploits the vulnerability at hands and compile the program with and without the candidate patch. Our harness poses as a malicious actor trying to crash the software by providing a malformed `cJSON` object with invalid lengths. Then, we run both programs and note which program crashes. Through this evaluation, it was confirmed that the candidate patch prevented the malicious actor from exploiting this vulnerability by rejecting the malformed object, demonstrating the effectiveness of the LLM4CVE pipeline. We provide the details of our testing harness in our public release of the pipeline, which can be found in Section 3.9.

3.6.3 Engineering Effort

Every potential software solution must be evaluated on the basis of its efficiency and real-world practicality. As a result, we compare the ease of use and complexity of traditional human repair, GPT-based solutions, and open-source LLMs. We estimate the human cost to repair security bugs from various compilations of data published on this subject. Other statistics are measured directly from our pipeline.

The setup of GPT models is relatively efficient, as OpenAI provides a simple API for inference. Moreover, the user does not need access to compute resources, unlike many open-source LLMs. On the other hand, open-source models provide data security, as the entire training and inference process can be run on-site. Open-source LLMs require significantly more setup time, as the user must manually configure their pipeline before obtaining valid output. Moreover, several factors may influence the speed of generation, such as the user's access to GPU compute resources.

Trained engineers are the slowest of all three methods, as skilled human labor is required to fix security vulnerabilities. While our pipeline does not require domain-specific experts to function, manual code intervention often does. In legacy codebases, this problem is exacerbated, as the requisite skilled labor is often inaccessible or simply missing. Several studies have estimated the average time taken for human engineers to repair a security vulnerability to be between several days [151], and one month [13]. Note that we combine the setup and execution time into one statistic for this category.

The time values provided for the LLMs are based on the average time taken for members of our team to set up a basic version of the GPT/Llama pipeline and output a single candidate patch. We note that the LoRA training process requires access to a relevant dataset of vulnerable functions (like *CVEFixes*), and we incorporate an estimate of the training time into our evaluation. A comparison of the average time cost between these three approaches is given in Table 3.4.

Technique	Setup Time	Execution Time
Human Intervention	×	28 days [13]
Open-Source LoRA+LLMs	24 hours	10 minutes
GPT LLMs	1 hour	**5 minutes**

Table 3.4: Comparison of engineer-hours required for selected vulnerability patching techniques

3.7 Discussion

Our proposed pipeline lowers the barrier to entry for repairing critical security vulnerabilities, especially in legacy codebases. Significantly less engineering effort is required in these types of projects with deteriorating knowledge bases and few active maintainers. Moreover, the speed and efficiency of our pipeline enable these fixes to be made with haste, lessening the time between the discovery of a bug and its associated patch.

3.7.1 Impact

The LLM4CVE pipeline enables the quick repair of vulnerabilities in critical system software. Moreover, our tool is of increased relevance to legacy codebases, where devoted maintainers are often in short supply, and turnaround times are often long. By lessening the dependence on these domain-specific experts, our pipeline enables critical security vulnerabilities to be patched faster, increasing the overall safety of these programs.

These benefits are further enhanced by the speed of our pipeline, as demonstrated in Table 3.4. Applying patches in time-sensitive environments – such as when a new vulnerability is discovered – is desirable to maintain system integrity and robustness.

3.7.2 Candidate Patch Assessment

It is of interest to compare a patch generated by the LLM4CVE pipeline to the original vulnerable function. We provide a candidate patch for the `snmp_ber_decode_type` function affected by CWE-125 and CVE-2020-12141. Below, Listing 3.1 is the original vulnerable code, while Listing 3.2 is generated by GPT-4o using our fully-featured pipeline.

```c
unsigned char * snmp_ber_decode_type(unsigned char *buff, uint32_t *buff_len, uint8_t *type)
{
  if(*buff_len == 0) {
    return NULL;
  }

  *type = *buff++;
  (*buff_len)--;

  return buff;
}
```

Listing 3.1: The original vulnerable function

```c
unsigned char* snmp_ber_decode_type(unsigned char *buff, uint32_t *buff_len, uint8_t *type) {
  if(buff == NULL || buff_len == NULL || type == NULL || *buff_len == 0) {
    return NULL;
  }

  *type = *buff++;
  (*buff_len)--;

  return buff;
}
```

Listing 3.2: A candidate patch generated by GPT-4o

We see that the LLM output checks the validity of the input variables `buff`, `buff_len`, and `type`, while the vulnerable code only attempts to validate `buff_len` (and that too with a blind dereference). Then, it is evident that our pipeline was able to patch the vulnerability in a viable, non-destructive manner.

3.7.3 GPT v. Llama

Throughout our experiments, the Llama 3 70B model consistently matched or outperformed other LLMs, especially using the full LLM4CVE pipeline. Even the relatively smaller Llama 3 8B model can compete with the GPT models once fine-tuning is performed. This demonstrates the effectiveness of Parameter-Efficient Fine-Tuning techniques like LoRAs, as without these adapters the GPT and Llama models perform roughly equivalently in the "guided" (one-shot) pipeline configuration. The largest gains in generation ability were derived from the few-shot iterative configuration of our pipeline, and this can be attributed to our fine-tuning of open-source models. Some LLMs such as GPT-4o are not fine-tunable, and so we are unable to apply all techniques mentioned in Section 3.4 to these models.

3.7.4 Ethical Considerations

Our tool serves to rectify vulnerabilities in codebases where the maintainers would otherwise be unable to do so themselves. As a result, we expect that: (1) public usage of our tool will serve the security community by keeping the end-user better protected from cybercriminals, and (2) there are no significant ethical risks posed by our pipeline.

In addition, all vulnerabilities studied were already publicly disclosed by nature of being a known CVE. Therefore, no further vulnerability disclosure is necessary.

3.8 Conclusion

The LLM4CVE pipeline serves to fix security vulnerabilities in critical system software with minimal human input. By combining traditional bug repair methods with state-of-the-art Large Language Model techniques, we improve the robustness and viability of automated program repair. Our iterative pipeline allows for gradual refinement of the generated code, which increases the likelihood of obtaining a viable candidate patch. We further extend LLM4CVE with automated code analysis tools, LLM fine-tuning, and Prompt Engineering. A thorough evaluation of real-world vulnerabilities through both automated and human-centered means has shown the efficacy of our approach, and we believe our contributions to the field will pave the way towards achieving automated program repair without any intervention from trained experts.

3.9 Code Availability

We publish our testing apparatus, fine-tuned weights, and experimental data on our website.[2]

[2] https://sites.google.com/view/llm4cve

Bibliography

[1] [n. d.]. . Technical Report. W3C.

[2] Toufique Ahmed and Premkumar Devanbu. 2023. Better Patching Using LLM Prompting, via Self-Consistency. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 1742–1746. https://doi.org/10.1109/ASE56229.2023.00065

[3] T. Ahmed, N. Ledesma, and P. Devanbu. 2023. SynShine: Improved Fixing of Syntax Errors. *IEEE Transactions on Software Engineering* 49, 04 (apr 2023), 2169–2181. https://doi.org/10.1109/TSE.2022.3212635

[4] Mohammad Abdullah Al Faruque, Sujit Rokka Chhetri, Arquimedes Canedo, and Jiang Wan. 2016. Acoustic Side-Channel Attacks on Additive Manufacturing Systems. In *2016 ACM/IEEE 7th International Conference on Cyber-Physical Systems (ICCPS)*. 1–10. https://doi.org/10.1109/ICCPS.2016.7479068

[5] Shadi Al-Sarawi, Mohammed Anbar, Rosni Abdullah, and Ahmad B. Al Hawari. 2020. Internet of Things Market Analysis Forecasts, 2020–2030. In *2020 Fourth World Conference on Smart Trends in Systems, Security and Sustainability (WorldS4)*. 449–453. https://doi.org/10.1109/WorldS450073.2020.9210375

[6] Nikolaos Alexopoulos, Manuel Brack, Jan Philipp Wagner, Tim Grube, and Max Mühlhäuser. 2022. How Long Do Vulnerabilities Live in the Code? A Large-Scale Empirical Measurement Study on FOSS Vulnerability Lifetimes. In *31st USENIX Security Symposium (USENIX Security 22)*. USENIX Association, Boston, MA, 359–376. https://www.usenix.org/conference/usenixsecurity22/presentation/alexopoulos

[7] Manos Antonakakis, Tim April, Michael Bailey, Matt Bernhard, Elie Bursztein, Jaime Cochran, Zakir Durumeric, J. Alex Halderman, Luca Invernizzi, Michalis Kallitsis, Deepak Kumar, Chaz Lever, Zane Ma, Joshua Mason, Damian Menscher, Chad Seaman, Nick Sullivan, Kurt Thomas, and Yi Zhou. 2017. Understanding the Mirai Botnet. In *26th USENIX Security Symposium (USENIX Security 17)*. USENIX Association, Vancouver, BC, 1093–1110. https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/antonakakis

[8] Mikel Artetxe, Shruti Bhosale, Naman Goyal, Todor Mihaylov, Myle Ott, Sam Shleifer, Xi Victoria Lin, Jingfei Du, Srinivasan Iyer, Ramakanth Pasunuru, et al. 2021. Efficient large scale language modeling with mixtures of experts. *arXiv preprint arXiv:2112.10684* (2021).

[9] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr. 2004. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing* 1, 1 (2004), 11–33. https://doi.org/10.1109/TDSC.2004.2

[10] Zhongjie Ba, Tianhang Zheng, Xinyu Zhang, Zhan Qin, Baochun Li, Xue Liu, and Kui Ren. 2020. Learning-based Practical Smartphone Eavesdropping with Built-in Accelerometer. *Proceedings 2020 Network and Distributed System Security Symposium* (2020). https://iqua.ece.toronto.edu/papers/tzheng-ndss20.pdf

[11] Gray Bachelor, Eugenio Brusa, Davide Ferretto, and Andreas Mitschke. 2020. Model-Based Design of Complex Aeronautical Systems Through Digital Twin and Thread Concepts. *IEEE Systems Journal* 14, 2 (2020), 1568–1579. https://doi.org/10.1109/JSYST.2019.2925627

[12] Anomadarshi Barua and Mohammad Abdullah Al Faruque. 2020. Hall Spoofing: A Non-Invasive DoS Attack on Grid-Tied Solar Inverter. In *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, 1273–1290. https://www.usenix.org/conference/usenixsecurity20/presentation/barua

[13] Guru Bhandari, Amara Naseer, and Leon Moonen. 2021. CVEfixes: automated collection of vulnerabilities and their fixes from open-source software. In *Proceedings of the 17th International Conference on Predictive Models and Data Analytics in Software Engineering* (Athens, Greece) *(PROMISE 2021)*. Association for Computing Machinery, New York, NY, USA, 30–39. https://doi.org/10.1145/3475960.3475985

[14] Jan Olaf Blech and Sidi Ould Biha. 2013. On Formal Reasoning on the Semantics of PLC using Coq. (2013). arXiv:arXiv:1301.3047

[15] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *Advances in neural information processing systems* 33 (2020), 1877–1901.

[16] Xiaoye Cai, Ruochen Shi, Alexander Kümpell, and Dirk Müller. 2022. Automated generation of PLC code for implementing mode-based control algorithms in buildings. In *2022 30th Mediterranean Conference on Control and Automation (MED)*. 1087–1092. https://doi.org/10.1109/MED54222.2022.9837182

[17] Arquimedes Canedo and Mohammad Abdulah Al-Faruque. 2012. Towards parallel execution of IEC 61131 industrial cyber-physical systems applications. In *2012 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. 554–557. https://doi.org/10.1109/DATE.2012.6176530

[18] Roberto Cavada, Alessandro Cimatti, Michele Dorigatti, Alberto Griggio, Alessandro Mariotti, Andrea Micheli, Sergio Mover, Marco Roveri, and Stefano Tonetta. 2014. The nuXmv symbolic model checker. In *Computer Aided Verification: 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings 26*. Springer, 334–342.

[19] Saikat Chakraborty, Rahul Krishna, Yangruibo Ding, and Baishakhi Ray. 2021. Deep learning based vulnerability detection: Are we there yet? *IEEE Transactions on Software Engineering* 48, 9 (2021), 3280–3296.

[20] Yung-Yu Chang, Pavol Zavarsky, Ron Ruhl, and Dale Lindskog. 2011. Trend Analysis of the CVE for Software Vulnerability Management. In *2011 IEEE Third International Conference on Privacy, Security, Risk and Trust and 2011 IEEE Third International Conference on Social Computing*. 1290–1293. https://doi.org/10.1109/PASSAT/SocialCom.2011.184

[21] Libo Chen, Yanhao Wang, Quanpu Cai, Yunfan Zhan, Hong Hu, Jiaqi Linghu, Qinsheng Hou, Chao Zhang, Haixin Duan, and Zhi Xue. 2021. Sharing More and Checking Less: Leveraging Common Input Keywords to Detect Bugs in Embedded Systems. In *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, 303–319. https://www.usenix.org/conference/usenixsecurity21/presentation/chen-libo

[22] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code.(2021). *arXiv preprint arXiv:2107.03374* (2021).

[23] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374* (2021).

[24] Wei-Han Chen and Kannan Srinivasan. 2022. Acoustic Eavesdropping from Passive Vibrations via mmWave Signals. In *GLOBECOM 2022 - 2022 IEEE Global Communications Conference*. 4051–4056. https://doi.org/10.1109/GLOBECOM48099.2022.10001108

[25] Zimin Chen, Steve Kommrusch, and Martin Monperrus. 2022. Neural transfer learning for repairing security vulnerabilities in c code. *IEEE Transactions on Software Engineering* 49, 1 (2022), 147–165.

[26] Sujit Rokka Chhetri, Anomadarshi Barua, Sina Faezi, Francesco Regazzoni, Arquimedes Canedo, and Mohammad Abdullah Al Faruque. 2021. Tool of Spies: Leaking your IP by Altering the 3D Printer Compiler. *IEEE Transactions on Dependable and Secure Computing* 18, 2 (2021), 667–678. https://doi.org/10.1109/TDSC.2019.2923215

[27] Sujit Rokka Chhetri, Arquimedes Canedo, and Mohammad Abdullah Al Faruque. 2016. KCAD: Kinetic Cyber-attack detection method for Cyber-physical additive manufacturing systems. In *2016 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. 1–8. https://doi.org/10.1145/2966986.2967050

[28] Sujit Rokka Chhetri, Sina Faezi, Nafiul Rashid, and Mohammad Abdullah Al Faruque. 2018. Manufacturing supply chain and product lifecycle security in the era of industry 4.0. *Journal of Hardware and Systems Security* 2 (2018), 51–68.

[29] Sujit Rokka Chhetri, Nafiul Rashid, Sina Faezi, and Mohammad Abdullah Al Faruque. 2017. Security trends and advances in manufacturing systems in the era of industry 4.0. In *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. 1039–1046. https://doi.org/10.1109/ICCAD.2017.8203896

[30] Robert Coop and Itamar Arel. 2013. Mitigation of catastrophic forgetting in recurrent neural networks using a Fixed Expansion Layer. In *The 2013 International Joint Conference on Neural Networks (IJCNN)*. 1–7. https://doi.org/10.1109/IJCNN.2013.6707047

[31] Drew Davidson, Benjamin Moench, Thomas Ristenpart, and Somesh Jha. 2013. FIE on Firmware: Finding Vulnerabilities in Embedded Systems Using Symbolic Execution. In *22nd USENIX Security Symposium (USENIX Security 13)*. USENIX Association, Washington, D.C., 463–478. https://www.usenix.org/conference/usenixsecurity13/technical-sessions/paper/davidson

[32] Paul Denny, Viraj Kumar, and Nasser Giacaman. 2023. Conversing with copilot: Exploring prompt engineering for solving cs1 problems using natural language. In *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1*. 1136–1142.

[33] Sahar Deppe, Lukas Brandt, Marc Brünninghaus, Jörg Papenkordt, Stefan Heindorf, and Gudrun Tschirner-Vinke. 2022. AI-Based Assistance System for Manufacturing. In *2022 IEEE 27th International Conference on Emerging Technologies and Factory Automation (ETFA)*. IEEE, 1–4.

[34] Tim Dettmers, Artidoro Pagnoni, Ari Holtzman, and Luke Zettlemoyer. 2024. Qlora: Efficient finetuning of quantized llms. *Advances in Neural Information Processing Systems* 36 (2024).

[35] Adam Doupé. 2019. History and Future of Automated Vulnerability Analysis. In *Proceedings of the 24th ACM Symposium on Access Control Models and Technologies* (Toronto ON, Canada) *(SACMAT '19)*. Association for Computing Machinery, New York, NY, USA, 147. https://doi.org/10.1145/3322431.3326331

[36] Nan Du, Yanping Huang, Andrew M Dai, Simon Tong, Dmitry Lepikhin, Yuanzhong Xu, Maxim Krikun, Yanqi Zhou, Adams Wei Yu, Orhan Firat, et al. 2022. Glam: Efficient scaling of language models with mixture-of-experts. In *International Conference on Machine Learning*. PMLR, 5547–5569.

[37] Alpana Dubey. 2011. Evaluating software engineering methods in the context of automation applications. In *2011 9th IEEE International Conference on Industrial Informatics*. 585–590. https://doi.org/10.1109/INDIN.2011.6034944

[38] Sina Faezi, Sujit Rokka Chhetri, Arnav Vaibhav Malawade, John Charles Chaput, William Grover, Philip Brisk, and Mohammad Abdullah Al Faruque. 2019. Oligo-snoop: a non-invasive side channel attack against DNA synthesis machines. In *Network and Distributed Systems Security (NDSS) Symposium 2019*.

[39] Mohamad Fakih, Rahul Dharmaji, Yasamin Moghaddas, Gustavo Quiros Araya, Oluwatosin Ogundare, and Mohammad Abdullah Al Faruque. 2024. LLM4PLC: Harnessing Large Language Models for Verifiable Programming of PLCs in Industrial Control Systems. *arXiv preprint arXiv:2401.05443* (2024).

[40] Habiba Farrukh, Tinghan Yang, Hanwen Xu, Yuxuan Yin, He Wang, and Z. Berkay Celik. 2021. S3: Side-Channel Attack on Stylus Pencil through Sensors. *Proc. ACM Interact. Mob. Wearable Ubiquitous Technol.* 5, 1, Article 8 (mar 2021), 25 pages. https://doi.org/10.1145/3448085

[41] Mohammad Al Faruque, Francesco Regazzoni, and Miroslav Pajic. 2015. Design methodologies for securing cyber-physical systems. In *2015 International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*. 30–36. https://doi.org/10.1109/CODESISSS.2015.7331365

[42] Wenfeng Feng, Chuzhan Hao, Yuewei Zhang, Yu Han, and Hao Wang. 2024. Mixture-of-LoRAs: An Efficient Multitask Tuning for Large Language Models. *arXiv preprint arXiv:2403.03432* (2024).

[43] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. 2020. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155* (2020).

[44] FFmpeg Team. [n. d.]. FFmpeg. https://www.ffmpeg.org/

[45] Emily First and Yuriy Brun. 2022. Diversity-Driven Automated Formal Verification. In *Proceedings of 44th International Conference on Software Engineering (ICSE 2022)*. ACM, 749–761. https://doi.org/10.1145/3510003.3510138

[46] FischerTechnik. 2023. Factory Simulation 24V. https://www.fischertechnik.de/en/products/industry-and-universities/training-models/536634-factory-simulation-24v Product description and specifications.

[47] Michael Fu, Van Nguyen, Chakkrit Tantithamthavorn, Dinh Phung, and Trung Le. 2024. Vision Transformer Inspired Automated Vulnerability Repair. *ACM Trans. Softw. Eng. Methodol.* 33, 3, Article 78 (mar 2024), 29 pages. https://doi.org/10.1145/3632746

[48] Michael Fu, Chakkrit Tantithamthavorn, Trung Le, Van Nguyen, and Dinh Phung. 2022. VulRepair: a T5-based automated software vulnerability repair. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (<conf-loc>, <city>Singapore</city>, <country>Singapore</country>, </conf-loc>) *(ESEC/FSE 2022)*. Association for Computing Machinery, New York, NY, USA, 935–947. https://doi.org/10.1145/3540250.3549098

[49] Zihao Fu, Haoran Yang, Anthony Man-Cho So, Wai Lam, Lidong Bing, and Nigel Collier. 2023. On the effectiveness of parameter-efficient fine-tuning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 37. 12799–12807.

[50] Xiang Gao, Bo Wang, Gregory J Duck, Ruyi Ji, Yingfei Xiong, and Abhik Roychoudhury. 2021. Beyond tests: Program vulnerability repair via crash constraint extraction. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 30, 2 (2021), 1–27.

[51] Joshua Garcia, Yang Feng, Junjie Shen, Sumaya Almanee, Yuan Xia, Chen, and Qi Alfred. 2020. A comprehensive study of autonomous vehicle bugs. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (Seoul, South Korea) *(ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 385–396. https://doi.org/10.1145/3377811.3380397

[52] Daniel Genkin, Adi Shamir, and Eran Tromer. 2017. Acoustic Cryptanalysis. *J. Cryptol.* 30, 2 (apr 2017), 392–443. https://doi.org/10.1007/s00145-015-9224-2

[53] Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. 2022. Unixcoder: Unified cross-modal pre-training for code representation. *Proceedings of the Annual Meeting of the Association for Computational Linguistics* 1 (2022).

[54] Li Hao, Jianqi Shi, Ting Su, and Yanhong Huang. 2019. Automated Test Generation for IEC 61131-3 ST Programs via Dynamic Symbolic Execution. In *2019 International Symposium on Theoretical Aspects of Software Engineering (TASE)*. 200–207. https://doi.org/10.1109/TASE.2019.00004

[55] Jacob Harer, Onur Ozdemir, Tomo Lazovich, Christopher Reale, Rebecca Russell, Louis Kim, et al. 2018. Learning to repair software vulnerabilities with generative adversarial networks. *Advances in neural information processing systems* 31 (2018).

[56] Weigang He, Jianqi Shi, Ting Su, Zeyu Lu, Li Hao, and Yanhong Huang. 2021. Automated test generation for IEC 61131-3 ST programs via dynamic symbolic execution. *Science of Computer Programming* 206 (2021), 102608.

[57] C.A.R. Hoare. 2000. Legacy code. In *ICFEM 2000. Third IEEE International Conference on Formal Engineering Methods*. 75–75. https://doi.org/10.1109/ICFEM.2000.873807

[58] James J. Horning. 1975. Yes! high level languages should be used to write systems software. In *Proceedings of the 1975 Annual Conference (ACM '75)*. Association for Computing Machinery, New York, NY, USA, 206–208. https://doi.org/10.1145/800181.810318

[59] Allen D. Householder, Jeff Chrabaszcz, Trent Novelly, David Warren, and Jonathan M. Spring. 2020. Historical Analysis of Exploit Availability Timelines. In *13th USENIX Workshop on Cyber Security Experimentation and Test (CSET 20)*. USENIX Association. https://www.usenix.org/conference/cset20/presentation/householder

[60] Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. 2021. LoRA: Low-Rank Adaptation of Large Language Models. arXiv:arXiv:2106.09685

[61] Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. 2022. Lora: Low-rank adaptation of large language models. *International Conference on Learning Representations* (2022).

[62] Pengfei Hu, Yifan Ma, Panneer Selvam Santhalingam, Parth H Pathak, and Xiuzhen Cheng. 2022. MILLIEAR: Millimeter-wave Acoustic Eavesdropping with Unconstrained Vocabulary. In *IEEE INFOCOM 2022 - IEEE Conference on Computer Communications*. 11–20. https://doi.org/10.1109/INFOCOM48880.2022.9796940

[63] Nicolas Huaman, Bennet von Skarczinski, Christian Stransky, Dominik Wermke, Yasemin Acar, Arne Dreißigacker, and Sascha Fahl. 2021. A Large-Scale Interview Study on Information Security in and Attacks against Small and Medium-sized Enterprises. In *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, 1235–1252. https://www.usenix.org/conference/usenixsecurity21/presentation/huaman

[64] International Electrotechnical Commission. 2013. *Programmable Controllers – Part 3: Programming Languages*. Technical Report IEC 61131-3. IEC. http://webstore.iec.ch/publication/4552

[65] Internet Crime Complaint Center. [n. d.]. 2023 Internet Crime Report. https://www.ic3.gov/Media/PDF/AnnualReport/2023_IC3Report.pdf

[66] Naman Jain, Skanda Vaidyanath, Arun Iyer, Nagarajan Natarajan, Suresh Parthasarathy, Sriram Rajamani, and Rahul Sharma. 2022. Jigsaw: large language models meet program synthesis. In *Proceedings of the 44th International Conference on Software Engineering* (Pittsburgh, Pennsylvania) *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 1219–1231. https://doi.org/10.1145/3510003.3510203

[67] Jeff C. Jensen, Danica H. Chang, and Edward A. Lee. 2011. A model-based design methodology for cyber-physical systems. In *2011 7th International Wireless Communications and Mobile Computing Conference*. 1666–1671. https://doi.org/10.1109/IWCMC.2011.5982785

[68] Albert Q. Jiang, Alexandre Sablayrolles, Antoine Roux, Arthur Mensch, Blanche Savary, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Emma Bou Hanna, Florian Bressand, Gianna Lengyel, Guillaume Bour, Guillaume Lample, Lélio Renard Lavaud, Lucile Saulnier, Marie-Anne Lachaux, Pierre Stock, Sandeep Subramanian, Sophia Yang, Szymon Antoniak, Teven Le Scao, Théophile Gervet, Thibaut Lavril, Thomas Wang, Timothée Lacroix, and William El Sayed. 2024. Mixtral of Experts. arXiv:arXiv:2401.04088

[69] Matthew Jin, Syed Shahriar, Michele Tufano, Xin Shi, Shuai Lu, Neel Sundaresan, and Alexey Svyatkovskiy. 2023. InferFix: End-to-End Program Repair with LLMs. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (<conf-loc>, <city>San Francisco</city>, <state>CA</state>, <country>USA</country>, </conf-loc>) *(ESEC/FSE 2023)*. Association for Computing Machinery, New York, NY, USA, 1646–1656. https://doi.org/10.1145/3611643.3613892

[70] Harshit Joshi, José Cambronero Sanchez, Sumit Gulwani, Vu Le, Gust Verbruggen, and Ivan Radiček. 2023. Repair is nearly generation: Multilingual program repair with llms. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 37. 5131–5140.

[71] Seulbae Kim, Major Liu, Junghwan "John" Rhee, Yuseok Jeon, Yonghwi Kwon, and Chung Hwan Kim. 2022. DriveFuzz: Discovering Autonomous Driving Bugs through Driving Quality-Guided Fuzzing. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security* (Los Angeles, CA, USA) *(CCS '22)*. Association for Computing Machinery, New York, NY, USA, 1753–1767. https://doi.org/10.1145/3548606.3560558

[72] Taegyu Kim, Vireshwar Kumar, Junghwan Rhee, Jizhou Chen, Kyungtae Kim, Chung Hwan Kim, Dongyan Xu, and Dave (Jing) Tian. 2021. PASAN: Detecting Peripheral Access Concurrency Bugs within Bare-Metal Embedded Applications. In *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, 249–266. https://www.usenix.org/conference/usenixsecurity21/presentation/kim

[73] William Klieber, Ruben Martins, Ryan Steele, Matt Churilla, Mike McCall, and David Svoboda. 2021. Automated code repair to ensure spatial memory safety. In *2021 IEEE/ACM International Workshop on Automated Program Repair (APR)*. IEEE, 23–30.

[74] Andreas Kogler, Jonas Juffinger, Lukas Giner, Lukas Gerlach, Martin Schwarzl, Michael Schwarz, Daniel Gruss, and Stefan Mangard. 2023. Collide+Power: Leaking Inaccessible Data with Software-based Power Side Channels. In *32nd USENIX Security Symposium (USENIX Security 23)*. USENIX Association, Anaheim, CA, 7285–7302. https://www.usenix.org/conference/usenixsecurity23/presentation/kogler

[75] Takeshi Kojima, Shixiang Shane Gu, Machel Reid, Yutaka Matsuo, and Yusuke Iwasawa. 2022. Large language models are zero-shot reasoners. *Advances in neural information processing systems* 35 (2022), 22199–22213.

[76] E.V. Kuzmin, A.A. Shipov, and D.A. Ryabukhin. 2013. Construction and verification of PLC programs by LTL specification. In *2013 Tools & Methods of Program Analysis*. 15–22. https://doi.org/10.1109/TMPA.2013.7163716

[77] Andrew Kwong, Wenyuan Xu, and Kevin Fu. 2019. Hard Drive of Hearing: Disks that Eavesdrop with a Synthesized Microphone. In *2019 IEEE Symposium on Security and Privacy (SP)*. 905–919. https://doi.org/10.1109/SP.2019.00008

[78] Davy Landman, Alexander Serebrenik, and Jurgen J Vinju. 2017. Challenges for static analysis of java reflection-literature review and empirical study. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, 507–518.

[79] Hung Le, Yue Wang, Akhilesh Deepak Gotmare, Silvio Savarese, and Steven C. H. Hoi. 2022. CodeRL: Mastering Code Generation through Pretrained Models and Deep Reinforcement Learning. arXiv:arXiv:2207.01780

[80] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, et al. 2020. Retrieval-augmented generation for knowledge-intensive nlp tasks. *Advances in Neural Information Processing Systems* 33 (2020), 9459–9474.

[81] Yang Li, Shihao Wu, and Quan Pan. 2023. Network security in the industrial control system: A survey. *arXiv preprint arXiv:2308.03478* (2023).

[82] Zongjie Li, Chaozheng Wang, Zhibo Liu, Haoxuan Wang, Dong Chen, Shuai Wang, and Cuiyun Gao. 2023. CCTEST: Testing and Repairing Code Completion Systems. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. 1238–1250. https://doi.org/10.1109/ICSE48619.2023.00110

[83] Zhen Li, Deqing Zou, Shouhuai Xu, Xinyu Ou, Hai Jin, Sujuan Wang, Zhijun Deng, and Yuyi Zhong. 2018. VulDeePecker: A Deep Learning-Based System for Vulnerability Detection. In *25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018*. The Internet Society. https://www.ndss-symposium.org/wp-content/uploads/2018/02/ndss2018_03A-2_Li_paper.pdf

[84] Qianru Liao, Yongzhi Huang, Yandao Huang, Yuheng Zhong, Huitong Jin, and Kaishun Wu. 2022. MagEar: Eavesdropping via Audio Recovery Using Magnetic Side Channel. In *Proceedings of the 20th Annual International Conference on Mobile Systems, Applications and Services* (Portland, Oregon) *(MobiSys '22)*. Association for Computing Machinery, New York, NY, USA, 371–383. https://doi.org/10.1145/3498361.3538921

[85] Linux Foundation. [n. d.]. Linux Kernel. https://www.kernel.org/

[86] Haokun Liu, Derek Tam, Mohammed Muqeeth, Jay Mohta, Tenghao Huang, Mohit Bansal, and Colin Raffel. 2022. Few-Shot Parameter-Efficient Fine-Tuning is Better and Cheaper than In-Context Learning.

[87] Haokun Liu, Derek Tam, Mohammed Muqeeth, Jay Mohta, Tenghao Huang, Mohit Bansal, and Colin A Raffel. 2022. Few-Shot Parameter-Efficient Fine-Tuning is Better and Cheaper than In-Context Learning. In *Advances in Neural Information Processing Systems*, S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh (Eds.), Vol. 35. Curran Associates, Inc., 1950–1965. https://proceedings.neurips.cc/paper_files/paper/2022/file/0cde695b83bd186c1fd456302888454c-Paper-Conference.pdf

[88] Haokun Liu, Derek Tam, Mohammed Muqeeth, Jay Mohta, Tenghao Huang, Mohit Bansal, and Colin A Raffel. 2022. Few-shot parameter-efficient fine-tuning is better and cheaper than in-context learning. *Advances in Neural Information Processing Systems* 35 (2022), 1950–1965.

[89] 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. In *Advances in Neural Information Processing Systems*, A. Oh, T. Neumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine (Eds.), Vol. 36. Curran Associates, Inc., 21558–21572. https://proceedings.neurips.cc/paper_files/paper/2023/file/43e9d647ccd3e4b7b5baab53f0368686-Paper-Conference.pdf

[90] Kui Liu, Li Li, Anil Koyuncu, Dongsun Kim, Zhe Liu, Jacques Klein, and Tegawendé F. Bissyandé. 2021. A critical review on the evaluation of automated program repair systems. *Journal of Systems and Software* 171 (2021), 110817. https://doi.org/10.1016/j.jss.2020.110817

[91] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. 2020. Ro{BERT}a: A Robustly Optimized {BERT} Pretraining Approach. arXiv:arXiv:1907.11692

[92] George Lo, Mohammad Abdullah Al Faruque, Hartmut Ludwig, Livio Dalloro, Barry R. Contrael, and Paul Terricciano. 2015. Network as automation platform for collaborative E-car charging at the residential premises. Google Patents. https://patents.google.com/patent/US8957634B2/en US Patent 8,957,634.

[93] Junyi Lu, Lei Yu, Xiaojia Li, Li Yang, and Chun Zuo. 2023. LLaMA-Reviewer: Advancing Code Review Automation with Large Language Models through Parameter-Efficient Fine-Tuning (Practical Experience Report). *arXiv preprint arXiv:2308.11148* (2023).

[94] Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, et al. 2021. Codexglue: A machine learning benchmark dataset for code understanding and generation. *arXiv preprint arXiv:2102.04664* (2021).

[95] Ziyang Luo, Can Xu, Pu Zhao, Qingfeng Sun, Xiubo Geng, Wenxiang Hu, Chongyang Tao, Jing Ma, Qingwei Lin, and Daxin Jiang. 2023. Wizardcoder: Empowering code large language models with evol-instruct. *arXiv preprint arXiv:2306.08568* (2023).

[96] David Martin, Massimo Paolucci, Sheila Mcilraith, Mark Burstein, Drew Mcdermott, Deborah Mcguinness, Bijan Parsia, Terry Payne, Marta Sabou, Monika Solanki, and Naveen Srinivasan. 2004. Bringing Semantics to Web Services: the OWL-S approach. *Lecture Notes in Computer Science* 3387, 26–42. https://doi.org/10.1007/978-3-540-30581-1_4

[97] Maximilian Schreiner. [n. d.]. GPT-4 architecture, datasets, costs and more leaked. https://the-decoder.com/gpt-4-architecture-datasets-costs-and-more-leaked/

[98] Yan Michalevsky, Dan Boneh, and Gabi Nakibly. 2014. Gyrophone: Recognizing Speech from Gyroscope Signals. In *23rd USENIX Security Symposium (USENIX Security 14)*. USENIX Association, San Diego, CA, 1053–1067. https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/michalevsky

[99] MITRE Corporation [n. d.]. *CVE - MITRE*. MITRE Corporation.

[100] Martin Monperrus. 2018. Automatic Software Repair: A Bibliography. *ACM Comput. Surv.* 51, 1, Article 17 (jan 2018), 24 pages. https://doi.org/10.1145/3105906

[101] Pieter Mosterman. 2007. Model-Based Design of Embedded Systems. In *2007 IEEE International Conference on Microelectronic Systems Education (MSE'07)*. 3–3. https://doi.org/10.1109/MSE.2007.65

[102] Manish Motwani, Mauricio Soto, Yuriy Brun, René Just, and Claire Le Goues. 2022. Quality of Automated Program Repair on Real-World Defects. *IEEE Transactions on Software Engineering* 48, 2 (2022), 637–661. https://doi.org/10.1109/TSE.2020.2998785

[103] Bhavesh Raju Mudhivarthi, Vaibhav Saini, Ayush Dodia, Pritesh Shah, and Ravi Sekhar. 2023. Model Based Design in Automotive Open System Architecture. In *2023 7th International Conference on Intelligent Computing and Control Systems (ICICCS)*. 1211–1216. https://doi.org/10.1109/ICICCS56967.2023.10142603

[104] National Vulnerability Database [n. d.]. *NVD - Home*. National Vulnerability Database.

[105] Humza Naveed, Asad Ullah Khan, Shi Qiu, Muhammad Saqib, Saeed Anwar, Muhammad Usman, Naveed Akhtar, Nick Barnes, and Ajmal Mian. 2023. A Comprehensive Overview of Large Language Models. arXiv:arXiv:2307.06435

[106] New Jersey Speech-Language-Hearing Association. 2021. The Speech Banana. https://www.njsha.org/pdfs/hearing-speech-banana.pdf

[107] Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2022. Codegen: An open large language model for code with multi-turn program synthesis. *arXiv preprint arXiv:2203.13474* (2022).

[108] Nucleron. 2017. MATIEC. https://github.com/nucleron/matiec Source code repository.

[109] International Society of Automation. [n. d.]. Industrial automation global market report. https://www.automation.com/en-us/articles/july-2022/industrial-automation-global-market-report-2022

[110] Amit Elazari Bar On. 2018. The Law and Economics of Bug Bounties. USENIX Association, Baltimore, MD.

[111] OpenAI. [n. d.]. Models - OpenAI API. https://platform.openai.com/docs/models

[112] OpenAI. [n. d.]. OpenAI | Pricing. https://openai.com/api/pricing

[113] OpenAI. 2023. GPT-4 Technical Report. arXiv:arXiv:2303.08774

[114] OSCAT. 2023. OSCAT BASIC. http://www.oscat.de/de/63-oscat-basic-321.html Product description and specifications.

[115] Osterman Research, Inc. [n. d.]. Second Annual State of Ransomware Report: Survey Results for Australia. https://go.malwarebytes.com/rs/805-USG-300/images/Second%20Annual%20State%20of%20Ransomware%20Report%20-%20Australia.pdf

[116] Tolga Ovatman, Atakan Aral, Davut Polat, and Ali Osman Ünver. 2016. An overview of model checking practices on verification of PLC software. *Software & Systems Modeling* 15, 4 (2016), 937–960.

[117] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*. 311–318.

[118] Natalie Paskoski. [n. d.]. *Using the NIST Cybersecurity Framework in Your Vulnerability Management Process*. Technical Report. Retail and Hospitality Information Security and Analysis Center. https://rhisac.org/vulnerability-management/nist-framework-vulnerability-management/

[119] Hammond Pearce, Benjamin Tan, Baleegh Ahmad, Ramesh Karri, and Brendan Dolan-Gavitt. 2021. Examining Zero-Shot Vulnerability Repair with Large Language Models. arXiv:arXiv:2112.02125

[120] Hammond Pearce, Benjamin Tan, Baleegh Ahmad, Ramesh Karri, and Brendan Dolan-Gavitt. 2023. Examining Zero-Shot Vulnerability Repair with Large Language Models. In *2023 IEEE Symposium on Security and Privacy (SP)*. 2339–2356. https://doi.org/10.1109/SP46215.2023.10179324

[121] Kexin Pei, David Bieber, Kensen Shi, Charles Sutton, and Pengcheng Yin. 2023. Can large language models reason about program invariants?. In *International Conference on Machine Learning*. PMLR, 27496–27520.

[122] Sean Peisert, Bruce Schneier, Hamed Okhravi, Fabio Massacci, Terry Benzel, Carl Landwehr, Mohammad Mannan, Jelena Mirkovic, Atul Prakash, and James Bret Michael. 2021. Perspectives on the SolarWinds Incident. *IEEE Security Privacy* 19, 2 (2021), 7–13. https://doi.org/10.1109/MSEC.2021.3051235

[123] Eduard Pinconschi, Rui Abreu, and Pedro Adão. 2021. A comparative study of automatic program repair techniques for security vulnerabilities. In *2021 IEEE 32nd international symposium on software reliability engineering (ISSRE)*. IEEE, 196–207.

[124] Eduard Pinconschi, Rui Abreu, and Pedro Adão. 2021. A Comparative Study of Automatic Program Repair Techniques for Security Vulnerabilities. In *2021 IEEE 32nd International Symposium on Software Reliability Engineering (ISSRE)*. 196–207. https://doi.org/10.1109/ISSRE52982.2021.00031

[125] Herbert Prähofer, Florian Angerer, Rudolf Ramler, and Friedrich Grillenberger. 2016. Static code analysis of IEC 61131-3 programs: Comprehensive tool support and experiences from large-scale industrial application. *IEEE Transactions on Industrial Informatics* 13, 1 (2016), 37–47.

[126] Herbert Prähofer, Florian Angerer, Rudolf Ramler, Hermann Lacheiner, and Friedrich Grillenberger. 2012. Opportunities and challenges of static code analysis of IEC 61131-3 programs. In *Proceedings of 2012 IEEE 17th International Conference on Emerging Technologies & Factory Automation (ETFA 2012)*. IEEE, 1–8.

[127] Juha Puttonen, Andrei Lobov, and José L Martinez Lastra. 2012. Semantics-based composition of factory automation processes encapsulated by web services. *IEEE Transactions on industrial informatics* 9, 4 (2012), 2349–2359.

[128] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. 2019. Language models are unsupervised multitask learners. *OpenAI blog* 1, 8 (2019), 9.

[129] Ossama Rashad, Omneya Attallah, and Iman Morsi. 2022. A PLC-SCADA Pipeline for Managing Oil Refineries. In *2022 5th International Conference on Computing and Informatics (ICCI)*. 216–220. https://doi.org/10.1109/ICCI54321.2022.9756108

[130] Joel Reardon, Álvaro Feal, Primal Wijesekera, Amit Elazari Bar On, Narseo Vallina-Rodriguez, and Serge Egelman. 2019. 50 Ways to Leak Your Data: An Exploration of Apps' Circumvention of the Android Permissions System. In *28th USENIX Security Symposium (USENIX Security 19)*. USENIX Association, Santa Clara, CA, 603–620. https://www.usenix.org/conference/usenixsecurity19/presentation/reardon

[131] Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, Neel Sundaresan, Ming Zhou, Ambrosio Blanco, and Shuai Ma. 2020. Codebleu: a method for automatic evaluation of code synthesis. *arXiv preprint arXiv:2009.10297* (2020).

[132] Meta Research. 2024. Meta Llama 3. https://llama.meta.com/llama3/

[133] James S. Rogers. 2011. Language choice for safety critical applications. *Ada Lett.* 31, 3 (nov 2011), 81–90. https://doi.org/10.1145/2070336.2070363

[134] Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, et al. 2023. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950* (2023).

[135] Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. 2023. Code Llama: Open Foundation Models for Code. arXiv:arXiv:2308.12950

[136] Norman Santiago and Janelli Mendez. 2023. Analysis of Common Vulnerabilities and Exposures to Produce Security Trends. In *Proceedings of the 2022 International Conference on Cyber Security* (<conf-loc>, <city>Hangzhou</city>, <country>China</country>, </conf-loc>) *(CSW '22)*. Association for Computing Machinery, New York, NY, USA, 16–19. https://doi.org/10.1145/3584714.3584718

[137] Sunny Sanyal, Jean Kaddour, Abhishek Kumar, and Sujay Sanghavi. 2023. Understanding the Effectiveness of Early Weight Averaging for Training Large Language Models. arXiv:arXiv:2306.03241

[138] Arman Sargolzaei, Alireza Abbaspour, Mohammad Abdullah Al Faruque, Anas Salah Eddin, and Kang Yen. 2018. Security challenges of networked control systems. *Sustainable Interdependent Networks: From Theory to Application* (2018), 77–95.

[139] Rylan Schaeffer, Brando Miranda, and Sanmi Koyejo. 2024. Are emergent abilities of large language models a mirage? *Advances in Neural Information Processing Systems* 36 (2024).

[140] Frank Schumacher, Sebastian Schröck, and Alexander Fay. 2013. Tool support for an automatic transformation of GRAFCET specifications into IEC 61131-3 control code. In *2013 IEEE 18th Conference on Emerging Technologies & Factory Automation (ETFA)*. IEEE, 1–4.

[141] Freda Shi, Xinyun Chen, Kanishka Misra, Nathan Scales, David Dohan, Ed H Chi, Nathanael Schärli, and Denny Zhou. 2023. Large language models can be easily distracted by irrelevant context. In *International Conference on Machine Learning*. PMLR, 31210–31227.

[142] Aleksandar Shtedritski, Christian Rupprecht, and Andrea Vedaldi. 2023. What does clip know about a red circle? visual prompt engineering for vlms. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*. 11987–11997.

[143] Mohammed Latif Siddiq, Beatrice Casey, and Joanna C. S. Santos. 2023. A Lightweight Framework for High-Quality Code Generation. arXiv:arXiv:2307.08220

[144] André Silva, Sen Fang, and Martin Monperrus. 2023. RepairLLaMA: Efficient Representations and Fine-Tuned Adapters for Program Repair. *arXiv preprint arXiv:2312.15698* (2023).

[145] Saleh Soltan, Prateek Mittal, and H. Vincent Poor. 2018. BlackIoT: IoT Botnet of High Wattage Devices Can Disrupt the Power Grid. In *27th USENIX Security Symposium (USENIX Security 18)*. USENIX Association, Baltimore, MD, 15–32. https://www.usenix.org/conference/usenixsecurity18/presentation/soltan

[146] Alexander Ivanov Sotirov. 2005. *Automatic vulnerability detection using static source code analysis*. Ph. D. Dissertation. Citeseer.

[147] Aleksandar J. Spasić and Dragan S. Janković. 2023. Using ChatGPT Standard Prompt Engineering Techniques in Lesson Preparation: Role, Instructions and Seed-Word Prompts. In *2023 58th International Scientific Conference on Information, Communication and Energy Systems and Technologies (ICEST)*. 47–50. https://doi.org/10.1109/ICEST58410.2023.10187269

[148] Franz-Josef Streit, Martin Letras, Stefan Wildermann, Benjamin Hackenberg, Joachim Falk, Andreas Becher, and Jürgen Teich. 2018. Model-Based Design Automation of Hardware/Software Co-Designs for Xilinx Zynq PSoCs. In *2018 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*. 1–8. https://doi.org/10.1109/RECONFIG.2018.8641736

[149] Hendrik Strobelt, Albert Webson, Victor Sanh, Benjamin Hoover, Johanna Beyer, Hanspeter Pfister, and Alexander M Rush. 2022. Interactive and visual prompt engineering for ad-hoc task adaptation with large language models. *IEEE transactions on visualization and computer graphics* 29, 1 (2022), 1146–1156.

[150] Hendrik Strobelt, Albert Webson, Victor Sanh, Benjamin Hoover, Johanna Beyer, Hanspeter Pfister, and Alexander M. Rush. 2023. Interactive and Visual Prompt Engineering for Ad-hoc Task Adaptation with Large Language Models. *IEEE Transactions on Visualization and Computer Graphics* 29, 1 (2023), 1146–1156. https://doi.org/10.1109/TVCG.2022.3209479

[151] Seyed Mohammadjavad Seyed Talebi, Zhihao Yao, Ardalan Amiri Sani, Zhiyun Qian, and Daniel Austin. 2021. Undo Workarounds for Kernel Bugs. In *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, 2381–2398. https://www.usenix.org/conference/usenixsecurity21/presentation/talebi

[152] Ze Tang, Jidong Ge, Shangqing Liu, Tingwei Zhu, Tongtong Xu, Liguo Huang, and Bin Luo. 2023. Domain Adaptive Code Completion via Language Models and Decoupled Domain Databases. *ACM International Conference on Automated Software Engineering* (2023).

[153] Artur Tarassow. 2023. The potential of LLMs for coding with low-resource and domain-specific programming languages. arXiv:arXiv:2307.13018

[154] Shailja Thakur, Baleegh Ahmad, Hammond Pearce, Benjamin Tan, Brendan Dolan-Gavitt, Ramesh Karri, and Siddharth Garg. 2023. VeriGen: A Large Language Model for Verilog Code Generation. *arXiv preprint arXiv:2308.00708* (2023).

[155] The OpenSSL Project. [n. d.]. OpenSSL. https://www.openssl.org/

[156] Alexander Tornede, Difan Deng, Theresa Eimer, Joseph Giovanelli, Aditya Mohan, Tim Ruhkopf, Sarah Segel, Daphne Theodorakopoulos, Tanja Tornede, Henning Wachsmuth, and Marius Lindauer. 2023. AutoML in the Age of Large Language Models: Current Challenges, Future Opportunities and Risks. arXiv:arXiv:2306.08107

[157] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. 2023. LLaMA: open and efficient foundation language models, 2023. *URL https://arxiv. org/abs/2302.13971* (2023).

[158] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, Dan Bikel, Lukas Blecher, Cristian Canton Ferrer, Moya Chen, Guillem Cucurull, David Esiobu, Jude Fernandes, Jeremy Fu, Wenyin Fu, Brian Fuller, Cynthia Gao, Vedanuj Goswami, Naman Goyal, Anthony Hartshorn, Saghar Hosseini, Rui Hou, Hakan Inan, Marcin Kardas, Viktor Kerkez, Madian Khabsa, Isabel Kloumann, Artem Korenev, Punit Singh Koura, Marie-Anne Lachaux, Thibaut Lavril, Jenya Lee, Diana Liskovich, Yinghai Lu, Yuning Mao, Xavier Martinet, Todor Mihaylov, Pushkar Mishra, Igor Molybog, Yixin Nie, Andrew Poulton, Jeremy Reizenstein, Rashi Rungta, Kalyan Saladi, Alan Schelten, Ruan Silva, Eric Michael Smith, Ranjan Subramanian, Xiaoqing Ellen Tan, Binh Tang, Ross Taylor, Adina Williams, Jian Xiang Kuan, Puxin Xu, Zheng Yan, Iliyan Zarov, Yuchen Zhang, Angela Fan, Melanie Kambadur, Sharan Narang, Aurelien Rodriguez, Robert Stojnic, Sergey Edunov, and Thomas Scialom. 2023. Llama 2: Open Foundation and Fine-Tuned Chat Models. arXiv:arXiv:2307.09288

[159] Kohei Tsujio, Mohammad Abdullah Al Faruque, and Yasser Shoukry. 2024. Rampo: A CEGAR-based Integration of Binary Code Analysis and System Falsification for Cyber-Kinetic Vulnerability Detection. *arXiv preprint arXiv:2402.12642* (2024).

[160] Priyan Vaithilingam, Tianyi Zhang, and Elena L Glassman. 2022. Expectation vs. experience: Evaluating the usability of code generation tools powered by large language models. In *Chi conference on human factors in computing systems extended abstracts*. 1–7.

[161] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention Is All You Need. *CoRR* abs/1706.03762 (2017). arXiv:1706.03762 http://arxiv.org/abs/1706.03762

[162] B. Vogel-Heuser, D. Witsch, and U. Katzke. 2005. Automatic code generation from a UML model to IEC 61131-3 and system configuration tools. In *2005 International Conference on Control and Automation*, Vol. 2. 1034–1039 Vol. 2. https://doi.org/10.1109/ICCA.2005.1528274

[163] Daniel Votipka, Kelsey R. Fulton, James Parker, Matthew Hou, Michelle L. Mazurek, and Michael Hicks. 2020. Understanding security mistakes developers make: Qualitative analysis from Build It, Break It, Fix It. In *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, 109–126. https://www.usenix.org/conference/usenixsecurity20/presentation/votipka-understanding

[164] Jiang Wan, Arquimedes Canedo, and Mohammad Abdullah Al Faruque. 2015. Security-aware functional modeling of Cyber-Physical Systems. In *2015 IEEE 20th Conference on Emerging Technologies Factory Automation (ETFA)*. 1–4. https://doi.org/10.1109/ETFA.2015.7301644

[165] Jiang Wan, Anthony Bahadir Lopez, and Mohammad Abdullah Al Faruque. 2016. Exploiting Wireless Channel Randomness to Generate Keys for Automotive Cyber-Physical System Security. In *2016 ACM/IEEE 7th International Conference on Cyber-Physical Systems (ICCPS)*. 1–10. https://doi.org/10.1109/ICCPS.2016.7479103

[166] Yue Wang, Weishi Wang, Shafiq Joty, and Steven CH Hoi. 2021. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. *arXiv preprint arXiv:2109.00859* (2021).

[167] Jason Wei, Maarten Bosma, Vincent Y. Zhao, Kelvin Guu, Adams Wei Yu, Brian Lester, Nan Du, Andrew M. Dai, and Quoc V. Le. 2021. Finetuned Language Models Are Zero-Shot Learners. arXiv:arXiv:2109.01652

[168] Jason Wei, Yi Tay, Rishi Bommasani, Colin Raffel, Barret Zoph, Sebastian Borgeaud, Dani Yogatama, Maarten Bosma, Denny Zhou, Donald Metzler, et al. 2022. Emergent abilities of large language models. *arXiv preprint arXiv:2206.07682* (2022).

[169] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Ed H. Chi, Quoc Le, and Denny Zhou. 2022. Chain of Thought Prompting Elicits Reasoning in Large Language Models. *Advances in Neural Information Processing Systems* Vol. 35 (2022).

[170] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. 2022. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems* 35 (2022), 24824–24837.

[171] Junyi Wei, Yicheng Zhang, Zhe Zhou, Zhou Li, and Mohammad Abdullah Al Faruque. 2020. Leaky DNN: Stealing Deep-Learning Model Secret with GPU Context-Switching Side-Channel. In *2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. 125–137. https://doi.org/10.1109/DSN48063.2020.00031

[172] Yuxiang Wei, Chunqiu Steven Xia, and Lingming Zhang. 2023. Copiloting the Copilots: Fusing Large Language Models with Completion Engines for Automated Program Repair. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (<conf-loc>, <city>San Francisco</city>, <state>CA</state>, <country>USA</country>, </conf-loc>) *(ESEC/FSE 2023)*. Association for Computing Machinery, New York, NY, USA, 172–184. https://doi.org/10.1145/3611643.3616271

[173] Guoyang Weng and Artur Andrzejak. 2023. Automatic Bug Fixing via Deliberate Problem Solving with Large Language Models. In *2023 IEEE 34th International Symposium on Software Reliability Engineering Workshops (ISSREW)*. 34–36. https://doi.org/10.1109/ISSREW60843.2023.00040

[174] Bernhard Werner, Birgit Vogel-Heuser, Simon Ziegltrum, Herbert Gröbl, and Claus Botzenhardt. 2020. Supporting troubleshooting in machine and plant manufacturing by backstepping of PLC-control software. In *2020 IEEE Conference on Industrial Cyberphysical Systems (ICPS)*, Vol. 1. 242–249. https://doi.org/10.1109/ICPS48405.2020.9274778

[175] 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. *arXiv preprint arXiv:2302.11382* (2023).

[176] Chunqiu Steven Xia, Yuxiang Wei, and Lingming Zhang. 2023. Automated Program Repair in the Era of Large Pre-trained Language Models. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. 1482–1494. https://doi.org/10.1109/ICSE48619.2023.00129

[177] Chunqiu Steven Xia and Lingming Zhang. 2023. Conversational automated program repair. *arXiv preprint arXiv:2301.13246* (2023).

[178] Chunqiu Steven Xia and Lingming Zhang. 2023. Keep the Conversation Going: Fixing 162 out of 337 bugs for $0.42 each using ChatGPT. *arXiv preprint arXiv:2304.00385* (2023).

[179] Honglin Xiong, Sheng Wang, Yitao Zhu, Zihao Zhao, Yuxiao Liu, Linlin Huang, Qian Wang, and Dinggang Shen. 2023. DoctorGLM: Fine-tuning your Chinese Doctor is not a Herculean Task. arXiv:arXiv:2304.01097

[180] Honglin Xiong, Sheng Wang, Yitao Zhu, Zihao Zhao, Yuxiao Liu, Linlin Huang, Qian Wang, and Dinggang Shen. 2023. DoctorGLM: Fine-tuning your Chinese Doctor is not a Herculean Task. (2023). arXiv:arXiv:2304.01097

[181] Frank F Xu, Uri Alon, Graham Neubig, and Vincent Josua Hellendoorn. 2022. A systematic evaluation of large language models of code. In *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming*. 1–10.

[182] Boyang Yang, Haoye Tian, Jiadong Ren, Hongyu Zhang, Jacques Klein, Tegawendé F Bissyandé, Claire Le Goues, and Shunfu Jin. 2024. Multi-Objective Fine-Tuning for Enhanced Program Repair with LLMs. *arXiv preprint arXiv:2404.12636* (2024).

[183] Zhuoyi Yang, Ming Ding, Yanhui Guo, Qingsong Lv, and Jie Tang. 2022. Parameter-Efficient Tuning Makes a Good Classification Head. arXiv:arXiv:2210.16771

[184] Junjie Ye, Xuanting Chen, Nuo Xu, Can Zu, Zekai Shao, Shichun Liu, Yuhan Cui, Zeyang Zhou, Chao Gong, Yang Shen, Jie Zhou, Siming Chen, Tao Gui, Qi Zhang, and Xuanjing Huang. 2023. A Comprehensive Capability Analysis of GPT-3 and GPT-3.5 Series Models. arXiv:arXiv:2303.10420

[185] Ziwen Han Keiran Paster Silviu Pitis Harris Chan Jimmy Ba Yongchao Zhou, Andrei Ioan Muresanu. 2022. Large Language Models Are Human-Level Prompt Engineers. In *Advances in Neural Information Processing Systems*. Curran Associates, Inc. https://openreview.net/forum?id=YdqwNaCLCx

[186] Fang Yu, Ching-Yuan Shueh, Chun-Han Lin, Yu-Fang Chen, Bow-Yaw Wang, and Tevfik Bultan. 2016. Optimal sanitization synthesis for web application vulnerability repair. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*. 189–200.

[187] Fang Yu, Ching-Yuan Shueh, Chun-Han Lin, Yu-Fang Chen, Bow-Yaw Wang, and Tevfik Bultan. 2016. Optimal sanitization synthesis for web application vulnerability repair. In *Proceedings of the 25th International Symposium on Software Testing and Analysis* (Saarbrücken, Germany) *(ISSTA 2016)*. Association for Computing Machinery, New York, NY, USA, 189–200. https://doi.org/10.1145/2931037.2931050

[188] Shih-Yuan Yu, Yonatan Gizachew Achamyeleh, Chonghan Wang, Anton Kocheturov, Patrick Eisen, and Mohammad Abdullah Al Faruque. 2023. CFG2VEC: Hierarchical Graph Neural Network for Cross-Architectural Software Reverse Engineering. In *2023 IEEE/ACM 45th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. 281–291. https://doi.org/10.1109/ICSE-SEIP58684.2023.00031

[189] Shih-Yuan Yu, Arnav Vaibhav Malawade, Sujit Rokka Chhetri, and Mohammad Abdullah Al Faruque. 2020. Sabotage Attack Detection for Additive Manufacturing Systems. *IEEE Access* 8 (2020), 27218–27231. https://doi.org/10.1109/ACCESS.2020.2971947

[190] Manzil Zaheer, Guru Guruganesh, Kumar Avinava Dubey, Joshua Ainslie, Chris Alberti, Santiago Ontanon, Philip Pham, Anirudh Ravula, Qifan Wang, Li Yang, et al. 2020. Big bird: Transformers for longer sequences. *Advances in neural information processing systems* 33 (2020), 17283–17297.

[191] Elad Ben Zaken, Shauli Ravfogel, and Yoav Goldberg. 2021. Bitfit: Simple parameter-efficient fine-tuning for transformer-based masked language-models. *arXiv preprint arXiv:2106.10199* (2021).

[192] Quanjun Zhang, Chunrong Fang, Yuxiang Ma, Weisong Sun, and Zhenyu Chen. 2023. A Survey of Learning-based Automated Program Repair. *ACM Trans. Softw. Eng. Methodol.* 33, 2, Article 55 (dec 2023), 69 pages. https://doi.org/10.1145/3631974

[193] Quanjun Zhang, Chunrong Fang, Bowen Yu, Weisong Sun, Tongke Zhang, and Zhenyu Chen. 2023. Pre-Trained Model-Based Automated Software Vulnerability Repair: How Far are We? *IEEE Transactions on Dependable and Secure Computing* (2023), 1–18. https://doi.org/10.1109/TDSC.2023.3308897

[194] Yuntong Zhang, Xiang Gao, Gregory J Duck, and Abhik Roychoudhury. 2022. Program vulnerability repair via inductive inference. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*. 691–702.

[195] Ying Zhang, Ya Xiao, Md Mahir Asef Kabir, Danfeng Yao, and Na Meng. 2022. Example-based vulnerability detection and repair in java code. In *Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension*. 190–201.

[196] Yicheng Zhang, Rozhin Yasaei, Hao Chen, Zhou Li, and Mohammad Abdullah Al Faruque. 2021. Stealing Neural Network Structure Through Remote FPGA Side-Channel Analysis. *IEEE Transactions on Information Forensics and Security* 16 (2021), 4377–4388. https://doi.org/10.1109/TIFS.2021.3106169

[197] Yunhui Zheng and Xiangyu Zhang. 2013. Path sensitive static analysis of web applications for remote code execution vulnerability detection. In *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 652–661.

[198] Xin Zhou, Sicong Cao, Xiaobing Sun, and David Lo. 2024. Large Language Model for Vulnerability Detection and Repair: Literature Review and Roadmap. *arXiv preprint arXiv:2404.02525* (2024).

[199] Yanqi Zhou, Tao Lei, Hanxiao Liu, Nan Du, Yanping Huang, Vincent Zhao, Andrew M Dai, Quoc V Le, James Laudon, et al. 2022. Mixture-of-experts with expert choice routing. *Advances in Neural Information Processing Systems* 35 (2022), 7103–7114.

Appendix A

Cross-Domain Security

This section contains a discussion regarding work on cross-domain security for Cyber-Physical Systems completed at the same time as the other works presented in this thesis.

The interconnectedness of the growing digital economy has spurred radical growth in sensor integration. Now, thousands of sensors are easily accessible in a plethora of widespread consumer devices. However, attacks on the computer systems that control these sensors are effective at stealing privileged information [10, 24, 40, 52, 62, 77, 84, 98]. With the further integration of these technologies into consumer-facing products, the attack surface is poised to expand significantly.

One domain where side-channel attacks have been particularly effective is with Additive Manufacturing systems. For example, it has been demonstrated that sensitive Intellectual Property (IP) can be extracted using only the acoustic emanations of a standard 3D printer [4, 27]. Even potentially sensitive information such as a user's IP address has been leaked through similar attacks [26]. However, methods to mitigate the stealing of proprietary IP have been implemented as well [189].

Additive Manufacturing is not the only domain where side-channel attacks are effective. It has been shown that Neural Networks are vulnerable to information leakage as well. By exploiting context switching between users of a shared GPU, attackers have extracted hyperparameters and layer composition from common Deep Learning models [171]. Even remote FPGAs are vulnerable to these types of attacks, where sensitive model information is leaked by an intelligent adversary [196]. Other systems at risk to data extraction include automotive wireless networks [165], DNA synthesis machines [38], and solar inverters [12].

As a concluding remark, a selected example of my research in security for Cyber-Physical Systems is presented. The distribution of phoneme frequency versus perceived loudness in Figure A.1 is described by otolaryngologists and speech pathologists [106] as a *Speech Banana*.

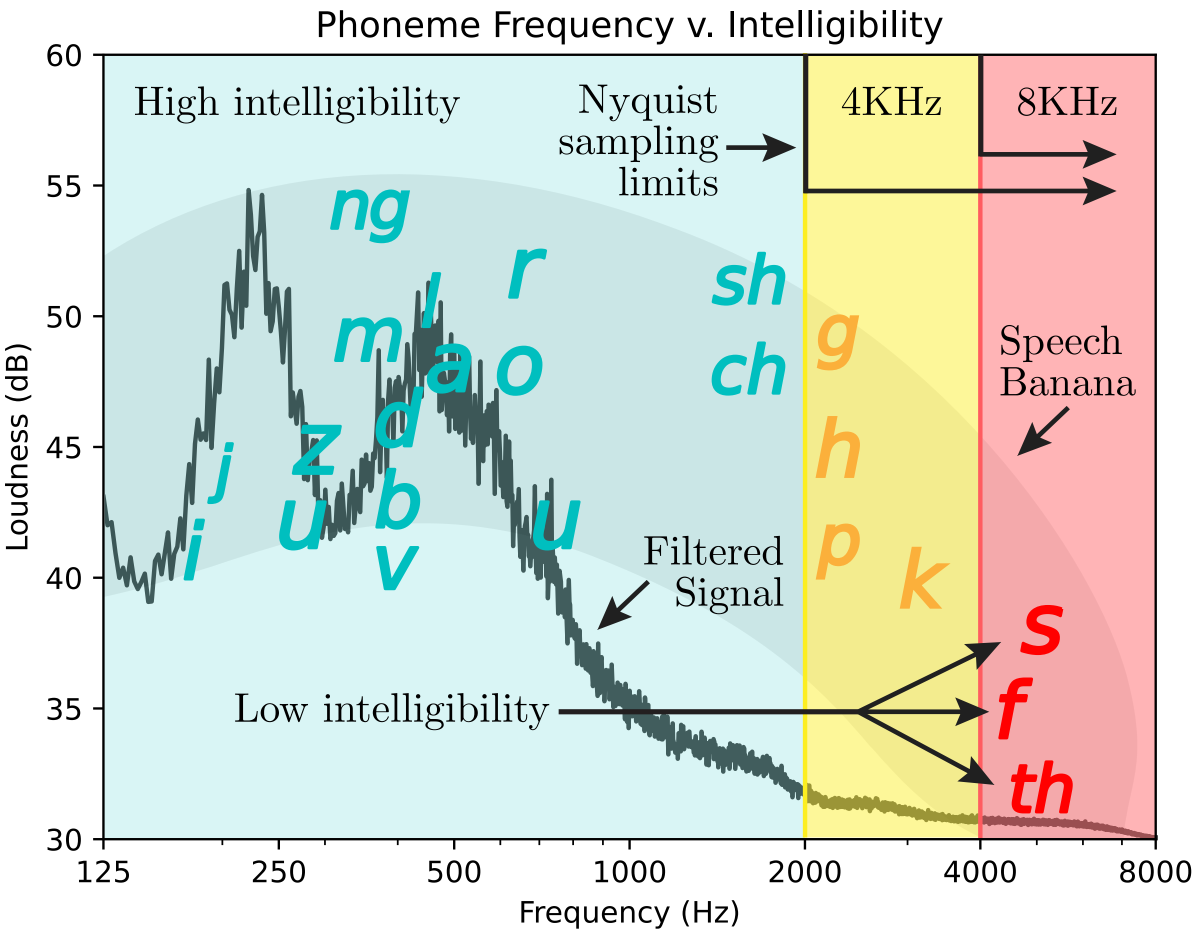

Figure A.1: The *Speech Banana*