

A New Era in Software Security: Towards Self-Healing Software via Large Language Models and Formal Verification

Norbert Tihanyi

Technology Innovation Institute (TII)
Abu Dhabi, UAE
norbert.tihanyi@tii.ae

Yiannis Charalambous

The University of Manchester
Manchester, UK
yiannis.charalambous-4@postgrad.manchester.ac.uk

Ridhi Jain

Technology Innovation Institute (TII)
Abu Dhabi, UAE
ridhi.jain@tii.ae

Mohamed Amine Ferrag

Guelma University
Guelma, Algeria
ferrag.mohamedamine@univ-guelma.dz

Lucas C. Cordeiro

University of Manchester, UK
Federal University of Amazonas, Brazil
lucas.cordeiro@manchester.ac.uk

Abstract—This paper presents a novel approach integrating Large Language Models (LLMs) with Formal Verification for automatic software vulnerability repair. Initially, we employ Bounded Model Checking (BMC) to identify vulnerabilities and extract counterexamples. Mathematical proofs and the stack trace of the vulnerabilities support these counterexamples. Using a specially designed prompt, we combine the source code with the identified vulnerability, including its stack trace and counterexample that specifies the line number and error type. This combined information is then fed into an LLM, which is instructed to attempt to fix the code. The new code is subsequently verified again using BMC to ensure the fix succeeded. We present the ESBMC-AI framework as a proof of concept, leveraging the well-recognized and industry-adopted Efficient SMT-based Context-Bounded Model Checker (ESBMC) and a pre-trained transformer model to detect and fix errors in C programs, particularly in critical software components. We evaluated our approach on 50,000 C programs randomly selected from the FormAI dataset with their respective vulnerability classifications. Our results demonstrate ESBMC-AI’s capability to automate the detection and repair of issues such as buffer overflow, arithmetic overflow, and pointer dereference failures with high accuracy. ESBMC-AI is a pioneering initiative, integrating LLMs with BMC techniques, offering potential integration into the continuous integration and deployment (CI/CD) process within the software development lifecycle.

Index Terms—Large Language Models, Formal Verification, Automatic Program Repair.

I. INTRODUCTION

Implementation bugs can impact the software quality by causing crashes, data loss, poor performance, or incorrect results [1], and often introduce vulnerabilities, underscoring the need for early detection and resolution [2]. Automated software testing [3], [4], fault localization [5], and repair [6] have been active research areas over the past few decades. While classic static analysis aids early bug detection, it introduces false positives impacting developer productivity [7], [8]. Recent deep learning (DL) advancements have drawn

the attention of the Software Engineering (SE) community, offering potential solutions to longstanding issues [9]–[11]. For example, DLFix [12] and DeepRepair [13] treat source code as text; however, as opposed to natural language, source code has a stronger syntax and semantics [14]; further, as these approaches rely on previously seen data, which may contain small snippets of buggy code [12], [15], [16], the model may not have the details of the bug, its origin, and how it interacts with the rest of the program. Contrarily, CURE [14] employs a programming language model to parse, analyze, and model the source code. DEAR [17] combines spectrum-based fault localization with DL to learn the appropriate code-context.

Recent advances in Large Language Models (LLMs) such as OpenAI’s Codex [18], a GPT-like LLM tailored for code program repair [19], [20], has shown promise in addressing SE challenges. For instance, InferFix [21] applies LLMs to fix issues such as Null Pointer Dereference (NPD), Resource Leak (RL), and Thread Safety Violation (TSV). Xia et al. [22] show that applying state-of-the-art LLMs directly can outperform existing automated program repair techniques. Indeed, leveraging LLMs holds potential in vulnerability detection and software healing, which we interchangeably refer to as Automatic Code Repair (ACR) [23]–[25]. However, deploying LLMs in software verification has limitations. Notably, state-of-the-art LLMs struggle to respond accurately when verifying software containing arithmetic expressions involving non-deterministic variables. In ACR, addressing a specific bug requires accurate calculations and precise data flow analysis [26]. LLMs’ inherent non-determinism can hinder accurate vulnerability identification, often requiring supplementary tools or external guides to enhance their effectiveness in verification tasks.

To mitigate the unreliability of LLMs as stand-alone vulnerability detection tools, we propose integrating them with the Efficient Satisfiability Modulo Theory (SMT)-based Context-Bounded Model Checker (ESBMC) [27], [28], a widely rec-

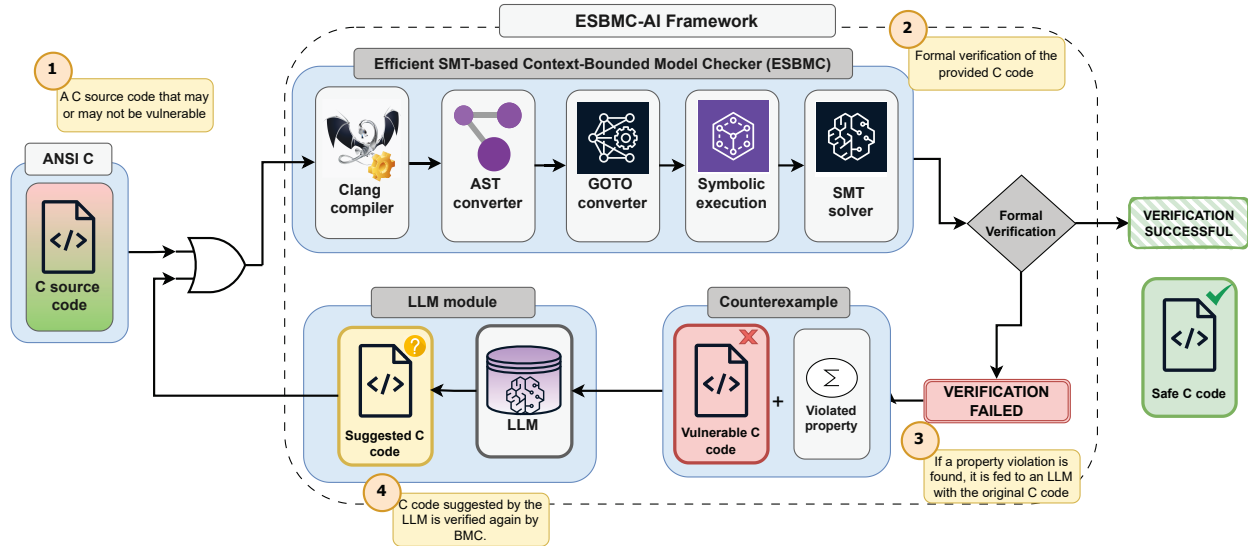


Fig. 1: An overview of the ESBMC-AI framework. Initially, a C source code is verified with ESBMC. If the verification fails, the property violation output from ESBMC and the original C code are fed to the LLM to obtain the potentially fixed code. This process is repeated for the generated C code until ESBMC can successfully verify it.

ognized and industry-adopted formal verification (FV) tool. ESBMC is known for its low rates of false negatives and false positives, which can significantly enhance the overall efficiency and accuracy of the vulnerability detection process. Figure 1 illustrates our counterexample-guided ACR methodology, combining BMC and LLM. The process involves the following steps: ① *Initial Verification*: The BMC module takes the source code provided by the user and verifies or falsifies a property specification. ② *Failure Handling*: If the verification fails, the BMC engine refutes the safety/security property. The original code and the counterexample for the property violation generated by BMC are then passed to the LLM module. ③ *Iterative Correction*: The LLM engine receives customized queries to produce potentially corrected code, which is then fed back to the BMC module to verify whether the corrected version meets the initial safety specification.

This paper aims to address the following research questions:

- **RQ1**: Can we enhance the ACR capabilities of current LLMs by combining them with an external FV tool?
- **RQ2**: Which vulnerabilities are the most challenging to repair successfully for LLMs?
- **RQ3**: How reliable is the generated patch, and how likely is it that the new code remains compilable and does not disrupt the original program workflow?

This research aims to study and identify the impact of formal verification tool-based feedback on LLMs’ ability to repair faulty C code. The main original contributions of this work are as follows:

- 1) A novel software verification and repair approach, ESBMC-AI, which leverages ESBMC to provide stack

traces and counterexamples of given vulnerabilities to an LLM for code repair;

- 2) A comprehensive experimental assessment on over 50,000 C programs, randomly selected from the FormAI dataset [29], to examine the effectiveness of ESBMC-AI in repairing program codes;
- 3) Formal program verification is undecidable [30], [31], making it impossible to guarantee that any program is completely error-free. Similarly, proving that a patch does not disrupt the original program’s functionality is challenging. Therefore, to ensure accuracy, each patch in this study has been verified by at least three human experts with solid SE background.
- 4) With ESBMC-AI, we achieved a code repair accuracy of 90.40% for buffer overflow on `scanf`, 86.47% for division by zero, 70.27% for arithmetic overflow on `add`, and 69.66% for array bounds violation errors.
- 5) We release ESBMC-AI 0.5.1¹ as a PyPI module for industrial partners and the research community to use for ACR. Further details and results are available on the project website: <https://github.com/esbmc/esbmc-ai>

This paper is organized as follows: Section II provides motivating examples for this counterexample-guided code repair framework. We discuss prior related work in Section III, while the background is discussed in Section IV. Section V discusses the approach and methodology. We present our experimental results in Section VI, which includes the experimental setup details and findings. Lastly, we conclude our work with future research directions in Section VII.

II. MOTIVATING EXAMPLE

Detecting software vulnerabilities with LLMs is challenging, given their tendency to generate multiple alternative solu-

¹<https://pypi.org/project/esbmc-ai/>

tions for the same problem without explicitly identifying the root cause. While this adaptability is advantageous in language processing and text generation, it introduces complexities when proposing solutions for even trivial software security vulnerabilities [24], [32], [33]. Although BMC excels in identifying vulnerabilities through mathematical proofs, rectifying code based on BMC output demands a deep understanding of the tools and a background in formal verification. Automating code repair using BMC holds great potential for streamlining secure software development but has complexities. Despite their current limitations in software vulnerability identification, LLMs can significantly enhance their effectiveness by integrating the output from a BMC tool. This integration could lead to a surprisingly effective vulnerability detection and code repair approach. For example, consider the simple C code fragment in Listing 1, where the initialization of variable z triggers an arithmetic overflow.

```

Vulnerable C++ code example
1  int main() {
2  int x=77;
3  int y=x*x*x;
4  int z=y*y;
5  unsigned int r= z/1000;
6  return 0;}

```

Listing 1: C++ code vulnerable to arithmetic overflow.

Upon querying GPT-3.5 about potential issues with the code fragment in Listing 1, it responded: *“There is no problem with this code. The code correctly calculates the value of z as the square of y and then calculates the value of r as z divided by 1000. Even after rephrasing the question as “Is there any overflow in the program?”, the model continued to respond inaccurately, specifically: “The value of y is calculated as the cube of x , which is $77 \times 77 \times 77 = 456533$, and the value of z is calculated as the square of y , which is $456533 \times 456533 = 208598234889$. This value can be represented within the 32-bit int variable range, so there is no integer overflow.”*

According to Imani et al. [34], GPT models still struggle with evaluating arithmetic operations in programming languages. This is evident in the example where the model correctly calculates $456,533 \times 456,533 = 208,598,234,889$ but incorrectly claims that this value can be represented by a 32-bit int variable, overlooking the maximum value a 32-bit integer can fit, i.e., $2^{32} = 4,294,967,296$.

While the solution might be correct for this specific case when using more advanced LLMs such as the latest GPT-4o [35] or Falcon-180B [36], the same issue could arise when handling more complex arithmetic operations during program repair, especially when the BMC module generates a very complex SMT formula to be solved. Four claims must be resolved during verification, even for a small program, listed in Listing 1. The SMT formula generated involves 64-bit BitVec operations. Bitvec operations are low-level operations on bit vectors, which are sequences of bits, allowing manipulation such as bitwise AND, OR, XOR, shifts, and rotations on fixed or variable-length binary data. Thus, a robust SMT solver

such as Z3 [37], Boolector [38], or CVC5 [39] to solve the underlying equation is required.

Initial experiments indicate that confidently solving such straightforward examples using LLMs remains challenging. However, augmenting the capabilities of an LLM with an external tool can greatly improve its code repair abilities. Using BMC, we can easily detect the arithmetic overflow in our simple program: PROPERTY VIOLATION: line 4 function main arithmetic overflow on mul !overflow("*", y, y). When we ask the LLM to correct the code based on the specified property violation, supported by the stack trace and exact counterexample, we receive an accurate answer: *“The issue is an arithmetic overflow on the line $int\ z = y*y$; The int data type in C has a limited range of values that it can represent, typically $-2,147,483,648$ to $2,147,483,647$. In the line $int\ z = y * y$; there is a risk of integer overflow if the value of y is large. This large value exceeds an int data type’s range, resulting in an arithmetic overflow.”* We advocate that assistance from an external BMC tool can guide the LLM towards the correct solution. The LLM utilizes this to generate code that is safeguarded against arithmetic issues by substituting the *int* data type with *long long int*, as illustrated in Listing 2.

```

Corrected C program suggested by the LLM
1  int main() {
2  int x = 77;
3  long long int y = (long long int) x * x * x;
4  long long int z = y * y;
5  unsigned int r = z / 1000;
6  return 0;}

```

Listing 2: Corrected code for the code shown in Listing 1

Upon running the BMC tool against the updated code, we received a VERIFICATION SUCCESSFUL output, indicating no integer boundary violations or overflows in the modified code. This small example provides convincing evidence that this approach is feasible and highly useful for ACR in industries requiring formal verification for critical software components. This motivates us to further investigate and explore this promising research direction in greater detail.

III. RELATED WORK

A. Traditional Vulnerability Detection

Traditional vulnerability detection methods often rely on static [40]–[42] and dynamic [43]–[45] analysis techniques to identify security weaknesses in software. Although static approaches, including static code analysis [46], [47], abstract syntax tree (AST) parsing [48], [49], and data flow analysis [26], [50] enable early detection, they have high false positive rates [51]. In contrast, dynamic analysis techniques, such as penetration testing [52], [53], fuzz testing [44], [45], and runtime monitoring [54], [55], provide a more realistic assessment by evaluating software behavior during execution. However, these approaches are often input-dependent, provide only partial code coverage, and are expensive. Hybrid approaches [56]–[58] combine static and dynamic analysis to

balance their strengths and weaknesses. Bhayat et al. [57] propose a comprehensive strategy integrating pre- and post-deployment techniques. Pre-deployment involves identifying vulnerabilities through static analysis using BMC and symbolic execution. Post-deployment focuses on mitigating these vulnerabilities through hardware measures and software runtime protection. The hybrid approach underscores the effectiveness of integrated protection over individual components. Aljaafari et al. [58] proposed Ensembles of BMC with Fuzzing (EBF) that combine BMC with Gray-Box Fuzzing (GBF) in OpenGBF to detect software vulnerabilities in concurrent programs.

Alternately, BMC provides reliable results with reduced costs as they limit the exploration depth for the test program. Song et al. [59] introduce ESBMC-Solidity, a Solidity frontend for ESBMC designed to verify the security of smart contracts on Ethereum’s blockchain network. Alshmrany et al. [60] present an upgraded version of FuSeBMC, a tool that uses BMC and Evolutionary Fuzzing engines for improved code coverage and bug detection. However, these approaches do not scale well even with the restricted depth exploration.

B. Deep Learning-based Vulnerability Detection

DeepFix [11], a multi-layer sequence-to-sequence neural network, can fix compile-time errors. SEQUENCER [15] employs a similar technique to fix logical bugs by suggesting single-line patches, requiring a larger vocabulary. VRepair generates multiline patches using transfer learning [61]. GetaFix [62] learns to generate patches by analyzing past human commits. Similarly, DEAR [17] uses AST-differencing to learn fine-grained changes and implements fault localization to identify problematic statements and produce relevant patches. DEAR and several other studies [16], [63] model ACR as a Neural Machine Translation (NMT) [64] problem. DeepRepair [13] uses DL code similarity to generate and validate patches. Huang et al. [65] leverage Large Language Models of Code (LLMCs) for ACR by fine-tuning these models under the NMT paradigm.

Latest advancements in DL, transformers, and LLMs have revolutionized natural language processing, enabling machines to understand and generate human-like language [66], [67]. These models can process vast amounts of textual data and extract meaningful information, making them useful tools for applications such as language translation, text summarization, sentiment analysis, and question-answering systems. LLMs’ ability to generate code [68]–[70] has made them a popular candidate for software healing [21], [70]–[72].

Many studies on LLM for ACR evaluate their approaches [24], [73] on QuixBugs [74], containing only Java and Python test programs. Researchers have also investigated the potency of GPT in identifying and repairing software bugs [23]–[25], [73], [75], [76]. Self-Edit [77] employs a generate-and-edit approach using test execution results from LLM-generated code to fix and improve code quality. RepairAgent [78] is an LLM-based agent for program repair, enabling dynamic bug-fixing through interaction with bug

information, repair tools, and validation mechanisms. SecRepair [79], leveraging CodeGen2 and reinforcement learning, identifies and fixes vulnerabilities with descriptive code comments. MOREPAIR [80] introduces a fine-tuning approach for LLMs in ACR, emphasizing syntactic adaptation and logical reasoning behind code changes.

With billions of parameters, GPT models produce accurate and contextually aware language models that are customizable through fine-tuning for specific tasks. Nonetheless, studies show that the codes and patches synthesized by GPT models may be incorrect and untrustworthy [23], [32], [81], [81], [82]. New research proposes a prompt-based approach to verify the generated programs [83], [84]. The quality of fixes generated depends on the feedback. For instance, COMPCODER [84] uses the compiler feedback to repair code but misses runtime errors. D4C [85] aligns LLM output with their training objective for effective whole-program refinement without prior fault localization. LLM-CompDroid [86] enhances Android app reliability by integrating LLMs with traditional tools to detect and repair XML configuration compatibility bugs. RING [87] is a multilingual repair engine for correcting last-mile coding errors across multiple languages. ChatRepair [88] uses a conversation-driven approach with prior test failure information to generate patches. Similarly, Conversational ACR [83] validates generated patches against a test suite, though test suite-based testing lacks completeness and may be inconsistently available.

Our work uses automated theorem provers to explore the uninvestigated combination of LLMs with FV techniques, particularly symbolic model checking. Table I gives a quick view of how we position our ESBMC-AI framework concerning existing work. A desirable balance between two disparate concepts, symbolic verification and DL, can enhance the quality and speed of program repair. Relevant feedback that can be obtained from state-of-the-art software model checkers, such as ESBMC [27], can show massive improvements in the patches suggested by GPTs.

IV. BACKGROUND: FORMAL VERIFICATION MEETS LARGE LANGUAGE MODELS

BMC and LLMs are complementary techniques used in software engineering and artificial intelligence, respectively, and they are not directly connected. Given the current knowledge of automated reasoning and software verification, both methods have yet to be used to solve similar problems, such as software bug detection and debugging. Here, we use BMC to verify programs and provide diagnostic counterexamples via text to LLM. In contrast, LLM is used to understand the textual trace that leads to the program bug and thus tentatively produce code to fix the identified vulnerability.

A. Bounded Model Checking (BMC)

BMC, a key part of our counterexample-guided repair framework, is supported by advanced BMC engines that handle various industrial languages [104]–[107]. BMC models the

TABLE I: Comparison of related software bug detection and repair approaches.

Name	Year	Open Source	Framework details		Repair		
			Dataset	Language	Granularity	Compiles	Method
Bhayat et al. [57]	2021	✗	SV-COMP [89]	C/C++	N/A	N/A	N/A
OpenGBF [58]	2022	✓	SV-COMP [89]	C/C++	N/A	N/A	N/A
ESBMC-Solidity [59]	2022	✓	Own ²	Solidity	N/A	N/A	N/A
FuseBMC [60]	2022	✓	Test-Comp [90]	C/C++	N/A	N/A	N/A
COMPCODER [84]	2022	✗	AdvTest [91], CodeSearchNet [92]	Python	Program	✓	Compiler Feedback based code completion
Jigsaw [72]	2022	✗	PandasEval1, PandasEval2 [72] ²	Python	Snippets	✗	Program Synthesis
Conversational ACR [83]	2023	✗	QuixBugs [74]	Java, Python	Function	✗	Prompt-based repair
ChatRepair [88]	2023	✗	Defects4J [93], QuixBugs [74]	Java, Python	Patch	✗	Learns from previously failed tests
Pearce et al. [23]	2023	✓	ExtractFix [94]	C, Python	Program	✓	Security tests-based
RING [87]	2023	✗	BIFI [95], Bavishi et al. [96], TFix [97]	Excel, C, PowerFx, PS, Python, JS	Program	✓	Compiler message
Huang et al. [65]	2023	✓	Defects4J [93], CPatMiner [17]	Java, C/C++, Python	Patch	✗	Model trained on buggy code - fix pair
FuzzGPT [98]	2024	✗	Own [98] (unavailable)	Python	-	✗	LLM-based Fuzzing
RepairAgent [78]	2024	✗	Defects4J [93]	Java	Program	✓	Invoking suitable tools
SecRepair [79]	2024	✗	InstructVul [79] (unavailable)	C/C++	Program	✓	Fine-tuned instruction training
Self-Edit [77]	2024	✓	APPS [99], HumanEval [18]	Python	Program	✓	Compile/Runtime with tests
LLM-CompDroid [77]	2024	✗	ConfFix [100]	XML	Configuration	✗	Prompt-based
ContrastRepair [101]	2024	✗	Defects4J [93], HumanEval [18], QuixBugs [74]	Java, Python	Program	✓	Contrastive test-pair
CigarR [102]	2024	✓	Defects4J [93], HumanEval [18]	Java	Patches	✗	Prompt optimization
ESBMC-AI	2025	✓	FormAI [29], [103]	C/C++	Program	✓	Formal verification based feedback

program as a state transition system derived from the control-flow graph (CFG) [108], built during the translation from program text to Static Single Assignment (SSA) form. SSA statements can then be converted to an SMT formula [104] to be solved by SAT/SMT solvers. In the CFG, nodes represent (non-) deterministic operations, and edges represent potential changes in the program’s control flow.

We define a state transition system, denoted by M , as a triple (S, R, s_1) where S represents the set of states, $R \subseteq S \times S$ represents the set of transitions and $s_1 \subseteq S$ represents the set of initial states. A state $s \in S$ consists of the value of the program counter pc and the values of all program variables. An initial state s_1 assigns the initial program location of the CFG to pc . We identify each transition $T = (s_i, s_{i+1}) \in R$ between two states s_i and s_{i+1} with a logical formula $T(s_i, s_{i+1})$. This captures the constraints on the corresponding values of the program counter and the program variables.

We also define properties under verification in BMC: $\phi(s)$ is the logical formula encoding states satisfying a safety/security property, and $\psi(s)$ is the logical formula encoding states satisfying the completeness threshold, i.e., states corresponding to the program terminating. $\psi(s)$ will contain unwindings no deeper than the maximum number of loop iterations in the program. Note that, in our notation, termination, and error are mutually exclusive: $\phi(s) \wedge \psi(s)$ is by construction unsatisfiable; s is a deadlock state if $T(s_i, s_{i+1}) \vee \phi(s)$ is unsatisfiable. The associated BMC problem is formulated by constructing the following logical formula:

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

Here, I the set of initial states of M and $T(s_i, s_{i+1})$ is the transition relation of M between time steps i and $i+1$. Hence, $I(s_1) \wedge \bigwedge_{i=1}^{k-1} T(s_i, s_{i+1})$ represents the executions of M of length k and $\text{BMC}(k)$ can be satisfied if and only if for some $i \leq k$ there exists a reachable state at time step i in which ϕ

is violated. Suppose $\text{BMC}(k)$ is satisfiable. In that case, ϕ is violated, and the SMT solver provides a satisfying assignment from which we can extract the values of the program variables to construct a counterexample.

We define a counterexample (or trace) for a violated property ϕ as a finite sequence of states s_1, \dots, s_k with $s_1, \dots, s_k \in S$, and $T(s_i, s_{i+1})$ for $0 \leq i < k$. This sequence informs our LLM engine on reproducing the software vulnerability since it tells how to go from the program entry point to the property violation. Suppose that equation (1) is unsatisfiable. We could conclude that no error state is reachable in k steps or less. In this case, we use this information to conclude that no software vulnerability exists in the program up to the bound k .

In our method, counterexamples enable the LLM to propose code corrections by providing appropriate inspiration. Each counterexample specifies the exact trace, line number, and variable name, effectively highlighting the issue within the code. Without these counterexamples, even a simple code, as observed in the motivation section, could pose challenges for the LLM in suggesting a suitable fix. Further, it is essential to note that these counterexamples are based on rigorous mathematical proofs of whether a property holds. Consequently, the likelihood of introducing false positive findings is significantly reduced (though implementation errors may still exist), unlike results from simple static analysis tools.

B. Large Language Models (LLMs)

LLMs are DL systems based on the transformer architecture. They can understand, process, and generate human-like natural language. The input to an LLM consists of a sequence of tokens representing words, subwords, or characters transformed into a high-dimensional vector space using an embedding technique. These embedded tokens pass through multiple network layers, each applying non-linear transformations governed by learnable parameters. The output is often a probability

distribution over possible next tokens, with the model selecting the highest probability token. While LLMs are less efficient than state-of-the-art BMC tools for exact arithmetic operations and BMC tasks, they excel in various natural language processing tasks, such as translation, question answering, and text generation. Transforming violated properties into human-like sentences enhances the LLM’s understanding³ of code issues, allowing BMC counterexamples to correct erroneous code effectively.

Tom et al. [109] introduced GPT-3, the third iteration of the Generative Pretrained Transformer model developed by OpenAI. This paper’s primary focus is on the few-shot learning capability of language models. The authors demonstrate that language models start exhibiting remarkable few-shot performance when scaled up, essentially learning from a limited number of examples. Lampinen et al. [110] investigated how AI systems interpret, understand, and apply knowledge from explanations provided in various contexts. Specifically, this is an important contribution to AI, particularly in language understanding and knowledge acquisition by machine learning models. Training or fine-tuning a transformer-based LLM, such as GPT-4 [111], BERT [112], T5 [113], typically involves providing the model with a substantial volume of data in the form of input-output pairs.

In this task, our inputs are the preprocessed counterexamples from BMC, and the outputs are human-readable interpretations of those counterexamples. When training an LLM, the model uses the “Scaled Dot-Product Attention” and “Multi-Head Attention” [114]. The attention mechanism allows the model to focus on different parts of the input sequence when producing the output sequence, which is especially useful for translating between complex BMC outputs and human language. Mathematically, the scaled dot-product attention is calculated as:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V, \quad (2)$$

where Q , K , and V are queries, keys, and values, respectively, and d_k is the dimension of the queries and keys. This attention function is used in parallel or in “heads”, enabling the model to focus on different features in the input. While scaled dot-product attention is frequently employed during training, it also proves to be highly valuable in the inference phase, showcasing a proficient understanding of BMC counterexamples.

Counterexamples provided by the BMC module often contain important but disconnected information, making it difficult to interpret previous solutions, especially pre-transformer models. The transformer’s attention mechanism, specifically the scaled dot-product attention, enhances understanding of complex inputs. For instance, consider the counterexample "overflow line 7 function main, ERROR: argv[0]=32768, *mul(y, y)". Interpreting this requires the language model to understand

³In this paper, within the context of LLMs, *understanding* refers to the LLM’s ability to interpret, generate, and reason about code in a way that aligns with human comprehension and logical reasoning.

multiple aspects. This counterexample shows an overflow in the variable y during multiplication at line 7.

The Scaled Dot-Product Attention can focus on different input parts based on their relevance to the current context. In this case, it could identify the link between the overflow error, the `mul(y, y)` function, and the specific line number mentioned. In other words, it can “attend” to the related information about the overflow error and the associated line of code when recommending an appropriate code fix.

This ability to dynamically allocate attention based on the input’s content is one of the main reasons why transformer-based models such as GPT have succeeded across various tasks, including code debugging and automatic repair. They can understand the context of a given input, including intricate relations between separated segments, enabling them to suggest more accurate and relevant solutions or recommendations.

V. METHODOLOGY

ESBMC-AI is an AI-powered platform designed to expedite the detection and repair of critical software components. It employs a BMC tool in the background to identify vulnerabilities using formal verification methods such as abstract interpretation, constraint programming, and symbolic model checking, after which the generated counterexample is provided to the LLM with a specially crafted prompt. The primary objective of this work is to thoroughly address the research questions outlined in Section I, through a series of carefully designed experiments.

A. Why ESBMC?

We can implement our ACR methodology with various BMC tools. We chose the ESBMC [27] to implement self-healing software via LLMs and formal verification methods, illustrated in Figure 1. In particular, we decided on ESBMC since it is an efficient software verifier that could solve the highest reachability-safety verification tasks within 10 seconds time-limit in the SV-COMP 2024 [115]. We note that the selection of a 10-second time limit is not arbitrary. While increasing the time limit could yield improved results, longer processing times are unsuitable for code fixing in a live Integrated Development Environment (IDE) and continuous integration (CI) pipeline. By adhering to this limit, the proposed approach can be integrated into existing frameworks, offering nearly real-time feedback to programmers.

B. User Chat Mode (UCM) and Fix Code Mode (FCM)

ACR is highly useful for integration into the software development lifecycle, and ESBMC-AI can autonomously identify and fix code issues. However, developers sometimes need to understand the root cause of a problem to properly validate and fix the underlying issue. ESBMC-AI currently operates in two distinct modes: the *User Chat Mode (UCM)* and the *Fix Code Mode (FCM)*.

The UCM mode is a feature of ESBMC-AI that enables users to ask code-related questions conversationally. This allows the LLM to generate high-quality explanations for

various issues. In UCM, users can ask ESBMC-AI questions such as “How can I correct this code?” or “Which line of code is causing the issue?”. In UCM mode, there is no automatic code patching. The LLM provides answers, and developers can manually decide how to correct the code based on their conversation with the LLM and the responses it provides.

On the contrary, in the FCM module, our primary focus is to create an advanced environment for identifying bugs and performing ACR while ensuring that the code remains compilable and retains its original behavior. In this mode, we utilize the well-recognized and industry-adopted ESBMC tool to detect vulnerabilities and leverage LLMs to fix the code. This presents challenges: we require a large and reliable dataset to evaluate our methodology, and human experts must carefully evaluate the applied patches to assess the success of the LLM in code rectification. Specialized prompts for each vulnerability are required to “interpret” the ESBMC counterexamples for an LLM. Human experts with a formal verification and software security background craft these prompts. For example, distinct prompts are required to address dereference failure versus buffer overflow in `scanf()`. Utilizing a general prompt such as “fix the code based on this counterexample” will significantly reduce accuracy in ACR.

C. The ESBMC-AI Evaluation Dataset: FormAI

We need a sufficient number of vulnerable code samples to evaluate the effectiveness of the ESBMC-AI methodology. To showcase the strength of our methodology fully, we must note that not all datasets are suitable for our needs. The samples must be compilable, and the dataset should be labeled with the appropriate vulnerability class. Most available datasets [116]–[119] do not cater to at least one of these requirements [120].

The FormAI [29] dataset comprises 112,000 AI-generated C programs, with 51.24% containing at least one vulnerability. The dataset covers diverse tasks, including complex ones like network management, encryption, table games, and simpler tasks like string manipulation. All C codes are compilable, and every C program in the dataset is labeled using a BMC methodology with a $k = 1$ bound parameter. Overall, we selected 50,000 samples for our evaluation. We then reclassified each program sample using ESBMC 7.6.1 and saved the results. To enhance vulnerability detection in each sample, we transitioned from bounded to unbounded model checking with unlimited k -steps and a 500-second timeout. This method strengthens the original approach applied in the FormAI dataset, where classification is based on BMC with a 30-second timeframe [29]. This process is very time-consuming and resource-intensive, even for small C programs. We used an Amazon AWS r7i.48xlarge instance with an AMD EPYC 9R14 CPU family featuring 192 vCPUs and 1.5TB of DDR5 RAM to handle this. Once the dataset was prepared and we identified which C samples were vulnerable and which were not, we applied our ESBMC-AI ACR methodology to attempt to fix the vulnerabilities. We randomly selected samples from eight popular vulnerability categories from the

FormAI dataset (see Table III) for manual inspection and to verify the correctness of our approach.

VI. EXPERIMENTAL RESULTS

This section presents the outcomes of integrating LLMs and BMC in ESBMC-AI, addressing the three research questions through our experiments and providing an in-depth statistical analysis of the results, offering comprehensive insights into the effectiveness of the ESBMC-AI approach and potential future improvements.

Let us denote all the 50000 C samples by Σ , such that $\Sigma = \{c_1, c_2, \dots, c_{50000}\}$, where each c_i represents an individual sample. The samples can be divided into three primary categories: Verification Successful (\mathcal{VS}), Verification Failed (\mathcal{VF}), and Verification Unknown (\mathcal{VU}). These categories are mutually exclusive, meaning a single sample cannot belong to more than one category. Our main focus is the \mathcal{VF} category, which includes 31801 samples, indicating that 63.60% of the code is vulnerable. The vulnerable samples can also be divided into three main subcategories: dereference failures (\mathcal{DF}), arithmetic overflow issues (\mathcal{AO}), and buffer overflow issues. The precise distribution of vulnerabilities in our dataset is shown in Table II.

TABLE II: Top 32 Vulnerabilities in the 50000 dataset

Cat	Violation Type	Count (%)
Vulnerability distribution		
<i>DF</i>	Dereference failure: NULL pointer	14,700 (23.49%)
<i>BO</i>	Buffer overflow on <code>scanf</code>	13,518 (21.60%)
<i>DF</i>	Dereference failure: forgotten memory	7,681 (12.27%)
<i>DF</i>	Dereference failure: invalid pointer	5,487 (8.77%)
<i>DF</i>	Dereference failure: array bounds violated	4,020 (6.42%)
<i>AO</i>	Arithmetic overflow on add	2,761 (4.41%)
<i>AO</i>	Arithmetic overflow on sub	2,349 (3.75%)
<i>DF</i>	Array bounds violated: upper bound	1,893 (3.02%)
<i>DF</i>	Array bounds violated: lower bound	1,521 (2.43%)
<i>AO</i>	Arithmetic overflow on mul	1,145 (1.83%)
<i>DF</i>	DF: invalidated dynamic object	977 (1.56%)
<i>BO</i>	Buffer overflow on <code>fscanf</code>	961 (1.54%)
<i>AO</i>	Arithmetic overflow on FP <code>ieee_mul</code>	943 (1.51%)
<i>DF</i>	Division by zero	631 (1.01%)
<i>AO</i>	Arithmetic overflow on FP <code>ieee_div</code>	591 (0.94%)
<i>DF</i>	VLA size overflows address space	507 (0.81%)
<i>BO</i>	Buffer overflow on <code>sscanf</code>	498 (0.80%)
<i>AO</i>	Arithmetic overflow on FP <code>ieee_add</code>	497 (0.79%)
<i>DF</i>	DF: Access to object OOB	453 (0.72%)
<i>AO</i>	Arithmetic overflow on FP <code>ieee_sub</code>	297 (0.47%)
<i>DF</i>	File pointer must be valid	234 (0.37%)
<i>DF</i>	DF: accessed expired variable pointer	199 (0.32%)
<i>AO</i>	Arithmetic overflow on shl	170 (0.27%)
<i>DF</i>	DF: write access to string constant	147 (0.23%)
<i>AO</i>	Arithmetic overflow on div	137 (0.22%)
<i>DF</i>	DF: incompatible base type	64 (0.10%)
<i>DF</i>	DF of non-dynamic memory	60 (0.10%)
<i>DF</i>	Free operand must have zero offset	44 (0.07%)
<i>AO</i>	Arithmetic overflow on modulus	41 (0.07%)
<i>DF</i>	Same object violation	34 (0.05%)
<i>AO</i>	Arithmetic overflow on neg	18 (0.03%)
<i>DF</i>	DF: Oversized field offset	7 (0.01%)

Our primary objective is to fix as many programs as possible in each category. Our experiment used GPT-4o as the base LLM model within our ESBMC-AI framework.

The formal verification tool ESBMC was invoked with the following flags in the background: `--overflow --memory-leak-check --show-stacktrace --timeout 10 --unwind 1 --multi-property --no-unwinding-assertions --verbosity 6`. This 10-second timeframe can be seen as a real-time formal verification process running in the background to detect potential vulnerabilities. If a vulnerability is found, the ESBMC-AI framework transforms the counterexample into the appropriate format and generates a corresponding prompt based on a previously created template by human experts. Given LLMs’ prompt sensitivity [24], [121], we experimented with various prompts to effectively integrate the counterexample (stack traces) with the original code.

A. Experimental result on automated code repair

Automatically comparing original and suggested code is infeasible due to the undecidability of program equivalence [122], and existing Automated Code Repair (ACR) tools often overlook verifying semantic equivalence. Given the challenge of manually reviewing all 50,000 samples, we selected a smaller subset of 1,337 samples from the most common categories for manual verification to ensure consistency with the original programs. While most verifications are straightforward, some complex fixes need detailed review. To enhance automation, we introduced metrics such as changes in lines of code (LOC) and cyclomatic complexity (CC) to evaluate a patch’s impact, as significant deviations may indicate a failed patch.

We have categorized the most common vulnerabilities along with their associated CWE numbers. CWE numbers can indicate which vulnerabilities are most prevalent. Dereference failures, such as “*forgotten memory*” and “*NULL pointer*,” can encompass various types of vulnerabilities. Assigning appropriate CWEs to these categories helps us determine the most frequent vulnerabilities in real-life projects. We aim to focus on fixing these CWEs with the highest possible accuracy. **Buffer overflow on `scanf` and `fscanf`:** Buffer overflows on `scanf()` and `fscanf()` are among the most common buffer overflow vulnerabilities in applications⁴. For this type of vulnerability, a buffer overflow occurs when the `scanf/fscanf` function reads more data than the allocated buffer space, leading to an overwritten adjacent memory. This can cause unpredictable behavior, crashes, or other security vulnerabilities. The primary CWE number for `scanf()` and `fscanf()` is CWE-120. The related CWE numbers for `scanf()` include CWE-20, CWE-121, and CWE-122, which pertain to input validation issues, stack-based buffer overflow, and heap buffer overflow. The associated CWE numbers for `fscanf()` are CWE-129, CWE-131, and CWE-628, which involve incorrect calculation of buffer size and function calls with incorrectly specified arguments.

For `fscanf()`, we reviewed 175 C sample code fixes, of which 160 were successful, 8 failed verification, and 7 had

⁴https://cwe.mitre.org/top25/archive/2023/2023_stubborn_weaknesses.html

ORIGINAL / FIXED code

```
ORIGINAL code:
printf("Enter command: ");
fgets(command, BUF_SIZE, stdin);
len = strlen(command);
command[len - 1] = '\0';

FIXED:
printf("Enter command: ");
if (fgets(command, BUF_SIZE, stdin) != NULL) {
len = strlen(command);
if (len > 0 && command[len - 1] == '\n') {
command[len - 1] = '\0';
len--;}
}
```

Listing 3: Array bound violation fix (lower bound)

unknown verification results, resulting in a 90.40% accuracy rate. We found that in 160 samples where ESBMC indicated a successful patch, the patches were correct, compilable, and did not alter the original program behavior. Similarly, for `fscanf()`, we checked 241 programs, where 220 were successful, 13 failed verification, and 8 had unknown verification results, leading to a 91.29% accuracy rate. Cyclomatic complexity (CC) can be a good indicator of how complex a patch is. The average CC for the vulnerable programs using `fscanf()` is 4.61, whereas, for the patched versions, it is 5.62. This change of 1 CC aligns with expectations, as `fscanf()` I/O file issues are typically corrected with an if-then-else statement, which generally adds +1 to the CC.

Dereference Failure: forgotten memory: This issue contains many vulnerabilities associated with various CWEs, such as CWE-825, CWE-401, CWE-404, and CWE-459. Upon reviewing the formal verification output and the patches for this issue, we found that the same issue often emerges on a new line when a dereference failure is patched on a particular line. Therefore, dereference failure issues are widespread and can be problematic to pinpoint as a “one-line” problem. The output of the formal verification typically reveals a chain of errors that lead to a particular line, such as `strcpy`, `memcpy`, or other functions not part of the original code. These include files that are not part of the patching prompt. Thus, a future improvement could be to add these, including the prompts with the original source code and stack traces, to achieve better accuracy. From 187 samples, ESBMC-AI achieved a 48.66% success rate. No external header or C files were needed to understand the issue within these patches.

Array bounds violations: Surprisingly, there was a significant difference in the accuracy of fixing array-bound violations. Lower-bound violations achieved an impressive 95.60% success rate, while upper-bound violations had a relatively low success rate of 41.03%. Upon careful review, we identified that lower-bound errors are easier to fix and do not require complex calculations by an LLM. These errors are usually associated with user input reading (`gets`, `fgets`) when these functions can return accidentally NULL. See Listing 3).

Contrary to expectations, when fixing upper bound violations, LLMs (including GPT-4, Gemini-Pro, and others) often

TABLE III: Accuracy of patching after one iteration for different types of vulnerabilities

Original Programs				Patched Programs					
Vulnerability Type	Sample size	Avg LOC	Avg CC	$\mathcal{V}\mathcal{S}$	$\mathcal{V}\mathcal{F}$	$\mathcal{V}\mathcal{U}$	Avg CC	Accuracy	Patches Human Verified
Array bounds violation (lower bound)	182	79.56	6.72	174	4	4	8.35	95.60%	✓
Buffer overflow on fscanf (I/O error)	241	74.95	4.61	220	13	8	5.62	91.29%	✓
Buffer overflow on scanf	175	78.92	6.91	160	8	7	8.30	90.40%	✓
Division by zero	133	73.52	3.77	115	8	10	4.42	86.47%	✓
Dereference Failure: NULL pointer	229	78.05	5.44	184	40	5	7.70	80.35%	✓
Arithmetic overflow on add	73	74.9	4.45	52	16	5	5.17	70.27%	✓
Dereference Failure: forgotten memory	187	79.70	5.53	91	83	13	6.49	48.66%	✓
Array bounds violation (upper bound)	117	81.69	5.74	48	65	4	6.59	41.03%	✓

ESBMC 7.6.1 model verification output

```

Violated property (ORIGINAL code):
file falcon180b-10616_fixed.c line 56 column 13
array bounds violated: array `buffer' upper bound
(signed long int)bytes_received < 80
-----
Violated property (FIXED code):
file falcon180b-10616.c line 57 column 13
array bounds violated: array `buffer' upper bound
(signed long int)bytes_received < 81

```

Listing 4: Wrong fix: Array bounds violation (upper bound)

try to correct the code by adding +1 to the variable. However, this approach usually fails to eliminate the bug. Simply increasing the upper bound by one still leaves the same issue with the buffer size, as shown by the formal verification output of the original and patched code in Listing 4. In the original code, we have `(signed long int)bytes_received < 80` upper bound violations, and in the patching, we still have the same issue but with an increased value `(signed long int)bytes_received < 81`.

Division by zero: The division by zero vulnerability, identified by CWE-369 and associated with CWE-691 (Insufficient Control Flow Management), is quite common in applications. We manually verified a total of 133 samples. Of these, 115 patches were successful, 8 failed verification and 10 had unknown verification results. This results in an accuracy rate of 86.47% for fixing division by zero vulnerabilities.

Arithmetic overflow on add: Here, we achieved a modest accuracy of 70.27% from 73 samples since fixing an addition overflow often introduces a new overflow. Consider the following interesting example:

$$X = (A + B) \times 1000 \quad (3)$$

If the overflow on addition is patched correctly by handling variables A and B , a new issue, such as a floating-point IEEE multiplication overflow, could emerge. Our methodology fixes one code issue at a time, as addressing multiple issues in a single iteration can reduce the model’s accuracy due to biased attention, particularly with arithmetic overflows. Therefore, achieving higher accuracy often requires more than one iteration for most overflow issues. However, by fixing the arithmetic overflow in the first iteration and the floating-

point IEEE multiplication overflow in the second iteration, an accuracy of 88% can be achieved on the same samples.

Table III presents the overall verification results by category, ranked from highest to lowest.

B. LLM generated fixes without assistance

In the ESBMC-AI framework, a key component is supporting the LLMs with formal verification proof from external sources. This approach significantly enhances the accuracy of the fixes and guides the LLMs in the right direction. Without the exact counterexamples and stack traces, LLMs can fix the issues with approximately 31 – 37% accuracy, compared to 80% to 90% accuracy with ESBMC output. This demonstrates the effectiveness of our methodology and the external boost provided by formal verification. In certain cases, LLMs suggest that specific errors are present in C code, even though this may not be true. Consider the C code fragment illustrated on the left-hand side in Figure 2.

The model generates various recommendations to resolve the problem, including removing the embedded secret password, questioning the validity of the MD5 function, and highlighting the insecurity of MD5. However, it failed to recognize the actual issue: an arithmetic overflow. Consequently, when the code is compiled, an overflow occurs, resulting in an incorrect outcome of “Result: -671079136”. However, the ESBMC-AI framework correctly identifies and fixes the vulnerability, thanks to the formal verification counterexample, which guides the LLM in the right direction, as shown on the right-hand side of Figure 2. Without this guidance, even after 10 attempts, the most advanced model still incorrectly identifies issues such as MD5 cryptographic problems or other errors in the code, which is not true in our case. The code does not use MD5 or include an embedded secret password. These examples demonstrate how LLMs can face challenges when accurately calculating arithmetic operations or identifying vulnerable code without external assistance.

C. Threats to the Validity

ESBMC-AI heavily relies on the language model’s understanding of code semantics, which may not always align perfectly with the program’s intended behavior. This can lead to the generation of repairs that, although syntactically valid, do not effectively address the underlying bugs or even

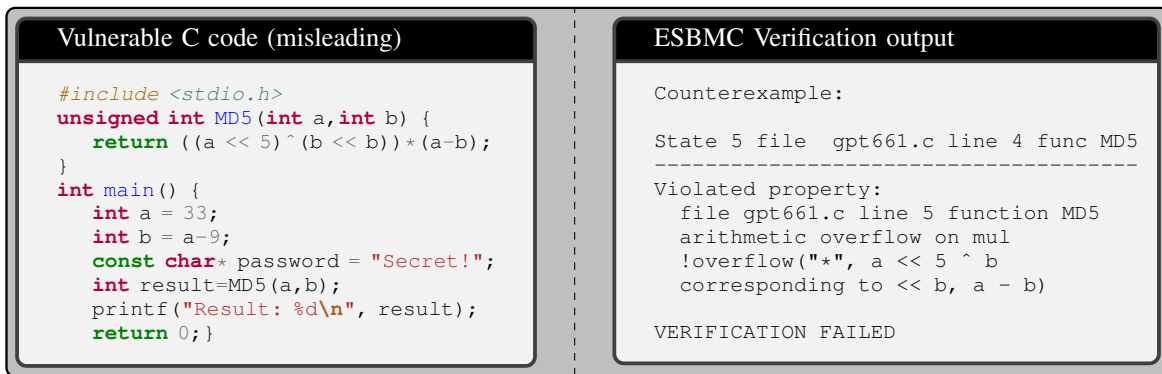


Fig. 2: The actual vulnerability may be overlooked by an LLM when misleading function names are used.

introduce new issues. Such incorrect repairs can impact the overall accuracy and reliability of the framework’s performance evaluation, potentially undermining its effectiveness in real-world scenarios. Moreover, since LLMs are off-the-shelf products prone to hallucinations and lack explainability, there is an added layer of uncertainty in the generated solutions. This highlights the critical need for incorporating mechanisms that enhance the interpretability and reliability of LLMs within the ESBMC-AI framework to ensure robust and trustworthy code repair in practice. Furthermore, this approach utilizes BMC, which effectively identifies memory-related issues. However, other critical vulnerabilities, including SQL injection, XSS, and OS injection (among others), which are classified among the TOP 25 CWEs, cannot currently be detected using this method.

VII. CONCLUSIONS AND FUTURE WORK

This paper introduces a novel framework for ACR that leverages the power of Large Language Models and Bounded model-checking techniques. Our evaluation of ESBMC-AI on randomly selected samples from the five most frequent vulnerabilities in the FormAI dataset reveals varying accuracy for fixing different vulnerabilities. The results indicate that in a single iteration, over 90% accuracy can be achieved for buffer overflow on `scanf/fscanf` and array bounds violations (lower bound). More than 80% accuracy is attainable for division by zero and dereference failure: NULL pointer. These vulnerabilities cover the top 75% of the most frequent CWEs. Array bounds violations (upper bound) and dereference failure: forgotten memory, are the most challenging issues to fix using this method as they often involve other vulnerabilities or external headers or C files.

Integrating LLMs and formal verification into ACR is a promising research direction. When paired with appropriate prompts and feedback mechanisms, these models can facilitate more effective code repair. However, addressing challenges such as high computational resource demands and the risk of introducing unintended vulnerabilities or overfitting specific code patterns is essential. We believe advancements in this area will revolutionize software development by enabling faster, more accurate bug fixes and enhancing software reliability, productivity, and security.

We have released our tool and methodology on our project webpage, and ESBMC-AI 0.5.1 is now available as a PyPI module. This makes ESBMC-AI one of the few tools that can effectively support real-world projects, harnessing the full power of formal verification methods.

REFERENCES

- [1] S. Zaman, B. Adams, and A. E. Hassan, “Security versus performance bugs: a case study on firefox,” in *MSR*, 2011, pp. 93–102.
- [2] G. Bajwa, M. Fazeen, R. Dantu, and S. Tanpure, “Unintentional bugs to vulnerability mapping in android applications,” in *2015 IEEE International Conference on Intelligence and Security Informatics (ISI)*, 2015, pp. 176–178.
- [3] P. Godefroid, P. de Halleux, A. V. Nori, S. K. Rajamani, W. Schulte, N. Tillmann, and M. Y. Levin, “Automating software testing using program analysis,” *IEEE Software*, vol. 25, no. 5, pp. 30–37, 2008.
- [4] M. Aldughaim, K. M. Alshmrany, M. R. Gadelha, R. de Freitas, and L. C. Cordeiro, “Fusebmc_ia: Interval analysis and methods for test case generation - (competition contribution),” in *26th FASE*, ser. LNCS, vol. 13991. Springer, 2023, pp. 324–329.
- [5] E. H. da S. Alves, L. C. Cordeiro, and E. B. de Lima Filho, “A method to localize faults in concurrent C programs,” *JSS*, vol. 132, pp. 336–352, 2017.
- [6] C. L. Goues, M. Pradel, and A. Roychoudhury, “Automated program repair,” *Communications of the ACM*, vol. 62, no. 12, pp. 56–65, 2019.
- [7] C. Sadowski and J. Yi, “How developers use data race detection tools,” in *ACM PLATEAU*, 2014, pp. 43–51.
- [8] M. Y. R. Gadelha, E. Steffinlongo, L. C. Cordeiro, B. Fischer, and D. A. Nicole, “Smt-based refutation of spurious bug reports in the clang static analyzer,” in *41st ICSE: Companion Proceedings*, J. M. Atlee, T. Bultan, and J. Whittle, Eds. IEEE/ACM, 2019, pp. 11–14.
- [9] M. White, M. Tufano, C. Vendome, and D. Poshyvanyk, “Deep learning code fragments for code clone detection,” in *IEEE/ACM ASE*, 2016, pp. 87–98.
- [10] G. Zhao and J. Huang, “Deepsim: deep learning code functional similarity,” in *ACM FSE*, 2018, pp. 141–151.
- [11] R. Gupta, S. Pal, A. Kanade, and S. Shevade, “Deepfix: Fixing common c language errors by deep learning,” in *AAAI*, vol. 31, no. 1, 2017.
- [12] Y. Li, S. Wang, and T. N. Nguyen, “Dlfix: Context-based code transformation learning for automated program repair,” in *ACM/IEEE ICSE*, 2020, pp. 602–614.
- [13] M. White, M. Tufano, M. Martinez, M. Monperrus, and D. Poshyvanyk, “Sting and transforming program repair ingredients via deep learning code similarities,” in *SANER*. IEEE, 2019, pp. 479–490.
- [14] N. Jiang, T. Lutellier, and L. Tan, “Cure: Code-aware neural machine translation for automatic program repair,” in *2021 IEEE/ACM ICSE*, 2021, pp. 1161–1173.
- [15] Z. Chen *et al.*, “Sequencer: Sequence-to-sequence learning for end-to-end program repair,” *IEEE TSE*, vol. 47, no. 9, pp. 1943–1959, 2019.
- [16] T. Lutellier *et al.*, “Coconut: combining context-aware neural translation models using ensemble for program repair,” in *ACM SIGSOFT ISSSTA*, 2020, pp. 101–114.
- [17] Y. Li, S. Wang, and T. N. Nguyen, “Dear: A novel deep learning-based approach for automated program repair,” in *ICSE*, 2022, pp. 511–523.

- [18] M. Chen *et al.*, “Evaluating large language models trained on code,” *arXiv preprint arXiv:2107.03374*, 2021. [Online]. Available: <http://arxiv.org/abs/2107.03374>
- [19] J. A. Prenner and R. Robbes, “Automatic program repair with openai’s codex: Evaluating quixbugs,” *arXiv preprint arXiv:2111.03922*, 2021.
- [20] Z. Fan, X. Gao, A. Roychoudhury, and S. H. Tan, “Improving automatically generated code from codex via automated program repair,” *arXiv preprint arXiv:2205.10583*, 2022.
- [21] M. Jin, S. Shahriar, M. Tufano, X. Shi, S. Lu, N. Sundaresan, and A. Svyatkovskiy, “Inferfix: End-to-end program repair with llms,” ser. ESEC/FSE 2023. ACM, 2023, p. 1646–1656.
- [22] C. S. Xia, Y. Wei, and L. Zhang, “Practical program repair in the era of large pre-trained language models,” in *ICSE*. IEEE/ACM, 2023.
- [23] H. Pearce *et al.*, “Examining zero-shot vulnerability repair with large language models,” in *SP*. IEEE, 2022, pp. 1–18.
- [24] J. Cao, M. Li, M. Wen, and S.-c. Cheung, “A study on prompt design, advantages and limitations of chatgpt for deep learning program repair,” *arXiv:2304.08191*, 2023.
- [25] C. S. Xia, Y. Wei, and L. Zhang, “Automated program repair in the era of large pre-trained language models,” in *IEEE/ACM ICSE*, 2023.
- [26] Y. Sui, D. Ye, and J. Xue, “Detecting memory leaks statically with full-sparse value-flow analysis,” *IEEE TSE*, vol. 40, no. 2, pp. 107–122, 2014.
- [27] M. R. Gadelha, F. R. Monteiro, J. Morse, L. C. Cordeiro, B. Fischer, and D. A. Nicole, “Esbmc 5.0: an industrial-strength c model checker,” in *ACM/IEEE ASE*, 2018, pp. 888–891.
- [28] R. S. Menezes and *et al.*, “Esbmc v7.4: Harnessing the power of intervals,” in *Tools and Algorithms for the Construction and Analysis of Systems*, B. Finkbeiner and L. Kovács, Eds. Cham: Springer Nature Switzerland, 2024, pp. 376–380.
- [29] N. Tihanyi, T. Bisztray, R. Jain, M. A. Ferrag, L. C. Cordeiro, and V. Mavroudis, “The formai dataset: Generative AI in software security through the lens of formal verification,” in *PROMISE*. ACM, 2023, pp. 33–43.
- [30] A. M. Turing, “On computable numbers, with an application to the entscheidungsproblem,” *London Mathematical Society*, vol. s2-42, no. 1, pp. 230–265, 1936.
- [31] M. Sipser, *Introduction to the Theory of Computation*. Cengage Learning, 2012.
- [32] Z. Fan, X. Gao, A. Roychoudhury, and S. H. Tan, “Automated repair of programs from large language models,” *arXiv preprint arXiv:2205.10583*, 2022.
- [33] A. Strasser, “On pitfalls (and advantages) of sophisticated large language models. doi: 10.48550/arXiv.2303.17511”, 2023.
- [34] S. Imani, L. Du, and H. Shrivastava, “Mathprompter: Mathematical reasoning using large language models,” *arXiv preprint arXiv:2303.05398*, 2023.
- [35] OpenAI, “Hello gpt-4o,” <https://openai.com/index/hello-gpt-4o/>, 2024, accessed: 2024-05-26.
- [36] E. Almazrouei *et al.*, “The falcon series of open language models,” *arXiv preprint arXiv:2311.16867*, 2023.
- [37] L. De Moura and N. Bjørner, “Z3: An efficient smt solver,” in *TACAS*. Springer, 2008, pp. 337–340.
- [38] R. Brummayer and A. Biere, “Boolector: An efficient smt solver for bit-vectors and arrays,” in *TACAS*. Springer, 2009, pp. 174–177.
- [39] H. Barbosa *et al.*, “cvc5: A versatile and industrial-strength smt solver,” in *TACAS*. Springer, 2022, pp. 415–442.
- [40] Q. Gao *et al.*, “Cobot: static c/c++ bug detection in the presence of incomplete code,” in *ICPC*, 2018, pp. 385–388.
- [41] I. Medeiros, N. Neves, and M. Correia, “Dekant: a static analysis tool that learns to detect web application vulnerabilities,” in *ISSSTA*, 2016, pp. 1–11.
- [42] H. Liang, L. Wang, D. Wu, and J. Xu, “Mlsa: a static bugs analysis tool based on llvm ir,” in *IEEE/ACIS SNPD*, 2016, pp. 407–412.
- [43] R. Jain, R. Purandare, and S. Sharma, “Bird: Race detection in software binaries under relaxed memory models,” *ACM TOSEM*, vol. 31, no. 4, pp. 1–29, 2022.
- [44] H. Chen *et al.*, “{MUZZ}: Thread-aware grey-box fuzzing for effective bug hunting in multithreaded programs,” in *USENIX Security Symposium*, 2020, pp. 2325–2342.
- [45] D. R. Jeong *et al.*, “Razzer: Finding kernel race bugs through fuzzing,” in *2019 IEEE SP*. IEEE, 2019, pp. 754–768.
- [46] C. Sadowski *et al.*, “Tricorder: Building a program analysis ecosystem,” in *IEEE/ACM ICSE*, vol. 1, 2015, pp. 598–608.
- [47] C. Calcagno *et al.*, “Moving fast with software verification,” in *NASA Formal Methods Symposium*. Springer, 2015, pp. 3–11.
- [48] R. Ma *et al.*, “Rejection: A ast-based reentrancy vulnerability detection method,” in *Trusted Computing and Information Security: 13th Chinese Conference, CTCSIS 2019*. Springer, 2020, pp. 58–71.
- [49] L. Xin and C. Wandong, “A program vulnerabilities detection frame by static code analysis and model checking,” in *2011 IEEE ICCSN*, 2011, pp. 130–134.
- [50] L. Sampaio and A. Garcia, “Exploring context-sensitive data flow analysis for early vulnerability detection,” *JSS*, vol. 113, pp. 337–361, 2016.
- [51] Z. Guo, T. Tan, S. Liu, X. Liu, W. Lai, Y. Yang, Y. Li, L. Chen, W. Dong, and Y. Zhou, “Mitigating false positive static analysis warnings: Progress, challenges, and opportunities,” *IEEE TSE*, 2023.
- [52] T. Lee, S. Wi, S. Lee, and S. Son, “Fuse: Finding file upload bugs via penetration testing,” in *NDSS*, 2020.
- [53] R. Li *et al.*, “Potassium: penetration testing as a service,” in *ACM SoCC*, 2015, pp. 30–42.
- [54] S. Varvaressos, K. Lavoie, S. Gaboury, and S. Hallé, “Automated bug finding in video games: A case study for runtime monitoring,” *Computers in Entertainment (CIE)*, vol. 15, no. 1, pp. 1–28, 2017.
- [55] S. A. Asadollah, D. Sundmark, S. Eldh, and H. Hansson, “A runtime verification tool for detecting concurrency bugs in freertos embedded software,” in *ISPDC*. IEEE, 2018, pp. 172–179.
- [56] Y. Smaragdakis and C. Csallner, “Combining static and dynamic reasoning for bug detection,” in *International Conference on Tests and Proofs*. Springer, 2007, pp. 1–16.
- [57] K. Alshmrany, A. Bhayat, F. Brauße, L. Cordeiro, K. Korovin, T. Melham, M. A. Mustafa, P. Olivier, G. Reger, and F. Shmarov, “Position paper: Towards a hybrid approach to protect against memory safety vulnerabilities,” in *2022 IEEE Secure Development Conference (SecDev)*, 2022, pp. 52–58.
- [58] F. K. Aljaafari, R. Menezes, E. Manino, F. Shmarov, M. A. Mustafa, and L. C. Cordeiro, “Combining bmc and fuzzing techniques for finding software vulnerabilities in concurrent programs,” *IEEE Access*, vol. 10, pp. 121 365–121 384, 2022.
- [59] K. Song, N. Matulevicius, E. B. de Lima Filho, and L. C. Cordeiro, “Esbmc-solidity: an smt-based model checker for solidity smart contracts,” in *ACM/IEEE ICSE: Companion Proceedings*, 2022, pp. 65–69.
- [60] K. M. Alshmrany, M. Aldughaim, A. Bhayat, and L. C. Cordeiro, “Fusebmc v4: Smart seed generation for hybrid fuzzing: (competition contribution),” in *FASE*. Springer, 2022, pp. 336–340.
- [61] Z. Chen, S. Komrusch, and M. Monperus, “Neural transfer learning for repairing security vulnerabilities in c code,” *IEEE TSE*, vol. 49, no. 1, pp. 147–165, 2023.
- [62] J. Bader, A. Scott, M. Pradel, and S. Chandra, “Getafix: Learning to fix bugs automatically,” *PACMPL*, vol. 3, no. OOPSLA, pp. 1–27, 2019.
- [63] Q. Zhu *et al.*, “A syntax-guided edit decoder for neural program repair,” in *ACM FSE*, 2021, pp. 341–353.
- [64] I. Sutskever, O. Vinyals, and Q. V. Le, “Sequence to sequence learning with neural networks,” *NeurIPS*, vol. 27, 2014.
- [65] K. Huang, X. Meng, J. Zhang, Y. Liu, W. Wang, S. Li, and Y. Zhang, “An empirical study on fine-tuning large language models of code for automated program repair,” in *IEEE/ACM ASE*, 2023, pp. 1162–1174.
- [66] A. Rajasekharan, Y. Zeng, P. Padalkar, and G. Gupta, “Reliable natural language understanding with large language models and answer set programming,” *arXiv preprint arXiv:2302.03780*, 2023.
- [67] Y. Ge, W. Hua, J. Ji, J. Tan, S. Xu, and Y. Zhang, “Openagi: When llm meets domain experts,” *arXiv preprint arXiv:2304.04370*, 2023.
- [68] D. Huang, Q. Bu, J. M. Zhang, M. Luck, and H. Cui, “Agentcoder: Multi-agent-based code generation with iterative testing and optimisation,” *arXiv preprint arXiv:2312.13010*, 2023.
- [69] S. Chakraborty, T. Ahmed, Y. Ding, P. T. Devanbu, and B. Ray, “Natgen: generative pre-training by “naturalizing” source code,” in *ACM FSE*, 2022, pp. 18–30.
- [70] A. M. Dakhel, V. Majdinasab, A. Nikanjam, F. Khomh, M. C. Desmarais, and Z. M. Jiang, “Github copilot ai pair programmer: Asset or liability?” *JSS*, p. 111734, 2023.
- [71] C. S. Xia and L. Zhang, “Less training, more repairing please: revisiting automated program repair via zero-shot learning,” in *ACM FSE*, 2022, pp. 959–971.
- [72] N. Jain, S. Vaidyanath, A. Iyer, N. Natarajan, S. Parthasarathy, S. Rajamani, and R. Sharma, “Jigsaw: Large language models meet program synthesis,” in *ICSE*, 2022, pp. 1219–1231.

- [73] D. Sobania, M. Briesch, C. Hanna, and J. Petke, “An analysis of the automatic bug fixing performance of chatgpt,” *arXiv preprint arXiv:2301.08653*, 2023.
- [74] D. Lin, J. Koppel, A. Chen, and A. Solar-Lezama, “Quixbugs: A multilingual program repair benchmark set based on the quixey challenge,” in *Companion of the ACM SIGPLAN SPLASH*, 2017, pp. 55–56.
- [75] M. Lajkó, D. Horváth, V. Csuvik, and L. Vidács, “Fine-tuning gpt-2 to patch programs, is it worth it?” in *ICCSA*. Springer, 2022, pp. 79–91.
- [76] M. Lajkó, V. Csuvik, and L. Vidács, “Towards javascript program repair with generative pre-trained transformer (gpt-2),” in *ACM APR*, 2022, pp. 61–68.
- [77] K. Zhang *et al.*, “Self-edit: Fault-aware code editor for code generation,” *arXiv preprint arXiv:2305.04087*, 2023.
- [78] I. Bouzenia, P. Devanbu, and M. Pradel, “Repairagent: An autonomous, llm-based agent for program repair,” *arXiv preprint arXiv:2403.17134*, 2024.
- [79] N. T. Islam, J. Khoury, A. Seong, G. D. L. T. Parra, E. Bou-Harb, and P. Najafirad, “LLM-Powered Code Vulnerability Repair with Reinforcement Learning and Semantic Reward,” *arXiv preprint arXiv:2401.03374*, 2024.
- [80] B. Yang and *et al.*, “Multi-objective fine-tuning for enhanced program repair with llms,” *arXiv preprint arXiv:2404.12636*, 2024.
- [81] J. Liu, C. S. Xia, Y. Wang, and L. Zhang, “Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation,” *Advances in Neural Information Processing Systems*, vol. 36, 2024.
- [82] W. Ma *et al.*, “The scope of chatgpt in software engineering: A thorough investigation,” *arXiv preprint arXiv:2305.12138*, 2023.
- [83] C. S. Xia and L. Zhang, “Conversational automated program repair,” *arXiv preprint arXiv:2301.13246*, 2023.
- [84] X. Wang, Y. Wang, Y. Wan, F. Mi, Y. Li, P. Zhou, J. Liu, H. Wu, X. Jiang, and Q. Liu, “Compilable neural code generation with compiler feedback,” *arXiv preprint arXiv:2203.05132*, 2022.
- [85] J. Xu and *et al.*, “Aligning llms for fl-free program repair,” *arXiv preprint arXiv:2404.08877*, 2024.
- [86] Z. Liu and *et al.*, “LLM-CompDroid: Repairing Configuration Compatibility Bugs in Android Apps with Pre-trained Large Language Models,” *arXiv preprint arXiv:2402.15078*, 2024.
- [87] H. Joshi, J. C. Sanchez, S. Gulwani, V. Le, G. Verbruggen, and I. Radiček, “Repair is nearly generation: Multilingual program repair with llms,” in *AAAI*, vol. 37, no. 4, 2023, pp. 5131–5140.
- [88] C. S. Xia and L. Zhang, “Keep the conversation going: Fixing 162 out of 337 bugs for \$0.42 each using chatgpt,” *arXiv preprint arXiv:2304.00385*, 2023.
- [89] D. Beyer, “Software verification: 10th comparative evaluation (sv-comp 2021),” in *27th TACAS*. Springer, 2021, pp. 401–422.
- [90] D. Beyer, T. A. Henzinger, R. Majumdar, and A. Rybalchenko, “Path invariants,” in *ACM SIGPLAN PLDI*, 2007, pp. 300–309.
- [91] S. Lu *et al.*, “Codexglue: A machine learning benchmark dataset for code understanding and generation,” *arXiv preprint arXiv:2102.04664*, 2021.
- [92] H. Husain *et al.*, “Codesearchnet challenge: Evaluating the state of semantic code search,” *arXiv preprint arXiv:1909.09436*, 2019.
- [93] R. Just, D. Jalali, and M. D. Ernst, “Defects4j: A database of existing faults to enable controlled testing studies for java programs,” in *ISSTA*, 2014, pp. 437–440.
- [94] X. Gao, B. Wang, G. J. Duck, R. Ji, Y. Xiong, and A. Roychoudhury, “Beyond tests: Program vulnerability repair via crash constraint extraction,” *ACM TOSEM*, vol. 30, no. 2, pp. 1–27, 2021.
- [95] M. Yasunaga and P. Liang, “Break-it-fix-it: Unsupervised learning for program repair,” in *ICML*. PMLR, 2021, pp. 11941–11952.
- [96] R. Bavishi, H. Joshi, J. Cambronoero, A. Fariha, S. Gulwani, V. Le, I. Radiček, and A. Tiwari, “Neurosymbolic repair for low-code formula languages,” *PACMPL*, vol. 6, no. OOPSLA2, pp. 1093–1122, 2022.
- [97] B. Berabi, J. He, V. Raychev, and M. Vechev, “Tfix: Learning to fix coding errors with a text-to-text transformer,” in *ICML*. PMLR, 2021, pp. 780–791.
- [98] Y. Deng *et al.*, “Large language models are edge-case fuzzers: Testing deep learning libraries via fuzzgpt,” 2023.
- [99] D. Hendrycks, S. Basart, S. Kadavath, M. Mazeika, A. Arora, E. Guo, C. Burns, S. Puranik, H. He, D. Song *et al.*, “Measuring coding challenge competence with apps,” *arXiv preprint arXiv:2105.09938*, 2021.
- [100] H. Huang, C. Xu, M. Wen, Y. Liu, and S.-C. Cheung, “Confix: Repairing configuration compatibility issues in android apps,” in *ACM SIGSOFT ISSTA*, 2023, pp. 514–525.
- [101] J. Kong, M. Cheng, X. Xie, S. Liu, X. Du, and Q. Guo, “Contrastrepair: Enhancing conversation-based automated program repair via contrastive test case pairs,” *arXiv preprint arXiv:2403.01971*, 2024.
- [102] D. Hidvégi, K. Etremadi, S. Bobadilla, and M. Monperrus, “Cigar: Cost-efficient program repair with llms,” *arXiv preprint arXiv:2402.06598*, 2024.
- [103] N. Tihanyi, T. Bisztray, M. A. Ferrag *et al.*, “How secure is ai-generated code: a large-scale comparison of large language models,” *Empirical Software Engineering*, vol. 30, p. 47, 2025. [Online]. Available: <https://doi.org/10.1007/s10664-024-10590-1>
- [104] L. C. Cordeiro, B. Fischer, and J. Marques-Silva, “Smt-based bounded model checking for embedded ANSI-C software,” *IEEE TSE*, vol. 38, no. 4, pp. 957–974, 2012.
- [105] L. C. Cordeiro, D. Kroening, and P. Schrammel, “JBMC: bounded model checking for java bytecode - (competition contribution),” in *TACAS*, ser. LNCS, vol. 11429. Springer, 2019, pp. 219–223.
- [106] K. Song, N. Matulevicius, E. B. de Lima Filho, and L. C. Cordeiro, “Esbmc-solidity: An smt-based model checker for solidity smart contracts,” in *IEEE/ACM ICSE*, 2022, pp. 65–69.
- [107] F. R. Monteiro, M. R. Gadelha, and L. C. Cordeiro, “Model checking C++ programs,” *Softw. Test. Verification Reliab.*, vol. 32, no. 1, 2022.
- [108] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, And Tools*, 2nd ed. Addison-Wesley Longman Publishing Co., Inc., 2006.
- [109] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell *et al.*, “Language models are few-shot learners,” *NeurIPS*, vol. 33, pp. 1877–1901, 2020.
- [110] A. K. Lampinen *et al.*, “Can language models learn from explanations in context?” *arXiv preprint arXiv:2204.02329*, 2022.
- [111] OpenAI, “Gpt-4,” 2023, accessed May 17, 2023. <https://openai.com/research/gpt-4>.
- [112] J. Devlin *et al.*, “Bert: Pre-training of deep bidirectional transformers for language understanding,” *arXiv preprint arXiv:1810.04805*, 2018.
- [113] C. Raffel, N. Shazeer, A. Roberts, K. Lee, S. Narang, M. Matena, Y. Zhou, W. Li, and P. J. Liu, “Exploring the limits of transfer learning with a unified text-to-text transformer,” *JMLR*, vol. 21, no. 1, pp. 5485–5551, 2020.
- [114] A. Vaswani *et al.*, “Attention is all you need,” *NeurIPS*, vol. 30, 2017.
- [115] D. Beyer, “State of the art in software verification and witness validation: Sv-comp 2024,” in *TACAS*. Springer, 2024, pp. 299–329.
- [116] P. E. Black, “A software assurance reference dataset: Thousands of programs with known bugs,” *Journal of Research NIST*, vol. 123, p. 1, 2018.
- [117] J. Fan, Y. Li, S. Wang, and T. N. Nguyen, “A c/c++ code vulnerability dataset with code changes and cve summaries,” in *MSR 2020*. ACM, 2020, p. 508–512.
- [118] Y. Zhou *et al.*, “Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks,” *NeurIPS*, vol. 32, 2019.
- [119] S. Chakraborty, R. Krishna, Y. Ding, and B. Ray, “Deep learning based vulnerability detection: Are we there yet,” *IEEE TSE*, 2021.
- [120] R. Jain, N. Gervasoni, M. Ndhlovu, and S. Rawat, “A code centric evaluation of c/c++ vulnerability datasets for deep learning based vulnerability detection techniques,” in *Proceedings of the 16th Innovations in Software Engineering Conference*, 2023, pp. 1–10.
- [121] J. White, Q. Fu, S. Hays, M. Sandborn, C. Olea, H. Gilbert, A. El-nashar, J. Spencer-Smith, and D. C. Schmidt, “A prompt pattern catalog to enhance prompt engineering with chatgpt,” *arXiv preprint arXiv:2302.11382*, 2023.
- [122] R. Goldblatt and M. Jackson, “Well-structured program equivalence is highly undecidable,” *ACM TOCL*, vol. 13, no. 3, pp. 1–8, 2012.