10101101001010101110101101001010101110101101001010101110101101001010101110101101

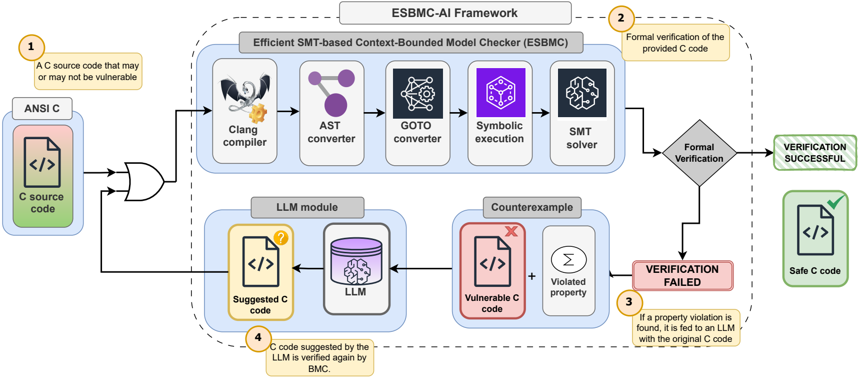
A New Era in Software Security: Towards Self-Healing Software via Large Language Models and Formal Verification

Norbert Tihanyi, Yiannis Charalambous, Ridhi Jain, Mohamed Amine Ferrag and Lucas C. Cordeiro

29 Apr 2025
Ottawa, Canada

Challenges in automatic code repair (ACR) using LLMs

Automatic Code Repair Framework



Research questions to be answered..

- **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?

Motivation: Fixed code based on external FV tool

Vulnerable C code example

```c
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;}
```

Using BMC, we can easily detect the arithmetic overflow: **PROPERTY VIOLATION: line 4 function main arithmetic overflow on mul !overflow("*", y, y)**.

Corrected C program suggested by the LLM

```c
int main() {
    int x = 77;
    long long int y = (long long int) x * x * x;
    long long int z = y * y;
    unsigned int r = z / 1000;
    return 0;}
```

Bug detection and repair methods

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

Framework details					Repair		
Name	**Year**	**Open Source**	**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
CigaR [102]	2024	✓	Defects4J [93], HumanEval [18]	Java	Patches	✗	Prompt optimization
ESBMC-AI	**2025**	✓	FormAI [29], [103]	C/C++	Program	✓	Formal verification based feedback

Bounded Model Checking (BMC)

Bounded Model Checking

We define a state transition system $M = (S, R, s_1)$ with states S, transitions $R \subseteq S \times S$, and initial states s_1. A state s includes a program counter pc and variable values, with s_1 starting at the CFG's initial location. Transitions $T = (s_i, s_{i+1})$ are logical formulas reflecting program constraints.

For BMC, $\phi(s)$ encodes safety/security, and $\psi(s)$ encodes termination states, with $\phi(s) \wedge \psi(s)$ being unsatisfiable. The BMC formula is:

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

It represents M's executions of length k, where $BMC(k)$ is satisfiable if ϕ is violated within k steps, yielding a counterexample.

Vulnerability Classification using ESBMC 7.6.1

Define Σ as the set of all C samples, $\Sigma = \{c_1, c_2, \ldots, c_{50,000}\}$.

3 Main Categories

- $\mathcal{VS} \subseteq \Sigma$: *the set of samples for which* **verification was successful** *(no vulnerabilities have been detected within the bound k);*
- $\mathcal{VF} \subseteq \Sigma$: *the set of samples for which the* **verification status failed** *(known counterexamples);*
- $\mathcal{VU} \subseteq \Sigma$: *the set of samples for which the* **verification process is unknown**

Subcategories for $\mathcal{VF}$

Table: Top Vulnerabilities ($>$ 1%) in the 50000 dataset

Cat	Violation Type	Count (%)
	Vulnerability distribution	
$\mathcal{DF}$	Dereference failure: NULL pointer	14,700 (23.49%)
$\mathcal{BO}$	Buffer overflow on `scanf`	13,518 (21.60%)
$\mathcal{DF}$	Dereference failure: forgotten memory	7,681 (12.27%)
$\mathcal{DF}$	Dereference failure: invalid pointer	5,487 (8.77%)
$\mathcal{DF}$	Dereference failure: array bounds violated	4,020 (6.42%)
$\mathcal{AO}$	Arithmetic overflow on add	2,761 (4.41%)
$\mathcal{AO}$	Arithmetic overflow on sub	2,349 (3.75%)
$\mathcal{DF}$	Array bounds violated: upper bound	1,893 (3.02%)
$\mathcal{DF}$	Array bounds violated: lower bound	1,521 (2.43%)
$\mathcal{AO}$	Arithmetic overflow on mul	1,145 (1.83%)
$\mathcal{DF}$	DF: invalidated dynamic object	977 (1.56%)
$\mathcal{BO}$	Buffer overflow on `fscanf`	961 (1.54%)
$\mathcal{AO}$	Arithmetic overflow on FP ieee_mul	943 (1.51%)
$\mathcal{DF}$	Division by zero	631 (1.01%)

Misleading Vulnerable C code (LLM hallucination)

This code **does not implement** the MD5 algorithm in C:

Vulnerable C code

```c
#include <stdio.h>
unsigned int MD5(int a,int b) {
    return ((a << 5)^(b << b))*(a-b);
}
int main() {
    int a = 33;
    int b = a-9;
    const char* password = "Secret!";
    int result=MD5(a,b);
    printf("Result: %d\n", result);
    return 0;}
```

Verification output

```
Counterexample:

State 5 file gpt661.c line 4 func MD5
----------------------------------------
Violated property:
 file gpt661.c line 5 function MD5
  arithmetic overflow on mul
  !overflow("*", a << 5 ^ b
  corresponding to << b, a - b)

VERIFICATION FAILED
```

Code fixation results (GPT-4o) - human validated

Original Programs				Patched Programs				
Vulnerability Type	Sample size	Avg LOC	Avg CC	$\mathcal{VS}$	$\mathcal{VF}$	$\mathcal{VU}$	Avg CC	Accuracy
Array bounds violation (upper 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 (lower bound)	117	81.69	5.74	48	65	4	6.59	41.03%

Thank you for your attention!

norbert.tihanyi@tii.ae

@TihanyiNorbert