1010110100101010111010110100101010111010110100

The FormAI Dataset: Generative AI in Software Security through the Lens of Formal Verification

Norbert Tihanyi, Tamas Bisztray, Ridhi Jain, Mohamed Amine Ferrag, Lucas C. Cordeiro, Vasileios Mavroeidis

Challenges in automatic code repair (ACR)

The FormAI Dataset

Motivation: To create a dataset where each sample code is correctly labeled as vulnerable or not, using formal verification methods, to minimize the occurrence of false positives and negatives.

FormAI Dataset

FormAI is a novel **AI-generated dataset** *comprising 112,000 compilable and independent C programs. All the programs in the dataset were generated by* **GPT-3.5-turbo** *using* **dynamic zero-shot prompting** *technique and comprises programs with varying levels of complexity. Each program is* **labelled** *based on vulnerabilities present in the code using a formal verification method based on the* **Efficient SMT-based Bounded Model Checker (ESBMC)**.

FormAI dataset - Availability

The dataset can be accessed on both GitHub and IEEE Dataport.

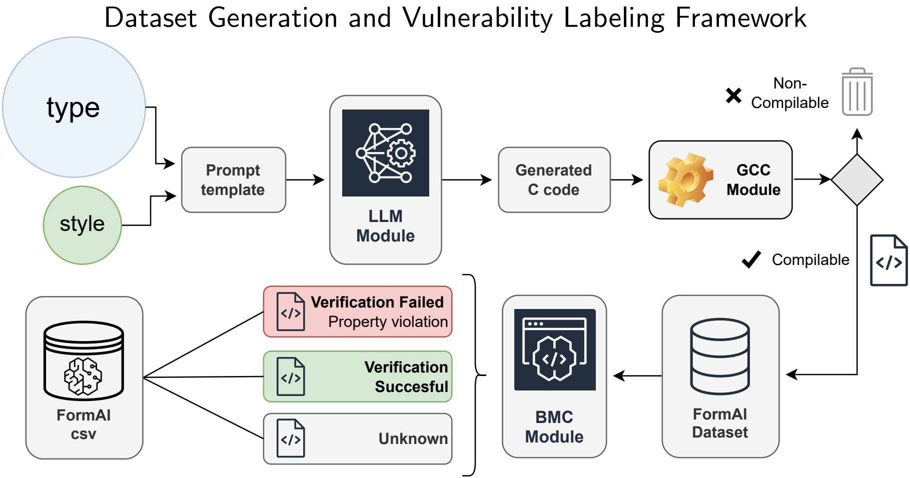
- **GitHub:** `https://github.com/FormAI-Dataset/`
- **IEEE dataport:** `https://dx.doi.org/10.21227/vp9n-wv96`

FormAI dataset - Structure

The dataset comprises three distinct files:

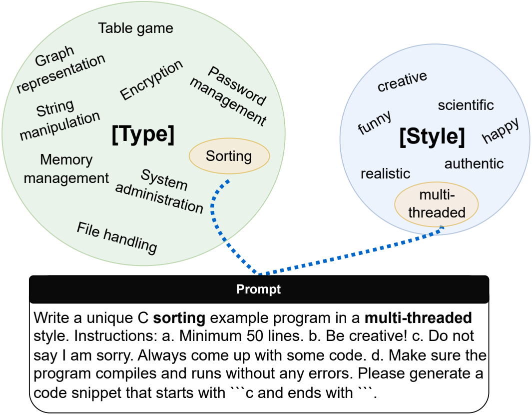
- **FormAI_dataset_C_samples-V1.zip** - This file contains all the 112,000 C files.
- **FormAI_dataset_classification-V1.zip** - This file contains a CSV file with the original code and vulnerability classification.
- **FormAI_dataset_human_readable-V1.csv** - Human readable version

Methodology for Dataset creation



- **LLM module** → GPT-3.5-turbo
- **BMC module** → ESBMC 7.3

Ensure Diversity



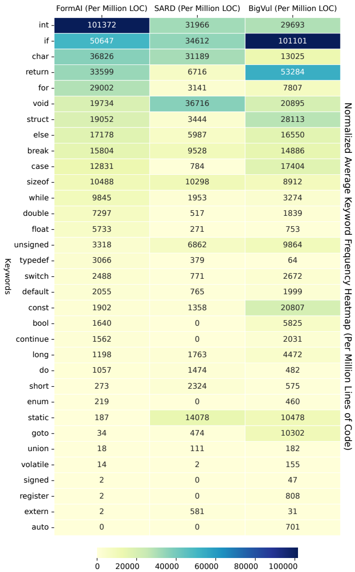
- Proper prompt engineering is crucial for achieving a diverse dataset.
- Each API call randomly chooses a type from 200 options in the Type category, including topics like Wi-Fi Signal Strength Analyzer, QR Code Reader, and others. Similarly, a coding style is selected from 100 options in the Style category during each query.

Enhancing code compilability

To minimize the error within the generated code, we have established five instructions in each specific prompt:

1. `Minimum 50 lines`: This encourages the LLM to avoid the generation of overly simplistic code with only a few lines (which occasionally still happens);
2. `Be creative!`: The purpose of this instruction is to generate a more diverse dataset;
3. `Do not say I am sorry`: The objective of this instruction is to circumvent objections and responses such as "As an AI model, I cannot generate code", and similar statements.
4. `Make sure the program compiles`: This instruction encourages the model to include header files and create a complete and compilable program.
5. ``Generate a code snippet that starts with ```c``: Enable easy extraction of the C code from the response.

C Keyword frequency in FormAI, SARD, and BigVul



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:*

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

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 7.3

Define Σ as the set of all C samples, $\Sigma = \{c_1, c_2, \ldots, c_{112,000}\}$.

4 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);*
- $\mathcal{TO} \subseteq \Sigma$: *the set of samples for which the* **verification process was not completed** *within the provided time frame (as a result, the status of these files remains uncertain);*
- $\mathcal{ER} \subseteq \Sigma$: *the set of samples for which the* **verification status resulted in an error***.*

9 subcategories for $\mathcal{VF}$

9 Subcategories

- $\mathcal{ARO} \subseteq \mathcal{VF}$: *Arithmetic overflow*
- $\mathcal{BOF} \subseteq \mathcal{VF}$: *Buffer overflow on* `scanf()/fscanf()`
- $\mathcal{ABV} \subseteq \mathcal{VF}$: *Array bounds violated*
- $\mathcal{DFN} \subseteq \mathcal{VF}$: *Dereference failure : NULL pointer*
- $\mathcal{DFF} \subseteq \mathcal{VF}$: *Dereference failure : forgotten memory*
- $\mathcal{DFI} \subseteq \mathcal{VF}$: *Dereference failure : invalid pointer*
- $\mathcal{DFA} \subseteq \mathcal{VF}$: *Dereference failure : array bounds violated*
- $\mathcal{DBZ} \subseteq \mathcal{VF}$: *Division by zero*
- $\mathcal{OTV} \subseteq \mathcal{VF}$: *Other vulnerabilities*

Which parameters are most effective?

Table: Classification results for different parameters

(u,t)	VULN	k-ind	Running time (m:s)	$\mathcal{VS}$	$\mathcal{VF}$	$\mathcal{TO}$	$\mathcal{ER}$
(2,1000)	2438	✗	758:09	371	547	34	48
(3,1000)	2373	✗	1388:39	366	527	57	50
(2,100)	2339	✗	175:38	367	529	61	43
(2,100)	2258	✓	400:54	340	603	20	37
(1,100)	2201	✗	56:29	416	531	17	36
(1,30)	2158	✓	146:13	349	581	34	36
(3,100)	2120	✗	284:22	354	483	120	43
(1,30)	**2116**	✗	**30:57**	**416**	**519**	**30**	**35**
(1,10)	2069	✓	61:58	360	553	52	35
(1,10)	2038	✗	19:32	413	503	51	33
(3,30)	1962	✗	125:19	342	444	172	42
(1,1)	1557	✓	10:59	355	406	208	31
(1,1)	1535	✗	6:22	395	374	201	30

✓: Enabled, ✗: Disabled, $(u, t) =$ unwind and timeout parameters

Vulnerabilities identified by ESBMC

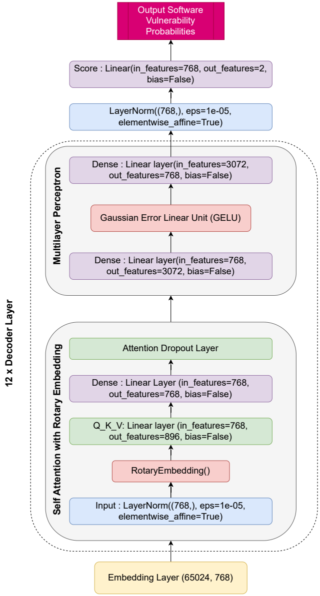
#Vulns	Vuln.	Associated CWE-numbers
88,049	$\mathcal{BOF}$	CWE-20, CWE-120, CWE-121, CWE-125, CWE-129, CWE-131, CWE-628, CWE-676, CWE-680, CWE-754, CWE-787
31,829	$\mathcal{DFN}$	CWE-391, CWE-476, CWE-690
24,702	$\mathcal{DFA}$	CWE-119, CWE-125, CWE-129, CWE-131, CWE-755, CWE-787
23,312	$\mathcal{ARO}$	CWE-190, CWE-191, CWE-754, CWE-680, CWE-681, CWE-682
11,088	$\mathcal{ABV}$	CWE-119, CWE-125, CWE-129, CWE-131, CWE-193, CWE-787, CWE-788
9823	$\mathcal{DFI}$	CWE-416, CWE-476, CWE-690, CWE-822, CWE-824, CWE-825
5810	$\mathcal{DFF}$	CWE-401, CWE-404, CWE-459
1620	$\mathcal{OTV}$	CWE-119, CWE-125, CWE-158, CWE-362, CWE-389, CWE-401, CWE-415, CWE-459, CWE-416, CWE-469, CWE-590, CWE-617, CWE-664, CWE-662, CWE-685, CWE-704, CWE-761, CWE-787, CWE-823, CWE-825, CWE-843
1567	$\mathcal{DBZ}$	CWE-369

Research Questions Answered

Research Questions

- **RQ1**: *How likely is purely LLM-generated code to contain vulnerabilities on the first output when using simple zero-shot text-based prompts?*
  Answer: *At least* 51.24% *of the samples from the 112,000 C programs contain vulnerabilities. This indicates that GPT-3.5 often produces vulnerable code. Therefore, one should exercise caution when considering its output for real-world projects.*
- **RQ2**: *What are the most typical vulnerabilities LLMs introduce when generating code?*
  Answer: *For GPT-3.5: Arithmetic Overflow, Array Bounds Violation, Buffer Overflow, and various Dereference Failure issues were among the most common vulnerabilities. These vulnerabilities are pertinent to MITRE's Top 25 list of CWEs.*

Future Research - Fine tuned BERT / Fuzzing



Thank you for your attention!

norbert.tihanyi@tii.ae

@TihanyiNorbert