Discover Artificial Intelligence

RESEARCH

Open Access

Helping LLMs improve code generation using feedback from testing and static analysis

Greta Dolcetti[3*], Vincenzo Arceri[1], Eleonora Iotti[1], Sergio Maffeis[2], Agostino Cortesi[3] and Enea Zaffanella[1]

*Correspondence:
Greta Dolcetti
greta.dolcetti@unive.it
[1]University of Parma, Parco Area delle Scienze 53/A, Parma 43124, PR, Italy
[2]Imperial College London, South Kensington Campus, London SW7 2AZ, United Kingdom
[3]Ca' Foscari University of Venice, Via Torino 155, Venice 30172, VE, Italy

Abstract
Large Language Models (LLMs) are one of the most promising developments in the field of artificial intelligence, and the software engineering community has readily noticed their potential role in the software development life-cycle. Developers routinely ask LLMs to generate code snippets, increasing productivity but also potentially introducing ownership, privacy, correctness, and security issues. Previous work highlighted how code generated by mainstream commercial LLMs is often not safe, containing vulnerabilities, bugs, and code smells. In this paper, we present a framework that leverages testing and static analysis to assess the quality, and guide the self-improvement, of code generated by general-purpose, open-source LLMs. First, we ask LLMs to generate C code to solve a number of programming tasks. Then we employ ground-truth tests to assess the (in)correctness of the generated code, and a static analysis tool to detect potential safety vulnerabilities. Next, we assess the models ability to evaluate the generated code, by asking them to detect errors and vulnerabilities. Finally, we test the models ability to fix the generated code, providing the reports produced during the static analysis and incorrectness evaluation phases as feedback. Our results show that models often produce incorrect code, and that the generated code can include safety issues. Moreover, they perform very poorly at detecting either issue. On the positive side, we observe a substantial ability to fix flawed code when provided with information about failed tests or potential vulnerabilities, indicating a promising avenue for improving the safety of LLM-based code generation tools.

Keywords Large language models, Code generation, Static analysis, Code repair

1 Introduction

The impressive acceleration we are witnessing in the offer of new and increasingly efficient Large Language Models (LLMs), on the one hand fascinates many researchers and on the other creates distrust and a priori attitudes of refusal.

One of the most worrying aspects is the ability of these models to generate source code automatically, with the perceived risk of losing control of the reliability of software systems, in a context in which all the main aspects of our lives depend on them. Many studies already aim at demonstrating the intrinsic weaknesses of LLMs and their applications [1]. A comprehensive list of such weaknesses is reported by OWASP (Open Web

© The Author(s) 2026. **Open Access** This article is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License, which permits any non-commercial use, sharing, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if you modified the licensed material. You do not have permission under this licence to share adapted material derived from this article or parts of it. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit http://creativecommons.org/licenses/by-nc-nd/4.0/.

Application Security Project) in its Top 10 vulnerabilities for LLMs,[1] where we can find vulnerabilities such as training data poisoning (LLM03) or sensitive information disclosure (LLM06). Despite that, programmers frequently use LLM-powered tools (e.g., GitHub Copilot [2]) for speeding up the development process or to get a starting point for finding solutions [3], be that for simple code snippets, specific functions or entire software applications. There is no doubt that this is a rapidly growing trend: Gartner [4] predicts that by 2027, 50% of enterprise software engineers will rely on AI-powered coding tools, up from fewer than 5% today. In this paper we propose initial steps towards the goal of ensuring that LLM-enabled code generation is safe and trustworthy. Our method addresses reliability issues that now prevent LLM-based code generation from being widely used in high-stakes software engineering scenarios, while also paving the path for its integration into realistic development pipelines by prioritizing automation and modularity. Specifically, we aim to: (i) experimentally evaluate the correctness and safety of code generated by LLMs, (ii) measure the ability of LLMs to identify and remediate issues in generated code, and (iii) investigate whether an LLM is better at understanding code generated by itself as opposed to by other models.

To achieve this, we propose a testing and analysis framework with three main phases: code generation, self-evaluation and repair. In the code generation phase we ask each LLM to generate code based on a natural language task specification. We then extract and clean the generated code, and we evaluate its correctness, by running it against ground-truth unit tests. Finally, we evaluate the code safety, by using a static analysis tool to look for safety-related issues. In the self-evaluation phase we ask each LLM to evaluate the correctness and safety of the code generated in the previous phase by itself and by the other models. This allows us to test a larger dataset, but also to measure the preference of each model for its own answers ("*self-preference*"). In the repair phase we present an LLM with incorrect or unsafe code, and the description of an issue identified in it during the generation phase. We ask the LLM to repair the code a small number of time, in case the first attempt failed.

For our experiments we consider four LLMs: Llama 3 8B and 70B [5], Gemma 7B [6], and Mixtral 8X7B [7]. We chose these popular models because they are recent and high-performing general models also capable of code generation, and they are freely available and open-source. As such, they can be easily used to support programming tasks, can be modified and integrated in practical software engineering tools. The models are diverse, allowing us to have a fair comparison between different numbers of parameters (7, 8, 46.7, 70), training datasets, and techniques (RLHF, Mixture of Experts, etc).

The prompt used to query an LLM can have a significant impact on the quality of the response. For each phase, we have performed prompt-engineering experiments to select the most effective prompts. The results reported in the paper are based on the best-performing prompt for that phase, while the other prompts and their results are reported in the Appendix.

We target the C programming language, which requires careful handling to produce correct and safe programs, particularly for memory management.

To automatically detect safety issues in generated C code, we use Infer [8], a state-of-the-art open-source static analysis tool by Meta. Infer is designed to identify issues in

[1] https://owasp.org/www-project-top-10-for-large-language-model-applications/

source code without compiling and executing it. The analysis completes within a configurable timeout. Due to well-known undecidability results, Infer's analysis is neither *complete* nor *sound*. In other words, it may suffer from both *false negatives* (missing existing issues) and *false positives* (reporting non-existing issues), which we need to manually investigate. It is important to note that the manual investigation of false positives is the only manual step in our experimental pipeline and only serves the purpose of further investigation; all other components of our methodology are fully automated.

1.1 Contributions

We summarize the novel contributions of this paper below.

- We provide a comprehensive evaluation of LLM-generated C code, combining correctness testing (46%–65% pass rate) and static vulnerability analysis with Infer (87%–96% safe programs);
- We evaluate LLMs' code repair capabilities using feedback from both correctness testing and static analysis, achieving 59% success for correctness issues and 89% for safety vulnerabilities;
- We investigate LLMs' self-preference in code assessment, finding no evidence of this effect but revealing strong one-sided prediction tendencies in some models;
- We conduct systematic prompt engineering experiments demonstrating the critical impact of prompt design on output quality across all phases;
- We provide a fully automated, modular pipeline requiring minimal human intervention, easily extensible to other models and analyzers. Code and data are availablenull at https://doi.org/10.6084/m9.figshare.26984716.

The uniqueness of this paper lies in (i) the comprehensive evaluation that combines correctness testing, vulnerability detection, and code repair capabilities in a unified framework, (ii) the novel investigation of LLMs' self-preference in code assessment, which reveals low awareness in model predictions, and (iii) a fully automated, modular pipeline that enables systematic experimentation while requiring minimal human intervention. The systematic analysis of both code quality and LLMs' self-assessment capabilities provides valuable insights into the current state of LLM-based code generation, while our modular framework enables straightforward extension to future models and analysis tools.

1.2 Paper structure

Section 2 outlines the experimental setup, including dataset creation, model selection and prompt engineering. Section 3 describes how we asked each model to generate code to solve a task, how we cleaned the code, and how we set criteria for correctness and safety. Sections 4 and 5 describe how we ask models to respectively evaluated the correctness of, and repair the issues in, generated code. Section 6 presents the experimental results for each experimental phase. Section 7 discusses the related work, Sect. 8 discusses the results and the limitations of our study, and Sect. 9 highlights future research directions and concludes the paper.

2 Experimental setup

2.1 Benchmark creation

We designed a benchmark of inputs to be fed to LLMs to evaluate their code generation capabilities, randomly collecting 100 suitable tasks from the Mostly Basic Python Problems Dataset (MBPP) [9],[2] and recasting them to the C language, keeping the same JSON structure provided by MBPP. The original benchmark contains 974 programming tasks, designed to be solvable by entry-level programmers and constructed by crowdsourcing to an internal pool of crowdworkers who have basic knowledge of Python. We manually inspected and selected each task, as detailed in the following. We opted to randomly select 100 tasks, rather than using the full benchmark suite, to enable a manual and precise investigation of results in later phases. In the code generation phase, LLMs are prompted to generate code to solve each task, generating 100 samples per model. To assess whether the generating model's identity impacts the analyzing model's reasoning, in the following evaluation and repair phases each model evaluates the code generated by itself and the other 3 LLMs, multiplying the number of samples by a factor of 4.

The choice of grounding this paper on the C programming language, rather than in other languages, is motivated by the need to address a well-recognized imbalance in existing code generation benchmarks. As highlighted in [10], over 95% of current benchmarks are dominated by Python, leaving LLM capabilities in other widely used programming languages, indeed, C, largely unexplored. By focusing on C, we aim to complement previous Python-focused research evaluations and provide empirical evidence on model performance in a language that is representative of systems-level programming and different in its abstractions and execution model.

The selection of 100 tasks is described as follows. We selected 100 tasks at random from the MBPP dataset and then screened them for compatibility with the C programming language. In particular, tasks whose specifications fundamentally relied on Python-specific built-in data structures (e.g., tuples) that lack direct counterparts in C were discarded. Each discarded task was replaced by an additional randomly sampled task, and this process was iterated until a final set of 100 distinct tasks satisfying all selection criteria was obtained, with no duplicates.

Then, the procedure used to translate each of the 100 Python tasks into the C language is detailed as follows[3]:

- *Function signature*: We extracted the target function name and input/output types from the MBPP task description and Python reference solution, and defined a corresponding C function signature with explicit types (e.g., pointers and size parameters for arrays, `char*` for strings, scalar types for numeric values);
- *Input construction*: Python literals used in tests, such as lists, strings, were translated into explicit C initializations. Specifically Python lists were rewritten as statically allocated C arrays, with their size passed explicitly to the function when required.
- *Explicit output storage*: When the Python function returned a list or string, the C corresponding task was designed to write the result into a caller-dynamically-allocated buffer and return a pointer, depending on the task. Output buffers were allocated and initialized in the test before invoking the function under test;

[2] https://github.com/google-research/google-research/tree/master/mbpp

[3] The resulting list of tasks for C is available at https://doi.org/10.6084/m9.figshare.26984716.

- *Function invocation*: Each Python `assert` in the form `assert f(input) == expected`, was rewritten as a sequence of C statements that; (i) initialize the input data, (ii) call the target C function, (iii) store the result in a predefined variable;
- *Result comparison*: Since C does not provide built-in equality operators for compound data structures, arrays were compared using `memcmp` over the relevant memory region, strings were compared using `strcmp`, while scalar values were compared using the standard equality operator.

For instance, let us consider the task #940 from MBPP, *"Write a function to sort the given array by using heap sort."*, with their unit tests (provided in the field `test_list` of MBPP), such as `assert heap_sort([12, 2, 4,3]) == [2, 3, 4, 12]`. Since C allows array literals only for array initialization, and it does not provide a built-in operator for array comparison, in the example we refactored the unit test as shown in Fig. 1, using `memcmp` for array comparison. It is important to highlight that this refactoring process is a one-time systematic preparation step performed during benchmark creation, not an ongoing manual intervention in the experimental pipeline.

To compile the programs we generate, we use the Clang compiler, version 15.0.0.

2.2 (Non-)Memorization in benchmark design

Non-memorization is a critical issue in code generation research. Using a crafted C version of the MBPP benchmark, instead of using existing C benchmarks such as HumanEval [11], is a deliberate attempt to minimize the risk of the models having memorized the original Python tasks they could have been trained on. Our solution does not entirely eliminate the risk, as some parts of the task formulation may still overlap with the training data of the considered models. Yet, the primary goal of this work is to evaluate models' ability to detect, repair, and improve safety vulnerabilities and correctness in code, rather than assessing the quality of the LLM-generated code, where a deep treatment of the non-memorization problem would be mandatory. As we will see in Sect. 6 (cf. Table 8), models frequently generate incorrect solutions, suggesting that the code is generated from scratch rather than reproduced from memorized data. Our solution suggests a direction for future work focused on evaluating a model generalization ability by testing whether the generated C code closely resembles or significantly differs from a direct translation of the original Python task.

2.3 Models

For our experiments, we chose the following instruction fine-tuned models, because they are among the most recent and high-performing general models also capable of code generation, in addition to being publicly released and freely accessible.

- Llama 3, developed by Meta, in the 8 billion and 70 billion parameters versions [5]. These are two state-of-the-art pre-trained models with capabilities like reasoning, code generation, and instruction following. These versions use supervised fine-

```
int a1[] = {12, 2, 4, 3}; int e1[] = {2, 3, 4, 12};
heap_sort(a1, 4);
assert(memcmp(a1, e1, 4 * sizeof(int)) == 0);
```

Fig. 1 An example test case of MBPP, translated to C. The unit test refers to the task #940.

tuning (SFT) and reinforcement learning with human feedback (RLHF) to align with human preferences for helpfulness and safety. The models were pre-trained on over 15 trillion tokens of data from publicly available sources and the fine-tuning data includes publicly available instruction datasets, as well as over 10 million human-annotated examples;

- Gemma with 7 billion parameters [6] is a small model developed by Google using a novel RLHF method, leading to substantial gains on quality, coding capabilities, factuality, instruction following a multi-turn conversation quality. The model was trained on 3 trillion tokens that include web documents, code, and mathematics content;
- Mixtral of Experts [7] is a Sparse Mixture-of-Experts (SMoE) LLM developed by Mistral which uses 12 billion active parameters out of 46.7 billion in total and a context of 32K tokens with strong code generation capabilities. This model has the same architecture as Mistral 7B [12] but each layer is composed of 8 feed-forward blocks (i.e., experts) and for every token, at each layer, a router network selects two experts to process the current state and combines their outputs, eventually with different experts at every step.

This selection allows us to have a fair comparison between the models that have different numbers of parameters (7, 8, 46.7, 70), training datasets, and architectures.

2.4 API

We conduct our experiments using the Groq API[4] to prompt the LLMs previously described for all the phases that require a direct interaction with the models. Due to the nature of the tasks performed, the temperature value is set to 0.2, following the documentation recommendation to make the output more focused and deterministic[5].

2.5 Prompt engineering

In the following sections we provide, for brevity, only a brief description of the system and content prompt we actually used to obtain the results being discussed.

For each phase of the experimental settings, we tested several prompts, adopting prompt engineering techniques to maximize the adequacy and correctness of the output generated for the requested task. Many of the prompts have been inspired by [13], regarding both the text arrangement and the structure. The prompts were specifically tailored to guide the models in producing outputs that aligned with the specific task constraints, such as compiling successfully and adhering to the function signatures. We used both system prompts, which provided explicit directives to the models, and content prompts, which described the task and requirements. We iterated on their structure and content to achieve a better performance in terms of code generation, self-evaluation, and repair capabilities. The discarded prompts and the corresponding experiments are reported in tables in the Appendix. Each experiment for each prompt was run multiple times, but since the results were consistent, we report the results of a single run.

[4] https://groq.com/

[5] https://console.groq.com/docs/api-reference#chat-create

3 Code generation

The first phase of our approach is code generation. In this first step, each model is instructed about both its role and output constraints. The system prompt, shown in the Appendix, Fig. 17, provides directives on how the code must solve the task correctly, compile, and be safe, the correct libraries must be included, the given signature must be adopted, and the output must be wrapped between the comments `//BEGIN` and `//END`. Both a correct and incorrect example of execution are provided and the expected output format is clarified. Additionally, the code should be followed by some explanations to stimulate Chain-of-Thought [14] reasoning. Thus, the system prompt concludes with *"Let's work this out in a step by step way to be sure we have the right answer"*, as suggested by [15]. The content prompt is more straightforward and contains the description of the task to be solved and the required signature. Two examples can be seen in Fig. 17.

3.1 Code cleaning

During each code generation phase in our experiments, the output produced by the model is fed to an automatic script-based cleaning phase, which extracts the source code from the LLM response and applies minor edits that preserve the code semantics: (i) removing backslashes that were used as escape characters in the code (this is typical of the `mixtral-8x7b-32768` model), (ii) including C standard libraries that the model might have forgotten (e.g., `math.h`, `limits.h`), and (iii) removing lines starting with ```` ``` ````, that are related to formatting.

3.2 Correctness

We run unit tests adapted from the MBPP dataset on the compiling programs. Note that, it is not possible to *prove* the correctness of the generated code with testing, yet it is possible to demonstrate its incorrectness. To better investigate and provide a more detailed feedback for the repair phase, we distinguish four possible outcomes from this phase: (i) all the tests passed, (ii) at least one test failed, (iii) an execution error (e.g., segmentation fault) or a 60 s timeout occurred, and (iv) a compilation error occurred.

3.3 Safety

The generated files compiling without errors are then statically analyzed by Infer. For each file, Infer generates an analysis report containing the issues that have been detected; this report will be used as feedback in the next repair phase.

Note that Infer does not attempt at proving that a piece of code is free of errors; rather, it searches for and reports likely issues. Infer's reports are often classified into "buckets" according to their expected precision: Level 1 (i.e., top priority) buckets are those corresponding to reports that are known to be *true positives*, i.e., definite programming errors; lower priority buckets are used for reports that are likely to be false positives. In our setting, we activated the following checkers that can be found in the Infer documentation[6]:

- `bufferoverrun`, which is meant to track programming errors leading to *buffer overrun*; to this end, it tracks and reports code that is likely to witness a memory allocation error (e.g., zero or negative or hugely sized allocations), an out-of-bound

[6] https://fbinfer.com/docs/all-checkers

array access, or an integer computation leading to overflow using an analysis based on the domain of intervals [16];

- `linters`, which implements a declarative-based linting framework providing syntax-based checkers tracking, e.g., dangerous pointer conversions;
- `pulse`, which targets memory safety issues and it can also report issues related to the use of uninitialized values, the escape of a stack variable's address, and the dereference of a constant address.

The aforementioned checkers can find the issues related to memory management, resource leaks, and access controls reported in Table 1 with different level of precision. The name of each issue is quite self-explanatory, and a full explanation can be found in the documentation.[7] These issues have been selected because they are popular, relevant, and less likely to result in false positives. From now on, we refer to these issues as *vulnerabilities*.

4 Self-evaluation

The second phase of our approach is self-evaluation, where we ask each model to detect correctness and safety issues in the code generated by itself and other models.

4.1 Correctness

We ask each model whether the generated code correctly solves a task. The collected responses are then tested against the results obtained from the correctness analysis of the first generation phase. The system prompt, shown in Fig. 2, instructs the LLM to be a classifier for a correctness check, highlighting the structure of the expected response: a YES or NO followed by an explanation (Chain-of-Thought), as in the code generation phase. Both a correct and incorrect example of responses are provided. The content prompt contains the description of the task and the output previously generated by the model itself or other models.

4.2 Safety

To detect vulnerabilities, the content prompt provides the generated source code as input along with the name and description of a single vulnerability type. The name and description of vulnerabilities were taken from the Infer documentation, to allow us to compare the responses of this phase with the ground truth obtained in the vulnerability

Table 1 Infer issues considered for the safety analysis

Issue Name
USE_AFTER_FREE
UNINITIALIZED_VALUE
STACK_VARIABLE_ADDRESS_ESCAPE
NULLPTR_DEREFERENCE
INFERBO_ALLOC_IS_ZERO
INFERBO_ALLOC_IS_NEGATIVE
CONSTANT_ADDRESS_DEREFERENCE
MEMORY_LEAK
INTEGER_OVERFLOW_{L1,L2}
BUFFER_OVERRUN_{L1,L2,L3,S2}

[7] https://fbinfer.com/docs/all-issue-types

```
SysPrompt >>> You are a chatbot whose purpose is to check the correctness of a function, written in the C
programming language, to solve a task.
The function, the task to solve, and the requested signature of the function are given to you as inputs.
You must reply clearly to the question 'Does this function match the specifications and solve the task?'
with 'YES' or 'NO', at the starting of your response.
An example of a correct function you must detect by replying 'YES' is the following:
PROMPT: The function is
//BEGIN
int sum(int a, int b) {
    return a + b;
}
//END
The signature of the function must be int sum(int a, int b). The task to solve is Write a C function to
return the sum of two integers. Does this function match the specifications and solve the task?
OUTPUT: YES

An example of an incorrect function you must detect by replying 'NO' is the following:
PROMPT: The function is
//BEGIN
int sum(int a, int b) {
    return a - b;
}
//END
The signature of the function must be int sum(int a, int b). The task to solve is Write a C function to
return the sum of two integers. Does this function match the specifications and solve the task?
OUTPUT: NO

Your answer must be in the following format: <reply> <reasoning>
Let's work this out in a step by step way to be sure we have the right answer.
```

Fig. 2 System prompt of the first task of the self-evaluation phase, to perform correctness classification

```
SysPrompt >>> You are a chatbot whose purpose is to evaluate the safety of a function, written in the C
programming language. The function and the possible vulnerability issue is given to you as inputs. You must reply
to the question with YES or NO, at the starting of your answer.
An example chat is the following, where you will be asked about the UNINITIALIZED_VALUE error, and the function
contains it.
PROMPT: The function is
//BEGIN
#include <stdio.h>
int sequence(int n) {
    if (n == 1) {
        return 0;
    } else if (n == 2) {
        return 1;
    } else {
        int a = 0, b = 1, temp;
        for (int i = 3; i <= n; i++) {
            temp = a + b;
            a = b;
            b = temp;
        }
        return temp;
    }
}
//END
Does the code contain the following vulnerability?
UNINITIALIZED_VALUE : A value is read before it has been initialized.
OUTPUT: YES

Another example chat is the following, where you will be asked about the UNINITIALIZED_VALUE error, but the
function is safe.
PROMPT: The function is
//BEGIN
int decimal_To_Binary(int N) {
    int binary = 0;
    int power = 0;
    while (N > 0) {
        int remainder = N % 2;
        N = N / 2;
        binary = binary + (remainder * pow(10, power));
        power++;
    }
    return binary;
}
//END
Does the code contain the following vulnerability?
UNINITIALIZED_VALUE : A value is read before it has been initialized.
OUTPUT: NO

Your answer must be in the following format: <reply> <reasoning>
Let's work this out in a step by step way to be sure we have the right answer.
```

Fig. 3 System prompt of the second task of the self-evaluation phase, to perform vulnerability detection

analysis. Thus, there are several queries for a single piece of code, one for each type of vulnerability that has been detected in the previous step. As instructed by the system prompt shown in Fig. 3, the model must reply YES or NO to this query. YES is a negative answer: at least a vulnerability of such a type was found. Conversely, when the model

```
SysPrompt >>> You are a chatbot whose purpose is to correct an incorrect function, written in the C
programming language, to solve a task.
The function, the task to solve, and the expected signature of the function are given to you as inputs as
well as the counterexamples for which the code is incorrect.
Provide the corrected code and wrap the code between "//BEGIN" and "//END".
```

Fig. 4 System prompt of the first task of the repair phase, for repairing incorrect code

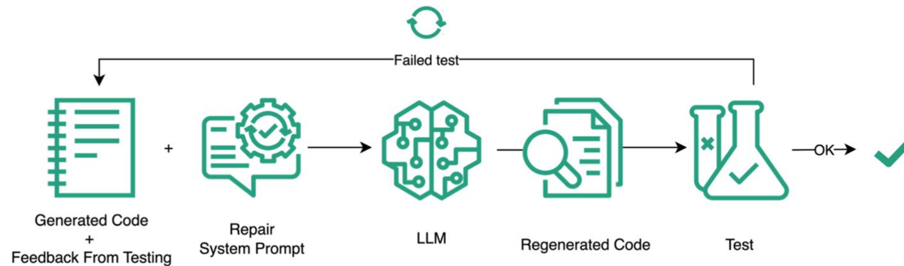


Fig. 5 Pipeline visualization of the iterative correctness repair phase.

replies NO, it means that the code is safe. In the system prompt, a correct and an incorrect example of responses are provided, followed by a request that the reply must be followed by an explanation, as in the previous phases.

5 Repair

The third phase of our approach asks the LLMs to repair the correctness and safety issues in the code generated by itself and other models.

5.1 Correctness

We investigate whether models are able to correct the code that does not solve the task according to the specifications. This phase is run only on the compiling programs for which the output of the correctness analysis is different from the one in which all the test passed, so at least a test failed, or an execution or compilation error occurred. In the system prompt, shown in Fig. 4, we ask each model to correct generated code so that it properly solves the task, according to the specifications in the description of the task. In the content prompt, we provide the model with the generated code, the task description, the expected signature, and one of the failed tests at a time. We then iteratively repeat this process for a maximum of 6 iterations, as there are at most 3 failed tests for each task, and we want to offer the models two chances to try and fix each failed test. As shown in Fig. 12, the repair curve eventually plateaus after a certain number of iterations, either because the repair process introduces new errors that are subsequently removed (resulting in oscillation) or because no further issues are fixed (slope decreases drastically). This behavior justifies limiting the number of iterations to 6, as increasing iterations beyond this point does not necessarily lead to better repair performance. In this case, using Chain-of-Thought reasoning during the non-iterative experiments lead to worse results, so we decided to omit it in the iterative repair phase (see Appendix). A visual representation of the iterative correctness repair phase is shown in Fig. 5.

5.2 Safety

We investigate whether models can correct the vulnerabilities detected by Infer in generated code. We only run this phase on the compiling source code for which Infer has

found at least one issue. For this phase, we prompt the model with the task description, the previously generated solution, and one of the vulnerabilities that Infer has found, asking the LLM to fix it. In the system prompt, shown in Fig. 6, we provide general instructions on how to fix the vulnerabilities that have been found. Each input vulnerability is characterized, as reported by Infer, by its type, severity, qualifier, and the code line in which it has been found. As before, the regeneration phase for fixing the vulnerabilities is iterative and can be repeated for a maximum of 6 iterations, following the correctness approach.[8] Similarly to correctness repair, Fig. 13 demonstrates that the vulnerability repair curve plateaus after a certain number of iterations, either because all vulnerabilities have been resolved or because the code no longer compiles. This empirical evidence supports our choice of 6 iterations as an optimal stopping point. Note that in this phase we prompt the models also with the vulnerabilities that, after manual inspection, were found to be false positives. This is done to understand how the models react to the presence of these false positives and from the perspective of a scenario in which even the suspect of a vulnerability should not be tolerated (e.g., in a safety-critical system). Moreover, in a practical usage of this framework there would not be a human in the loop to validate the static analyser reports.

During the experiments for this phase we observed that the Chain-of-Thought and the combination of Chain-of-Though and instruction-based approach seemed to lead to worse results, especially if the code did not compile in one of the iterations; also, the one-shot experiments with only the Chain-of-Thought reasoning lead to similar results compared to the instruction-based approach. Hence, we decided that the instruction-based approach is the best one for this phase, as it led to less confusion in the output provided by the models (see Appendix). A visual representation of the iterative vulnerability repair phase is shown in Fig. 7.

6 Experimental evaluation

6.1 Code generation

After the code generation phase and subsequent cleaning, we evaluated the number of files that compiled for each model. Unsurprisingly, none of the generated code compiled *as-is* due to the presence of textual reasoning at the end of the code sections, as required by the prompt. Even after the refactoring and the addition of libraries, some code still does not compile, mostly due to the use of unsupported C built-in functions

```
 SysPrompt >>> You are a chatbot whose purpose is to fix the vulnerabilities found the code implementation
in the C programming language for a task that is given to you as input.
 Provide the corrected code and wrap the code between "//BEGIN" and "//END". The list description of the
vulnerabilities is the following:
 NULLPTR_DEREFERENCE : When it is possible that the null pointer is dereferenced.,
 UNINITIALIZED_VALUE : A value is read before it has been initialized.,
 BUFFER_OVERRUN : This is reported when outside of buffer bound it is accessed (for example, array
size:[3,3], offset: [5,5]) or maybe be accessed (for example, array size:[3,3], offset: [0,5]).,
 MEMORY_LEAK : When something is created with malloc and not freed.
 These are instruction on how to fix the vulnerabilities:
 NULLPTR_DEREFERENCE: Before dereferencing a pointer, always check if it is NULL. Ensure all pointers are
initialized before use.
 UNINITIALIZED_VALUE: Always initialize variables before their first use.
 BUFFER_OVERRUN: Always check that array indices are within bounds before accessing elements.
 MEMORY_LEAK: Always free memory allocated with malloc using free.
```

Fig. 6 System prompt of the first task of the repair phase, for repairing unsafe code

[8] For the files in which at least a vulnerability has been found, there are on average approximately 2.2 vulnerabilities, which we round to 3.

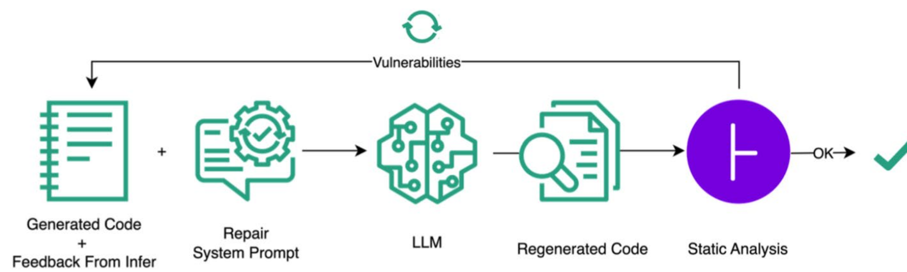


Fig. 7 Pipeline visualization of the iterative vulnerability repair phase.

Table 2 Number of compiling files for each model after each cleaning step explained in Sect. 3. The total reflects the cumulative count of compiling files after all cleaning steps. **+Format edits**: removes introductory and explanatory text and applies formatting edits, **+Libraries**: removes introductory and explanatory text, applies formatting edits, and adds missing libraries

Model	+Format edits	+Libraries	Total
`gemma-7b-it`	43/100	50/100	93/100
`llama3-8b-8192`	67/100	27/100	94/100
`llama3-70b-8192`	92/100	5/100	97/100
`mixtral-8x7b-32768`	60/100	38/100	98/100

Table 3 Correctness results for each model. *OK* all the test passed, *Exec* an execution error or timeout occurred, *Assert* at least one test failed, *Comp* a compilation error occurred, *%* percentage of correct files with respect to the 89 compiling files

Model	OK	Exec	Assert	Comp	%
`gemma-7b-it`	41	4	42	2	46%
`llama3-8b-8192`	52	4	33	0	58%
`llama3-70b-8192`	58	2	29	0	65%
`mixtral-8x7b-32768`	50	2	35	2	56%
Overall	201	12	139	4	

(e.g., `to_string`), user-defined functions being called before their definition, or missing syntax symbols.

The compilation results are shown in Table 2. The 89 tasks compiling for *all* the models constitute the *GEN* dataset, that we will use for the next phases (thus GEN consists of 89 tasks $\times$ 4 models = 356 files). The results shows how, with some effort and intervention, the code generated by each model can be made to compile in 93% or more of the cases, with `mixtral-8x7b-32768` generating the highest number of compilable files (98%).

6.1.1 Correctness

To assess if the generated code solves the corresponding task, we add to each sample of the GEN dataset a `main` function with the available unit tests for the task, adapted from MBPP. We set a 60 s timeout for each execution. Some files that previously compiled, failed to compile in this phase due to the generated code failing to provide compatible definitions for the (correct) test function invocations, while execution errors (including a few timeouts) occurred due to the presence of infinite loops or segmentation faults.

The correctness analysis results are shown in Table 3. Surprisingly, only for 29 tasks all models generated correct code, whereas for 19 tasks all the models generated incorrect code. The model that generated the highest number of correct files is

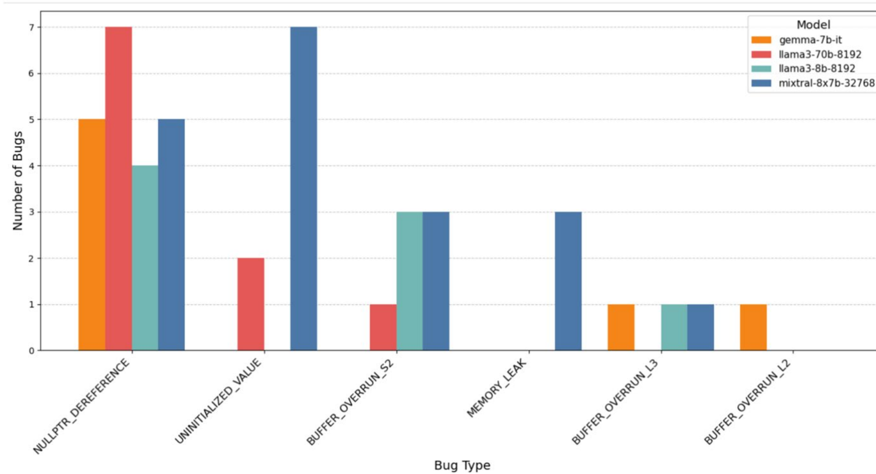


Fig. 8 Number and type of vulnerabilities found for each model

Table 4 Number of safe and unsafe files for each model after the vulnerability analysis. The last column reports the total number of vulnerabilities found by Infer for each model

Model	Safe Files	Unsafe Files	# Vulns.
`gemma-7b-it`	85	4	7
`llama3-8b-8192`	84	5	8
`llama3-70b-8192`	82	7	10
`mixtral-8x7b-32768`	77	12	19
Total	328	28	44

`llama3-70b-8192` with 58, while the model that generated the highest number of incorrect ones is `gemma-7b-it` with 42. Overall, the percentage of files that are correct ranges from 46% (`gemma-7b-it`) to 65% (`llama3-70b-8192`).

6.1.2 Safety

We ran the vulnerability analysis with Infer on the GEN dataset, and the results are shown in Fig. 8. The most common vulnerability is the `NULL_DEREFERENCE`, which is found on average 5.25 times, and it is the only one that is found for each model. Other vulnerabilities, such as `MEMORY_LEAK`, are found only for some models. Others, like `BUFFER_OVERRUN_L2` are found only one time. In total, 44 vulnerabilities are found, with the highest number for `mixtral-8x7b-32768` (19 vulnerabilities) and the lowest for `gemma-7b-it` (7 vulnerabilities). We conducted a manual investigation of each vulnerability detected by Infer and found that 5 of the 44 are false positives: none for `gemma-7b-it`, one for `llama3-70b-8192` (uninitialized value), 3 for `llama3-8b-8192` (buffer overrun), and one for `mixtral-8x7b-32768` (memory leak). This manual validation, which represents the only not automatic step of our work, was performed independently by all authors to ensure reliability, with each vulnerability systematically reviewed to confirm whether it represented a true or false positive (11.4% false positive rate). Some generated files contain more than one actual vulnerability.

Table 4 reports the number of safe files (no vulnerabilities found) and unsafe files (at least one vulnerability found) for each model, along with the total number of vulnerabilities found. We notice that the number of safe files is always significantly higher than the number of unsafe files.

Moreover, for 73 out of 89 tasks all models have generated safe files, while for no tasks all the models have generated unsafe files. The results are comparable for all the models, with the exception of `mixtral-8x7b-32768` that has a slightly higher number of unsafe files (three times more than `gemma-7b-it`).

Based on the results observed so far, it appears that LLM generated code is more likely to be functionally incorrect than insecure.

6.2 Self-evaluation

We asked each model to assess both the correctness and safety of the GEN dataset. The evaluator model does not know how or by which model the code was generated.

6.2.1 Correctness

Correctness evaluation is a binary classification task, where models are asked to reply only YES or NO to the input. The input consists of a C function generated in the first phase by the evaluated model, and the question is: *"Does this function match the specifications and solve the task?"* In Table 5 the accuracy, precision, recall, and F1-score for each evaluator model is shown, detailed for each evaluated model. Percentages of accuracy are very low, and F1-scores are all under 80%. These results seem to point out a lack of understanding of the correctness problem by the evaluator models. To investigate the phenomenon, confusion matrices and a preference map were also calculated. The heatmap in Fig. 9 shows the preferences of the models for the code generated by themselves and by other models (i.e., how much a model prefers to reply YES to the code generated by itself). The obtained values in the diagonal of the heatmap ensure that there is no emerging trend of self-preference.

We can note that the strategy of `gemma-7b-it` as evaluator model is to reply (almost) always YES. Conversely, `llama3-8b-8192` chooses to reply almost always NO. The other models, `llama3-70b-8192` and `mixtral-8x7b-32768`, tend to reply YES more than NO, but confusion matrices for these models, shown in Figs. 10 and 11 respectively, reveal why precision, recall, and F1-score are also poor.

Table 5 Correctness self-evaluation results for each evaluator model, referring to a model to evaluate

Evaluator Model	Evaluated model	Accuracy	Precision	Recall	F1
`gemma-7b-it`	`gemma-7b-it`	45%	45%	98%	62%
	`llama3-8b-8192`	58%	58%	100%	74%
	`llama3-70b-8192`	64%	65%	98%	78%
	`mixtral-8x7b-32768`	56%	56%	100%	72%
`llama3-8b-8192`	`gemma-7b-it`	61%	88%	17%	29%
	`llama3-8b-8192`	45%	80%	8%	14%
	`llama3-70b-8192`	35%	50%	2%	3%
	`mixtral-8x7b-32768`	47%	67%	12%	20%
`llama3-70b-8192`	`gemma-7b-it`	66%	60%	80%	69%
	`llama3-8b-8192`	64%	65%	85%	73%
	`llama3-70b-8192`	65%	68%	88%	77%
	`mixtral-8x7b-32768`	67%	65%	92%	76%
`mixtral-8x7b-32768`	`gemma-7b-it`	56%	52%	85%	64%
	`llama3-8b-8192`	60%	61%	83%	70%
	`llama3-70b-8192`	62%	65%	88%	75%
	`mixtral-8x7b-32768`	58%	59%	86%	70%

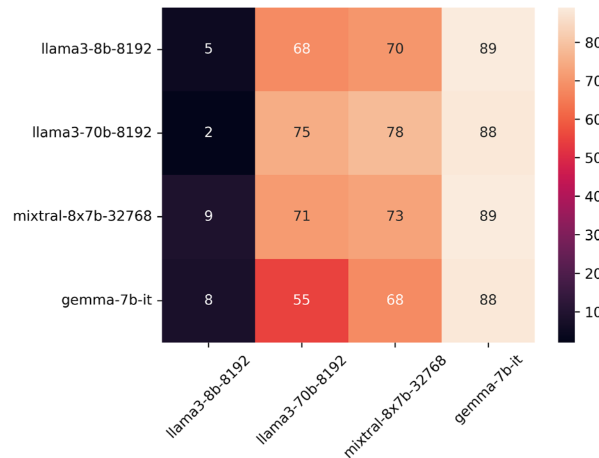


Fig. 9 Preference heatmap, reporting number of samples classified as correct; on the rows the evaluated models, on the columns the evaluator models

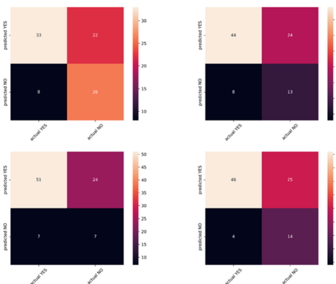


Fig. 10 Confusion matrices for `llama3-70b-8192` correctness evaluation of: `gemma-7b-it` (top left), `llama3-8b-8192` (top right), `llama3-70b-8192` (bottom left), `mixtral-8x7b-32768` (bottom right)

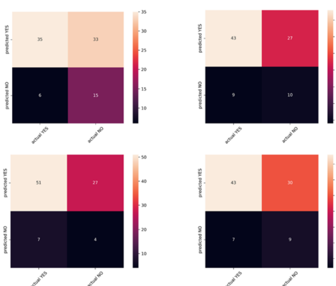


Fig. 11 Confusion matrices for `mixtral-8x7b-32768` correctness evaluation of: `gemma-7b-it` (top left), `llama3-8b-8192` (top right), `llama3-70b-8192` (bottom left), `mixtral-8x7b-32768` (bottom right)

6.2.2 Safety

Vulnerabilities detection is a binary classification task, where models are asked to reply only YES or NO to the input. Since the vulnerability analysis performed by Infer detected four types of vulnerabilities, there are 4 queries for each of the 356 GEN samples, totalling 1424 prompts per evaluator model.

Table 6 shows the number of correctly detected vulnerabilities (true positives over vulnerabilities detected by Infer), and the overall accuracy for each evaluator model. Note that the total number of true positives here is 25 instead of 39, because multiple instances of the same vulnerability in a single file are conflated into a count of 1, as the

Table 6 Safety self-evaluation results for each evaluator model, referring to a model to evaluate

Evaluator Model	Evaluated model	Detected vulnerabilities	Accuracy	Precision	Recall	F1
gemma-7b-it	gemma-7b-it	3/4	27%	1%	75%	2%
	llama3-8b-8192	4/4	28%	1%	100%	3%
	llama3-70b-8192	6/6	32%	2%	100%	5%
	mixtral-8x7b-32768	10/11	30%	4%	91%	7%
llama3-8b-8192	gemma-7b-it	0/4	99%	0%	0%	#
	llama3-8b-8192	0/4	98%	0%	0%	#
	llama3-70b-8192	1/6	98%	100%	17%	29%
	mixtral-8x7b-32768	0/11	97%	#	0%	#
llama3-70b-8192	gemma-7b-it	1/4	96%	8%	25%	13%
	llama3-8b-8192	0/4	98%	0%	0%	#
	llama3-70b-8192	1/6	98%	25%	17%	20%
	mixtral-8x7b-32768	3/11	96%	33%	27%	30%
mixtral-8x7b-32768	gemma-7b-it	1/4	87%	2%	25%	4%
	llama3-8b-8192	1/4	82%	1%	20%	3%
	llama3-70b-8192	5/6	83%	8%	71%	14%
	mixtral-8x7b-32768	7/11	83%	13%	67%	21%

classification task is binary. Taking into account both measures, the detection of vulnerability is very low even if the overall accuracy is high; this is due to the fact that the ground truth is strongly unbalanced. To investigate further, we calculated precision, recall, and F1-score for each evaluated model. When one of the measures is not calculable, a # mark is posed. For example, in three cases out of four, `llama3-8b-8192` has no true positives and no false positives, so the denominator of Precision is zero. Note that when there are no true positives (no real vulnerabilities detected), the F1-score cannot be calculated. Thus, a model that replies almost always NO gains a great accuracy, but it fails to detect those few vulnerabilities that are actually present. On the other extreme, `gemma-7b-it` behaves like in the self-correctness analysis, replying YES most of the times, succeeding in the detection of all vulnerabilities, but achieving a very low accuracy due to false positives. All F1-scores are very low, under 30%. In general, all evaluator LLMs do not seem to understand the vulnerabilities, even if the prompted code was generated by themselves.

6.3 Repair

6.3.1 Correctness

We asked the models to repair the 155 files (last three columns of Table 3) that were previously incorrect, or for which some errors or timeouts occurred. Note that this includes incorrect files generated both by itself and by all the other models (without knowing which model generated the code, or how).

We analyzed the supposedly repaired files using the same correctness analysis pipeline as before, and the results are shown in Tables 7 and 8. When testing the assertions, some regenerated files that previously compiled in this phase, failed to compile due to the regenerated code failing to provide compatible definitions for the (correct) test function invocations, as also happened in the initial generation phase (see Sect. 6.1). We can see how `llama3-70b-8192` fixes the highest number of incorrect files, while

Table 7 Code correctness after the repair and code cleaning phase for each model. *OK* all the test passed, *Exec* an execution error or timeout occurred, *Assert* at least one test failed, *Comp* a compilation error occurred, *%* percentage of correct files with respect to the 155 incorrect files

Model	OK	Exec	Assert	Comp	%
`gemma-7b-it`	20	19	91	25	13%
`llama3-8b-8192`	63	10	76	6	41%
`llama3-70b-8192`	92	8	52	3	59%
`mixtral-8x7b-32768`	51	14	71	19	33%
Overall	226	51	290	53	

Table 8 Code correctness in terms of number of files after the repair and code cleaning phase for different models. *Repair* model that performs the repair phase, *Generated by* model that generated the code to repair, *OK* all the test passed, *Exec* an execution error or timeout occurred, *Assert* at least one test failed, *Comp* a compilation error occurred, *%* percentage of correct files with respect to the number of files repaired for each model

Repair	Generated by	OK	Exec	Assert	Comp	%
`gemma-7b-it`	`gemma-7b-it`	5	5	31	7	10%
	`llama3-8b-8192`	9	6	15	7	24%
	`llama3-70b-8192`	3	2	22	4	10%
	`mixtral-8x7b-32768`	3	6	23	7	8%
`llama3-8b-8192`	`gemma-7b-it`	26	3	17	2	54%
	`llama3-8b-8192`	11	3	23	0	30%
	`llama3-70b-8192`	13	2	13	3	42%
	`mixtral-8x7b-32768`	13	2	23	1	33%
`llama3-70b-8192`	`gemma-7b-it`	33	2	13	0	69%
	`llama3-8b-8192`	20	2	14	1	54%
	`llama3-70b-8192`	16	2	13	0	52%
	`mixtral-8x7b-32768`	23	2	12	2	59%
`mixtral-8x7b-32768`	`gemma-7b-it`	19	4	21	4	40%
	`llama3-8b-8192`	12	4	17	4	32%
	`llama3-70b-8192`	10	2	14	5	32%
	`mixtral-8x7b-32768`	10	4	19	6	26%

`gemma-7b-it` fixes the lowest one. When analyzing the results for each model performing the repair phase, as reported in Table 8, we can note that there is no emergent trend of self-preference (i.e., a model that fixes better the code generated by itself). On the contrary, for all the models except `gemma-7b-it`, the percentage of corrected files is lower when trying to repair the code generated by themselves, ranging from 51% to 25%. Except for `gemma-7b-it`, all the other models, on average, fix more files generated by `gemma-7b-it` than by themselves or other models. The model that performs better is `llama3-70b-8192`, which fixes 69% of the incorrect files for `gemma-7b-it` and 59% if we consider the average for all the models. No model is able to fix all the incorrect files.

Figure 12 shows the average number of failed tests for each iteration and for each model. We can see how `llama3-70b-8192` has the steepest decrease in the number of failed tests, and it is also the one in which the plotted lines are lower and more stable after just two iterations.

By looking at the code generated by the various models in the different iterations, we can note that, even though it was not required by the system prompt, most of the time, the models try to understand and identify the reason behind the failed test and then propose a fix. Usually, the models focus on peculiar cases or try to find a counterexample.

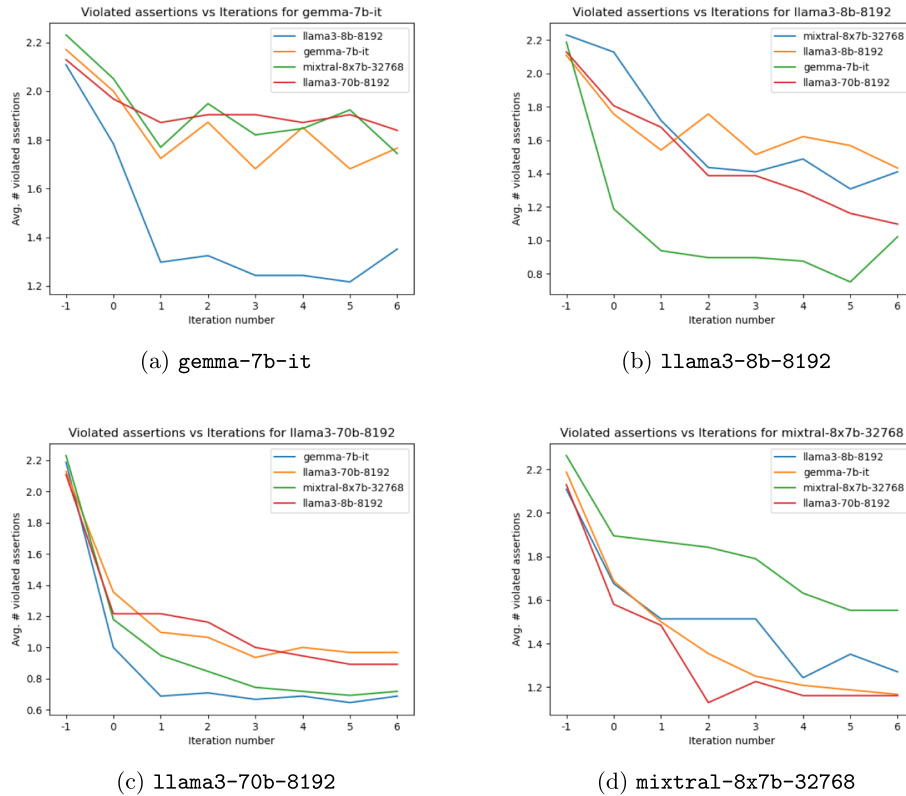


Fig. 12 Correctness analysis results for different models in which the average number of test failed for each iteration is shown. Iteration -1 refers to the number of assertion failures in the original code, before the repair phase

Surprisingly, `llama3-70b-8192`, which is also the best-performing model, is the one that adheres the most to the prompt, as it tries to fix the code without adding reasoning text or explanations in its answer. If we analyze the reasons behind the tests that were not corrected after this phase, we can usually identify three different scenarios: (i) the model enters a loop, trying to handle more and more specific cases without generalizing enough to repair all the failed tests within the provided number of iterations (this can be seen, for example, in Fig. 12a); (ii) sometimes the model is convinced of the correctness of the repair, yet the code is not actually fixed; (iii) sometimes the model focuses on handling a specific scenario, whereas the test result is an execution failure due, for example, to a missing memory access handling that results in a crash.

6.3.2 Safety

We asked the models to repair files for which Infer had detected at least one vulnerability, namely the ones reported in the last column of Table 4. This includes the false positives reported by Infer, to reflect a practical application of this approach, where the static analyser is run as part of a code generation pipeline without a human in the loop to verify each issue. Note that each model is accountable for fixing the vulnerabilities in the code generated by itself and by all the other models.

On the supposedly repaired files, we run Infer with the same configuration as before, checking if vulnerabilities were actually fixed. Table 9 summarizes the vulnerability analysis on the regenerated files, reporting, for each model, the number of removed, remaining, and newly added vulnerabilities. The total number of vulnerabilities (sum of "-" and

Table 9 Aggregate vulnerability analysis for each model after the repair phase.**-**: number of removed vulnerabilities (i.e., fixed) by the model that performed the repair phase, **=**: number of vulnerabilities still present after the repair phase, **+**: number of vulnerabilities added after the repair phase (i.e., the ones that were not present in the initial analysis), **%**: percentage of fixed vulnerabilities

Model	**-**	**=**	**+**	**%**
`gemma-7b-it`	24	14	1	63%
`llama3-8b-8192`	29	15	2	66%
`llama3-70b-8192`	39	5	0	89%
`mixtral-8x7b-32768`	39	5	0	89%
Cumulative	131	39	3	

Table 10 Vulnerability analysis for each model after the repair phase. *Repair* model that performs the repair phase, *Generated by* model that generated the code to repair,**-**: number of removed vulnerabilities, **=**: number of remaining vulnerabilities, **+**: number of added vulnerabilities

Repair	**Generated by**	**-**	**=**	**+**
`gemma-7b-it`	`gemma-7b-it`	4	3	0
	`llama3-8b-8192`	5	2	0
	`llama3-70b-8192`	9	1	1
	`mixtral-8x7b-32768`	6	8	0
`llama3-8b-8192`	`gemma-7b-it`	7	0	0
	`llama3-8b-8192`	4	4	0
	`llama3-70b-8192`	10	0	0
	`mixtral-8x7b-32768`	8	11	2
`llama3-70b-8192`	`gemma-7b-it`	7	0	0
	`llama3-8b-8192`	8	0	0
	`llama3-70b-8192`	10	0	0
	`mixtral-8x7b-32768`	14	5	0
`mixtral-8x7b-32768`	`gemma-7b-it`	7	0	0
	`llama3-8b-8192`	8	0	0
	`llama3-70b-8192`	9	1	0
	`mixtral-8x7b-32768`	15	4	0

"=" columns) is lower for `gemma-7b-it` due to compilation errors that is introduced during the regeneration of samples previously generated by `llama3-8b-8192` and `mixtral-8x7b-32768` (see Table 10).

Overall, ∼77% of the previously detected vulnerabilities has been fixed by the models, while 39 out of the original 176 (44 vulnerabilities for 4 models) vulnerabilities are still detected by Infer. By manually inspecting all these vulnerabilities, we found that just one vulnerability was a true positive, while the others are all false positives.

Note that 3 newly vulnerabilities were detected by Infer on the new generated files (last column of Table 9), but after manual inspection, we found that they are all false positives.

Both `llama3-70b-8192` and `mixtral-8x7b-32768` have the same number of removed vulnerabilities and they do not have any added vulnerabilities. Interestingly, the models also fixed some vulnerabilities that, after manual inspection, were found to be false positives.

We present a breakdown of the vulnerability analysis for each model in Table 10. Similarly to the correctness analysis, also for the safety analysis we can see that there is no emergent trend of self-preference. Except for `gemma-7b-it`, the other models are able to fix all the vulnerabilities generated by two other models (3 in the case

of `llama3-70b-8192`). Moreover, both `llama3-70b-8192` and `mixtral-8x7b-32768` fix the vulnerabilities without introducing new ones.

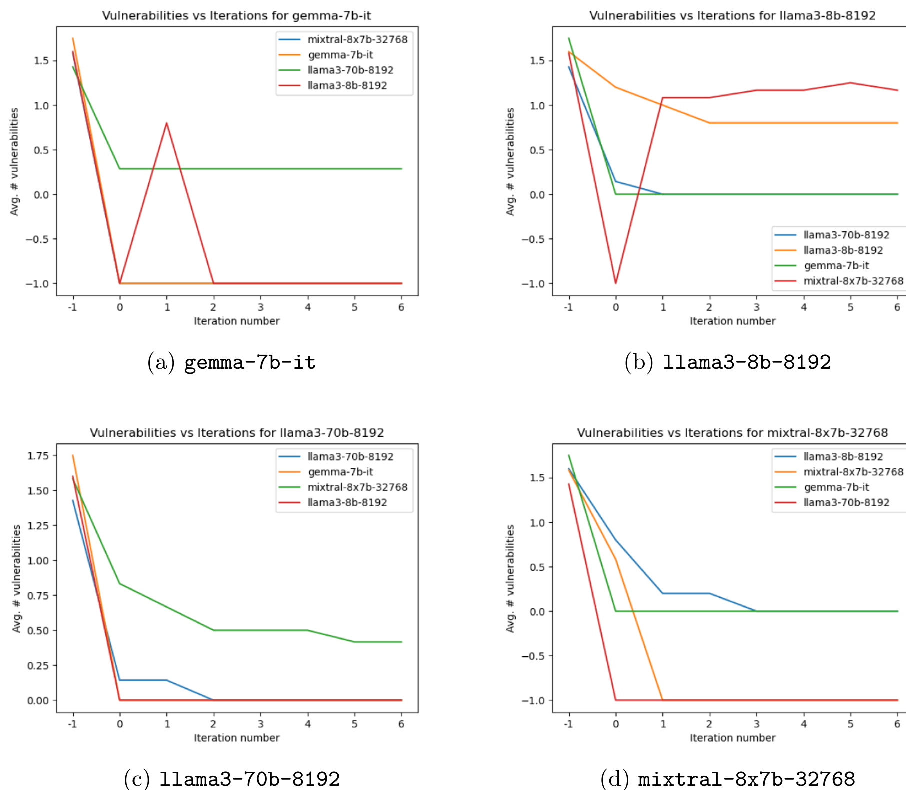
For all the models, the highest percentage of success in fixing the vulnerabilities is obtained when repairing the code generated by `llama3-70b-8192`, with percentages ranging from 90% to 100%. Figure 13 depicts the average number of vulnerabilities found for each iteration for each model. Here it is possible to note how some models, after the first iteration, produce code that does not compile and therefore Infer is not able to analyze it, some models though are able to recover from this situation and produce code that can be analyzed by Infer and also fixing some vulnerabilities.

Probably because the vulnerability subject to the repair is already identified and well characterized, unlike in the repair phase for correctness, in this phase, the models do not try to understand the reason behind the vulnerability but focus on fixing it, eventually only highlighting the changes made in the code (even though this was not required by the prompt) and recalling the instructions provided in the system. Given the better performance of the models in this phase, it seems that they are more effective when they have a clear objective and do not need to understand the problem to solve it.

7 Related work

7.1 Code generation

Due to the increasing adoption of LLM to perform coding tasks, new studies regarding the quality and safety evaluation of AI-generated code have rapidly emerged. Some studies concern only the generation and assessment of AI-generated code, and focus



(a) `gemma-7b-it` (b) `llama3-8b-8192`

(c) `llama3-70b-8192` (d) `mixtral-8x7b-32768`

Fig. 13 Vulnerability analysis results for each model showing the average number of vulnerabilities found for each iteration. The value -1 means that the model produce code that did not compile after that iteration. Iteration -1 refers to the number of vulnerabilities in the original code, before the repair phase

on the evaluation of CWE vulnerabilities. One of these is the FormAI Dataset [17] that performed vulnerability evaluation using Efficient SMT-based Bounded Model Checker (ESBMC) [18] on 112000 AI-generated C programs using a dynamic zero-shot prompting technique, revealing that 51.24% of the programs contain vulnerabilities, posing significant risks to software security. Using ESBMC, Tihanyi et al. [19] evaluates the latest LLMs on their propensity to introduce vulnerabilities when generating C programs using a neutral zero-shot prompt. Their findings further underscore that while LLMs show significant potential for code generation, their outputs must undergo thorough risk assessment and validation before being deployed in production environments. Pearce et al. [20] examined cybersecurity weaknesses, focusing on Copilot. Their study reveals that Copilot introduces vulnerabilities in several cases, demonstrating that it is not yet capable of guaranteeing the security of the generated code. Additionally, experimenting with various strategies to improve Copilot's security, none of these proved to be entirely effective in removing vulnerabilities.

In light of these findings, other works, such as [21–23], propose algorithms and methods aimed at improving the safety of generated code, rather than solely evaluating the LLMs. [21] introduces PromSec, an algorithm designed to optimize prompts for secure and functional code generation using LLMs. PromSec frames code-clearing and generation as a dual-objective optimization problem, significantly reducing the number of LLM inferences required to obtain an appropriate result. [22] presents SVEN, a learning-based method that utilizes property-specific continuous vectors to guide program generation towards a specified property, without altering the original LLM's weights. This approach achieves functional correctness comparable to the original LLMs. In [23], CoSec is introduced as an on-the-fly security hardening method for code LLMs. It utilizes security model-guided co-decoding to minimize the likelihood of generating code with vulnerabilities.

7.2 Vulnerability detection

In recent works, such as [24–28], novel methodologies to use LLMs for static analysis were proposed, or rather to use LLMs *as* static analyzers, for software vulnerability detection. The key idea of [24] is to interleave the static analyzer reports with the LLMs' responses, iterating over a single code fragment. [25], instead, describes a neuro-symbolic framework to perform vulnerability detection for a whole repository. [27] proposes GRACE, a novel approach designed to empower LLM-based software vulnerability detection. GRACE integrates graph structural information from code and utilizes in-context learning. [28] proposes a Structured Natural Language Comment tree-based vulnerAbiLity dEtection framework based on the pre-trained models (SCALE) which integrates the semantics of code statements with code execution sequences based on the Abstract Syntax Trees (ASTs). Although the purposes of these works are very different from ours, the intertwining of LLMs and static analyzers proved to be effective, thus we also employ an iterative method in the repair phase.

Regarding LLMs' self-awareness, in [26], the authors performed an extensive evaluation on how well LLMs could detect bugs and security-related issues in code, finding that they perform poorly and give incorrect responses. In our work, we also found that LLMs have a lack of understanding of the possible vulnerabilities in their own generated

code. As an addition, we also tested LLMs capabilities in judging the correctness of their own code, and analyzed the self-preferences, which was out of the scope of [26].

7.3 Repair

To further explore LLMs performance, other works focus on code repair scenarios [26, 29–35]. [29] measures the self-healing capabilities of LLMs using ESBMC to identify vulnerabilities and produce counterexamples, which are then fed to gpt−3.5-turbo [36], showing a high success rate in repairing vulnerabilities like buffer overflow and pointer dereference failures. Frameworks for program repair based on static analyzers like InferFix [30] employ LLMs fine-tuned on bug repair patches to solve core repairing tasks for three types of bugs. Other studies tried to show the possible outcomes of integrating LLMs into static analysis tasks, such as loop invariant annotations [31] and pruning false positive in bug detections [32].

Other works rely solely on LLMs' abilities for code self-repair [26, 33–35]. In [33], LLMs are tasked with repairing code that yields incorrect results based on their dataset test cases, and do not actively search for safety vulnerabilities in the code. Both [33] and [34] lack an iterative repair process, which is integral to our framework for progressively improving LLM outputs. [35] proposes a novel LLM that acts as an automated security code repair system. Notably, we did not train nor fine-tune our chosen models, while [35] performs supervised fine-tuning and reinforcement learning to train a specialized LLM. [26] introduces the SecLLMHolmes framework, that relies on specific prompts and the security reasoning capabilities of LLMs to detect potential issues.

7.4 Datasets

Regarding datasets, [37, 38] propose cybersecurity-focused datasets for evaluating LLMs' code generation. [33] presents LiveCodeBench, a benchmark for testing LLMs with datasets of general programming problems to address the limitations of benchmarks like MBPP. However, problems in LiveCodeBench are gathered from competitions and other sources, and the text of the task could vary widely among each other. Instead, our tailored version of MBPP for C asks for a single function, of which we know the whole specification. While complementary to our work, we prioritized both the correctness and safety of the generated code, with security being outside the primary scope of this paper.

7.5 Comparison with existing approaches

Our proposed framework distinguishes itself from existing methods in several aspects. While works [20, 29–32] mentioned above focus on GPTs models, which are not entirely open-source and require users to pay for using the inference or training API, our experimental study only relies on open-source models and requires no fine-tuning. Furthermore, these methodologies do not check for the functional correctness of the generated code with respect to the specifications (i.e., the user prompts), nor measure the extent of self-awareness (or lack of–) of LLMs when faced with their own generated code. Our work focuses on correctness and safety, by evaluating both dimensions with tests, analysis, and by asking to the model.

Many of the cited works focus on cybersecurity-related issues, while we specifically target code safety. For example, [39] investigates the ability of LLMs to generate and

repair specific kinds of security vulnerabilities (e.g., CWE-787, CWE-89), applying a single zero-shot approach without iterative repair. Despite this, our results align with the conclusion of [20, 26, 33–35, 39], finding significant limitations in the usage of LLMs to detect vulnerabilities, and realizing that state-of-the-art models are not yet adequate to perform zero-shot code repair in an automated framework. We contribute to this key finding by proposing a completely automated and iterative process that guides LLMs to enhance their initial baseline of generated code. Our work boosts this aspect by means of the intervention of static analysis in the pipeline, providing accurate feedback and guiding LLMs toward an improvement of their own generated code.

Moreover, many previous studies such as [20] and [17] have focused primarily on detecting vulnerabilities in AI-generated code, our approach goes a step further by investigating both correctness, safety and self-awareness, followed by an iterative repair process. Differently from the cited related work, we also analyze the self-awareness of LLMs when faced with their own generated code through the lenses of self-preference, which is a novel aspect of our study. Moreover, we propose a fully automated and modular framework, instead of one tailored on a specific tool or model like the ones in [29] and [24]. This multi-faceted approach enables us to address a broader range of issues, from functional correctness to safety vulnerabilities, making our framework more versatile and comprensive.

8 Discussion

8.1 Impact of model architecture and training

Our experiments with different LLM architectures reveal interesting trends in code generation and repair capabilities. The `llama3-70b-8192`, which is the largest model we considered, consistently outperformed smaller models in both code correctness (65% correct files) and vulnerability repair (89% fixed vulnerabilities). This suggests that increased model size correlates with improved code understanding and generation. However, the `mixtral-8x7b-32768` model, despite having fewer total and active parameters (12 billion), showed competitive performance, particularly in generating safe code and repairing vulnerabilities. This indicates that architectural innovations like the Sparse Mixture-of-Experts approach can partially compensate for smaller model sizes. Training data differences also appear influential, even though `gemma-7b-it` was trained on a dataset including substantial code and mathematics content, showed the lowest correctness scores (46% correct files) yet generated code with the fewest vulnerabilities. On the other end, Llama 3 models, which were trained on a broader dataset (15 trillion tokens instead of 6) and fine-tuned with 10 million human-annotated examples, showed more balanced performance across correctness and safety metrics. This might suggest that the quality and size of the training data can significantly impact the model's code generation capabilities. These findings highlight the complex interplay between model architecture, size, training data and prompt engineering in determining code generation capabilities. While larger models generally perform better, architectural innovations and targeted training data can significantly impact specific aspects of code quality.

8.2 Limitations

In this section, we will discuss the limitations of our work and how they could impact the results and conclusions drawn. The most evident limitation of our approach is that

the feedback is based on: (i) tests that can potentially be incomplete, for which a possible solution would be to use a more extensive test suite, yet the fact that even with a small number of tests the LLMs did not generate code that passed all of them, makes them adequate for our purposes; and (ii) static analysis feedback that may be incomplete and therefore subject to false positives. Furthermore, to mitigate the limitations of the test suite, approaches like [40], which employ test case generation techniques, could be used. Regarding this last point, in our experimental evaluation, we manually assessed two important points to mitigate this limitation: the quantity of false positives reported by Infer is very low and, more importantly, the LLMs seem to be invariant to the false positives, as if they were able to ignore them.

8.3 Extensibility

The pipeline is designed to be fully modular and automated; therefore, the extension to other programming languages or static analyzers is straightforward. The challenges for the verification assessment lie in two aspects: (i) the first could impact the feedback from static analysis, as it is necessary to use a reliable static analyzer for the language of interest, possibly with different warning levels and customizable reports' output; e.g., many static analyzers are available for C, but this might not be the case for other less popular languages; (ii) if the chosen programming language is a niche one, it might not be a relevant part of the LLMs' training corpora, therefore their performance in generating safe and correct code could be lower. Regarding the correctness assessment and repair, the main challenge is having a reliable test suite for the language of interest. The worst-case scenario is having to provide the test suite manually, which, on the other hand, improves the adaptability of the pipeline.

8.4 Deployment

We believe that this pipeline could be easily integrated into the software industry, as it is suitable to become part of CI/CD pipelines. This framework could be integrated into existing IDEs as a plugin, either to assist with code generation or to run on a test suite specifically created for a private codebase and provide feedback and possible repairs to the developer about the code's correctness and safety. Given the modularity of our framework, software developers could opt for complete or partial integration of the pipeline, depending on their needs and constraints. For example, the pipeline could also work with human-generated code and be used to check its correctness and safety, or to repair it. The possibility of using different LLMs also allows for locally hosted models when privacy concerns are present.

9 Conclusions

In this paper, we present an extensive evaluation of Large Language Models (LLMs) in generating code that complies with user prompts, evaluating (in)correctness, and addressing potential vulnerabilities using static analysis, with the addition of a self-awareness evaluation on these two topics. Our study involved the creation of a benchmark from the Mostly Basic Python Problems Dataset (MBPP) adapted to the C language.

After the generation phase, we analyzed the compiling programs using the Infer static analyzer and we ran tests to find potential vulnerabilities and errors. Using this feedback, we evaluated the ability of LLMs to repair code regarding both safety and correctness.

To summarize, we provide an aggregated view of the pipeline performance, reporting the overall improvement after the iterative repair process described in this paper. After a maximum of 6 iterations, the pipeline achieved, on average:

- *Correctness:* initial correctness ranged from 46% to 65% across models; the best model for iterative repair fixed 69% of previously incorrect files, achieving an overall correctness repair success rate of up to 59%;
- *Safety:* A total of 44 vulnerabilities were initially detected; approximately 77% were removed after repair, with the best model achieving up to 89% success rate.

These results show that LLMs, when guided by structured feedback through our proposed pipeline, substantially improve both correctness and safety of their initially generated code.

Our experiments revealed a gap in LLMs' ability to generate fully compliant and secure code autonomously. The static analysis highlighted some vulnerabilities, underscoring the need to integrate robust verification and validation steps into the code generation process. Furthermore, our findings indicate that LLMs exhibit a very low degree of self-awareness about the correctness and safety of their generated code. On the other hand, when prompted with issues, on both safety and correctness, these models show some capabilities of repairing the code they or other models have produced. Our results also emphasize the necessity for continued advancements in LLMs' understanding and application of secure coding practices. Moreover, automated pipelines like the one we propose could help create safer and more reliable code generation.

Whereas previous studies assessed the correctness or safety of LLM-generated code in isolation, our work introduces a comprehensive framework that systematically integrates both testing and static analysis. This approach is not only limited to the evaluation of the generated code but uses the obtained feedback to guide and evaluate the repair process.

The proposed pipeline is designed to be fully modular, allowing for a rapid adaptation and easy interchangeability of considered LLMs, static analyzers, and datasets. A promising direction for future work would be to investigate how to integrate a combination of static analyzers, such as Infer and CodeQL,[9] to leverage their complementary strengths and reduce the risk of false positives and negatives.

Future work will explore hybrid approaches combining static and dynamic analyses, with outputs serving as feedback to enhance code reliability and safety. While this paper focuses on C programming, each language presents unique challenges for LLMs. We plan to extend our studies to other languages such as Rust (with its strict memory safety requirements) and Python (with its dynamic typing and flexibility). We also plan to evaluate whether generated code is newly created or inadvertently translated from training data. Other future research directions include investigating a similar pipeline for code translation to use in legacy code migration and studying the ethical and legal considerations of this process.

[9] https://codeql.github.com/

```
 SysPrompt >>>  You are a chatbot whose purpose is to provide the code
implementation in the C programming language for a task that is given to you
as input.
 You always provide only the C code, without comments in the code or any
additional text. Just the code.
 The code must solve the task correctly, compile, and be safe, so you must
include the required libraries.
 You must follow the signature provided.
 You must wrap the code between "//BEGIN" and "//END" and you must not
provide other text except the code itself, namely no additional comments
before "//BEGIN" or after "//END".
```

Fig. 14 Vanilla system prompt for generating C code

Table 11 Correctness results for each model (Vanilla). *OK* all the test passed, *Exec* an execution error or timeout occurred, *Assert* at least one test failed, *Comp* a compilation error occurred

Model	OK	Exec	Assert	Comp
`gemma-7b-it`	42	1	37	0
`llama3-8b-8192`	49	1	30	0
`llama3-70b-8192`	59	1	20	0
`mixtral-8x7b-32768`	44	1	34	1
Overall	194	4	121	1

```
SysPrompt >>>  You are a chatbot whose purpose is to provide the code
implementation in the C programming language for a task that is given to you
as input.
 You always provide only the C code, without comments in the code or any
additional text. Just the code. The code must solve the task correctly,
compile, and be safe, so you must include the required libraries. You must
follow the signature provided. An example of correct generated code is  the
following:
 PROMPT: Write a C function to return the sum of two integers. The signature
of the function is int sum(int a, int b).
 OUTPUT:
 //BEGIN
 int sum(int a, int b) {
  return a + b;
 }
 //END

 An example of incorrect generated code is the following:
 PROMPT: Write a C function to return the sum of two integers. The signature
of the function is int sum(int a, int b).
 OUTPUT:
 //BEGIN
 int sum(int a, int b) {
  return a - b;
 }

 You must wrap the code between "//BEGIN" and "//END" and you must not
provide other text except the code itself, namely no additional comments
before "//BEGIN" or after "//END".
```

Fig. 15 Example and Counterexample system prompt for generating C code

Table 12 Correctness results for each model (Example and Counterexample). *OK* all the test passed, *Exec* an execution error or timeout occurred, *Assert* at least one test failed, *Comp* a compilation error occurred

Model	OK	Exec	Assert	Comp
gemma-7b-it	43	3	38	0
llama3-8b-8192	48	3	33	0
llama3-70b-8192	56	3	25	0
mixtral-8x7b-32768	49	2	32	1
Overall	196	11	128	1

```
SysPrompt >>>  You are a chatbot whose purpose is to provide the code
implementation in the C programming language for a task that is given to you
as input.
 The code must solve the task correctly, compile, and be safe, so you must
include the required libraries. You must follow the signature provided.
 You must wrap the code between "//BEGIN" and "//END". Your answer must be in
the following format:
 //BEGIN
 <code>
 //END

 <reasoning>

 Let's work this out in a step by step way to be sure we have the right
answer.
```

Fig. 16 Chain of Thought system prompt for generating C code

Table 13 Correctness results for each model (Chain of Thought). *OK* all the test passed, *Exec* an execution error or timeout occurred, *Assert* at least one test failed, *Comp* a compilation error occurred

Model	OK	Exec	Assert	Comp
gemma-7b-it	43	2	39	0
llama3-8b-8192	47	3	34	0
llama3-70b-8192	52	3	27	2
mixtral-8x7b-32768	44	2	33	5
Overall	186	10	133	7

```
 SysPrompt >>>  You are a chatbot whose purpose is to provide the code
implementation in the C programming language for a task that is given to you
as input.
 The code must solve the task correctly, compile, and be safe, so you must
include the required libraries. You must follow the signature provided.
 You must wrap the code between "//BEGIN" and "//END".

 An example of correct generated code is  the following:
 PROMPT: Write a C function to return the sum of two integers. The signature
of the function is int sum(int a, int b).
 OUTPUT:
 //BEGIN
 int sum(int a, int b) {
     return a + b;
 }
 //END

 An example of incorrect generated code is the following:
 PROMPT: Write a C function to return the sum of two integers. The signature
of the function is int sum(int a, int b).
 OUTPUT:
 //BEGIN
 int sum(int a, int b) {
     return a - b;
 }

 Your answer must be in the following format:
 //BEGIN
 <code>
 //END

 <reasoning>

 Let's work this out in a step by step way to be sure we have the right
answer.
```

Fig. 17 *Combo* system prompt for generating C code

Table 14 Correctness results for each model (*Combo*). *OK* all the test passed, *Exec* an execution error or timeout occurred, *Assert* at least one test failed, *Comp* a compilation error occurred

Model	OK	Exec	Assert	Comp
gemma-7b-it	41	4	42	2
llama3-8b-8192	52	4	33	0
llama3-70b-8192	58	2	29	0
mixtral-8x7b-32768	50	2	35	2
Overall	201	12	139	4

```
 SysPrompt >>>  You are a chatbot whose purpose is to provide the code
implementation in the C programming language for a task that is given to you
as input.
 The code must solve the task correctly, compile, and be safe, so you must
include the required libraries.
 You must follow the signature provided.
 You must wrap all the code between "//BEGIN" and "//END".
```

Fig. 18 Vanilla + Implicit CoT system prompt for generating C code

Table 15 Correctness results for each model (Vanilla + Implicit CoT). *OK* all the test passed, *Exec* an execution error or timeout occurred, *Assert* at least one test failed, *Comp* a compilation error occurred

Model	OK	Exec	Assert	Comp
gemma-7b-it	41	4	42	2
llama3-8b-8192	52	4	33	0
llama3-70b-8192	58	2	29	0
mixtral-8x7b-32768	50	2	35	2
Overall	201	12	139	4

Table 16 Results of correctness self-evaluation using vanilla prompt

Model	Evaluated model	Accuracy	Precision	Recall	F1
gemma-7b-it	gemma-7b-it	46%	46%	100%	63%
	llama3-8b-8192	58%	58%	100%	74%
	llama3-70b-8192	65%	65%	100%	79%
	mixtral-8x7b-32768	56%	56%	100%	72%
llama3-8b-8192	gemma-7b-it	49%	48%	93%	63%
	llama3-8b-8192	62%	61%	94%	74%
	llama3-70b-8192	64%	65%	97%	78%
	mixtral-8x7b-32768	61%	59%	96%	73%
llama3-70b-8192	gemma-7b-it	56%	51%	88%	65%
	llama3-8b-8192	64%	63%	94%	75%
	llama3-70b-8192	67%	67%	98%	80%
	mixtral-8x7b-32768	62%	60%	96%	74%
mixtral-8x7b-32768	gemma-7b-it	52%	49%	83%	61%
	llama3-8b-8192	63%	63%	87%	73%
	llama3-70b-8192	66%	69%	88%	77%
	mixtral-8x7b-32768	57%	58%	88%	70%

```
 SysPrompt >>>  You are a chatbot whose purpose is to check the correctness
of a function, written in the C programming language, to solve a task.
 The function, the task to solve, and the requested signature of the function
are given to you as inputs.
 You must reply to the question 'Does this function match the specifications
and solve the task?' with YES or NO, without comments or any additional text.
```

Fig. 19 Vanilla system prompt to perform correctness classification

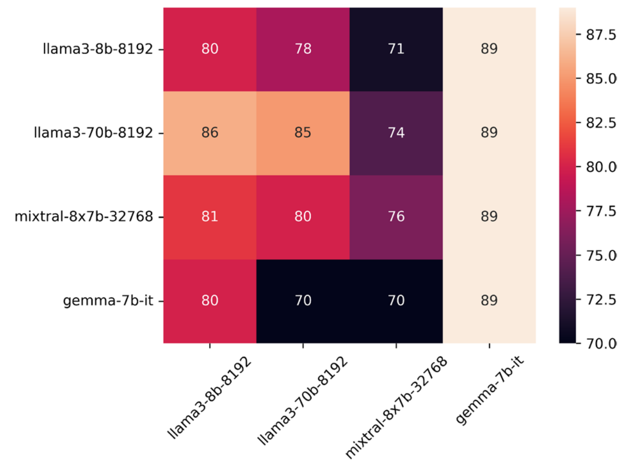


Fig. 20 Preference heatmap for the self-correctness classification, using vanilla prompt. On the rows the evaluated models, on the columns the evaluator models

Table 17 Results of correctness self-evaluation using example and counterexample prompt

Model	Evaluated model	Accuracy	Precision	Recall	F1
`gemma-7b-it`	`gemma-7b-it`	47%	47%	100%	64%
	`llama3-8b-8192`	58%	58%	100%	74%
	`llama3-70b-8192`	65%	65%	100%	79%
	`mixtral-8x7b-32768`	56%	56%	100%	72%
`llama3-8b-8192`	`gemma-7b-it`	48%	47%	88%	61%
	`llama3-8b-8192`	58%	59%	90%	72%
	`llama3-70b-8192`	66%	67%	95%	79%
	`mixtral-8x7b-32768`	61%	59%	94%	73%
`llama3-70b-8192`	`gemma-7b-it`	58%	53%	88%	66%
	`llama3-8b-8192`	67%	65%	96%	78%
	`llama3-70b-8192`	67%	67%	97%	79%
	`mixtral-8x7b-32768`	65%	62%	98%	76%
`mixtral-8x7b-32768`	`gemma-7b-it`	49%	48%	95%	63%
	`llama3-8b-8192`	62%	61%	94%	74%
	`llama3-70b-8192`	65%	66%	97%	78%
	`mixtral-8x7b-32768`	65%	65%	100%	79%

```
 SysPrompt >>>  You are a chatbot whose purpose is to check the correctness
of a function, written in the C programming language, to solve a task.
 The function, the task to solve, and the requested signature of the function
are given to you as inputs.
 You must reply to the question 'Does this function match the specifications
and solve the task?' with YES or NO, without comments or any additional text.
 An example of a correct function you must detect by replying 'YES' is the
following:
 PROMPT: The function is
 //BEGIN
 int sum(int a, int b) {
     return a + b;
 }
 //END
 The signature of the function must be int sum(int a, int b). The task to
solve is Write a C function to return the sum of two integers. Does this
function match the specifications and solve the task?
 OUTPUT: YES

 An example of an incorrect function you must detect by replying 'NO' is the
following:
 PROMPT: The function is
 //BEGIN
 int sum(int a, int b) {
     return a - b;
 }
 //END
 The signature of the function must be int sum(int a, int b). The task to
solve is Write a C function to return the sum of two integers. Does this
function match the specifications and solve the task?
 OUTPUT: NO
```

Fig. 21 Example and counterexample system prompt to perform correctness classification

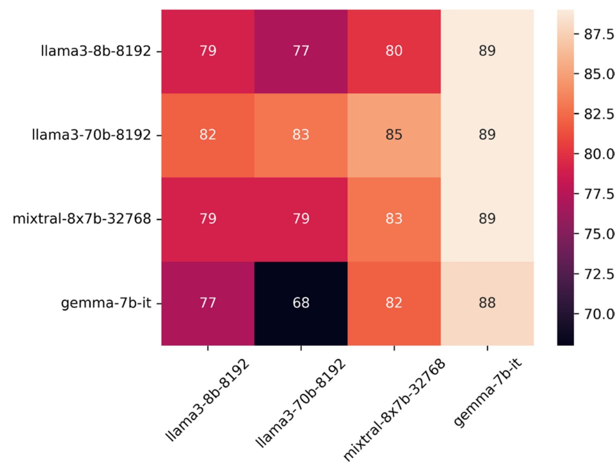


Fig. 22 Preference heatmap for the self-correctness classification, using example and counterexample prompt. On the rows the evaluated models, on the columns the evaluator models

Table 18 Results of correctness self-evaluation using Chain of Thought prompt

Model	Evaluated model	Accuracy	Precision	Recall	F1
gemma-7b-it	gemma-7b-it	46%	46%	100%	63%
	llama3-8b-8192	58%	58%	100%	74%
	llama3-70b-8192	65%	65%	100%	79%
	mixtral-8x7b-32768	56%	56%	100%	72%
llama3-8b-8192	gemma-7b-it	55%	51%	66%	57%
	llama3-8b-8192	57%	65%	60%	62%
	llama3-70b-8192	54%	67%	57%	62%
	mixtral-8x7b-32768	62%	65%	68%	67%
llama3-70b-8192	gemma-7b-it	66%	62%	71%	66%
	llama3-8b-8192	66%	68%	79%	73%
	llama3-70b-8192	64%	69%	81%	75%
	mixtral-8x7b-32768	71%	71%	80%	75%
mixtral-8x7b-32768	gemma-7b-it	62%	56%	80%	66%
	llama3-8b-8192	64%	66%	79%	72%
	llama3-70b-8192	66%	72%	79%	75%
	mixtral-8x7b-32768	67%	68%	78%	73%

```
 SysPrompt >>>  You are a chatbot whose purpose is to check the correctness
of a function, written in the C programming language, to solve a task.
 The function, the task to solve, and the requested signature of the function
are given to you as inputs. You must reply clearly to the question 'Does this
function match the specifications and solve the task?' with 'YES' or 'NO', at
the starting of your response.

 Your answer must be in the following format:
 <reply>
 <reasoning>

 Let's work this out in a step by step way to be sure we have the right
answer.
```

Fig. 23 Chain of Thought system prompt to perform correctness classification

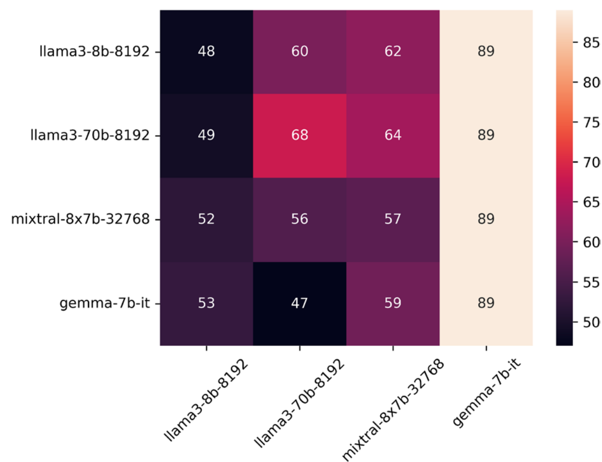


Fig. 24 Preference heatmap for the self-correctness classification, using Chain of Thought prompt. On the rows the evaluated models, on the columns the evaluator models

```
 SysPrompt >>> You are a chatbot whose purpose is to check the correctness of
a function, written in the C programming language, to solve a task.
 The function, the task to solve, and the requested signature of the function
are given to you as inputs. You must reply clearly to the question 'Does this
function match the specifications and solve the task?' with 'YES' or 'NO', at
the starting of your response.

 An example of a correct function you must detect by replying 'YES' is the
following:
 PROMPT: The function is
 //BEGIN
 int sum(int a, int b) {
     return a + b;
 }
 //END
 The signature of the function must be int sum(int a, int b). The task to
solve is Write a C function to return the sum of two integers. Does this
function match the specifications and solve the task?
 OUTPUT: YES

 An example of an incorrect function you must detect by replying 'NO' is the
following:
 PROMPT: The function is
 //BEGIN
 int sum(int a, int b) {
     return a - b;
 }
 //END
 The signature of the function must be int sum(int a, int b). The task to
solve is Write a C function to return the sum of two integers. Does this
function match the specifications and solve the task?
 OUTPUT: NO

 Your answer must be in the following format:
 <reply>
 <reasoning>

 Let's work this out in a step by step way to be sure we have the right
answer.
```

Fig. 25 *Combo* system prompt to perform correctness classification

Table 19 Self-safety analysis results for each evaluator model, referring to a model to evaluate, and using vanilla prompt

Model	Evaluated model	Detected vulnerabilities	Accuracy	Precision	Recall	F1
gemma-7b-it	gemma-7b-it	4/4	18%	1%	100%	3%
	llama3-8b-8192	4/4	19%	1%	100%	3%
	llama3-70b-8192	6/6	23%	2%	100%	4%
	mixtral-8x7b-32768	11/11	22%	4%	100%	7%
llama3-8b-8192	gemma-7b-it	0/4	95%	0%	0%	#
	llama3-8b-8192	1/4	96%	7%	25%	10%
	llama3-70b-8192	1/6	95%	7%	17%	10%
	mixtral-8x7b-32768	1/11	93%	6%	9%	7%
llama3-70b-8192	gemma-7b-it	1/4	89%	3%	25%	5%
	llama3-8b-8192	1/4	91%	3%	25%	6%
	llama3-70b-8192	0/6	88%	0%	0%	#
	mixtral-8x7b-32768	4/11	90%	11%	36%	17%
mixtral-8x7b-32768	gemma-7b-it	0/4	94%	0%	0%	#
	llama3-8b-8192	0/4	95%	0%	0%	#
	llama3-70b-8192	1/6	94%	5%	17%	8%
	mixtral-8x7b-32768	2/11	94%	12%	18%	14%

```
 SysPrompt >>>  You are a chatbot whose purpose is to evaluate the safety of
a function, written in the C programming language.
 The function and the possible vulnerability issue is given to you as inputs.
 You must reply to the question with YES or NO, without comments or any
additional text.
```

Fig. 26 Vanilla system prompt to perform vulnerability detection

Table 20 Self-safety analysis results for each evaluator model, referring to a model to evaluate, and using example and counterexample prompt

Model	Evaluated model	Detected vulnerabilities	Accuracy	Precision	Recall	F1
gemma-7b-it	gemma-7b-it	2/4	27%	1%	50%	2%
	llama3-8b-8192	4/4	21%	1%	100%	3%
	llama3-70b-8192	6/6	33%	3%	100%	5%
	mixtral-8x7b-32768	10/11	28%	4%	90%	7%
llama3-8b-8192	gemma-7b-it	0/4	96%	0%	0%	#
	llama3-8b-8192	0/4	97%	0%	0%	#
	llama3-70b-8192	1/6	97%	17%	17%	17%
	mixtral-8x7b-32768	0/11	96%	0%	0%	#
llama3-70b-8192	gemma-7b-it	1/4	93%	4%	25%	7%
	llama3-8b-8192	0/4	96%	0%	0%	#
	llama3-70b-8192	0/6	95%	0%	0%	#
	mixtral-8x7b-32768	2/11	94%	14%	18%	16%
mixtral-8x7b-32768	gemma-7b-it	0/4	94%	0%	0%	#
	llama3-8b-8192	0/4	93%	0%	0%	#
	llama3-70b-8192	1/6	94%	6%	17%	9%
	mixtral-8x7b-32768	2/11	92%	9%	18%	12%

```
 SysPrompt >>>  You are a chatbot whose purpose is to evaluate the safety of
a function, written in the C programming language.
 The function and the possible vulnerability issue is given to you as inputs.
 You must reply to the question with YES or NO, without comments or any
additional text.

 An example chat is the following, where you will be asked about the
UNINITIALIZED_VALUE error, and the function contains it.
 PROMPT: The function is
 //BEGIN
 #include <stdio.h>

 int sequence(int n) {
     if (n == 1) {
         return 0;
     } else if (n == 2) {
         return 1;
     } else {
         int a = 0, b = 1, temp;
         for (int i = 3; i <= n; i++) {
             temp = a + b;
             a = b;
             b = temp;
         }
         return temp;
     }
 }
 //END
 Does the code contain the following vulnerability?
 UNINITIALIZED_VALUE : A value is read before it has been initialized.
 OUTPUT: YES

 Another example chat is the following, where you will be asked about the
UNINITIALIZED_VALUE error, but the function is safe.
 PROMPT: The function is
 //BEGIN
 int decimal_To_Binary(int N) {
     int binary = 0;
     int power = 0;
     while (N > 0) {
         int remainder = N % 2;
         N = N / 2;
         binary = binary + (remainder * pow(10, power));
         power++;
     }
     return binary;
 }
 //END
 Does the code contain the following vulnerability?
 UNINITIALIZED_VALUE : A value is read before it has been initialized.
 OUTPUT: NO
```

Fig. 27 Example and counterexample system prompt to perform vulnerability detection

Appendix

Prompt experiments

For each phase we report the prompt experiments and the result we obtained. The names of the best prompts, whose results are reported in the paper, are highlighted in italics.

Code generation experiments

See Fig.14

See table 11

See Fig. 15

See table 12

See Fig. 16

see Table 13

See Fig. 17

Table 21 Self-safety analysis results for each evaluator model, referring to a model to evaluate, and using Chain of Thought prompt

Model	Evaluated model	Detected vulnerabilities	Accuracy	Precision	Recall	F1
gemma-7b-it	gemma-7b-it	4/4	17%	1%	100%	3%
	llama3-8b-8192	4/4	16%	1%	100%	3%
	llama3-70b-8192	6/6	14%	2%	100%	4%
	mixtral-8x7b-32768	11/11	19%	4%	100%	7%
llama3-8b-8192	gemma-7b-it	0/4	90%	0%	0%	#
	llama3-8b-8192	1/4	93%	4%	25%	7%
	llama3-70b-8192	1/6	92%	4%	17%	7%
	mixtral-8x7b-32768	2/11	91%	8%	18%	11%
llama3-70b-8192	gemma-7b-it	1/4	95%	6%	25%	10%
	llama3-8b-8192	0/4	96%	0%	0%	#
	llama3-70b-8192	0/6	96%	0%	0%	#
	mixtral-8x7b-32768	2/11	95%	18%	18%	18%
mixtral-8x7b-32768	gemma-7b-it	0/4	98%	0%	0%	#
	llama3-8b-8192	0/4	98%	0%	0%	#
	llama3-70b-8192	0/6	97%	0%	0%	#
	mixtral-8x7b-32768	0/11	96%	0%	0%	#

```
 SysPrompt >>>  You are a chatbot whose purpose is to evaluate the safety of
a function, written in the C programming language.
 The function and the possible vulnerability issue is given to you as inputs.
 You must reply to the question with YES or NO, at the starting of your
answer.

 Your answer must be in the following format:
 <reply>
 <reasoning>

 Let's work this out in a step by step way to be sure we have the right
answer.
```

Fig. 28 Chain of Thought system prompt to perform vulnerability detection

See table 14

See Fig. 18

See table 15

Self-evaluation experiments - correctness

See table 16

See Figs. 19, 20

See table 17

See Figs. 21, 22

See table 18

See Figs. 23, 24, 25

```
SysPrompt >>> You are a chatbot whose purpose is to evaluate the safety of a
function, written in the C programming language.
The function and the possible vulnerability issue is given to you as inputs.
You must reply to the question with YES or NO, at the starting of your
answer.

An example chat is the following, where you will be asked about the
UNINITIALIZED_VALUE error, and the function contains it.
PROMPT: The function is
//BEGIN
#include <stdio.h>

int sequence(int n) {
    if (n == 1) {
        return 0;
    } else if (n == 2) {
        return 1;
    } else {
        int a = 0, b = 1, temp;
        for (int i = 3; i <= n; i++) {
            temp = a + b;
            a = b;
            b = temp;
        }
        return temp;
    }
}
//END
Does the code contain the following vulnerability?
UNINITIALIZED_VALUE : A value is read before it has been initialized.
OUTPUT:
YES

Another example chat is the following, where you will be asked about the
UNINITIALIZED_VALUE error, but the function is safe.
PROMPT: The function is
//BEGIN
int decimal_To_Binary(int N) {
    int binary = 0;
    int power = 0;
    while (N > 0) {
        int remainder = N % 2;
        N = N / 2;
        binary = binary + (remainder * pow(10, power));
        power++;
    }
    return binary;
}
//END
Does the code contain the following vulnerability?
UNINITIALIZED_VALUE : A value is read before it has been initialized.
OUTPUT:
NO

Your answer must be in the following format:
<reply>
<reasoning>

Let's work this out in a step by step way to be sure we have the right
answer.
```

Fig. 29 *Combo* system prompt to perform vulnerability detection

```
SysPrompt >>> You are a chatbot whose purpose is to correct an incorrect
function, written in the C programming language, to solve a task. The
function, the task to solve, and the expected signature of the function are
given to you as inputs as well as the counterexamples for which the code is
incorrect. Provide the corrected code and wrap the code between "//BEGIN" and
"//END".
```

Fig. 30 Vanilla Prompt

Table 22 Overall results of code correctness after the repair and code cleaning phase for each model (Vanilla)

Model	OK	Exec	Assert	Comp	%
gemma-7b-it	4	15	103	3	2.6%
llama3-8b-8192	32	10	110	1	20.6%
llama3-70b-8192	71	9	73	2	45.8%
mixtral-8x7b-32768	23	12	98	3	14.8%
Overall	130	46	384	9	

```
 SysPrompt >>> You are a chatbot whose purpose is to correct an incorrect
function, written in the C programming language, to solve a task. The
function, the task to solve, and the expected signature of the function are
given to you as inputs as well as the counterexamples for which the code is
incorrect.
 Provide the corrected code and wrap the code between "//BEGIN" and "//END".

 Your answer must be in the following format:
 //BEGIN
 <code>
 //END

 <reasoning>

 Let's work this out in a step by step way to be sure we have the right
answer.
```

Fig. 31 Chain of Thought Prompt

Table 23 Overall results of code correctness after the repair and code cleaning phase for each model (CoT)

Model	OK	Exec	Assert	Comp	%
gemma-7b-it	7	14	119	7	4.5%
llama3-8b-8192	34	15	103	2	21.9%
llama3-70b-8192	60	9	82	4	38.7%
mixtral-8x7b-32768	22	14	106	3	14.2%
Overall	123	52	410	16	

```
 SysPrompt >>> You are a chatbot whose purpose is to correct an incorrect
function, written in the C programming language, to solve a task.
 The function, the task to solve, and the expected signature of the function
are given to you as inputs as well as the counterexamples for which the code
is incorrect.
 Provide the corrected code and wrap the code between "//BEGIN" and "//END".
```

Fig. 32 *One Assert at the Time* Prompt. The system prompt is the vanilla prompt but we provide one failed assertion at the time, iteratively, for a maximum of 6 iterations

Table 24 Overall results of code correctness after the repair and code cleaning phase for each model (One Assert at the Time)

Model	OK	Exec	Assert	Comp	%
gemma-7b-it	20	19	91	4	12.9%
llama3-8b-8192	63	10	76	1	40.6%
llama3-70b-8192	92	8	52	2	59.4%
mixtral-8x7b-32768	51	14	71	2	32.9%
Overall	226	51	290	9	

Table 25 Results of code correctness after the repair and code cleaning phase for `gemma-7b-it` with the *One Assert at the Time* prompt

Model	OK	Exec	Assert	Comp
`gemma-7b-it`	5	5	31	2
`llama3-8b-8192`	9	6	15	0
`llama3-70b-8192`	3	2	22	0
`mixtral-8x7b-32768`	3	6	23	2
Overall	20	19	91	4

Table 26 Results of code correctness after the repair and code cleaning phase for `llama3-8b-8192` with the *One Assert at the Time* prompt

Model	OK	Exec	Assert	Comp
`gemma-7b-it`	26	3	17	0
`llama3-8b-8192`	11	3	23	0
`llama3-70b-8192`	13	2	13	1
`mixtral-8x7b-32768`	13	2	23	0
Overall	63	10	76	1

Table 27 Results of code correctness after the repair and code cleaning phase for `llama3-70b-8192` with the *One Assert at the Time* prompt

Model	OK	Exec	Assert	Comp
`gemma-7b-it`	33	2	13	0
`llama3-8b-8192`	20	2	14	1
`llama3-70b-8192`	16	2	13	0
`mixtral-8x7b-32768`	23	2	12	1
Overall	92	8	52	2

Table 28 Results of code correctness after the repair and code cleaning phase for `mixtral-8x7b-32768` with the *One Assert at the Time* prompt

Model	OK	Exec	Assert	Comp
`gemma-7b-it`	19	4	21	0
`llama3-8b-8192`	12	4	17	0
`llama3-70b-8192`	10	2	14	1
`mixtral-8x7b-32768`	10	4	19	1
Overall	51	14	71	2

```
 SysPrompt >>> You are a chatbot whose purpose is to fix the vulnerabilities
found the code implementation in the C programming language for a task that
is given to you as input.
 Provide the corrected code and wrap the code between "//BEGIN" and "//END".
 The list description of the vulnerabilities is the following:

 <list of vulnerabilities>
```

Fig. 33 Vanilla Prompt

Self-evaluation experiments - safety

See table 19

See Fig. 26

See table 20

See Fig. 27

See table 21

Table 29 Aggregate vulnerability analysis for each model after the repair phase (Vanilla)

Model	Removed	Remaining	Added
gemma-7b-it	20	16	1
llama3-70b-8192	32	12	0
llama3-8b-8192	31	12	2
mixtral-8x7b-32768	32	7	1
Cumulative	115	47	4

```
 SysPrompt >>> You are a chatbot whose purpose is to fix the vulnerabilities
found the code implementation in the C programming language for a task that
is given to you as input.
 Provide the corrected code and wrap the code between "//BEGIN" and "//END".
 The list description of the vulnerabilities is the following:

 <list of vulnerabilities>

 Let's work this out in a step by step way to be sure we have the right
answer.
```

Fig. 34 Chain of Thought Prompt

Table 30 Aggregate vulnerability analysis for each model after the repair phase (Chain of Thought)

Model	Removed	Remaining	Added
gemma-7b-it	20	22	0
llama3-70b-8192	33	11	0
llama3-8b-8192	30	13	0
mixtral-8x7b-32768	32	6	0
Cumulative	115	52	0

```
 SysPrompt >>> You are a chatbot whose purpose is to fix the vulnerabilities
found the code implementation in the C programming language for a task that
is given to you as input.
 Provide the corrected code and wrap the code between "//BEGIN" and "//END".
 The list description of the vulnerabilities is the following:

 <list of vulnerabilities>

 These are instruction on how to fix the vulnerabilities:
 NULLPTR_DEREFERENCE: Before dereferencing a pointer, always check if it is
NULL. Ensure all pointers are initialized before use.
 UNINITIALIZED_VALUE: Always initialize variables before their first use.
 BUFFER_OVERRUN: Always check that array indices are within bounds before
accessing elements.
 MEMORY_LEAK: Always free memory allocated with malloc using free.
```

Fig. 35 Instructions Prompt

Table 31 Aggregate vulnerability analysis for each model after the repair phase (Instructions)

Model	Removed	Remaining	Added
gemma-7b-it	23	14	0
llama3-70b-8192	37	7	0
llama3-8b-8192	27	17	0
mixtral-8x7b-32768	27	10	1
Cumulative	114	48	1

```
 SysPrompt >>> You are a chatbot whose purpose is to fix the vulnerabilities
found the code implementation in the C programming language for a task that
is given to you as input.
 Provide the corrected code and wrap the code between "//BEGIN" and "//END".
 The list description of the vulnerabilities is the following:

 <list of vulnerabilities>

 These are instruction on how to fix the vulnerabilities:
 NULLPTR_DEREFERENCE: Before dereferencing a pointer, always check if it is
NULL. Ensure all pointers are initialized before use.
 UNINITIALIZED_VALUE: Always initialize variables before their first use.
 BUFFER_OVERRUN: Always check that array indices are within bounds before
accessing elements.
 MEMORY_LEAK: Always free memory allocated with malloc using free.
 Let's work this out in a step by step way to be sure we have the right
answer.
```

Fig. 36 Combo Prompt

Table 32 Aggregate vulnerability analysis for each model after the repair phase (Combo)

Model	Removed	Remaining	Added
gemma-7b-it	22	18	0
llama3-70b-8192	30	13	2
llama3-8b-8192	26	18	0
mixtral-8x7b-32768	27	9	1
Cumulative	105	58	3

```
 SysPrompt >>> You are a chatbot whose purpose is to fix the memory-related
vulnerabilities found in the code implementation in the C programming
language for a task that is given to you as input.
 You must provide a safe code, ensuring that no vulnerabilities occur at
runtime.
 Provide the corrected code and wrap the code between "//BEGIN" and "//END".
```

Fig. 37 No Info Prompt. In the content prompt, in this experiment, we do not provide any information for the kind of vulnerabilities present in the code

Table 33 Aggregate vulnerability analysis for each model after the repair phase (No Info)

Model	Removed	Remaining	Added
gemma-7b-it	9	34	1
llama3-70b-8192	30	14	4
llama3-8b-8192	29	15	4
mixtral-8x7b-32768	29	13	5
Cumulative	97	76	14

```
 SysPrompt >>> You are a chatbot whose purpose is to fix the vulnerabilities
found the code implementation in the C programming language for a task that
is given to you as input.
 Provide the corrected code and wrap the code between "//BEGIN" and "//END".
 The list description of the vulnerabilities is the following:

 <list of vulnerabilities>
```

Fig. 38 No Line Prompt. In the content prompt, in this experiment, we do not provide the line number where the vulnerability is present

Table 34 Aggregate vulnerability analysis for each model after the repair phase (No Line)

Model	Removed	Remaining	Added
gemma-7b-it	19	22	1
llama3-70b-8192	32	12	0
llama3-8b-8192	35	9	1
mixtral-8x7b-32768	28	10	0
Cumulative	114	53	2

```
 SysPrompt >>> You are a chatbot whose purpose is to fix the vulnerabilities
found the code implementation in the C programming language for a task that
is given to you as input.
 Provide the corrected code and wrap the code between "//BEGIN" and "//END".
 The list description of the vulnerabilities is the following:

 <list of vulnerabilities>

 These are instruction on how to fix the vulnerabilities:
 NULLPTR_DEREFERENCE: Before dereferencing a pointer, always check if it is
NULL. Ensure all pointers are initialized before use.
 UNINITIALIZED_VALUE: Always initialize variables before their first use.
 BUFFER_OVERRUN: Always check that array indices are within bounds before
accessing elements.
 MEMORY_LEAK: Always free memory allocated with malloc using free.
```

Fig. 39 *One Vulnerability at the Time Prompt*

Table 35 Aggregate vulnerability analysis for each model after the repair phase (One Vulnerability at the Time)

Model	Removed	Remaining	Added
gemma-7b-it	24	14	1
llama3-8b-8192	29	15	2
llama3-70b-8192	39	5	0
mixtral-8x7b-32768	39	5	0
Cumulative	131	39	3

Table 36 Breakdown of vulnerabilities for model: `gemma-7b-it`. Prompt: *One Vulnerability at the Time*

Model	Removed	Remaining	Added
gemma-7b-it	4	3	0
llama3-70b-8192	9	1	1
llama3-8b-8192	5	2	0
mixtral-8x7b-32768	6	8	0
Cumulative	24	14	1

Table 37 Breakdown of vulnerabilities for model: `llama3-70b-8192`. Prompt: *One Vulnerability at the Time*

Model	Removed	Remaining	Added
gemma-7b-it	7	0	0
llama3-70b-8192	10	0	0
llama3-8b-8192	8	0	0
mixtral-8x7b-32768	14	5	0
Cumulative	39	5	0

Table 38 Breakdown of vulnerabilities for model: `llama3-8b-8192`. Prompt: *One Vulnerability at the Time*

Model	Removed	Remaining	Added
`gemma-7b-it`	7	0	0
`llama3-70b-8192`	10	0	0
`llama3-8b-8192`	4	4	0
`mixtral-8x7b-32768`	8	11	2
Cumulative	29	15	2

Table 39 Breakdown of vulnerabilities for model: `mixtral-8x7b-32768`. Prompt: *One Vulnerability at the Time*

Model	Removed	Remaining	Added
`gemma-7b-it`	7	0	0
`llama3-70b-8192`	9	1	0
`llama3-8b-8192`	8	0	0
`mixtral-8x7b-32768`	15	4	0
Cumulative	39	5	0

See Figs. 28, 29

Code repair experiments - correctness

See Fig. 30

See table 22

See Fig. 31

see Table 23

See Fig. 32

See tables 24, 25, 26, 27, 28

Code repair experiments - safety

See Fig. 33

See table 29

See Fig. 34

See table 30

See Fig. 35

See table 31

See Fig. 36

See table 32

See Fig. 37

See table 33

See Fig. 38

See table 34

See Fig. 39

See tables 35, 36, 37, 38, 39

Acknowledgements

This work was supported by Bando di Ateneo 2024 per la Ricerca, funded by University of Parma (FIL_2024_PROGETTI_B_IOTTI - CUP D93C24001250005) and by SERICS (PE00000014 - CUP H73C2200089001) project funded by PNRR NextGeneration EU.

Author contributions
G.D.: Software, Data Curation, Writing, Visualization, Experiments V.A.: Software, Supervision, Writing, Experiments E.I.: Software, Writing, Visualization, Experiments S.M.: Conceptualization, Supervision, Writing A.C.: Resources, Supervision, Writing E.Z.: Validation, Supervision, Writing All authors reviewed the manuscript.

Data availability
We provide https://doi.org/10.6084/m9.figshare.26984716, containing (i) the benchmark suite of tasks (Sect. 2), including both the raw and cleaned source code generated by the models we considered, (ii) the source code for the scripts used in our experimental evaluation (Sect. 6), (iii) the analysis results computed by Infer, covering both the vulnerability analysis and the repair phases. We also provide a README file with the instructions on how to reproduce our experiments for each phase of the pipeline. To reproduce our experimental setting, the user will need to install Infer and get a token for the GROQ API.

Declarations

Conflict of interest
The authors declare no Conflict of interest.

Ethical approval
Not applicable.

Consent to participate
Not applicable.

Consent to publish
Not applicable.

Received: 23 September 2025 / Accepted: 9 February 2026
Published online: 05 March 2026

References

1. Zhao S, Jia M, Tuan LA, Wen J. Universal vulnerabilities in large language models: in-context learning backdoor attacks. CoRR 2024 https://doi.org/10.48550/ARXIV.2401.05949, arXiv:abs/2401.05949.
2. Zhang B, Liang P, Zhou X, Ahmad A, Waseem M. Practices and challenges of using github copilot: An empirical study. In: Chang, S. (ed.) The 35th International Conference on Software Engineering and Knowledge Engineering, SEKE 2023, KSIR Virtual Conference Center, USA, July 1–10, 2023, pp. 124–129. KSI Research Inc., 2023. https://doi.org/10.18293/SEKE2023-077.
3. Barke S, James MB, Polikarpova N. Grounded copilot: how programmers interact with code-generating models. Proc ACM Program Lang. 2023;7(OOPSLA1):85–111. https://doi.org/10.1145/3586030.
4. Gartner: gartner hype cycle shows ai practices and platform engineering will reach mainstream adoption in software engineering in two to five years 2024. https://www.gartner.com/en/newsroom/press-releases/2023-11-28-gartner-hype-cycle-shows-ai-practices-and-platform-engineering-will-reach-mainstream-adoption-in-software-engineering-in-two-to-five-years.
5. Team ML. https://ai.meta.com/blog/meta-llama-3/ 2024. https://ai.meta.com/blog/meta-llama-3/.
6. Mesnard T, Hardin C, Dadashi R, Bhupatiraju S, Pathak S, Sifre L, Rivière M, Kale MS, Love J, Tafti P, Hussenot L, Chowdhery A, Roberts A, Barua A, Botev A, Castro-Ros A, Slone A, Héliou A, Tacchetti A, Bulanova A, Paterson A, Tsai B, Shahriari B, Lan CL, Choquette-Choo CA, Crepy C, Cer D, Ippolito D, Reid D, Buchatskaya E, Ni E, Noland E, Yan G, Tucker G, Muraru G, Rozhdestvenskiy G, Michalewski H, Tenney I, Grishchenko I, Austin J, Keeling J, Labanowski J, Lespiau J, Stanway J, Brennan J, Chen J, Ferret J, Chiu J, et al. Gemma: Open models based on gemini research and technology. 2024 https://doi.org/10.48550/ARXIV.2403.08295, CoRR arXiv:abs/2403.08295.
7. Jiang AQ, Sablayrolles A, Roux A, Mensch A, Savary B, Bamford C, Chaplot DS, Las Casas D, Hanna EB, Bressand F, Lengyel G, Bour G, Lample G, Lavaud LR, Saulnier L, Lachaux M, Stock P, Subramanian S, Yang S, Antoniak S, Scao TL, Gervet T, Lavril T, Wang T, Lacroix T, Sayed WE. Mixtral of experts. https://doi.org/10.48550/ARXIV.2401.04088, 2024 CoRR arXiv:abs/2401.04088.
8. Calcagno C, Distefano D. Infer: An automatic program verifier for memory safety of C programs. In: Bobaru MG, Havelund K, Holzmann GJ, Joshi R (eds.) NASA Formal Methods - Third International Symposium, NFM 2011, Pasadena, CA, USA, April 18–20, 2011. Proceedings. Lecture Notes in Computer Science, vol. 6617, pp. 459–465. Springer, 2011 https://doi.org/10.1007/978-3-642-20398-5_33.
9. Austin J, Odena A, Nye MI, Bosma M, Michalewski H, Dohan D, Jiang E, Cai CJ, Terry M, Le QV, Sutton C. Program synthesis with large language models. 2021 CoRR arXiv:abs/2108.07732.
10. Xu R, Cao J, Lu Y, Wen M, Lin H, Han X, He B, Cheung S-C, Sun L. CRUXEval-X: a benchmark for multilingual code reasoning, understanding and execution 2025. https://arxiv.org/abs/2408.13001.
11. Chen M, Tworek J, Jun H, Yuan Q, Oliveira Pinto HP, Kaplan J, Edwards H, Burda Y, Joseph N, Brockman G, Ray A, Puri R, Krueger G, Petrov M, Khlaaf H, Sastry G, Mishkin P, Chan B, Gray S, Ryder N, Pavlov M, Power A, Kaiser L, Bavarian M, Winter C, Tillet P, Such FP, Cummings D, Plappert M, Chantzis F, Barnes E, Herbert-Voss A, Guss WH, Nichol A, Paino A, Tezak N, Tang J, Babuschkin I, Balaji S, Jain S, Saunders W, Hesse C, Carr AN, Leike J, Achiam J, Misra V, Morikawa E, Radford A, Knight M, Brundage M, Murati M, Mayer K, Welinder P, McGrew B, Amodei D, McCandlish S, Sutskever I, Zaremba W. Evaluating large language models trained on code 2021 arXiv:2107.03374 [cs.LG].
12. Jiang AQ, Sablayrolles A, Mensch A, Bamford C, Chaplot DS, Las Casas D, Bressand F, Lengyel G, Lample G, Saulnier L, Lavaud LR, Lachaux M, Stock P, Scao TL, Lavril T, Wang T, Lacroix T, Sayed WE. Mistral 7b. 2023 https://doi.org/10.48550/ARXIV.2310.06825, CoRR arXiv:abs/2310.06825.

13. Schulhoff S, Ilie M, Balepur N, Kahadze K, Liu A, Si C, Li Y, Gupta A, Han H, Schulhoff S, Dulepet PS, Vidyadhara S, Ki D, Agrawal S, Pham C, Kroiz GC, Li F, Tao H, Srivastava A, Costa HD, Gupta S, Rogers ML, Goncearenco I, Sarli G, Galynker I, Peskoff D, Carpuat M, White J, Anadkat S, Hoyle AM, Resnik P. The prompt report: A systematic survey of prompting techniques. https://doi.org/10.48550/ARXIV.2406.06608, CoRR arXiv:abs/2406.06608 (2024).
14. Wei J, Wang X, Schuurmans D, Bosma M, Ichter B, Xia F, Chi EH, Le QV, Zhou D. Chain-of-thought prompting elicits reasoning in large language models. In: Koyejo S, Mohamed S, Agarwal A, Belgrave D, Cho K, Oh A. (eds.) Advances in Neural Information Processing Systems 35: Annual Conference on Neural Information Processing Systems 2022, NeurIPS 2022, New Orleans, LA, USA, November 28 - December 9, 2022 2022. http://papers.nips.cc/paper_files/paper/2022/hash/9d5609613524ecf4f15af0f7b31abca4-Abstract-Conference.html.
15. Zhou Y, Muresanu AI, Han Z, Paster K, Pitis S, Chan H, Ba J. Large language models are human-level prompt engineers. In: The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1–5, 2023. OpenReview.net, 2023. https://openreview.net/forum?id=92gvk82DE-.
16. Cousot P, Cousot R. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Graham RM, Harrison MA, Sethi R. (eds.) Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, Los Angeles, California, USA, January 1977, pp. 238–252. ACM, 1977. https://doi.org/10.1145/512950.512973.
17. Tihanyi N, Bisztray T, Jain R, Ferrag MA, Cordeiro LC, Mavroeidis V. The formai dataset: Generative AI in software security through the lens of formal verification. In: McIntosh S, Choi E, Herbold S. (eds.) Proceedings of the 19th International Conference on Predictive Models and Data Analytics in Software Engineering, PROMISE 2023, San Francisco, CA, USA, 8 December 2023, pp. 33–43. ACM, 2023. https://doi.org/10.1145/3617555.3617874.
18. Gadelha MYR, Monteiro FR, Morse J, Cordeiro LC, Fischer B, Nicole DA. ESBMC 5.0: an industrial-strength C model checker. In: Huchard M, Kästner C, Fraser G. (eds.) Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3–7, 2018, pp. 888–891. ACM, 2018. https://doi.org/10.1145/3238147.3240481
19. Tihanyi N, Bisztray T, Ferrag MA, Jain R, Cordeiro LC. How secure is ai-generated code: a large-scale comparison of large language models. Empir Softw Eng. 2025;30(2):1–42.
20. Pearce H, Ahmad B, Tan B, Dolan-Gavitt B, Karri R. Asleep at the keyboard? assessing the security of github copilot's code contributions. In: IEEE Symposium on Security and Privacy, S&P 2022, pp. 754–768 2022. IEEE.
21. Nazzal M, Khalil I, Khreishah A, Phan N. Promsec: Prompt optimization for secure generation of functional source code with large language models (llms). In: Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security, pp. 2266–2280, 2024.
22. He J, Vechev M. Large language models for code: Security hardening and adversarial testing. In: Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security, pp. 1865–1879, 2023.
23. Li D, Yan M, Zhang Y, Liu Z, Liu C, Zhang X, Chen T, Lo D. Cosec: On-the-fly security hardening of code llms via supervised co-decoding. In: Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis, pp. 1428–1439, 2024.
24. Chapman PJ, Rubio-González C, Thakur AV. Interleaving static analysis and LLM prompting. In: Proceedings of the 13th ACM SIGPLAN International Workshop on the State Of the Art in Program Analysis, SOAP 2024, pp. 9–17, 2024.
25. Li Z, Dutta S, Naik M. LLM-assisted static analysis for detecting security vulnerabilities. arXiv preprint, 2024 arXiv:2405.17238.
26. Ullah S, Han M, Pujar S, Pearce H, Coskun A, Stringhini G. LLMs cannot reliably identify and reason about security vulnerabilities (yet?): A comprehensive evaluation, framework, and benchmarks. In: IEEE Symposium on Security and Privacy, S&P 2024, 2024.
27. Lu G, Ju X, Chen X, Pei W, Cai Z. Grace: empowering llm-based software vulnerability detection with graph structure and in-context learning. J Syst Softw. 2024;212:112031.
28. Wen X-C, Gao C, Gao S, Xiao Y, Lyu MR. Scale: Constructing structured natural language comment trees for software vulnerability detection. In: Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis, pp. 235–247, 2024.
29. Charalambous Y, Tihanyi N, Jain R, Sun Y, Ferrag MA, Cordeiro LC. A new era in software security: Towards self-healing software via large language models and formal verification, 2023. https://doi.org/10.48550/ARXIV.2305.14752, CoRR arXiv:abs/2305.14752.
30. Jin M, Shahriar S, Tufano M, Shi X, Lu S, Sundaresan N, Svyatkovskiy A. Inferfix: End-to-end program repair with llms. In: Chandra S, Blincoe K, Tonella P. (eds.) Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2023, San Francisco, CA, USA, December 3–9, 2023, pp. 1646–1656. ACM, 2023. https://doi.org/10.1145/3611643.3613892.
31. Janßen C, Richter C, Wehrheim H. Can chatgpt support software verification?, 2023. https://doi.org/10.48550/ARXIV.2311.02433, CoRR arXiv:abs/2311.02433.
32. Li H, Hao Y, Zhai Y, Qian Z. Assisting static analysis with large language models: A chatgpt experiment. In: Chandra S, Blincoe K, Tonella P. (eds.) Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2023, San Francisco, CA, USA, December 3–9, 2023, pp. 2107–2111. ACM, 2023. https://doi.org/10.1145/3611643.3613078.
33. Jain N, Han K, Gu A, Li W-D, Yan F, Zhang T, Wang S, Solar-Lezama A, Sen K, Stoica I: Livecodebench: Holistic and contamination free evaluation of large language models for code, 2024. arXiv preprint arXiv:2403.07974.
34. Olausson TX, Inala JP, Wang C, Gao J, Solar-Lezama A. Is self-repair a silver bullet for code generation? In: The Twelfth International Conference on Learning Representations, 2023.
35. Islam NT, Khoury J, Seong A, Bou-Harb E, Najafirad P. Enhancing source code security with llms: demystifying the challenges and generating reliable repairs. In: Network and Distributed System Security (NDSS) Symposium 2024, 2024.
36. OpenAI: GPT-3.5-turbo, 2022. Accessed March 2024 . https://platform.openai.com/docs/models/gpt-3-5-turbo.
37. Bhatt M, Chennabasappa S, Nikolaidis C. Wan S, Evtimov I, Gabi D, Song D, Ahmad F, Aschermann C, Fontana L, Frolov S, Giri RP, Kapil D, Kozyrakis Y, LeBlanc D, Milazzo J, Straumann A, Synnaeve G, Vontimitta V, Whitman S, Saxe J. Purple llama cyberseceval: A secure coding benchmark for language models. 2023. https://doi.org/10.48550/ARXIV.2312.04724, CoRR arXiv:abs/2312.04724.

38. He J, Vero M, Krasnopolska G, Vechev MT. Instruction tuning for secure code generation. In: Forty-first International Conference on Machine Learning, ICML 2024, Vienna, Austria, July 21–27, 2024. OpenReview.net, 2024. https://openreview.net/forum?id=MgTzMaYHvG.
39. Pearce H, Tan B, Ahmad B, Karri R, Dolan-Gavitt B. Examining zero-shot vulnerability repair with large language models. In: IEEE Symposium on Security and Privacy, S&P 2023, pp. 2339–2356, 2023. IEEE.
40. Plein L, Ouédraogo WC, Klein J, Bissyandé TF. Automatic generation of test cases based on bug reports: a feasibility study with large language models. In: Proceedings of the 2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings, ICSE Companion 2024, Lisbon, Portugal, April 14–20, 2024, pp. 360–361. ACM, 2024. https://doi.org/10.1145/3639478.3643119..

Publisher's Note

Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.