Safe4U: Identifying Unsound Safe Encapsulations of Unsafe Calls in Rust using LLMs

HUAN LI, Zhejiang University, China
BEI WANG[*], Zhejiang University, China
XING HU[*], Zhejiang University, China
XIN XIA, Zhejiang University, China

Rust is an emerging programming language that ensures safety through strict compile-time checks. A Rust function marked as `unsafe` indicates it has additional safety requirements (e.g., *initialized*, *not null*), known as *contracts* in the community. These unsafe functions can only be called within explicit unsafe blocks and the contracts must be guaranteed by the caller. To reuse and reduce unsafe code, the community recommends using *safe encapsulation of unsafe calls* (EUC) in practice. However, an EUC is *unsound* if any contract is not guaranteed and could lead to undefined behaviors in safe Rust, thus breaking Rust's safety promise. It is challenging to identify unsound EUCs with conventional techniques due to the limitation in cross-lingual comprehension of code and natural language. Large language models (LLMs) have demonstrated impressive capabilities, but their performance is unsatisfactory owing to the complexity of contracts and the lack of domain knowledge. To this end, we propose a novel framework, Safe4U, which incorporates LLMs, static analysis tools, and domain knowledge to identify unsound EUCs. Safe4U first utilizes static analysis tools to retrieve relevant context. Then, it decomposes the primitive contract description into several fine-grained classified contracts. Ultimately, Safe4U introduces domain knowledge and invokes the reasoning capability of LLMs to verify every fine-grained contract. The evaluation results show that Safe4U brings a general performance improvement and the fine-grained results are constructive for locating specific unsound sources. In real-world scenarios, Safe4U can identify 9 out of 11 unsound EUCs from CVE. Furthermore, Safe4U detected 22 new unsound EUCs in the most downloaded crates, 16 of which have been confirmed.

CCS Concepts: • **Software and its engineering** → **Software defect analysis**.

Additional Key Words and Phrases: Rust, Unsafe Call, Unsound Encapsulation, Large Language Model

ACM Reference Format:
Huan Li, Bei Wang, Xing Hu, and Xin Xia. 2025. Safe4U: Identifying Unsound Safe Encapsulations of Unsafe Calls in Rust using LLMs. *Proc. ACM Softw. Eng.* 2, ISSTA, Article ISSTA021 (July 2025), 24 pages. https://doi.org/10.1145/3728890

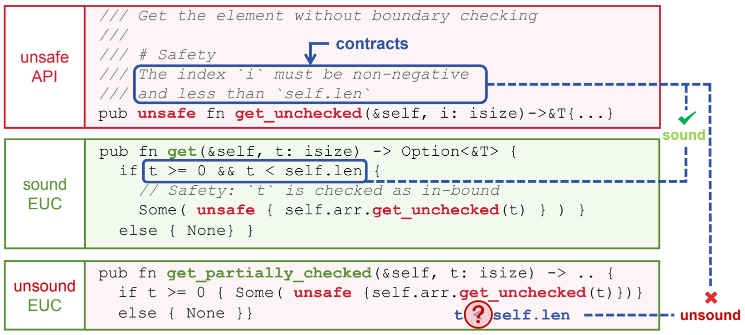
1 Introduction

Rust is a promising programming language acclaimed for its efficiency and security [76]. Through strict compile-time checks, it provides a safety promise that programs written in safe Rust are guaranteed to be memory-safe [73, 77]. To satisfy flexibility and extreme performance, Rust provides the keyword `unsafe` to bypass safety checks and allow unsafe low-level operations within explicit `unsafe` blocks [27]. A function (API) can be explicitly marked as `unsafe`, meaning it has additional

[*]Corresponding Authors

Authors' Contact Information: Huan Li, Zhejiang University, Hangzhou, China, huanlee@zju.edu.cn; Bei Wang, Zhejiang University, Hangzhou, China, bwang9410@gmail.com; Xing Hu, Zhejiang University, Hangzhou, China, xinghu@zju.edu.cn; Xin Xia, Zhejiang University, Hangzhou, China, xin.xia@acm.org.

ACM 2994-970X/2025/7-ARTISSTA021
https://doi.org/10.1145/3728890



```
/// Get the element without boundary checking
///
/// # Safety
/// The index `i` must be non-negative
/// and less than `self.len`
pub unsafe fn get_unchecked(&self, i: isize)->&T{...}
```

```
pub fn get(&self, t: isize) -> Option<&T> {
  if t >= 0 && t < self.len {
    // Safety: `t` is checked as in-bound
    Some( unsafe { self.arr.get_unchecked(t) } ) }
  else { None} }
```

```
pub fn get_partially_checked(&self, t: isize) -> .. {
  if t >= 0 { Some( unsafe {self.arr.get_unchecked(t)})}
  else { None }}
```

Fig. 1. An unsafe API *get_unchecked* and two simple *safe encapsulations of unsafe calls* (EUCs) *get* and *get_partially_checked* that are sound and unsound, respectively.

safety requirements for its caller, referred to as *contracts* in the community [54, 63]. Typically, these contracts are written in unstructured natural language in the *Safety* section of the documents [49].

Calling unsafe APIs introduces extra unsafe blocks and remarkably increases the burden on developers, thereby weakening the benefit brought by Rust's safety promise. Therefore, the Rust community recommends encapsulating unsafe calls behind safe functions to hide and reuse the unsafe implementation [7, 54, 63]. According to the specification, the *safe encapsulation of unsafe calls* (EUC) must guarantee all contracts when calling unsafe APIs. As shown in Figure 1, the EUC `get` guarantees the contracts of unsafe API `get_unchecked`, "*The index i must be non-negative and less than* `self.len`", with corresponding boundary checks. Since unsafe blocks bypass compiler checks, it is the developers' responsibility to ensure that all contracts are guaranteed [54]. Occasionally, developers fail to guarantee all contracts of interior unsafe calls for various reasons (e.g., lack of expertise or sudden negligence), which leads to unsoundness. An encapsulation is *unsound* means it can be called in `safe` code but causes undefined behavior, which breaks the safety promise of Rust [66]. Taking the unsound EUC `get_partially_checked` in Figure 1 as an example, this EUC calls `get_unchecked` without guaranteeing "$t < self.len$". Therefore, developers can use `get_partially_checked` in safe code, but access to uninitialized elements or illegal memory with specific input. Furthermore, the unsoundness of EUC could propagate widely through function calls and data flows, leading to sophisticated bugs or even hidden vulnerabilities [23]. In general, it is crucial to eliminate unsoundness to maintain the safety promise of Rust.

To identify the soundness of an EUC, we need to check whether it guarantees all contracts described in the *Safety* section, which is written in unstructured and flexible natural language. Accordingly, conventional techniques for unsoundness detection, including formal verification [30, 43] and static analysis[5, 40, 53], can hardly be adopted due to their limitation in cross-lingual comprehension of natural language and code. Deep learning models demonstrate the capability to understand both code and natural language [6, 12, 75], which requires task-specific data for fine-tuning. Nevertheless, there is no existing dataset for identifying unsound EUCs, and it is challenging to create a new dataset with sufficient samples. The emergence of Large Language Models (LLMs) and their excellent achievement in multi-lingual tasks indicate their potential to identify unsound EUCs [11, 31, 34, 41, 59, 69]. However, as discussed in Section 5.1, it turns out that the LLMs underperform in this task for various reasons. Firstly, the *Safety* sections of unsafe APIs are commonly redundant and intricate, illustrating the intrinsic challenge of checking EUC with the original *Safety* sections. In addition, since LLMs lack the relevant context of the EUC, they often generate unfaithful responses with notable hallucinations [29]. Furthermore, due to the deficiency of domain knowledge, LLMs tend to check contracts based on their limited or even wrong understanding (e.g., requiring all contracts to be explicitly validated), leading to low accuracy [70].

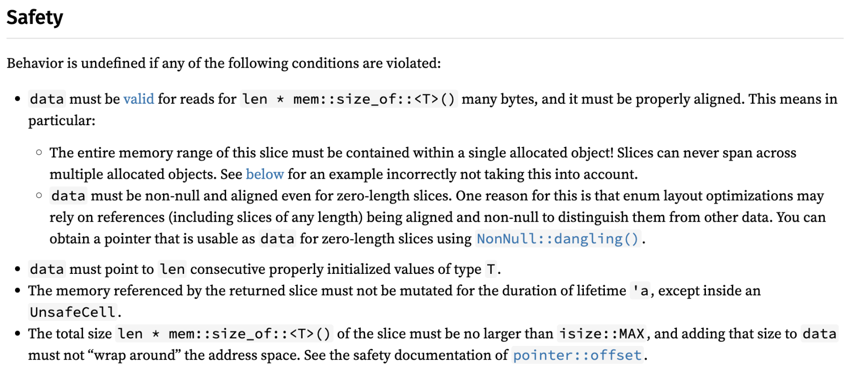
To obtain insight into how contracts are guaranteed in real-world EUCs, we first conducted a preliminary study on the Rust standard library [65] and the top 500 most downloaded crates (i.e., packages in Rust) [16]. After manually studying their documents and code, we ultimately summarized 16 contract types, covering various requirements: *Memory & Pointer*, *Value*, *Concurrency*, *Lifetime*, *Ownership*, *Dataflow*, and *Environment*. We also distilled 34 guarantee patterns (GPs) for these contract types, meaning a contract can be guaranteed in any corresponding GP associated with its type. For instance, the contract "`i` *must be less than* `self.len`" can be guaranteed either by explicit validation or by the context that `i` is the remainder of `self.len`.

Based on the preliminary study, we propose a novel framework, Safe4U, to identify **u**nsound **safe** encaps**u**lations of **u**nsafe calls in R**u**st. Given an EUC, Safe4U identifies its soundness with three modules, Context Retrieval, Decomposition & Classification, and Pattern-Oriented Checks. Firstly, the Context Retrieval module deploys static analysis tools that collaborate with the compiler to retrieve the context of the target EUC. The context involves additional information crucial for identification (including the *Safety* section). Secondly, the Decomposition & Classification module applies the LLM to decompose the verbose *Safety* section into several fine-grained contracts. During decomposition, the LLM also classifies every decomposed contract into predefined contract types. Through decomposition, the fine-grained contracts can be checked independently, significantly decreasing the difficulty of unsoundness identification. Through classification, Safe4U can dynamically select examples containing specific domain knowledge to promote subsequent checks. In particular, the Pattern-Oriented Checks module adds examples to the prompt and checks the contract utilizing LLMs' in-context learning and reasoning capabilities. Ultimately, the results for all fine-grained contracts are aggregated. If one contract is identified as *Unguaranteed*, this contract is deemed as the fine-grained unsoundness that requires a fix, and this EUC is considered *Unsound*.

To evaluate the performance of Safe4U, we extract samples (EUCs) from the most downloaded crates. Since there is no dataset for unsound EUCs, we propose to simulate unsound samples from unsafe encapsulations. The evaluation results show that Safe4U outperforms the baseline for all LLMs, reflecting its effectiveness and generalizability. Additionally, we find that Safe4U can locate the fine-grained unguaranteed contract which requires a fix. We also evaluate Safe4U in real-world scenarios. Firstly, it reveals its effectiveness by identifying 9 of 11 unsound EUCs from CVE [19, 20]. Then, we deployed Safe4U for a practical scan on the most downloaded crates, involving 2,849 EUCs and 1,409 different unsafe APIs. It ultimately detected 22 unsound EUCs, 16 of which have been confirmed and fixed. The replication package is publicly available [3].

The main contributions of our paper are as follows:

- We conduct a preliminary study of both the documents and the usage of unsafe APIs in the standard library and the most commonly used crates. We summarize 16 contract types and 34 corresponding guarantee patterns.
- To the best of our knowledge, we propose the automated framework, Safe4U, that is the first to introduce LLMs to identify unsound EUCs in Rust.
- Safe4U not only decomposes the identification into sub-tasks but also effectively integrates static analysis and domain knowledge into the advanced capabilities of LLM. The evaluation results indicate that Safe4U brings generalizable performance improvements for all LLMs.
- Safe4U can identify 9 out of 11 existing unsound EUCs from CVE and found 22 new unsound EUCs from real-world Rust crates.


Safety

Behavior is undefined if any of the following conditions are violated:

- `data` must be valid for reads for `len * mem::size_of::<T>()` many bytes, and it must be properly aligned. This means in particular:
  - The entire memory range of this slice must be contained within a single allocated object! Slices can never span across multiple allocated objects. See below for an example incorrectly not taking this into account.
  - `data` must be non-null and aligned even for zero-length slices. One reason for this is that enum layout optimizations may rely on references (including slices of any length) being aligned and non-null to distinguish them from other data. You can obtain a pointer that is usable as `data` for zero-length slices using `NonNull::dangling()`.
- `data` must point to `len` consecutive properly initialized values of type `T`.
- The memory referenced by the returned slice must not be mutated for the duration of lifetime `'a`, except inside an `UnsafeCell`.
- The total size `len * mem::size_of::<T>()` of the slice must be no larger than `isize::MAX`, and adding that size to `data` must not "wrap around" the address space. See the safety documentation of `pointer::offset`.

Fig. 2. The *Safety* section of commonly used unsafe API *from_raw_parts* in the standard library.

2 Background and Preliminary

This section presents the background of our research and a preliminary study conducted on (1) what types the contracts can be categorized into and (2) how each type can be guaranteed.

2.1 Background

Safe and Unsafe Rust. Rust conducts strict checks during the compilation phase to deliver the safety promise that programs written in safe Rust are guaranteed to be memory-safe as long as they can pass the compile checks [73, 77]. In practice, developers usually need more flexibility to write low-level code for optimization or system calls, which is not allowed by safe Rust [27]. Accordingly, explicit `unsafe` blocks can be used to bypass compiler checks and allow unsafe operations [18]. Previous work found that calling unsafe functions (APIs) is the primary purpose of `unsafe` use [23]. Typically, a Rust function is declared as `unsafe` to emphasize its additional safety requirements described in the *Safety* section, referred to as *contracts* [54, 63, 76]. It is the caller's responsibility to guarantee all these contracts, otherwise, the unsafe call may lead to undefined behavior [18].

Unsound Encapsulation of Unsafe Calls. According to unsafe code guidelines [54, 63], it is recommended to encapsulate unsafe code within safe functions. This strategy is commonly adopted in the standard library and third-party crates so that users can directly use the safe APIs without worrying about the safety requirements. A sound *safe encapsulation of unsafe calls* (EUC) must not impose additional requirements beyond the parameter types, ensuring that its correctness can be checked by the Rust compiler. To achieve this, the EUC itself must guarantee all the contracts. Otherwise, the encapsulation must be marked as `unsafe` and inherit all unguaranteed contracts by stating them in its *Safety* section. Once unsafe calls are encapsulated in a safe function without guaranteeing all contracts, it would introduce *unsoundness* problems. An *unsound* EUC means that if its caller inputs specific values, the unguaranteed contract can lead to undefined behavior, breaking the safety promise of Rust [73, 77]. Furthermore, the unsoundness of EUC can propagate through function calls and data flow, resulting in intricate defects and significantly undermining the advantages of Rust. Therefore, unsoundness is intolerable for the community [49] and some unsound EUCs have been disclosed as CVEs [19].

Challenges to Check Unsafe Calls. As illustrated in Figure 2, the *Safety* section describes the contracts in unstructured and flexible natural language. Besides, the *Safety* section of one unsafe API often consists of several contracts, making the contract description rather verbose and complicated. It also involves auxiliary descriptions (e.g., examples and consequences), resulting in terribly redundant content [15]. Therefore, it is challenging to analyze various *Safety* sections with conventional natural language process techniques. Moreover, some contracts requiring independent checks are unconsciously mixed in one sentence [18], which increases the potential of omissions in subsequent checks. Apart from the complicity of the *Safety* section, it is challenging to verify these contracts. Firstly, to complete this task, developers or LLMs must have sophisticated domain knowledge for contracts, some of which are related to features of Rust. Furthermore, checking contracts may require inferences in complex contexts, which could involve many structs, functions, traits, and variable types. Overall, it is laborious and error-prone to verify the soundness of unsafe calls by checking whether all contracts are guaranteed.

2.2 Preliminary

The previous study [18] defines 19 Safety Properties (SPs) based on the documents of unsafe APIs in the Rust standard library [65], focusing on the safety requirements of unsafe APIs and the undefined behavior that can be triggered. However, the study does not address how each SP can be guaranteed, which is crucial for the practical use of unsafe APIs. To bridge this gap, we conduct a preliminary study on how unsafe APIs are practically guaranteed in EUCs.

Despite the complexity and flexibility of EUCs, we propose that there are some high-level general patterns for EUCs to guarantee contracts. For instance, the contract "`i < len`" can be guaranteed by direct validation in `if` statement or inferable context (e.g., `i` is the remainder of `len`; `i < 0` and `len > 0`). According to features of different contract types, each type is associated with distinct guarantee patterns (GPs). In general, the preliminary study aims to summarize contract types and GPs corresponding to each type. Before the investigation, we refer to prior work [18] and set the criteria for contract types and GPs:

- **Unambiguous**: Each contract type has a clear example; same for GPs.
- **Non-overlapping**: The definitions of different contract types do not overlap; same for GPs.
- **Generalizable**: Each contract type must abstract many similar cases; same for GPs.
- **Essentiality**: Any unguaranteed contract would lead to undefined behavior.
- **Practical**: One contract can be guaranteed in any GPs corresponding to its contract type.

We analyzed the EUCs in the standard library [65] and the top 500 most downloaded crates in `crates.io` [16]. All repositories were cloned with the latest commit on GitHub on May 25, 2024. These crates span various fields and have been widely reviewed, ensuring the quality and comprehensiveness of the preliminary study. After collecting all EUCs and the documents of referenced unsafe APIs, the first and second authors conducted two rounds of manual analysis. The first round is dedicated to categorizing the *Safety* section content into distinct contract types and the second round aims to distill GPs for each contract type. In the first round, we analyzed the documents of involved unsafe APIs one by one to summarize the contract types and classify each mentioned contract. We started with an empty set of contract types and incrementally defined a new type when the target contract cannot be classified into existing types. The classification was conducted independently, but the two authors would discuss the new case to determine the name and definition of the new type. After finishing the initial labeling, we cross-checked the results to ensure consistency. In the second round, we distill GPs for each contract type individually based on the results of the first round. For a specific type, we manually examined the code and documents of all related EUCs. These EUCs are then aggregated into several subgroups according to how they guarantee the contract is upheld in all conditions. Subsequently, we conducted a cross-checking for this group and summarized guarantee patterns (GPs) corresponding to each subgroup. To ensure the criterion Practical, upon distilling GPs of a certain type, we would attentively verify whether all specific contracts of this type can be guaranteed in any of its GPs. Failure to do so indicates that

Table 1. Contract Types and Corresponding Guarantee Patterns

Contract Type	Definition of Contract Type	No	Guarantee Pattern	Definition of Guarantee Pattern
Aligned	The pointer must be properly aligned.	1	**Trusted Source**	Get the pointer from an aligned object.
		2	**Aligned Offset**	Make sure the computed offset of the pointer is aligned.
		3	**Type Invariant**	The type is intrinsically guaranteed to be aligned, e.g., u8.
Allocated	The pointer must point to writable allocated memory.	4	**Trusted Source**	Get the pointer from an allocated object or the trusted function.
Dereferencable	The memory range (indicated by one pointer and another pointer or length) must be within the bounds of a single object.	5	**Trusted Source**	Get the complete memory range from a single object.
		6	**Explicit Validation**	Explicitly validate whether the range is in-bound.
Initialized	The pointer must point to initialized memory so that the object is readable.	7	**Trusted Source**	Get the pointer from an initialized object or a trusted function.
		8	**Extra Mark**	Mark the state with extra Wrap, e.g., MaybeUninit.
		9	**No Read**	Do not read nor return the uninitialized data before initialization.
Not Null	The pointer must not be null.	10	**Trusted Source**	Get the pointer from an allocated object or a trusted function.
		11	**Explicit Validation**	Explicitly validate whether the pointer value equals Null or zero.
		12	**Extra Mark**	Mark the state with extra Wrap, e.g., Option, Result, NonNull.
Not Overflow	The pointer calculation must not cause overflow nor wrap around the address space.	13	**Trusted Source**	Get all parameters of calculation from trusted objects or functions.
		14	**Explicit validation**	Explicitly validate the value range will not overflow.
Type Constraint	The value must satisfy additional constraints beyond its type, e.g., Encoding and Parity.	15	**Trusted Source**	Get from object satisfying the constraint or trusted function.
		16	**Explicit Validation**	Explicitly validate whether the constraint is satisfied.
		17	**Inferable Context**	The value is initialized according to the constraint.
		18	**Type Invariant**	The constraint is satisfied intrinsically by the type invariant.
Numeric Relation	The direct or indirect numeric relation between variables or constants must be satisfied.	19	**Direct Validation**	Validate the numeric relation and handles all cases.
		20	**Inferable Context**	The relation can be inferred from the context.
Thread Safety	The function or the function call context must be thread-safe to avoid data race or deadlock.	21	**Type Invariant**	The function and the value passed are intrinsically thread-safe.
		22	**Additional Helper**	Use additional lock to protect the critical resource.
Locked	The specific lock must be held by current thread.	23	**Explicit Validation**	Explicitly validate the required lock is hold.
		24	**Extra Mark**	Mark the lock is held with extra Wrap, e.g., LockGuard.
Lifetime Coverage	The object must or must not outlive another object.	25	**Additional Helper**	Trigger interprocess lifetime checking with PhantomData.
		26	**Explicit Terminated**	Explicitly terminate the short-lifetime object before another one.
		27	**Extra Mark**	Mark the lifetime relation with extra lifetime identifiers.
End of Use	The method or the object must not be used again afterward.	28	**Explicit Terminated**	Explicitly terminate the target object before another use.
Exclusive	The generated object must not be accessed by the original reference or pointer during its lifetime.	29	**Taken Ownership**	The generated object takes the ownership from the origin.
		30	**Explicit Terminated**	Explicitly terminate the generated object before external access.
Workflow	The function call must follow specific workflow or the parameter must originate from specific source.	31	**Complete Encapsulation**	Encapsulate the complete workflow as a single method according to the requirements.
Unreachable	The specific code branch must not be reachable.	32	**Inferable Context**	The context guarantees specific branches are not reachable.
Environment	The compile environment or target must satisfy extra requirements, such as architecture, instruction set.	33	**Explicit Validation**	Explicitly validate whether the environment meets the requirement.
		34	**Extra Mark**	Use extra attributes, traits to trigger the compile-time checks.

this contract type is excessively general and abstract. Consequently, we will subdivide the group of current contract type into several new contract types and resummarize their GPs.

As shown in Table 1, we defined 16 contract types and 34 corresponding GPs. These 16 contract types emphasize requirements for distinct aspects, covering *Memory & Pointer*, *Value*, *Concurrency*, *Lifetime*, *Ownership*, *Dataflow*, and *Environment*. Owing to the criterion Generalizable, these contract types abstract multiple similar contracts. For example, the contract type *Type Constraint* involves encoding and parity, and *Numeric Relation* includes all numeric relations between variables or constants. According to the criterion Practical, contracts associated with distinct GPs are categorized into different types. For instance, the contracts "not null" and "not dangling" both fall under the SP *Allocated* [18], while "not null" is significantly easier to validate and should be classified as the new contract type *Not Null*. Despite the correlation between contract types, they do not overlap with others. Namely, the *Initialized* memory also implicitly requires to be *Allocated*. However, they are categorized into two types since *Initialized* has additional requirements.

Each contract type is associated with at least one GP, indicating how contracts of this type can be guaranteed practically. Note that these GPs do not necessarily represent explicit operations. In other words, they can represent the abstractions or invariants of the given context, e.g., a variable of `usize` is guaranteed to be non-negative. Besides, different GPs of one contract type are not in conflict but composable. For example, the return value of a trusted function (*Trusted Source*), can still be marked with an additional wrapper (*Extra Mark*). Given the correlation between contract types, different contract types may have the same GPs, representing the practical scenarios in which multiple contract types are guaranteed simultaneously. Moreover, many GPs are related to Rust features, such as *Extra Mark*, *Additional Helper*, and *Taken Ownership*.

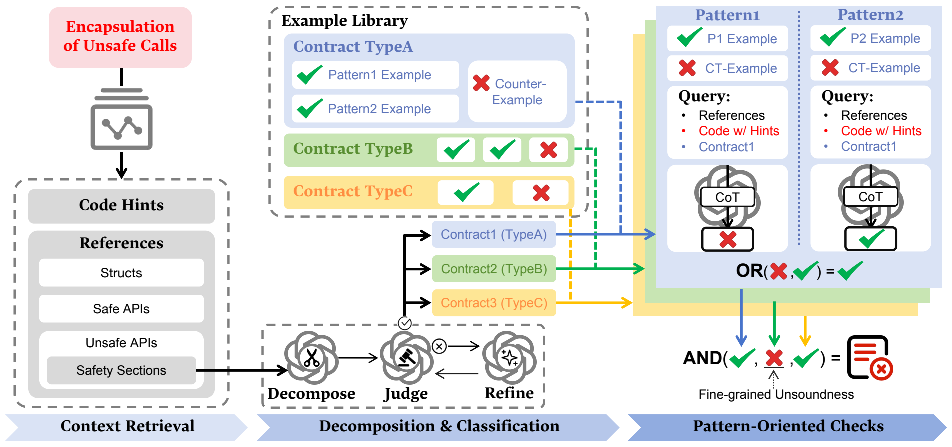


Fig. 3. Overview of Safe4U. The ✔ and ✘ denote the final answer of *Guaranteed* and *Unguaranteed* in both examples and response. **Context Retrieval**: Retrieving context of EUCs with static analysis tools. **Decomposition & Classification**: Utilizing the LLM to decompose the *Safety* section into fine-grained classified contracts and ensure the quality with Self-Judge and Refine. **Pattern-Oriented Checks**: Checking whether the contract is guaranteed in any corresponding GP. **Example Library**: Providing dynamic examples for Pattern-Oriented Checks.

3 Approach

In this section, we present our automated framework, Safe4U. Given an encapsulation that contains one unsafe call, Safe4U verifies its soundness by checking whether it guarantees all contracts described in its *Safety* section. As shown in Figure 3, Safe4U consists of three modules, i.e., Context Retrieval, Decomposition & Classification, and Pattern-Oriented Checks. Initially, the Context Retrieval module retrieves the context from the Rust project using static analysis tools. The context includes the *Safety* section of the interior unsafe API. Then, the Decomposition & Classification module applies the LLM to decompose and classify the original *Safety* section into several fine-grained classified contracts. To ensure the quality of decomposition and classification, the LLM is further leveraged to self-judge and refine the decomposition results iteratively. Subsequently, in the Pattern-Oriented Checks module, the LLM checks each fine-grained contract independently. Particularly, one contract is analyzed in multiple parallel rounds and each round corresponds to one guarantee pattern (GP) associated with its contract type. In each round, the LLM determines whether the target contract is *Guaranteed* by the EUC in the specific GP. The determinations from parallel rounds are aggregated with *OR* operation to obtain the identification result for one contract. After checking all contracts, the results are further aggregated with *AND* operation to obtain the identification result for the EUC. Specifically, Safe4U considers this EUC *Sound* only if all its contracts are *Guaranteed*. Otherwise, it is *Unsound*, and the *Unguaranteed* contract is identified as *fine-grained unsoundness* which requires a corresponding fix.

3.1 Context Retrieval

Safe4U retrieves two kinds of context with static analysis tools, including code hints and references. Code hints are code snippets that can be attached to the original code to make the code more informative, involving inferred variable types and parameter names displayed in function calls. References include information about items (e.g., structs, functions, and traits) referenced by the target EUC.

3.1.1 Code Hints. Rust is an implicit statically typed language, meaning the variable types can be inferred from their code context instead of being explicitly annotated. Additionally, the contracts of unsafe APIs are typically described with parameter names, so soundness checking requires matching the specific parameters with local variables. Such a match process is error-prone for LLMs and always leads to unfaithful responses. For example, `unsafe fn foo(a: u8, b: u8) -> u32` has a contract "*a must be larger than b*" and the caller is `let c = foo(b, a)`. It is confusing for LLMs to match the variables `b, a` to the parameters `a, b`, not to mention checking the unsafe call. Furthermore, the type of new variable `c` is unclear. The variable types could supplement important information for guaranteeing contracts, e.g., variables of `usize` must be non-negative. After adding code hints, the caller code becomes `let c: u32 = foo(a: b, b: a)`, which is clearer for the explicit variable type and the unambiguous matching relationship between variables and parameters.

To acquire the variable types precisely and efficiently, we construct a retriever to obtain the type inference results from the compiler. Subsequently, inferred types are attached to the original code, within formats that conform to Rust syntax. In particular, for a new variable declared with `let var` statement, the type is append behind the variable "`: type`". As for the return type of closure, its format is "`-> type`". To obtain the correspondence between parameters and variables passed into the callee function, Safe4U analyzes the parameter list declared in its function signature. Then, the parameter names are added preceding the variable names in the format "`para_name: var_name`".

3.1.2 References. Given the final target of practical scans, the target EUC derives from real-world projects and often involves many external items, such as structs and functions. If the references of items are not provided, the LLM always outputs unexpected answers that contain notable hallucinations. Thus, it is crucial to provide information about these items in the prompt.

To implement reference retrieval, we extend *Rust Analyzer*, the language server of Rust [47, 64]. First, by analyzing the abstract syntax tree of the target EUC, Safe4U extracts a set of referenced items. Then, it collaborates with Rust Analyzer, which analyzes the whole project, to retrieve the documents and code of these items. Since the references could expand exponentially, Safe4U only involves items that are directly referenced. However, the retrieved information may still be too redundant to be completely added to the prompt. Thus, Safe4U slices the reference with the following rules. The code snippets of `structs` (i.e., struct definitions) are completely kept since their fields are informative. In terms of the code of referenced functions and macros, the slicer omits their lengthy implementation details, retaining only their signatures. Similarly, despite the value of the documents, they are too verbose to be used directly. Considering the trade-off, we retain the description sections in the documents, which briefly summarize the functionality of items. In addition, the *Safety* sections of used unsafe APIs are extracted for further checks.

3.2 Decomposition & Classification

As described in the Background 2.1, an unsafe API typically has multiple contracts, which are mixed in the *Safety* section. To identify the soundness of the target EUC, we have to check whether all of these contracts are guaranteed, i.e., check these contracts one by one. However, the original *Safety* section is too complicated and redundant to be directly checked by the LLM. Therefore, it is crucial to decompose the original *Safety* section into non-overlapping fine-grained contracts. Besides, previous studies have demonstrated that the decomposition of questions can remarkably improve the faithfulness of model-generated reasoning and overall performance [13, 51, 74]. Overall, Safe4U applies the LLM to decompose the original *Safety* section into fine-grained contracts so that each contract can be checked independently. The decomposition has the following requirements: 1) *Consistency*: The decomposed contracts must derive from the original *Safety* section and cover all described safety requirements. 2) *Non-overlapping*: The decomposed contracts cannot overlap with others. 3) *Atomic*: The decomposed contracts cannot be further decomposed. 4) *Exclusively-Typed*: Each decomposed contract can only belong to one categorical contract type. 5) *Clarity*: Each decomposed contract should be condensed but clear. To achieve *Exclusively-Typed* and promote further analysis, we display the names and definitions of all contract types T in the prompt and require the LLM to classify each decomposed contract c simultaneously. Compared to the strategy of classifying after decomposition, integrating these two objectives into a single step gives the LLM a clearer understanding of the decomposition criteria, thereby achieving a more thorough and accurate decomposition. For more accurate classification, the prompt also involves seven demonstrations to invoke the ICL capability of LLM. The detailed prompt is displayed in the replication package and we denote LLM with this specific prompt as $LLM_{dec\&cls}$. By decomposing and classifying the *Safety* section $safety$, Safe4U gets n_c fine-grained contracts c and corresponding types $type$.

The quality of the decomposition and classification results is essential since they could significantly influence the subsequent checks. To this end, Safe4U employs the LLM to further self-judge and refine the decomposition results [24]. Specifically, LLM with a judge prompt LLM_{jud} is used to check whether the current decomposed contracts meet all requirements and are classified correctly. The judge provides elaborate review comments towards each requirement and ultimately determines whether the refinement is needed. If necessary, Safe4U passes $safety$, current decomposed contracts, and the judge comments to LLM with a refinement prompt LLM_{ref} to refine the decomposition results. This process is repeated iteratively until LLM_{jud} confirms that no further refinement is required. To avoid infinite self-judge loops, the maximum number of iterations is limited to $round\_limit$. Overall, the pseudocode of *Decomposition & Classification* with iterative Self-Judge and Refine is defined as follows:

Algorithm 1: Decomposition & Classification

```
  Input: safety
  Output: $S = [(c_i, \text{type}_i)]$ where $i \in [1, n_c]$
1 $S \leftarrow LLM_{dec\&cls}(safety)$;
2 for $r = 1$ to $round\_limit$ do
3     $judgement \leftarrow LLM_{jud}(S)$;
4     if $refinement\_needed(judgement)$ then
5         $S \leftarrow LLM_{ref}(safety, S, judgement)$;
6     else
7         break
8     end
9 end
```

3.3 Pattern-Oriented Checks

3.3.1 Example Library. Prior works demonstrate that LLMs have in-context learning capability to learn knowledge from few-shot demonstrations without updating parameters [25, 56]. To fully activate the in-context learning capability of LLMs, Safe4U includes a set of pre-constructed examples that can be added to the prompt as demonstrations. There are two categories of examples, namely *Pattern Example* and *Counter-Example*. Each Pattern Example corresponds to one GP, describing how a contract is guaranteed by the EUC in this GP. Contrarily, the Counter-Example explains why a contract is considered *Unguaranteed*. According to the 16 contract types and 34

GPs defined in the preliminary, the example library involves 34 Pattern Examples and 16 Counter-Examples. Both Pattern Examples and Counter-Examples are composed of a query and a check answer. Particularly, the check query consists of references, code of EUC, and one fine-grained contract to be checked. The check answer is the human-written step-by-step analysis, which implicitly demonstrates the specific guarantee pattern or explains why the contract is unguaranteed.

3.3.2 Basic Process. To check whether the contract c_i is guaranteed, Safe4U instructs the LLM to answer the query $query_i$, which is constructed by filling the $template$ with sliced references ref, code augmented with code hints $code'$, and the target contract c_i. To provide accurate domain knowledge to the LLM, Safe4U selects examples from the example library according to the contract type type_i of the target contract c_i. Nevertheless, the significant differences between guarantee patterns (GPs) make it challenging for LLMs to learn all GPs simultaneously. Therefore, to check contract c_i, Safe4U performs multiple rounds of checks, each corresponding to one of the GPs P_{type_i} associated with type_i. In particular, for the round of GP p, Safe4U picks a Pattern Example exp_p associated with GP p and a Counter-Example $CT\text{-}exp_{\text{type}_i}$ corresponding to type_i. In the round of GP p, the LLM finally makes the determination $g_{i,p}$, meaning contract c_i is *Guaranteed* by the EUC in GP p or not, as defined below:

$$\begin{aligned} query_i &= template\left(ref, code', c_i\right),\ i \in [1, n_c] \\ g_{i,p} &= LLM_{\text{check}}\left(exp_p, CT\text{-}exp_{\text{type}_i}, query_i\right),\ p \in P_{\text{type}_i} \end{aligned} \tag{1}$$

After pattern-oriented checks, we get determinations $g_{i,p}$ corresponding to each GP p. With *OR* operation, these determinations $g_{i,p}$ are aggregated to g_i as the determination for contract c_i. Ultimately, considering a sound EUC must guarantee all contracts, all determinations of every contract are further aggregated with *AND* operation as the final determination *soundness*, indicating whether the EUC is sound or not. These aggregating steps are defined below:

$$\begin{aligned} g_i &= OR\left[g_{i,p}\right],\ p \in P_{\text{type}_i},\ i \in [1, n_c] \\ soundness &= AND\left[g_i\right] \end{aligned} \tag{2}$$

If the fine-grained result g_i is *Unguaranteed*, indicating contract c_i is not guaranteed by the EUC and requires a corresponding fix. We define such an unguaranteed fine-grained contract c_i as a *contract-level unsoundness.*

3.3.3 Chain-Of-Thought (CoT). To effectively invoke the reasoning capability of the LLM, Safe4U instructs the LLM to check with CoT [13, 71]. Instead of simply asking the LLM to analyze step by step, it requires the LLM to follow the given procedure:

(1) Locate the code snippet of the unsafe call.
(2) List the variables related to the contract.
(3) Analyze step by step towards the contract.
(4) Determine whether the contract is guaranteed.

In the first step, the LLM explicitly clarifies its analysis goal by repeating the code, which contributes to minimizing input-conflicting hallucinations [67]. Similarly, in the second step, the LLM extracts key expressions from the code, thereby promoting subsequent reasoning steps. Step three is the main part of reasoning where the LLM leverages inherent capabilities and domain knowledge acquired from the given examples to conduct a thorough, step-by-step analysis. Ultimately, the LLM is required to answer *Yes* or *No* as the final determination, enabling the automation of output processing. This check chain is both stated in the $system$ prompt and demonstrated by the few-shot examples. The difficulty of these four steps increases gradually, and each step depends on the output of the former. Furthermore, CoT can also make the output more interpretable [13], allowing human reviewers to check whether it is a false alarm effortlessly.

3.4 Handle Corner Cases

To improve generalizability, Safe4U implements alternative schemes to address corner cases that could damage the automation. First, since Safe4U checks soundness according to the *Safety* section, unsafe APIs lacking document or *Safety* section will be filtered. During the *Decomposition & Classification* step, the LLM may accidentally classify the contract into some undefined types or directly output *Unknown*. For these contracts, Safe4U obtains their embedding and then determines the contract type by identifying the most similar contract in the example library with cosine similarity. When checking contracts, although LLMs are required to finally answer *Yes* or *No*, they may output *Unknown* if they are not sure about the determination. When aggregating results, Safe4U treats *Unknown* as a variant of *Unguaranteed*.

For EUC involving more than one unsafe call, Safe4U decomposes each of their *Safety* sections and conducts pattern-oriented checks for all fine-grained contracts. Similarly, this EUC is deemed as *Sound* only if all contracts are identified as *Guaranteed*. Additionally, if any contracts of the unsafe call are *Unguaranteed*, this unsafe call is considered unsound, named *function-level unsoundness*.

4 Experimental Setup

4.1 Research Questions

In the evaluation, we answer the following research questions with experiments:

- **RQ1: How generalizable and effective is Safe4U?** Safe4U is a generalizable framework that can be applied to different LLMs. Accordingly, in this RQ, we first apply Safe4U to various LLMs to evaluate its generalizability. Besides, to assess the effectiveness of Safe4U, we compare Safe4U with state-of-the-art non-LLM techniques.
- **RQ2: How effective are the components in Safe4U?** There are multiple elements in Safe4U, including *References (Ref)*, *Code Hints (Hints)*, *Chain of Thought (CoT)*, *Decomposition (Dec)*, *Classification (Class)*, *Self-Judge (Judge)*, and *Pattern-Oriented Checks (Pattn)*. To investigate the contribution of each component, in this RQ, we conduct a comprehensive ablation experiment.
- **RQ3: How effective is Safe4U in locating fine-grained unsoundness?** Safe4U provides fine-grained unsoundness analysis for each contract. These fine-grained analyses may be helpful for human reviewers to locate and validate the source of unsoundness, i.e., unsound unsafe calls and unguaranteed contracts. To quantify the value of these fine-grained responses, in this RQ, we manually label the ground truth of fine-grained contracts and evaluate the results in two levels:
  - **Function-Level**: Locate the unsound unsafe call from all interior unsafe calls.
  - **Contract-Level**: Locate the fine-grained *Unguaranteed* contracts.
- **RQ4: How effective is Safe4U in different types?** Apart from the capabilities of the applied backbone LLM, the type of contract could be a significant factor affecting the performance of Safe4U, i.e., some contract types may be more difficult to analyze. Accordingly, in this RQ, we investigate the relationship between the performance of unsoundness identification and contract types.

4.2 Data Collecting

Both sound and unsound samples are required to evaluate the performance of identifying unsound EUCs. However, there is no existing dataset and it is impractical to build samples manually. Thus, we design the following procedures to collect real-world EUCs from crates in practice.

Public Functions. Third-party crates commonly create *private* helper functions that are not completely sound and intentionally mark them as `safe` for code reusing. Since these private functions are not accessible outside the crate, this strategy is acceptable if the maintainers ensure that these unsound functions are used correctly. Hence, we collect public functions that directly encapsulate interior unsafe calls to prevent data pollution.

Sound Samples. As discussed in Background 2.1, the Rust community recommends developers write a *Safety* comment above every unsafe call to explain why it is safe, i.e., how the contracts are guaranteed. The *Safety* comment reveals the developers' awareness and respect for the contracts. Thus, the safe EUCs with *Safety* comments are supposed to be sound. In addition, these EUCs derive from popular and commonly used crates, meaning that they have been frequently reviewed, tested, and widely verified in practice. Therefore, the public EUCs with complete *Safety* comments can be considered sound samples approximately. To prevent the LLM from referring to the *Safety* comment, we delete all comments around unsafe calls.

Unsound Samples. Since there is no existing dataset for unsound EUCs and their amount in CVE [20] is inadequate for evaluation, we can only simulate unsound samples from unsafe encapsulations. Specifically, some unsafe encapsulations partially guarantee the contracts of interior unsafe calls and inherit the unguaranteed contracts by declaring them in their *Safety* sections. Accordingly, we simulate unsound samples from these unsafe encapsulations with the following steps. First, extracting public unsafe encapsulations of unsafe calls, together with both *Safety* sections of encapsulations and unsafe calls. Second, we manually check these unsafe encapsulations to determine whether they inherit any contracts of its interior unsafe callees by comparing the *Safety* sections. An encapsulation is filtered if it is `unsafe` for other reasons, such as dereferencing raw pointers or data race. Finally, the remaining unsafe encapsulations are preprocessed: 1) delete the `unsafe` keyword in the function signature, 2) add an unsafe block to the function body if needed, and 3) delete the *Safety* comments.

Sample Details. To ensure the quality of samples, we reuse the top 500 most downloaded crates on `crates.io` [16], which have been examined in the preliminary study. From these repositories, 244 public EUCs and 143 public unsafe encapsulations are extracted. Subsequently, the unsafe encapsulations are manually filtered by the first and second authors independently and the initial results achieve a Cohen's Kappa [44] value of 0.856, indicating a high inter-rater reliability. After cross-checking and reaching a consensus, we construct 105 simulated unsound samples. Then, 16 sound and 8 unsound samples are used to construct examples in the example library. Ultimately, we get an evaluation dataset with 325 samples, including 228 sound and 97 unsound samples. The ratio of unsound samples is 29.8%, which may be higher than the real-world fraction but still reflects the fact that unsound EUCs are less than sound ones. Note that these samples include no manual labels except the ground truth marking whether the EUC is sound. Furthermore, these samples involve 224 unsafe APIs, significantly exceeding that in the example library.

4.3 Implementation Details

Baselines. To measure the improvement introduced by Safe4U, we first adopt LLMs with the basic pipeline and prompt as the LLM baseline. Given the code of EUC and the original *Safety* section, the baseline queries the LLM to determine whether the EUC guarantees all contracts described in the *Safety* section. The LLM is instructed to respond either *Yes* or *No*, indicating the EUC is *Sound* or not. To objectively reflect the performance of Safe4U, we compare it with three state-of-the-art non-LLM techniques: 1) LockBud [48, 50]: An advanced static bug detector tailored for concurrency and memory bugs caused by unsafe Rust; 2) Rudra [8]: A static tool that efficiently recognizes three important bug patterns in unsafe Rust; 3) Kani [32]: A formal verification tool that can verify

Table 2. Results of Comparison Experiments. Table (a) shows the results of applying Safe4U to different LLMs. Table (b) presents the results of both Safe4U and state-of-the-art non-LLM techniques.

(a) Applying Safe4U to different LLMs

Method	Acc	Rec	Prec	F1
Llama	33.2%	**87.6%**	29.3%	43.9%
+Safe4U	**68.3%**↑106%	78.3%↓10.5%	**48.1%**↑64.1%	**59.6%**↑35.7%
Qwen-C	41.5%	62.9%	28.4%	39.1%
+Safe4U	**67.4%**↑62.2%	**92.8%**↑47.5%	**47.6%**↑67.8%	**62.9%**↑61.0%
Qwen	52.9%	52.6%	32.3%	40.0%
+Safe4U	**80.3%**↑51.7%	**57.7%**↑9.80%	**70.9%**↑120%	**63.6%**↑59.1%
GPT-4	40.9%	**97.9%**	33.3%	49.7%
+Safe4U	**87.7%**↑114%	88.7%↓9.47%	**74.8%**↑124%	**81.1%**↑63.1%

(b) Comparison with SOTAs

Method	Acc	Rec	Prec	F1	Succ
RG	50%	50%	29.8%	37.3%	100%
LockBud	70.1%	10.3%	40%	16.4%	100%
Rudra	34.7%	54.6%	18.6%	32.1%	42.2%
Kani	81.0%	**100%**	63.6%	77.8%	12.3%
Safe4U$_{\text{Qwen}}$	80.3%	57.7%	70.9%	63.6%	100%
Safe4U$_{\text{GPT}}$	**87.7%**	88.7%	**74.8%**	**81.1%**	100%

user-specific assertions, memory safety, run-time panics, and undefined behaviors in unsafe Rust. Typically, Kani requires manually constructing test harnesses for each EUC, but we automated the testing through scripts that generate harnesses based on the function signature. Additionally, we use *Random Guess (RG)* as a reference baseline. Due to the challenge of gathering enough task-specific data, fine-tuning models are not included as baselines.

Studied LLMs. Since Safe4U is a generic framework that can be applied to many LLMs, we study the RQ1 with the following LLMs to better evaluate the effectiveness of Safe4U: 1) Meta-Llama3.1-8b-Instruct (Llama) [62], 2) Qwen2.5-Coder-7B-Instruct (Qwen-C) [61], 3) Qwen2-7B-Instruct (Qwen) [60], and 4) gpt-4o-2024-05-13 (GPT-4) [4]. The first three open-source LLMs, each approximately 7B in size, offer a balanced trade-off between performance and deployment costs. Qwen-C is selected to examine whether code-specific LLMs are more suitable for Safe4U. GPT-4 is utilized to evaluate the generalizability of Safe4U on state-of-the-art LLMs. Considering the cost of GPT-4, we intend to utilize only open-source LLMs in actual deployment. Therefore, we choose the best-performing open-source LLM identified in RQ1 to conduct the following RQs. GPT-4 is accessed through the API [1] and the other LLMs are downloaded from Huggingface [2].

Parameter Settings. A previous study proposes that greedy decoding is typically more effective for reasoning tasks and coding problems [55]. In addition, we found that LLMs with higher temperatures are more likely to output responses in unexpected formats (for example, not providing "Yes" or "No" as definitive answers), which complicates the extraction of results using automated parsers. Accordingly, we perform greedy decoding with `temperature = 0` for all LLMs throughout the experiment. Moreover, we fix the random seed and keep other settings as the default. The *round_limit* of self-judge and refinement is set to 3.

Evaluating Metrics. Since our target is to identify unsound EUCs, we set unsound samples as positive while sound samples are negative. Based on that, we evaluate the effectiveness of Safe4U using four common metrics: *Accuracy (Acc)*, *Recall (Rec)*, *Precision (Prec)*, and *F1-score (F1)*. Moreover, given that analysis-based methods may fail on certain samples, we also considered the *success rate (Succ)* as a metric. In the computation of first four metrics, failed samples will be excluded.

5 Results

5.1 RQ1: How generalizable and effective is Safe4U?

Table 2a shows that Safe4U remarkably outperforms the LLM baseline in accuracy, precision, and F1-score for all studied LLMs. The consistent increase in these metrics reflects the generalizability of Safe4U. The LLM baseline does not have much advantage over random guessing, especially in accuracy and precision. This indicates that these LLMs themselves cannot identify unsound EUC well when lacking relevant contextual information and domain knowledge.

In addition, Table 2a shows that Safe4U brings different degrees of performance improvement to different LLMs, which is caused by the varied in-context learning and reasoning capabilities between LLMs. Among the selected LLMs, the performance of GPT-4 + Safe4U significantly surpasses others, owing to the superior capabilities of the backbone LLM. This means that the performance of Safe4U can be extended by deploying more advanced LLMs. Among other LLMs with similar parameter scales, Qwen + Safe4U achieves the best overall performance, with the highest accuracy of 80.3%, highest precision of 70.9%, highest F1-score of 63.6%, and an acceptable recall of 57.7%. Despite the similar performance in many benchmarks, these four LLMs' capability differences are magnified by Safe4U. In particular, a weak LLM is more likely to get non-atomic contracts, classify contracts into wrong types, and perform worse in final checks. Notably, Qwen-C surpasses Qwen in many code-related benchmarks [26, 28, 78], but it does not outperform Qwen in identifying unsound EUC according to F1-score. This demonstrates that Safe4U depends on the more comprehensive capabilities of LLMs, encompassing their understanding and reasoning abilities for NL and code, capacity for ICL, and so on.

Safe4U can notably improve precision but may sacrifice some recall. For example, Safe4U can increase the accuracy and precision of Llama by 106% and 64.1% respectively, but it will lose part of the recall. As shown in Table 2, we found that Llama and GPT-4 with the baseline achieve high recall scores but quite low precision, which is marginally better than Random Guss. These situations show that Llama and GPT-4 themselves are inclined to identify EUCs as unsound. After adopting Safe4U, the inherent propensities of LLMs are corrected, allowing a more reliable check.

Table 2b demonstrates that Safe4U also outperforms SOTA non-LLM techniques in identifying unsound EUCs, particularly in accuracy, precision, and F1-score. LockBud can merely identify 10.3% of unsound EUCs since it only supports specific bug patterns of unsafe code. Rudra detects 54.6% of unsound EUCs, but it also produces numerous false positives due to the lack of understanding of contextual semantics. Besides, Rudra depends on a specific version of middle-level intermediate representation, making it incompatible with projects that use newer compilers. Kani appears to identify all unsound EUCs with a precision of 63.6%. However, Kani's success rate only reaches 12.3%, as its automated harness generation is limited to simple samples. The simplicity of these successful samples leads to an overestimation of Kani's performance. In contrast, Safe4U supports all types of EUCs and is easy to maintain.

Answer for RQ1: Safe4U is generalizable to all studied LLMs and it can achieve better performance by applying more advanced LLMs. Safe4U outperforms both the LLM baseline and SOTA non-LLM techniques in identifying unsound EUCs, reflecting its effectiveness.

5.2 RQ2: How effective are the components in Safe4U?

We compare the performance of Safe4U with seven variants, each lacking a component of Safe4U. Additionally, considering the significance of CoT, we include a variant *w/ CoT* that adds CoT to the LLM baseline. To eliminate CoT, the LLM is instructed to directly answer *Yes* or *No*. For variants that ablate the references or the code hints, we remove the references from the prompt or provide the original code, respectively. Owing to the dependency relationship between *Safety Decomposition*, *Self-Judge*, *Contract Classification*, and *Pattern-Oriented Checks*, ablating the decomposition disables the other three parts and the pattern-oriented check is not available without the classification. The *w/o Dec* directly checks the original *Safety* section with random examples from the example

Table 3. Results of Ablation Experiment

Name	Settings							Accuracy	Recall	Precision	F1-Score
	Ref	Hints	CoT	Dec	Judge	Class	Pattn				
Qwen	☐	☐	☐	☐	☐	☐	☐	52.9% ↓34.1%	52.6% ↓8.93%	32.3% ↓54.5%	40.0% ↓37.1%
w/ CoT	☐	☐	✓	☐	☐	☐	☐	60.0% ↓25.3%	18.6% ↓67.9%	26.1% ↓63.2%	21.7% ↓65.9%
w/o CoT	✓	✓	☐	✓	✓	✓	✓	59.7% ↓25.7%	56.7% ↓1.79%	38.2% ↓46.1%	45.6% ↓28.3%
w/o Ref	☐	✓	✓	✓	✓	✓	✓	75.1% ↓6.51%	47.4% ↓17.9%	60.5% ↓14.6%	53.2% ↓16.4%
w/o Hints	✓	☐	✓	✓	✓	✓	✓	72.0% ↓10.3%	**64.9%** -	52.5% ↓25.9%	58.1% ↓8.76%
w/o Dec	✓	✓	✓	☐	☐	☐	☐	68.9% ↓14.1%	11.3% ↓80.4%	42.3% ↓40.3%	17.9% ↓71.9%
w/o Judge	✓	✓	✓	✓	☐	✓	✓	78.2% ↓2.68%	41.2% ↓28.6%	**74.1%** -	53.0% ↓16.7%
w/o Class	✓	✓	✓	✓	✓	☐	☐	77.8% ↓3.07%	47.4% ↓17.9%	68.7% ↓3.14%	56.1% ↓11.8%
w/o Pattn	✓	✓	✓	✓	✓	✓	☐	74.5% ↓7.28%	43.3% ↓25.0%	60.0% ↓15.4%	50.3% ↓21.0%
Safe4U	✓	✓	✓	✓	✓	✓	✓	**80.3%** -	57.7% -	70.9% -	**63.6%** -

library. Variant *w/o Class* checks decomposed contracts with examples selected based on embedding similarity. To ablate *Self-Judge*, *w/o Judge* simply uses the initial decomposition results for further identification. As for *w/o Pattn*, it checks each fine-grained contract in one round, provided with all Pattern Examples and one Counter-Example. Due to the best overall performance of Qwen in RQ1, it is selected for experiments in this RQ and all other settings are kept as the same.

The results are presented in Table 3, the best result for each metric is highlighted in bold and the second is underlined. The relative performance decreases compared to Safe4U are included to make the comparison more perceivable. Compared to Safe4U, all variants exhibit varying degrees of performance degradation, which underscores the contribution of each component to the identification of unsound EUCs. The most significant performance drop in *w/o Dec* and *w/o CoT* illustrates the critical role of decomposition and CoT in Safe4U. The notable drop in *w/o Dec* is expected since the absence of decomposition disables many other components. As for *w/o CoT*, ablating CoT results in the LLM's reasoning capability not being explicitly activated. However, the poor performance of *w/ CoT* and *w/o Dec* indicates that the lack of reasoning capability is not the only cause of the performance drop. The main issue is that without CoT, the answers in the given examples are limited to a simple *Yes* or *No*, which fails to provide the LLM with domain knowledge about guarantee patterns. Similarly, compared to *w/o Class* and *w/o Pattn*, Safe4U achieves better performance by providing the LLM with more accurate domain knowledge. As for *w/o Judge*, the performance difference indicates that the self-judge mechanism effectively improves the quality of the decomposition and classification.

Answer for RQ2: All ablating variants exhibit varying degrees of performance degradation, reflecting the effectiveness of each component. The notable performance drop observed after ablating decomposition and CoT underscores their crucial importance for Safe4U.

5.3 RQ3: How effective is Safe4U in Locating Fine-grained Unsoundness?

To evaluate the performance of Safe4U in locating fine-grained unsoundness, we first manually labeled the ground truth (GT) for the decomposed contracts. These contracts are exclusively marked as *Guaranteed* or *Unguaranteed*. A contract is considered *Guaranteed* only if it remains guaranteed regardless of how the EUC is used, otherwise, it is deemed *Unguaranteed*. Additionally, despite the Self-Judge during the decomposition, there are fabricated contracts that do not exist in the original *Safety* section due to the hallucination. We marked these contracts as *Hallucinated* and excluded them from the evaluation. Since the decomposed results for each LLM are different, we only manually labeled the GT based on the fine-grained contracts decomposed and self-judged by Qwen. The contract-level GT was labeled by the first and second authors independently and the initial results reached a Cohen's Kappa [44] value of 0.868. Then the results were cross-checked until a consensus was met on all contracts. The function-level GT is inferred from contract-level labels. Specifically, an unsafe call is considered as *Unsound* if any associated contract is *Unguaranteed*, otherwise, it is *Sound*.

We treat the *Unguaranteed* contracts as positive and evaluate the results with recall and precision. The results are presented in Table 4. For function-level evaluation, the precision of 98.2% reveals that the majority of unsafe calls alarmed by Safe4U are indeed unsound. Similarly, the contract-level precision of 94.1% reveals the effectiveness of Safe4U in locating *Unguaranteed* contracts. In addition to the precise classification, the output of Safe4U elaborates the reasoning process of identifying whether a contract is Unguaranteed, thereby reducing the burden to review. Overall, the fine-grained results of Safe4U can effectively assist human programmers in locating and validating unsound unsafe calls and unguaranteed contracts.

Table 4. Results of Locating Fine-grained Unsoundness

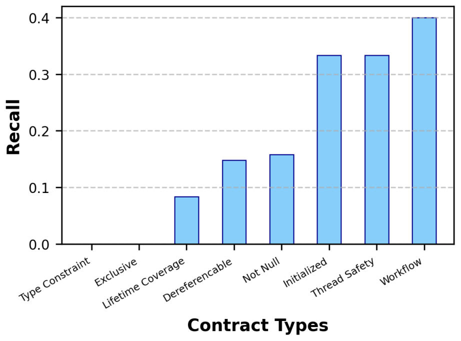
Granularity	Recall	Precision
Function-Level	57.4%	98.2%
Contract-Level	35.6%	94.1%

However, the function-level recall of 57.4% and contract-level of 35.6% indicates that Safe4U fails to locate many unguaranteed contracts. With detailed analysis, we find that the low recall may derive from the contradiction between the parallel checks of fine-grained contracts and the implicit topological relationships of these contracts. In practical scenarios, it is common for a fine-grained contract to depend on other contracts of the former unsafe calls. We label this contract as *Unguaranteed* if any of its predecessors is not *Guaranteed*. Nevertheless, in Safe4U, this fine-grained contract is checked independently. This means that the LLM is unaware of the other contracts, not to mention their implicit topological relationships. Consequently, the LLM assumes that other unsafe calls are sound, focuses solely on the current contract, and predicts it as *Guaranteed*. Take a simple code `data.get_unchecked(i).load()` for example, `load` has a contract "*The object must be initialized*". To check this contract, the LLM focuses on whether `data` is initialized regardless of the contract of `get_unchecked` that "`i` *must be in-bound*". Fortunately, the contradiction between topological relationships and parallel checks has little impact on identifying unsoundness in practical scenarios, where all contracts of every unsafe call ought to be guaranteed. In other words, despite the lower contract-level recall, checks regardless of topological relationships can better locate the real unsoundness source without predicting all implicated contracts as *Unguaranteed*.

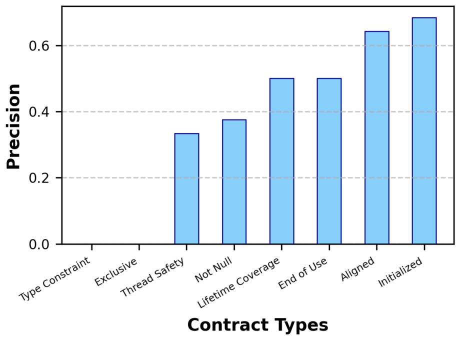
Answer for RQ3: Safe4U achieves good precision in identifying fine-grained unsoundness, although the recall is relatively low due to the topological relationships of contracts. Overall, the fine-grained results of Safe4U are beneficial for human programmers in locating and validating unsound unsafe calls and unguaranteed contracts.

5.4 RQ4: How effective is Safe4U in different contract types?

To investigate the relationship between the performance of Safe4U and contract types, we conduct a in-depth analysis for the fine-grained results of Qwen + Safe4U, including both sound and unsound samples. For sound samples, the expected results for all decomposed contracts are *Guaranteed*. As for unsound samples, we reuse the ground truth manually labeled in RQ3. To distinguish the performance in terms of missed detections and false positives across various types, we employ recall



(a) Contract Types with Lowest Recall



(b) Contract Types with Lowest Precision

Fig. 4. Results of Safe4U across Different Contract Types

and precision as evaluation metrics, and the results are shown in Figure 4. For clearer presentation, these contract types are sorted in ascending order based on precision or recall, with only the eight lowest values being presented.

Apparently, the recall and precision of different contract types vary significantly, indicating that some types are harder for Safe4U to analyze. It is noticeable that the recall and precision for Type Constraint and Exclusive are both zero. This is due to the limited amount of such types of Unguaranteed contracts in the evaluation dataset, all of which are identified as false negatives. Besides, Lifetime Coverage and Not Null exhibit both low recall and unsatisfactory precision. The subpar performance on Lifetime Coverage attributes to the fact that lifetime is a unique characteristic of Rust that requires complex reasoning abilities. As for Not Null, we conduct an in-depth analysis and find that numerous contracts of this type are described in formats like "`self` *must be non-null*". In these cases, the unsafe APIs are called like "`obj.read()`". As a result, the code hints of parameter names cannot be attached to the original code in the format "`para_name: var_name`". Consequently, the LLM may be confused about which variable the `self` refers to since the EUC may also have a parameter named `self`. These special cases highlight the significance of code hints and illustrate the limitation of current presentation format of code hints. This limitation also accounts for the suboptimal performance in other similar contract types. Fortunately, after reviewing the responses of Safe4U with GPT-4 in these cases, we discover that LLM with better capabilities is not influenced by this issue and can still provide an accurate analysis.

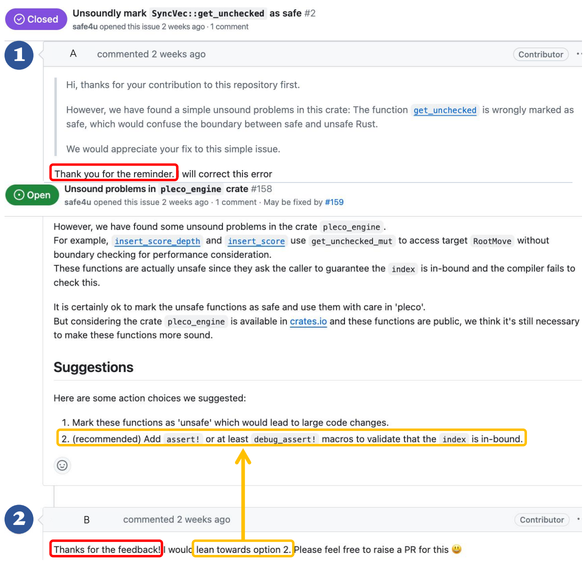
Answer for RQ4: Both the recall and precision of Safe4U significantly vary across distinct contract types, indicating that some types are more difficult to analyze. The suboptimal performance of some contract types is due to the intrinsic difficulty of the contract types, the limitation of presenting code hints, and the capabilities of LLMs.

6 Case Study

In this section, we deploy Safe4U to identify unsound EUCs in practical scenarios.

6.1 Evaluation on CVEs

We evaluate the performance of Safe4U with real-world unsound EUCs disclosed as CVEs. There are merely about 400 Rust CVEs by 2024 [20]. We refer to the organized dataset of previous work [77]


Closed Unsoundly mark `SyncVec::get_unchecked` as safe #2
safe4u opened this issue 2 weeks ago · 1 comment

1

A commented 2 weeks ago Contributor

> Hi, thanks for your contribution to this repository first.
>
> However, we have found a simple unsound problems in this crate: The function `get_unchecked` is wrongly marked as safe, which would confuse the boundary between safe and unsafe Rust.
>
> We would appreciate your fix to this simple issue.

Thank you for the reminder. I will correct this error

Open Unsound problems in `pleco_engine` crate #158
safe4u opened this issue 2 weeks ago · 1 comment · May be fixed by #159

However, we have found some unsound problems in the crate `pleco_engine`.
For example, `insert_score_depth` and `insert_score` use `get_unchecked_mut` to access target `RootMove` without boundary checking for performance consideration.
These functions are actually unsafe since they ask the caller to guarantee the `index` is in-bound and the compiler fails to check this.

It is certainly ok to mark the unsafe functions as safe and use them with care in 'pleco'.
But considering the crate `pleco_engine` is available in crates.io and these functions are public, we think it's still necessary to make these functions more sound.

Suggestions

Here are some action choices we suggested:

1. Mark these functions as 'unsafe' which would lead to large code changes.
2. (recommended) Add `assert!` or at least `debug_assert!` macros to validate that the `index` is in-bound.

2

B commented 2 weeks ago Contributor

Thanks for the feedback! I would lean towards option 2. Please feel free to raise a PR for this 😃

Fig. 5. Two responded issue reports of unsound EUCs found by Safe4U.

and manually reviewed the repair commits to find unsound EUCs. Unfortunately, most Rust CVEs are filtered for various reasons, such as no patch yet (16.8%), commit not found (13.6%), and unsafe unrelated (22.0%). 162 CVEs are associated with unsafe code, but the majority of them are screened out since they derive from raw pointers or concurrency. After excluding unsound EUCs missing *Safety* sections, we eventually extract 11 unsound EUCs with clear unguaranteed contracts and the metadata is accessible in the replication package. This also accounts for why we have to simulate unsound EUCs to construct the evaluation dataset.

The Llama3 with Safe4U identifies 9 of 11 sophisticated unsound EUCs from CVEs. The recall reaches 81.8%, close to its recall in the RQ1 (78.3%). This shows the effectiveness of the previous evaluation based on unsound samples simulated from unsafe EUCs. Despite the complexity of these real-world unsound vulnerabilities, Safe4U can identify their unsoundness, indicating the feasibility of applying Safe4U for practical scans.

6.2 Evaluation on crates.io

We further deployed Safe4U to scan crates on `crates.io` [16]. There are more than 150K crates on Jun 20, 2024, while we only focus on the top 10% most downloaded crates. The scan target is *public* EUCs, regardless of the existence of the *Safety* comment. The *Safety* comment (if it exists) around the unsafe call is removed so as not to affect the check. We deployed Safe4U with both Llama3 and Qwen2 to double-confirm the identified unsound EUCs, while other settings remain unchanged. The results of Safe4U are further reviewed by the first and second authors.

Since the top 500 have been used for evaluation, we first scanned the top 500 to 2,000 crates. After filtering out crates that are out of maintenance or have no public EUC, 931 EUCs are extracted from 131 repositories, involving 435 unsafe APIs. By deploying Safe4U and manually checking the results, we only detected one unsound EUC, which was instantly confirmed and fixed. This unsound EUC can be easily accessed in the crate `wasmtime-jit-debug`, which has over 5.9M downloads [9]. In general, the frequency of unsoundness derived from unsafe calls in top crates is notably low.

This meets our expectations since these top projects prioritize minimizing the use of unsafe code and mandate that any unsafe code undergo thorough reviews.

To investigate the situation regarding unsound EUCs in relatively less popular crates, we scan the most downloaded crates ranging from 10K to 15K, which are still in the top 10% with over 10K downloads. There are 1,918 EUCs from 281 repositories, involving 1,046 unsafe APIs. By checking the results of Safe4U, we ultimately detected 21 unsound EUCs, 15 of which have been confirmed and fixed. The average number of EUCs per repository (6.83) is comparable to the top 2K projects (7.11), but the frequency of unsoundness increases remarkably. This indicates the deficiencies of reviews despite thousands of downloads. The mismatch between the need for code review and the limited manpower underscores the importance of automated unsoundness detection for the community.

Figure 5 shows two cases of our confirmations, which received thanks from developers. Typically, we provide repair suggestions in issue reports, roughly divided into (1) Marking the unsound EUC as `unsafe` and documenting the unguaranteed contracts in its *Safety* section, and (2) Adding corresponding guarantee patterns or replacing the unsafe calls with safe alternatives to make the EUC sound. As shown in Figure 5.❷, the latter is not only recommended by us but also preferred by developers, as it requires fewer code changes and does not introduce additional unsafe code.

7 Threats to Validity

Threats to Internal Validity. The evaluation dataset and examples in the example library derive from identical crates, meaning that the performance of Safe4U may be overestimated owing to the implicit correlation. In the ablation experiment, the variant *w/o CoT*, which includes examples identical to Safe4U without providing the pattern-specific analysis, performs significantly worse than Safe4U. This result reflects that it is the abstract domain knowledge that improves the performance instead of the label of examples. Therefore, the impact of implicit correlation is negligible. Another threat to internal validity is that the unsound samples in the evaluation dataset are simulated from unsafe encapsulations instead of real unsound EUCs, so the evaluation results can be biased. Nevertheless, there is no existing dataset for unsound EUCs. We had attempted to collect real-world unsound EUCs from CVEs but the number is insufficient for evaluation. Future work is required to collect more real-world unsound EUCs to better evaluate Safe4U. The randomness of LLMs' output may considerably threaten the evaluation validity as well. For example, one fine-grained contract might be classified into distinct contract types in multiple passes, leading to significantly different results. Due to the cost limitation, we did not repeat experiments with different settings, e.g., different seeds and temperatures. To ensure fairness, we fix these parameters for all LLMs throughout the experiment for more consistent responses.

Threats to External Validity. The check procedure of Safe4U depends on the *Safety* section of the unsafe API, so the entire check will fail if the unsafe API lacks the *Safety* section. If the *Safety* section is incomplete or contains errors, Safe4U will incorrectly consider the EUC as sound. Fortunately, missing the Safety section of unsafe API would be warned by the lint checker [14], and the amount of EUCs with complete Safety sections remains substantial. Accordingly, it is still practically valuable to deploy Safe4U. Generating and checking the *Safety* sections are challenging upstream tasks of identifying unsound EUCs. We will address these tasks in future work, with the hope of eventually creating a complete toolchain. Another threat is that the contract types and guarantee patterns summarized in the preliminary study merely cover the standard library and a limited number of crates, which could also affect the generalizability of Safe4U. To mitigate this problem, we extend the number of examined crates to 500 to widen the scope and conduct a more comprehensive preliminary study. Furthermore, we consciously abstract and simplify the definition of contract types and guarantee patterns to enhance their generalizability. We will also open-source this project and continually complete contract types and corresponding patterns. Moreover, the current implementation of Safe4U encounters reliability challenges stemming from the probabilistic nature of LLMs and insufficient safeguard mechanisms. This is the trade-off between generalizability and trustworthiness, where Safe4U prioritizes broad applicability in the development phase. To enhance trustworthiness, we will integrate verification mechanisms in future versions.

8 Related Work

In this section, we introduce some related work to discuss the achievement of existing approaches and highlight the distinctiveness of our work.

Unsafe in Rust. The related research on unsafe in Rust can be roughly categorized into three groups, including formal verification [30, 35, 42, 43], unsafe isolation [5, 40, 53], and detecting vulnerabilities with static analysis [8, 17, 38, 45]. Formal approaches are typically designed to prove the safety promise of Rust. Unsafe isolation methods treat Rust programs with safe and unsafe code spaces and take various measures to ensure the security of the unsafe space. This strategy has excellent generalizability but brings in notable overhead. Prior works based on static analysis mainly focus on conventional unsafe operations, including raw pointer operations, memory allocation, and object deconstruction. These works require dedicated rules manually designed for specific targets. However, the contracts of different unsafe APIs vary significantly and are written in unstructured natural language, reflecting its requirements for cross-lingual comprehension. Thus, it is virtually unachievable to implement general detection for unsound EUCs merely through static analysis.

LLM for Vulnerability Detection. Two primary methods to detect vulnerabilities with code language models are fine-tuning pretrained models [22, 39, 75] and in-context learning (ICL) [36, 58, 59]. The fine-tuned techniques require dedicated datasets and have poor generalizability to unseen projects and weak robustness to noises in the code [10, 52, 57]. ICL is a new paradigm for various downstream natural language or code tasks without updating the parameters of LLMs [33, 72]. Furthermore, several effective techniques, particularly CoT variants [13, 71], can be integrated into ICL to enhance reasoning capabilities, showing significant promise. However, LLMs often perform poorly in emerging programming languages [46] and lack domain knowledge for identifying unsoundness, leading to notable hallucinations [21, 37, 68].

9 Conclusion

In this paper, we conduct a preliminary study and summarize 16 contract types and 34 corresponding GPs. Then we propose a novel framework, Safe4U, which incorporates LLMs, static analysis tools, and domain knowledge to identify unsound EUCs. The evaluation experiments show that Safe4U can bring generalizable performance improvements and its fine-grained results are valuable in locating detailed unsoundness. Additionally, Safe4U can identify 9 out of 11 unsound EUCs from CVE and detect 22 new unsound EUCs.

10 Data Availability

The replication package of our work is publicly available at [3].

Acknowledgments

This work was supported by the National Key R&D Program of China (No. 2024YFB4506400) and sponsored by CCF-Huawei Populus Grove Fund.

References

[1] 2024. API Reference - OpenAI API. https://platform.openai.com/docs/api-reference. Last accessed Jun. 2024.

[2] 2024. Models - Hugging Face. https://huggingface.co/models. Last accessed Jun. 2024.

[3] 2024. Replication package. https://github.com/huanli-00/Safe4U-replication.git. replication package.

[4] Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. 2023. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774* (2023).

[5] Hussain M. J. Almohri and David Evans. 2018. Fidelius Charm: Isolating Unsafe Rust Code. In *Proceedings of the Eighth ACM Conference on Data and Application Security and Privacy, CODASPY 2018, Tempe, AZ, USA, March 19-21, 2018.* 248–255. https://doi.org/10.1145/3176258.3176330

[6] Jimin An, YunSeok Choi, and Jee-Hyong Lee. 2024. Code Defect Detection Using Pre-trained Language Models with Encoder-Decoder via Line-Level Defect Localization. In *Proceedings of the 2024 Joint International Conference on Computational Linguistics, Language Resources and Evaluation, LREC/COLING 2024, 20-25 May, 2024, Torino, Italy*, Nicoletta Calzolari, Min-Yen Kan, Véronique Hoste, Alessandro Lenci, Sakriani Sakti, and Nianwen Xue (Eds.). ELRA and ICCL, 3446–3456. https://aclanthology.org/2024.lrec-main.306

[7] Vytautas Astrauskas, Christoph Matheja, Federico Poli, Peter Müller, and Alexander J. Summers. 2020. How do programmers use unsafe rust? *Proc. ACM Program. Lang.* 4, OOPSLA (2020), 136:1–136:27. https://doi.org/10.1145/3428204

[8] Yechan Bae, Youngsuk Kim, Ammar Askar, Jungwon Lim, and Taesoo Kim. 2021. Rudra: Finding Memory Safety Bugs in Rust at the Ecosystem Scale. In *SOSP '21: ACM SIGOPS 28th Symposium on Operating Systems Principles, Virtual Event / Koblenz, Germany, October 26-29, 2021.* 84–99. https://doi.org/10.1145/3477132.3483570

[9] bytecodealliance. 2024. wasmtime-jit-debug crate. https://crates.io/crates/wasmtime-jit-debug. Last accessed Jun. 2024.

[10] Saikat Chakraborty, Rahul Krishna, Yangruibo Ding, and Baishakhi Ray. 2022. Deep Learning Based Vulnerability Detection: Are We There Yet? *IEEE Trans. Software Eng.* 48, 9 (2022), 3280–3296. https://doi.org/10.1109/TSE.2021.3087402

[11] Junkai Chen, Zhiyuan Pan, Xing Hu, Zhenhao Li, Ge Li, and Xin Xia. 2024. Reasoning Runtime Behavior of a Program with LLM: How Far Are We? *CoRR* abs/2403.16437 (2024). https://doi.org/10.48550/ARXIV.2403.16437 arXiv:2403.16437

[12] Yujia Chen, Cuiyun Gao, Zezhou Yang, Hongyu Zhang, and Qing Liao. 2024. Bridge and Hint: Extending Pre-trained Language Models for Long-Range Code. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2024, Vienna, Austria, September 16-20, 2024*, Maria Christakis and Michael Pradel (Eds.). ACM, 274–286. https://doi.org/10.1145/3650212.3652127

[13] Zheng Chu, Jingchang Chen, Qianglong Chen, Weijiang Yu, Tao He, Haotian Wang, Weihua Peng, Ming Liu, Bing Qin, and Ting Liu. 2024. Navigate through Enigmatic Labyrinth A Survey of Chain of Thought Reasoning: Advances, Frontiers and Future. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2024, Bangkok, Thailand, August 11-16, 2024*, Lun-Wei Ku, Andre Martins, and Vivek Srikumar (Eds.). Association for Computational Linguistics, 1173–1203. https://doi.org/10.18653/V1/2024.ACL-LONG.65

[14] Rust Clippy. 2024. Clippy Lints of Missing Safety Document. https://rust-lang.github.io/rust-clippy/master/index.html#/missing_safety_doc. Last accessed Jun. 2024.

[15] Filipe Roseiro Côgo, Xin Xia, and Ahmed E. Hassan. 2023. Assessing the Alignment between the Information Needs of Developers and the Documentation of Programming Languages: A Case Study on Rust. *ACM Trans. Softw. Eng. Methodol.* 32, 2 (2023), 43:1–43:48. https://doi.org/10.1145/3546945

[16] The crates.io Team. 2024. The Rust community's crate registry. https://crates.io. Last accessed Jun. 2024.

[17] Mohan Cui, Chengjun Chen, Hui Xu, and Yangfan Zhou. 2023. SafeDrop: Detecting Memory Deallocation Bugs of Rust Programs via Static Data-flow Analysis. *ACM Trans. Softw. Eng. Methodol.* 32, 4 (2023), 82:1–82:21. https://doi.org/10.1145/3542948

[18] Mohan Cui, Shuran Sun, Hui Xu, and Yangfan Zhou. 2024. Is unsafe an Achilles' Heel? A Comprehensive Study of Safety Requirements in Unsafe Rust Programming. (2024), 106:1–106:13. https://doi.org/10.1145/3597503.3639136

[19] CVE. 2024. Common Vulnerabilities and Exposures. https://cve.mitre.org/cve/. Last accessed Jun. 2024.

[20] CVE. 2024. Common Vulnerabilities and Exposures related to Rust. https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=rust. Last accessed Jun. 2024.

[21] Yangruibo Ding, Yanjun Fu, Omniyyah Ibrahim, Chawin Sitawarin, Xinyun Chen, Basel Alomair, David A. Wagner, Baishakhi Ray, and Yizheng Chen. 2024. Vulnerability Detection with Code Language Models: How Far Are We? *CoRR* abs/2403.18624 (2024). https://doi.org/10.48550/ARXIV.2403.18624 arXiv:2403.18624

[22] Xu Duan, Jingzheng Wu, Shouling Ji, Zhiqing Rui, Tianyue Luo, Mutian Yang, and Yanjun Wu. 2019. VulSniper: Focus Your Attention to Shoot Fine-Grained Vulnerabilities. In *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI 2019, Macao, China, August 10-16, 2019.* 4665–4671. https://doi.org/10.24963/IJCAI.2019/

648

[23] Ana Nora Evans, Bradford Campbell, and Mary Lou Soffa. 2020. Is rust used safely by software developers?. In *ICSE '20: 42nd International Conference on Software Engineering, Seoul, South Korea, 27 June - 19 July, 2020*. 246–257. https://doi.org/10.1145/3377811.3380413

[24] Thomas Palmeira Ferraz, Kartik Mehta, Yu-Hsiang Lin, Haw-Shiuan Chang, Shereen Oraby, Sijia Liu, Vivek Subramanian, Tagyoung Chung, Mohit Bansal, and Nanyun Peng. 2024. LLM Self-Correction with DeCRIM: Decompose, Critique, and Refine for Enhanced Following of Instructions with Multiple Constraints. In *Findings of the Association for Computational Linguistics: EMNLP 2024, Miami, Florida, USA, November 12-16, 2024*. 7773–7812. https://aclanthology.org/2024.findings-emnlp.458

[25] Shuzheng Gao, Xin-Cheng Wen, Cuiyun Gao, Wenxuan Wang, Hongyu Zhang, and Michael R. Lyu. 2023. What Makes Good In-Context Demonstrations for Code Intelligence Tasks with LLMs?. In *38th IEEE/ACM International Conference on Automated Software Engineering, ASE 2023, Luxembourg, September 11-15, 2023*. 761–773. https://doi.org/10.1109/ASE56229.2023.00109

[26] Alex Gu, Baptiste Rozière, Hugh James Leather, Armando Solar-Lezama, Gabriel Synnaeve, and Sida Wang. 2024. CRUXEval: A Benchmark for Code Reasoning, Understanding and Execution. (2024). https://openreview.net/forum?id=Ffpg52swvg

[27] Sandra Höltervennhoff, Philip Klostermeyer, Noah Wöhler, Yasemin Acar, and Sascha Fahl. 2023. "I wouldn't want my unsafe code to run my pacemaker": An Interview Study on the Use, Comprehension, and Perceived Risks of Unsafe Rust. In *32nd USENIX Security Symposium, USENIX Security 2023, Anaheim, CA, USA, August 9-11, 2023*, Joseph A. Calandrino and Carmela Troncoso (Eds.). USENIX Association, 2509–2525. https://www.usenix.org/conference/usenixsecurity23/presentation/holtervennhoff

[28] Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. 2024. LiveCodeBench: Holistic and Contamination Free Evaluation of Large Language Models for Code. *CoRR* abs/2403.07974 (2024). https://doi.org/10.48550/ARXIV.2403.07974 arXiv:2403.07974

[29] Xuhui Jiang, Yuxing Tian, Fengrui Hua, Chengjin Xu, Yuanzhuo Wang, and Jian Guo. 2024. A Survey on Large Language Model Hallucination via a Creativity Perspective. *CoRR* abs/2402.06647 (2024). https://doi.org/10.48550/ARXIV.2402.06647 arXiv:2402.06647

[30] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. 2018. RustBelt: securing the foundations of the rust programming language. *Proc. ACM Program. Lang.* 2, POPL (2018), 66:1–66:34. https://doi.org/10.1145/3158154

[31] Sungmin Kang, Gabin An, and Shin Yoo. 2024. A Quantitative and Qualitative Evaluation of LLM-Based Explainable Fault Localization. *Proc. ACM Softw. Eng.* 1, FSE (2024), 1424–1446. https://doi.org/10.1145/3660771

[32] Kani. 2024. Kani Rust Verifier. https://github.com/model-checking/kani.

[33] Avishree Khare, Saikat Dutta, Ziyang Li, Alaia Solko-Breslin, Rajeev Alur, and Mayur Naik. 2023. Understanding the Effectiveness of Large Language Models in Detecting Security Vulnerabilities. *CoRR* abs/2311.16169 (2023). https://doi.org/10.48550/ARXIV.2311.16169 arXiv:2311.16169

[34] Heiko Koziolek, Sten Grüner, Rhaban Hark, Virendra Ashiwal, Sofia Linsbauer, and Nafise Eskandani. 2024. LLM-based and Retrieval-Augmented Control Code Generation. In *LLM4CODE@ICSE*. 22–29. https://doi.org/10.1145/3643795.3648384

[35] Andrea Lattuada, Travis Hance, Chanhee Cho, Matthias Brun, Isitha Subasinghe, Yi Zhou, Jon Howell, Bryan Parno, and Chris Hawblitzel. 2023. Verus: Verifying Rust Programs using Linear Ghost Types. *Proc. ACM Program. Lang.* 7, OOPSLA1 (2023), 286–315. https://doi.org/10.1145/3586037

[36] Haonan Li, Yu Hao, Yizhuo Zhai, and Zhiyun Qian. 2024. Enhancing Static Analysis for Practical Bug Detection: An LLM-Integrated Approach. *Proc. ACM Program. Lang.* 8, OOPSLA1 (2024), 474–499. https://doi.org/10.1145/3649828

[37] Yingcong Li, Kartik Sreenivasan, Angeliki Giannou, Dimitris Papailiopoulos, and Samet Oymak. 2023. Dissecting Chain-of-Thought: Compositionality through In-Context Filtering and Learning. (2023). http://papers.nips.cc/paper_files/paper/2023/hash/45e15bae91a6f213d45e203b8a29be48-Abstract-Conference.html

[38] Zhuohua Li, Jincheng Wang, Mingshen Sun, and John C. S. Lui. 2021. MirChecker: Detecting Bugs in Rust Programs via Static Analysis. In *CCS '21: 2021 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, Republic of Korea, November 15 - 19, 2021*. 2183–2196. https://doi.org/10.1145/3460120.3484541

[39] Zhen Li, Deqing Zou, Shouhuai Xu, Xinyu Ou, Hai Jin, Sujuan Wang, Zhijun Deng, and Yuyi Zhong. 2018. VulDeePecker: A Deep Learning-Based System for Vulnerability Detection. (2018). https://www.ndss-symposium.org/wp-content/uploads/2018/02/ndss2018_03A-2_Li_paper.pdf

[40] Peiming Liu, Gang Zhao, and Jeff Huang. 2020. Securing unsafe rust programs with XRust. In *ICSE '20: 42nd International Conference on Software Engineering, Seoul, South Korea, 27 June - 19 July, 2020*. 234–245. https://doi.org/10.1145/3377811.3380325

[41] Zhe Liu, Chunyang Chen, Junjie Wang, Mengzhuo Chen, Boyu Wu, Xing Che, Dandan Wang, and Qing Wang. 2024. Make LLM a Testing Expert: Bringing Human-like Interaction to Mobile GUI Testing via Functionality-aware Decisions.

In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering, ICSE 2024, Lisbon, Portugal, April 14-20, 2024*. 100:1–100:13. https://doi.org/10.1145/3597503.3639180

[42] Yusuke Matsushita, Xavier Denis, Jacques-Henri Jourdan, and Derek Dreyer. 2022. RustHornBelt: a semantic foundation for functional verification of Rust programs with unsafe code. In *PLDI '22: 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation, San Diego, CA, USA, June 13 - 17, 2022*. 841–856. https://doi.org/10.1145/3519939.3523704

[43] Yusuke Matsushita, Takeshi Tsukada, and Naoki Kobayashi. 2021. RustHorn: CHC-based Verification for Rust Programs. *ACM Trans. Program. Lang. Syst.* 43, 4 (2021), 15:1–15:54. https://doi.org/10.1145/3462205

[44] Mary L McHugh. 2012. Interrater reliability: the kappa statistic. *Biochemia medica* 22, 3 (2012), 276–282.

[45] Miri. 2019. An interpreter for Rust's mid-level intermediate representation. https://github.com/rust-lang/miri.

[46] Gabriel Orlanski, Kefan Xiao, Xavier Garcia, Jeffrey Hui, Joshua Howland, Jonathan Malmaud, Jacob Austin, Rishabh Singh, and Michele Catasta. 2023. Measuring the Impact of Programming Language Distribution. In *International Conference on Machine Learning, ICML 2023, 23-29 July 2023, Honolulu, Hawaii, USA*. 26619–26645. https://proceedings.mlr.press/v202/orlanski23a.html

[47] Zhiyuan Pan, Xing Hu, Xin Xia, and Xiaohu Yang. 2024. Enhancing Repository-Level Code Generation with Integrated Contextual Information. *CoRR* abs/2406.03283 (2024). https://doi.org/10.48550/ARXIV.2406.03283 arXiv:2406.03283

[48] Boqin Qin, Yilun Chen, Haopeng Liu, Hua Zhang, Qiaoyan Wen, Linhai Song, and Yiying Zhang. 2024. Understanding and Detecting Real-World Safety Issues in Rust. *IEEE Trans. Software Eng.* 50, 6 (2024), 1306–1324. https://doi.org/10.1109/TSE.2024.3380393

[49] Boqin Qin, Yilun Chen, Zeming Yu, Linhai Song, and Yiying Zhang. 2020. Understanding memory and thread safety practices and issues in real-world Rust programs. In *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*. 763–779. https://doi.org/10.1145/3385412.3386036

[50] Boqin Qin, Yilun Chen, Zeming Yu, Linhai Song, and Yiying Zhang. 2020. Understanding memory and thread safety practices and issues in real-world Rust programs. In *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*. 763–779. https://doi.org/10.1145/3385412.3386036

[51] Ansh Radhakrishnan, Karina Nguyen, Anna Chen, Carol Chen, Carson Denison, Danny Hernandez, Esin Durmus, Evan Hubinger, Jackson Kernion, Kamile Lukosiute, Newton Cheng, Nicholas Joseph, Nicholas Schiefer, Oliver Rausch, Sam McCandlish, Sheer El Showk, Tamera Lanham, Tim Maxwell, Venkatesa Chandrasekaran, Zac Hatfield-Dodds, Jared Kaplan, Jan Brauner, Samuel R. Bowman, and Ethan Perez. 2023. Question Decomposition Improves the Faithfulness of Model-Generated Reasoning. *CoRR* abs/2307.11768 (2023). https://doi.org/10.48550/ARXIV.2307.11768 arXiv:2307.11768

[52] Niklas Risse and Marcel Böhme. 2024. Uncovering the Limits of Machine Learning for Automatic Vulnerability Detection. (2024). https://www.usenix.org/conference/usenixsecurity24/presentation/risse

[53] Elijah Rivera, Samuel Mergendahl, Howard E. Shrobe, Hamed Okhravi, and Nathan Burow. 2021. Keeping Safe Rust Safe with Galeed. In *ACSAC '21: Annual Computer Security Applications Conference, Virtual Event, USA, December 6 - 10, 2021*. 824–836. https://doi.org/10.1145/3485832.3485903

[54] Rust-nomicon. 2024. The Rustonomicon. https://doc.rust-lang.org/nomicon/. Last accessed Jun. 2024.

[55] Yifan Song, Guoyin Wang, Sujian Li, and Bill Yuchen Lin. 2024. The Good, The Bad, and The Greedy: Evaluation of LLMs Should Not Ignore Non-Determinism. *CoRR* abs/2407.10457 (2024). https://doi.org/10.48550/ARXIV.2407.10457 arXiv:2407.10457

[56] Yisheng Song, Ting Wang, Puyu Cai, Subrota K. Mondal, and Jyoti Prakash Sahoo. 2023. A Comprehensive Survey of Few-shot Learning: Evolution, Applications, Challenges, and Opportunities. *ACM Comput. Surv.* 55, 13s (2023), 271:1–271:40. https://doi.org/10.1145/3582688

[57] Benjamin Steenhoek, Md Mahbubur Rahman, Richard Jiles, and Wei Le. 2023. An Empirical Study of Deep Learning Models for Vulnerability Detection. In *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023*. 2237–2248. https://doi.org/10.1109/ICSE48619.2023.00188

[58] Yuqiang Sun, Daoyuan Wu, Yue Xue, Han Liu, Wei Ma, Lyuye Zhang, Miaolei Shi, and Yang Liu. 2024. LLM4Vuln: A Unified Evaluation Framework for Decoupling and Enhancing LLMs' Vulnerability Reasoning. *CoRR* abs/2401.16185 (2024). https://doi.org/10.48550/ARXIV.2401.16185 arXiv:2401.16185

[59] Yuqiang Sun, Daoyuan Wu, Yue Xue, Han Liu, Haijun Wang, Zhengzi Xu, Xiaofei Xie, and Yang Liu. 2024. GPTScan: Detecting Logic Vulnerabilities in Smart Contracts by Combining GPT with Program Analysis. (2024), 166:1–166:13. https://doi.org/10.1145/3597503.3639117

[60] Alibaba Qwen Team. 2024. Qwen2 7B Instruct. https://huggingface.co/Qwen/Qwen2-7B-Instruct. Last accessed Jun. 2024.

[61] Alibaba Qwen Team. 2024. Qwen2.5-Coder 7B Instruct. https://huggingface.co/Qwen/Qwen2.5-Coder-7B-Instruct. Last accessed Jun. 2024.

[62] Meta Llama Team. 2024. Meta Llama3 8B Instruct. https://huggingface.co/meta-llama/Meta-Llama-3-8B-Instruct. Last accessed Jun. 2024.

[63] Rust Team. 2024. Mission Statement of the Secure Code Working Group. https://github.com/rust-secure-code/wg. Last accessed Jun. 2024.

[64] Rust Team. 2024. Rust Analyzer. https://rust-analyzer.github.io. Last accessed Jun. 2024.

[65] Rust Team. 2024. The Rust Standard Library. https://doc.rust-lang.org/std. Last accessed Jun. 2024.

[66] Rust Team. 2024. Usafe Code Guidelines. https://rust-lang.github.io/unsafe-code-guidelines/. Last accessed Jun. 2024.

[67] S. M. Towhidul Islam Tonmoy, S. M. Mehedi Zaman, Vinija Jain, Anku Rani, Vipula Rawte, Aman Chadha, and Amitava Das. 2024. A Comprehensive Survey of Hallucination Mitigation Techniques in Large Language Models. *CoRR* abs/2401.01313 (2024). https://doi.org/10.48550/ARXIV.2401.01313 arXiv:2401.01313

[68] Saad Ullah, Mingji Han, Saurabh Pujar, Hammond Pearce, Ayse K. Coskun, and Gianluca Stringhini. 2024. LLMs Cannot Reliably Identify and Reason About Security Vulnerabilities (Yet?): A Comprehensive Evaluation, Framework, and Benchmarks. In *IEEE Symposium on Security and Privacy, SP 2024, San Francisco, CA, USA, May 19-23, 2024*. 862–880. https://doi.org/10.1109/SP54263.2024.00210

[69] Nalin Wadhwa, Jui Pradhan, Atharv Sonwane, Surya Prakash Sahu, Nagarajan Natarajan, Aditya Kanade, Suresh Parthasarathy, and Sriram K. Rajamani. 2024. CORE: Resolving Code Quality Issues using LLMs. *Proc. ACM Softw. Eng.* 1, FSE (2024), 789–811. https://doi.org/10.1145/3643762

[70] Cunxiang Wang, Xiaoze Liu, Yuanhao Yue, Xiangru Tang, Tianhang Zhang, Cheng Jiayang, Yunzhi Yao, Wenyang Gao, Xuming Hu, Zehan Qi, Yidong Wang, Linyi Yang, Jindong Wang, Xing Xie, Zheng Zhang, and Yue Zhang. 2023. Survey on Factuality in Large Language Models: Knowledge, Retrieval and Domain-Specificity. *CoRR* abs/2310.07521 (2023). https://doi.org/10.48550/ARXIV.2310.07521 arXiv:2310.07521

[71] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed H. Chi, Quoc V. Le, and Denny Zhou. 2022. Chain-of-Thought Prompting Elicits Reasoning in Large Language Models. (2022). http://papers.nips.cc/paper_files/paper/2022/hash/9d5609613524ecf4f15af0f7b31abca4-Abstract-Conference.html

[72] Noam Wies, Yoav Levine, and Amnon Shashua. 2023. The Learnability of In-Context Learning. (2023). http://papers.nips.cc/paper_files/paper/2023/hash/73950f0eb4ac0925dc71ba2406893320-Abstract-Conference.html

[73] Hui Xu, Zhuangbin Chen, Mingshen Sun, Yangfan Zhou, and Michael R. Lyu. 2022. Memory-Safety Challenge Considered Solved? An In-Depth Study with All Rust CVEs. *ACM Trans. Softw. Eng. Methodol.* 31, 1 (2022), 3:1–3:25. https://doi.org/10.1145/3466642

[74] Eric Zelikman, Qian Huang, Gabriel Poesia, Noah D. Goodman, and Nick Haber. 2023. Parsel: Algorithmic Reasoning with Language Models by Composing Decompositions. (2023). http://papers.nips.cc/paper_files/paper/2023/hash/6445dd88ebb9a6a3afa0b126ad87fe41-Abstract-Conference.html

[75] Junwei Zhang, Zhongxin Liu, Xing Hu, Xin Xia, and Shanping Li. 2023. Vulnerability Detection by Learning From Syntax-Based Execution Paths of Code. *IEEE Trans. Software Eng.* 49, 8 (2023), 4196–4212. https://doi.org/10.1109/TSE.2023.3286586

[76] Yuchen Zhang, Ashish Kundu, Georgios Portokalidis, and Jun Xu. 2023. On the Dual Nature of Necessity in Use of Rust Unsafe Code. In *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*, Satish Chandra, Kelly Blincoe, and Paolo Tonella (Eds.). ACM, 2032–2037. https://doi.org/10.1145/3611643.3613878

[77] Xiaoye Zheng, Zhiyuan Wan, Yun Zhang, Rui Chang, and David Lo. 2024. A Closer Look at the Security Risks in the Rust Ecosystem. *ACM Trans. Softw. Eng. Methodol.* 33, 2 (2024), 34:1–34:30. https://doi.org/10.1145/3624738

[78] Terry Yue Zhuo, Minh Chien Vu, Jenny Chim, Han Hu, Wenhao Yu, Ratnadira Widyasari, Imam Nur Bani Yusuf, Haolan Zhan, Junda He, Indraneil Paul, Simon Brunner, Chen Gong, Thong Hoang, Armel Randy Zebaze, Xiaoheng Hong, Wen-Ding Li, Jean Kaddour, Ming Xu, Zhihan Zhang, Prateek Yadav, Naman Jain, Alex Gu, Zhoujun Cheng, Jiawei Liu, Qian Liu, Zijian Wang, David Lo, Binyuan Hui, Niklas Muennighoff, Daniel Fried, Xiaoning Du, Harm de Vries, and Leandro von Werra. 2024. BigCodeBench: Benchmarking Code Generation with Diverse Function Calls and Complex Instructions. *CoRR* abs/2406.15877 (2024). https://doi.org/10.48550/ARXIV.2406.15877 arXiv:2406.15877

Received 2025-02-26; accepted 2025-03-31