Towards Learning Unsafe Source Code Patterns in Rust Blocks

***Abstract*—Rust is a multi-paradigm general-purpose programming language, which ensures memory safety at compile time using its innovative borrow checker. With an emphasis on performance, Rust has also been established as a favourite programming language among developers. The main conduit for information exchange between programmers and Rust's safety system is the Rust compiler. Despite the mechanisms ensuring memory safety by the Rust compiler, the latter does not check type safety for all unsafe cases marked as unsafe by programmers who need to ensure their safety. To confirm whether or not each of unsafe cases can pass the safety check, it would require multiple time-consuming compilations. The use of computational approaches for identifying safety in Rust source code has become an emergent and promising field. However, existing work processes code in the same way as natural language, ignoring code-specific characteristics. Here, we investigate pre-trained programming language models that reserve rich code semantics. Moreover, we force the models to focus on the reformulated natural language processing (NLP) tasks and associated Rust code patterns. The experiments show that codeBERT is the most effective and efficient for classifying Rust source code. On the other hand, despite the variation in performance across different reformulations of NLP tasks, we do not notice a significant difference. We discuss the limitations and map future directions towards this field. Second, we show that encapsulating contextual information elicited by different types of inlay hints facilitates the learning process. Third, we analyse misclassified unsafe cases and explore prominent code patterns using an unsupervised clustering technique. The error analysis provides insights into safety rules that the best performance model fails to identify.**

***Index Terms*—Rust, safety, code representation learning, reformulated NLP tasks**

I. Introduction

Memory safety issues are a major source of security bugs in software and can cause random program crashes. Memory safety vulnerabilities are common. A recent study found that 60-70% of vulnerabilities in iOS and MacOS were memory safety vulnerabilities [1]; earlier Microsoft estimated that 70% of all vulnerabilities in their products over the last decade have been memory safety issues [2]; Google also estimated that 90% of Android vulnerabilities were memory safety issues [3]. On the other hand, Rust is a multi-paradigm, high-level, general-purpose programming language designed to address such memory safety issues without sacrificing much efficiency [4]. Rust code consists of two subsets: *safe* Rust and *unsafe* Rust. Safe Rust code guarantees no violation of memory safety properties. This guarantee is ensured at compile time using Rust's unique *Ownership* and *Borrow* Checking mechanisms. Ownership is a constraint ensuring a memory allocated value can only be owned by one variable at a time. The ownership mechanism manages the lifetime of variables and automatically frees them when their lifetime ends, preventing in this way memory issues, such as use-after-free, double-free, or other undefined behaviors [5]. Borrowing is a concept in Rust where a reference to a variable is borrowed without having ownership. There are two types of borrowing; *mutable*, i.e., a variable's value can be mutated, and *immutable*, i.e., granting only read permission. The borrow checker ensures that there can be multiple immutable references, but not both an immutable and mutable reference to the variable. Despite such restrictions ensuring memory safety issues, sometimes they are against some types of applications that require multiple mutable access, e.g., double-linked lists.

Unsafe Rust allows developers to bypass the above restrictions by enclosing unsafe operations in Rust code, e.g., dereference a pointer or call external library functions. In this case, the Rust code has to be marked with the "unsafe" keyword and the responsibility of memory safety is transferred from the Rust compiler to programmers. However, the unnecessary mixture of safe and unsafe Rust code can lead to thread safety issues [6]. That is especially common for Rust beginners who are unfamiliar with concepts such as ownership and lifetime. Rust-analyzer allows various code editors, such as VS Code and Emacs, through a language server protocol to show *inlay type hints* for elided lifetimes in function signatures. Inlay hints are special markers that appear in the editor and provide additional code information, but they do not affect functionality. Such an attribute in Rust might facilitate Rust developers to reason about thread safety issues.

The main conduit for information exchange between programmers and Rust's safety system is the Rust compiler. It compares itself to the aforementioned safety regulations and, if seeing a rule breach, reports an error. Due to the unsafe keywords, however, Rust compiler is not a silver bullet. Some people have expressed concern about the proliferation of unsafe code in Rust projects despite the language's safety guarantees [7]. When programmers develop applications, there is a need to suggest whether a part of Rust implementation is safe or not before waiting for the Rust compilation. However, to maximize the safety guarantees by the Rust compiler exhaustively, it would require 2^n compilations if a library crate has n blocks, i.e. the number of curly braces. Typically a compilation of a Rust crate requires 2^n minutes, which is prohibitively expensive. That raises the need for an automatic process of checking safety issues in blocks of Rust source code, bypassing multiple compilations that might burden software development in production. That motivates our current work to exploit neural approaches to identify unsafe source code and help Rust programmers to avoid safety risks when developing applications.

The use of neural approaches to tackle challenging software engineering problems, such as program functionality classification [8], bug localization [9], code summarization [10], code clone detection [11], program refactoring [12], program translation [12] , and code synthesis [13], has become an emergent and promising field. More advanced work exploits transformer-based architectures, such as codeBERT [14] and codeT5 [15], trained on multi-programming-lingual data to support downstream tasks. However, the use of neural approaches for studying the misuse of unsafe Rust code has seen less attention. Preliminary work utilizes a Siamese graph neural network to extract embedding from Rust code and then cosine similarity to search similar Rust code in a knowledge base that serves as an essential reference for Rust beginners [16]. Other work trains a classifier to learn the relation between unsafe code and memory safety bugs [17].

Despite such preliminary work showing promising results, there are many limitations and open research questions to be addressed in this field. First, all the above prior work simply processes code in the same way as natural language and largely ignores the code-specific characteristics. In particular, it does not count the identifier information in code over the pre-training process. Moreover, the tasks are prone to learn program language discrimination across Rust safe and unsafe source code ignoring explicit information entailed in the source code itself. This is because the optimization objective is to learn a unified label space and thus important source code patterns never have any gradient signal [18].

In this work, we exploit neural approaches to identify unsafe source code and recommend Rust programmers about the safety of source code bypassing the multiple time-consuming compilations of Rust compiler in software development. To this goal, we investigate state-of-the-art neural architectures that leverage the identifier information in Rust code and explore tasks that force the neural models to get signal from the source code patterns themselves over the optimization process. We conduct experiments on a collected corpus of 300 million lines of idiomatic Rust code in 'crates-io'. The contributions of this work have as follows:

- We investigate pre-trained programming language models, namely codeBERT [14] and codeT5 [15], that specifically focus on the identifiers that reserve rich code semantics for pre-training. Moreover, we force the models to focus on reformulated natural language processing (NLP) tasks and associated Rust code patterns investigating three different settings: a) standard fine-tuning, b) multitask fine-tuning with mask language model (MLM) objective and c) prompt-based fine-tuning. The experiments show the effectiveness of the proposed architectures over the baseline ones. In particular, fine-tuning codeBERT is the most effective and efficient way for classifying Rust source code (see Section V-A). One the other hand, despite the variation in performance across the three different reformulations of NLP tasks, we do not notice a significant difference. We discuss the findings in the Limitations section and map future directions towards this field (see Section VI).
- We encapsulate contextual information conveyed by various types of inlay hints and investigate how it facilitates the learning process. The experiments show that all the approaches improve the performance on identifying safety when the Rust code conveys additional information elicited by inlay hints (see Section V-B).
- We exploit an unsupervised data-driven approach to simultaneously learn feature representations and clustering assignments on the misclassified unsafe Rust cases as those inferred by the best performance model, i.e., codeBERT with standard fine-tuning. We then conduct an error analysis to investigate prominent patterns within the clusters. The error analysis provides insights into safety rules that are more difficult to be identified by the best performance model (see Section V-C).

II. Related Work

A. Rust Code

Existing work regarding the safeness of Rust code has mostly focused on empirical studies. For example, how unsafe code is used [6], [19], how many external C/C++ libraries are used by Rust libraries [20], and the buggy code patterns of safety issues that get past Rust's compiler checks [21] are just a few of the empirical studies that have been done to understand real-world Rust code from various points of view. The compilable Rust programmes have been the primary aspect of those empirical research.

Other work has conducted usability studies on Rust to understand how different factors impact programmers' learning [22]. These studies provide valuable findings and insights about Rust's grammar and the safety checks that impact its learning and programming software development processes. On the other hand, the Mozilla Rust team surveys Rust programmers every year to better understand their backgrounds and think of ways to make Rust better. According to a poll conducted in 2020, longevity and ownership are the two concepts that programmers find the most challenging to understand. Zeng and Crichton have analysed posts and comments from Rust communities and concluded that Rust is hampered by several factors [23]. Crichton has conducted a case study to investigate difficulties in deciphering Rust's compiler error signals [24]. Abtahi and Dietz have investigated the strategies over the Rust learning process [25]. They observed that compiler mistakes and online code examples were beneficial for learning Rust. Additionally, they concluded that Rust beginners occasionally found compiler error messages difficult to be interpretted. Fulton et al. have pinpointed the advantages and difficulties of adopting Rust by conducting online semi-structured interviews [26].

The use of computational approaches for identifying unsafe Rust code has received even less attention. Preliminary work has utilized a siamese graph neural network to extract embedding from Rust code, and then cosine similarity to retrieve Rust code from a knowledge base that acts as a crucial resource for Rust newcomers [16]. In other research, classifiers have been trained with the objective of recognising the connection between memory safety defects and dangerous code [17]. Such methods, however, frequently ignore explicit information entailed in the source code itself and instead force a model to learn the discrimination across Rust safe and unsafe source code. This is because the learning objective optimization targets learning a unified label space, and hence significant source code patterns never exhibit a gradient signal. In this study, we investigate approaches that force models to get signal from the source code itself through reformulated NLP tasks.

B. Code Representation Learning

A lot of recent research has attempted to apply NLP pre-training techniques to source code in the developing field of pre-training on programming languages. The two pioneer models are Cu-BERT [27] and CodeBERT [14]. To generate a generic code-specific representation, CuBERT uses BERT's potent masked language modelling objective. CodeBERT also includes a replace token task to train NL-PL cross-modal representation [28]. For code generation tasks, Svyatkovskiy et al. [29] and Liu et al. [30] use GPT and UniLM [31], respectively. Transcoder [32] explores programming language translation in an unsupervised setting. Different from them, codeT5 explores encoder-decoder models based on T5 for programming language pre-training to support a more comprehensive set of tasks [15].

The T5 framework has also explored in some recent work for code representation learning [33], [34]. However, such work only concentrates on a small-scale subset of generation tasks. In contrast, PLBART [35] is based on a different encoder-decoder BART model and can support both understanding and generating activities. However, all of the aforementioned earlier research just treats code in the same manner as spoken language and essentially overlooks the properties of the source code. Instead, in this work, we exploit neural models and reformulation tasks that leverage explicitly programme language patterns from the source code itself.

The data flow elicited by the code structure has recently been incorporated into CodeBERT by GraphCodeBERT [36], while Rozière et al. [37] suggest a deobfuscation objective to use the structure and context of programming language. However, these approaches only concentrate on improving the training of code-specific encoders. According to Zügner et al. [38], the relative distances between code tokens should be captured over the code structure, and this makes sequence to sequence architecture more appropriate, e.g., codeT5. To this end, in this work we specifically focus on pre-trained programming language models that reserve rich code semantics over pre-training process, such as codeBERT and codeT5.

C. Pretrained Language Models

Language models, a common NLP tool, are typically trained with the Cloze goal, which involves removing some of the context from a text and asking the model to guess what that context should have been [39]. The Cloze-based denoising objective known as "masked language modelling" (MLM) has been utilised extensively in pre-training language model learning [40]. In order to reuse previously trained language models, several works have reconstructed learning problems as cloze questions [41], [42]. In other research, task descriptions (prompts) and examples with annotations of demonstrations have been used to facilitate few-shot learning for downstream tasks [43], [44]. Due to their success in overcoming the problem of expensive data annotation and preprocessing [45], such methods have emerged as a significant study area. However, it can be challenging to develop ways to reformulate tasks as cloze questions that optimally utilise the knowledge held in language models, according to Schick et al. [46]. In this work, we are additionally exploring the task reformulation paradigm to investigate how the identification of unsafe Rust code benefits when models get signal from the source code itself over the optimization process.

III. Background

This section provides fundamentals about Rust's safety mechanism, inlay hints, and the information provided by the Rust compiler for safety-rule violations.

A. Rust's Safety Mechanism

Ownership and lifespan are two crucial ideas at the heart of Rust's safety mechanism. The fundamental rule stipulates that a value must have one and only one owner variable, and that when the lifetime of that owner variable expires, the value must be drooped (freed). Sometimes, such as at the conclusion of a function or a matched curly bracket, it is simple to establish where a variable's lifespan ends. However, there are several circumstances when determining a variable's lifespan is far more difficult than looking at its lexical scope. Rust expands its core safety rule into a set of rules while still ensuring memory safety and thread safety in order to increase programmatic freedom.

Ownership Move. Rust permits shifting a value's ownership to a new owner variable or to a different scope (such as a function or closure), but it forbids access to the old owner variable once the transfer is complete. For example, as illustrated in Figure 1, in line 7 the ownership of the resource v2 is transferred to display since the parameter type is v2 and not &v2. Thus, the Rust compiler reports an error at line 9, since v2 is invalidated after the move.

Ownership Borrow. In Rust, a reference that can be either immutable for read-only reads or mutable for read-write accesses can be used to temporarily borrow a variable's ownership. The last place where a reference was used is where a borrow terminates. A reference must only be used once during the lifetime of a borrowed variable in Rust. Rust allows more than one immutable reference to a variable to be present at once, but it only allows a single mutable reference to be present at any given moment. These constraints effectively ensure that all accesses to a variable are made throughout

```
fn main(){
    //  vector  v owns the object
    let  v = vec![1,2,3];
    //  moves ownership to v2
    let  v2 = v;
    //  v2 is  moved to display  and v2 is  invalidated
    display(v2);
    //v2  is  No longer usable  here
    println!(”In␣main␣{:?}”,v2);
}

fn display(v:Vec<i32>){
    println!(”inside␣display␣{:?}”,v);
}
```

Fig. 1. An example of ownership move. The program cannot be compiled, since the ownership of v2 is moved to display and v2 is invalidated after the move.

```
1  fn main() {
2       let  s1 = String::from(”hello”);
3  
4       let  len = calculate_length(&s1);
5  
6       println!(”The␣length␣of␣’{}’␣is␣{}.”,      s1,  len);
7  }
8  
9  fn calculate_length(s:    &String) −> usize {
10      s.len()
11 }
```

Fig. 2. An example of ownership borrowing. The program can be compiled.

its lifetime and disallow concurrent mutability and aliasing, preventing a number of serious memory and concurrency issues. For example, in Figure 2, variable s1 is immutably borrowed when calling calculate_length() at line 4. Although the lexical scope of s1 does not end until the end of function main() at line 7, Rust compiler does not overlap it with the its mutable borrow at line 4.

Lifetime Annotation. In Rust, programmers can use an apostrophe and an annotation name to explicitly annotate a variable's lifetime. The lifetime relationship between parameters and the return value can be specified at the function declaration sites using lifetime annotations, and the lifetime requirement between a struct object and its reference fields can be described at the struct definition sites using lifetime annotations. When a function or struct is being checked, the compiler will indicate errors when it detects safety-rule violations based on the lifespan annotations of the function or struct. The compiler examines if the actual parameters satisfy the corresponding annotations when invoking a function. To lessen the overhead of annotations, Rust supports lifetime elision, and the compiler automatically infers elided annotations during safety checks.

Safe and Unsafe Rust. All the above discussion refers to secure Rust code. However, Rust enables programmers to perform unsafe operations (such as pointer manipulations and calling an unsafe function) by allowing them to use the "unsafe" keyword to get around some safety checks. The C programming language is analogous to unsafe code. For Rust, a function can also be utilised as a safe function while yet containing unsafe code internally and offering a safe API externally. Since safe code must rigorously adhere to Rust's safety mechanism and is utilised considerably more frequently than unsafe code in Rust programs [21], our focus in this study is to particularly identify unsafe source code patterns in Rust.

B. Rust Compiler

The main conduit for information exchange between programmers and Rust's safety system is the Rust compiler. It compares itself to the aforementioned safety regulations and, if seeing a rule breach, reports an error. An error message typically consists of three parts: (1) the safety rule that was broken and the associated error code; (2) the lines of code or programme tokens that broke the rule; and (3) some explanations of the violation. Sometimes error messages entail information on how to resolve a problem. Additionally, programmers can run rustc and get general explanations for code errors.

However, Rust's safety regulations are convoluted [24] and some are irrational [21]. Compiler error messages might sometimes be vague or even provide false information [47]. Therefore, Rust programmers may not be able to diagnose and correct safety-rule violations from compiler error messages. Moreover, the Rust compiler requires 2^n compilations for a library crate consisting of n blocks. That means, a compilation of the Rust crates takes some minutes to maximize the safety guarantees. In this study, we explore alternative time efficient and effective neural approaches to guarantee safeness in Rust source code.

C. Inlay Type Hints.

Rust-analyzer provides inlay type hints for variables and method chaining. Inlay hints enclose additional intra-text-information appearing in code that adds contextual and type data, making the code easier to read and navigate. In particular, they are special markers that appear in the editor and provide additional information about the code, such as the names of the parameters that a called method expects, annotations, method parameters, usages etc. Figure 3 illustrates an example of Rust source code with inlay hits inserted. Inlay hints can be helpful in improving code readability and understanding, especially when working with complex code. They are configurable and can be customized to enable or disable specific kinds of hints according the user's preferences.

In this study, we investigate the contribution of inlay hints on identifying safeness in Rust code. When collecting the Rust source dataset, we enable different kinds of inlay hints:

- *AdjustmentHints* in Rust refers to coercing a value to a different type of value. It represents transforming values by following a number of Adjust steps in order. The AdjustmentHints enum in Rust's ide crate provides hints for how to adjust a value to a different type of value.

```
// original
fn main(c: i32) {
    let a = ”abc”;
    let b = 3;
    let d: usize = c;
    println!(”Hello,␣world!”);
}

// inlay hints inserted
fn main(c: i32) {
    let a: &str = ”abc”;
    let b: i32 = 3;
    let d: usize = c;
    println!(”Hello,␣world!”);
}
```

Fig. 3. An example with inlay hints in Rust.

- *AdjustmentHintsMode* is an enum in the ide crate of the Rust programming language. It is used to specify the mode of adjustment hints for a text edit operation. The AdjustmentHintsMode enum has two possible values: Line and Character. Line mode adjusts the text edit to ensure that it starts and ends on a line boundary, while Character mode adjusts the text edit to ensure that it starts and ends on a character boundary. Note that AdjustmentHints is part of the ide crate, which provides ”ide-centric” APIs for the Rust Analyzer. It generally operates with files and text ranges, and returns results as Strings.
- *ClosureReturnTypeHints* is a configuration option in the Rust programming language's Rust Analyzer language server. It is used to control whether the return type of a closure should be displayed as a hint in the editor. It has three possible values: a) Always: Always show the return type of closures as a hint, b) WithBlock: Show the return type of closures as a hint only if the closure body spans multiple lines, and c) Never: Never show the return type of closures as a hint.
- *DiscriminantHints* is a feature of the Rust programming language's rust-analyzer tool that provides hints for enum discriminants. Enum discriminants are values that identify the different variants of an enum type.
- *LifetimeElisionHints* is a feature in Rust that allows the compiler to infer the lifetimes of references in function signatures in certain cases, without requiring the programmer to explicitly specify them. The Rust compiler uses three rules to infer the lifetimes of references in function signatures: a) each parameter that is a reference gets its own lifetime parameter, b) if there is exactly one input lifetime parameter, that lifetime is assigned to all output lifetime parameters and c) if there are multiple input lifetime parameters, but one of them is &self or &mut self, the lifetime of self is assigned to all elided output lifetimes.

IV. Problem Setup

A. Task Definition

Given a pre-trained programming language model (PPLM) $\mathcal{L}$, we want to fine-tune it on a collection of Rust source codes crawled by Rust project repositories. Since language models have limitations of accepting long sequences [1] and Rust does not have a standard for the number of lines of code, we decompose Rust code into $\mathcal{N}$ code blocks $\{c_i\}_{i=1}^{N}$. Each Rust block is mapped to a label space $\mathcal{Y}$, where $y_i \in \{safe, unsafet\}$. The task requires one to predict for each Rust block c_i the unique involved class label $y_i \in \mathcal{Y}$, i.e., a binary classification task.

B. Evaluation Data

To prepare labelled Rust code for the binary classification task, we extract the datasets from two sources: (a) Rust project repositories, and (b) their dependent Rust crates automatically pulled while compiling these projects. Both sources are up to date at the time of our experiments and can be updated regularly in the future.

For each of these Rust projects, the unsafe code can be extracted in two methods, one is `cargo-geiger`, the other one is `tree-grepper`. `Cargo-geiger` is a tool that detects usage of unsafe Rust in a Rust crate and its dependencies. On the other hand, `tree-grepper` is a tool that works like grep, but uses `tree-sitter` to search for structure instead of string. In particular, `tree-grepper` searches by looking at code structure instead of strings, which lets you match on whatever structures a language defines without worrying about the exact syntax. The former is recommended by the *rust-secure-code* organisation, while the latter has a more efficient code parser. Moreover, `tree-grepper` is fault-tolerant in the sense that it can extract code even if it does not compile correctly. We compared the results of both on the compiled projects (which is the case for our selection criteria), and in fact they lead to the same results. So in the remaining, we will use `tree-grepper`.

In this work, we focus on safety of Rust blocks. Despite for the function-level unsafe code is required only a query to extract the source code, for the block-level the extraction process is more complicated and time-consuming. In particular, in order to extract unsafe blocks from functions, `tree-grepper` requires n queries, where n is the amount of blocks in a library crate, to check if unsafe keywords are entailed within the smallest amounts of code that the Rust compiler considers at a time.

Note that for each crate library, there are multiple versions in the cargo registry through the dependencies while building multiple Rust projects; therefore, we always select the latest version of each named crates. We also use the hash of the function content to remove duplicate Rust source code from the collected corpora.

[1] The original models' implementations truncate longer sequences automatically. However, such an approach can lead to thread safety issues since Rust code does not have a standard for the position of undefined behaviour.

When pre-processing, we remove unsafe keywords from Rust blocks. The collected Rust dataset entails 25,188 safe and 12,899 unsafe Rust blocks. That is the ratio between safe and unsafe Rust code is 2:1. Table I overviews the statistics of the data used for training, validation, and testing.

TABLE I
OVERVIEW OF THE EXTRACTED RUST DATASET USED TO CONDUCT EXPERIMENTS ACROSS DIFFERENT MODELS AND REFORMULATED NLP TASKS.

Subset	# Safe	# Unsafe
Train	20,150	10,319
Validation	2,519	1,290
Test	2,519	1,290

C. Learning Tasks

We experiment with three different tasks: a) standard fine-tuning, b) multitask fine-tuning with mask language model (MLM), and c) prompt-based fine-tuning. The standard fine-tuning objective is to learn the discrimination between safe and unsafe Rust code. While standard fine-tuning has been a popular approach in NLP, achieving remarkable performance for various tasks, it does not consider explicit information from tokens within the source code. That means source code tokens will never have any gradient signal. In this work we address this limitation and experiment with two other tasks that explicitly consider source code information over the optimization process; those are multitasking with MLM and prompt-based fine-tuning. For the classification setting, when experimenting with multitasking and prompt-based, we reformulate the problem to text-to-text format. We formally define each of the above tasks as follows:

a) ***Standard fine-tuning:*** Given a PPLM $\mathcal{L}$, we first transform a Rust source code c to a sequence of tokens $\bar{c}$ in a structure form as illustrated in Figure 4, where $[CLS]$ and $[SEP]$ are special tokens attached at the beginning and the end of source code. The model receives the sequence of tokens $\bar{c}$ and maps the source code to a sequence of hidden states $\{h_k \in \mathbb{R}\}$. For the classification task with a label space $\mathcal{Y}$, we train a task-specific head, $softmax(W_o h_{[CLS]})$, by maximizing the log-probability of the correct label, where $h_{[CLS]}$ is the hidden vector of $[CLS]$, and $W_o \in \mathbb{R}^{|\mathcal{Y}|n \times d}$ is a set of randomly initialized parameters introduced at the start of fine-tuning. We fine-tune the model $\mathcal{L}$ winth binary cross-entropy (BCE) loss.

```
[CLS]
fn main() {
 let  mut x = 10;
     let  ptr_x  = &mut x as *mut i32;

     {
          *ptr_x  = 20;
     }
}
[SEP]
```

Fig. 4. Transformed Rust source code input $\bar{c}$ for the standard fine-tuning task.

```
[CLS]
fn main() {
     let  [MASK] x = 10;
     let  ptr_x  = [MASK] x as *mut i32;

     {
          *[MASK] = 20;
     }
}
[SEP]
```

Fig. 5. Transformed Rust source code input $\bar{c}$ for the multitask fine-tuning with mask language model (MLM) objective.

b) ***Multitask Fine-Tuning with Mask Language Model:*** We fine-tune a PPLM model $\mathcal{L}$ with two objectives. The first one is the mask language model (MLM) objective to understand particular source patterns in Rust. We first transform a source code c to a sequence of tokens $\bar{c}$ and then we dynamically [2] mask 15% of tokens within the source code [48]. Given a supervised sample (c, y) , the model receives a MLM input as illustrated in Figure 5, where $[MASK]$ is a special token used to overwrite source code tokens within the original source code. The model maps each of the $[MASK]$ tokens to a sequence of logits $\mathcal{L}(\bar{c}) \in \mathbb{R}^{|\mathcal{V}|}$, where $\mathcal{V}$ is the vocabulary of $\mathcal{L}$. The training process casts to a high-dimensional multi-class classification problem by predicting the original token t corresponding to $[MASK]$ with cross-entropy (CE) loss as follows:

$$Loss = CE(p(t^{\mathcal{L}}|\bar{c}), t) \tag{1}$$

, where $p(t^{\mathcal{L}}|\bar{c})$ softmax over t calculated as

$$p(t^{\mathcal{L}}|\bar{c}) = \frac{exp([\![\mathcal{L}(\bar{c})]\!]_t)}{\sum_{t'\in\mathcal{V}} exp([\![\mathcal{L}(\bar{u})]\!]'_y)} \tag{2}$$

The second objective is to predict for the Rust source code c the involved class label $y \in \mathcal{Y}$ (for a detailed description of standard fine-tuning please refer to IV-C0a). We train a model $\mathcal{L}$ jointly on both objectives. The combined loss is a linear weighted sum of loss functions of the two objectives. The assignment of weights is an open research question. Here, we set the weights empirically, based on the minimum loss function values when fine-tuning the model on the two objectives separately.

c) ***Prompt-based Fine-tuning:*** A PPLM model $\mathcal{L}$ is tasked with ”auto-completing” natural language prompts [42]. In particular, for each source code c let $\mathcal{T}(c)$ be a MLM input with one $[MASK]$ token as illustrated in Figure 6, where natural language corresponds to a task specific template and $[MASK] \in \mathcal{M}(y)$. Let $\mathcal{M} : \mathcal{Y} \rightarrow \mathcal{V}^{|\mathcal{Y}|}$ be a one-to-one mapping from the task label space $\mathcal{Y}$ to individual words in the vocabulary $\mathcal{V}$ of $\mathcal{L}$. The model $\mathcal{L}$ receives a template $\mathcal{T}(u)$ and maps the $[MASK]$ token to a sequence

[2] Different tokens are randomly masked in each epoch.

```
[CLS]
fn main() {
 let  mut x = 10;
     let  ptr_x  = &mut x as *mut i32;

     {
          *ptr_x  = 20;
     }
}
It  is  [MASK]
[SEP]
```

Fig. 6. Transformed Rust source code input $\overline{c}$ for the prompt-based fine-tuning, where natural language corresponds to a task specific template and $[MASK] \in \mathcal{M}(y)$.

of logits $\mathcal{L}(\mathcal{T}(u)) \in \mathbb{R}^{|\mathcal{V}|}$. We cast the problem of predicting the probability of $y \in \mathcal{Y}$ as a MLM task:

$$p(y \mid c) = p([MASK] = \mathcal{M}(y) \mid \mathcal{T}(c)). \quad (3)$$

For a set of instances $\{c, y\}$, $\mathcal{L}$ is fine-tuned to minimize the cross-entropy loss.

D. Pre-trained Programming Language Models

In this study, we experiment with both moderately- and large-sized pre-trained programming language models (PPLM) that previously exhibited state-of-the-art performance in solving challenging software-engendering tasks.

*a) **Baselines**:* We compare the performance of the state-of-the-art models with the following baselines:

- **Random Rate:** We use weighted guessing as a baseline classifier where accuracy is guessed at the weighted percentages of each class.
- **BiLSTM for Code Classification:** We use a shallow BiLSTM [49], i.e., one-layer BiLSTM followed by a fully connected layer and a binary classifier for Rust code classification. The model receives as input code features extracted by considering three types of tokens, i.e., identifiers, operators, and brackets. At the prediction time, we apply softmax for binary classification. We train the model with binary cross-entropy (BCE) loss.
- **CNN for Code Classification:** We use a Convolutional Neural Network (CNN) architecture [50] for code representation using code embeddings. Here, we use the extracted features of BiLSTM to represent code tokens into code embedding. Then, a convolutional layer with four filters, i.e., a filter size of $[2, 3, 4, 5]$, captures code token interactions within n-grams $n \in [2, 5]$. The convolutional layer is followed by a max-pooling layer with a fully connected layer, projecting the input into a two-dimensional vector. At the inference time, we use softmax and train the model with BCE loss.

*b) **Pre-trained Programming Language Models**:* We fine-tune PPLM to the objective of recognising undefined behaviour in idiomatic Rust that is not defined by the language specification and can lead to memory unsafety. In particular, we experiment with the following encoder and encoder-decoder state-of-the-art transformers, respectively.

- **CodeBERT** [14] is a multi-programming-lingual model pre-trained on natural language (NL) - programming language (PL) pairs in 6 programming languages, yet Rust is not included. It learns general-purpose representations that support downstream NL-PL applications. CodeBERT is developed with a Transformer-based neural architecture, and is trained with a hybrid objective function that incorporates the pre-training task of replaced token detection, which is to detect plausible alternatives sampled from generators. This enables the utilization of both bimodal data of NL-PL pairs and unimodal data (i.e., PL), where the former provides input tokens for model training while the latter helps to learn better generators. In this study, we fine-tuned CodeBERT on unimodal data, i.e., we used the Rust code disregarding directives and comments over the learning process.
- **codeT5** [15] is a pre-trained encoder-decoder model for programming languages that is designed to better leverage the code semantics conveyed from the developer-assigned identifiers. It is pre-trained on 8.35 million functions in eight programming languages, including Python, Java, JavaScript, PHP, Ruby, Go, C, and C#, using the masked span prediction objective. CodeT5 employs a unified framework to support both code understanding and generation tasks and allows for multi-task learning. It achieves state-of-the-art results on 14 sub-tasks in a code intelligence benchmark called CodeXGLUE. For the classification setting, when experiment with codeT5, we have to reformulate all the learning tasks to text-to-text format.

E. Evaluation Metrics

For evaluating the learning tasks and all investigated neural architectures, we report accuracy and micro $F1$ score for each class y. We also calculate the macro-averaged accuracy and $F1$ score. We chose macro-averaged since we are interested in valuing the minority class, i.e., Rust source code that uses unsafe features. For both metrics, higher values denote better performance.

F. Experimental Settings

We implemented all the models on PyTorch. For codeBERT and codeT5 we used Hugging Face's model sharing platform to instantiate them from the corresponding pre-trained model configurations. We used a grid search optimization technique to optimize the parameters. For consistency, we used the same experimental settings for all models. We first trained and fine-tuned all models by performing a twenty-times grid search over their parameter pool. We empirically experimented with learning rate (lr): $lr \in \{0.00001, 0.00002, 0.00005, 0.0001, 0.0002\}$, batch size ($bs$): $bs \in \{16, 32, 64, 128\}$ and optimization (O): $O \in \{AdamW, Adam\}$. After the fine-tuning process, we trained again all the models for 50 epochs with 4 epochs early stopping, three times. We reported the average performance on the test set for all experiments. Model checkpoints were selected based on the minimum validation loss. Experiments were conducted on four GPUs, Nvidia V-100.

REPRODUCABILITY

We release the data and implementations on the anonymous repository https://anonymous.4open.science/r/ase23-3554/.

V. EXPERIMENT RESULTS

A. Quantitative Results

Table II summarizes the experimental results on identifying safety in Rust blocks with contextual information as elicited by inlay hints across different models and reformulated NLP tasks. All fine-tuning and learning strategies yielded significantly better performance than random weighted guessing. Despite the ratio of safe and unsafe code snippets was 2:1, we did not notice a bias towards the majority class (i.e., safe). On the other hand, we noticed a trade-off between precision and recall when comparing the fine-grained performance (see Safe and Unsafe columns in Table II). However, that is a common issue in classification problems. To this end, our decision for the best performance model was based on the weighted precision, recall and $F1$ score.

Both baselines, i.e., Bi-LSTM and CNN, achieved a fair performance compared to state-of-the-art (SOTA) approaches on the safety Rust classification task. Yet both baselines underperformed all the SOTA models and the reformulated NLP tasks in Table II. In particular, Bi-LSTM exhibited the lowest performance. We attribute this to the shallow architecture of BiLSTM (i.e., a hidden layer followed by a fully connected one). On the other hand, CNN achieved an increased f_1 score for the unsafe class of 0.80 compared to 0.68 for Bi-LSTM. We assume this is because the convolution layer filters can capture both local (i.e.,2- grams) and global (i.e., 5-grams) source code interactions across the sequence and hence the structure of source code.

codeBERT, a multilayer bi-directional transformer encoder, on the fine-tuning setting, was one of two approaches that achieved the best weighted $F1$ score for classifying safe versus unsafe block-level source code. Forcing codeBERT to extract signal from the Rust source code during the optimization process (i.e., multitask with MLM in Table II) or leveraging knowledge stored in the pre-trained model (i.e., prompt in Table II) did not improve the performance. In particular, for both multitasking and prompt-based settings, the weighted f_1 score exhibited a decrease of 1% as compared to fine-tuning. Having said that, codeBERT on multitask fine-tuning with MLM objective achieved the best performance for the unsafe class. For multitasking with MLM, this setting may be more appropriate when masking is targeted rather than random. For prompt-based learning, more targeted prompts may benefit the performance. We discuss possible directions towards this field in the next section.

When conducting experiments on codeT5, we did not fine-tune it on prompt-based learning since codeT5 treats the classification as a text-to-text problem, i.e., equivalent to prompt-based fine-tuning applied on codeBERT. Fine-tuning codeT5 yielded the same weighted f_1 score with fine-tuned codeBERT. However, codeT5 is pre-trained on 8.35M functions in 8 programming languages and has 220M parameters. On the other side, codeBERT is trained on 2.1M bimodal data points and 6.4M unimodal codes across six programming languages and has 125M parameters. When training codeT5 jointly with MLM, we obtained similar results with multitasking codeBERT.

Overall, the experiments showed that codeBERT with standard fine-tuning was the most efficient and effective approach compared to the rest approaches in Table II.

B. Contribution of Inlay Hints in Learning Process

We investigated how Rust representations encapsulated by contextual information elicited by inlay hints could facilitate the learning process and hence the classification of Rust blocks. To this goal, we trained all the models for different settings as present in Table II by deactivating all the types of inlay hints in pre-processing. Table III summarizes the experimental results when inlay hints are not integrated within the Rust source code across different models and reformulated NLP tasks. The experimenters showed that all the approaches underperformed their equivalent when trained on Rust source code with conveying information from inlay hints. Overall, the performance is analogous to that one in Table II. In particular, codeBERT with standard fine-tuning was the most efficient and effective approach for classifying Rust code.

Despite the experiments showing that inlay hints facilitated the performance in the learning process, it is an open research question of how inlay hints can help Rust developers identify Rust safety issues. Indeed, inlay hints can provide additional information about code, such as variable types and method chaining, to help with code comprehension and can help developers identify potential issues. However, further research is needed to determine the specific contribution of inlay hints to identifying Rust safety issues. Moreover, in this study, we investigated the contribution of different types of inlay hints. From the experiments, it is not clear how individual types contribute to the representation process. In the future, we aim to conduct comprehensive experiments by learning Rust representations with one-type inlay hint each time. However, this requires different versions of the Rust corpus activating one type of inlay hints over the collection. Currently, such dataset versions are not available.

C. Error Analysis on the Misclassified Unsafe Rust Code

We conducted an error analysis on the misclassified unsafe Rust blocks as inferred by the best performance model, i.e., codeBERT with standard fine-tuning, to detect Rust source code patterns that the model struggled to classify. For the analysis, we used a subset of 78 unsafe blocks. To this goal, we utilized a data-driven methodology which simultaneously

TABLE II

Experimental results on identifying safety in Rust blocks encapsulated with contextual information as elicited by inlay hints across different neural models and reformulated NLP tasks. The Safe and Unsafe columns report the fine-grained performance. The Weighted column reports the overall weighted average performance in the test subset. Best performances are marked in bold.

		Safe			**Unsafe**			**Weighted**		
Model	**Task**	Precision	Recall	F1	Precision	Recall	F1	Precision	Recall	F1
Random Rate	-	0.40	-	-	0.10	-	-	0.50	-	-
Bi-LSTM	Train	0.85	0.89	0.87	0.87	0.56	0.72	0.86	0.73	0.80
CNN	Train	0.88	0.90	0.89	0.86	0.75	0.80	0.88	0.83	0.86
codeBERT	Fine-tuning	**0.94**	0.97	**0.96**	0.94	0.88	0.91	**0.94**	**0.94**	**0.94**
codeBERT	Multitask	**0.94**	0.95	0.95	0.92	**0.91**	**0.92**	0.93	0.93	0.93
codeBERT	Prompt	**0.94**	0.95	0.94	0.93	0.90	0.91	0.93	0.93	0.93
codeT5	Fine-tuning	0.93	**0.98**	**0.96**	**0.97**	0.86	0.91	**0.94**	**0.94**	**0.94**
codeT5	Multitask	**0.94**	0.97	0.95	0.96	0.87	0.91	0.93	0.93	0.93

TABLE III

Experimental results on identifying safety in Rust blocks without integrating inlay hints information across different neural models and reformulated NLP tasks. The Safe and Unsafe columns report the fine-grained performance. The Weighted column reports the overall weighted average performance in the test subset. Best performances are marked in bold.

		Safe			**Unsafe**			**Weighted**		
Model	**Task**	Precision	Recall	F1	Precision	Recall	F1	Precision	Recall	F1
Random Rate	-	0.40	-	-	0.10	-	-	0.50	-	-
Bi-LSTM	Train	0.81	0.85	0.83	0.79	0.58	0.69	0.80	0.72	0.76
CNN	Train	0.81	0.88	0.85	0.83	0.73	0.78	0.82	0.81	0.82
codeBERT	Fine-tuning	**0.90**	0.94	**0.92**	0.88	**0.90**	**0.89**	**0.91**	**0.91**	**0.91**
codeBERT	Multitask	**0.90**	0.92	0.91	0.88	0.89	**0.89**	0.90	0.90	0.90
codeBERT	Prompt	0.88	0.92	0.90	0.90	0.86	0.88	0.89	0.89	0.89
codeT5	Fine-tuning	0.88	**0.95**	**0.92**	**0.92**	0.86	**0.89**	0.90	**0.91**	**0.91**
codeT5	Multitask	0.88	0.94	0.91	0.89	0.87	0.88	0.89	**0.91**	0.90

learns feature representations and clustering assignments using a shallow neural network [51]. In particular, we used a two-layer denoising autoencoder to first transform the embedding Rust representation extracted by codeBERT into a smaller latent feature space $Z \in \mathcal{R}$ and fine-tuned it to minimize reconstruction loss. The decoder layers were then discarded and only encoder layers were used as initial mapping between the codeBERT space and the feature space Z. Then k-means clustering was performed in the feature space Z to obtain k initial centroids. Thus the unsupervised representation learned by the autoencoder naturally facilitates the learning of clustering representations. Figure 7 summarizes the overall approach.

Table IV summarizes the experimental results of the unsupervised clustering method when applied to the misclassified cases of the fine-tuned codeBERT. The selection of the centroid count k was based on the minimum loss. Here, the minimum loss is obtained when $k = 3$. Finally, we manually analysed the Rust blocks across clusters to find out prominent source patterns that characterized them. For the first cluster which amounts to 43 samples, we noticed that most of the samples were related to the use of lifetime annotations. For the second cluster which amounts to 28 samples, we observed element reference and ownership issues. However, for the third cluster, we could not reach a safe conclusion.

Overall, the analysis showed that safety rules associated with life-time annotation and borrowing from containers were more challenging to be identified by codeBERT.

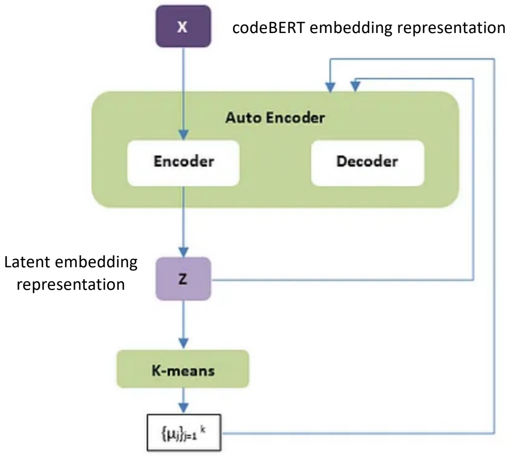
X
codeBERT embedding representation
Auto Encoder
Encoder
Decoder
Latent embedding representation
Z
K-means
$\{\mu_j\}_{j=1}^{k}$

Fig. 7. Summarization of the unsupervised clustering approach.

VI. Limitations

In this study, we investigated pre-trained programming language models that reserve rich code semantics and force them to focus on reformulated natural language processing (NLP) tasks and associated Rust code patterns. Our optimal goal is to automatize the identification of safety in Rust source code and help Rust programmers to avoid safety risks when

		# Samples				
# Clusters	Loss	C_1	C_2	C_3	C_4	C_5
k=1	0.344	78				
k=2	0.286	69	9			
k=3	0.121	43	28	7		
k=4	0.301	36	23	10	2	
k=5	0.378	30	19	15	8	6

TABLE IV
EXPERIMENTAL RESULTS OF THE UNSUPERVISED CLUSTERING METHOD APPLIED ON THE MISCLASSIFIED CASES OF FINE-TUNED CODEBERT. THE FIRST TWO COLUMNS REFER TO THE NUMBER OF CLUSTERS AND THEIR ASSOCIATED LOSS OVER THE OPTIMIZATION PROCESS. THE THIRD COLUMN LISTS THE NUMBER OF RUST SAMPLES WITHIN THE CLUSTERS.

```
#[allow(unused_unsafe)]
fn my_safe_function() {
    let mut data = vec![0u8; 1024];
    let ptr = data.as_mut_ptr();

    unsafe {
        // This is safe because we know that the pointer
        // is valid and points to a valid object,
        // since it was obtained from a vector.
        *ptr = 42;
    }
}
```

Fig. 8. Unsafe Rust code using comments to explain why the code is unsafe and what precautions need to be taken when using it.

developing applications. Even though the experiments showed that standard fine-tuning is the most effective approach for classifying Rust, we cannot conclude that reformulating NLP tasks for classification do not benefit performance as many challenges have to be addressed.

The denoising masked language modelling (MLM) objective has been extensively utilised to pre-train large language models in NLP. However, in this study, the experiments showed that multitasking fine-tuning with MLM did not improve performance. We suppose this is because we randomly masked 15% within the sequences of source code tokens rather than applied specific masking. More elaborated masking approaches might improve the Rust classification performance. For example, masking particular identifiers that violate safety rules in Rust. However, it is an open research question on how to identify chunks in source code violating the rules as such a tool is not currently available. An alternative approach to implicitly target particular identifiers violating safety rules might be the exploitation of comments explaining why the code is unsafe. Figure 8 illustrates a Rust code with comments indicating what the unsafe code relies on for correctness. CodeBERT has been pre-trained with the objective to bridge information between natural language documentation and corresponding code pairs and might be effective for capturing such associations. However, crawling high-quality Rust code with comments from repositories is another challenge. It requires further human processing that can be expensive.

For prompt-based fine-tuning, the experiments showed that the performance was similar to the one of standard fine-tuning, yet slightly lower. Indeed, finding ways to reformulate tasks as cloze questions that make the best use of knowledge stored in pre-trained models can be difficult [52]. Exploring ways to identify templates and label words might lead to better performance compared to hand-picked ones [43].

Finally, despite codeT5 being a larger model than codeBERT, the former did not improve the performance for the Rust classification. In the future, it is worth investigating generative models and fine-tuning them on tasks similar to the ones used for pre-training. One possible direction would be the generation of safe Rust code given its corresponding unsafe one.

VII. CONCLUSIONS AND FUTURE WORK

Our motivation in this work is to automate the identification of safety in Rust source code blocks and hence help Rust programmers to avoid safety risks when developing applications. To this goal, we are the first to introduce reformulated NLP tasks for learning unsafe source code patterns from Rust by forcing pre-trained programming language models to obtain signal from the source code. Our experiments show the effectiveness of the proposed approaches over the baseline ones. Surprisingly, larger pre-trained programming language models and reformulated NLP tasks achieve similar, yet slightly lower, performance than the one when we fine-tune a moderate-size pre-trained model, i.e., codeBERT. We show that encapsulated contextual information, elicited by different types of inlay hints, facilitates the learning process and models to capture source code patterns that violate safety rules in Rust blocks. We use an unsupervised method for the misclassified unsafe Rust blocks as inferred by the best performance model, i.e., codeBERT, which simultaneously learns feature representations and clustering assignments. After searching for the optimal number of clusters, we manually analyse the samples within them to identify prominent patterns. The analysis shows that safety rules associated with life-time annotation and ownership borrowing from containers are the most challenging for codeBERT.

Instead of just detecting safeness in the Rust code, in the future, we aim to investigate the effectiveness of large pre-trained generative models, i.e., GPT, for replacing the unsafe Rust blocks with equivalent safe Rust ones.

REFERENCES

[1] "Blog - towards the next generation of xnu memory safety: - apple security research — security.apple.com," https://security.apple.com/blog/towards-the-next-generation-of-xnu-memory-safety/, [Accessed 28-Feb-2023].

[2] MSRC, "We need a safer systems programming language — msrc blog — microsoft security response center — msrc.microsoft.com," https://msrc.microsoft.com/blog/2019/07/we-need-a-safer-systems-programming-language/, [Accessed 28-Feb-2023].

[3] "The pixel phones may be getting a long overdue feature — zdnet.com," https://www.zdnet.com/article/the-pixel-phones-may-be-getting-a-long-overdue-feature/, [Accessed 28-Feb-2023].

[4] H. Xu, Z. Chen, M. Sun, Y. Zhou, and M. R. Lyu, "Memory-safety challenge considered solved? an in-depth study with all rust cves," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 31, no. 1, pp. 1–25, 2021.

[5] L. Szekeres, M. Payer, T. Wei, and D. Song, "Sok: Eternal war in memory," in *2013 IEEE Symposium on Security and Privacy*. IEEE, 2013, pp. 48–62.

[6] V. Astrauskas, C. Matheja, F. Poli, P. Müller, and A. J. Summers, "How do programmers use unsafe rust?" *Proceedings of the ACM on Programming Languages*, vol. 4, no. OOPSLA, pp. 1–27, 2020.

[7] Y. Bae, Y. Kim, A. Askar, J. Lim, and T. Kim, "Rudra: finding memory safety bugs in rust at the ecosystem scale," in *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, 2021, pp. 84–99.

[8] N. D. Bui, L. Jiang, and Y. Yu, "Cross-language learning for program classification using bilateral tree-based convolutional neural networks," *arXiv preprint arXiv:1710.06159*, 2017.

[9] X. Cheng, H. Wang, J. Hua, G. Xu, and Y. Sui, "Deepwukong: Statically detecting software vulnerabilities using deep graph neural network," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 30, no. 3, pp. 1–33, 2021.

[10] W. U. Ahmad, S. Chakraborty, B. Ray, and K.-W. Chang, "A transformer-based approach for source code summarization," *arXiv preprint arXiv:2005.00653*, 2020.

[11] N. D. Bui, Y. Yu, and L. Jiang, "Infercode: Self-supervised learning of code representations by predicting subtrees," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 1186–1197.

[12] X. Hu, G. Li, X. Xia, D. Lo, and Z. Jin, "Deep code comment generation," in *Proceedings of the 26th conference on program comprehension*, 2018, pp. 200–210.

[13] N. D. Bui, Y. Yu, and L. Jiang, "Sar: learning cross-language api mappings with little knowledge," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019, pp. 796–806.

[14] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang *et al.*, "Codebert: A pre-trained model for programming and natural languages," *arXiv preprint arXiv:2002.08155*, 2020.

[15] Y. Wang, W. Wang, S. Joty, and S. C. Hoi, "Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation," *arXiv preprint arXiv:2109.00859*, 2021.

[16] X. Lu, Z. Zhang, and H. Xu, "Safe rust code recommendation based on siamese graph neural network," in *2022 IEEE 27th Pacific Rim International Symposium on Dependable Computing (PRDC)*. IEEE, 2022, pp. 1–11.

[17] S. Park, X. Cheng, and T. Kim, "Unsafe's betrayal: Abusing unsafe rust in binary reverse engineering toward finding memory-safety bugs via machine learning," *arXiv preprint arXiv:2211.00111*, 2022.

[18] D. Tam, R. R. Menon, M. Bansal, S. Srivastava, and C. Raffel, "Improving and simplifying pattern exploiting training," *arXiv preprint arXiv:2103.11955*, 2021.

[19] A. N. Evans, B. Campbell, and M. L. Soffa, "Is rust used safely by software developers?" in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, 2020, pp. 246–257.

[20] P. Kirth, M. Dickerson, S. Crane, P. Larsen, A. Dabrowski, D. Gens, Y. Na, S. Volckaert, and M. Franz, "Pkru-safe: automatically locking down the heap between safe and unsafe languages," in *Proceedings of the Seventeenth European Conference on Computer Systems*, 2022, pp. 132–148.

[21] B. Qin, Y. Chen, Z. Yu, L. Song, and Y. Zhang, "Understanding memory and thread safety practices and issues in real-world rust programs," in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2020, pp. 763–779.

[22] S. Zhu, Z. Zhang, B. Qin, A. Xiong, and L. Song, "Learning and programming challenges of rust: A mixed-methods study," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 1269–1281.

[23] A. Zeng and W. Crichton, "Identifying barriers to adoption for rust through online discourse," *arXiv preprint arXiv:1901.01001*, 2019.

[24] W. Crichton, "The usability of ownership," *arXiv preprint arXiv:2011.06171*, 2020.

[25] P. Abtahi and G. Dietz, "Learning rust: How experienced programmers leverage resources to learn a new programming language," in *Extended Abstracts of the 2020 CHI Conference on Human Factors in Computing Systems*, 2020, pp. 1–8.

[26] K. R. Fulton, A. Chan, D. Votipka, M. Hicks, and M. L. Mazurek, "Benefits and drawbacks of adopting a secure programming language: rust as a case study," in *Symposium on Usable Privacy and Security*, 2021.

[27] A. Kanade, P. Maniatis, G. Balakrishnan, and K. Shi, "Learning and evaluating contextual embedding of source code," in *International conference on machine learning*. PMLR, 2020, pp. 5110–5121.

[28] K. Clark, M.-T. Luong, Q. V. Le, and C. D. Manning, "Electra: Pre-training text encoders as discriminators rather than generators," *arXiv preprint arXiv:2003.10555*, 2020.

[29] A. Svyatkovskiy, S. K. Deng, S. Fu, and N. Sundaresan, "Intellicode compose: Code generation using transformer," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020, pp. 1433–1443.

[30] F. Liu, G. Li, Y. Zhao, and Z. Jin, "Multi-task learning based pre-trained language model for code completion," in *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, 2020, pp. 473–485.

[31] L. Dong, N. Yang, W. Wang, F. Wei, X. Liu, Y. Wang, J. Gao, M. Zhou, and H.-W. Hon, "Unified language model pre-training for natural language understanding and generation," *Advances in neural information processing systems*, vol. 32, 2019.

[32] M.-A. Lachaux, B. Roziere, L. Chanussot, and G. Lample, "Unsupervised translation of programming languages," *arXiv preprint arXiv:2006.03511*, 2020.

[33] C. B. Clement, D. Drain, J. Timcheck, A. Svyatkovskiy, and N. Sundaresan, "Pymt5: multi-mode translation of natural language and python code with transformers," *arXiv preprint arXiv:2010.03150*, 2020.

[34] A. Elnaggar, W. Ding, L. Jones, T. Gibbs, T. Feher, C. Angerer, S. Severini, F. Matthes, and B. Rost, "Codetrans: Towards cracking the language of silicon's code through self-supervised deep learning and high performance computing," *arXiv preprint arXiv:2104.02443*, 2021.

[35] W. U. Ahmad, S. Chakraborty, B. Ray, and K.-W. Chang, "Unified pre-training for program understanding and generation," *arXiv preprint arXiv:2103.06333*, 2021.

[36] D. Guo, S. Ren, S. Lu, Z. Feng, D. Tang, S. Liu, L. Zhou, N. Duan, A. Svyatkovskiy, S. Fu *et al.*, "Graphcodebert: Pre-training code representations with data flow," *arXiv preprint arXiv:2009.08366*, 2020.

[37] B. Roziere, M.-A. Lachaux, M. Szafraniec, and G. Lample, "Dobf: A deobfuscation pre-training objective for programming languages," *arXiv preprint arXiv:2102.07492*, 2021.

[38] D. Zügner, T. Kirschstein, M. Catasta, J. Leskovec, and S. Günnemann, "Language-agnostic representation learning of source code from structure and context," *arXiv preprint arXiv:2103.11318*, 2021.

[39] W. L. Taylor, ""cloze procedure": A new tool for measuring readability," *Journalism quarterly*, vol. 30, no. 4, pp. 415–433, 1953.

[40] D. Yang, Z. Zhang, and H. Zhao, "Learning better masking for better language model pre-training," *arXiv preprint arXiv:2208.10806*, 2022.

[41] T. Schick and H. Schütze, "Exploiting cloze questions for few shot text classification and natural language inference," *arXiv preprint arXiv:2001.07676*, 2020.

[42] P. Liu, W. Yuan, J. Fu, Z. Jiang, H. Hayashi, and G. Neubig, "Pre-train, prompt, and predict: A systematic survey of prompting methods in natural language processing," *ACM Computing Surveys*, vol. 55, no. 9, pp. 1–35, 2023.

[43] T. Gao, A. Fisch, and D. Chen, "Making pre-trained language models better few-shot learners," *arXiv preprint arXiv:2012.15723*, 2020.

[44] S. Wang, H. Fang, M. Khabsa, H. Mao, and H. Ma, "Entailment as few-shot learner," *arXiv preprint arXiv:2104.14690*, 2021.

[45] Z. Li, S. Li, and G. Zhou, "Pre-trained token-replaced detection model as few-shot learner," *arXiv preprint arXiv:2203.03235*, 2022.

[46] T. Schick and H. Schütze, "It's not just size that matters: Small language models are also few-shot learners," *arXiv preprint arXiv:2009.07118*, 2020.

[47] L. A. Goodman, "Snowball sampling," *The annals of mathematical statistics*, pp. 148–170, 1961.

[48] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," *arXiv preprint arXiv:1810.04805*, 2018.

[49] M. Schuster and K. K. Paliwal, "Bidirectional recurrent neural networks," *IEEE transactions on Signal Processing*, vol. 45, no. 11, pp. 2673–2681, 1997.

[50] Y. Chen, “Convolutional neural network for sentence classification,” Master’s thesis, University of Waterloo, 2015.

[51] J. Xie, R. Girshick, and A. Farhadi, “Unsupervised deep embedding for clustering analysis,” in *International conference on machine learning*. PMLR, 2016, pp. 478–487.

[52] T. Schick and H. Schütze, “It’s not just size that matters: Small language models are also few-shot learners,” in *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*. Online: Association for Computational Linguistics, Jun. 2021, pp. 2339–2352. [Online]. Available: https://aclanthology.org/2021.naacl-main.185