

SPDZCoder: Combining Expert Knowledge with LLMs for Generating Privacy-Computing Code

XIAONING DONG, Tsinghua University, China

PEILIN XIN, Tsinghua University, China

LING HUANG, AHI Fintech Limited, China

YANLIN WANG, Sun Yat-sen University, China

JIA LI^{*✉}, College of AI, Tsinghua University, China

WEI XU^{*✉}, Tsinghua University, China and Shanghai Qi Zhi Institute, China

Privacy computing receives increasing attention but writing privacy computing code remains challenging for developers due to limited library functions, necessitating function implementation from scratch, and data-obliviousness requirement, contradicting intuitive thinking and usual practices of programmers. Automating the generation of privacy computing code with large language models (LLMs) can streamline development effort and lower the barrier to using privacy computing frameworks since LLMs exhibit strong capabilities in coding tasks. However, existing LLMs still encounter challenges in code translation for privacy-preserving computation—such as translating Python to MP-SPDZ—due to the scarcity of MP-SPDZ data required for effective pre-training or fine-tuning. Moreover, the lack of benchmark further hinders the evaluation of translation quality. To address the limitations, this work proposes SPDZCoder, a training-free framework that combines LLMs with expert knowledge for generating privacy computing code. Specifically, SPDZCoder employs a rigorous procedure for collecting high-quality expert knowledge to represent the semantic-expressing differences between Python and MP-SPDZ, and derives transformation rules for translating Python to MP-SPDZ based on these knowledge. Then, SPDZCoder progressively converts Python code into MP-SPDZ code using transformation rules in a three-stage pipeline. To evaluate SPDZCoder, we manually constructed a benchmark dataset, SPDZEval, which comprises six data splits, each representing a distinct class of challenging tasks in MP-SPDZ implementation. Extensive experiments demonstrate that SPDZCoder achieves superior performance, significantly surpassing baselines in pass@1 and pass@2. Specifically, SPDZCoder with GPT-4 as its backbone attains overall correctness values of 85.94% and 92.01% in pass@1 and pass@2, respectively, whereas the best-performing baseline achieves only 63.58% and 76.36%, respectively.

CCS Concepts: • **Computing methodologies** → **Artificial intelligence**; • **Software and its engineering**;

Additional Key Words and Phrases: Privacy-preserving Code Translation, Large Language Models, MP-SPDZ

^{*}Both authors jointly advised this work.

Authors' Contact Information: Xiaoning Dong, dongxn20@mails.tsinghua.edu.cn, Tsinghua University, Beijing, China; Peilin Xin, xinpl21@mails.tsinghua.edu.cn, Tsinghua University, Beijing, China; Ling Huang, linghuang@fintec.ai, AHI Fintech Limited, Beijing, China; Yanlin Wang, wangyin36@mail.sysu.edu.cn, Sun Yat-sen University, Zhuhai, Guang Zhou Province, China; Jia Li, jia_li@mail.tsinghua.edu.cn, College of AI, Tsinghua University, Beijing, China; Wei Xu, weixu@tsinghua.edu.cn, Tsinghua University, Beijing, China and Shanghai Qi Zhi Institute, Shanghai, China.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

© 2026 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ACM 1557-7392/2026/9-ART

<https://doi.org/10.1145/3845985>

1 Introduction

Multi-party computation (MPC) [20, 67] enables multiple mutually distrustful parties to jointly compute a function over their private inputs while revealing only the computation output. As regulatory frameworks such as GDPR [17] and HIPAA [56] impose stringent constraints on data sharing, MPC has become a foundational technique for privacy-preserving data collaboration, particular in domains such as finance [6] and healthcare [37, 43].

MPC is commonly provided through software frameworks that support protocol execution and arithmetic operations. Among existing open-sourced MPC frameworks, though most of them remain prototype-oriented and provide mainly low-level primitives that require users to compose high-level functionality on their own, *Multi-Protocol SPDZ (MP-SPDZ)* [35] stands out as a particularly comprehensive and widely adopted platform since it provides specialized data types and a modest yet practical set of essential data structures, library functions and classes, and has been leveraged in academic studies and prototype systems for secure machine learning and private data analytics.

Despite the growing demand for MPC applications and frameworks such as MP-SPDZ, their adoption remains far below expectations. One reason is the shortage of skilled developers for privacy-preserving code, as most developers rely on general programming languages (GPL) such as Python. Moreover, the intricate features of MPC increase the cost to training programmers, *e.g.*, the data-oblivious requirement (discussed in Sec. 2.1) contradicts usual programming practices of developers. Thus, this work aims to explore a question: *can GPL code be automatically translated into MP-SPDZ code*, enabling developers and researchers to reuse plaintext implementations to develop privacy-preserving ones, thereby improving the development productivity?

Recently, large language models (LLMs) [3, 13, 19, 32, 49, 51, 53–55] have made remarkable progress. Building on their success, code LLMs [12, 23, 27, 48] are widely employed in code intelligence tasks such as code translation [46, 66]. More recently, reasoning LLMs [14, 50, 52] further enhance LLMs’ inference capabilities, improving their effectiveness in code generation. However, these LLMs still struggle to generate privacy computing code, such as translating Python to MP-SPDZ. As shown in Figure 1, OpenAI o1 only achieves an overall pass@1 of 35%, indicating the lack of MP-SPDZ-specific knowledge in existing models. We attribute this limitation to the scarcity of publicly available MP-SPDZ code and Python–MP-SPDZ parallel corpora, which prevents LLMs from adequately learning the significant semantic-expressing differences between Python and MP-SPDZ.

Moreover, the failure to capture these semantic-expressing differences further reduces the effectiveness of enhancement strategies [46, 66], since such methods implicitly assume that LLMs already possess target-language knowledge, and then offer incremental gains through test-case generation [66] or intermediate-language transformations [46]. When the backbone models lack the knowledge, the gains of these enhancement strategies are limited.

In this work, we propose a data-free approach that directly targets the semantic-expressing differences between Python and MP-SPDZ by incorporating expert knowledge. Our core idea is to collect high-quality expert knowledge to explicitly represent the semantic-expressing differences, and then derive transformation rules for translating Python to MP-SPDZ based on this knowledge. Thus, with the guidance of rules, we leverage the powerful LLMs to automatically translate Python code into MP-SPDZ code. We refer to this approach as SPDZCoder. SPDZCoder does not require additional training data and can be applied to a range of LLMs.

There are two main challenges in implementing the SPDZCoder. The first challenge is *how to collect expert knowledge?* To address this challenge, we propose a rigorous procedure, consisting of four steps: (1) MPC expert recruitment, (2) knowledge elicitation and validation, (3) semantic difference summarization, and (4) transformation rule derivation. In particular, we categorize the summarized semantic-expressing differences into low-level and high-level ones. The former refers to cases where Python’s data types, library functions, data structures, and methods have equivalent or approximate counterparts in MP-SPDZ but differ in naming or syntax,

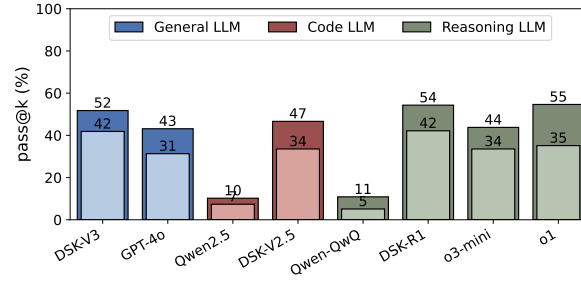


Fig. 1. Overall functional correctness (pass@1 and pass@2) of direct translation across recent representative general-purpose, code and reasoning LLMs on SPDZEval (described in 4.3). Pass@1 is represented in light colors, while pass@2 is represented in dark colors. DSK stands for DeepSeek.

manifesting as misaligned names. The latter includes deeper discrepancies, such as MP-SPDZ’s data-obliviousness requirement and the absence of Python-equivalent library functions. Regarding the derivation of transformation rules, we formulate a set of *refactoring rules* that can address high-level semantic-expressing differences by refactoring Python code into a simplified basic form, and we derive a *generation rule*, which comprises of multiple in-context learning demonstrations, to address low-level semantic differences by guiding LLMs in replacing misaligned names and syntactic expressions.

The second challenge is *how to combine expert knowledge with LLMs*? To address this, we propose a three-stage pipeline that progressively translates Python code into MP-SPDZ code using transformation rules. Specifically, SPDZCoder generates MP-SPDZ code through the following stages:

Refactoring stage. We apply a set of refactoring rules to refactor Python by constructing the missing library functions with basic ones, refactoring non-oblivious statements such as *branch* and *loop* into oblivious form, and, most importantly, replacing non-oblivious algorithms with semantically equivalent functions designed for privacy computing, which may restructure the computation workflow. We refer to the refactored Python code as *Canonical Form Python code (CFP)*.

Generation stage. We leverage the generation rule to translation CFP into MP-SPDZ code. As CFP contains only “easy” expressions, statements and functions to translate, we dynamically select applicable instructions and demonstrations from the generation rule, integrate them into a single prompt, and let the LLM generate MP-SPDZ code from CFP in a single step. We also incorporate a self-reflection rule immediately after the generation process to alleviate hallucinations of module or function names.

Repair Stage. If test cases are available, we may optionally execute the generated MP-SPDZ code against them and collect any execution error messages if the generated code is incorrect. We classify error messages into two categories: compilation/runtime errors (e.g., syntax errors and misused APIs) and logic errors (i.e., incorrect functionality). We then prompt LLMs with these error messages to fix any bugs or logical issues in the generated MP-SPDZ code, further improving performance.

To evaluate the effectiveness of SPDZCoder, we manually construct a benchmark named SPDZEval, comprising 313 pairs of <Python, MP-SPDZ> functions with test cases. Furthermore, SPDZEval is partitioned into six data splits, each representing a distinct class of challenging tasks in MP-SPDZ implementation. We compare SPDZCoder with the recently advanced LLM-based translation methods, including UniTrans [66] and InterTrans [46], as well as the state-of-the-art general-purpose, code, and reasoning LLMs with MP-SPDZ documentation on the SPDZEval dataset. Extensive results show that SPDZCoder outperforms strong baselines in pass@1 and pass@2. Specifically, SPDZCoder with GPT-4 as its backbone achieves overall functional correctness values of

85.94% and 92.01% in pass@1 and pass@2, respectively, whereas the best-performing baseline achieves only 63.58% and 76.36%, respectively.

To the best of our knowledge, SPDZCoder is the first to explore how to translate GPL (Python) to MP-SPDZ code. SPDZCoder explicitly teaches LLM the semantic-expressing differences at different levels by incorporating expert knowledge. The contribution of our work are as follows:

- We propose a novel training-free framework, named SPDZCoder, that combines LLMs with expert knowledge to translate Python to MP-SPDZ, including a rigorous procedure for collecting expert knowledge and a pipeline to progressively perform translation.
- We construct a manually-written benchmark dataset containing over 300 pairs of <Python, MP-SPDZ> functions with test cases for evaluating future approaches.
- We conducted extensive experiments to evaluate SPDZCoder against baselines. Evaluation results demonstrate the effectiveness of SPDZCoder in performing code translation for privacy computing.

2 Background

2.1 Multi-party Computation

Multi-party computation enables multiple parties who do not trust each other or any trusted third party to securely compute a function over their private inputs. Concretely, all parties agree on a function to compute, and then run an MPC protocol to jointly compute its output. All parties learn only the output of the computation, while dishonest parties cannot infer the private inputs of honest parties. A classical two-party computation (2PC) problem is Yao’s Millionaires Problem [67] introduced by Andrew Yao in the 1980s: “Two millionaires wish to know who is richer; however, they do not want to find out inadvertently any additional information about each other’s wealth.” In such a problem, the function to compute is $f(x_1, x_2) = x_1 \leq x_2$, where x_1 and x_2 are the private inputs of the two parties, respectively.

MPC requires data-oblivious algorithms. Formally, an algorithm $\mathcal{A}$ is considered data-oblivious if, for any two inputs x and y of the same length, the sequence of instructions executed and the sequence of memory accesses performed by $\mathcal{A}$ on x and y are identical. When an algorithm is not data-oblivious, an adversary can potentially infer information about the private inputs by observing the execution pattern. Data-oblivious algorithms ensure that the observable behavior of the computation, which refers to the sequence of executed instructions as well as the timing and locations of memory accesses, remains independent of the secret inputs, thereby preserving privacy. Not all algorithms have an oblivious counterpart; for example, there is no *oblivious quicksort*.

From a software engineering perspective, the motivation to make code privacy-aware stems from the growing need for collaborative analytics, such as cross-institution credit approval, under data-privacy regulations and among mutually untrusted parties. Such privacy-aware design is especially critical in finance, where collaboration yields better intelligence but data privacy remains paramount. For example, **Bank A** holds users’ income and asset features, while **Bank B** holds the users’ behavioral features. The two parties jointly use a decision tree model to perform credit approval, but neither party is allowed to access the other’s private features.

The code snippet in Figure 2 illustrates how, during inference, the original plaintext Python implementation of decision tree¹ are supposed to be rewritten into a privacy-preserving one. As shown in the Figure, in plaintext settings, the decision tree inference code simply takes all features as direct inputs and performs comparisons and threshold-based branching, *i.e.*, it follows a single path down the tree (simulated by arrays) until reaching a leaf. However, when the features are distributed across Bank A and B, and must remain confidential, this plaintext implementation becomes infeasible. Python provides no secret data types or data structures to load encrypted

¹In practical machine learning, decision trees are almost universally stored in an array-based representation rather than as pointer-linked node objects. This layout enables efficient inference.

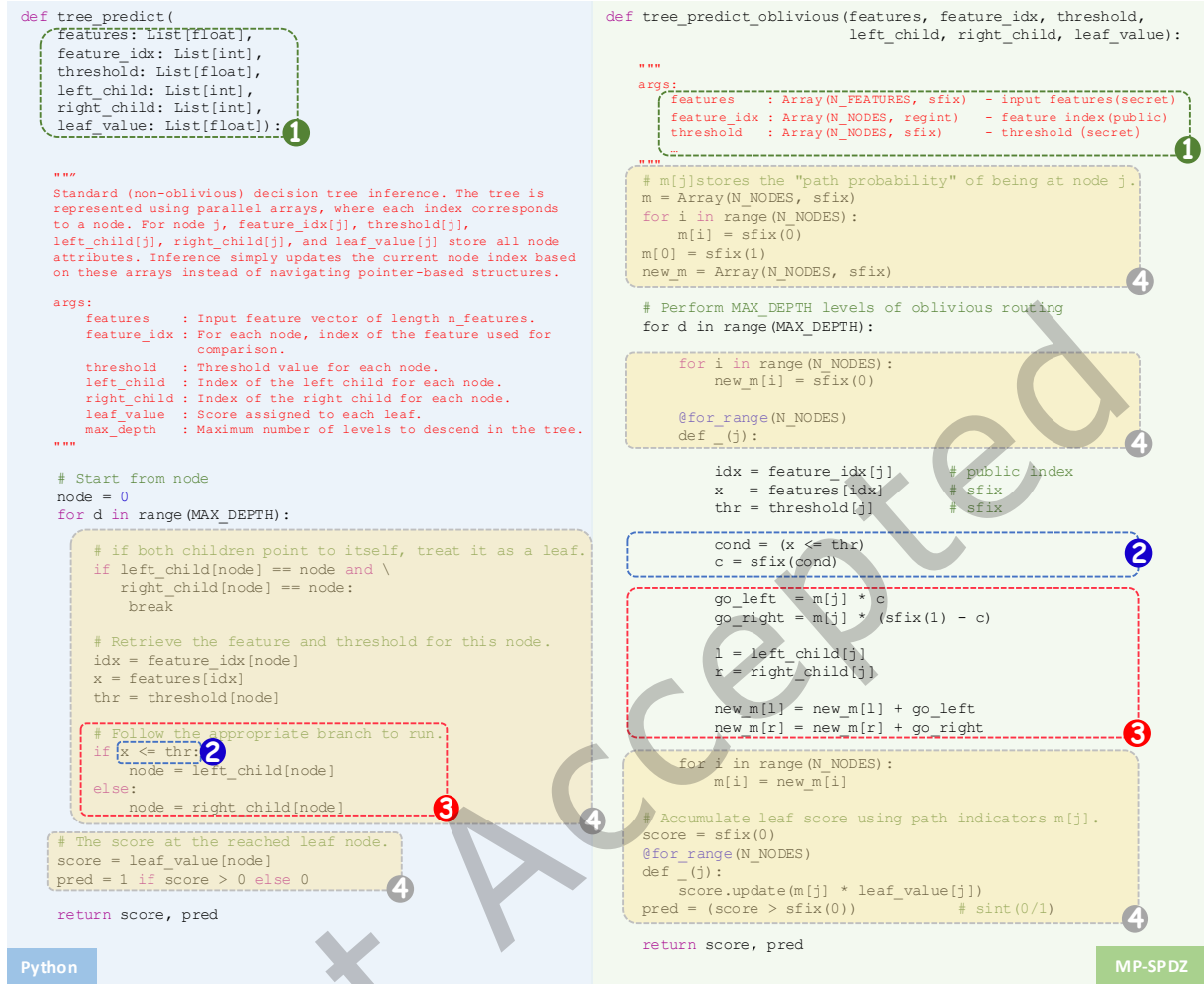


Fig. 2. An example of decision tree inference illustrating why secure collaborative analytics requires MPC programs. The tree is represented by arrays. The left side shows the original Python code, and the right side shows its MP-SPDZ counterpart. Four changes are introduced in the MPC version, each marked with an Arabic numeral and a matching color block: ① data types, ② comparison operations, ③ data-oblivious branching, and ④ data-oblivious tree traversal and array access with a secret index.

values, nor does it support secure operators, secure branching, and array access with secret index. In fact, vanilla branching itself can leak sensitive information, e.g., high income leading to the right subtree.

Thus, we transform the computation into an MP-SPDZ program. In this example, several key changes are made: (1) Data types. Plaintext types such as `int` and `float` are replaced with secret-shared types, such as `sint` and `sfix`, to ensure that encrypted values cannot be directly inspected. (2) Secure comparison. Operations, e.g., comparison, in plaintext cannot support computation on secret-shared values, and is substituted with secure comparison. (3) Secure branching. All branching is implemented using the combination of results from each branch instead of using `if` statements directly. (4) Oblivious traversal. To compute the leaf node value, the oblivious formulation replaces the data-dependent tree traversal with a parallel propagation of path weights

across all nodes. Only the nodes on the true decision path retain a weight of 1, while all others remain 0. This removes data-dependent branches entirely and prevents any leakage of the tree traversal path. Furthermore, because array access by a secret index is not permitted, the leaf value is computed through a loop that aggregates all leaf values weighted by these path indicators. All these changes ensure that private data remain usable but invisible throughout computation.

To support such secure collaboration, MPC platform developers build and integrate the underlying cryptographic primitives, MPC protocols, and developer interfaces into a unified software platform. This platform is then provided to banks, where application developers are required to write MPC programs by implementing the computational or operational functions needed for specific financial analytics tasks, such as decision tree, while ensuring that no raw data are exposed. A real-world commercial case is that TsingJ Corporation launched its production-level MPC frameworks, named PrivPy [41], in China Everbright Bank. In this deployment, software engineers at the bank utilize the provided framework to develop MPC-enabled programs at the API or user level.

However, we would also like to emphasize that, even with such platforms available, writing MPC code remains non-trivial due to the limited availability of rich library functions and the requirement of data obliviousness. SPDZCoder aims to enhance LLMs to automatically translate Python code into MP-SPDZ, allowing developers and researchers to reuse plaintext implementations and reducing manual effort.

2.2 MP-SPDZ Framework

As discussed above, MPC is a key technique for secure collaboration on sensitive data. While commercial platforms such as PrivPy have been deployed in production, they are proprietary and closed-source.

In contrast, most open-source MPC frameworks are prototype-oriented and require developers to manually assemble higher-level operations. Despite this, they remain indispensable for research and applications. Among them, MP-SPDZ stands out for its flexibility: it provides developers with a high-level programming interface for privacy-preserving computation, including specialized data types and a modest yet practical set of data structures, library functions, and classes. Although MP-SPDZ adopts a Python-like surface syntax, its data types and operations follow a semantic model defined by secret sharing and cryptographic protocols, which lies outside Python’s plaintext computation. As a result, Python and MP-SPDZ exhibit fundamental differences in how program semantics are expressed.

First, the data types and arithmetic operations are not aligned. MP-SPDZ additionally provides secure and clear data types such as secret integers (`sint`), secret fixed-point numbers (`sfix`), clear integers (`cint`) and clear fixed-point numbers (`cfix`), among others. On the one hand, MP-SPDZ uses fixed-point data types and corresponding arithmetic operations instead of floating-point data types due to efficiency considerations. On the other hand, each data type supports a limited set of arithmetic operations and has certain restrictions, *e.g.*, values of type `sint` do not support bitwise logical operations.

Second, memory access and control flow must be implemented in a data-oblivious manner. For example, when the *condition* in `if` statements is a secret variable, we should represent the final result as a combination of results from each branch in written code so as to achieve a data-oblivious execution pattern. In the following simple example, when the variable `x` is secret, the code:

```
# Assume x is a floating-point number
from math import sqrt
if x>0:
    y = sqrt(x)
else:
    y = sqrt(-x)
```

should be written as a combination of the two branches as follows:

```
# Assume x is a sfix number
from Compiler.mpc_math import sqrt
y = (x>0) * sqrt(x) + (x<=0) * sqrt(-x)
```

We can see that, regardless of the value of variable x , the execution path remains unchanged. Note that, for such a simple case, one can also use the built-in secure branching construct: $y = (x>0).if_else(sqrt(x), sqrt(-x))$.

Third, unlike the comprehensive algorithm libraries such as Numpy, MP-SPDZ merely provides a few basic APIs, requiring programmers to implement many advanced computational functions and operations on Array, Matrix from scratch. For example, MP-SPDZ only offers 7 basic trigonometric functions and 6 non-linear math functions in its `mpc_math` module while Numpy provides 18 and 38 respectively.

3 Approach

In this section, we present SPDZCoder for privacy-preserving code translation from Python to MP-SPDZ. We first describe how we collect expert knowledge to identify semantic-expressing differences between Python and MP-SPDZ and derive the corresponding transformation rules (§ 3.1). We then present the rule-guided translation pipeline, which applies these transformation rules to translate Python code into MP-SPDZ code through three stages: refactoring, generation, and repair (§ 3.2). Together, these two parts enable SPDZCoder to incorporate expert knowledge into LLM-based privacy-preserving code translation.

3.1 Collecting Expert Knowledge

We propose a rigorous and structured procedure for collecting expert knowledge. Specifically, we adopt a multi-session expert elicitation process with well-defined objectives, guided tasks, and iterative validation. We then summarize semantic-expressing differences between Python and MP-SPDZ and formulate transformation rules.

Expert Recruitment and Session Setup. We engaged four senior MPC experts from TsingJ Corporation², each of whom has at least three years of experience in developing MPC frameworks and applications. Notably, one participant was the lead designer and developer of the prototype of TsingJ’s proprietary MPC framework (PrivPy). Although all experts are currently employed at TsingJ, we intentionally selected experts with diverse educational and professional backgrounds, including both academic and industry experience. The experts were recruited through institutional collaboration and received standard compensation for their participation. Before the elicitation sessions, we conducted a short preparatory briefing to familiarize the experts with the research objectives, the translation task, and the preliminary findings from our pilot analysis, so that all participants shared a common understanding of the problem scope and evaluation criteria.

Following this preparation, we held four structured elicitation sessions over a two-week period, each lasting approximately three hours and facilitated by one researcher. All four experts participated in each session, which followed a consistent agenda focusing on: (1) identifying semantic-expressing differences between Python and MP-SPDZ and (2) validating and refining the identified differences across examples. The four sessions were conducted iteratively, with earlier sessions emphasizing exploration and later ones focusing on consolidation and validation.

Knowledge Elicitation and Validation. We adopt both reactive and proactive elicitation to capture expert knowledge. For reactive elicitation, we design a semi-structured process combining open-ended discussions with guided code walkthroughs. During the sessions, experts were presented with representative Python code snippets covering arithmetic operations, data structures, control flows, and algorithms such as sorting, and

²TsingJ is a startup company that specializes in developing privacy-computing platform and algorithm library (<https://www.tsingj.com/index.html>), and the research group works closely with the company.

were asked to explain how equivalent semantics should be expressed in MP-SPDZ. In addition, we incorporate proactive elicitation by encouraging experts to propose and supplement semantic-expressing differences they had encountered in their own development experience, beyond those illustrated in the provided examples. This dual-mode strategy helps us to capture both prompted and spontaneous expert knowledge, improving the coverage of domain-specific knowledge.

After each session, we consolidated expert inputs into a structured list of semantic differences. In the subsequent session, experts cross-checked and refined this list to further improve its completeness, consistency³, and accuracy. The finalized set of differences was then used by the research team as the foundation for designing translation rules in the next stage.

Summarizing and Categorizing Semantic-Expressing Differences. Based on the elicited knowledge, we identified semantic-expressing differences between Python and MP-SPDZ that characterize the translation challenges, particularly the unique constraints and conventions for expressing common semantics in MP-SPDZ.

First, among these differences, some stem from disparities in the richness of ways to express semantic or fundamental differences in computational paradigms, primarily involving the absence of equivalent library functions and the data-obliviousness requirements of MPC. For example, the *condition* in *branching* statements cannot include any secret data type to ensure data-obliviousness, and the *quicksort* algorithm should be replaced with an oblivious sorting algorithm, such as oblivious radix sort[24], due to the non-existence of oblivious quicksort. We refer to these differences as high-level semantic-expressing differences, and we summarize them in Table 1.

Second, the remaining differences primarily involve misaligned names, meaning that data types, library functions, data structures, and methods in Python have equivalent or approximate counterparts in MP-SPDZ but differ in naming conventions. For example, `numpy.power(x, y)` is semantically equivalent to `mpc_math.pow_fx(x, y)` in MP-SPDZ. We refer to these as low-level semantic-expressing differences.

Deriving Transformation Rules. Building upon the categorized differences, we design two sets of transformation rules. To address high-level semantic differences, we introduce a set of *refactoring rules* that systematically rewrite Python code into a simplified *basic form*. This basic form serves as an MP-SPDZ-friendly representation, in which complex semantics are expressed primarily in terms of primitives that are natively supported by MP-SPDZ. For example, to address the absence of many computational functions in MP-SPDZ, we leverage the fact that sophisticated nonlinear functions can be expressed using a small set of four fundamental nonlinear functions. Accordingly, we establish a rule to simplify the implementation of complex nonlinear functions. Similarly, high-level semantic expressing differences in operations associated with container data types are decomposed into more fundamental ones, and complex Python syntactic constructs, such as syntactic sugar, are rewritten using basic statements. Overall, we designed ten types of refactoring rules to cover and address the identified high-level semantic-expressing differences, and Table 2 presents them and their descriptions related to certain high-level semantic-expressing difference.

To address low-level semantic differences, we introduce a *generation rule* based on in-context learning. This rule applies a single-step generation procedure and provides demonstrations to guide the LLM in resolving misalignments in data types and naming conventions (including function, method, and module names) covered by the refactoring rules. For example, we use few-shot demonstrations to instruct the LLM to replace `numpy.power(x, y)` with `mpc_math.pow_fx(x, y)`.

Finally, to validate whether LLMs adhere to the intended transformation logic encoded in a rule, we prompt LLMs with the corresponding rule prompts on sample code snippets and refine the prompts if the generated transformations deviate from the expected rule behavior.

³Consistency means that the list is free of duplicated or conflicting definitions

Table 1. Overview of high-level semantic differences with descriptions. Text in **red** and **blue** highlights aspects related to data-obliviousness and those unrelated to it.

High-level Differences	Detailed Description
Math and Universal function	For non-linear functions, MP-SPDZ includes only six basic functions: <code>exp2_fx</code> , <code>InvertSqrt</code> , <code>log2_fx</code> , <code>log_fx</code> , <code>pow_fx</code> , and <code>sqrt</code> . For trigonometric functions, it contains only seven basic ones such as <code>atan</code> , <code>acos</code> , and <code>asin</code> . MP-SPDZ supports only basic math functions and does not include any advanced mathematical functions.
Data structure method	MP-SPDZ does not support Python one-dimensional data structure methods such as <code>append</code> .
Syntax sugar	MP-SPDZ lacks support for comprehensions, ternary expressions, and the <code>for-else</code> syntactic sugar found in Python.
While loop	For encrypted computations in MP-SPDZ, iteration is required to be performed using <code>for-loops</code> .
Array/Matrix Creation	MP-SPDZ provides creation methods similar to NumPy's <code>zeros</code> , <code>zeros_like</code> , <code>full</code> , <code>full_like</code> , <code>array</code> , <code>as_array</code> , <code>mat</code> , <code>matrix</code> , and <code>as_matrix</code> . However, NumPy offers a much broader set of creation functions, including <code>eye</code> , <code>identity</code> , <code>empty</code> , <code>diag</code> , <code>linspace</code> , and <code>logspace</code> , among others.
Array/Matrix Indexing	MP-SPDZ supports indexing methods similar to NumPy's integer array indexing, <code>[i]</code> , <code>[i, j]</code> , <code>[:, :]</code> , and <code>[i, :]</code> . However, NumPy provides a richer variety of indexing functions such as <code>take</code> , <code>choose</code> , <code>diag</code> , <code>select</code> , <code>place</code> , and <code>put</code> .
Array/Matrix Manipulation	MP-SPDZ supports operations similar to Python's <code>concatenate</code> , <code>transpose</code> . In contrast, NumPy offers a more extensive set of operations, including <code>reshape</code> , <code>copy</code> , <code>delete</code> , <code>insert</code> , <code>append</code> , <code>stack</code> , and <code>split</code> .
Branch	For conditional branches based on secret variables, MP-SPDZ requires the computation to be represented as a linear combination of each branch.
Chained comparison	MP-SPDZ does not support chained comparisons on encrypted data.
Multiple return	MP-SPDZ does not allow multiple return statements in encrypted computations.
Break & Continue	MP-SPDZ does not allow jump statements in encrypted computations.
Sort	When sorting encrypted data, MP-SPDZ mandates the use of data-oblivious sorting algorithms.

3.2 Automatic Code Translation Pipeline

In this section, we describe how SPDZCoder leverages expert knowledge, *i.e.*, transformation rules, to guide LLMs to translate Python code to MP-SPDZ code. As shown in Figure 3, the pipeline comprises three main stages: (1) **Refactoring stage**: We apply a set of refactoring rules to prompt the LLM to refactor Python code into

Table 2. Overview of refactoring rules with corresponding descriptions. Text in **red** and **blue** highlights rules related to and unrelated to data-obliviousness, respectively.

Refactoring Rules	Description
Linear & Nonlinear Function	When refactoring advanced non-linear function, this rule limit LLM to use given four basic non-linear functions: ‘exp’, ‘ln’, ‘sqrt’, ‘inversqrt’.
Data Structure Method	This rule explicitly reminds LLM never using List/Set/Tuple method <i>e.g.</i> , ‘append’, ‘extend’, ‘insert’, ‘reverser’, ‘count’, and asks LLM to use simple, commonly-used, rigid statement.
Syntax Sugar	This rule reminds LLMs removing all Python syntax sugars.
While Loop	This rule asks LLM to refactor while loop into for loop.
Array Operations	This rule asks LLM to break down advanced array operations (array creation, indexing, slicing and manipulation) into given operations.
Break & Continue	This rule demonstrates examples of how to eliminate the break and continue keywords in code and then refactor the code into data-oblivious form
NestedIf & MultipleReturn	Refactor into plain conditions (without nest condition statement) and single return statement
Chained Comparison	Refactor chained comparison into comparisons with logic operation
Oblivious Format	Refactor branch block into the format of combinations of branches.
Sort	Refactor sort block into radix sort.

Canonical Form Python (CFP). (2) **Generation stage**: We apply the generation rule to CFP to generate MP-SPDZ code, and incorporate a self-reflection component to mitigate hallucinations. (3) **Repair stage**: We utilize error messages produced by executing test cases to further refine the translation. We provide an illustrative example of SPDZCoder’s translation in Appendix § A.1.

Stage 1: Refactor Python Code to CFP. Each refactoring rule specifies a target modification for a certain category of high-level semantic-expressing differences. We prompt the LLM with these rules one by one to perform Python-to-Python refactoring. At this stage, the rules implement missing library functions using more basic ones(*e.g.*, we implement all non-linear math functions with four basic ones: exp, ln, sqrt and inversqrt, all of which have counterparts in MP-SPDZ), convert non-oblivious statements such as branches and loops into data-oblivious form, and, most importantly, replace the non-oblivious algorithms with semantically equivalent functions designed for privacy-preserving computation, *e.g.*, replacing *quicksort* with *radix sort*. After applying these rules, we obtain intermediate Python code in which high-level semantic differences are systematically mitigated. The resulting code is more explicit and constrained: the functions and data structures it uses have direct counterparts in MP-SPDZ, and its control flow and algorithms are data-oblivious. This reduces the difficulty of subsequent MP-SPDZ code generation. We refer to this intermediate representation as *Canonical-Form Python (CFP)*.

For example, one category of refactoring rules is the Break rule (row 6 in Table 2), which guides the LLM to eliminate the **break** keyword from a Python program, while preserving the original semantics. The following code example contains a for loop with a **break** statement.

```
a = INIT_ARRAY # Assume a is a secret Array
for i in range(len(a)):
    if a[i]>2:
        break
    a[i] += 1
```

To eliminate **break**, the rule introduces a boolean flag initialized to False before the loop and uses this flag to simulate the effect of the **break** statement. The resulting CFP is as follows:

```
a = INIT_ARRAY
flag = False
for i in range(len(a)):
    flag = flag or (a[i]>2)
    a[i] = flag*a[i] + (1-flag)*(a[i]+1)
```

At the refactoring stage, a given Python program may not exhibit all of the summarized high-level semantic difference patterns. To address this, we introduce a pattern-matching strategy that enables SPDZCoder to dynamically select applicable refactoring rules. Specifically, we either parse the input program into an abstract syntax tree (AST) to search for targeted syntactic constructs, such as **break**, or query the LLM to analyze whether the input program exhibits specific high-level semantic differences. If a pattern match succeeds, SPDZCoder applies the corresponding refactoring rule; otherwise, the rule is skipped. This strategy prevents unnecessary rule applications and improves overall efficiency.

In the released replication package (introduced at the end of the paper), each refactoring rule is implemented in a separate file together with the corresponding pattern-matching logic that determines when the rule is triggered. This organization makes the rule inventory and pattern-selection process directly inspectable and reproducible.

Stage 2: Translate CFP to MP-SPDZ Code. In the generation stage, we apply a generation rule to instruct the LLM to translate CFP into MP-SPDZ. Specifically, the generation rule consists of multiple in-context learning demonstrations addressing various low-level semantic-expressing differences, such as data types, logical operations, and function names. Each demonstration illustrates how a specific CFP construct is translated into its MP-SPDZ counterpart. For example, the float type is translated into `sfix`; the logical operation `(x > -3 and x <= 0)` is translated into `(x > -3).bit_and(x <= 0)`; and the NumPy function `numpy.exp(x)` is translated into the `mpc_math` function `mpc_math.pow_fx(cfix(math.e), x)`. We employ pattern matching to select applicable demonstrations and organize them into chat messages for the LLM, each consisting of a user prompt and an LLM response that illustrate how to translate CFP into MP-SPDZ code for specific low-level semantic differences.

We observe that buggy code can be generated due to hallucinations, and this issue becomes more pronounced when generating MP-SPDZ code. We attribute this to the lack of a comprehensive function library in MP-SPDZ and to LLMs' insufficient knowledge of MP-SPDZ. Therefore, we incorporate a self-reflection component [11, 31] immediately after the generation rule, which prompts the LLM to verify whether the operations, module names, and function names conform to the generation rules. If deviations are found, the LLM revises the corresponding part of the code.

Stage 3: Compilation and Runtime Messages as Feedback. In the repair stage, we optionally execute the generated MP-SPDZ code against test cases if they are available, and collect any resulting error messages. These

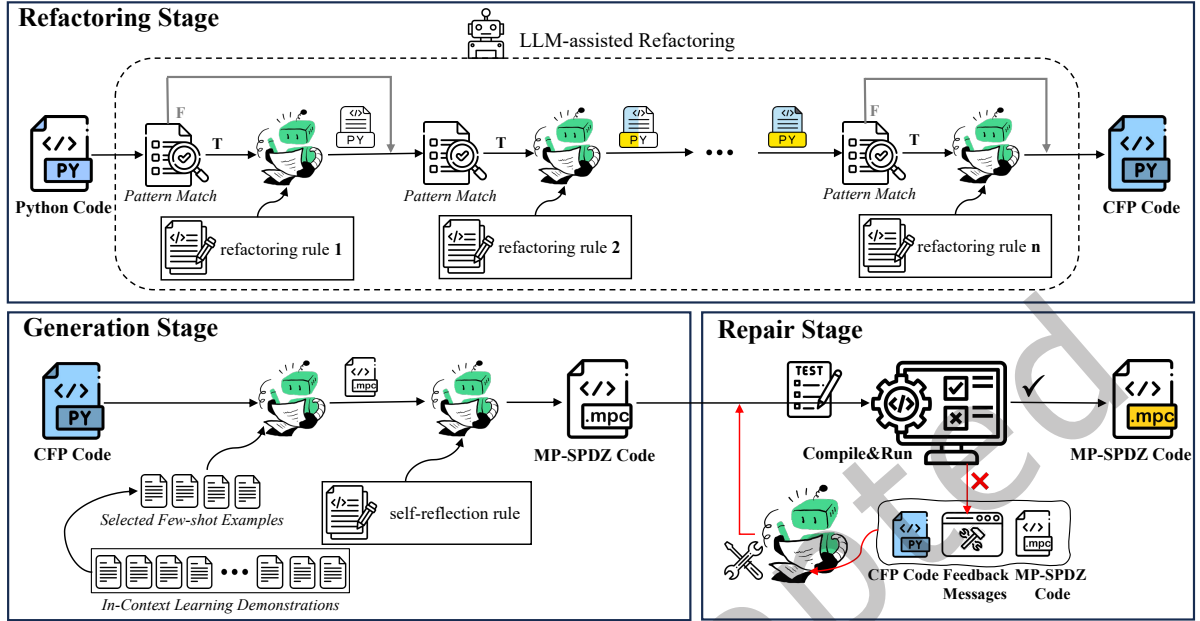


Fig. 3. Overview of the translation pipeline. The pipeline consists of three stages: (1) the refactoring stage converts Python code into Canonical Form Python (CFP) using refactoring rules; (2) the generation stage translates CFP into MP-SPDZ code using the generation rule, and (3) the repair stage utilizes error messages produced by executing test cases to further refine the translation.

error messages indicate whether the code fails to execute due to compilation or runtime errors, or instead contains logical errors, *i.e.*, incorrect functionality.

If a compilation or runtime error occurs, we populate the `FixCompilationRuntimeError` prompt template with the placeholders {CFP}, {error message}, and {generated MP-SPDZ code}. If the code exhibits incorrect functionality, we populate the `FixFunctionalityError` prompt template with {CFP} and {generated MP-SPDZ code}. We then use these prompts together with the few-shot demonstrations selected in the generation stage to prompt the LLM to repair the code through another round of translation. The repair process repeats until the MP-SPDZ code is correct or a predefined retry limit is reached. If the limit is reached, we accept the last revised MP-SPDZ code as the final output.

4 Experimental Setup

4.1 Research Questions

RQ1. What is the effectiveness of SPDZCoder compared to baselines? We evaluate SPDZCoder against three categories of LLM-based translation approaches on SPDZEval (§ 4.3). We also investigate the performance of SPDZCoder using various LLMs as backbones to mitigate bias.

RQ2. What is the impact of refactoring and repair stages? We evaluate the contributions of the refactoring and repair stages in SPDZCoder through ablation studies. In addition, we perform an error taxonomy analysis to characterize the failure modes of SPDZCoder.

RQ3. What is the correctness ratio of refactoring Python to CFP in SPDZCoder? We investigate the refactoring correctness in refactoring stage since mis-refactoring can lead to performance drops in the generation stage.

RQ4. What are the token usage, runtime cost, and token savings achieved through pattern-matching strategy? We compare SPDZCoder with the baselines on average token consumption and runtime overhead. We also quantify the token savings brought by the rule pattern-matching strategy in the refactoring and generation stages separately.

4.2 SPDZCoder Setup

LLM Backbone. We employ a set of general-purpose, code, and reasoning LLMs as the backbone models for SPDZCoder, including GPT-4 [51], GLM-4 [19], DeepSeek-V2.5 [12], DeepSeek-V3 [13], OpenAI o1 [50] and DeepSeek-R1 [14], to evaluate its effectiveness across diverse LLMs.

Generation Configuration and Hyper-parameters. Following prior work on code generation evaluation, such as HumanEval [9] and MBPP [5], we adopt stochastic decoding with a temperature of 0.7 to obtain diverse candidate outputs for pass@k estimation. Using a non-zero temperature is necessary for meaningful pass@k evaluation: when the temperature approaches 0, the model becomes nearly deterministic, resulting in k identical samples and causing pass@k to collapse to pass@1. The choice of 0.7 follows widely used decoding configurations in prior literature. For reproducibility, an evaluation stability analysis under temperature sampling is provided in § 6.5. All other generation configurations of the backbone LLMs remain unchanged.

Our experiments involve two hyperparameters: repetition and max_feedback. The repetition parameter determines the number of MP-SPDZ code samples generated by our pipeline for each Python function in SPDZEval, while max_feedback limits the maximum number of retries allowed in the repair stage. We set repetition to 2 and max_feedback to 3.

4.3 Evaluation Dataset

In this section, we first review existing benchmarks and then present the construction process of SPDZEval.

Review of Existing Resources. Evaluating Python-to-MP-SPDZ code translation requires paired source-target program artifacts, program units at an appropriate level of granularity, and coverage of MPC-specific features. Therefore, we examine existing code-generation benchmarks such as HumanEval [9], MBPP [5], LLMTrans [63], and RepoTransBench [61]. Our findings are summarized in Table 3.

Overall, the existing benchmarks do not meet the requirements of our task for several reasons: (1) While existing benchmarks have significantly advanced empirical evaluation for code generation, they are scoped to conventional plaintext programming settings. As a result, they do not support cross-language translation across fundamentally different semantic models, and thus do not incorporate MPC-specific data types, data structures, data-oblivious control flow, or arithmetic constraints. (2) Many benchmarks provide only natural-language descriptions or partial code fragments, without paired source-target snippets necessary for translation evaluation (e.g., HumanEval, MBPP, MathQA-Python [5]). (3) Several benchmarks target incompatible task types, such as code completion or issue-driven code repair (e.g., SWE-Bench Lite [33]), or they emphasize problem domains unrelated to numerical computation or MPC semantics (e.g., competitive-programming tasks or UI/UX-oriented logic). Therefore, we develop SPDZEval, a new benchmark tailored for Python-to-MP-SPDZ code translation.

Our Benchmark Dataset. To construct SPDZEval, we engaged an undergraduate student and several engineers with strong proficiency in Python and MPC to carry out problem collection, implementation, and quality review.

Table 3. Overview of representative code benchmarks and their limitations for privacy-preserving code translation. SPDZEval is the only dataset designed for Python-to-MP-SPDZ translation. #Samples denotes the number of samples in the dataset, while #LOC denotes lines of code. PL stands for programming language, and <Python, MSZ> stands for <Python, MP-SPDZ>.

Dataset	Language	Code Task	Source	Granularity	# Samples	#LOC	Misalignment with Our Task
DS-1000 [38]	Python	Code infilling	Stack Overflow	Statement	1000	3.8	Targets code tasks that are largely unrelated to numerical computation and include only limited computational content.
APPS [25]	Python	Code synthesis	Codewars, AtCoder	Function	10000	18	Designed for algorithmic problem solving rather than paired code translation, with tasks emphasizing competitive-programming logic instead of computation or MPC semantics.
HumanEval [9]	Python	Code completion	Human	Function	164	11.5	Provides only function signatures and docstrings without paired source–target code, and lacks coverage of numerical computation and MPC-specific structural constraints.
MBPP [5]	Python	Code synthesis	—	Function	974	6.8	Not a paired translation benchmark; tasks consist of small, self-contained exercises that do not capture numerical computation patterns or MPC requirements.
MathQA-Python [5]	Python	Code synthesis	Math Study Sites	Statement	2,985	7.6	Targets NL mathematical reasoning rather than code translation, focuses on symbolic word problems, and does not provide paired executable programs.
CoderEval-Python [69]	Python	Code synthesis	Github	Function	230	30	Generates non-standalone functions that depend on large contextual codebases, making it incompatible with function-level paired translation.
ClassEval [16]	Python	Code completion	—	Class	100	45.7	Targets class-skeleton completion for management, formatting, or game logic, and its class-level granularity is overly coarse for early-stage MPC translation studies.
CodeArena [65]	44 PLs	Code completion	Online Code QA	Function	400	—	Focuses on UI/UX, tooling, or data-handling tasks common in Q&A platforms, with limited computational Python coverage.
SWE-Bench Lite [33]	Python	Code edit	Github	Repository	300	—	Built around PyPI <issue, PR> pairs, reflecting software-maintenance workflows rather than code translation tasks.
CoST [72]	7 PLs	Code translations	GeeksforGeek	Statement	132046	3	Uses similarity-based evaluation and omits MP-SPDZ as a target language; its statement-level granularity is too fine-grained for assessing MPC-specific translation semantics.
HumanEval-X [71]	5 PLs	Code translations	HumanEval	Function	164	8	A multilingual HumanEval variant that lacks MP-SPDZ as a target language and cannot capture MPC behavior.
LLMTrans [63]	8 PLs	Code translations	Rosetta Code, d2l-ai	File	350	24	Uses similarity-based evaluation without MP-SPDZ support, and its class-level granularity is too coarse for exploratory MPC translation.
RepoTransBench [61]	<Python, Java>	Code translation	The Stack v1&v2	Repository	100	352	Requires additional configuration files, and its repository-level granularity is too coarse for preliminary MPC translation evaluation.
SPDZEval	<Python, MSZ>	Code translation	Programming sites & Data science package library	Function	313	9.5	✓

For problem collection, under the engineers’ guidance, the student curated candidate problems from Python programming websites [57, 58] and instantiated each problem as a function. In addition, the student curated functions directly from the NumPy library. The curation process followed three principles: diversity (e.g., varied syntax, control-flow, and data-manipulation patterns), coverage (e.g., basic and advanced arithmetic, array and matrix operations, and complex mathematical routines), and semantic relevance to computational workloads. For implementation, the student first drafted a docstring for each selected problem or function to ensure an unambiguous specification then implemented the function in both Python and MP-SPDZ, producing a (source, target) program pair. Each pair was accompanied by a set of input–output test cases designed by the student.

All implementations and test cases underwent manual inspection by the engineers. The review assessed: (1) the correctness of both Python and MP-SPDZ implementations; (2) the faithfulness between the docstring and the two program artifacts; and (3) whether the examples exhibited adequate complexity in MPC-related data types, control flow, and computation patterns. Pairs that failed to meet these criteria were revised or discarded. If certain common computational workloads were underrepresented, the student was instructed to supplement the collection accordingly.

Through this process, we obtained 313 (Python, MP-SPDZ) function pairs. A comparison between SPDZEval and existing benchmarks is also presented in Table 3. Overall, SPDZEval is specifically designed for MPC-oriented code translation, whereas existing datasets primarily focus on conventional plaintext code completion tasks. Moreover, SPDZEval uses function-level granularity⁴, which occupies a middle ground between statement-level benchmarks (often too simple) [5, 38, 68] and class- or repository-level datasets (often much more difficult) [16].

⁴Function-level code tasks ask an LLM to generate a single, standalone code unit, providing richer semantics than statement-level tasks while remaining tractable for an exploratory translation study.

Table 4. Overview of the data splits in SDPZEval

Split Name	Abbr.	# Entries	Split Description and Corresponding Challenging Tasks
array	AR	43	Array access or traverse, including list indexing, slicing, iteration and access.
loop	LO	97	Loop, iteration, multiple if, break and continue, multiple return, and tasks containing them.
branch	BR	34	Multiple kinds of branch, e.g., nested if, multiple if, multiple return, and tasks consisting of them.
math	MA	38	Python scalar math functions and computations
numpy	NP	66	NumPy array universal functions and NumPy array operation, e.g., creation, indexing, manipulation.
syntax	SX	35	Advanced syntax or syntax sugars in Python and built-in sophisticated functions, e.g., map().

In practice, the task difficulty in SPDZEval is moderately higher than that of function-level datasets such as MBPP [5] and HumanEval [9].

We divide SPDZEval into six data splits, as summarized in Table 4. Each split corresponds to a distinct set of code patterns and is designed to evaluate a method’s ability to translate functions exhibiting those patterns. For instance, the NumPy split (the fifth row) focuses on array creation, manipulation, and NumPy universal functions, namely ndarray-level mathematical functions, thereby enabling an assessment of translation performance on code involving such operations.

4.4 Baselines and Metric

4.4.1 Baselines. As shown earlier in Figure 1 in § Introduction, existing LLMs fail to directly translate Python code into MP-SPDZ program. To evaluate SPDZCoder, we design or adopt three strong baseline approaches.

API Documentation (API Doc). Modern LLMs are capable of capturing long-range dependencies and can effectively learn from extended input contexts. Therefore, we enhance translation by providing MP-SPDZ API documentation as in-context guidance for long-form reasoning, explicitly highlighting semantic-expressing differences in the prompt. For general-purpose LLMs, we additionally instruct them to summarize the Python code before translation, placing the code summary after the MP-SPDZ API documentation in the prompt. However, we do not apply this to code LLMs and reasoning LLMs, as code LLMs are less adept at generating natural language, and reasoning LLMs inherently follow a think-before-answer paradigm.

UniTrans. UniTrans [66] enhances LLM-based translation by augmenting the translation with LLM-generated test cases. It employs an error analyzer to extract error information—including error lines and error messages—from the execution results of incorrectly translated programs. This error information serves as guidance for correcting incorrect programs. We find that LLMs, when using their original prompt, cannot generate valid test cases for this task, either in format or in expected behavior. Thus, we replace the automatically generated test cases with our manually prepared ones, *i.e.*, the same test case suite used throughout our experiments.

InterTrans. InterTrans [46] leverages the multilingual capabilities of LLMs to enhance code translation via transitive intermediate translations. InterTrans first employs a collection of intermediate programming languages and utilizes a planning algorithm (ToCT) to generate candidate translation paths (source–target or source–intermediate–target). It then sequentially executes translation paths and validate the correctness of the translated

Table 5. Functional correctness (pass@1 and pass@2) of SPDZCoder and baseline methods across six data splits on SPDZEval. Each method is evaluated with a comprehensive set of LLM backbones, covering general, code and reasoning models. All results are presented in percentage format (%). The best results are highlighted in **bold**.

Method	pass@1								pass@2							
	array	loop	branch	math	numpy	syntax	overall		array	loop	branch	math	numpy	syntax	overall	
API Doc	GPT-4	44.19	24.74	26.47	10.53	39.39	37.14	30.35	60.47	53.61	32.35	26.32	53.03	62.86	49.84	
	DeepSeek-V3	51.16	32.99	35.29	10.53	34.85	34.29	33.55	60.47	53.61	55.88	26.32	40.91	42.86	47.61	
	GLM-4	34.88	21.65	5.88	23.68	21.21	17.14	21.4	55.81	22.68	20.59	31.58	34.85	25.71	30.99	
	Qwen2.5-Coder-32B	72.09	47.42	44.12	28.95	34.85	45.71	45.37	83.72	63.92	64.71	36.84	43.94	62.86	59.11	
	Qwen2.5-Coder-7B	48.84	15.46	17.65	18.42	15.15	22.86	21.41	62.79	27.84	26.47	39.47	19.7	31.43	32.59	
	DeepSeek-V2.5	79.07	52.58	55.88	47.37	45.46	62.86	55.59	1.4	60.82	64.71	50.0	51.51	65.71	61.34	
	DeepSeekCoder-33B	37.21	21.65	20.59	7.89	16.67	11.43	19.81	48.84	34.02	26.47	15.79	25.76	31.43	30.99	
	OpenAI-o1	69.77	60.82	79.41	65.79	53.03	65.71	63.58	81.4	80.41	97.06	65.79	62.12	77.14	76.36	
	OpenAI-o3-mini	58.14	71.13	79.41	55.26	57.57	60.0	64.21	67.44	81.44	85.29	73.68	66.67	68.57	74.44	
	DeepSeek-R1	55.81	46.39	88.24	60.53	45.45	54.29	54.63	79.07	71.13	94.12	73.68	60.6	77.14	73.48	
	Qwen-QwQ	34.88	22.68	50.0	31.58	28.79	17.14	29.07	48.84	35.05	79.41	44.74	39.39	31.43	43.45	
UniTrans	DeepSeek-V3	51.16	40.21	26.47	23.68	42.42	48.57	39.62	69.77	52.58	47.06	31.58	51.51	57.14	52.08	
	DeepSeek-V2.5	48.84	34.02	26.47	10.53	31.82	34.29	31.95	62.79	44.33	50.0	23.68	40.91	51.43	45.05	
	OpenAI-o1	34.88	47.42	44.12	23.68	33.33	28.57	37.38	51.16	64.95	61.76	34.21	46.97	45.71	53.03	
InterTrans	DeepSeek-V3	55.81	43.3	35.29	26.32	39.39	45.71	41.53	67.44	53.61	52.94	36.84	51.52	54.29	53.04	
	DeepSeek-V2.5	46.51	32.99	29.41	13.16	28.78	28.57	30.67	60.47	43.3	55.88	23.68	43.93	42.86	44.73	
	OpenAI-o1	46.51	48.45	50.0	31.58	27.27	42.86	41.2	60.47	63.92	61.76	42.11	42.43	57.14	55.27	
SPDZCoder	GPT-4	83.72	84.54	100	94.74	83.33	74.29	85.94	93.02	92.78	100	97.37	86.36	85.71	92.01	
	GLM-4	74.42	75.26	94.12	92.11	60.61	60.0	74.44	81.40	82.47	97.06	94.74	69.69	71.43	81.47	
	DeepSeek-V2.5	90.7	83.51	97.06	81.58	77.27	62.86	82.11	97.67	86.6	100	92.11	80.3	71.43	87.22	
	DeepSeek-V3	86.05	83.51	100	97.37	83.33	68.57	85.63	95.35	90.72	100	97.37	87.88	77.14	91.05	
	OpenAI-o1	86.05	92.78	97.06	86.84	78.79	77.14	86.9	95.35	95.88	97.06	89.47	84.85	82.86	91.38	
	DeepSeek-R1	88.37	91.75	100	89.47	80.30	80.00	88.18	90.70	96.91	100	92.11	87.88	88.57	92.97	

program through test cases, enabling early termination. Consistent with their setting, we employ Rust, JS, C++, and Go as intermediate programming languages.

The baseline methods use the following backbone LLMs. For API Doc, we employ recent and high-performing general-purpose, code, and reasoning LLMs. Specifically, we adopt the following general-purpose LLMs: GPT-4 [51], DeepSeek-V3 [13], and GLM-4 [19]; the following code LLMs: Qwen2.5-Coder-7B and 32B [27], DeepSeek-V2.5 [12], and DeepSeek Coder-33B [23]; and the following reasoning LLMs: OpenAI o1 and o3-mini [50], DeepSeek-R1 [14], and Qwen-QwQ [52]. For UniTrans [66] and InterTrans [46], we replace their original LLM backbones with more advanced alternatives: DeepSeek-V3, DeepSeek-V2.5, and OpenAI o1.

4.4.2 Evaluation Metric. The pass@ k metric [36] measures functional correctness, where k code samples are generated per problem, and if any of the k samples passes a given set of unit tests, the problem is deemed solved [9]. The metric reports the proportion of successfully solved problems. Consistent with mainstream work [5, 9, 16, 42, 69], we adopt this metric to evaluate whether the generated MP-SPDZ code is functionally correct. Specifically, we report **pass@1** and **pass@2** for SPDZCoder on SPDZEval.

5 Evaluation and Analysis

In this section, we comprehensively evaluate SPDZCoder and report the experimental results of the research questions.

5.1 RQ1. Effectiveness of SPDZCoder.

We evaluate the effectiveness of SPDZCoder on SPDZEval against strong baselines that employ general-purpose, code, and reasoning LLMs as backbones, with results shown in Table 5. We also report pass@1 and pass@2 for

SPDZCoder, together with 95% confidence intervals across different backbone models, in Table 6. Our observations are fourfold:

(1) SPDZCoder consistently surpasses the most recently advanced baselines in pass@1 and pass@2 by a large margin across different LLM backbones. The superiority remains robust under the reported 95% confidence intervals. For example, SPDZCoder with GPT-4 as its backbone achieves an overall functional correctness of 85.94% (95% CI: [82.11, 89.46]) and 92.01% (95% CI: [88.82, 94.89]) for pass@1 and pass@2, respectively, whereas the best-performing baseline attains 63.58% and 76.36%.

SPDZCoder’s advantage over the baselines can be attributed to two main factors. First, SPDZCoder decomposes the translation task into a refactoring stage and a generation stage, where semantic-expressing differences at different levels are separately addressed. In particular, the refactoring stage reduces the difficulty of the subsequent generation process, as the CFP code consists solely of basic statements and functions while satisfying the data-obliviousness requirement. This enables the target MP-SPDZ code to be generated in a single step. Second, SPDZCoder incorporates error messages returned by test-case execution as feedback, guiding LLMs to rectify incorrectly translated programs, as discussed later in § 5.2.

Table 6. Functional correctness (pass@1 and pass@2) of SPDZCoder across different LLM backbones, reported with 95% confidence intervals. All results are reported as percentages (%).

	GPT-4	GLM-4	DeepSeek-V2.5	DeepSeek-V3	OpenAI-o1	DeepSeek-R1
pass@1	85.94 ([82.11, 89.46])	74.44 ([69.33, 79.23])	82.11 ([77.64, 86.26])	85.63 ([81.47, 89.46])	86.90 ([83.07, 90.73])	88.18 ([84.35, 91.69])
pass@2	92.01 ([88.82, 94.89])	81.47 ([77.00, 85.62])	87.22 ([83.39, 90.73])	91.05 ([87.86, 93.93])	91.38 ([88.18, 94.26])	92.97 ([90.10, 95.85])

(2) Together with the direct translation results shown earlier in Figure 1, we observe that API Doc modestly improves LLM performance. For example, when using the reasoning LLM DeepSeek-R1 as the backbone, API Doc achieves pass@1 and pass@2 scores of 64% and 75%, respectively, whereas direct translation achieves only 42% and 54%. This improvement may stem from API documentation providing LLMs with coarse-grained domain knowledge through long-context reasoning over MP-SPDZ APIs. These results suggest that effectively supplying MP-SPDZ knowledge to LLMs is important for translation.

(3) Similarly, together with the direct translation results, we observe that UniTrans and InterTrans provide almost no enhancement for Python-to-MP-SPDZ code translation. Specifically, UniTrans, using OpenAI o1 as the backbone, yields only a 2.23% absolute improvement in pass@1 and no improvement in pass@2. InterTrans, with the same backbone, yields only 6.05% and 0.63% absolute improvements in pass@1 and pass@2, respectively. We conjecture that this is because these approaches primarily target widely used programming languages, for which extensive online resources exist. Consequently, LLMs have already been well trained on these languages and exhibit strong baseline translation capabilities, allowing such methods to further enhance performance in those settings. However, MP-SPDZ code and parallel corpora for ⟨Python, MP-SPDZ⟩ are scarce, meaning that the LLMs lack task-specific knowledge for MP-SPDZ. As a result, these methods cannot effectively perform Python-to-MP-SPDZ code translation.

(4) The performance varies slightly across LLM backbones. When adopting the reasoning-oriented LLM DeepSeek-R1 as the backbone, SPDZCoder achieves overall correctness of 88.18% and 92.97% in pass@1 and pass@2, respectively, indicating that reasoning LLMs can provide an incremental improvement over general-purpose LLMs. In contrast, replacing GPT-4 with a considerably weaker backbone model, such as GLM-4-9B, as its backbone leads to absolute drops of 11.5% and 10.54% in pass@1 and pass@2, although SPDZCoder still surpasses baselines that employ substantially more capable LLMs as backbones.

5.2 RQ2. Impact of Refactoring and Repair Stages in SPDZCoder.

We evaluate the contributions of SPDZCoder’s refactoring and repair stages through ablation studies, with the results summarized in Table 7. For the refactoring stage, we observe that removing it causes the overall functional correctness to drop drastically to 33.23% and 45.37% in pass@1 and pass@2, respectively, highlighting the importance of separately addressing semantic differences at different levels in SPDZCoder.

For the repair stage, adding it improves the overall functional correctness from 73.16% and 83.39% to 85.94% and 92.01%, yielding absolute gains of 12.78% and 8.62% in pass@1 and pass@2, respectively. These results indicate that runtime error messages provide useful feedback that helps SPDZCoder repair incorrectly translated programs. Additionally, SPDZCoder still outperforms all baselines even without the repair stage.

Table 7. Functional correctness of SPDZCoder under different ablations of the refactoring and repair stages. All results are obtained with GPT-4 as the backbone model and reported as percentages. Performance drop is calculated as the absolute difference.

Setting	pass@1								pass@2						
	array	loop	branch	math	numpy	syntax	overall (Δ)		array	loop	branch	math	numpy	syntax	overall (Δ)
SPDZCoder	83.72	84.54	100	94.74	83.33	74.29	85.94		93.02	92.78	100	97.37	86.36	85.71	92.01
w/o refactoring	58.14	29.9	20.59	21.05	34.85	34.29	33.23 (52.7%↓)		65.12	41.24	41.18	31.58	43.94	54.29	45.37 (46.6%↓)
w/o repair	74.42	72.16	85.29	78.95	71.21	60.0	73.16 (12.8%↓)		90.7	84.54	85.29	92.11	78.79	68.57	83.39 (8.6%↓)

Moreover, to better understand the failure modes of SPDZCoder, we conduct an error taxonomy analysis. We inspect all translation failures and categorize them into several classes: (1) *Type mismatch or misuse*, referring to incorrect use of plaintext types where MP-SPDZ secret types are required, or overly restrictive use of specific secret types; (2) *Function or method misuse*, including incorrect use of type conversions, operators, initializers, functions, methods, or control structures, as well as invoking nonexistent functions or methods; (3) *Package misuse or missing arguments*, referring to importing incorrect packages, using types/functions/classes that do not exist in the imported package, omitting required function arguments, or referencing undefined variables; (4) *Wrong functionality*, where the generated code contains logical errors; (5) *Others*, such as leaving placeholders unimplemented or triggering index errors.

For each error category, we report the number of occurrences before and after the repair stage, together with the repair success rate. As shown in Table 8, the distribution of error types before the repair stage is relatively balanced overall. After the repair stage, we observe a clear divergence in repair effectiveness across error categories. In particular, package and/or variable misuse and wrong functionality errors are corrected most effectively, achieving repair success rates of 61% and 67%, respectively, whereas the other error categories remain difficult to repair. These results indicate that the repair stage is more effective at resolving package- and variable-related issues and certain logic errors, while errors related to type inference and method misuse are comparatively harder to correct.

To illustrate how the repair stage rectifies incorrect function logic, we provide an example in Figure 4. In this example, although no compilation errors occur, the feedback component identifies the faulty logic in its generated `bubble_sort` function and successfully restores correct functionality by adjusting the assignment statement.

Table 8. Error taxonomy of SPDZCoder. Pre-Rep. and Post-Rep. denote the number of occurrences of each error category that are corrected before and after the repair stage, respectively. Prop. denotes the repair success rate for each error category.

Error Category	Pre-Rep.	Post-Rep.	Prop.
Type mismatch or misuse	16	12	25%
Function or method misuse	24	14	42%
Package or variable misuse	23	9	61%
Wrong functionality	18	6	67%
Others	3	2	33%
Total	84	43	49%

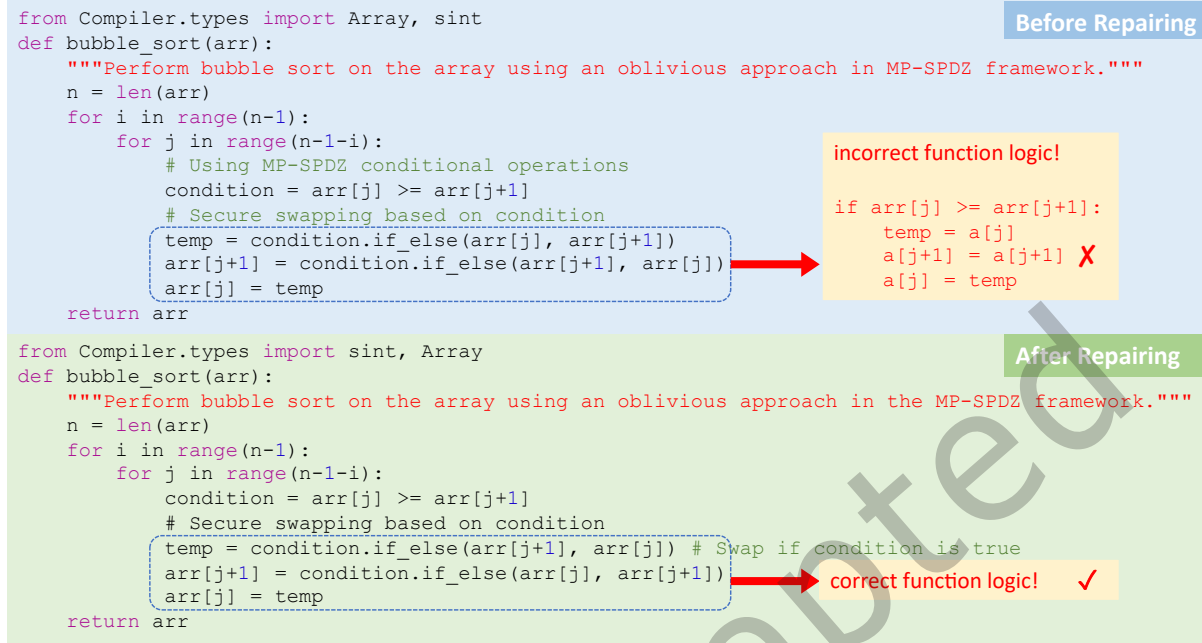


Fig. 4. Example of the repair component correcting the faulty logic in a bubble sort function.

Finally, we analyze the contribution of the self-reflection to correcting two name-related error types: *Function or method misuse* and *Package or variable misuse* since it mainly targets hallucinations of module and function names. Specifically, before self-reflection, the numbers of these two error types are 29 and 26, respectively; after self-reflection, they decrease to 24 and 23, respectively; these latter numbers correspond to those reported in Table 8. These results show that self-reflection modestly contributes to reducing name-related translation errors.

5.3 RQ3. Evaluating the Accuracy of Python Code Refactoring in SPDZCoder.

A high accuracy in the refactoring stage is important for establishing a strong foundation for the subsequent generation stage. Accordingly, we evaluate the correctness of the refactoring stage in SPDZCoder. Specifically, GPT-4 is used as the default backbone of SPDZCoder, and we further replace it with GLM-4, DeepSeek-V2.5, and DeepSeek-V3 to examine the generality of the refactoring stage across different LLM backbones. We then use GPT-4-turbo to generate five test cases for each Python code sample in SPDZEval, and evaluate the functional correctness of the generated CFP code using Python unit tests, with the results shown in Table 9.

As shown in Table 9, GPT-4 performs the refactoring task effectively, achieving an overall correctness of 91.24% in pass@1 and 95.26% in pass@2. Replacing GPT-4 with weaker models, such as GLM-4 and DeepSeek-V2.5, results in a slight decline in accuracy. For instance, when GLM-4 serves as the backbone model, the overall

Table 9. Overall accuracy of refactoring stage using various LLMs as backbone models. SPDZCoder uses GPT-4 as its backbone model.

	pass@1	pass@2
SPDZCoder	91.24	95.26
with GLM-4	85.77	93.43
with DeepSeek-V2.5	90.88	91.61
with DeepSeek-V3	91.24	95.26

pass@1 and pass@2 scores drop to 85.77% and 93.43%, respectively. Meanwhile, DeepSeek-V3 achieves the same performance as GPT-4, suggesting comparable refactoring capabilities.

Together with the earlier results in Table 5, we further observe that although GLM-4 achieves refactoring correctness comparable to GPT-4, its translation performance is not equivalently comparable. This discrepancy arises because SPDZCoder’s translation process requires CFPs to be both functionally correct and structurally compliant with refactoring rules, whereas the latter cannot be automatically verified. To investigate further, we manually inspected the CFPs generated by GLM-4 and found that they sometimes did not strictly adhere to the refactoring rule prompts. Common deviations included failing to utilize the provided basic nonlinear mathematical functions to rewrite advanced ones and, albeit rarely, violating the obliviousness requirement. This observation suggests that strong in-context learning and instruction-following capabilities are required for SPDZCoder’s backbone.

5.4 RQ4. What are the Token Usage, Runtime Cost, and Token Savings Achieved through Rule Pattern Matching?

We first measure the average token consumption of SPDZCoder in comparison with the baselines, presenting the results in Figure 5. As shown in the figure, SPDZCoder does not incur a significant increase in prompt token consumption compared with API Doc, the best-performing baseline. Specifically, SPDZCoder consumes 10,343 prompt tokens and 1,824 completion tokens per code example. Additionally, removing the repair stage reduces prompt token consumption to 6,377, which is comparable to API Doc.

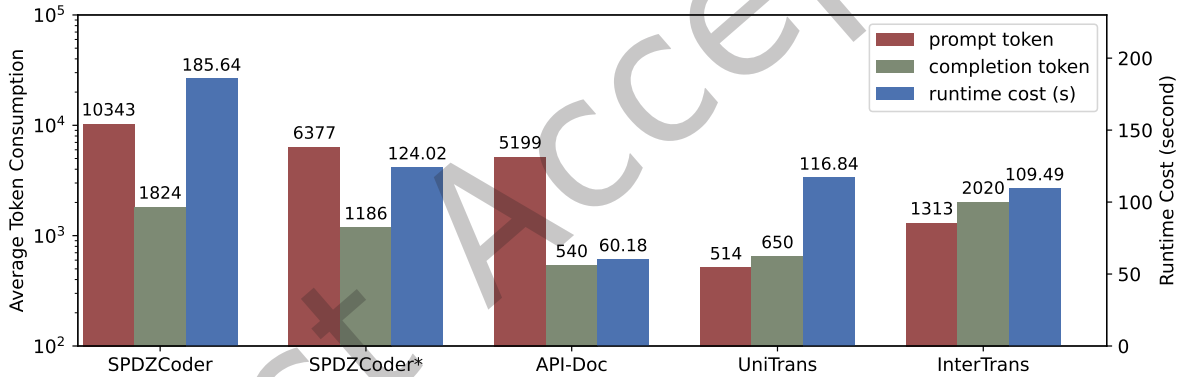


Fig. 5. Average token consumption and runtime cost of SPDZCoder and baseline methods. SPDZCoder* denotes the variant without the repair stage. The left y-axis shows the average token consumption on a logarithmic scale, while the right y-axis reports the corresponding execution time cost in seconds.

We then evaluate the runtime overhead of SPDZCoder against the same baselines. As shown in Figure 5, SPDZCoder has a longer execution time than all baseline methods, with an average runtime of 185.64 seconds. This overhead mainly stems from the additional repair stage. In particular, removing the repair stage, denoted by SPDZCoder*, reduces the runtime to 124.02 seconds, making it comparable to the UniTrans and InterTrans baselines. Overall, while the baseline methods are faster, both SPDZCoder and SPDZCoder* achieve substantially higher functional correctness at the cost of moderately higher execution time.

Finally, we quantify the token savings achieved by our pattern-matching strategies in the refactoring and generation stages. The savings are computed as the difference between the total token count of all rule prompts and the average prompt token consumption per code example after applying pattern matching.

Table 10. Prompt token consumption with and without pattern-matching strategies in the refactoring and generation stages.

Setting	Refactoring Stage	Generation Stage
w/o pattern matching	4355	8973
w/ pattern matching	1567	4810

As shown in Table 10, employing pattern matching significantly reduces prompt token consumption, improves translation efficiency, and prevents unnecessary rules from being applied. Pattern-matching strategies not only improve resource efficiency but also allow us to incorporate a larger set of designed rules into SPDZCoder.

6 Discussion

6.1 Threats to Validity

Internal Validity. The threats to internal validity are as follow: (1) One potential threat is parameter selection, including seed, temperature, and top-p. To mitigate this, we uniformly set these values across all LLMs. While these settings may not be optimal for any specific LLM, they do not impact our ability to analyze performance improvements over direct translation introduced by different methods. (2) Data leakage is another possible concern. However, as it affects all evaluated methods equally, the relative comparisons remain valid. Moreover, publicly available training data typically contain limited parallel corpora, particularly when the target programming language involves privacy-preserving computation. (3) A third threat is the choice of LLM backbone. While different LLMs may yield varying absolute performance values, the comparative evaluation between SPDZCoder and baseline methods remains unaffected.

External Validity. The first threat to external validity concerns the quality and representativeness of the SPDZEval benchmark dataset. To mitigate quality-related threat, we involved an undergraduate student with a strong programming background in constructing the dataset, and the final dataset was manually inspected by development engineers to ensure its quality. The functions in the dataset vary in complexity and diversity. However, although SPDZEval includes several MPC-oriented micro-kernels, such as secure numerical computations, secure tree inference, and secure logistic regression, the benchmark still primarily focuses on Python programming tasks. As a result, it may not fully capture the diversity and complexity of real-world privacy-preserving computation workloads implemented in MP-SPDZ. Future work may extend SPDZEval with more tasks that directly target privacy-preserving computations and explicitly require MP-SPDZ-specific capabilities.

The second threat lies in the choice of experimental models. To reduce the risk that our findings are specific to a particular model family, we evaluated SPDZCoder on representative general-purpose, code, and reasoning LLMs. In addition, we used a consistent prompt template for SPDZCoder across all LLM backbones to ensure fair comparisons.

The third threat is the limited diversity of participating experts. Our expert recruitment process involved four MPC experts from the same organization, which may limit the diversity of perspectives represented during knowledge collection. To mitigate this risk, we intentionally recruited experts with diverse educational and professional backgrounds, including both academic and industry experience. Nevertheless, the resulting semantic-expressing discrepancy set may still implicitly reflect development conventions or practices specific to that organization.

6.2 Transpilation Techniques for Privacy-Computing Code Translation.

Transpilation typically performs code translation by applying rule-based, surface-level syntactic mappings between two programming languages, without requiring deep semantic reasoning about the code. Therefore, such

approaches primarily target programming language pairs with similar structural correspondences and semantic models. In our literature review, we identify only three transpilers, all of which operate in plaintext settings and require nontrivial engineering effort to build: C2Rust [28], CxGo [21], and Java2C# [30]. For these language pairs, rule-based syntactic mappings used by transpilers are generally feasible, but may still involve multiple transformation passes and manual cleanup.

In the context of translating Python into MP-SPDZ, many required transformations depend on understanding the underlying semantics. For example, as discussed earlier, plaintext `quicksort` cannot be translated directly into MP-SPDZ. A correct MP-SPDZ implementation must first recognize that the code block performs sorting and then restructure the entire computation workflow into an oblivious algorithm, such as oblivious radix sort. Another example is presented in change ④ of Figure 2. Traditional transpilers cannot perform such transformation and would systematically fail in this setting. Consequently, we do not consider transpilers as competitive baselines for Python-to-MP-SPDZ translation.

Beyond the choice of baseline space, a related but distinct question concerns the internal design of the generation stage in SPDZCoder. In this stage, the required transformations could, in principle, be implemented using transpilers. However, an LLM enables these transformations to be performed in an end-to-end manner, integrating multiple syntactic transformations within a single step. Moreover, the LLM-based approach is relatively easier to extend to new libraries and emerging patterns, and is well suited to support future extensions such as class-level translation. We therefore adopt a unified LLM-based generation design.

6.3 The Potential Benefits of Fine-tuning with Parallel Data.

Effective fine-tuning requires a large and diverse parallel corpus between Python and MP-SPDZ to achieve robust generalization. Such data is currently scarce for this task, and constructing it would be labor-intensive. Nevertheless, to illustrate the potential impact of fine-tuning when a certain amount of parallel data is available, we fine-tune three representative code models, namely Yi-Coder-9B-Chat [2], CodeGeeX4-All-9B [71], and Qwen2.5-Coder-32B-Instruct [27], using a subset of SPDZEval.

Notably, both Yi-Coder-9B-Chat and CodeGeeX4-All-9B rank among the strongest publicly available code-generation models under 10B parameters on recent benchmarks [29, 70, 73]. In particular, CodeGeeX4-All-9B is a multilingual model continually trained on the GLM-4-9B family, which substantially enhances its code-generation capability. For fair comparison, we report the performance of the three fine-tuned models against SPDZCoder with GLM-4-9B as its backbone, rather than comparing against much larger models such as GPT-4 or DeepSeek-V3.

We use one-third and one-half of the SPDZEval benchmark for supervised fine-tuning and evaluate the resulting models on the remaining two-thirds or one-half, respectively. Formally, let the full dataset be denoted as $\mathcal{D}$, and we fine-tune models on two types of training splits, $\mathcal{D}_{\text{SFT}}^*$ and $\mathcal{D}_{\text{SFT}}^\dagger$, where the former is obtained by randomly sampling one-third of the data within each split of $\mathcal{D}$ and the latter by randomly sampling one-half of the data within each split. The evaluation sets are their complements, namely $\mathcal{D}_{\text{eval}}^* = \mathcal{D} \setminus \mathcal{D}_{\text{SFT}}^*$ and $\mathcal{D}_{\text{eval}}^\dagger = \mathcal{D} \setminus \mathcal{D}_{\text{SFT}}^\dagger$, respectively. Table 11 summarizes the fine-tuning results, and we highlight the main observations below.

First, fine-tuning consistently improves the translation performance of all three vanilla LLMs. For example, when fine-tuned on $\mathcal{D}_{\text{SFT}}^*$, CodeGeex-9B achieves an overall accuracies of 46.84% and 51.23% in pass@1 and pass@2 on $\mathcal{D}_{\text{eval}}^*$, respectively, significantly surpassing the API-doc-based long-text reasoning approach. This demonstrates that parallel supervision is effective for enhancing Python-to-MP-SPDZ translation. However, even with fine-tuning, these models still fall considerably short of SPDZCoder with GLM-4-9B as its backbone, which achieves an overall functional correctness of 77.57% and 83.92% in pass@1 and pass@2, respectively. Using the larger training split $\mathcal{D}_{\text{SFT}}^\dagger$ yields qualitatively consistent findings.

Second, the extent of improvement varies with model size. For smaller models, fine-tuning yields large gains, outperforming the API Doc baseline by over 20% on average. For the mid-sized model, Qwen2.5-Coder-32B-Instruct, the relative advantage of fine-tuning becomes less pronounced, likely because its stronger long-context reasoning allows it to better exploit documentation embedded in the prompt. We observe similar trends under both the $\mathcal{D}_{\text{SFT}}^*$ and $\mathcal{D}_{\text{SFT}}^\dagger$ training settings, suggesting a scale-dependent effect: smaller models benefit more from explicit parallel supervision, whereas larger ones can partially compensate through long-context reasoning.

Finally, increasing the amount of fine-tuning data, from $\mathcal{D}_{\text{SFT}}^*$ to $\mathcal{D}_{\text{SFT}}^\dagger$, further improves the performance of all three fine-tuned models, indicating that access to more high-quality parallel data can further strengthen the effectiveness of supervised fine-tuning.

Table 11. Functional correctness (pass@1 and pass@2) of supervised fine-tuning compared with API Doc and SPDZCoder. SPDZCoder is evaluated with GLM-4-9B as its backbone. Qwen2.5-Coder represents the Qwen2.5-Coder-32B-Instruct model. All evaluation results are presented in percentage format. The best results are highlighted in **bold**.

Method	pass@1								pass@2							
	array	loop	branch	math	numpy	syntax	overall		array	loop	branch	math	numpy	syntax	overall	
<i>Fine-tune on $\mathcal{D}_{\text{SFT}}^*$ and evaluate on $\mathcal{D}_{\text{eval}}^*$</i>																
SFT	Yi-Coder-9B	78.57	48.44	59.09	32.0	13.96	43.48	43.9	2.14	56.25	59.09	40.0	16.26	47.83	48.78	
	CodeGeeX4-All-9B	89.29	56.25	59.09	28.0	16.33	34.78	46.84	89.29	67.19	59.09	32.0	18.64	34.78	51.23	
	Qwen2.5-Coder	85.71	60.94	63.64	40.0	21.04	34.78	50.76	89.29	70.31	63.64	60.0	25.69	43.48	58.56	
API Doc	Yi-Coder-9B	35.71	23.44	9.09	12.0	13.85	34.78	21.44	46.43	28.12	22.73	32.0	16.2	43.48	29.74	
	GLM-4-9B	39.29	21.88	4.55	24.0	23.22	17.39	22.43	57.14	23.44	18.18	32.0	41.79	26.09	32.67	
	Qwen2.5-Coder	64.29	48.44	40.91	28.0	37.17	47.83	44.87	78.57	68.75	59.09	36.0	48.81	60.87	59.99	
SPDZCoder w. GLM-4-9B		78.57	79.69	95.45	92.0	65.17	60.87	77.57	82.14	87.5	100.0	92.0	72.16	73.91	83.92	
<i>Fine-tune on $\mathcal{D}_{\text{SFT}}^\dagger$ and evaluate on $\mathcal{D}_{\text{eval}}^\dagger$</i>																
SFT	Yi-Coder-9B	85.71	60.42	70.59	68.42	21.21	29.41	54.19	85.71	64.58	76.47	73.68	30.3	47.06	60.64	
	CodeGeeX4-All-9B	80.95	58.33	64.71	52.63	33.33	29.41	52.9	80.95	64.58	70.59	73.68	45.46	41.18	61.94	
	Qwen2.5-Coder	61.9	77.08	47.06	84.21	30.3	52.94	60.0	76.19	87.5	58.82	84.21	54.54	52.94	71.61	
API Doc	Yi-Coder-9B	23.81	18.75	11.76	21.05	21.21	23.53	20.0	38.1	22.92	17.65	36.84	21.21	35.29	27.1	
	GLM-4-9B	28.57	27.08	5.88	15.79	27.27	11.76	21.93	52.38	27.08	29.41	31.58	36.36	17.65	32.26	
	Qwen2.5-Coder	71.43	50.0	47.06	36.84	45.46	41.18	49.03	85.71	58.33	76.47	42.11	51.52	52.94	60.0	
SPDZCoder w. GLM-4-9B		80.95	83.33	94.12	84.21	48.48	52.94	73.55	80.95	87.5	100.0	89.47	63.64	64.71	80.65	

6.4 Trade-offs Between Manual MP-SPDZ Programming and Automatic Translation Pipelines.

An alternative to SPDZCoder is to train developers to write MP-SPDZ code. However, this approach introduces cognitive and engineering overhead in practice. MP-SPDZ differs from conventional programming languages in its data obliviousness and the limited availability of mature high-level libraries. Therefore, developers require considerable domain-specific training to efficiently implement correct MP-SPDZ code.

In contrast, leveraging LLMs to translate existing Python code into MP-SPDZ can improve developer productivity. Consistent with prior LLM-based code generation methods, SPDZ coder does not aim to eliminate human involvement or automate deployment, but targets to better harness LLMs for assisting code development.

Reviewing LLM-generated MP-SPDZ code is less labor-intensive than manually writing it from scratch, while still allowing developers to enforce domain-specific correctness and privacy-preservation requirements through inspection and testing. Moreover, manually preparing MP-SPDZ test cases in the optional repair stage is less costly than full-scale manual coding. Test cases typically encode input-output behavior and can be reused across iterations, leading to a much lower marginal cost. Finally, Addressing the unreliability of LLM-generated test

cases in this task or opting not to use the optional repair stage can further reduce the human effort required by the current approach.

6.5 Evaluation Stability under Temperature-Based Stochastic Decoding

In SPDZCoder, we adopt stochastic decoding to generate diverse candidate outputs for pass@k estimation. A natural concern for reproducibility is whether the evaluation results of SPDZCoder remain stable under temperature-based stochastic decoding. To examine this, we perform an evaluation stability analysis through repeated runs with the temperature fixed at 0.7. Specifically, we employ DeepSeek-V3 as the backbone LLM and repeat the evaluation three times. The results are summarized in Table 12.

We find that both pass@1 and pass@2 exhibit only minor fluctuations across runs, and the variations fall well within the confidence intervals reported in Table 6. For example, the overall pass@2 score varies within a narrow range of roughly $\pm 1.6\%$ across repeated runs. These observations indicate that the decoding randomness does not undermine reproducibility.

Table 12. Functional correctness (pass@1 and pass@2) of SPDZCoder evaluated over repeated runs with DeepSeek-V3 as the backbone model. All results are presented in percentage format (%). Δ is calculated as the absolute difference.

Run ID	pass@1							pass@2						
	array	loop	branch	math	numpy	syntax	overall (Δ)	array	loop	branch	math	numpy	syntax	overall (Δ)
0	86.05	83.51	100	97.37	83.33	68.57	85.63	95.35	90.72	100	97.37	87.88	77.14	91.05
1	90.7	84.54	97.06	84.21	77.27	80.0	84.67 (0.96%↓)	95.35	91.75	97.06	94.74	84.85	85.71	91.05 (0%)
2	90.7	85.57	91.18	89.47	78.79	74.29	84.67 (0.96%↓)	95.35	92.78	94.12	100	87.88	88.57	92.65 (1.6%↑)
3	86.05	81.44	94.12	86.84	78.79	74.29	82.75 (2.88%↓)	93.02	91.75	97.06	89.47	84.85	80.0	89.46 (1.59%↓)

6.6 Limitations

The methodology of SPDZCoder is reusable when the source language changes; however, it requires a new analysis of the semantic-expressing differences between that language and MP-SPDZ. Specifically, refactoring rules that address obliviousness requirements can be directly reused, as these constraints remain consistent across source languages. In contrast, the remaining refactoring rules and the generation rule depend on the syntactic complexity of the source language, making them less transferable without modification when the language contains more sophisticated features beyond Python. In addition, examples used in rule prompts need to be adapted to the corresponding source language. These adaptation requirements also reflect inherent limitations of rule-based approaches.

Another limitation is that SPDZCoder is primarily designed to improve the functional correctness of Python-to-MP-SPDZ translation. As a result, our evaluation does not further measure the runtime communication overheads of the translated MP-SPDZ code. Moreover, although we report confidence intervals for pass@k on the whole benchmark, we do not conduct more fine-grained per-split statistical tests. In addition, our functional correctness evaluation relies on author-written test cases, which may introduce potential bias despite our manual inspection and validation efforts. Future work may incorporate more systematic differential or metamorphic testing to further improve the robustness and impartiality of the evaluation.

Finally, our supervised fine-tuning dataset is derived from a subset of SPDZEval and remains relatively small, while no large-scale public parallel corpus exists for this task. As a result, the performance of fine-tuned models may not fully reflect the potential achievable with larger or more diverse supervision, and the generalizability of our observations may therefore be constrained. Expanding the availability of high-quality parallel data remains an important challenge for future work.

7 Related Work

7.1 Large Language Models

General LLMs [3, 7, 10, 13, 19, 32, 49, 51, 53–55, 62] have rapidly advanced, demonstrating remarkable capabilities across diverse tasks. For example, GPT-4o has achieved state-of-the-art performance on code benchmarks [5, 9]. Building on the success of LLMs, many research efforts have leveraged their capabilities to address specific challenges in software engineering [15, 26, 40, 45, 64], including code translation [46, 66].

Code LLMs are trained on multiple programming languages to support code understanding and generation. A collection of studies focuses pre-training code LLMs, such as DeepSeek-V2.5 [12], Qwen2.5-Coder [27], CodeGen [48], CodeGeeX [71], StarCoder [39], and DeepSeek-Coder [23]. Another line of research focuses on fine-tuning them to improve performance or to adapt them to specific coding tasks, such as WizardCoder [44] and InstructCodeT5+ [59].

Reasoning LLMs enhance LLMs' capability in structured logical inference, which is particularly crucial for coding and mathematical tasks. DeepSeek-R1 [14] is trained via large-scale reinforcement learning (RL) as the preliminary step, and it achieves performance comparable to OpenAI-o1 on reasoning tasks. Qwen-QwQ [52] and OpenAI o1 [50] also belong to RLLMs.

Pre-training or fine-tuning LLMs requires large amount of high-quality code corpora, which is scarce in privacy computing. Instead, we employ long-text reasoning to enhance code translation in baselines.

7.2 Code Translation

Code translation approaches generally fall into three categories: transpiler-based, learning-based and LLM-based methods. Transpiler approaches [21, 28, 30] rely on program analysis techniques to convert code between languages but are limited to syntactic transformations. Learning-based methods leverage Transformer-based architectures, requiring training data to develop deep models for either specific or general code translation tasks. Notable works in this category include CodeBERT [18], CuBERT [34], GraphCodeBERT [22], CodeT5 [60] and PLBART [1].

Beyond deep learning, LLMs are employed as backbone to perform code translation. For example, UniTrans [66] incorporates test cases to augment code translation and iteratively repairs bugs for incorrectly translated programs prompted by test case execution results. InterTrans [46] utilizes a planning algorithm (ToCT) to generate candidate translation paths, and leverages intermediate translations to enhance code translation. SPDZCoder also uses LLMs for code translation, but unlike UniTrans and InterTrans, our target programming language is not familiar to LLMs.

7.3 Code Generation Benchmark

Many works were proposed to benchmark the code generation ability of LLMs [4, 5, 8, 9, 16, 25, 38, 42, 69]. The Mostly Basic Programming Problems (MBPP) dataset [5], consisting of 974 programming tasks, measures the ability of LLMs to synthesize short Python programs from natural language descriptions. Human-eval [9] measures functional correctness for synthesizing programs from docstring. Liu et al. proposed an enhanced code synthesis evaluation framework named EvalPlus [42], which augments HumanEval with large amounts of newly produced test-cases. Similar work also includes Multi-HumanEval [4]. More recently, parallel with ClassEval [16], Yu et al. proposed CoderEval [69] to evaluate the effectiveness of code LLMs in generating code beyond only standalone functions. There are also code generation benchmarks for specific purpose, *e.g.*, type inference [47]. SPDZEval differs from them in terms of coding task and target programming language.

8 Conclusion

We propose SPDZCoder, a data-free framework designed to teach LLMs to harness code translation in privacy computing settings. SPDZCoder utilizes expert-defined rules to help LLMs learn the unique ways and constraints of expressing common semantics in MP-SPDZ, without requiring experts to manually generate massive amounts of training data. It addresses this task by considering semantic-expressing differences at various levels, leveraging the refactoring and generation stages to mitigate these differences at high and low levels, respectively. Furthermore, SPDZCoder incorporates execution messages from test cases as feedback, guiding LLMs to rectify incorrectly translated programs. To evaluate SPDZCoder against recent advanced baselines, we introduce SPDZEval, a benchmark dataset tailored for this task. Extensive experimental results demonstrate that SPDZCoder significantly outperforms strong baselines and maintains consistent effectiveness across different LLM backbones. Our work provides new insights into teaching LLMs to synthesize privacy computing code from existing plaintext implementation and suggests that SPDZCoder can facilitate better code translation practices within the privacy computing community. For future work, we aim to extend SPDZCoder to support translation at file-level program granularity. We provide the research artifact for this study⁵, which includes the source code, execution scripts, dataset, and documentation.

Acknowledgments

This work was supported in part by the National Key R&D Program of China under Grant 2023YFC3304802 and the National Natural Science Foundation of China (NSFC) under Grants U2268202 and 62176135. Jia Li was supported by the Beijing Natural Science Foundation under Grant 4264107.

References

- [1] Wasi Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2021. Unified pre-training for program understanding and generation. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*. Association for Computational Linguistics, Online, 2655–2668. doi:10.18653/v1/2021.naacl-main.211
- [2] 01. AI, :, Alex Young, Bei Chen, Chao Li, Chengen Huang, Ge Zhang, Guanwei Zhang, Guoyin Wang, Heng Li, Jiangcheng Zhu, Jianqun Chen, Jing Chang, Kaidong Yu, Peng Liu, Qiang Liu, Shawn Yue, Senbin Yang, Shiming Yang, Wen Xie, Wenhao Huang, Xiaohui Hu, Xiaoyi Ren, Xinyao Niu, Pengcheng Nie, Yanpeng Li, Yuchi Xu, Yudong Liu, Yue Wang, Yuxuan Cai, Zhenyu Gu, Zhiyuan Liu, and Zonghong Dai. 2025. Yi: Open foundation models by 01.AI. arXiv:2403.04652 <https://arxiv.org/abs/2403.04652>
- [3] Anthropic. 2023. The Claude 3 model family: Opus, Sonnet, Haiku. <https://www-cdn.anthropic.com>
- [4] Ben Athiwaratkun, Sanjay Krishna Gouda, Zijian Wang, Xiaopeng Li, Yuchen Tian, Ming Tan, Wasi Uddin Ahmad, Shiqi Wang, Qing Sun, Mingyue Shang, Sujun Kumar Gonugondla, Hantian Ding, Varun Kumar, Nathan Fulton, Arash Farahani, Siddhartha Jain, Robert Giaquinto, Haifeng Qian, Murali Krishna Ramanathan, and Ramesh Nallapati. 2023. Multi-lingual evaluation of code generation models. In *Proceedings of the Eleventh International Conference on Learning Representations (Kigali, Rwanda) (ICLR '23)*. OpenReview.net. <https://openreview.net/pdf?id=Bo7eeXm6An8>
- [5] Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and Charles Sutton. 2021. *Program synthesis with large language models*. arXiv:2108.07732 <https://arxiv.org/abs/2108.07732>
- [6] Peter Bogetoft, Dan Lund Christensen, Ivan Damgård, Martin Geisler, Thomas Jakobsen, Mikkel Kroigaard, Janus Dam Nielsen, Jesper Buus Nielsen, Kurt Nielsen, Jakob Pagter, Michael Schwartzbach, and Tomas Toft. 2009. Secure multiparty computation goes live. In *Financial Cryptography and Data Security (Accra Beach, Barbados) (FC '09)*. Springer-Verlag, Berlin, Heidelberg, 325–343. https://doi.org/10.1007/978-3-642-03549-4_20
- [7] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, and Gretchen Krueger. 2020. Language models are few-shot learners. In *Advances in Neural Information Processing Systems 33 (Virtual Event) (NeurIPS '20)*, Hugo Larochelle, Marc'Aurelio Ranzato, Raia Hadsell, Maria-Florina Balcan, and Hsuan-Tien Lin (Eds.). <https://proceedings.neurips.cc/paper/2020>
- [8] Federico Cassano, John Gouwar, Daniel Nguyen, Sydney Nguyen, Luna Phipps-Costin, Donald Pinckney, Ming-Ho Yee, Yangtian Zi, Carolyn Jane Anderson, Molly Q Feldman, Arjun Guha, Michael Greenberg, and Abhinav Jangda. 2023. MultiPL-E: A scalable

⁵<https://doi.org/10.5281/zenodo.17974882>.

- and polyglot approach to benchmarking neural code generation. *IEEE Transactions on Software Engineering* 49, 7 (2023), 3675–3691. doi:10.1109/TSE.2023.3267446
- [9] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, and Jared Kaplan. 2021. *Evaluating large language models trained on code*. arXiv:2107.03374 <https://arxiv.org/abs/2107.03374>
- [10] Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, and Hyung Won Chung. 2022. *PaLM: Scaling language modeling with pathways*. arXiv:2204.02311 <https://arxiv.org/abs/2204.02311>
- [11] Badhan Chandra Das, M. Hadi Amini, and Yanzhao Wu. 2025. Security and privacy challenges of large language models: A survey. *ACM Comput. Surv.* 57, 6, Article 152 (Feb. 2025), 39 pages. doi:10.1145/3712001
- [12] DeepSeek-AI. 2024. *DeepSeek-V2: A strong, economical, and efficient mixture-of-experts language model*. arXiv:2405.04434 <https://arxiv.org/abs/2405.04434>
- [13] DeepSeek-AI. 2024. *DeepSeek-V3 technical report*. arXiv:2412.19437 <https://arxiv.org/abs/2412.19437>
- [14] DeepSeek-AI, Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, and Qihao Zhu. 2025. DeepSeek-R1: Incentivizing reasoning capability in LLMs via reinforcement learning. *Nature* 645, 8081 (2025), 633–638. doi:10.1038/s41586-025-09422-z
- [15] Yinlin Deng, Chunqiu Steven Xia, Chenyuan Yang, Shizhuo Dylan Zhang, Shujing Yang, and Lingming Zhang. 2024. Large language models are edge-case generators: Crafting unusual programs for fuzzing deep learning libraries. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering (Lisbon, Portugal) (ICSE '24)*. Association for Computing Machinery, New York, NY, USA, 70:1–70:13. doi:10.1145/3597503.3623343
- [16] Xueying Du, Mingwei Liu, Kaixin Wang, Hanlin Wang, Junwei Liu, Yixuan Chen, Jiayi Feng, Chaofeng Sha, Xin Peng, and Yiling Lou. 2024. Evaluating large language models in class-level code generation. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering (Lisbon, Portugal) (ICSE '24)*. Association for Computing Machinery, New York, NY, USA, 81:1–81:13. doi:10.1145/3597503.3639219
- [17] European Union. 2016. Regulation (EU) 2016/679 of the European Parliament and of the Council of 27 April 2016 on the protection of natural persons with regard to the processing of personal data and on the free movement of such data, and repealing Directive 95/46/EC (General Data Protection Regulation) (text with EEA relevance). 88 pages. <https://eur-lex.europa.eu/eli/reg/2016/679/oj>
- [18] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. CodeBERT: A pre-trained model for programming and natural languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, Trevor Cohn, Yulan He, and Yang Liu (Eds.). Association for Computational Linguistics, Online, 1536–1547. doi:10.18653/v1/2020.findings-emnlp.139
- [19] Team GLM, Aohan Zeng, Bin Xu, Bowen Wang, Chenhui Zhang, Da Yin, Diego Rojas, Guanyu Feng, Hanlin Zhao, Hanyu Lai, Hao Yu, and Hongning Wang. 2024. *ChatGLM: A family of large language models from GLM-130B to GLM-4 all tools*. arXiv:2406.12793 <https://arxiv.org/abs/2406.12793>
- [20] O. Goldreich, S. Micali, and A. Wigderson. 1987. How to play ANY mental game. In *Proceedings of the Nineteenth Annual ACM Symposium on Theory of Computing* (New York, New York, USA) (STOC '87). Association for Computing Machinery, New York, NY, USA, 218–229. doi:10.1145/28395.28420
- [21] gotranspile. 2025. *cxgo*. Retrieved August 27, 2026 from <https://github.com/gotranspile/cxgo>
- [22] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, Michele Tufano, Shao Kun Deng, Colin B. Clement, Dawn Drain, Neel Sundaresan, Jian Yin, Daxin Jiang, and Ming Zhou. 2021. GraphCodeBERT: Pre-training code representations with data flow. In *Proceedings of the Ninth International Conference on Learning Representations* (Virtual Event, Austria) (ICLR '21). OpenReview.net. <https://openreview.net/forum?id=jLoC4ez43PZ>
- [23] Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Y. Wu, Y. K. Li, Fuli Luo, Yingfei Xiong, and Wenfeng Liang. 2024. *DeepSeek-Coder: When the large language model meets programming — the rise of code intelligence*. arXiv:2401.14196 <https://arxiv.org/abs/2401.14196>
- [24] Koki Hamada, Dai Ikarashi, Koji Chida, and Katsumi Takahashi. 2014. *Oblivious radix sort: An efficient sorting algorithm for practical secure multi-party computation*. <https://eprint.iacr.org/2014/121>
- [25] Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, and Jacob Steinhardt. 2021. Measuring coding challenge competence with APPS. In *Advances in Neural Information Processing Systems*.
- [26] Yuchao Huang, Junjie Wang, Zhe Liu, Yawen Wang, Song Wang, Chunyang Chen, Yuanzhe Hu, and Qing Wang. 2024. CrashTranslator: Automatically reproducing mobile application crashes directly from stack trace. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering (Lisbon, Portugal) (ICSE '24)*. Association for Computing Machinery, New York, NY, USA, 18:1–18:13. doi:10.1145/3597503.3623298
- [27] Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Kai Dang, et al. 2024. *Qwen2.5-Coder technical report*.
- [28] Immunant. 2026. *C2Rust*. Retrieved August 27, 2026 from <https://github.com/immunant/c2rust>

- [29] Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. 2025. LiveCodeBench: Holistic and contamination free evaluation of large language models for code. In *The Thirteenth International Conference on Learning Representations*. <https://openreview.net/forum?id=chfJYC3iL>
- [30] Java2CSharp Project. 2011. Java 2 CSharp translator for Eclipse. Retrieved August 27, 2026 from <https://sourceforge.net/projects/j2cstranslator/>
- [31] Ziwei Ji, Tiezheng Yu, Yan Xu, Nayeon Lee, Etsuko Ishii, and Pascale Fung. 2023. Towards mitigating LLM hallucination via self reflection. In *Findings of the Association for Computational Linguistics: EMNLP 2023*, Houda Bouamor, Juan Pino, and Kalika Bali (Eds.). Association for Computational Linguistics, Singapore, 1827–1843. doi:10.18653/v1/2023.findings-emnlp.123
- [32] Albert Q. Jiang, Alexandre Sablayrolles, Antoine Roux, Arthur Mensch, Blanche Savary, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Emma Bou Hanna, Florian Bressand, Gianna Lengyel, Guillaume Bour, Guillaume Lample, L  lio Renard Lavaud, Lucile Saulnier, Marie-Anne Lachaux, Pierre Stock, Sandeep Subramanian, Sophia Yang, Szymon Antoniak, Teven Le Scao, Th  ophile Gervet, Thibaut Lavril, Thomas Wang, Timoth  e Lacroix, and William El Sayed. 2024. *Mixtral of experts*. arXiv:2401.04088 <https://arxiv.org/abs/2401.04088>
- [33] Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R Narasimhan. 2024. SWE-bench: Can language models resolve real-world GitHub issues?. In *The Twelfth International Conference on Learning Representations*. <https://openreview.net/forum?id=VTF8yNQM66>
- [34] Aditya Kanade, Petros Maniatis, Gogul Balakrishnan, and Kensen Shi. 2020. Learning and evaluating contextual embedding of source code. In *Proceedings of the 37th International Conference on Machine Learning (Virtual Event) (Proceedings of Machine Learning Research, Vol. 119)*. PMLR, 5110–5121. <http://proceedings.mlr.press/v119/kanade20a.html>
- [35] Marcel Keller. 2020. MP-SPDZ: A versatile framework for multi-party computation. <https://eprint.iacr.org/2020/521> Publication info: Published elsewhere. Minor revision. ACM CCS 2020.
- [36] Sumith Kulal, Panupong Pasupat, Kartik Chandra, Mina Lee, Oded Padon, Alex Aiken, and Percy Liang. 2019. SPoC: Search-based pseudocode to code. In *Advances in Neural Information Processing Systems 32 (Vancouver, BC, Canada) (NeurIPS ’19)*, Hanna M. Wallach, Hugo Larochelle, Alina Beygelzimer, Florence d’Alch  -Buc, Emily B. Fox, and Roman Garnett (Eds.). 11883–11894. <https://proceedings.neurips.cc/paper/2019>
- [37] Chengzhe Lai, Hanyue Zhang, Rongxing Lu, and Dong Zheng. 2024. Privacy-preserving medical data sharing scheme based on two-party cloud-assisted PSI. *IEEE Internet of Things Journal* 11, 9 (2024), 15855–15868. doi:10.1109/JIOT.2024.3350029
- [38] Yuhang Lai, Chengxi Li, Yiming Wang, Tianyi Zhang, Ruiqi Zhong, Luke Zettlemoyer, Scott Wen tau Yih, Daniel Fried, Sida Wang, and Tao Yu. 2022. *DS-1000: A natural and reliable benchmark for data science code generation*. arXiv:2211.11501 <https://arxiv.org/abs/2211.11501>
- [39] Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, and Jenny Chim. 2023. *StarCoder: May the source be with you!* arXiv:2305.06161 <https://arxiv.org/abs/2305.06161>
- [40] Yi Li, Shaohua Wang, and Tien N. Nguyen. 2022. Fault localization to detect co-change fixing locations. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (Singapore Singapore)*. Association for Computing Machinery, New York, NY, USA, 659–671. doi:10.1145/3540250.3549137
- [41] Yi Li and Wei Xu. 2019. PrivPy: General and scalable privacy-preserving data mining. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (Anchorage, AK, USA) (KDD ’19)*, Ankur Teredesai, Vipin Kumar, Ying Li, R  mer Rosales, Evimaria Terzi, and George Karypis (Eds.). Association for Computing Machinery, New York, NY, USA, 1299–1307. doi:10.1145/3292500.3330920
- [42] Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. 2023. Is your code generated by ChatGPT really correct? Rigorous evaluation of large language models for code generation. In *Thirty-seventh Conference on Neural Information Processing Systems*. <https://openreview.net/forum?id=1qv610Cu7>
- [43] Xiaoning Liu, Yifeng Zheng, Xun Yi, and Surya Nepal. 2022. Privacy-preserving collaborative analytics on medical time series data. *IEEE Transactions on Dependable and Secure Computing* 19, 3 (2022), 1687–1702. doi:10.1109/TDSC.2020.3035592
- [44] Ziyang Luo, Can Xu, Pu Zhao, Qingfeng Sun, Xiubo Geng, Wenxiang Hu, Chongyang Tao, Jing Ma, Qingwei Lin, and Daxin Jiang. 2024. WizardCoder: Empowering code large language models with Evol-Instruct. In *Proceedings of the Twelfth International Conference on Learning Representations*.
- [45] Lipeng Ma, Weidong Yang, Bo Xu, Sihang Jiang, Ben Fei, Jiaqing Liang, Mingjie Zhou, and Yanghua Xiao. 2024. KnowLog: Knowledge enhanced pre-trained language model for log understanding. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering (Lisbon, Portugal) (ICSE ’24)*. Association for Computing Machinery, New York, NY, USA, 32:1–32:13. doi:10.1145/3597503.3623304
- [46] Marcos Macedo, Yuan Tian, Pengyu Nie, Filipe R. Cogo, and Bram Adams. 2025. InterTrans: Leveraging transitive intermediate translations to enhance LLM-based code translation. In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. IEEE Computer Society, Los Alamitos, CA, USA.
- [47] Amir M. Mir, Evaldas Lato  skinas, and Georgios Gousios. 2021. ManyTypes4Py: A benchmark Python dataset for machine learning-based type inference. IEEE Computer Society, Los Alamitos, CA, USA, 585–589. doi:10.1109/MSR52588.2021.00079

- [48] Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2023. CodeGen: An open large language model for code with multi-turn program synthesis. In *Proceedings of the Eleventh International Conference on Learning Representations (Kigali, Rwanda) (ICLR '23)*. OpenReview.net. https://openreview.net/pdf?id=iaYcJKpY2B_
- [49] OpenAI. 2024. GPT-4o. <https://openai.com/index/gpt-4o-system-card/>.
- [50] OpenAI. 2025. OpenAI o1. <https://openai.com/o1/>.
- [51] OpenAI, Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altschmidt, Sam Altman, Shyamal Anadkat, and Red Avila. 2023. *GPT-4 technical report*. arXiv:2303.08774 <https://arxiv.org/abs/2303.08774>
- [52] Qwen Team. 2024. QwQ: Reflect deeply on the boundaries of the unknown. <https://qwenlm.github.io/blog/qwq-32b-preview/>
- [53] Gemini Team, Rohan Anil, Sebastian Borgeaud, Yonghui Wu, Jean-Baptiste Alayrac, Jiahui Yu, Radu Soricut, Johan Schalkwyk, and Andrew M. Dai. 2023. *Gemini: A family of highly capable multimodal models*. arXiv:2312.11805 <https://arxiv.org/abs/2312.11805>
- [54] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurelien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. 2023. *LLaMA: Open and efficient foundation language models*. arXiv:2302.13971 <https://arxiv.org/abs/2302.13971>
- [55] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, and Prajjwal Bhargava. 2023. *Llama 2: Open foundation and fine-tuned chat models*. arXiv:2307.09288 <https://arxiv.org/abs/2307.09288>
- [56] United States Congress. 1996. Health Insurance Portability and Accountability Act of 1996. Public Law 104-191, 110 Stat. 2548 (1996). <https://www.govinfo.gov/app/details/PLAW-104publ191>
- [57] w3resource. [n. d.]. w3resource. Retrieved August 27, 2026 from <https://www.w3resource.com/python-exercises/>
- [58] W3Schools. [n. d.]. w3schools. Retrieved August 27, 2026 from <https://www.w3schools.com/python/>
- [59] Yue Wang, Hung Le, Akhilesh Gotmare, Nghi Bui, Junnan Li, and Steven Hoi. 2023. CodeT5+: Open code large language models for code understanding and generation. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*. Houda Bouamor, Juan Pino, and Kalika Bali (Eds.). Association for Computational Linguistics, Singapore, 1069–1088. doi:10.18653/v1/2023.emnlp-main.68
- [60] Yue Wang, Weishi Wang, Shafiq Joty, and Steven C.H. Hoi. 2021. CodeT5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, Marie-Francine Moens, Xuanjing Huang, Lucia Specia, and Scott Wen-tau Yih (Eds.). Association for Computational Linguistics, Online and Punta Cana, Dominican Republic, 8696–8708. doi:10.18653/v1/2021.emnlp-main.685
- [61] Yanli Wang, Yanlin Wang, Suquan Wang, Daya Guo, Jiachi Chen, John Grundy, Xilin Liu, Yuchi Ma, Mingzhi Mao, Hongyu Zhang, and Zibin Zheng. 2026. RepoTransBench: A Real-World Multilingual Benchmark for Repository-Level Code Translation. *IEEE Transactions on Software Engineering* 52, 02 (Feb. 2026), 675–690. doi:10.1109/TSE.2025.3645056
- [62] BigScience Workshop, Teven Le Scao, Angela Fan, Christopher Akiki, Ellie Pavlick, Suzana Ilić, Daniel Hesslow, Roman Castagné, and Alexandra Sasha Luccioni. 2022. *BLOOM: A 176B-parameter open-access multilingual language model*. arXiv:2211.05100 <https://arxiv.org/abs/2211.05100>
- [63] Weixiang Yan, Yuchen Tian, Yunzhe Li, Qian Chen, and Wen Wang. 2023. CodeTransOcean: A comprehensive multilingual benchmark for code translation. In *Findings of the Association for Computational Linguistics: EMNLP 2023*, Houda Bouamor, Juan Pino, and Kalika Bali (Eds.). Association for Computational Linguistics, Singapore, 5067–5089. doi:10.18653/v1/2023.findings-emnlp.337
- [64] Aidan Z. H. Yang, Claire Le Goues, Ruben Martins, and Vincent J. Hellendoorn. 2024. Large language models for test-free fault localization. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering (Lisbon, Portugal) (ICSE '24)*. Association for Computing Machinery, New York, NY, USA, 17:1–17:12. doi:10.1145/3597503.3623342
- [65] Jian Yang, Jiaxi Yang, Ke Jin, Yibo Miao, Lei Zhang, Liqun Yang, Zeyu Cui, Yichang Zhang, Binyuan Hui, and Junyang Lin. 2024. *Evaluating and aligning CodeLLMs on human preference*. arXiv:2412.05210 <https://arxiv.org/abs/2412.05210>
- [66] Zhen Yang, Fang Liu, Zhongxing Yu, Jacky Wai Keung, Jia Li, Shuo Liu, Yifan Hong, Xiaoxue Ma, Zhi Jin, and Ge Li. 2024. Exploring and unleashing the power of large language models in automated code translation. *Proceedings of the ACM on Software Engineering* 1, FSE (July 2024), 1585–1608. doi:10.1145/3660778
- [67] Andrew C. Yao. 1982. Protocols for secure computations. In *23rd Annual Symposium on Foundations of Computer Science (sfcs 1982)*. 160–164. doi:10.1109/SFCS.1982.38 ISSN: 0272-5428.
- [68] Pengcheng Yin, Bowen Deng, Edgar Chen, Bogdan Vasilescu, and Graham Neubig. 2018. Learning to mine aligned code and natural language pairs from Stack Overflow. In *Proceedings of the 15th International Conference on Mining Software Repositories (Gothenburg, Sweden) (MSR '18)*. Association for Computing Machinery, New York, NY, USA, 476–486. doi:10.1145/3196398.3196408
- [69] Hao Yu, Bo Shen, Dezhi Ran, Jiaxin Zhang, Qi Zhang, Yuchi Ma, Guangtai Liang, Ying Li, Qianxiang Wang, and Tao Xie. 2024. CoderEval: A benchmark of pragmatic code generation with generative pre-trained models. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering (Lisbon, Portugal) (ICSE '24)*. Association for Computing Machinery, New York, NY, USA, 37:1–37:12. doi:10.1145/3597503.3623316
- [70] Shudan Zhang, Hanlin Zhao, Xiao Liu, Qinkai Zheng, Zehan Qi, Xiaotao Gu, Yuxiao Dong, and Jie Tang. 2024. NaturalCodeBench: Examining coding performance mismatch on HumanEval and natural user prompts. In *Findings of the Association for Computational*

- Linguistics: ACL 2024*, Lun-Wei Ku, Andre Martins, and Vivek Srikumar (Eds.). Association for Computational Linguistics, Bangkok, Thailand, 7907–7928. doi:10.18653/v1/2024.findings-acl.471
- [71] Qinkai Zheng, Xiao Xia, Xu Zou, Yuxiao Dong, Shan Wang, Yufei Xue, Lei Shen, Zihan Wang, Andi Wang, Yang Li, Teng Su, Zhilin Yang, and Jie Tang. 2023. CodeGeeX: A pre-trained model for code generation with multilingual benchmarking on HumanEval-X. In *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining* (Long Beach, CA, USA) (KDD '23). Association for Computing Machinery, New York, NY, USA, 5673–5684. doi:10.1145/3580305.3599790
- [72] Ming Zhu, Karthik Suresh, and Chandan K Reddy. 2022. Multilingual code snippets training for program translation. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 36. 11783–11790. doi:10.1609/aaai.v36i10.21434
- [73] Terry Yue Zhuo, Vu Minh Chien, Jenny Chim, Han Hu, Wenhao Yu, Ratnadira Widyasari, Imam Nur Bani Yusuf, Haolan Zhan, Junda He, Indraneil Paul, Simon Brunner, Chen GONG, James Hoang, Arnel 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. 2025. BigCodeBench: Benchmarking code generation with diverse function calls and complex instructions. In *The Thirteenth International Conference on Learning Representations*. <https://openreview.net/forum?id=YrycTjllL0>

A Appendix

A.1 Translation Example

We provide a qualitative example below to illustrate how SPDZCoder progressively translates Python code into MP-SPDZ code through the transformation rules introduced in Section 3.2. The example implements an RMSProp update step and demonstrates transformations from the original Python program to Canonical Form Python (CFP), and subsequently from CFP to MP-SPDZ.

In the refactoring stage, SPDZCoder first rewrites Python-specific constructs into simpler and more explicit forms while preserving the original program semantics. As shown in Figure 6, the list comprehension used to initialize `s` is expanded into an explicit `for` loop. Similarly, the loop based on `enumerate(zip(grads, layers))` is transformed into an index-based loop. These transformations remove high-level Python syntactic sugar and expose the underlying data access and control-flow structure, yielding a CFP program that is structurally closer to MP-SPDZ.

In the generation stage, SPDZCoder translates the CFP program into MP-SPDZ code while largely preserving its control-flow structure. The remaining translation mainly involves replacing Python and NumPy abstractions with MP-SPDZ-specific types and operations. For example, floating-point constants are represented using `sfix`, and the state vector is initialized using an `Array` of `sfix` values followed by `assign_all(0)`. NumPy operations are also rewritten using their MP-SPDZ-compatible counterparts: element-wise squaring is expressed through explicit multiplication, while square-root computation is performed using the `sqrt()` operation provided by the MP-SPDZ type. Consequently, after refactoring, the CFP-to-MP-SPDZ transformation becomes largely a structurally aligned translation.

This example highlights the pipeline of SPDZCoder: instead of requiring LLMs to directly bridge the semantic differences between Python and MP-SPDZ in a single step, SPDZCoder first normalizes the source program into CFP and then performs a more localized domain-specific translation.

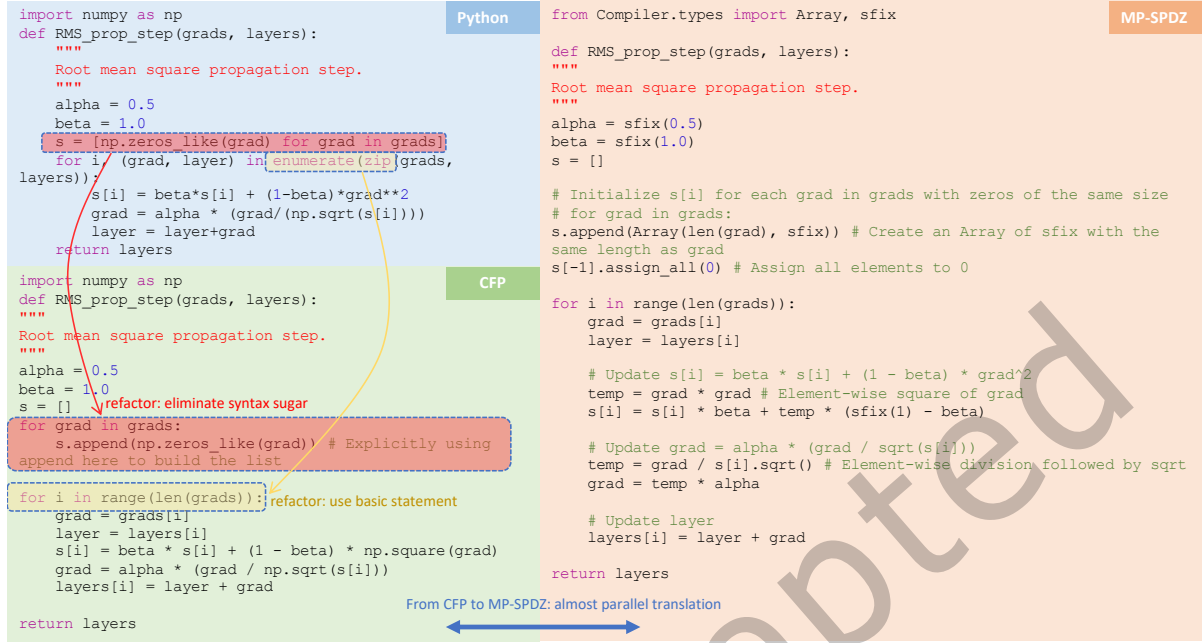


Fig. 6. A translation example of SPDZCoder

Received 16 June 2025; revised 22 May 2026; accepted 18 August 2026