PyLingual: Toward Perfect Decompilation of Evolving High-Level Languages

Josh Wiedemeier[1], Elliot Tarbet[1], Max Zheng[1], Sangsoo Ko[1], Jessica Ouyang[1], Sang Kil Cha[2], and Kangkook Jee[1]

[1]The University of Texas at Dallas
{jdw170000, elliot, Max.Zheng, Sangsoo.Ko, Jessica.Ouyang, Kangkook.Jee}@utdallas.edu
[2]Korea Advanced Institute of Science and Technology
sangkilc@softsec.kaist.ac.kr

2025 IEEE Symposium on Security and Privacy (SP) | 979-8-3315-2236-0/25/$31.00 ©2025 IEEE | DOI: 10.1109/SP61157.2025.00052

Python is one of the most popular programming languages among both industry developers and malware authors. Despite demand for Python decompilers, community efforts to maintain automatic Python decompilation tools have been hindered by Python's aggressive language improvements and unstable bytecode specification. Every year, language features are added, code generation undergoes significant changes, and opcodes are added, deleted, and modified.

Our research aims to integrate Natural Language Processing (NLP) techniques with classical Programming Language (PL) theory to create a Python decompiler that accomodates evolving language features and changes to the bytecode specification with minimal human maintenance effort. PyLingual plugs in data-driven NLP components to a version-agnostic core to automatically absorb superficial bytecode and compiler changes, while leveraging programmatic components for abstract control flow reconstruction. To establish trust in the decompilation results, we introduce a stringent correctness measure based on "perfect decompilation", a statically verifiable refinement of semantic equivalence.

We demonstrate the efficacy of our approach with extensive real-world datasets of benign and malicious Python source code and their corresponding compiled `PYC` binaries. Our research makes three major contributions: *(1)* we present PyLingual, a scalable, data-driven decompilation framework with state-of-the-art support for Python versions 3.6 through 3.12, improving the perfect decompilation rate by an average of 45% over the best results of existing decompiler across four datasets; *(2)* we provide a Python decompiler evaluation framework that verifies decompilation results with perfect decompilation; and *(3)* we launch PyLingual as a public online service.

1. Introduction

Python is an attractive choice for hackers and industry developers alike due to its straightforward development, wide user base, mature ecosystem with pre-built modules, and multi-platform compatibility [1–8]. Programmers aim to budget their time and resources to maximize their productivity , thus demanding infrastructural support to accelerate their development cycles. The growing diversity of computing environments and the desire for custom attack vectors (*i.e.,* crafted for specific targets and scenarios) only heighten such demands. Closed-source Python projects package compiled `PYC` binaries and their dependencies with an interpreter for their target platforms [9, 10] to create a standalone executable. Python decompilers reverse the above process: unpacking the packaged executable to extract the `PYC` binaries [11], disassembling them into bytecode sequences [12], and ultimately recovering the source code [13, 14].

Our work aims to recover Python source code from disassembled bytecode sequences. Compared to traditional binaries, `PYC` binaries contain substantially more information, are more decomposable, and do not contain indirect jumps. These properties trivialize many challenges from traditional binary decompilation. However, Python's unique development model imposes one key challenge that has *prevented the maturation* of community Python decompilation efforts: instruction set instability [13, 14].

Python's bytecode specification is dynamic and constantly evolving, as it is not bound to any underlying hardware architecture, and the language developers eschew forwards and backwards compatibility of the bytecode in favor of design flexibility. With each language version release, opcodes are removed, added, and modified to support new language features and improve the performance of existing language features. For example, the exception handling mechanism has been *reworked twice in the last five years*, with additional code generation changes between the reworks. Further, recent Python versions have been adopting aggressive optimizations [15–18] that impact code generation and control flow structures. The `PYC` bytecode specification instability poses a practical challenge: while implementing a traditional decompiler for any one version is achievable, the maintenance effort required to provide cross-version decompilation that quickly supports new version releases is formidable [13, 14].

To address this research challenge, we introduce PyLingual, a data-driven framework that integrates recent advances in NLP research with foundational binary analysis

DOI 10.1109/SP61157.2025.00052

principles. PyLingual aims to demonstrate the effectiveness of ML-based approaches to correctness-sensitive, PL domains. PyLingual is a version-agnostic decompilation pipeline that uses version-specific pluggable NLP models for simple but labor-intensive translation tasks while applying classical binary analysis techniques for instruction parsing and control flow reconstruction. PyLingual consists of three distinct subcomponents: *(1)* bytecode segmentation, *(2)* statement translation, and *(3)* control flow reconstruction. Component boundaries are carefully drawn to minimize engineering friction.

To verify the correctness of decompilation results to end-users, PyLingual boldly embraces *perfect decompilation* [19, 20] (§2), which enables our innovative design decisions by eliminating the need to trust the decompiler. PyLingual presents each decompilation result alongside a verification of correctness or an appropriate failure indicator, preventing statistical errors and decompiler bugs from eroding trust in the results. Further, perfect decompilation provides the foundation of feedback loops both for end-users to improve individual results, and for decompiler designers to detect and repair decompiler bugs.

Our research is built on an extensive collection of real-world datasets from both benign and malicious sources [21–23], which we plan to publish alongside source code and established models. Evaluated against an extensive collection of real-world datasets, PyLingual achieves a 75% perfect decompilation rate on average across Python 3.6 ~ 3.12, marking an average improvement of 45% over State-Of-The-Art (SOTA) Python decompilers [13, 14, 24]. PyLingual makes the following contributions:

- PyLingual explores a unique design direction integrating principled binary analysis theories with neural NLP models to decompile `PYC` binaries.
- We introduce applications of *perfect decompilation* to the design and evaluation of automatic decompilers by verifying the correctness of decompilation results with differential testing against the input binary.
- We evaluate PyLingual against existing Python decompilers across a wide range of Python versions with extensive datasets from benign and malicious sources using the proposed metric.

By being the only service (https://pylingual.io) that provides high-quality decompilation for the latest Python versions, the PyLingual service has been recognized as a de facto framework for reversing modern Python binaries [25, 26]. Regarding the web service's compliance with privacy, legal, and ethical guidelines, we have consulted with our university's IRB[1] and the university legal department. Further details can be found in the appendix in §A.5.

Finally, to assist reverse engineers and future research, we plan to publish our source code, datasets, and models.

1. UTD-IRB-25-6: PyLingual: A Python Decompilation Framework for Evolving Python Versions

2. Perfect Decompilation

PyLingual adopts the notion of *perfect decompilation* to verify the results of automatic decompilation. Used under various names in prior work [19, 20], perfect decompilation defines the process of statically verifying the semantic relationship between a high-level program and a corresponding compiled low-level program through differential testing. Perfect decompilation provides a strict notion of semantic program equivalence, which facilitates the automation of analyzing the outputs of arbitrary decompilers.

Traditional measures of decompilation accuracy, such as Equivalence Modulo Inputs (EMI) [27, 28] or manual verification, rely on dynamic analysis and are constrained by their input space. They are thus expensive to measure and offer limited guarantees of semantic equivalence. Moreover, they can fail to identify misleading decompilation results that users find difficult to verify efficiently [29].

Formally, given a low-level program L produced by a compiler C, a high-level program H is a *perfect decompilation* of L if and only if $C(H) \approx L$, where $\approx$ denotes a decidable refinement of general semantic program equivalence. That is, $C(H) \approx L$ implies that $C(H)$ is semantically equivalent to L. Note that any choice of $\approx$ will be necessarily incomplete because general program equivalence is undecidable. An ideal choice of $\approx$ is efficient to verify, simple to understand, and approximates general program equivalence.

Burk et al. [19], in the original work proposing perfect decompilation, used bytewise equality of assembly to specialize semantic equivalence (*i.e.,* $C(H) = L$), but this formulation assumes that the compiler build chain is *fully reproducible*, providing the exact same output for the same high-level program across different builds. In Python, the compiler may output any number of equivalent bytecode variations depending on the context of the build: function objects may be moved, constant and symbol tables may be reordered, debugging metadata like the build path and timestamp could change, and so on.

To overcome the challenge of non-reproducible builds, we coarsen the refinement of semantic equivalence by canonicalizing non-semantic information and common equivalent low-level implementations. Specifically, we apply three semantics-preserving transformations: *(1)* strip metadata that does not affect runtime execution (*e.g.,* debugging symbols) from the binary; *(2)* remove unreachable code (we show in §3.2 that reliable control flow graphs for Python bytecode can be statically constructed); *(3)* merge consecutive unconditional jumps that have the same targets. These conservative choices enable the practical application of perfect decompilation to real-world `PYC` binaries without compromising its benefits.

Benefits to automatic decompilers. Perfect decompilation provides a strong guarantee of semantic equivalence between the input binary and the decompiled source code, and is easy to verify for any individual input binary, enabling users to trust each decompilation result without needing to trust the decompiler. That is, each correct decompilation result can be presented alongside *verification of correctness*.

The validation of perfect decompilation *does not involve the decompiler*, so potential decompiler bugs do not erode trust in the verified decompilation results, and decompiler designers are able to make adventurous design decisions without independently proving their soundness. Perfect decompilation provides the foundation for an efficient and effective feedback loop for identifying and repairing decompilation failures both on an ad-hoc case-by-case basis and by informing decompiler improvements (§A.3).

Limitations and suitability for Python decompilation. Despite perfect decompilation's undeniable merits, it has not been seriously pursued by previous automatic decompiler research because: *(1)* the compiler C and the configuration used to generate the input binary must be known; and *(2)* satisfying perfect decompilation is much more difficult than satisfying a weaker equivalence metric.

On both fronts, Python decompilation is an ideal frontier to pursue perfection because: *(1)* Python compilation is dominated by CPython, which offers very few configuration options; and *(2)* Python decompilation is easier than traditional binary decompilation because Python bytecode contains more information and is more structured than traditional binaries. Indeed, §4 shows that the core challenges of Python decompilation are quite different from those of traditional binary decompilation.

3. Python Bytecode

We provide background on the structure of Python bytecode, summarize its key properties with respect to decompilation, and briefly discuss existing Python decompilation approaches and their pitfalls.

3.1. Code Organization

Overview. Python bytecode is organized as a tree of "code objects" (visualized in Figure 1), each of which corresponding to one function or class. Several language features such as list comprehensions and lambda expressions are implemented as anonymous code objects, and the code in the top-level script is the `__main__` code object (alternatively called the `<module>` code object). These code objects consist of bytecode instructions, "semantically important" metadata, and "debugging" metadata. Semantically important metadata primarily includes tables for constants and variable symbols, as well as flags used by the interpreter. Debugging metadata includes line number information, the source file name, and the name of the code object, which support error reporting and tracebacks.

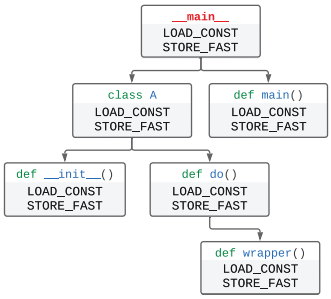
Useful properties. The organization of Python bytecode trivializes several subtasks that are challenging in traditional decompilation. Function boundaries are clearly delineated, with each function consisting of one code object, enabling each code object to be considered independently. Further, within each code object, the instructions are separated from the data and symbol tables. Finally, variable names are semantically important and are included in the bytecode, which improves the readability of the decompiled code.

```python
# A class definition with
# nested functions
class A:
    def __init__(self):
        ...
    def do(self):
        def wrapper():
            ...

def main():
    ...

if __name__ == "__main__":
    main()
```

(a) Python source code



(b) Nested code object (CO) structure

Figure 1: Python source code with nested code object structure. Code objects are classes, functions, and comprehensions.

3.2. Control Flow Considerations

There are four broad categories of control flow in Python: *(1)* jumps, *(2)* function calls, *(3)* return statements, and *(4)* exceptions. Perhaps surprisingly, jump targets in Python are statically determined and cannot cross code object boundaries. Function call targets, in contrast, are determined at runtime; this design choice supports the dynamic reassignment of function symbols. Return statements halt the execution of a function and return a value, but the function may resume execution later in the case of a `yield` statement. Exceptions may be raised at any point during execution to engage a secondary control flow mechanism that engages exception handlers, executes cleanup code, and potentially exits the code object.

Fortunately, control flow that is dynamic in the bytecode is *also dynamic in the source code*. To build intuition, consider the simple case of calling the builtin `print` function. At compile time, the code object has no way of knowing whether `print` will have been overwritten by an unrelated function; the function call in the bytecode simply references the symbol "`print`", which the interpreter resolves at run time. For decompilers, this means that as long as the correct symbols are used for function calls, only static control flow within each code object needs to be structured to correctly recover the source code. It is quite straightforward to create a control flow graph that models jumps and returns, and as we have just seen, function calls can be effectively ignored when modelling control flow for Python decompilation.

Modelling Python's exception handling structures requires version-dependent logic, as it has been the subject of substantial changes in recent years. Prior to Python 3.11, exception handlers were tracked with a block stack at runtime, and relied on the compiler's code generation to ensure that the block stack would be correctly managed in each code path. In Python 3.11 and beyond, the block stack was removed in favor of introducing a new metadata table that maps ranges of instructions to their exception handlers.

4. Python Decompilation Challenges

While the structure of Python bytecode simplifies many aspects of decompilation, the language's development methodology introduces new challenges that have prevented the maturation of community decompilation efforts.

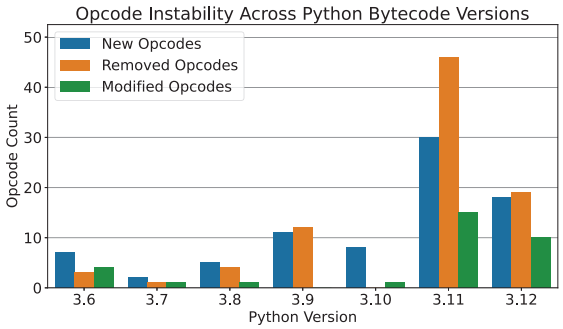


Figure 2: Changes to Python opcodes between versions, with 115 opcodes prior to Python 3.6 and 110 opcodes in Python 3.12

4.1. Specification Instability

The most significant challenge in Python decompilation is the instability of the Python bytecode specification. Since its initial 1991 release, Python has rapidly deployed feature updates, bug fixes, and performance improvements. Each year, minor version releases introduce significant language features and substantial changes to the bytecode representation [30], including the addition, deletion, and modification of instruction opcodes. We summarize the opcode changes for recent versions in Figure 2. Compared to Java bytecode, which has undergone no opcode changes in the last decade, it is typical for significant portions of the Python opcode specification to change every year. This difference in stability stems from philosophical differences between Python and Java, with the Python core development team preferring to adventurously change the language in pursuit of better performance and useful language features. Beyond changes to the opcode definitions, each version of Python introduces insufficiently documented changes to code generation, which further increases the maintenance effort for Python decompilers. Recently, the Python community committed to the "Faster CPython" project [31], resulting in optimizations that emphasize reordering instructions to reduce the time spent by the interpreter managing control flow. Of these optimizations, the most noteworthy was "zero-cost exceptions" introduced in Python 3.11 [32], which completely reworked the exception handling mechanism to use an exception range table instead of a block stack.

4.2. Previous Python Decompilers

While creating a viable Python decompiler for any given Python version is merely a matter of engineering, the core challenge of Python decompilation is to scale across versions, despite the introduction of new source code features and unpredictable changes to the bytecode specification. `uncompyle6` and `decompyle3` are the two most prominent decompilers for Python [13, 14]. `uncompyle6` evolved from earlier iterations that typically supported only one version of Python at a time (*e.g.,* `uncompyle2` [33]). Another Python decompilation framework is `pycdc`, a Python decompiler written in C++. Like `uncompyle6` and `decompyle3` , it seeks to support a broad range of Python versions.

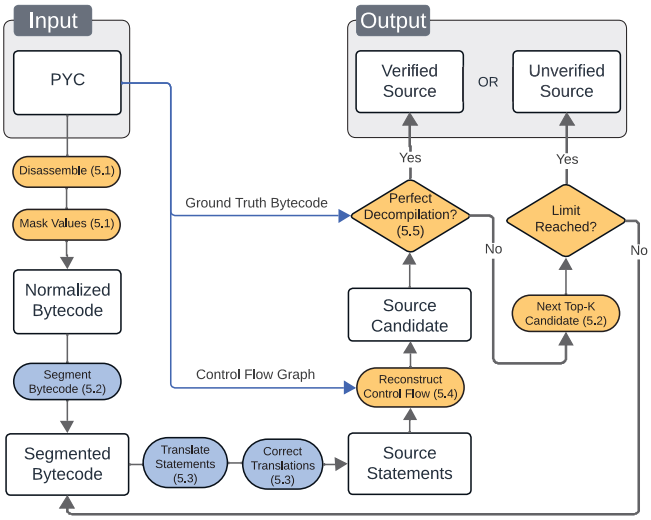


Figure 3: PyLingual architecture.

Existing Python decompilation frameworks depend on hand-crafted version-specific grammars and statement patterns, and have been unable to keep pace with Python's rapid release cycle. Since the launch of Python 3.9 in October 2020, existing decompilers have failed to provide sufficient coverage for practical reverse engineering. Abstractly, PyLingual innovates over existing decompilers by replacing pluggable hand-crafted grammars with pluggable learned NLP models, dramtically reducing maintenance requirements. While recent works such as PYFET [29] aimed to improve the coverage of these decompilers through input preprocessing, they have lacked strong output verification to establish trust in the modified results. Such preprocessing methods are orthogonal and beneficial to the development of decompilers that can sustainably support Python's aggressive development cycle.

5. PyLingual Overview

As shown in Figure 3, PyLingual operates in five stages centered around three major components. First, PyLingual conducts *code normalization* against the source code and disassembled bytecodes [12] to reduce the complexity of the inputs to the NLP models (§5.1). Second, normalized code objects are provided to the *bytecode segmentation* component to identify statement boundaries (§5.2). Next, the *statement translation* component translates each statement of bytecode into the corresponding Python source code statement (§5.3). Then, the *control*

flow reconstruction component mechanically reconstructs the necessary indentation to reproduce the control flow in the input bytecode (§5.4). Finally, *code equivalence verification* conducts instruction-level code comparison to validate the correctness of the decompiled Python source code (§5.5).

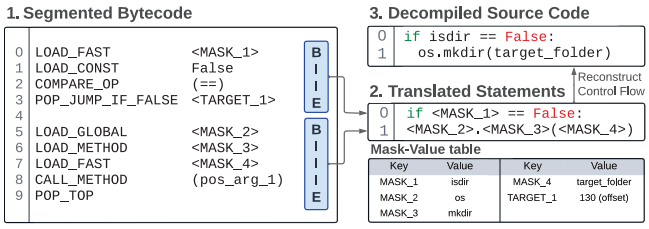


Figure 4: Simplified example of PyLingual's workflow.

5.1. Code Normalization

To disassemble PYC files from different Python versions, PyLingual uses `xdis`, a version-agnostic open-source disassembler [12], to which we contributed supporting code to disassemble Python 3.11 and Python 3.12 binaries [34–39]. Then, to reduce the complexity of the PYC bytecode for the segmentation and translation models, PyLingual replaces distracting details such as variable names and constant values with generic masks [40] derived from metadata table indices in the PYC binary (§3). The original value is mechanically restored at the source level at the end of decompilation. Further, PyLingual enhances the presentation of the bytecode instructions by annotating jump targets and exception-handling structures. This input preprocessing step allows PyLingual to standardize the inputs to the NLP models, even when the bytecode representation changes significantly. An illustrative example of normalized source code and bytecode is provided in Figure 4.

5.2. Bytecode Segmentation

The segmentation module has two goals: *(1)* divide the bytecode into independently digestible "statements" to support the translation module (§5.3); and *(2)* associate bytecode instructions with their corresponding source code statements to support control flow reconstruction (§5.4). Leveraging the grammatical notion of "statements" from source code, we define a "bytecode statement" to be the bytecode instructions that are directly attributed to a source code statement by the CPython compiler.

In a controlled environment with source code access, we can obtain bytecode statements by first using Python's `ast` module to place each source code statement on a separate line, then referring to the line number information (§3) in the resulting PYC binary to associate bytecode instructions with their corresponding source code statements. However, in an uncontrolled environment, the line number information in the PYC binary is not a reliable reference for identifying bytecode statements. Not only is it debugging metadata that can freely be manipulated or omitted without affecting the execution of the bytecode, but it is also common for statements to be spread across multiple lines, or even for multiple statements to appear on the same line.

Therefore, to scalably identify bytecode statements during decompilation, we train a BERT [41] segmentation model for each target Python version (details in appendix §A.1). This segmentation model is tasked with predicting whether each bytecode instruction **B**egins a new statement, is **I**nternal to the current statement, or **E**nds the current statement (illustrated in Figure 4). The transformer model architecture is an ideal choice for segmentation modelling because of its ability to capture long-range bidirectional dependencies. For example, the decision to split a simple `if A and B:` may have irreparable control flow ramifications, depending on the presence of an `else:` block at some distant point in the source code (Figure 5). On the other hand, for a given code object, there might be several valid segmentations (*e.g.,* `import a` and `import b` could become the equivalent `import a, b`). Candidate segmentations can only be mechanically verified by completing the subsequent decompilation process and verifying the result.

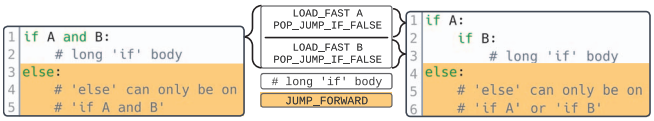


Figure 5: Two different bytecode segmentations, resulting in different source-level meanings due to the long-range dependencies.

Top-k segmentation search. Enabled by the output validation provided by Perfect Decompilation (§2), we search the top k candidate segmentations during decompilation, balancing speed and accuracy. These additional segmentation candidates are obtained by inverting low-confidence statement boundary predictions from the segmentation model (details in appendix §A.3). During this search, PyLingual inverts at most two statement boundary decisions to generate at most ten candidates, which we empirically found to provide an effective tradeoff between decompilation accuracy gain and runtime overhead.

5.3. Statement Translation

The statement translation module translates bytecode statements into Python source statements. The generic sequence-to-sequence translation problem has already been extensively explored in the NLP community [42]. For the single-statement Python bytecode to source code translation task, we train a standard encoder-decoder transformer model (details in appendix §A.1); this architecture is known to provide strong performance on a wide variety of tasks [43]. As illustrated in Figure 4, the translation module produces a flat list of translated source code statements to be stitched together by the control flow reconstruction module (§5.4).

Custom Python tokenizer. Taking advantage of the structured nature of Python source code and bytecode, we equip the statement translation model with a custom tokenizer

built from the documentation of Python's bytecode instructions [44] and source code grammar [45]. This limits the dilution of semantic meaning across multiple tokens, improving semantic coherence. Further details are available in the appendix in §A.2.

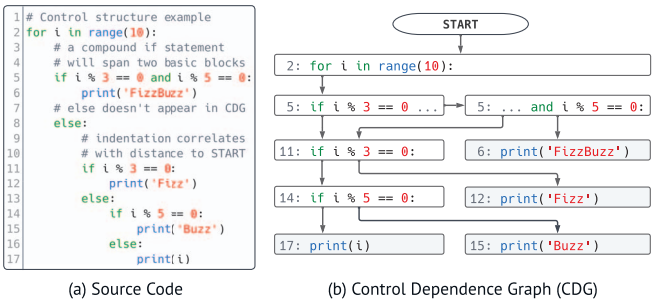
Statement corrector model. To improve the statement translation module's accuracy and reliability, we train a corresponding corrector model that specializes in repairing the translations of "difficult" bytecode sequences (model details in appendix §A.1). These difficult sequences are typically list comprehensions, large boolean expressions, or complex function definitions (heuristic discussed in the appendix in §A.4). The difficulty of these statements stems from their complexity and from the necessity of knowing the end of the source line to correctly translate the beginning of the source line, which is a known weakness of autoregressive decoders [46–48]. By providing the original bytecode and first translation attempt as input to the corrector model, it gets a preview of the full translation, which enables it to fix inconsistencies in the translation.

5.4. Control Flow Reconstruction

The control flow reconstruction module composes source code lines from the translation module into a complete Python program that aims to reflect the control flow of the original `PYC` binary. We achieve this composition in three stages: *(1)* Control Dependency Graph (CDG) construction, *(2)* indentation annotation, and *(3)* source line arrangement.

CDG construction. As discussed in §3.2, Python bytecode has no indirect jumps, and each code object can be considered separately, which enables the efficient construction of a static Control Flow Graph (CFG) for each code object. Broadly, in the CFG for a code object, each instruction is a node, and each node may have up to three outgoing edges: *(1)* a "natural" edge that goes to the next instruction in offset order; *(2)* a "jump" edge which may be conditional and whose target is determined by static arguments in the bytecode; and *(3)* an "exception" edge, which indicates the beginning of an exception handling construct and is directed towards the exception handler. For each instruction, its edges can be identified by inspecting its opcode and static argument. Once the CFG is generated, we condense groups of instructions with no extra control flow into basic blocks to simplify the rest of the process. From the CFG, we create a CDG [49] where each node is a basic block and there exists an edge (u, v) if and only if a control decision in u decides whether v *may* execute. An example CDG is provided in Figure 6; although the CDG in the figure is generated from bytecode instructions, we have labelled the nodes with the corresponding source code for illustrative purposes.

Indentation annotation. Leveraging the ability of the CDG to isolate independent control structures, we aim to assign each basic block an "indentation level", which indicates how deeply nested it is in the code object's control flow structures. The core of the algorithm is a breadth-first implementation of single-source shortest paths from the `START` node.



(a) Source Code (b) Control Dependence Graph (CDG)

Figure 6: A control dependence graph.

In most cases, the indentation level for a node corresponds exactly with the distance to the `START` node in the CDG. However, there are edge cases where this straightforward approach overestimates the indentation level, which we must recognize and account for.

For example, statements with internal control flow, such as `assert` statements and short-circuited boolean expressions, increase the depth of the CDG without actually containing deep nesting in the source code, and exit statements like `break`, `continue`, and `return` can add explicit control dependencies in the CDG where they are implicit in the source code. For these cases, we leverage the segmentation and translation results to constrain the indentation with knowledge of the association between the bytecode and source code statements.

Source line arrangement. To produce a complete Python program, we must arrange the source code of each code object according to their control dependencies, then stitch together the source code for the code object tree (as discussed in §3). For each code object, we first assign each source code statement to the node containing its first instruction and apply the indentation level of the CDG node. Then, we order the statements by the offset of their first instruction, and insert control flow statements that are explicit in the source code, but implicit in the bytecode; these statements are `else`, `finally`, and `while True` after the `SETUP_LOOP` opcode was removed in Python 3.8. Finally, to combine the source code snippets of each code object into a complete program, we recursively traverse the code object hierarchy and inspect the bytecode to identify function and class definitions, which load the associated child code object. We indent and insert the code for the function or class body immediately after its definition to preserve the code hierarchy of the decompiled source code.

5.5. Perfect Decompilation Verification

The general program equivalency problem is known to be undecidable [50], so to verify the results of PyLingual, we adopt a strict notion of decompilation correctness that can be efficiently verified for each input binary (§2). After recompiling the decompiled source code with the appropriate version's compiler (identified by a magic number from the `PYC` specification), we remove unreachable code

and merge consecutive unconditional jumps with the same target to mitigate non-semantic differences. Then, for each code object in the original PYC file, we confirm that the corresponding code object in the recompiled PYC file has the same instructions with the same arguments in the same order, and verify that all semantically important metadata (*e.g.,* exception tables) matches exactly. By showing direct equivalence between the original PYC file and the recompiled decompilation result, PYLINGUAL verifies the correctness of decompilation outputs.

While perfect decompilation yields false negatives for imperfect but semantically equivalent code (*e.g.,* independent statements appearing out of order), it importantly never results in a false positive. This equivalence metric allows for fully automatic verification of the decompilation results, which reduces the time cost for reverse engineers and improves trust in the decompilation system.

6. Implementation

PYLINGUAL incorporates NLP models and mechanical components for its training and translation tasks. Written in Python, the source code spans approximately 5.6K lines, excluding contributions of external open-source projects. Along with the datasets and trained models, we will make PYLINGUAL's source code publicly available.

Python bytecode disassembler. Despite bundled disassembler support in Python releases, PYLINGUAL still requires cross-Python disassembler support due to its design objective being a generic decompilation framework. PYLINGUAL depends on `xdis` [12] for version agnostic disassembler support. Although outside our research scope, we have collaborated with the project maintainer, contributing bug reports, new features, and new language release support.

Transformer models. PYLINGUAL extends two transformer models. The bytecode segmentation module uses Bidirectional Encoder Representations from Transformers (BERT) [41], an encoder-only language model; our statement translation module uses a code-oriented T5 [51] encoder-decoder language model. For each Python version, using one Nvidia RTX 4090 GPU (24G memory), we trained a segmentation model in 8 hours and a statement model in 20 hours. The model training pipelines were fully automated using the Huggingface and PyTorch libraries in 938 lines of Python code.

Training data generation. To train the segmentation and statement translation models, we prepare ground-truth segmented pairs of source code and bytecode. We leverage the CodeSearchNet dataset [21] and our collection of over 1,000,000 real-world Python source files [22] by randomly sampling 200,000 files to serve as the training set, compiling them in the target version, then constructing ground-truth segmentations from line number information derived from debugging symbols. However, in Python, one line can contain multiple statements split by semicolons (`;`), and a single statement can stretch over multiple lines with the line break (\) construct. To ensure that one line always maps to one statement, we use Python's `ast` module to standardize the source code prior to compilation, which removes unnecessary whitespace, comments, and other irrelevant source-level artifacts. Finally, we apply code normalization (§5.1) to ensure that the data representation during training matches the representation that is used during decompilation. The training data generation pipeline is fully automated in 955 lines of Python code.

Mechanical components. Beyond the data-driven NLP components, PYLINGUAL integrates mechanical components for stable and accuracy-critical tasks. We first implement a generic PYC manipulation interface in 1,123 lines of Python code, which is shared by the control flow reconstructor (1,083 lines) and the perfect decompilation verifier (177 lines). The decompiler pipeline that ties all the modules together is written in 336 lines of code. A key component of PYLINGUAL's scalability is the low engineering effort required to scale across versions, with only $\approx$ 400 lines of version-specific code across the seven Python versions supported at this time.

7. Evaluation

To demonstrate the efficacy of PYLINGUAL, we conducted a comprehensive set of experiments leveraging our extensive Python datasets. Specifically, we answer the following research questions:

- **RQ1:** Does PYLINGUAL decompile Python binaries more accurately than existing decompilers? (§7.2)
- **RQ2:** Does PYLINGUAL scale Python decompilation across versions better than existing decompilers? (§7.2)
- **RQ3:** How does perfect decompilation compare to other equivalence metrics? (§7.3)

First and foremost, we evaluate PYLINGUAL across different Python versions compared to existing Python decompilers. Then, we showcase case studies that illustrate the strengths and weaknesses of PYLINGUAL compared to traditional decompilation. Finally, we analyze recent usage statistics of our public online decompilation service to demonstrate the impact of PYLINGUAL. Our evaluations were run on the same server from §6, which is equipped with an AMD Threadripper 5955WX CPU, 128 GB of RAM, and one Nvidia RTX 4090 GPU.

7.1. Datasets

Table 1 shows the basic compositions of our datasets, which we plan to release alongside our source code and models. Given the data-intensive nature of our research, it is critical to establish extensive datasets from credible sources. Our datasets originate from three sources:

(1) Code Search Net (CSN) is a community-verified dataset of source files gathered and open-sourced by GitHub [21]. Originally designed to support code analysis tasks, the CSN dataset is carefully curated by open-source experts to encompass diverse aspects of various languages including Python. However, CSN only captures a static dataset composition as of its presentation in 2019, which precludes it

Table 1: Dataset summaries. Instruction counts were collected from the test set and averaged across versions 3.6-3.12.

Dataset	Version	Total # Files	Instructions per File (Mean / Std)
CSN	source	412,179	76.0 / 84.8
PyPI		1,507,547	929.4 / 4,221.4
VirusTotal	3.6	388	10,188.7 / 32,006.5
	3.7	1,363	3,525.7 / 67,022.8
	3.8	2,390	3,883.3 / 49,071.2
	3.9	5,839	3,336.5 / 97,791.9
PyLingual.io	3.6	21	1,603.0 / 3,071.4
	3.7	164	1,140.9 / 1,988.1
	3.8	429	1,450.5 / 2,216.7
	3.9	331	1,060.3 / 2,010.3
	3.10	765	1,156.8 / 1904.6
	3.11	535	1,403.2 / 2,055.2
	3.12	308	1,273.9 / 1,997.9

from representing source-level language features introduced in Python 3.9 and beyond.

(2) The Python Package Index (PyPI) is the de facto repository for Python modules [22], where thousands of developers publish, update, and maintain their projects daily to share with the rest of the community. Every day since July 2022, our autonomous collection framework has downloaded and deduplicated project contents from PyPI's latest update RSS feed to capture the diverse characteristics of real-world users and reflect new features as they are adopted.

(3) VirusTotal provides Python malware samples that were packaged using open-source tools. We collected the dataset by querying Python-related keywords via VirusTotal's API from June to August 2022. In contrast to benign sources, malicious files only include the `PYC` binary. The version coverage of the VirusTotal dataset is limited to 3.9 and below because Python 3.10 was not yet well-adopted at the time of collection, and 3.11 and 3.12 had not yet been released.

(4) PyLingual.io is our public online decompilation web service, which has experienced organic user growth in recent months, gaining online mentions [52–54] and hundreds of daily `PYC` uploads, and peaking at over 1,000 uploads in one day. This dataset, containing one month of uploads, directly captures demand for Python decompilation in the wild, and demonstrates the necessity of providing decompilation coverage for new Python versions. Due to privacy concerns regarding online user data, this dataset will not be released. For more details regarding usage statistics and privacy considerations, refer to §A.5 in the appendix.

Train and test set composition. To train the segmentation and statement translation models, we randomly selected 100,000 source code files each from CSN and PyPI to serve as the training set. From the remaining files, we evaluate the decompiler performance across a random sample of 2,000 source code files from CSN, 3,000 source code files from PyPI, and all available `PYC` files from our VirusTotal and PyLingual.io datasets for the relevant version. The size of the test set was chosen to balance the comprehensiveness of the results against the evaluation overhead. Source code files are compiled to the appropriate version for each experiment.

Test set characteristics. We observed quantitative and qualitative differences in the Python code from each of our three data sources. Figure 7 illustrates the distribution of the size of the largest code object in each file. Generally, moving from CSN to PyPI to VirusTotal, the length of the largest code object increases alongside the complexity of the code. CSN files typically include only a single short function, with scarce use of Python's advanced features. PyPI files vary widely, from scattered utility files to large class definitions. Further, because the PyPI and PyLingual.io datasets are continuously updated, they are the only datasets to include new source features such as `match` statements and exception groups. In PyLingual.io, there are distinct gradations of difficulty: simple scripts have been uploaded to presumably test the decompiler, projects and small products have been uploaded to recover lost source code, and several real-world malware samples have been uploaded to assist reverse engineers. Finally, VirusTotal files typically place their entire payload in one function, contain nested exception handlers, and sometimes even include obfuscation. The increasing hostility of each data source provides a gradient of difficulty to measure the performance of Python decompilers.

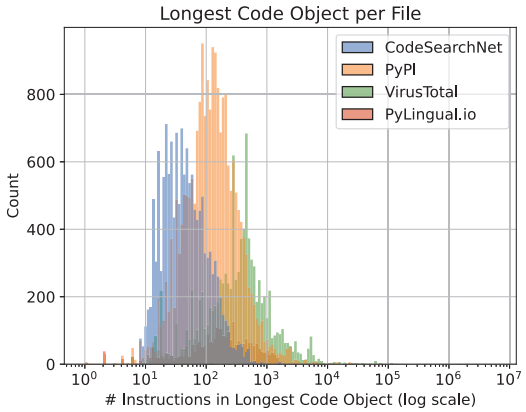


Figure 7: Distribution of the largest code objects in the test set.

7.2. Perfect Decompilation Evaluation

In our evaluation, we measure the accuracy of decompilation using perfect decompilation [19] instead of EMI [27, 28] for three reasons: *(1)* perfect decompilation provides strong semantic equivalence guarantees; *(2)* perfect decompilation does not depend on the selection of inputs; *(3)* perfect decompilation is a static verification technique, eliminating the requirement to run untrusted code from online sources. These result in improved reliability, reproducibility, and safety of our evaluation.

The perfect decompilation implements an efficient and rigorous accuracy measure, serving as a foundation for many subsequent analyses. However, its strict accuracy criteria can be overly conservative, resulting in the incorrect labeling of semantically equivalent decompilation results as errors. In §7.3, we manually investigate cases where the perfect decompilation metric makes such misclassifications.

Table 2: Decompilation accuracy comparison. PyLingual is configured with $k = 10$.

Dataset	Version	PyLingual				Uncompyle6 [13]				Decompyle3 [14]				Pycdc [44]			
		Perfect	Semantic Error	Syntax Error	No Output	Perfect	Semantic Error	Syntax Error	No Output	Perfect	Semantic Error	Syntax Error	No Output	Perfect	Semantic Error	Syntax Error	No Output
CSN	3.6	**96.3%**	2.4%	1.2%	0.1%	85.7%	13.8%	0.4%	0.1%	-	-	-	-	22.1%	61.9%	16.1%	0.0%
	3.7	**96.0%**	2.9%	1.0%	0.1%	82.5%	15.7%	0.7%	1.1%	85.9%	12.9%	1.1%	0.2%	21.8%	62.1%	16.1%	0.0%
	3.8	**97.0%**	1.9%	0.9%	0.1%	64.1%	17.8%	11.6%	6.5%	75.8%	19.0%	1.1%	4.1%	18.8%	62.9%	18.4%	0.0%
	3.9	**98.5%**	0.8%	0.5%	0.2%	-	-	-	-	-	-	-	-	18.8%	68.8%	12.4%	0.0%
	3.10	**95.0%**	3.7%	1.1%	0.3%	-	-	-	-	-	-	-	-	18.2%	69.9%	11.8%	0.0%
	3.11	**95.9%**	2.5%	1.5%	0.2%	-	-	-	-	-	-	-	-	16.4%	81.5%	2.0%	0.0%
	3.12	**93.4%**	3.3%	3.2%	0.2%	-	-	-	-	-	-	-	-	17.5%	81.5%	1.0%	0.0%
PyPI	3.6	**79.3%**	10.5%	8.5%	1.7%	47.3%	48.0%	1.6%	3.1%	-	-	-	-	9.9%	43.5%	46.7%	0.0%
	3.7	**80.9%**	9.0%	8.4%	1.7%	40.0%	49.1%	5.6%	5.3%	51.0%	41.1%	6.7%	1.2%	9.3%	41.7%	49.0%	0.0%
	3.8	**83.3%**	7.2%	7.9%	1.6%	32.1%	37.1%	17.9%	12.9%	36.6%	48.9%	5.8%	8.7%	8.6%	40.0%	51.4%	0.0%
	3.9	**87.3%**	6.2%	4.8%	1.6%	-	-	-	-	-	-	-	-	8.6%	46.0%	45.4%	0.0%
	3.10	**81.1%**	7.5%	9.7%	1.7%	-	-	-	-	-	-	-	-	6.7%	42.2%	51.1%	0.0%
	3.11	**82.6%**	7.2%	8.4%	1.7%	-	-	-	-	-	-	-	-	8.4%	45.8%	45.9%	0.0%
	3.12	**77.7%**	8.0%	12.7%	1.5%	-	-	-	-	-	-	-	-	7.2%	44.1%	48.7%	0.0%
VirusTotal	3.6	**46.9%**	11.9%	36.7%	4.5%	28.6%	37.6%	19.6%	14.3%	-	-	-	-	15.8%	40.1%	42.9%	1.1%
	3.7	**49.7%**	7.1%	33.2%	10.0%	28.1%	25.1%	30.8%	16.0%	30.3%	26.0%	36.5%	7.2%	14.5%	42.0%	42.6%	0.9%
	3.8	**53.9%**	6.7%	29.1%	10.2%	22.1%	12.6%	14.6%	50.7%	24.3%	16.5%	14.2%	44.9%	17.1%	40.1%	42.8%	0.0%
	3.9	**57.5%**	9.4%	24.4%	8.7%	-	-	-	-	-	-	-	-	9.5%	50.6%	39.9%	0.0%
PyLingual.io	3.6	**52.4%**	4.8%	33.3%	9.5%	28.6%	38.1%	19.0%	14.3%	-	-	-	-	9.5%	57.1%	33.3%	0.0%
	3.7	**80.5%**	8.5%	7.3%	3.7%	63.4%	15.2%	11.6%	9.8%	43.3%	37.2%	12.2%	7.3%	15.9%	51.8%	32.3%	0.0%
	3.8	**59.4%**	12.1%	25.2%	3.3%	17.9%	19.6%	5.8%	56.6%	22.4%	28.4%	4.0%	45.2%	10.0%	28.4%	61.3%	0.2%
	3.9	**73.4%**	11.5%	12.4%	2.7%	-	-	-	-	-	-	-	-	16.3%	53.2%	30.5%	0.0%
	3.10	**54.9%**	14.8%	25.8%	4.6%	-	-	-	-	-	-	-	-	13.2%	44.4%	42.4%	0.0%
	3.11	**54.8%**	13.1%	26.0%	6.2%	-	-	-	-	-	-	-	-	9.3%	51.6%	39.1%	0.0%
	3.12	**43.5%**	9.4%	42.9%	4.2%	-	-	-	-	-	-	-	-	14.3%	49.7%	36.0%	0.0%

Table 2 measures the effectiveness of PyLingual against other SOTA Python decompilers: `uncompyle6`, `decompyle3`, and `pycdc`. We evaluate a strawman LLM decompiler in §7.5. To provide a comprehensive view of the decompilation landscape, we examine `PYC` binaries from research-oriented (CSN), production-oriented (PyPI), malicious (VirusTotal), and wild (PyLingual.io) environments. For each decompiler, the decompiled source for each `PYC` binary falls into one of four categories:

1) It is *Perfect* and compiles to the input binary (§2, §5.5).
2) It has *Semantic Errors* and compiles to a different `PYC`.
3) It has *Syntax Errors* and is not valid Python code.
4) The decompiler produced *No Output*.

The *No Output* category indicates an internal error in the decompiler, with causes varying across different decompiler families. In PyLingual's case, these failures mainly arise from GPU memory limitations, which can easily be addressed at the cost of runtime efficiency using a sliding window mechanism, or with additional hardware.

PyLingual produces substantially more correct decompilation results than any of the other Python decompilers. Even in versions 3.6-3.8, which were previously considered to be well-supported, PyLingual improves over the best available traditional decompiler by 13.9% in CSN, 36.2% in PyPI, 26.0% in PyLingual.io, and 22.4% in VirusTotal on average. PyLingual also offers competent support for newer Python versions that were previously not well-supported; in versions 3.9-3.12, PyLingual improves over the best available traditional decompiler by 78.0% in CSN, 74.4% in PyPI, 48.0% in VirusTotal, and 43.4% in PyLingual.io on average.

All the evaluated Python decompilers exhibit some instability across Python versions, even for the source code datasets where the same files were compiled with different versions of Python. In the most extreme cases on the PyPI dataset, PyLingual's perfect decompilation rate varied from 77.7% to 87.3%, `uncompyle6`'s varied from 32.1% to 47.3%, `decompyle3`'s varied from 36.6% to 51.0%, and `pycdc`'s varied from 6.7% to 9.9%. This instability is intrinsically tied to the instability in Python's bytecode across versions, leading to variations in the effectiveness of specific techniques as well as variation in the overall difficulty of Python decompilation. In the bytecode-only datasets (VirusTotal and PyLingual.io), there is additional variation both in the representation of each version, and in the specific files that are available in each version.

Decompiler design choices also influence the impact of the dataset on the decompilation accuracy. In fact, the most surprising result from Table 2 is that `pycdc` is the only decompiler to perform better against VirusTotal than against PyPI. In our experience, VirusTotal samples are more likely to contain deep combinations of simple language primitives, where PyPI samples are more likely to leverage advanced language features. Despite this interesting twist, `pycdc` still had the lowest overall accuracy, even on VirusTotal data. On the other side of the table, PyLingual scored the lowest on VirusTotal relative to itself on the other datasets, primarily due to the size of the functions and the prevalence of deeply nested exception handling structures, but still maintained the highest accuracy among the decompilers.

7.3. Equivalence Metric Comparison

Comparison to EMI. In comparison to EMI [27, 28], which is a relaxation of semantic equivalence that relies on dynamic testing, perfect decompilation is a strict, statically verifiable test. This naturally raises the question of whether perfect decompilation is *too* strict, falsely rejecting a large proportion of semantically equivalent but imperfect decompilations. Table 3 summarizes the decompilation results for PyLingual, `uncompyle6`, and `decompyle3` on the open-source `numpy` [55] library in Python 3.8. EMI was deter-

Table 3: Summary of decompilation results for Numpy in Python 3.8. Files that were EMI-only were manually investigated.

Decompiler	Perfect & EMI	EMI Only		Neither	No Output
		Equivalent	Different		
PyLingual	145	2	16	53	8
Uncompyle6	72	17	23	64	48
Decompyle3	79	19	37	55	34

Table 4: Comparison between PyLingual and PyFET [29] using the PyFET's released reproducibility dataset.

Version	PyLingual			PyFET [29]		
	Perfect	Semantic Error	Syntax Error	Perfect	Semantic Error	Syntax Error
3.7	**86.6%**	6.0%	7.5%	28.4%	59.7%	11.9%
3.8	**90.8%**	9.2%	0.0%	2.3%	90.8%	6.9%
3.9	**96.8%**	3.2%	0.0%	82.1%	17.9%	0.0%

Table 5: Comparison of decompilation accuracy with and without the statement corrector model.

Dataset	Version	With Corrector				Without Corrector			
		Perfect	Semantic Error	Syntax Error	No Output	Perfect	Semantic Error	Syntax Error	No Output
CSN	3.8	97.5%	1.8%	0.7%	0.1%	97.0%	2.1%	0.9%	0.0%
	3.10	95.0%	3.7%	1.1%	0.2%	94.7%	3.9%	1.2%	0.2%
	3.12	93.5%	3.3%	3.2%	0.1%	93.3%	3.4%	3.3%	0.1%
PyPI	3.8	84.1%	6.5%	7.8%	1.6%	80.4%	7.5%	10.9%	1.2%
	3.10	81.1%	7.7%	9.6%	1.6%	77.2%	9.4%	12.1%	1.3%
	3.12	77.8%	8.1%	12.7%	1.5%	75.0%	8.2%	15.6%	1.1%
PyLingual.io	3.8	59.9%	12.1%	26.8%	1.2%	58.3%	13.3%	28.0%	0.5%
	3.10	54.9%	16.1%	25.1%	3.9%	53.6%	16.3%	27.7%	2.4%
	3.12	43.5%	9.7%	43.5%	3.2%	43.2%	9.7%	44.2%	2.9%

Table 6: Strawman LLM decompiler accuracy, on version 3.10.

Dataset	gpt-4o-mini				gpt-4o			
	Perfect	Semantic Error	Syntax Error	No Output	Perfect	Semantic Error	Syntax Error	No Output
CSN	12.0%	84.9%	3.1%	0.0%	20.8%	70.0%	9.2%	0.0%
PyLingual.io	4.8%	79.2%	14.9%	1.0%	13.7%	58.2%	27.2%	1.0%

mined by decompiling one file, replacing that file with the decompiled result in the library, then running the included unit tests.

We find that the false rejection rate of perfect decompilation is low, but non-negligible. For PyLingual, only 2 out of 147 semantically equivalent decompilations were falsely rejected by perfect decompilation. The false rejection rates for other decompilers were higher, with 17 of `uncompyle6`'s 89, and 19 of `decompyle3`'s 98 semantically equivalent decompilations being falsely rejected. The primary reason for this discrepancy is common formatting choices from `uncompyle6` and `decompyle3` that subtly affect the bytecode, including converting type hints to strings, explicitly implementing the `if/raise` behavior of `assert` statements, and explicitly calling the `complex` function instead of using the native complex number constant syntax. Although the accuracy improvement in semantically equivalent decompilation is smaller than in perfect decompilation, PyLingual still substantially advances decompilation accuracy.

Comparison to forensic equivalence. PyFET [29] attempts to improve the accuracy and scalability of Python decompilation by preprocessing inputs that trigger fatal decompiler bugs such that the decompilation completes and the result includes a recognizable "forensically equivalent" artifact. Some of these artifacts are automatically reversible – the original source code can be restored from the artifact with a string replacement scheme. The authors of [29] have released a set of Python bytecode files that trigger fatal errors in `uncompyle6` [13] and `decompyle3` [14], along with patched versions that have been manually verified to be "forensically equivalent". Using these forensically equivalent decompilation results as the output of PyFET (after reversing reversible artifacts to get as close to the original source code as possible), Table 4 compares the perfect decompilation rate of PyFET against that of PyLingual on those files.

Even using the provided selection of files, which were chosen to be challenging for existing decompilers and manually checked by the PyFET authors, we observe that PyLingual provides substantially more accurate decompilation results. Beyond demonstrating PyLingual's efficacy, these results primarily reflect a difference in objectives between PyLingual and PyFET [29]. PyLingual aims for perfect decompilation, which is a strict refinement of program equivalence that is automatically verifiable. PyFET aims for forensic equivalence, which is an informal relaxation of program equivalence that requires manual verification. Recognizing this difference in objectives, the dramatic evaluation result in Table 4 is unsurprising, and highlights that imperfect decompilation can still be useful.

7.4. Ablation Study

Top-k segmentation ablation. Figure 8 illustrates the effect of the top-k segmentation search on the overall decompilation accuracy on Python 3.8, 3.10, and 3.12. In this evaluation, the statement corrector model was not used. We observe that the impact of the segmentation search is more pronounced in more difficult datasets, and begins to show diminishing returns around $k = 10$, which is the limit used in our main evaluation in Table 2.

Statement corrector model ablation. Table 5 shows the benefit from the statement corrector model, which shows an outsized effect on PyPI compared to the CodeSearchNet and PyLingual.io datasets. Overall, the effect of the statement corrector model is marginal, but provides improvements on a small number of challenging cases.

7.5. Strawman LLM Decompilation

To provide a baseline reference for the ability of language models to interpret and decompile Python bytecode, we referred to [56] to use gpt-4o-mini and gpt-4o [57] to decompile CodeSearchNet binaries in Python 3.10. Although the dataset and version coverage of this baseline were limited to reduce evaluation costs, Table 6 clearly and unsurprisingly shows that off-the-shelf Large Language Models (LLMs) are less capable of Python decompilation than the best available Python decompilers. This naïve decompilation method simply prompts the LLM to provide

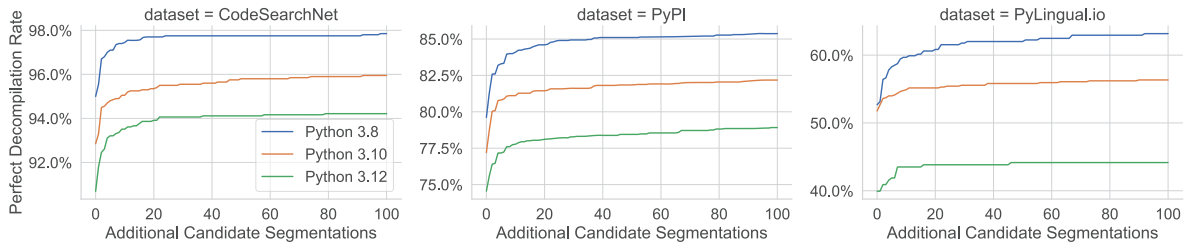


Figure 8: Increase in decompilation accuracy due to top-k segmentation searching.

the original Python source code for a given `xdis` [12] bytecode dump in a zero-shot fashion. Future work may choose to explore more sophisticated prompting strategies and bytecode preprocessing steps to improve generic LLMs' ability to perform Python decompilation.

8. Case Studies

Here, we compare PyLingual to existing Python decompilers by contrasting their results on illustrative bytecode examples. The accuracy of decompilation was verified via perfect decompilation (§2). Readers may refer to Table 7 in the appendix for the full case study results.

8.1. Complex Conditional Expressions

Across our datasets, we encountered cases of complex conditional statements that can contain multiple expressions and statements joined together. This is usually a quite difficult task to approach for current decompilers. Our segmentation model is able to accurately decide which conditional components should be joined on one line or nested, and our statement model is capable of translating these complex boolean expressions into equivalent source code. For example, the VirusTotal dataset contains a file (Figure 9) with nested conditionals and multiple statements. As this sample is a Python 3.9 bytecode, `decompyle3` and `uncompyle6`, which only support versions 3.8 and below, produced no result. However, `pycdc` fails to correctly segment the boolean expression, splitting off the last `e[2]>e[3]` and incorrectly grouping the terms of the expression for translation.

A similar sample from our CSN dataset for Python 3.7 shows how PyLingual outperforms other state-of-the-art decompilers in cases with complex conditionals and control flow (Figure 10). Comparing the results of the existing decompilers with those of PyLingual, it is clear that traditional pattern matching approaches are inadequate for managing complicated conditional expressions. Where existing decompilers consistently failed to correctly place the final return statement, PyLingual not only reconstructs the original control flow, but also correctly places all the conditional pieces on the same line.

PyLingual's efficacy in reconstructing complex conditional statements strongly validates the design decision

```python
# pycdc:
def is_pma(matrix):
    for line in matrix:
        for e in line:
            if not (e[3] != 255 or e[0] > e[3]) and e[1] > e[3]:
                if e[2] > e[3]:
                    return False
                continue
                return True

# PyLingual:
def is_pma(matrix):
    for line in matrix:
        for e in line:
            if e[3] != 255 and (e[0] > e[3] or e[1] > e[3] or e[2] > e[3]):
                return False
    else:
        return True
```

Figure 9: VirusTotal 3.9 sample with a complex condition.

to dedicate a segmentation module for the bytecode segmentation, which inherently carries crucial control flow semantics. This focused approach not only enhances the precision in identifying bytecode statement boundaries but also highlights subtle, essential control flow semantics. Even if the segmentation model produced inaccurate segmentations for the examples above, PyLingual's top-k segmentation design (§5.2) provides a robust fallback, adding an extra layer of error protection.

```python
# original:
def d_cost(ch1, ch2):
    if ch1 != ch2 and (ch1 == 'H' or ch1 == 'W'):
        return group_cost
    else:
        return r_cost(ch1, ch2)

# pycdc:
def d_cost(ch1 = None, ch2 = None):
    if ch1 != ch2:
        if ch1 == 'H' or ch1 == 'W':
            return group_cost
        return None(ch1, ch2)

# decompyle3:
def d_cost(ch1, ch2):
    if ch1 != ch2:
        if ch1 == 'H' or ch1 == 'W':
            return group_cost
        return r_cost(ch1, ch2)

# PyLingual:
def d_cost(ch1, ch2):
    if ch1 != ch2 and (ch1 == 'H' or ch1 == 'W'):
        return group_cost
    return r_cost(ch1, ch2)
```

Figure 10: CSN 3.7 sample with complex condition.

```python
# original:
list_str = str([i.ProductID
    for i in product_user_list
    if i.ProductID > 0 or product_id == 0]).strip('[').strip(']')

# pycdc:
list_str = None((lambda .0 = None: [ i.ProductID
    for i in .0 if product_id == 0])
     (product_user_list)).strip('[').strip(']')

# uncompyle:
list_str = str([i.ProductID
    for i in product_user_list if not i.ProductID > 0
    if product_id == 0]).strip('[').strip(']')

# decompyle3:
list_str = str([i.ProductID
    for i in product_user_list if not i.ProductID > 0
    if product_id == 0]).strip('[').strip(']')

# PyLingual:
list_str = str([i.ProductID
    for i in product_user_list
    if i.ProductID > 0 or product_id == 0]).strip('[').strip(']')
```

Figure 11: Python 3.7 sample from PyPI of list comprehension.

8.2. Lambdas and List Comprehensions

Throughout the evaluation, we notice that existing decompilers often struggle with lambda functions and comprehensions. These source-level constructs are implemented as anonymous code objects that take arguments and capture values from their surrounding scope. `pycdc` struggles the most with these constructs, often including the internal names of arguments in the comprehensions, rather than their source-level names from the outer scope. `pycdc`'s relative lack of support for advanced Python structures is showcased in Figure 11. In the same line of code, `uncompyle6` and `decompyle3` suffer from a different issue, which stems from the incorrect translation of short-circuited boolean expressions. In an attempt to explicitly capture the control flow of the `or` condition, the parsing grammars ensured that the second condition would only be evaluated if the first condition was false, but failed to capture the wider picture.

PyLingual's success in these cases can be attributed to segmenting and translating the entire list comprehension as one statement while the other decompilers attempt to decompile the statement expression by expression. This mechanism allows PyLingual to effectively consider the full semantics of the comprehension without creating confusion with similar bytecode expressions with different translations. We discuss the current limitations of PyLingual and future mitigations leveraging new advances in NLP [46] in §10.

8.3. Semantic Error Localization

When PyLingual decompiles a `PYC` sample, and our strict equivalency metric (§2, §5.5) indicates that incorrect semantics were generated, PyLingual is able to report strict and localized information about where it has failed. This information gives a reverse engineer using PyLingual a specific source line number and bytecode instruction offset to focus additional reversing and debugging. In Figure 12 we demonstrate an example of this error localization on a VirusTotal 3.9 sample where PyLingual yielded source code with semantic errors.

Although the problem is clear when comparing the incorrect source line to the correct manually decompiled source line (depicted in Figure 13), the difference in the bytecode would be difficult to notice without an automatic error detection mechanism. The arguments to `BUILD_LIST` at offset 434 and `CALL_FUNCTION_KW` at offset 440 were swapped, and the "help" item was removed from the `LOAD_CONST` at offset 438. At the source level, this is represented by the choices list being too long, which misaligned the arguments to the `run_parser.add_argument` call. PyLingual's perfect decompilation verification was able to identify and locate the exact instructions affected by the semantic decompilation error. The error localization facilitates a feedback loop with a human reverse engineer, which allows "almost perfect" decompilations to become perfect with a small amount of expert analysis.

9. Related Work

The challenges faced in traditional binary decompilation research differ significantly from those faced by PyLingual. Traditional binaries are characterized by their stable Abstract Binary Interface (ABI) and Instruction Set Architectures (ISAs), as well as aggressive compiler toolchain optimizations that strip away critical information for source recovery. In contrast, Python binaries are characterized by constantly evolving bytecode specifications and rapid, unpredictable deployment of new language features, but suffer substantially less information loss from optimizations. The constant evolution of Python and other High-level Dynamic Languages (HDLs) demands significant maintenance effort for their reverse engineering infrastructure.

9.1. Traditional Binary Decompilation

Traditional binary analysis is a well-established research field due to high demand from reverse engineers who want to understand binaries without having access to the source and from security analysts who need to analyze malware payloads. The field has been extensively explored by both industry and academia [58–61]. Despite the availability of mature, off-the-shelf tools, numerous research problems related to pushing the limits of decompilation remain.

Traditional decompilation. Since Cifuentes et al. [62] first pioneered the field, decompilation research has evolved to address various practical and theoretical challenges, which can be primarily summarized into two sub-problems: *(1) statement translation* to restore type information and data dependencies [63], and *(2) structural analysis* to identify code blocks and restore control dependencies among them [62, 64–66]. Structural analysis has more impact on the performance and usability of a decompiler, so it has been the primary focus of recent research [64–66].

Neural decompilation. Recent advances in neural translation have sparked interest in their use for binary analysis and

```
400 LOAD_NAME            (run_parser)
402 LOAD_ATTR            (add_argument)
404 LOAD_CONST           ('-ll')
406 LOAD_CONST           ('log_level')
408 LOAD_NAME            (int)
410 LOAD_NAME            (loggirg)
412 LOAD_ATTR            (WARNING)
414 LOAD_NAME            (loggirg)
416 LOAD_ATTR            (DEBUG)
418 LOAD_NAME            (loggirg)
420 LOAD_ATTR            (INFO)
422 LOAD_NAME            (loggirg)
424 LOAD_ATTR            (WARNING)
426 LOAD_NAME            (loggirg)
428 LOAD_ATTR            (ERROR)
430 LOAD_NAME            (loggirg)
432 LOAD_ATTR            (CRITICAL)
434 BUILD_LIST           5
436 LOAD_CONST           ('Log level. DEBUG - 10,\
                          INFO - 20, WARNING - 30,\
                          ERROR  - 40, CRITICAL - 50.\
                          Default is WARNING.')
438 LOAD_CONST           (('dest', 'type', 'default', 'choices', 'help'))
440 CALL_FUNCTION_KW     6
442 POP_TOP
```

(a) Original PYC disassembly.

```
400 LOAD_NAME            (run_parser)
402 LOAD_ATTR            (add_argumert)
404 LOAD_CONST           ('-ll')
406 LOAD_CONST           ('log_level')
408 LOAD_NAME            (int)
410 LOAD_NAME            (logging)
412 LOAD_ATTR            (WARNING)
414 LOAD_NAME            (logging)
416 LOAD_ATTR            (DEBUG)
418 LOAD_NAME            (logging)
420 LOAD_ATTR            (INFO)
422 LOAD_NAME            (logging)
424 LOAD_ATTR            (WARNING)
426 LOAD_NAME            (logging)
428 LOAD_ATTR            (ERROR)
430 LOAD_NAME            (logging)
432 LOAD_ATTR            (CRITICAL)
434 BUILD_LIST           6
436 LOAD_CONST           ('Log level. DEBUG - 10,\
                          INFO - 20, WARNING - 30,\
                          ERROR  - 40, CRITICAL - 50.\
                          Default is WARNING.')
438 LOAD_CONST           (('dest', 'type', 'default', 'choices'))
440 CALL_FUNCTION_KW     5
442 POP_TOP
```

(b) Disassembly of decompiled source, then recompiled.

Figure 12: Virustotal 3.9 PYC disassembly comparison to demonstrate error localization utility. Subtle decompilation errors (highlighted) may be overlooked by manual analysis, but are easily detected by automated strict bytecode comparison.

```
<module>: Failure detected at line number 90 and instruction
offset 434: Different bytecode
<module>.print_info: Success: Equal
<module>.configure_logging: Success: Equal
```

(a) Sample Equivalence Report.

```
run_parser.add_argument('-ll', dest='log_level', type=int,
                        default=[logging.WARNING, logging.DEBUG,
                                 logging.INFO, logging.WARNING,
                                 logging.ERROR, logging.CRITICAL],
                        choices='Log level. DEBUG - 10, INFO - 20,\
                                 WARNING - 30, ERROR  - 40,\
                                 CRITICAL - 50. Default is WARNING.')
```

(b) Line 90 of PyLingual's decompilation of this sample.

```
run_parser.add_argument('-ll', dest='log_level', type=int,
                        default=logging.WARNING,
                        choices=[logging.DEBUG, logging.INFO,
                                 logging.WARNING, logging.ERROR,
                                 logging.CRITICAL],
                        help='Log level. DEBUG - 10,\
                              INFO - 20, WARNING - 30,\
                              ERROR - 40, CRITICAL - 50.\
                              Default is WARNING.')
```

(c) Manual decompilation of line 90 of this sample.

Figure 13: Local error analysis on a VirusTotal 3.9 sample.

decompilation. Although large public code datasets meet the data demands of NLP approaches, decompilation requires strict syntax compliance and semantic accuracy, posing new challenges to natural language translation and complicating the generation of trustworthy results.

Wartell et al. [67] proposed one of the first ML-assisted binary analyses, using a predication by partial matching model to differentiate code from data in x86 binaries. Later, Shin et al. [68] built a multi-layer RNN network that consumes one byte at a time to predict a byte sequence that identifies function boundaries. Katz et al. [69] first proposed an RNN-based model similar to the those used for natural language translations. However, their work employed a naïve seq2seq model that struggled to identify PL-specific features such as function and statement boundaries, number and type of instruction operands, *etc.* CODA [70] implements a type-aware encoder and AST decoder to preserve important code structures. They also implemented an Error Correction post-processor to improve the prediction accuracies. Neutron [71] uses long-short-term-memory (LSTM) models to segment and translate unoptimized assembly into C code. Similar to CODA, Neutron relies heavily on mechanical correction of common model translation errors.

9.2. Reversing and Decompilation for HDLs

The rising popularity of HDLs such as Ruby, Lua, and Golang, is driving demand for portable packaging and deployment to support the highly heterogeneous and fragmented IoT (Internet of Things) and CPS (Cyber-Physical Systems) computing sectors. In response, developers and malware authors alike have minimized external dependencies with architecture-neutral formats, standardized modules, and adaptable runtime components [4, 72, 73]. Compared to regular binaries directly compiled from low-level system languages (*i.e.,* assembly and C), HDL families largely lack reversing support. When dealing with languages that incorporate an intermediate bytecode representation for their compiled code (*e.g.,* `PYC` files for Python and CIL files for `.NET` framework), reverse engineers often depend on incomplete or inaccurate solutions for analyzing malicious binaries in this intermediate form.

Python decompilers. The two most popular decompilers for Python binaries are `uncompyle6` [13] and `decompyle3` [14]. `uncompyle6` evolved from early attempts at creating a decompiler that leveraged the same strategies as compilers. Because bytecode is ambiguous without context, `uncompyle6` uses an Earley parser [74] to generate many possible parallel parse trees when decompiling bytecode. `decompyle3` is a reworking of

uncompyle6 to improve its overall maintainability, focusing on control flow support for Python 3.7 and 3.8.

Since decompyle3 first released in 2021 as a fork of the previous uncompyle6 project, over 10,000 lines of code have been added to improve performance on Python 3.7 and 3.8, with the most recent release in 2024 still providing no public support to Python 3.9 or later, although the maintainer has mentioned private initial developments to support Python 3.9 and 3.10. The foundational work to extend support from Python 3.7 to 3.8 goes even further back to 2019 in the uncompyle6 project; nearly 30,000 lines of code have since been added to provide maintenance and improvements to the coverage of Python 3.8 and below.

pycdc [24] is a less popular Python decompiler due to its limited coverage of language features. However, pycdc does provide limited support to Python 3.9 and above, which decompyle3 does not. pycdc attempts to track control flow structures using a stack, similar to how the Python interpreter, and matches bytecode statements against a known list of patterns. While pycdc has undergone a modest $\approx 4,000$ lines of code modification to support Python 3.9 and 3.10, the accuracy of the decompilation results is lacking. In §7, we saw that pycdc was unable to decompile even 25% of PYC binaries for any evaluated Python version.

Decompilers for other HDLs. Soot [75], designed by Vallée-Rai et al., provides a framework to decompile binaries written in Java and Dalvik bytecodes. The Soot framework is actively maintained by the open-source community to stay up-to-date with Java. Furthermore, the framework supports code reassembly to instrument additional functionalities. Several stable decompilers for the .Net framework [76, 77] are also actively maintained. Although niche and thus not actively maintained, decompilers also exist for other HDL families such as Ruby and Lua [78, 79]. Although malware written using these HDLs exists, the community lacks reliable support for these languages. Demands for systematic approaches to fix failures and reduce maintenance efforts are also high for these decompilers.

10. Discussion and Future Works

Limitations of NLP models. PYLINGUAL faces several challenges due to its extension of NLP techniques. The segmentation models' capacity is limited, struggling with exceptionally lengthy input bytecode. The capacity of the statement translation model, defined by its model parameters, thus ties directly to GPU memory size. While we demonstrate that modern transformer models support a large enough context to process most real-world Python samples, it is desirable for a decompiler to gracefully handle even arbitrarily long statements, functions, and files. The context limitation problem has been and continues to be a subject of intense focus in the NLP community, and potential solutions can be adopted from the NLP literature.

Beyond mechanical solutions that decompose or otherwise simplify the inputs to the segmentation and statement translation models [29], NLP researchers have been exploring transformer architectures that leverage sparse attention to handle longer sequences [80, 81]. Because only control flow statements can induce long-range dependencies in segmentation, future work may improve the coverage of neural decompilers by incorporating guided sparse attention into the segmentation model. Combined with a sliding window approach, new model architectures are a promising direction for processing long sequences of code.

Data lag for new language features. For models to effectively learn to segment and translate a given language structure, that structure must be adequately present in the training data. Although we have constructed a continuously evolving dataset using real-world source code from PyPI, the representation of new language features in the dataset is dependent on the speed of user adoption of those features. According to data from the JetBrains Python developers surveys [82–84], Python 3.9 adoption was only 12% in 2020 but rose to 35% in 2021, until falling to 23% in 2022 due to Python 3.10's explosive 45% adoption rate. More research is needed on the time it takes for the new language features in each version to gain sufficient representation in datasets collected from real-world deployment. Future works may explore meta-learning, super-sampling, and artificial data generation to reduce the reliance on user adoption of new features to train effective models.

Automation of control flow reconstruction. To scale across language versions, PYLINGUAL relies on three components that require manual maintenance: *(1)* version-agnostic Python disassembler [12], *(2)* code normalization to mask variable names and constant values, and *(3)* version-specific control flow reconstruction. While components *(1)* and *(2)* demand minimal engineering efforts, due to an open-source project for the version-agnostic Python disassembler and the relatively simple code normalization process, the control flow reconstruction has required significant work and greatly influences decompilation accuracy.

The growing adoption of bytecode-level optimizations extending beyond basic block boundaries introduces maintenance challenges for PYLINGUAL's control flow reconstruction module. Expecting more aggressive optimizations in future versions, we will develop a GNN (Graph Neural Network)-based strategy to automatically train pluggable control flow reconstruction models for new Python releases.

Applying PYLINGUAL to other languages. The direction of our research hinges on our ability to extend PYC decompilation to other programming languages. While we prioritize Python binaries, it is essential to provide reversing support to other HDLs. From our experience with Python, there are certain criteria that indicate that a language will benefit significantly from PYLINGUAL's analysis. These include: *(1)* A modular code object structure, *(2)* A rich source of source code datasets, *(3)* Availability of auxiliary or debugging information, and *(4)* Language-specific optimizations. Given these considerations, we're exploring HDLs with execution models similar to Python, such as Lua and Ruby, as prospective candidates for PYLINGUAL's analysis.

Human-in-the-loop decompilation. Recent work [19, 66] has begun to scratch the surface of systematically under-

standing the interactions between human reverse engineers and their assistive tools. Perfect decompilation provides a mechanism for an automatic decompilation system to recognize and localize decompilation failures to request assistance from a human reverser, as well as provides direct feedback to the human when the failure has been resolved. An application of this feedback loop is illustrated in §8.3. Future works may explore the synergistic and iterative relationship between reverse engineers and assistive tooling, which becomes promising in the context of recently popularized "copilot" systems powered by generative AI.

11. Conclusion

PyLingual's innovative design balances traditional binary analysis principles with data-driven statistical approximations. The rigorous code equivalence requirements of perfect decompilation address the inability of NLP models to deterministically comply with strict syntactic and semantic accuracy requirements in high-stakes domains. Further, for outputs that are not provably correct, PyLingual automatically localizes semantic errors to aid reverse engineers. PyLingual represents the first research effort to address translation instability due to weakly-defined binary interfaces and continuously evolving language versions, and impacts real-world reverse engineers by scaling Python decompilation support across versions.

Evaluated against an extensive collection of real-world datasets, PyLingual achieved a high perfect decompilation rate of 75% on average across Python 3.6 ~ 3.12, marking an average improvement of 45% over SOTA Python decompilers [13, 14, 24] across four datasets. To promote progress in this research field, we will release associated research artifacts, encompassing source code, benign and malicious sample datasets, and established models, and we have launched PyLingual as a public online decompilation service at https://pylingual.io.

Acknowledgements

We thank David Wank and Anthony Maranto for their foundational bytecode manipulation tools. We also thank Jordan Frimpter and Albert Jean for the PyLingual.io user interface. Finally, we thank Dr. Kevin Hamlen for his assistance with formalization. This work was supported in part by Eugene McDermott Graduate Fellowship 202208, as well as NSF grants NSF-2331424 and NSF-2321117. This work was partly supported by the Institute of Information & communications Technology Planning & Evaluation (IITP) grant funded by the Korean government (MSIT) (No.2021-0-01332, Developing Next-Generation Binary Decompiler).

References

[1] Dhiru Kholia and Przemyslaw Wegrzyn. "Looking inside the (Drop) box". In: *USENIX Workshop on Offensive Technologies (WOOT)*. Aug. 2013.

[2] Alessandro Di Pinto, Younes Dragoni, and Andrea Carcano. "TRITON: The First ICS Cyber Attack on Safety Instrument Systems". In: Black Hat USA. Aug. 2018.

[3] Dragos Inc. *TRISIS Malware*. Tech. rep. Dec. 2017. URL: https://dragos.com/wp-content/uploads/TRISIS-01.pdf.

[4] Cyborg Security. *Python Malware On The Rise*. https://www.cyborgsecurity.com/cyborg_labs/python-malware-on-the-rise/. July 2020.

[5] Vasilios Koutsokostas and Constantinos Patsakis. *Python and Malware: Developing Stealth and Evasive Malware Without Obfuscation*. https://arxiv.org/pdf/2105.00565.pdf.

[6] Ian Kenefick et al. *A Closer Look at the Locky Poser, PyLocky Ransomware*. https://www.trendmicro.com/en_us/research/18/i/a-closer-look-at-the-locky-poser-pylocky-ransomware.html.

[7] Josh Grunzweig. *Python-Based PWOBot Targets European Organizations*. https://unit42.paloaltonetworks.com/unit42-python-based-pwobot-targets-european-organizations/.

[8] Claud Xiao, Cong Zheng, and Xingyu Jin. *Xbash Combines Botnet, Ransomware, Coinmining in Worm that Targets Linux and Windows*. https://unit42.paloaltonetworks.com/unit42-xbash-combines-botnet-ransomware-coinmining-worm-targets-linux-windows/.

[9] *PyInstaller Quickstart — PyInstaller bundles Python applications*. https://www.pyinstaller.org/.

[10] *py2exe*. https://www.py2exe.org/.

[11] *PyInstaller Extractor*. https://github.com/extremecoders-re/pyinstxtractor.

[12] Rocky Bernstein. *python-xdis*. https://github.com/rocky/python-xdis.

[13] *uncompyle6 · PyPI*. https://pypi.org/project/uncompyle6/.

[14] *decompyle3 · PyPI*. https://pypi.org/project/decompyle3/.

[15] Skip Montanaro. *A Peephole Optimizer for Python*. 1998.

[16] *faster-cpython*. https://github.com/faster-cpython/ideas.

[17] Mark Shannon. *PEP 659 – Specializing Adaptive Interpreter — peps.python.org*. https://peps.python.org/pep-0659/.

[18] Carl Meyer. *PEP 709 – Inlined comprehensions — peps.python.org*. https://peps.python.org/pep-0709/.

[19] Kevin Burk et al. "Decomperson: How Humans Decompile and What We Can Learn From It". In: *USENIX Security Symposium (USENIX Security)*. USENIX Association, Aug. 2022.

[20] Eric Schulte et al. "Evolving Exact Decompilation". In: *Proceedings 2018 Workshop on Binary Analysis Research* (2018). DOI: 10.14722/bar.2018.23008.

[21] Hamel Husain et al. "CodeSearchNet Challenge: Evaluating the State of Semantic Code Search". In: *arXiv* cs.LG (Sept. 2019). eprint: 1909.09436. URL: arXiv.org.

[22] *PyPI - The Python Package Index*. https://pypi.org/.

[23] virustotal. *metasploit*. https://www.virustotal.com/gui/home/upload. 2021.

[24] *zrax/pycdc: C++ python bytecode disassembler and decompiler*. https://github.com/zrax/pycdc.

[25] Joshua Wiedemeier et al. "PyLingual: A Python Decompilation Framework for Evolving Python Versions". In: *BlackHat USA*. BlackHat USA. 2024.

[26] Joshua Wiedemeier. *There and Back Again: Reverse Engineering Python Binaries - - PyCon US 2024*. May 2024.

[27] Zhibo Liu and Shuai Wang. "How far we have come: testing decompilation correctness of C decompilers". In: *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ISSTA 2020. Virtual Event, USA: Association for Computing Machinery, 2020, pp. 475–487. ISBN: 9781450380089.

[28] Nicolas Harrand et al. "The Strengths and Behavioral Quirks of Java Bytecode Decompilers". In: *2019 19th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. 2019, pp. 92–102. DOI: 10.1109/SCAM.2019.00019.

[29] Ali Ahad et al. "PYFET: Forensically Equivalent Transformation for Python Binary Decompilation". In: *IEEE Symposium on Security and Privacy (SP)*. May 2023.

[30] *Python Documentation by Version*. https://www.python.org/doc/versions/.

[31] *faster-cpython/plan.md at master · markshannon/faster-cpython*. https://github.com/markshannon/faster-cpython/blob/master/plan.md.

[32] *"Zero cost" exception handling · Issue #84403 · python/cpython*. https://github.com/python/cpython/issues/84403. (Accessed on 04/28/2024).

[33] *uncompyle*. https://github.com/gstarnberger/uncompyle.

[34] *Partial 3.12 Support*. https://github.com/rocky/python-xdis/pull/122. (Accessed on 2024-04-30). 2023.

[35] *3.12 Opcodes*. https://github.com/rocky/python-xdis/pull/124. (Accessed on 2024-04-30). 2023.

[36] *3.12 and 3.11 cross-version improvements*. https://github.com/rocky/python-xdis/pull/126. (Accessed on 2024-04-30). 2023.

[37] *3.11 exception table*. https://github.com/rocky/python-xdis/pull/108. (Accessed on 2024-04-30). 2023.

[38] *3.11 Fix calculation of JUMP_BACKWARD jump target*. https://github.com/rocky/python-xdis/pull/109. (Accessed on 2024-04-30). 2023.

[39] *3.11 correct definition of CALL instruction*. https://github.com/rocky/python-xdis/pull/112. (Accessed on 2024-04-30). 2023.

[40] Wei Zhao et al. *Improving Grammatical Error Correction via Pre-Training a Copy-Augmented Architecture with Unlabeled Data*. 2019. arXiv: 1903.00138 [cs.CL].

[41] Jacob Devlin et al. *BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding*. 2019. arXiv: 1810.04805 [cs.CL]. URL: https://arxiv.org/abs/1810.04805.

[42] Shuoheng Yang, Yuxin Wang, and Xiaowen Chu. "A Survey of Deep Learning Techniques for Neural Machine Translation". In: *CoRR* abs/2002.07526 (2020). arXiv: 2002.07526. URL: https://arxiv.org/abs/2002.07526.

[43] Yue Wang et al. *CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation*. 2021. arXiv: 2109.00859 [cs.CL].

[44] *dis — Disassembler for Python bytecode*. https://docs.python.org/3/library/dis.htmll.

[45] *Full Python Grammar Specification*. https://docs.python.org/3/reference/grammar.html.

[46] Aydar Bulatov, Yuri Kuratov, and Mikhail S. Burtsev. *Scaling Transformer to 1M tokens and beyond with RMT*. 2023. arXiv: 2304.11062 [cs.CL].

[47] Bo Peng et al. *RWKV: Reinventing RNNs for the Transformer Era*. 2023. arXiv: 2305.13048 [cs.CL].

[48] Chu-Cheng Lin et al. *Limitations of Autoregressive Models and Their Alternatives*. 2021. arXiv: 2010.11939 [cs.LG].

[49] Jeanne Ferrante, Karl Ottenstein, and Joe Warren. "The program dependence graph and its use in optimization". eng. In: *ACM transactions on programming languages and systems* 9.3 (1987), pp. 319–349. ISSN: 0164-0925.

[50] D. Tsichritzis. "The Equivalence Problem of Simple Programs". In: *J. ACM* 17.4 (Oct. 1970), pp. 729–738. ISSN: 0004-5411. DOI: 10.1145/321607.321621. URL: https://doi.org/10.1145/321607.321621.

[51] Colin Raffel et al. "Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer". In: *CoRR* abs/1910.10683 (2019). arXiv: 1910.10683. URL: http://arxiv.org/abs/1910.10683.

[52] *PyLingual*. https://www.pylingual.io/. (Accessed on 04/27/2024).

[53] *How to Turn your .EXE files back to precious Python code! — Arcane Codex — Readers Hope*. https://medium.com/readers-digests/how-to-turn-your-exe-files-back-to-precious-code-01eb2863cac7. (Accessed on 04/27/2024).

[54] *CTF-CheatSheet — Eritque arcus's blog*. https://ikuyo.dev/ctf-cheatsheet/. (Accessed on 04/27/2024).

[55] Charles R. Harris et al. "Array programming with NumPy". In: *Nature* 585.7825 (Sept. 2020), pp. 357–362. DOI: 10.1038/s41586-020-2649-2. URL: https://doi.org/10.1038/s41586-020-2649-2.

[56] Takemaru Kadoi. *PyChD: ChatGPT-powered Python Decompiler*. https://github.com/diohabara/pychd/tree/main. Accessed: 2024-10-10. 2024.

[57] OpenAI et al. *GPT-4 Technical Report*. 2024. arXiv: 2303.08774 [cs.CL]. URL: https://arxiv.org/abs/2303.08774.

[58] Chris Eagle. *The IDA Pro Book: The Unofficial Guide to the World's Most Popular Disassembler*. USA: No Starch Press, 2011. ISBN: 1593272898.

[59] Chris Delikat Brian Knighton. *Ghidra - Journey from Classified NSA Tool to Open Source*. Aug. 2019.

[60] Radare2 Team. *Radare2 GitHub repository*. https://github.com/radare/radare2. 2017.

[61] *RetDec :: Home*. https://retdec.com/.

[62] Cristina Cifuentes. "Reverse Compilation Techniques". PhD thesis. 1994.

[63] Mike Van Emmerik. "Static Single Assignment for Decompilation". PhD thesis.

[64] David Brumley et al. "Native x86 Decompilation Using Semantics-Preserving Structural Analysis and Iterative Control-Flow Structuring". In: *USENIX Security Symposium (SEC)*. Washington, D.C., July 2013.

[65] Khaled Yakdan et al. "No More Gotos: Decompilation Using Pattern-Independent Control-Flow Structuring and Semantics-Preserving Transformations". In: *Network Distributed Security Symposium (NDSS)*. Feb. 2015.

[66] Khaled Yakdan et al. "Helping Johnny to Analyze Malware: A Usability-Optimized Decompiler and Malware Analysis User Study". In: *2016 IEEE Symposium on Security and Privacy (SP)*. May 2016, pp. 158–177.

[67] Richard Wartell et al. "Differentiating Code from Data in x86 Binaries". In: *Machine Learning and Knowledge Discovery in Databases*. Ed. by Dimitrios Gunopulos et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 522–536. ISBN: 978-3-642-23808-6.

[68] Eui Chul Richard Shin, Dawn Song, and Reza Moazzezi. "Recognizing Functions in Binaries with Neural Networks". In: *USENIX Security Symposium (SEC)*. Washington, D.C.: USENIX Association, Aug. 2015.

[69] Omer Katz et al. *Towards Neural Decompilation*. 2019. arXiv: 1905.08325 [cs.PL].

[70] Cheng Fu et al. "Coda: An End-to-End Neural Program Decompiler". In: *Advances in Neural Information Processing Systems*. Ed. by H. Wallach et al. Vol. 32. Curran Associates, Inc., 2019. URL: https://proceedings.neurips.cc/paper_files/paper/2019/file/093b60fd0557804c8ba0cbf1453da22f-Paper.pdf.

[71] Ruigang Liang et al. "Neutron: an attention-based neural decompiler". In: *Cybersecurity* (May 2021). DOI: 10.1186/s42400-021-00070-0. URL: https://cybersecurity.springeropen.com/articles/10.1186/s42400-021-00070-0.

[72] *Old Dogs New Tricks: Attackers adopt exotic programming languages*. Tech. rep. BlackBerry Research & Intelligence Team, 2021. URL: https://blogs.blackberry.com/en/2021/07/old-dogs-new-tricks-attackers-adopt-exotic-programming-languages.

[73] *This malware was written in an unusual programming language to stop it from being detected — ZDNet*. https://www.zdnet.com/article/this-malware-was-written-in-an-unusual-programming-language-to-stop-it-from-being-detected/.

[74] Rocky Bernstein. *python-xdis*. https://github.com/rocky/python-spark.

[75] Raja Vallée-Rai et al. "Soot - a Java Bytecode Optimization Framework". In: *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research*. CASCON '99. Mississauga, Ontario, Canada: IBM Press, 1999, p. 13.

[76] *icsharpcode/ILSpy: .NET Decompiler with support for PDB generation, ReadyToRun, Metadata (&more) - cross-platform!* https://github.com/icsharpcode/ILSpy.

[77] *dotPeek: Free .NET Decompiler & Assembly Browser by JetBrains*. https://www.jetbrains.com/decompiler/.

[78] *cout/ruby-decompiler: A decompiler for ruby code (both MRI and YARV)*. https://github.com/cout/ruby-decompiler.

[79] *Tool: Lua 5.1 Decompiler*. https://lua-decompiler.ferib.dev/.

[80] Manzil Zaheer et al. "Big Bird: Transformers for Longer Sequences". In: *arXiv* (Jan. 2021). arXiv: 2007.14062. URL: https://arxiv.org/abs/2007.14062.

[81] Iz Beltagy, Matthew E. Peters, and Arman Cohan. "Longformer: The Long-Document Transformer". In: *CoRR* abs/2004.05150 (2020). arXiv: 2004.05150. URL: https://arxiv.org/abs/2004.05150.
[82] JetBrains and Python Software Foundation. *Python Developers Survey 2020*. Accessed: 2024-06-06. 2021. URL: https://www.jetbrains.com/lp/python-developers-survey-2020/.
[83] JetBrains and Python Software Foundation. *Python Developers Survey 2021*. Accessed: 2024-06-06. 2022. URL: https://lp.jetbrains.com/python-developers-survey-2021/.
[84] JetBrains and Python Software Foundation. *Python Developers Survey 2022*. Accessed: 2024-06-06. 2023. URL: https://lp.jetbrains.com/python-developers-survey-2022/.
[85] Kotaro Ogino. *Unboxing Snake - Python Infostealer Lurking Through Messaging Services*. https://www.cybereason.com/blog/unboxing-snake-python-infostealer-lurking-through-messaging-service.
[86] Josh Grunzweig. *Unit 42 Technical Analysis: Seaduke*. https://unit42.paloaltonetworks.com/unit-42-technical-analysis-seaduke/.
[87] jinye. *Necro Frequent Upgrades, New Version Begins Using PyInstaller and DGA*. https://blog.netlab.360.com/not-really-new-pyhton-ddos-bot-n3cr0m0rph-necromorph/.
[88] Warren Mercer. *PoetRAT: Python RAT uses COVID-19 lures to target Azerbaijan public and private sectors*. https://blog.talosintelligence.com/poetrat-covid-19-lures/.

Appendix A. Technical Details and Online Service

We provide technical descriptions of top-k segmentation search and corrector heuristics for statement translation model. Then, we introduce PyLingual online service and discuss its legal and ethical considerations. Table 7 presents links to the PyLingual web service for the case study samples presented in §8.

Table 7: PyLingual decompilation for case studies and code examples.

Case study analysis	PyLingual decompilation
Figure 9	https://pylingual.io/bcfcf97
Figure 10	https://pylingual.io/9973b72
Figure 11	https://pylingual.io/4c5de2b
Figure 13	https://pylingual.io/4c5de2b

A.1. Model Training Configuration

This section contains technical details for the segmentation, statement translation, and corrector models.

Segmentation. For each covered Python version, the segmentation model follows an encoder-only BERT [41] architecture, consisting of: *(1)* a 768-wide embedding layer with a vocabulary of 30,000 tokens and a maximum position embedding of 2,050; *(2)* twelve transformer blocks, each consisting of a 768-wide self-attention layer, a 768 wide-dense layer with normalization, a 768-in 3072-out dense layer with normalization, and finally a 3072-in 768-out dense layer with normalization; *(3)* a 768-in 3-out single-layer linear classifier. Each model is randomly initialized and initially trained to perform masked language modeling for Python bytecode (without the final linear classifier layer) to prepare appropriate embeddings. This initial training period lasts for 2 epochs with a batch size of 2 and a learning rate of 2e-5. The resulting model is then fine-tuned to perform BIE segmentation using the linear classifier layer, following the same hyperparameter settings. The segmentation model architecture has a total of 109,673,475 (≈ 110M) parameters.

Statement Translation. For each covered python version, the statement translation model is a fine-tuned version of CodeT5 [43] trained to perform sequence-to-sequence mapping between Python bytecode statements and Python source code statements. The statement translation models used our custom Python tokenizer (§A.2) to improve the semantic density of tokens during translation. Each model was trained for 2 epochs with a batch size of 24 and a learning rate of 2e-5. The translation model architecture has a total of 222,882,048 (≈ 223M) parameters.

Statement Corrector. For each covered python version, the statement corrector model is a fine-tuned version of the corresponding statement translation model, so their architectures are identical. For "difficult" bytecode sequences (§A.4), the training data for the statement corrector is pairs of ((original bytecode, translation model output), original source code). The training hyperparameters are the same as for statement translation, but with a smaller batch size of 8 to accommodate the longer training inputs.

A.2. Custom Python Tokenizer

Taking advantage of the structure of Python source code and bytecode compared to natural language, we were able to improve the semantic coherence of tokens by initializing an off-the-shelf Roberta [41] tokenizer with additional special tokens for each literal in the Python source grammar [45], operator name in the Python bytecode specification [44], and additional common literals introduced during our code normalization process (§5.1). By integrating the Python grammar with the tokenization process, there is a closer match between the tokenized feature-space translation task and the Python problem space. This tokenizer includes the relevant tokens for all covered Python versions, enabling it to be used as a common tokenizer across all of our statement translation and corrector models. Although the current version of the tokenizer requires a small amount of annual maintenance to keep up with Python releases, the relevant information could be automatically scraped from the Python documentation with minimal engineering effort.

A.3. Top-k Segmentation Search

To address the limitations of the NLP-based segmentation module, PyLingual performs a local search in the space of segmentations, guided by the confidence of the segmentation model. This often enables the recovery of a correct segmentation of a code object, when the originally predicted segmentation contains errors. We assert that accurately segmenting bytecode streams regarding their statement boundaries plays an essential role in achieving accurate decompilation results.

Here, we describe the family of m-deep-top-k search strategies, from which PyLingual's 2-deep confidence-guided segmentation search was derived. Any segmentation

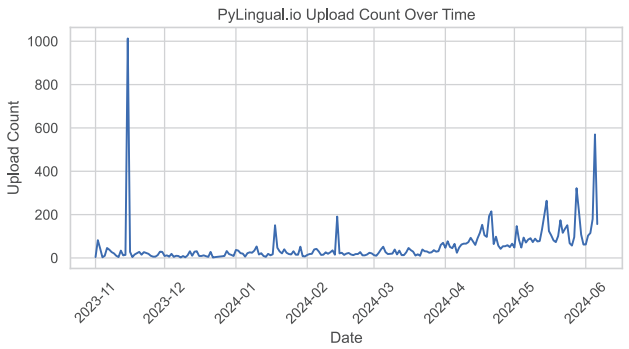


Figure 14: Uploads to PyLingual.io since November 2023.

mechanism produces, for each disassembled bytecode instruction, *(1)* an indication of if the instruction begins a new statement, and *(2)* a set of features (*e.g.,* the confidence score from the corrector model); the sequence of these outputs for a given code object constitutes a segmentation.

In an m-deep-top-k strategy, we map the segmentation to a binary string s of length n, where each bit corresponds to one instruction, and is 1 if the corresponding instruction begins a new statement. The search then proceeds over a set of "corrector masks", which are also binary strings of length n, where a bit is 1 if the segmentation decision at the corresponding instruction should be flipped. To generate a corrected segmentation s' with a corrector mask c, we can simply compute $s' = s \oplus c$. The "search distance" m represents the maximum number of errors that can be corrected by the search, and is therefore the maximum number of 1s in the corrector masks that are considered. The flexibility of m-deep-top-k searching stems from strategy-specific priority functions, which establish a total ordering over the corrector masks; in PyLingual's case, we prioritize exploration in order of least-confidence, such that the statement boundaries that the model is unsure about is altered first. Finally, k provides a constant upper bound on the number of variants to search, ensuring that not too much time is wasted searching low-priority candidates.

Because only binary strings with at most m ones are considered, the total search space is

$$\sum_{k=0}^{m} \binom{n}{k} = 1 + n + \frac{n(n-1)}{2} + \cdots + \frac{n!}{m!(n-m)!}$$

where n is the number of instructions being segmented. The asymptotically dominant term is $\frac{n!}{m!(n-m)!}$, for which we can show that $\frac{n!}{m!(n-m)!} < n^m$. Therefore, the total search space is $O(n^m)$, which is a significant reduction from the original exponential search space of all length n binary strings. The key strength of m-deep-top-k searching is that when the initial segmentation is expected to be close to a correct segmentation, a low constant m can include a correct segmentation with high probability in a polynomial slice of the exponential search space.

A.4. Statement Corrector Heuristic

Heuristically, "difficult" statements: *(1)* contain type annotations or default arguments; *(2)* are comprehensions; *(3)* contain four or more function calls, jumps, or sequence creations; or *(4)* contain six or more binary operations.

A.5. PyLingual Online Service Usage

In November 2023, we began hosting PyLingual online service to improve accessibility to Python decompilation tools. Since then, word slowly began to spread, and a steady stream of users began to visit the site to decompile Python bytecode. Figure 14 shows the daily upload trends since the service's launch. The first three large spikes were caused by ambitious users who scripted out a bulk decompilation of a full project; for those projects that appear to be private intellectual property, we are in communication with our University's legal department to ensure responsible disclosure. The fourth and longest spike in usage coincided with a Capture-The-Flag (CTF) event which involved Python reversing.

PyLingual's growing traction in the community [25, 26, 53, 54], along with over 7,500 GitHub stars across `uncompyle6`, `decompyle3`, and `pycdc`, demonstrate clear and present demand for Python decompilation tools. However, even with PyLingual improving the perfect decompilation rate by 17.1% to 45.5% over the next best decompiler on files that real users want to reverse engineer, there is still a large gap for improvement in the future.

Data collection and ethical considerations. We discussed relevant concerns and perspectives with our university's legal department and IRB to design PyLingual.io's data collection and handling procedures. To support basic operations, the web service logs source IPs and timestamps of API calls, and stores uploaded `PYC` files to be served back to users. Our university's IRB does not consider these items to be personally identifiable information, and therefore this research does not constitute human subjects research. However, we recognize that this research is supported in part by `PYC` files contributed by human users, so we have taken appropriate measures to respect their privacy and anonymity. The web service prominently presents a notice that uploaded files may be used for research purposes, which must be accepted before using the service. The uploaded files and decompiled source code may be automatically deleted at the user's request using the red delete button in the top-right corner of the decompilation result page. The web service does not attempt to identify users beyond logging IP addresses associated with API calls, and maintains no cookies or session information.

Appendix B. Meta-Review

The following meta-review was prepared by the program committee for the 2025 IEEE Symposium on Security and Privacy (S&P) as part of the review process as detailed in the call for papers.

B.1. Summary

This paper proposes a new technique in the decompilation of Python bytecode: a hybrid, ML-and-PL system that intrinsically adapts to changes in the Python language as the latter evolves. This adaptability enables PyLingual to maintain its efficacy across 7 versions of Python (the two other leading decompilers only support a union of three, older Python versions) while maintaining a significant success rate.

B.2. Scientific Contributions

- Provides a New Data Set For Public Use
- Creates a New Tool to Enable Future Science
- Addresses a Long-Known Issue
- Provides a Valuable Step Forward in an Established Field

B.3. Reasons for Acceptance

1) The authors have explored and addressed a long-standing problem in Python decompilation.
2) The reviewers appreciated the careful integration of the ML and PL components of the pipeline, and observed that it might work on other HDLs.
3) The availability of the source and service of PyLingual is a good contribution for future research.