PyLingual: A Python Decompilation Framework for Evolving Python Versions

Josh Wiedemeier, Elliot Tarbet, Max Zheng, Jerry Teng, Ximeng Liu,
Muhyun Kim, Sang Kil Cha, Jessica Ouyang, Kangkook Jee

Abstract

Python has become a popular choice for creating malware due to its ease of development, wide user base, pre-built modules, and multi-platform compatibility. Python's popularity has induced demand for Python decompilers, but community efforts to maintain automatic Python decompilation tools have been hindered by Python's 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 adapts to new language features and changes to the bytecode specification with minimal human maintenance effort. PyLingual uses data-driven NLP components to automatically absorb superficial bytecode and compiler changes, while leveraging engineered programmatic components for abstract control flow reconstruction.

We demonstrate the efficacy of our approach with extensive real-world datasets of benign and malicious Python sources 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 — 3.12; *(2)* we provide a Python decompiler evaluation framework that verifies decompilation results with "perfect decompilation"; and *(3)* we launch PyLingual as a free online service [1].

1 Introduction

Python has been an attractive choice for hackers and industry developers alike due to its ease of development, wide user base, mature ecosystem with pre-built modules, and multi-platform compatibility [1]–[12]. Malware writers aim to budget their time and resources in a way that maximizes their productivity, thus demanding environmental 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 [13], [14] to create a standalone executable. Python decompilers reverse the above process: unpacking the packaged executable to extract the PYC binaries [15], disassembling them into bytecode sequences [16], and ultimately recovering the source code [17], [18].

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 [17], [18].

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. Every year, 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*. Further, recent Python versions have been adopting aggressive optimizations [19]–[22] 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 feasible, the maintenance effort required to provide cross-version support that quickly adapts to new version releases is daunting [17], [18].

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

[1] `pylingual.io`

PyLingual aims to demonstrate the effectiveness of ML-based statistical approaches to correctness-sensitive, PL domains. While offloading simple but labor-intensive translation tasks to NLP models, PyLingual falls back to classical PL principles 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 for each to be as self-contained as possible, minimizing engineering friction between them. To validate decompilation outputs, PyLingual boldly embraces perfect decompilation, which enforces strict code equivalence between the input PYC binary and the decompiled source code, and facilitates a feedback loop to detect and understand decompilation errors.

Our research is built on an extensive collection of real-world datasets from both benign and malicious sources [23]–[25], which we plan to publish alongside source code and established models. Evaluated against an extensive collection of real-world datasets, PyLingual achieves a 77% perfect decompilation rate on average across Python 3.6 - 3.12, marking an average improvement of 47% over State-Of-The-Art (SOTA) Python decompilers [17], [18], [26]. Demonstrating PyLingual's adaptability, we were able to seamlessly extend PyLingual to support Python version 3.12 with only two weeks of effort.

PyLingual makes the following contributions:

- PyLingual explores a unique design direction integrating principled PL theories with neural NLP models.
- We introduce a rigorous "perfect decompilation" metric for evaluating the correctness of decompilation results using differential testing against the input binary.
- We evaluate PyLingual across a wide range of Python versions with the extensive dataset from benign and malicious sources using the proposed metric.

To assist reverse engineers and future research efforts, we plan to publish our source code, datasets, and models, and we provide PyLingual as a free online service [2] to offer streamlined support to reverse engineers.

2 Perfect Decompilation

In our work, we emphasize the impact of considering *perfect decompilation* [27] in the design of automatic decompilers. Decompiled code is "perfect" when recompiling that code with the compiler configuration as the original source code, the exact same instructions are produced. Similar terms for the same general idea are: "exact decompilation", "round-trip testing", and "function/inverse pairs".

Benefits to automatic decompilers. Perfect decompilation provides a strong guarantee of semantic equivalence between the input binary and the decompiled source code, and is trivial to verify by simply testing for strict equality between the original and candidate instruction sequences. These two properties enable a black-box decompiler to easily prove the correctness of perfectly decompiled code, which enables users to trust the decompiler's output, even if they do not trust the soundness of the decompiler.

Traditional measures of decompilation accuracy, such as equivalence modulo inputs [28], [29] or manual verification, provide limited semantic equivalence guarantees and are expensive to measure, which often results in misleading decompilation that users are unable to efficiently verify [30]. In perfect decompilation, decompilation correctness errors are easily detected at runtime, preventing unknown decompiler bugs from eroding trust in the decompilation results.

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 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, we will see in §3 that the core challenges of Python decompilation are quite different from those of traditional binary decompilation.

3 Python Bytecode

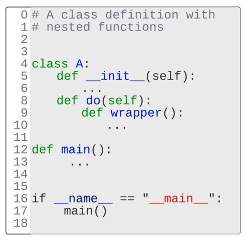
We provide background on the structure of Python bytecode, summarize its key properties with respect to decompilation, and briefly discuss existing Python decompilation approaches and their pitfalls.

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. The code in the top-level script is the "__main__" code object, and several language features such as list comprehensions and lambda expressions are implemented as anonymous code objects. 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

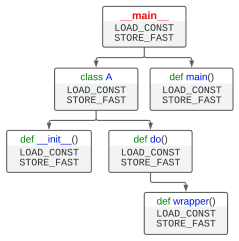
[2] `https://pylingual.io`

tracebacks. For perfect decompilation, only the bytecode instructions and semantically important metadata need to match between the original binary and the decompiled result.



```
# 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 corresponding nested code object structure.

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.

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 `print` function. At compile time, the code object has no way of knowing if `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.

Unfortunately, Python's exception handling structures are more complex, as they combine jumps with the exception control flow mechanism. Exceptions can be raised at any time by any instruction, with different control implications depending on the current execution context. Further, as we discuss next, the exception handling system has been the subject of substantial modification in recent years, complicating the proposition of cross-version support.

3.3 Specification Instability

The most significant challenge in Python decompilation is the instability of the Python bytecode specification. Since its initial 1991 release, 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 [31], including the addition, deletion, and modification of instruction opcodes. Recently, the Python community committed to the "Faster CPython" project [32], resulting in several optimizations that emphasize reordering instructions to reduce the time spent by the interpreter managing control flow. Of these optimizations, the most noteworthy was the introduction of "zero-cost exceptions" in Python 3.11, which completely reworked the exception handling mechanism and related bytecode generation for exception handling structures.

3.4 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 [17], [18]. `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.

Some existing Python decompilation frameworks depend on version-specific grammars and statement patterns, and have been unable to keep pace with dramatic bytecode changes on Python's now annual release cycle. Since the launch of Python 3.9 in October 2020, existing decompilers have failed to provide sufficient coverage for serious reverse engineering. While there have been efforts to improve the coverage of these decompilers through input preprocessing [30], ad-hoc error correction producing unsound approximations is not a sustainable approach for the long-term scalability of Python decompilation.

4 PyLingual Overview

As shown in Figure 2, PyLingual operates in five stages centered around three major components. First, PyLingual conducts *code normalization* against the source code and disassembled bytecodes [16] to reduce the complexity of the inputs to the NLP models (§4.1). Second, normalized code objects are provided to the *bytecode segmentation* component to divide the bytecode stream by statement boundaries (§4.2). Next, the *statement translation* component translates each statement of bytecode into the corresponding Python source code statement (§4.3). Then, the *control flow reconstruction* component mechanically reconstructs the necessary indentation to reproduce the control flow in the input bytecode (§4.4). Finally, *code equivalence verification* conducts instruction-level code comparison to validate the correctness of the decompiled Python source code (§4.5).

4.1 Code Normalization

To make PYC bytecode suitable for the segmentation and translation tasks, PyLingual replaces distracting details such as variable names and constant values with generic masks [34], which can be easily located in dedicated lookup tables in the PYC binary; a generic mask derived from the table index will be seen by the neural components, and the original value will be mechanically restored at the source level at the end of decompilation. This is similar to the idea of "tokenization" in the context of compilers, which is different from "tokenization" as used in language modelling; we use the term "masking" to avoid confusion between these concepts. Further, PyLingual enhances the presentation of the bytecode instructions by annotating jump targets and exception-handling structures using the PYC metadata. This input preprocessing step allows PyLingual to standardize the inputs to the NLP models, even when the bytecode representation changes significantly.

To disassemble PYC files from different Python versions, PyLingual uses `xdis`, a version-agnostic, open-source disassembler [16], to which we contributed supporting code to disassemble Python 3.11 and Python 3.12 binaries.

4.2 Bytecode Segmentation

Given a sequence of disassembled bytecode instructions, the segmentation module has two goals: *(1)* divide the bytecode into independently digestible statements; and *(2)* establish an association between the bytecode instructions and their corresponding statements. Statement-level segmentation is convenient because it is simple to collect ground-truth segmentations from known source code samples using debugging symbols from the compiler, which we can use to train a model for use in decompilation.

Alternatively, we can consider using segmenting at basic block boundaries, which are also easy to identify mechanically. However, Python bytecode statements often go beyond control block boundaries, causing common control statements to spread across several blocks, which presents significant challenges to the downstream translation component. Furthermore, basic blocks can include arbitrarily many statements, necessitating a comprehensive paragraph-to-paragraph language model. Language models are bound to their input size, requiring larger models that require significantly more data and computing power to train and translate.

The segmentation model, while accurate in most cases, is inherently limited in identifying all statement boundaries. Both mechanical parsing and data-driven analysis fall short, especially across different Python versions. The segmentation problem's core nature complicates the matter: bytecode segmentation often embeds the programmer's control flow decisions, which might be apparent only in distant code locations. For example, the decision to split a simple `if A and B:` may have irreparable control flow ramifications later in decompilation, depending on the presence of an `else:` block at some distant point in the source code (Figure 3). 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`). We can only mechanically verify a candidate segmentation by verifying the result of the subsequent decompilation process.

To remediate imperfect segmentation, we employ a differential testing strategy with top-*k* candidate segmentations, relying on PyLingual's strict accuracy assurance. We discuss the details of top-*k* segmentation in §5.

4.3 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 [35]. Extending a pretrained T5 translation model from `Salesforce/codet5-base` [36], we fine-tune the model for the Python bytecode to Python source code translation task.

The statement translation model often struggles with long or complex statements. While we can easily verify syntactic correctness, immediately verifying semantic accuracy is challenging. The same source code statement can generate different bytecode depending on its surrounding context. To address the issue and improve the translation accuracy, we introduce a corrector model tailored for problematic statement samples. In §6, we explore our strategy to augment the statement model and the hurdles in crafting filter rules to pinpoint error-prone statements. The ablation study in §8.4 highlights the benefit of the corrector model.

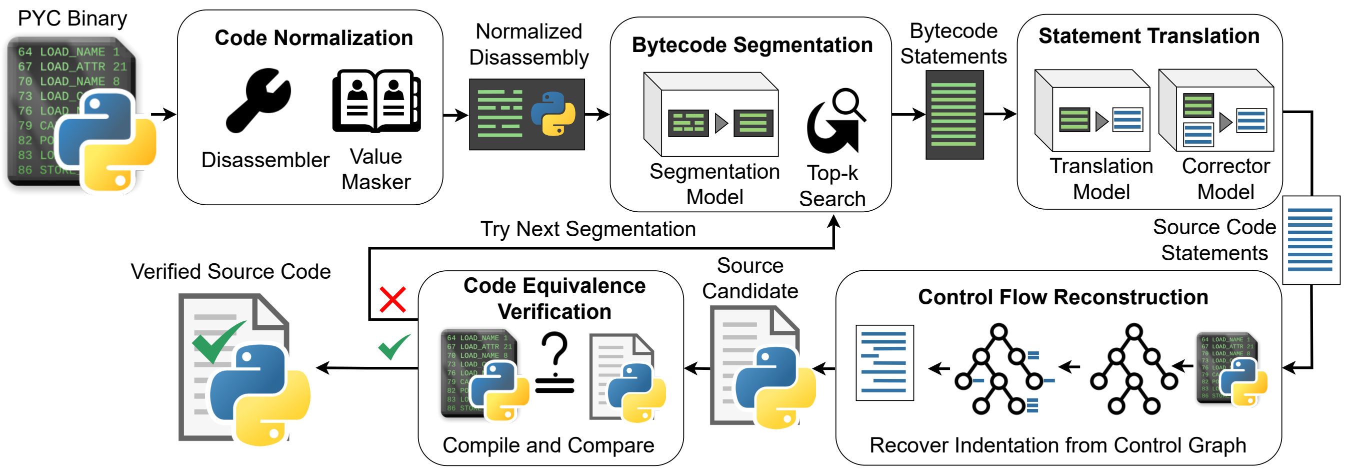
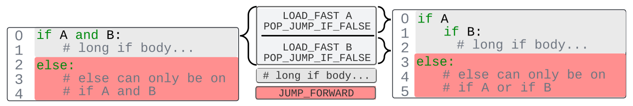


Figure 2: PYLINGUAL architecture.



```
if A and B:
    # long if body...
else:
    # else can only be on
    # if A and B
```

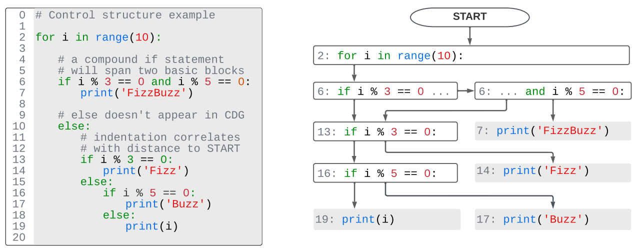
```
LOAD_FAST A
POP_JUMP_IF_FALSE

LOAD_FAST B
POP_JUMP_IF_FALSE

# long if body...

JUMP_FORWARD
```

```
if A
    if B:
      # long if body...
else:
    # else can only be on
    # if A or if B
```

Figure 3: Two different possible bytecode segmentations, resulting in different source-level meanings due to the long-range dependencies.

4.4 Control Flow Reconstruction

Given a flat list of source code statements from the statement translation module, the control flow reconstruction module *mechanically* determines each statement's indentation level to reconstruct the control flow in the original bytecode. By analyzing input bytecode, PYLINGUAL constructs a Control Dependency Graph (CDG) where each node's distance from the START node corresponds to the indentation level. An example is provided in Figure 4. Since Python's control dependencies can be statically determined, we first construct the Control Flow Graph (CFG), which can then be extended to create a CDG [37].



```
 0 # Control structure example
 1
 2 for i in range(10):
 3
 4     # a compound if statement
 5     # will span two basic blocks
 6     if i % 3 == 0 and i % 5 == 0:
 7         print('FizzBuzz')
 8
 9     # else doesn't appear in CDG
10     else:
11         # indentation correlates
12         # with distance to START
13         if i % 3 == 0:
14             print('Fizz')
15         else:
16             if i % 5 == 0:
17                 print('Buzz')
18             else:
19                 print(i)
20
```

(a) Source Code

(b) Control Dependence Graph (CDG)

Figure 4: A control dependence graph illustrating that indentation depth corresponds to the shortest-path distance to the START node.

In the CDG, each node is a basic block, and there exists an edge (u, v) if and only if a control decision in u decides if v *may* execute. Some source lines can span multiple basic blocks; in these cases, we associate the source line with the basic block containing its first instruction. The core algorithm assigns indentation levels to basic blocks by calculating the length of the shortest path to the `START` node. However, several cases require special handling. For example:

(1) certain control flow statements (`else`, `finally`) do not have any bytecode instructions associated with them and therefore cannot be generated by the statement translation model. However, we can recognize that we need to insert an `else` if there is a basic block that is only reachable by jumps from a unique source line. On the other hand, recognizing basic blocks that should have `finally` is not quite as simple, as the bytecode representation of this control flow structure has changed significantly across different Python versions. Detection of this structure usually requires inspection of the CFG for instructions that affect the block stack, a separate stack that the Python interpreter used to use to handle exception handling prior to 3.11. *(2)* certain control flow statements (`break`, `continue`, `return`) can break out of several control structures at once, which creates control dependencies between every basic block that could lead to the premature exit and everything after the premature exit within the control flow structure that would be exited. As an illustrative example, every decision leading to a deeply nested `return` will now have an execution path where every subsequent statement at the same indentation level *may* not execute. Because we only consider the shortest path to `START`, only the top-level added dependency affects the indentation calculation, so we decrease the affected basic block's indentation by one when this behavior is detected. *(3)* some basic blocks do not contain any source lines, which can be caused by source lines that

span multiple basic blocks, such as Boolean expressions with short-circuit evaluation. This can yield indentation levels that are much deeper than the intended value. We address this not counting basic blocks without any corresponding source line along the shortest path to the `START` node, ensuring that each source line is only ever able to contribute one indentation level, even when spread across multiple basic blocks.

As the primary mechanical component in PyLingual, a certain amount of maintenance effort is expected to catch up to the evolving Python versions. For example, Python 3.11 overhauled the bytecode representation of `try` ... `except` structures, which required a follow-up modification on PyLingual's control flow reconstruction module.

4.5 Code Equivalence Verification

The general program equivalency problem is known to be undecidable [38], so to verify the results of PyLingual, we adopt a strict notion of bytecode equivalency that can be efficiently verified. We consider two `PYC` files to be equivalent if and only if all reachable instructions in the two `PYC` files appear in the same order and have the same arguments. An instruction i in a `PYC` file is reachable if and only if there exists a path from the first instruction in the `PYC` file to i in the control flow graph for that `PYC` file. Correct decompilation produces source code that yields bytecode that is equivalent to the original bytecode when compiled. While our extremely strict definition of code equivalence will yield false negatives for imperfect but semantically equivalent code (*e.g.,* independent statements appearing out of order), it will importantly never result 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.

5 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 mechanism will produce, 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 will be 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. In our evaluation and online service, we set $m = 2$.

6 Statement Corrector Model

The statement translation model (§4.3) struggles with some subtasks of bytecode to source code translation. Namely: *(1)* long statements can exceed the input capacity of the model, resulting in a loss of information; and *(2)* lists that must be aligned from the right side (*e.g.,* default and keyword arguments) pose a structural challenge to the translation model's autoregressive decoding scheme. That is, because the model is generating each token of the translation one-by-one from left to right, it doesn't "know" how long the lists will be when it needs to decide how to align them, so it will guess based on common trends. For example, it is very common for function definitions in classes to start with `self` followed by default arguments, so the statement translation model will start outputting default argument values after the first argument.

The above limitations are well-known in the NLP community [39]–[41], and the traditional solution is to train a corrector model which sees the inputs and outputs of the first translation pass, then outputs a corrected result. While the architecture is common practice, the implementation challenge lies in deciding when a given translation is likely to be incorrect. Without an effective filter to limit the number of correct cases sent to the corrector model, the corrector simply learns to accept the original answer, which is correct most of the time. We originally explored methods to validate the correctness of a single bytecode statement translation directly to decide what to forward to the corrector, but the significant context-sensitive variety of bytecode that could correspond to a given source code statement proved to be difficult; we are only able to validate the correctness of statement translations at the end of decompilation, once the surrounding context has been recovered. As an easier and more streamlined alternative, we turned to heuristically determining if a given bytecode statement would be "difficult" to translate. Heuristically, "difficult" statements are those that: *(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. With these rules, we are able to train an effective corrector model, which we demonstrate in our evaluation (§8.4).

7 Implementation

PyLingual consists of NLP models and mechanical components for its training and translation tasks. Written in Python, the source code spans approximately 5.6K lines, excluding contributions to an external open-source project. Along with the datasets and trained models, we will make PyLingual 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 has depended on python-xdis [16] 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-based models. The bytecode segmentation module uses Bidirectional Encoder Representations from Transformers (BERT) [42], an encoder-only language model; our statement translation module uses a code-oriented T5 [43] encoder-decoder language model. Both language models are pre-trained on large amounts of text data. By fine-tuning these pre-trained models on task-specific data, they can transfer knowledge learned from one domain to another, achieving state-of-the-art results on a range of downstream NLP tasks. In PyLingual, we leverage this transfer learning technique to fine-tune generic public code models to perform Python bytecode segmentation and translation. 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 Transformers, Huggingface Datasets, and PyTorch libraries in 938 lines of Python code.

Training data generation. To train the segmentation and statement translation models, we must prepare ground-truth segmented pairs of source code and bytecode. We leverage the CodeSearchNet dataset [23] and our collection of over 1,000,000 real-world Python source files [24] by randomly sampling 80,000 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 will remove unnecessary whitespace, comments, and other irrelevant source-level artifacts. Finally, we apply code normalization (§4.1) to ensure that the data representation during training matches the representation that will be used during decompilation. The training data generation pipeline was 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 implemented 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 code equivalence verifier (177 lines). The decompiler pipeline that ties all the modules together was 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 ≈ 400 lines of version-specific code across the seven Python versions supported at this time.

8 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 accurately decompile Python binaries? (§8.2)
- **RQ2:** Does PyLingual help scale Python decompilation across different versions? (§8.2)
- **RQ3:** What are the benefits of the auxiliary components? (§8.3, §8.4)

First and foremost, we evaluate PyLingual across different Python versions compared to existing Python decompilers. Then, we examine the accuracy and overhead impacts of top-*k* segmentation and the statement corrector model. Finally, we showcase case studies that illustrate the strengths and weaknesses of PyLingual compared to traditional decompilation. Our evaluations were run on the same server from §7, which is equipped with an AMD Threadripper 5955WX CPU, 128 GB of RAM, and one Nvidia RTX 4090 GPU.

8.1 Datasets

Given the data-intensive nature of our research, it is critical to establish extensive datasets from credible sources. Our datasets originate from three sources:

CodeSearchNet (CSN) is an open-source and community-verified dataset of Python source files [23]. Originally designed to support code analysis tasks, the Code Search Net (CSN) dataset is carefully curated by open-source experts to encompass diverse aspects of the Python language. However, CSN is relatively small and only captures a static dataset composition as of its presentation in 2019, which precludes it from representing source-level language features introduced in Python 3.9 and beyond.

Python Package Index (PyPI) is the de facto repository for Python modules, where thousands of developers publish, update, and maintain their projects daily to share with the rest of the community. Our autonomous collection framework follows PyPI to capture the diverse characteristics of real-world users and reflect new features as they are adopted.

VirusTotal provides Python malware samples that were packaged using open-source packager 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.

Table 1 shows the basic compositions of datasets, which we plan to publicize alongside our source code and models. We notice from Table 1 that the number of instructions in each file increases dramatically in VirusTotal as compared to PyPI, and in PyPI as compared to CSN; in §8.2, we will show how this impacts the segmentation model's ability to consume bytecode samples.

Test dataset composition. Throughout our evaluation, we measure performance metrics against a sample of 2,000 Python source code files from CSN, 3,000 Python source code files from PyPI, and all available PYC files from our VirusTotal dataset 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 evaluation run.

Table 1: Dataset summaries. For the source datasets, instruction counts were collected from the test sample and averaged across versions 3.6-3.12.

Dataset	Version	Total # Files	Training Set # Files	Instructions per File (Mean / Std)
CodeSearchNet	source	412,179	40,000	76.0 / 84.8
PyPI	source	1,095,180	40,000	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

8.2 Decompilation Accuracy

Table 2 measures the effectiveness of PyLingual against other SOTA Python decompilers: Uncompyle6, Decompyle3, and Pycdc. To provide a comprehensive view of the decompilation landscape, we examine PYC binaries from research-oriented (CSN), production-oriented (PyPI), and malicious (VirusTotal) environments. The decompiled source for each PYC binary falls into one of four categories: *(1)* it is *Equal* to the input binary (§4.5) and was automatically verified; *(2)* it has *Semantic Errors* and could not be verified; *(3)* it has *Syntax Errors* and could not be compiled; or *(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 input length limitations in the segmentation model, which can easily be addressed by substituting a model with a larger input capacity at the expense of higher resource requirements.

PyLingual produces significantly 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 13.7% in CSN, 29.4% in PyPI, and 20.4% 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 85.0% in CSN, 69.3% in PyPI, and 38.1% in VirusTotal on average. While PyLingual was able to support Python 3.12 with two weeks of effort, the overall accuracy slightly dropped compared to previous versions (*e.g.*, ≈4% in PyPI). We attribute this marginal decrease to the growing adoption of optimizations that span across multiple basic blocks, affecting the control flow reconstructor. We elaborate on our efforts to support 3.12 in § 8.5.3.

We highlight the sharp drop in decompilation accuracy experienced by Pycdc in Python 3.11, down to just 3.0% from 17.5% in CSN, and down to 9.5% from 17.1% in VirusTotal. We expected this drop because Python 3.11 completely overhauled the bytecode specification for exception handling

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

Dataset	Version	PyLingual (Equal / Semantic Error / Syntax Error / No Output)	Uncompyle6 (Equal / Semantic Error / Syntax Error / No Output)	Decompyle3 (Equal / Semantic Error / Syntax Error / No Output)	Pycdc (Equal / Semantic Error / Syntax Error / No Output)
CodeSearchNet	3.6	**96.2%**/ 2.1%/ 0.9%/ 0.9%	85.7%/ 13.8%/ 0.4%/ 0.1%	- / - / - / -	21.4%/ 63.0%/ 15.6%/ 0.0%
	3.7	**95.8%**/ 2.5%/ 0.9%/ 0.8%	82.5%/ 15.7%/ 0.7%/ 1.1%	85.9%/ 12.9%/ 1.1%/ 0.2%	21.2%/ 63.2%/ 15.6%/ 0.0%
	3.8	**96.5%**/ 1.9%/ 0.8%/ 0.8%	64.1%/ 17.8%/ 11.6%/ 6.5%	75.8%/ 19.0%/ 1.1%/ 4.1%	18.6%/ 63.0%/ 18.4%/ 0.0%
	3.9	**96.8%**/ 1.6%/ 0.8%/ 0.9%	- / - / - / -	- / - / - / -	18.6%/ 69.0%/ 12.4%/ 0.0%
	3.10	**95.7%**/ 2.4%/ 1.0%/ 0.9%	- / - / - / -	- / - / - / -	17.5%/ 71.7%/ 10.8%/ 0.0%
	3.11	**95.2%**/ 2.5%/ 1.3%/ 1.0%	- / - / - / -	- / - / - / -	3.0%/ 84.5%/ 12.5%/ 0.0%
	3.12	**91.3%**/ 4.3%/ 3.5%/ 0.9%	- / - / - / -	- / - / - / -	- / - / - / -
PyPI	3.6	**75.4%**/ 8.5%/ 7.0%/ 9.2%	47.3%/ 48.0%/ 1.6%/ 3.1%	- / - / - / -	8.9%/ 39.5%/ 51.5%/ 0.1%
	3.7	**72.7%**/ 10.1%/ 8.0%/ 9.2%	40.0%/ 49.1%/ 5.6%/ 5.3%	51.0%/ 41.1%/ 6.7%/ 1.2%	8.4%/ 38.0%/ 53.5%/ 0.0%
	3.8	**74.9%**/ 8.4%/ 7.6%/ 9.1%	32.1%/ 37.1%/ 17.9%/ 12.9%	36.6%/ 48.9%/ 5.8%/ 8.7%	7.9%/ 35.6%/ 56.6%/ 0.0%
	3.9	**76.7%**/ 7.3%/ 6.9%/ 9.1%	- / - / - / -	- / - / - / -	7.8%/ 41.3%/ 50.9%/ 0.0%
	3.10	**75.7%**/ 6.5%/ 6.4%/ 11.5%	- / - / - / -	- / - / - / -	6.5%/ 39.5%/ 54.0%/ 0.0%
	3.11	**73.9%**/ 8.1%/ 8.7%/ 9.3%	- / - / - / -	- / - / - / -	4.6%/ 43.7%/ 51.6%/ 0.0%
	3.12	**69.6%**/ 7.7%/ 13.1%/ 9.5%	- / - / - / -	- / - / - / -	- / - / - / -
VirusTotal	3.6	**44.1%**/ 4.5%/ 6.2%/ 45.2%	28.6%/ 37.6%/ 19.6%/ 14.3%	- / - / - / -	15.0%/ 38.5%/ 46.0%/ 0.5%
	3.7	**50.0%**/ 4.3%/ 5.1%/ 40.6%	28.1%/ 25.1%/ 30.8%/ 16.0%	30.3%/ 26.0%/ 36.5%/ 7.2%	14.2%/ 45.5%/ 39.5%/ 0.8%
	3.8	**50.3%**/ 6.4%/ 5.5%/ 37.8%	22.1%/ 12.6%/ 14.6%/ 50.7%	24.3%/ 16.5%/ 14.2%/ 44.9%	17.1%/ 39.1%/ 43.8%/ 0.0%
	3.9	**47.6%**/ 3.9%/ 7.7%/ 40.8%	- / - / - / -	- / - / - / -	9.5%/ 49.2%/ 41.4%/ 0.0%

to support "zero-cost exceptions", which only incur performance costs if an exception is triggered. A specialized table was added to hold exception jump targets, where they had previously been presented in bytecode instruction arguments. PyLingual was able to seamlessly overcome this major specification change with minimal engineering effort by annotating instructions associated with exception table entries during code normalization, from which the NLP components were able to derive the necessary information for segmentation and translation. Currently, Pycdc has not received updates to support Python 3.12.

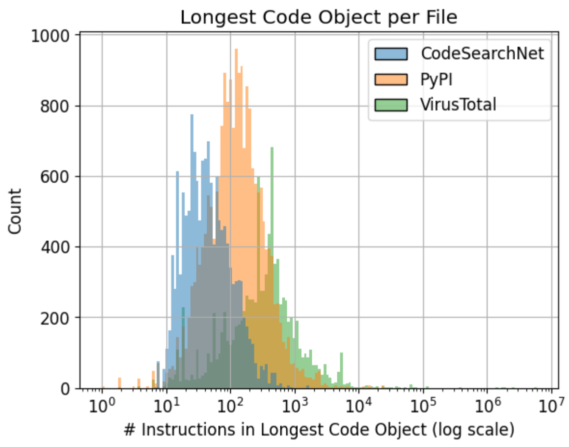


Figure 5: Distribution of largest code object size in each PYC file.

We notice that the share of binaries that result in *no output* from PyLingual grows from ≈1% in CSN, to ≈9% in PyPI, up to ≈40% in VirusTotal. The predominant cause of these failures is the size limitation of the segmentation model, which can only accept up to 512 tokens, which corresponds to roughly 400~430 instructions. In Figure 5, we show the distribution of instruction counts of the largest code object in each file in the test set. We observe that large classes are common in production code from PyPI, and that malware authors frequently place their entire payload logic in one large code object. Common source-level obfuscation patterns, which we will discuss more in §8.5, create large sequences of simple bytecode that challenge the segmentation model's input size limitation. Despite this, PyLingual still produces more correct decompilations on the VirusTotal dataset than any other existing Python decompiler.

8.3 Top-k Segmentation Improvements

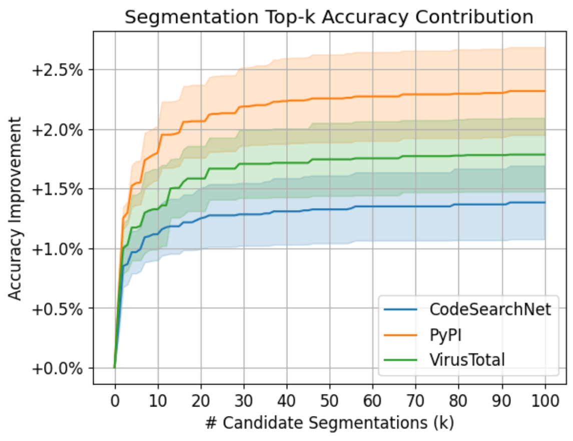


Figure 6: Decompilation accuracy provided by segmentation search, aggregated across versions 3.6-3.12. Standard error around the mean is highlighted.

In Figure 6, we show the impact of configuring the segmentation search limit k. Across all versions and datasets, we see an initial jump in decompilation accuracy, followed by a period of diminishing returns. Typically, the inclusion of the segmentation search will improve the overall results by 1-2% when $k \geq 10$, with most of the improvement occur-

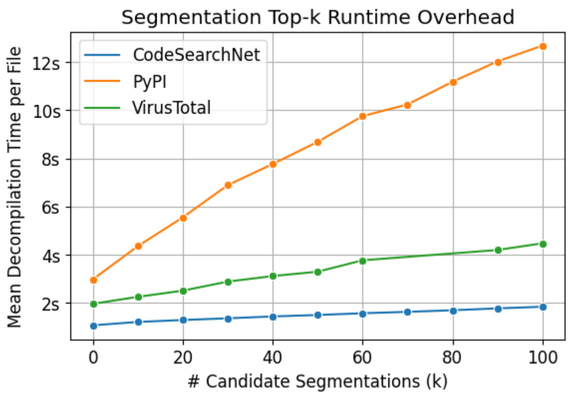


Figure 7: Extra computation time incurred by top-k segmentation search, measured against Python 3.9.

ing within the first 10 candidates. Figure 7 shows the runtime overhead tradeoff, where increasing k results in approximately linear increases in runtime overhead. Notice that exploring each segmentation candidate only takes a small fraction of the time needed to produce the initial decompilation result; only the bytecode statements that were modified by the search will need to be retranslated. Further, because decompilation completes as soon as a correct result is confirmed, the segmentation search limit k parameterizes how long the user will need to wait for a negative result; setting a k that is higher than needed to successfully decompile a given PYC binary incurs no additional time cost on that binary. From Figure 7, we see that increasing k incurs higher costs in PyPI than in CSN or VirusTotal because the top-k search exits early in the event of an *equal* or *no output* result. PyPI has the largest share of files with syntactic and semantic errors, so it has the most files who will bear an increased decompilation cost due to increasing k.

8.4 Corrector Model Improvements

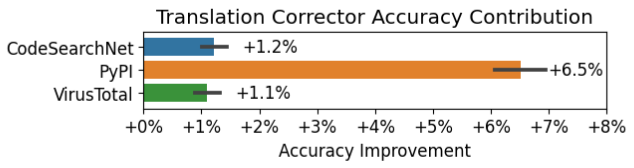


Figure 8: Decompilation accuracy provided by the translation corrector models, aggregated across versions 3.6-3.12.

Figure 8 shows the benefit of adding a corrector model to the statement translation module. We see that the benefit is much larger in PyPI, contributing ≈6.5% to the success rate, as compared to ≈1.1% in the other datasets. The production code seen in the PyPI dataset often involves long function definitions, lists, and complex expressions, which benefit significantly from a second translation pass. Comparatively, the code we see from the CSN dataset is relatively straightforward to comprehend. In VirusTotal, files containing complex statements also tend to be large enough to overwhelm the segmentation module, limiting the effectiveness of the corrector model. Adding the corrector model to the decompiler pipeline adds a one-time cost of ≈10 hours for translating the training set and training the corrector model, and increases the runtime overhead by approximately 20%. Even with this additional overhead, PyLingual can typically process each binary in less than 10 seconds, so it is affordable in practice.

8.5 Case Studies

Here, we compare the strengths and weaknesses of PyLingual to existing Python decompilers by contrasting their results on illustrative bytecode examples.

8.5.1 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. An example of this can be found in our VirusTotal Dataset where we encountered 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` clearly demonstrated a failure to correctly segment the boolean expression, splitting off the last `e[2]>e[3]` and incorrectly grouping the terms of the expression for translation.

```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 complex condition (§12.2).

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 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 segmentations design 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
    return r_cost(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)

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

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

Figure 10: CSN 3.7 sample with complex conditional expression.

8.5.2 Very Long but Simple Code Objects

Due to its inclusion of NLP models, PyLingual encounters specific constraints, including a limit on the number of tokens each code object can handle. In Figure 11, we illustrate a VirusTotal 3.9 sample that continuously appends portions of an encoded string to a variable. While the bytecode is not complex, it is over 1,700 instructions long, which overwhelms the segmentation model's input capacity. Across our datasets, we have found individual code objects with over 1,000,000 instructions. Traditional decompilers are better able to iterate over long bytecode sequences when there is not much control flow complexity. We further discuss the current limitations of PyLingual and future research directions to mitigate those leveraging new advances in NLP [39] in §10.

```
  84 LOAD_NAME                 (x)
  86 LOAD_CONST                ('dXNlciA9IG')
  88 INPLACE_ADD
  90 STORE_NAME                (x)
     ...
3418 LOAD_NAME                 (x)
3420 EXTENDED_ARG              (256)
3422 LOAD_CONST                ('o=')
3424 INPLACE_ADD
3426 STORE_NAME                (x)
```

(a) Simplified Bytecode

```python
x += "dXNlciA9IG"
...
x += "o="
```

(b) Corresponding Source

Figure 11: Virustotal Python 3.9 PYC sample. This sample is not complex, but has over 1,700 instructions (§12.2).

8.5.3 Supporting New Python 3.12

Supporting new Python versions in PyLingual involves three key tasks: *(1)* integrating a version-agnostic disassembler, *(2)* consistently masking variables and constants, and *(3)* updating the control flow reconstruction module. For Python 3.12, we focused our efforts on the control flow module, as the open-source disassembler was able to be extended, and the masking process needed no updates. The entire effort was accomplished by a single junior engineer with only ≈12 hours of work and ≈200 lines of code changes.

Before Python 3.12, instruction offsets correponded to source line positions, simplifying source reconstruction. However, Python 3.12 now places exception-handling code at the end of the code object, necessitating the proper relocation of `except` blocks in the reconstructed source. Additionally, we manually updated `finally` statement insertion, as the translation model does not generate them, as discussed in §4.4. The update extends the Python 3.11 major update, which introduced a specialized table for handling exceptions more efficiently [44], encompassing various constructs like `except`, `finally`, and `with`. While Python 3.12 retains the exception table, the heuristics used to determine the type of each table entry in 3.11 no longer work, requiring new heuristics to be created.

We also made several minor adjustments. In version 3.12, Python introduced the `RETURN_CONST` instruction, requiring us to update the control flow reconstructor to check for this new instruction instead of just `RETURN_VALUE`. Additionally, we encountered and addressed a problem where `if` statements at the end of `for` loops became inverted, causing the control flow reconstructor to mislabel the `if` body as an `else`.

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 [45]–[48]. 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. [49] 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 accurately restore type information and data dependencies [50], and *(2) structural analysis* to accurately identify code blocks and restore control dependencies among them [49], [51]–[53]. Structural analysis has more impact on the performance and usability of a decompiler, so it has been the primary focus of recent research [51]–[53].

Neural decompilation. Recent advances in neural translation have sparked interest in their use for binary analysis and decompilation. Although large public code datasets meet the data demands of NLP approaches, decompilation demands strict syntax compliance and semantic accuracy, posing new challenges to natural language translation and complicating the generation of trustworthy results.

Shin et al. [54] proposed one of the first ML-assisted binary analyses. They built a multi-layer RNN network that consumes one byte at a time to predict a byte sequence that maps to the function boundary. Katz et al. [55] 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 [56] 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 [57] 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 [1], [58], [59]. 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. `uncompyle6` [17] and `decompyle3` [18] are the two most popular and well-supported decompilers for Python binaries. `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 [60] 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` [26] 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 much more 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 §8, we saw that `pycdc` was unable to decompile even 25 % of `PYC` binaries for any of the Python versions we tested. Although `pycdc` technically supports Python 3.11, it was unable to adapt to the exception handling overhaul, and its already poor accuracy suffered immensely as a result.

Decompilers for other HDLs. `Soot` [61], 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 bytecode specification changes. Furthermore, the framework supports code reassembly to instrument additional functionalities. Several stable decompilers for the `.Net` framework [62], [63] are also actively maintained. Although niche and thus not actively maintained, decompilers also exist for other HDL families such as Ruby and Lua [64], [65]. 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 [30], NLP researchers have been exploring transformer architectures that leverage sparse attention to handle longer sequences [66], [67]. 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, 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, supersampling, and artificial data generation as mechanisms to reduce the reliance on user adoption of new features to train effective models.

Adversarial considerations. Adversaries could exploit PyLingual's limitations to disrupt decompilation by crafting malware payloads with exceptionally long statements or employing rare language features. While we can enhance PyLingual's performance by leveraging ongoing advancements in NLP modelling, including improving the corrector model to handle challenging statements, the system's efficacy fundamentally relies on the external components of NLP models, which largely remain black boxes. This dependency complicates the tasks of tracing and understanding translation errors. However, through accountable and detailed error reporting, PyLingual can pinpoint errors, reducing the effort security analysts must expend in reversing `PYC` payloads.

LLMs for Decompilation. Recently, Large Language Models (LLMs) have revolutionized various sectors. However, even LLMs will eventually start to produce inaccurate predictions when the input is long. To counter these errors, PyLingual introduces a unique architecture that establishes clear interaction boundaries, allowing us to identify and improve underperforming components. Future work may explore the effectiveness of an all-encompassing language model for decompilation, utilizing extensive accumulated datasets. Once established, it will be crucial to study how swiftly this model can handle complex changes in language specification. We caution, however, that using an end-to-end black box will pose substantial challenges in debugging and development.

Automation of control flow reconstruction. To scale across language versions, PyLingual relies on three components that require manual maintenance: (1) a version-agnostic Python disassembler [16], (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, owing 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, often extending beyond basic block boundaries, has heightened the challenges of creating a universal control flow reconstruction module. Expecting more aggressive optimizations in future versions, we will develop a GNN (Graph Neural Network)-based strategy to automatically adapt to control flow dependencies across various 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 signifcantly from PyLingual's analysis. These include: *(1)* A modular code object structure, *(2)* A rich source of datasets similar to Python's PyPI, *(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.

11 Conclusion

PyLingual's innovative design balances theoretical PL 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 77% on average across Python 3.6 - 3.12, marking an average improvement of 47% over SOTA Python decompilers [17], [18], [26]. 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 freely available online decompilation service [3].

[3] pylingual.io

References

[1] C. Security, *Python malware on the rise*, https://www.cyborgsecurity.com/cyborg_labs/python-malware-on-the-rise/, Jul. 2020.

[2] C. Xiao, C. Zheng, and X. 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/.

[3] U. 42, *Pymicropsia: New information-stealing trojan from aridviper*, https://unit42.paloaltonetworks.com/pymicropsia/.

[4] R. Falcone and S. Conant, *New shameless commodity cryptocurrency stealer (westeal) and commodity rat (wecontrol)*, https://unit42.paloaltonetworks.com/westeal/.

[5] V. Koutsokostas and C. Patsakis, *Python and malware: Developing stealth and evasive malware without obfuscation*, https://arxiv.org/pdf/2105.00565.pdf.

[6] D. Inc, "TRISIS Malware," Tech. Rep., Dec. 2017. [Online]. Available: https://dragos.com/wp-content/uploads/TRISIS-01.pdf.

[7] K. Ogino, *Unboxing snake - python infostealer lurking through messaging services*, https://www.cybereason.com/blog/unboxing-snake-python-infostealer-lurking-through-messaging-service.

[8] J. Grunzweig, *Unit 42 technical analysis: Seaduke*, https://unit42.paloaltonetworks.com/unit-42-technical-analysis-seaduke/.

[9] jinye, *Necro frequent upgrades, new version begins using pyinstaller and dga*, https://blog.netlab.360.com/not-really-new-pyhton-ddos-bot-n3cr0m0rph-necromorph/.

[10] I. Kenefick, M. Yambao, A. Nieto, and K. Ang, *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.

[11] W. Mercer, *Poetrat: Python rat uses covid-19 lures to target azerbaijan public and private sectors*, https://blog.talosintelligence.com/poetrat-covid-19-lures/.

[12] J. Grunzweig, *Python-based pwobot targets european organizations*, https://unit42.paloaltonetworks.com/unit42-python-based-pwobot-targets-european-organizations/.

[13] *Pyinstaller quickstart — pyinstaller bundles python applications*, https://www.pyinstaller.org/.

[14] *Py2exe*, https://www.py2exe.org/.

[15] *Pyinstaller extractor*, https://github.com/extremecoders-re/pyinstxtractor.

[16] R. Bernstein, *Python-xdis*, https://github.com/rocky/python-xdis.

[17] *Uncompyle6 · pypi*, https://pypi.org/project/uncompyle6/.

[18] *Decompyle3 · pypi*, https://pypi.org/project/decompyle3/.

[19] S. Montanaro, *A peephole optimizer for python*, 1998.

[20] *Faster-cpython*, https://github.com/faster-cpython/ideas.

[21] M. Shannon, *Pep 659 – specializing adaptive interpreter | peps.python.org*, https://peps.python.org/pep-0659/.

[22] C. Meyer, *Pep 709 – inlined comprehensions | peps.python.org*, https://peps.python.org/pep-0709/.

[23] H. Husain, H.-H. Wu, T. Gazit, M. Allamanis, and M. Brockschmidt, "CodeSearchNet Challenge: Evaluating the State of Semantic Code Search," *arXiv*, vol. cs.LG, Sep. 2019. eprint: 1909.09436. [Online]. Available: arXiv.org.

[24] *Pypi - the python package index*, https://pypi.org/.

[25] virustotal, *Metasploit*, https://www.virustotal.com/gui/home/upload, 2021.

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

[27] K. Burk, F. Pagani, C. Kruegel, and G. Vigna, "Decomperson: How humans decompile and what we can learn from it," in *31st USENIX Security Symposium (USENIX Security 22)*, Boston, MA: USENIX Association, Aug. 2022, pp. 2765–2782, ISBN: 978-1-939133-31-1. [Online]. Available: https://www.usenix.org/conference/usenixsecurity22/presentation/burk.

[28] Z. Liu and S. 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*, ser. ISSTA 2020, Virtual Event, USA: Association for Computing Machinery, 2020, pp. 475–487, ISBN: 9781450380089. DOI: 10.1145/3395363.3397370. [Online]. Available: https://doi.org/10.1145/3395363.3397370.

[29] N. Harrand, C. Soto-Valero, M. Monperrus, and B. Baudry, "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.

[30] A. Ahad, C. Jung, A. Askar, D. Kim, T. Kim, and Y. Kwon, "PYFET: Forensically Equivalent Transformation for Python Binary Decompilation," ser. IEEE Symposium on Security and Privacy (SP), May 2023.

[31] *Python documentation by version*, https://www.python.org/doc/versions/.

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

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

[34] W. Zhao, L. Wang, K. Shen, R. Jia, and J. Liu, *Improving grammatical error correction via pre-training a copy-augmented architecture with unlabeled data*, 2019. arXiv: 1903.00138 [cs.CL].

[35] S. Yang, Y. Wang, and X. Chu, "A survey of deep learning techniques for neural machine translation," *CoRR*, vol. abs/2002.07526, 2020. arXiv: 2002.07526. [Online]. Available: https://arxiv.org/abs/2002.07526.

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

[37] J. Ferrante, K. Ottenstein, and J. Warren, "The program dependence graph and its use in optimization," eng, *ACM transactions on programming languages and systems*, vol. 9, no. 3, pp. 319–349, 1987, ISSN: 0164-0925.

[38] D. Tsichritzis, "The equivalence problem of simple programs," *J. ACM*, vol. 17, no. 4, pp. 729–738, Oct. 1970, ISSN: 0004-5411. DOI: 10.1145/321607.321621. [Online]. Available: https://doi.org/10.1145/321607.321621.

[39] A. Bulatov, Y. Kuratov, and M. S. Burtsev, *Scaling transformer to 1m tokens and beyond with rmt*, 2023. arXiv: 2304.11062 [cs.CL].

[40] B. Peng, E. Alcaide, Q. Anthony, *et al.*, *Rwkv: Reinventing rnns for the transformer era*, 2023. arXiv: 2305.13048 [cs.CL].

[41] C.-C. Lin, A. Jaech, X. Li, M. R. Gormley, and J. Eisner, *Limitations of autoregressive models and their alternatives*, 2021. arXiv: 2010.11939 [cs.LG].

[42] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of deep bidirectional transformers for language understanding," in *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, Minneapolis, Minnesota: Association for Computational Linguistics, Jun. 2019, pp. 4171–4186. DOI: 10.18653/v1/N19-1423. [Online]. Available: https://aclanthology.org/N19-1423.

[43] C. Raffel, N. Shazeer, A. Roberts, *et al.*, "Exploring the limits of transfer learning with a unified text-to-text transformer," *CoRR*, vol. abs/1910.10683, 2019. arXiv: 1910.10683. [Online]. Available: http://arxiv.org/abs/1910.10683.

[44] *Pep 654 – exception groups and except* | peps.python.org*, https://peps.python.org/pep-0654/.

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

[46] C. D. B. Knighton, *Ghidra - Journey from Classified NSA Tool to Open Source*, Aug. 2019.

[47] R. Team, *Radare2 github repository*, https://github.com/radare/radare2, 2017.

[48] *Retdec :: Home*, https://retdec.com/.

[49] C. Cifuentes, "Reverse Compilation Techniques," Ph.D. dissertation, 1994.

[50] M. V. Emmerik, "Static Single Assignment for Decompilation," Ph.D. dissertation.

[51] D. Brumley, J. Lee, E. J. Schwartz, and M. Woo, "Native x86 Decompilation Using Semantics- Preserving Structural Analysis and Iterative Control-Flow Structuring," in *USENIX Security Symposium (USENIX Security)*, Washington, D.C., Jul. 2013.

[52] K. Yakdan, S. Eschweiler, E. Gerhards-Padilla, and M. Smith, "No More Gotos: Decompilation Using Pattern-Independent Control-Flow Structuring and Semantics-Preserving Transformations," in *Network Distributed Security Symposium (NDSS)*, Feb. 2015.

[53] K. Yakdan, S. Dechand, E. Gerhards-Padilla, and M. Smith, "Helping Johnny to Analyze Malware: A Usability-Optimized Decompiler and Malware Analysis User Study," pp. 158–177, May 2016.

[54] E. C. R. Shin, D. Song, and R. Moazzezi, "Recognizing functions in binaries with neural networks," in *24th USENIX Security Symposium (USENIX Security 15)*, Washington, D.C.: USENIX Association, Aug. 2015, pp. 611–626, ISBN: 978-1-939133-11-3. [Online]. Available: https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/shin.

[55] O. Katz, Y. Olshaker, Y. Goldberg, and E. Yahav, *Towards neural decompilation*, 2019. arXiv: 1905.08325 [cs.PL].

[56] C. Fu, H. Chen, H. Liu, *et al.*, "Coda: An end-to-end neural program decompiler," in *Advances in Neural Information Processing Systems*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, Eds., vol. 32, Curran Associates, Inc., 2019. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2019/file/093b60fd0557804c8ba0cbf1453da22f-Paper.pdf.

[57] R. Liang, Y. Cao, P. Hu, and K. Chen, "Neutron: an attention-based neural decompiler," *Cybersecurity*, May 2021. DOI: 10.1186/s42400-021-00070-0. [Online]. Available: https://cybersecurity.springeropen.com/articles/10.1186/s42400-021-00070-0.

[58] "Old Dogs New Tricks: Attackers adopt exotic programming languages," BlackBerry Research & Intelligence Team, Tech. Rep., 2021. [Online]. Available: https://blogs.blackberry.com/en/2021/07/old-dogs-new-tricks-attackers-adopt-exotic-programming-languages.

[59] *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/.

[60] R. Bernstein, *Python-xdis*, https://github.com/rocky/python-spark.

[61] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan, "Soot - a java bytecode optimization framework," in *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research*, ser. CASCON '99, Mississauga, Ontario, Canada: IBM Press, 1999, p. 13.

[62] *Icsharpcode/ilspy: .net decompiler with support for pdb generation, readytorun, metadata (&more) - cross-platform!* https://github.com/icsharpcode/ILSpy.

[63] *Dotpeek: Free .net decompiler & assembly browser by jetbrains*, https://www.jetbrains.com/decompiler/.

[64] *Cout/ruby-decompiler: A decompiler for ruby code (both mri and yarv)*, https://github.com/cout/ruby-decompiler.

[65] *Tool: Lua 5.1 decompiler*, https://lua-decompiler.ferib.dev/.

[66] M. Zaheer, G. Guruganesh, A. Dubey, *et al.*, "Big Bird: Transformers for Longer Sequences," *arXiv*, Jan. 2021. arXiv: 2007.14062. [Online]. Available: https://arxiv.org/abs/2007.14062.

[67] I. Beltagy, M. E. Peters, and A. Cohan, "Longformer: The long-document transformer," *CoRR*, vol. abs/2004.05150, 2020. arXiv: 2004.05150. [Online]. Available: https://arxiv.org/abs/2004.05150.

12 Appendix

12.1 Semantic Error Localization

When PyLingual decompiles a PYC sample, and our strict equivalency metric §4.5 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, the difference in the bytecode is 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 code equivalency verification was able to identify and locate the exact instructions affected by the semantic decompilation error.

```
***<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.

```
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                (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               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
```

(d) Original PYC disassembly.

```
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                (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
```

(e) Disassembly of decompiled source, then recompiled.

(f) Virustotal 3.9 PYC disassembly.

Figure 12: Local error analysis on a Virustotal 3.9 sample (§12.2).

12.2 Virustotal 3.9 Sample Hashes

Figure 9 Sample Sha256 hash:
`c834a279f9132e1d5dc156938f07c2de3238a714a4d153eabc500fd069dd95d4`

Figure 11 Sample Sha256 hash:
`05d0b4ddac8d01ac25beaf2c6a1cf1efd47c48acbfb4a4db0d82dd8c261758f6`

Figure 12 Sample Sha256 hash:
`066281026bb847c68c748782308f8dab62dd31818a5443f0db5d51793f2af4b0`