Walking The Last Mile: Studying Decompiler Output Correction in Practice

Josh Wiedemeier
jdw170000@utdallas.edu
The University of Texas at Dallas
Richardson, TX, United States

Simon Klancher
srk200018@utdallas.edu
The University of Texas at Dallas
Richardson, TX, United States

Joel Flores
jjf190004@utdallas.edu
The University of Texas at Dallas
Richardson, TX, United States

Max Zheng
mxz200025@utdallas.edu
The University of Texas at Dallas
Richardson, TX, United States

Jaehyun Park
jxp220079@utdallas.edu
The University of Texas at Dallas
Richardson, TX, United States

Sang Kil Cha
sangkilc@kaist.ac.kr
Korea Advanced Institute of Science and Technology
Daejon, Republic of Korea

Kangkook Jee
kxj190011@utdallas.edu
The University of Texas at Dallas
Richardson, TX, United States

Abstract

The increasing prevalence of Python has spurred interest in decompiling Python PYC bytecode. This work presents the first large-scale study on human-assisted Python decompilation *in the wild*, leveraging extensive data from pylingual.io, spanning 181,646 PYC binaries, 9,003 user-submitted patches, and 393 accuracy-verified patches. We investigate how reverse engineers respond to inaccurate decompilation and identify factors influencing their efforts to achieve accurate decompilation. We complement this unprecedented observational data with a controlled user study that isolates the technical difficulty of patching imperfect Python decompilations.

By contrasting real-world patching behavior with that of the controlled setting, we discover that reversers' decision to repair a decompilation result is more strongly driven by the semantic content of the program (*e.g.*, malware binaries or malicious tools) than by the technical difficulty of the patch. That is, a reverser's motivation is more important than their expertise.

Our study reveals common patterns observed in the patching process, including how users approached the patching task, the types of errors they encountered, and the strategies they employed to resolve them. We also examine the strengths and limitations of assistive tools in the pursuit of perfect decompilation. Our findings offer unique insights into the practical dynamics of human-decompiler interaction, providing actionable recommendations for integrating human intelligence into the decompilation workflow and demonstrating the research potential of reliable decompilation accuracy verification.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
CCS '25, Taipei, Taiwan.
© 2025 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-1525-9/2025/10
https://doi.org/10.1145/3719027.3765040

CCS Concepts

• **Security and privacy → Software reverse engineering**; • **Social and professional topics** → Malware / spyware crime; • **Human-centered computing → User studies**.

Keywords

Reverse Engineering; User Study; Malware; Decompilation

ACM Reference Format:
Josh Wiedemeier, Simon Klancher, Joel Flores, Max Zheng, Jaehyun Park, Sang Kil Cha, and Kangkook Jee. 2025. Walking The Last Mile: Studying Decompiler Output Correction in Practice. In *Proceedings of the 2025 ACM SIGSAC Conference on Computer and Communications Security (CCS '25), October 13–17, 2025, Taipei, Taiwan.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3719027.3765040

1 Introduction

Many malware payloads and benign software applications are now written and compiled directly from source code developed in High-level Dynamic Languages (HDLs). This practice is becoming increasingly common as it saves significant development time by leveraging well-established language ecosystems. Among various HDLs, Python is the most popular among both software developers and malware authors for its versatility and ease of use [1–13]. Consequently, there has been growing interest in research focused on reversing and decompiling software binaries created from Python source code. This momentum is reflected in several research proposals [14, 15] and the emergence of open-source projects [16–20] implementing decompilers for PYC binaries.

Of these recent endeavors, PyLingual [15] stands out as the most effective Python decompiler for modern Python versions (3.6 and onward). PyLingual has been able to keep up with Python's rapid development cycle by incorporating Natural Language Processing (NLP) models into its decompilation pipeline. However, these models are prone to producing unpredictable decompilation failure modes. PyLingual employs *perfect decompilation*, a strict

post-hoc decompilation accuracy verification [21, 22]. When the decompilation result is not immediately accurate, PyLingual exposes a web-based IDE patching interface, enabling reversers to apply human intelligence and expertise to localize and repair decompilation inaccuracies. Between its initial launch as a public service in November 2023 and this paper's writing in April 2025, PyLingual has become the *de facto* framework for modern PYC decompilation, processing 181,646 PYC binaries, among which 1,977 (≈ 1%) are identified as malicious by VirusTotal. It also gathered 9,003 user-submitted patches, including 393 verifiably accurate patches. Between the time of writing and publication, PyLingual has processed 142,399 additional PYCs, 16,727 additional user patches, and 565 additional verified patches, which are not included in the study.

The goal of this study is to understand both *why* reverse engineers choose to patch incomplete decompilation outputs and *how* they approach patching incomplete decompilation outputs to fully recover the program logic. By doing so, we can derive valuable insights to improve PYC reversing and benefit a broader class of decompilation studies. Paying close attention to ethical guidelines and maintaining contact with UTD's legal department and Institutional Review Board (IRB)[1], our study observes, but does not release, PYC binaries and user-submitted patches uploaded to pylingual.io. Leveraging this extensive dataset of PYC binaries and source-level patches that users submitted to the PyLingual web service, we conduct two analyses to study reverse engineers' responses to incomplete decompilation results. First, we perform an uncontrolled observational study of PYC samples and patches submitted to the PyLingual web service by anonymous Internet users. Second, building on the observational study, we design and conduct a controlled user study to observe how cybersecurity undergraduates approach patching 40 inaccurately decompiled PYC malware samples, followed by an exit survey to ask participants' opinions on the PYC patching process and its effectiveness in understanding malware semantics.

With these, we specifically aim to address the following research questions: *(1) are users inclined to fix incomplete decompilation outputs? If so, does the type of program affect their motivation?* To explore this, we analyze the types of programs that users are more likely to attempt to fix when faced with incomplete decompilation results. We categorize the PYC binaries based on their functionality, complexity, and usage context. Our analysis reveals that users are most motivated to address decompilation inaccuracies in unobfuscated malware and its related utilities. *(2) how difficult is it to repair incomplete decompilations?* In particular, we investigate whether the difficulty of patching incomplete decompilation results presents a significant obstacle. For this, our study compares two sets of user interactions: those from in-class participants and those from users in the wild. Our findings reveal a clear discrepancy in fixing even the simplest challenges with minimal bytecode differences. Most participants in the classroom study could easily resolve these challenge with minimal effort, whereas online users rarely even attempted to repair them. This discrepancy underscores that the difficulty of patching is not a major barrier preventing users from attempting to fix erroneous decompilations. *(3) how can the design of automatic decompilers be improved using perfect decompilation?* For this, we meticulously followed and analyzed the patch history of participants participated in controlled study, identifying patterns and practices commonly observed among successful patchers. Based on these observations, we provide several actionable recommendations for effective PYC reversing and patching tasks.

Through our user study, we investigate how human-assisted Python decompilation improves accuracy in practice. Furthermore, we examine efficient and effective methods for integrating human intelligence into the overall process. By analyzing a novel dataset collected from a production-grade decompilation framework, our research aims to establish a robust knowledge base for human intelligence, strengthening the feedback loop and further enhancing the accuracy and performance of automatic decompilation systems and their user interfaces. In summary, our study delivers the following contributions:

- **Human-in-the-loop decompilation in the wild:** Our study is the first to observe and analyze how reversers engage with perfect decompilation-driven patching interfaces in an uncontrolled setting, providing an unprecedented look into the motivations of real reverser engineers.
- **Patching difficulty and effort estimation:** The study examines how reverse engineers repair incomplete decompilations, showing that creating accurate patches to inaccurate PYC decompilation results is a practical goal with a moderate amount of effort and knowledge.
- **Comprehensive study on real-world dataset:** Through both observational and controlled studies of PYC decompilation and user-submitted patches, the study provides valuable insights in improving automatic decompilers and their interactions with users.

To benefit the research community, we publish the PYC challenges and exit survey from our in-class controlled study. [2]

2 Python Decompilation

Python bytecode is organized as a tree of "code objects", each of which correspond to one function or class. Several language constructs 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 tracebacks.

Control flow in Python can be categorized into four main types: *(1)* jumps, *(2)* function calls, *(3)* return statements, and *(4)* exceptions. `Jump` targets in Python are statically determined and cannot cross the boundaries of code objects. In contrast, function call targets are resolved at runtime, allowing for the dynamic reassignment of function symbols. Modeling Python's exception handling structures requires version-specific logic due to substantial changes in recent years. In summary, control flow that is dynamic in the bytecode is *also dynamic in the source code.*

[1] UTD-IRB-25-73

[2] https://github.com/syssec-utd/CCS25-WalkingTheLastMile-Supplementary

While PYC binaries carry more information within cleanly identified code object boundaries, the most significant challenge in Python decompilation lies in the instability of the Python bytecode specification. Since its initial 1991 release, it 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 [23], including the addition, deletion, and modification of instruction opcodes. Beyond changes to the opcode definitions, each version introduces insufficiently documented changes to code generation, which further increases the maintenance effort for Python decompilers.

2.1 PyLingual Overview

To address these challenges, PyLingual integrates NLP-models with programming language components to decompile PYC binaries, supporting evolving Python versions with minimal human maintenance. The PyLingual framework consists of three components: *(1)* bytecode segmentation, *(2)* statement translation, and *(3)* control flow reconstruction. Lastly, *perfect decompilation* verifies whether the decompiled code is semantically equivalent to the original bytecode.

Bytecode segmentation. The bytecode segmenter uses a BERT model [24] to divide bytecode into independently digestible segments called "bytecode statements". These statements correspond to Python source code statements, facilitating accurate translation in later stages. Correct segmentation is critical for accurate decompilation because it determines the association between bytecode instructions and source code statements.

Statement translation. The statement translation module is responsible for converting bytecode segments into Python source code statements. Each statement is translated independently by a CodeT5 transformer [25, 26], and depends on accurate segmentation. Because statements can grow arbitrarily long and complex, the translation model's fixed 512-token context window can be overwhelmed, which results in statement-level decompilation failures. The canonical example is long dictionary definitions being truncated, despite the simple code contained in the statement.

Control flow reconstruction. After translating the individual statements, the control flow reconstruction module combines them into a coherent Python program that attempts to reflect the original control flow in the bytecode. The control flow reconstruction process relies heavily on heuristics, which may fail in complex control flow cases. Notably, the control flow reconstructor often struggles when overlapping control structures coincide with control statements that break out of multiple layers of control structures at once.

Equivalence verification with perfect decompilation. The final stage, code equivalence verification, confirms the accuracy of the decompilation process. This step uses a differential testing approach [27] by recompiling the decompiled source code and comparing it with the original bytecode to verify exact equivalence.

By confirming that both the bytecode instructions and important metadata, like exception tables, match perfectly, a successful perfect decompilation test guarantees the correctness of the decompiled code. This verification process is strict, ensuring no false positives — imperfect but semantically equivalent code might fail this check, but it prevents any incorrect decompilation from being falsely validated.

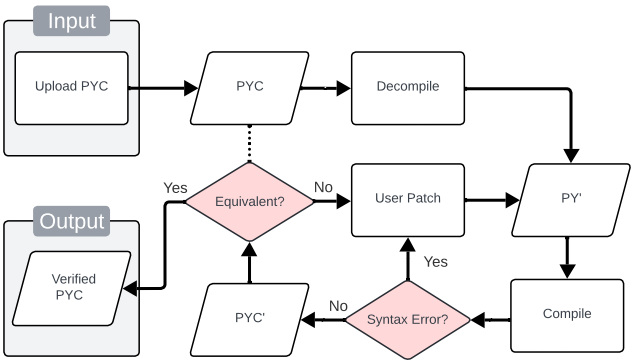


Figure 1: PyLingual patching process. PyLingual first decompiles user-uploaded PYC to be verified by perfect decompilation. In cases for syntactic and semantic errors, users can work on Web IDE to fix errors and submit patches.

Because the perfect decompilation test checks instruction-level equivalence, it indicates where in the bytecode differences were detected, similar to a simple instruction-diff. This information gives reverse engineers a specific source line number and bytecode instruction offset to focus additional reversing efforts.

Algorithm 1 Perfect Decompilation Workflow

Input: Python bytecode file PYC
Output: Perfectly decompiled Python source file PY'

function DecompileAndPatch(PYC)
    $PY' \leftarrow$ Decompile(PYC) ▷ Potentially erroneous
    loop ▷ Verification Loop.
        if PY' contains syntax errors **then**
            Notify user of syntax errors
            User modifies PY' to fix syntax errors
            Continue loop
        end if
        $PYC' \leftarrow$ Compile(PY')
        if PYC' is not strictly equivalent to PYC **then**
            Notify user of semantic errors
            User modifies PY' to fix semantic errors
            Continue loop
        end if
        Return PY' ▷ verified perfect decompilation
    end loop
end function

2.2 Correcting Failed Decompilations

Perfect decompilation verification forms the foundation of a user-oriented feedback loop in which reversers can iteratively "patch" a

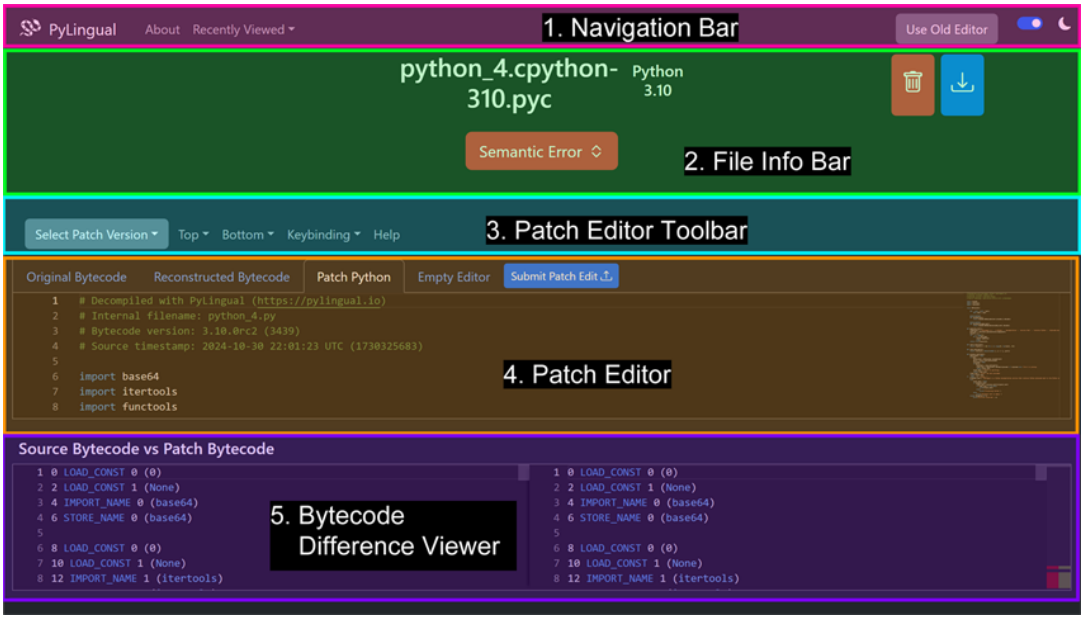


Figure 2: PyLingual Web IDE for patching erroneous decompilations.

decompilation result to ultimately achieve perfect decompilation. PyLingual's online web service incorporates perfect decompilation to validate its outputs and enable user-oriented patching. The web interface enables users to upload PYC files and receive the decompiled source code, accompanied by an "equivalence report" that summarizes the decompilation status of each code object along with the error types and their locations. When the decompilation is imperfect, the patching interface depicted in Figure 2 is exposed, featuring a web-based IDE [28], allowing users to correct incomplete decompilation outputs by submitting patches. The patching interface consists of three major components: a file information bar, which includes the Python version and an expandable equivalence report; the patch editor, which can be configured to show different views of the decompilation result; and a bytecode diff view, which helps users locate semantic errors by highlighting the differences between the original bytecode and recompiled patch bytecode.

The patching process generally follows illustrated in Figure 1, detailed in Algorithm 1. Using bytecode difference viewer to locate and repair decompilation errors, a human reverser can iteratively close the gap between the original PYC and the decompiled PYC' until the accuracy of the decompilation is confirmed. Verification of user patches only involves CPython compilation and static code equivalency verification, which has acceptable computational overhead. Importantly, the dominance of CPython in the PYC compiler ecosystem combined with its limited configurability makes compiler provenance [29] trivial, enabling reliable perfect decompilation tests even on user-uploaded PYC files.

While the current version of the user interface requires familiarity with Python bytecode to use effectively, perfect decompilation provides the foundation for supporting novice reverse engineers. By improving and refining the information presented, the time, effort, and expertise required to identify and repair decompilation failures can be minimized. Further, repaired files provide a rich source of information regarding the weaknesses of the decompiler, guiding and assisting in future developments.

> Files and patches submitted to the PyLingual web service are retained to support future research and development.
>
> By using this service, you warrant that you are not violating export control laws, intellectual property rights, licenses, or other legal or contractual obligations, and that you are not using this service for improper, unauthorized, or malicious purposes. Proprietary files uploaded to PyLingual may be disclosed to relevant third parties.

Figure 3: PyLingual's end-user privacy agreement. The language was decided through a collaboration between our team (the PyLingual web service maintainers), UTD's IRB, and UTD's legal team.

3 Ethical Considerations

As we utilize PYC binaries and user-patches uploaded to PyLingual web service for user and data analysis studies, it is paramount to adhere to legal and ethical guidelines to ensure the prevention of both intentional and unintentional harm to participants. Below, we outline the legal and ethical risks associated with PyLingual and the measures implemented to mitigate them.

Potential privacy risks of PYC binaries and patches. From a user privacy perspective, as we are using user-uploaded files and patches for our research, we can consider three kinds of risk

sources: *(1)* PYC binaries, *(2)* user patches, and *(3)* network logs. Users who want to use PyLingual web service primarily share the PYC binary to reverse the original Python source code. Additionally, when PyLingual produces incomplete/erroneous source, users may attempt to fix those errors by editing the incomplete source code and upload their patches to be verified by PyLingual's accuracy verification.

For the PYC binaries, it is generally assumed that the original author of the program is not the same as the user who uploaded the PYC file for decompilation. As such, during discussions with the university's IRB office, we determined that PYC uploads did not present any significant privacy risks. Similarly, no concrete privacy concerns were associated with the user-submitted patches. While IP addresses were considered, they were not identified as a privacy risk since they typically cannot be used to directly identify a specific user. Discussions primarily focused on cases where IP addresses and web service log entries could potentially be linked to PYC binaries and user patches, especially when augmented with external intelligence tools (*e.g.,* `IP WHOIS`, geolocation, or passive DNS). After a thorough review, the university IRB concluded that the user study did not introduce privacy risks requiring IRB oversight.

To further mitigate any potential privacy risks, we hashed all IP addresses used in the study. This approach ensures that the IP addresses remain distinguishable for analysis while preventing the disclosure of any associated information. Also, the PyLingual web service prominently features an end-user agreement (Figure 3) that explicitly requests users' consent to share their PYC binaries and user patches for research purposes. Furthermore, PyLingual offers users the option to delete their files and patches at any time.

File retainment and intellectual property. To use the PyLingual web service, all users must accept a simple privacy agreement (Figure 3) that warrants they are aware that files and patches submitted will be retained for research purposes, and that by uploading files they are not violating any legal or contractual obligations.

Ethical consideration for classroom study. For the classroom user study conducted in a controlled environment with human participants, the university IRB approved the study under the exempt category[1]. The approval included administrative recommendations to ensure students were treated fairly and allowed to voluntarily decide whether to participate, free from any external pressure or coercion.

4 Decompilation Errors

In general, the perfect decompilation test recognizes two kinds of failures: *(1)* syntax errors (line 6 in Algorithm 1), and *(2)* semantic errors (line 12 in Algorithm 1). While different decompilers will have their own failure trends and idiosyncrasies, our study focuses on PyLingual for its broad accessibility and support for recent Python versions.

Syntactic errors. In many cases, recovered source code contains syntax errors that prevent it from being recompiled into a PYC binary to conduct bytecode-level analysis or to be compared with the original. These errors typically stem from issues such as incorrect indentation, truncated statements, or mismatched parentheses. To apply perfect decompilation and generate an equivalence report, users must first resolve all syntax errors.

Fixing syntax errors is straightforward — for example, by deleting code statements or blocks that contain errors. However, heavy-handed approaches often worsen inaccuracies in the decompilation, resulting in even greater discrepancies between the original PYC binary and the recompiled PYC binary. From our controlled user study, we found that users effectively utilized AI-based code assistance tools for these.

Semantic errors. When the recovered source code is a valid program, but contains instruction-level differences with the original binary, the recovered code contains semantic errors. While both — original and recompiled binaries are valid Python programs, their static representations are different assuming both are compiled using the same CPython compiler. In the context of perfect decompilation, it is possible for functionally equivalent decompilations to contain semantic errors.

PyLingual produces semantically incorrect results primarily due to its statistical nature, which relies on NLP components. NLP-based approaches are inherently error-prone and exhibit weaknesses when handling certain input types, such as long sequences, rare language syntax, and ambiguous code constructs. Additionally, deeply nested control structures and complex conditional clauses pose significant challenges for accurate decompilation. These limitations manifest as discrepancies in the decompiled output, which are identified during the equivalency verification phase. Correcting semantic errors requires users to have experience and knowledge of Python bytecode and its translation into Python statements. During both observational and controlled user studies, we observed that such knowledge and experience can be rapidly acquired over a few iterations of patching erroneous decompilations.

5 Perfect Decompilation User Study

By analyzing uploaded binaries and user patches collected from the PyLingual web service, we conducted an observational study against patches uploaded by Internet users (§5.1) and designed in-class assignments for students enrolled in a system security course assigning 40 malware samples incompletely decompiled by PyLingual with an exit survey at the end (§5.3, §5.4). Specifically, we aimed to address the following research questions:

- **RQ1:** Aare users inclined to fix incomplete decompilation outputs? If so, does the type of program affect their motivation? (§5.1, §5.2)
- **RQ2:** How difficult is it to repair incomplete decompilations? In particular, which subtasks of PYC decompilation are perceived as easy or difficult? (§5.3, §5.4)
- **RQ3:** How can the design of automatic decompilers be improved using perfect decompilation? (§5.4)

We answer these research questions using two studies with distinct methodologies and objectives. We begin with a purely observational study of Python binary files and subsequent source-level patches that anonymous internet users have uploaded to pylingual.io [15] to answer RQ1 (§5.1). The observational setting uniquely allows us to concretely study real-world decompiler users' motivations *without* the use of an artificial research setting (§5.2). However, the uncontrolled observational setting is ill-equipped to answer RQ2 because users unsurprisingly upload and subsequently

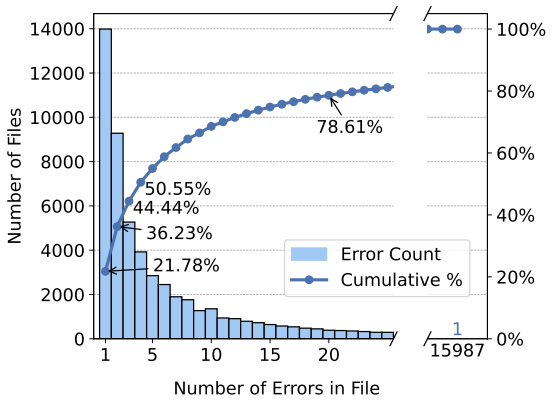


Figure 4: The number of *bytecode errors* in PYC files decompiled by PyLingual, including only files with one or more errors.

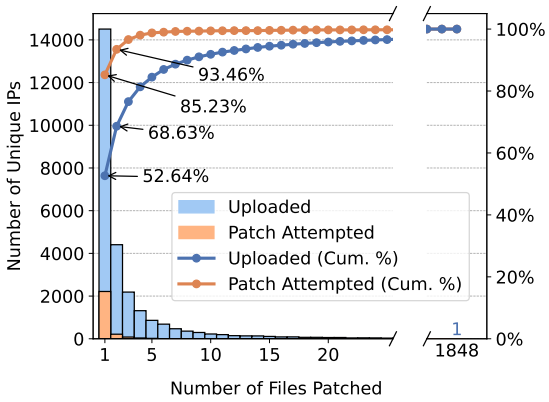


Figure 5: The number of *uploaded* and *patched* files by unique IPs in the wild. Most users only interact with a small number of PYCs.

patch different files, which makes direct patch difficulty comparisons challenging. Therefore, we pursue RQ2 with a controlled classroom study that examines the difficulty of patching incomplete decompilations by directly asking participants to patch several preselected incomplete decompilations. Similarly, answering RQ3 requires direct user feedback, which is not obtainable by a purely observational study, so we pursue RQ3 through a classroom study exit survey in §5.4.

5.1 Python Decompilation In The Wild

Our study focuses exclusively on PYC binaries uploaded to PyLingual because it is the only online decompiler that currently exposes a patching interface with perfect decompilation verification.

Since November 2023, pylingual.io has received 181,646 user-uploaded PYC binaries for decompilation from 27,555 unique IP addresses, of which 111,204 PYC (≈ 61.22%) decompiled perfectly without any patches needed. We focus on the remaining 70,442 PYC binaries that *did not* accurately decompile, and the subsequent efforts of users to repair the decompilation results. To this end, users have submitted 9,003 patches across 2,761 files and 2,599 IP addresses. These efforts have resulted in 393 additional files with semantically verified decompilations.

Contrasting these patch statistics with the decompilation statistics in Figure 4, it is immediately apparent that only a small fraction of incomplete decompilations receive patches from users, and only a small fraction of users engage in patching. Despite the small proportion of incomplete decompilations that are repaired by user-submitted patches, we find that most incomplete but syntactically valid decompilations are near-correct, with nearly 21.78% only including one bytecode error, and over 36.23% including just two or fewer bytecode errors. Here, a bytecode error is a contiguous run of incorrect bytecode instructions which were inserted, modified, or deleted, as compared to the original bytecode by a typical diff tool. However, we also find that the majority (≈ 63.78%) of incomplete decompilations contain syntax errors, which prevents bytecode-level analysis.

In Figure 5, the number of uploaded and patched files for each user heavily skews left, with single-file usage accounting for the majority of the total file uploads and patches. This pattern is evident from the bar graphs and cumulative percentage curve, which highlight a steep decline in activity beyond the first instance. Specifically, assuming each unique IP address corresponds to a unique user, 52.64% of file uploaders uploaded files only once and 85.23% of patchers patched a file only once. The curved graph, which represents the cumulative percentage, shows a sharp incline for the initial upload and patch, followed by a flattening trend. This trend highlights that how drastically the number of file uploads and patches decreases after the first instance. These observations suggest that most users visit PyLingual to decompile only a single PYC file, then depart after applying their patch.

Table 1: At $p \approx 1.7 \times 10^{-36}$ and $\chi^2 \approx 160$, online users whose first patch was successful were over 3 times more likely to patch further files.

	First Patch Failed	First Patch Success
Only Patched Once	1,990	225
Patched Multiple Times	252	132

Moving on to the initial analysis, in Table 1, a χ^2 test [30] for independence indicates that the success of a user's first patched file positively correlates with that user returning to patch additional files moderately but significantly. In a χ^2 test, the correlation coefficient r varies from -1, indicating a perfect negative correlation to 1, indicating a perfect positive correlation, with r values near 0 indicating weak to no correlation. The observed $r \approx 0.25$ indicates a weak positive correlation at $p \approx 1.7 \times 10^{-36}$, which is well beyond the conventional threshold for statistical significance of $p < 0.05$. In natural language, those users who experience an "early win" when patching files are over three times more likely to continue on to patch multiple files compared to the rest of PyLingual users. While this test does not establish causation, a successful patch indicates that the user is capable of using the patching interface, aware that

patching to achieve correctness guarantees is possible, and confident enough to approach the patching problem. In this analysis, we make the simplifying assumption that IP addresses map one-to-one with human users. Although this assumption is unlikely to hold in general, we still find that it leads to useful insights.

Given the low observed rates of patch attempts and successes, alongside the correlation between early successes and continued attempts, we will later investigate if difficulty is a key inhibiting factor of patching. Because an observational study lacks the ability to control for user knowledge and motivation, we design a controlled classroom study with systems security students that evaluates the ability of novice reverse engineers to successfully patch incomplete decompilation results in §5.3.

5.2 Semantics-Dependent Patching In The Wild

To identify factors that motivate online PyLingual users to decompile and subsequently patch imperfect decompilation results, we analyze the correlation between the contents of uploaded `PYC` files in the wild and online users' patching behavior. For this, we used the `jina-embeddings-v2-base-code` [31] long-document multilingual code-tuned embedding model to generate neural embeddings for each decompilation result in the dataset, then used unsupervised k-means clustering to produce 34 semantic clusters of files. We then labeled each cluster by uploading files to Gemini-2.0-Flash and receiving up to five descriptive tags per file. Joint tag frequencies per cluster were computed, and tags occurring fewer than 10 times were filtered out. Next, we calculated pairwise mutual information scores between each remaining tag and cluster. For each cluster, the top five tags with the highest mutual information scores were selected. Finally, these selected tags are used to describe the class of programs represented by each cluster. The full list of clusters is provided in Table 2, and related program demographic statistics are provided in our supplementary materials[2]. Although the clusters have some overlap and the accuracy of the cluster labels is limited by the stochastic and manual processes used to produce them, these semantic program clusters provide valuable insights into the relationship between program file contents and the varied goals of real-world reverse engineers.

Among the diverse `PYC` files that reversers seek to decompile in practice, we find malware payloads, web servers, desktop automation scripts, creative software, and more. We also observe multiple flavors of obfuscation, including traditional tools like PyArmor [32], variable and string scramblers like Oxyry [33], to powershell-esque recursive decompression and `exec` calls on opaque bytestrings. We even observe substantial representation of Python standard library files, which are included in standalone executable builds from packaging tools like PyInstaller [34].

Patching activity by cluster. Figure 6 shows the univariate distribution of patching activity by cluster. The most frequently patched clusters, 15 (Game Cheats), 31 (Credential Stealers), and 30 (Trading Bots), are also some of the most commonly co-occurring clusters containing a high proportion of malicious programs. Other highly patched clusters include cluster 20 (Automation Bots) and cluster 33 (Graphical Interfaces), which commonly co-occurred with cluster 30. Three of the least frequently patched clusters are 19 (Plotly Objects), 3 (Plotly Validators), and 24 (Oracle Cloud Infrastructure),

Table 2: Descriptions and proportion of `PYC`s with VirusTotal alerts for each semantic cluster from the wild.

ID	Description	VT Alerts	ID	Description	VT Alerts
0	Data Visualization	0.00%	17	Concurrency	0.00%
1	Database ORM	0.00%	18	Code Analysis	0.00%
2	Interface Definitions	0.00%	19	Plotly Objects	0.00%
3	Plotly Validators	0.00%	20	Automation Bots	0.13%
4	Cryptography and Auth	0.00%	21	Web Clients	0.00%
5	System Tools	0.64%	22	Google API Clients	0.00%
6	Testing Frameworks	0.00%	23	API Definitions	0.00%
7	Obfuscated Code	10.84%	24	Oracle Cloud Infrastructure	0.00%
8	Syntax Highlighting	0.00%	25	Machine Learning	0.00%
9	LLM Applications	0.00%	26	Web Frameworks	0.00%
10	Simulation and Modelling	1.87%	27	Cloud APIs	0.00%
11	Cloud API Clients	0.00%	28	Encryption/Decryption	8.27%
12	File Management	0.20%	29	Code Packing	0.00%
13	Package Management	0.20%	30	Trading Bots	1.84%
14	Game Modding	0.00%	31	Credential Stealers	7.61%
15	Game Cheats	0.35%	32	Cloud Workflows	0.00%
16	Scientific Computing	0.00%	33	Graphical Interfaces	0.04%

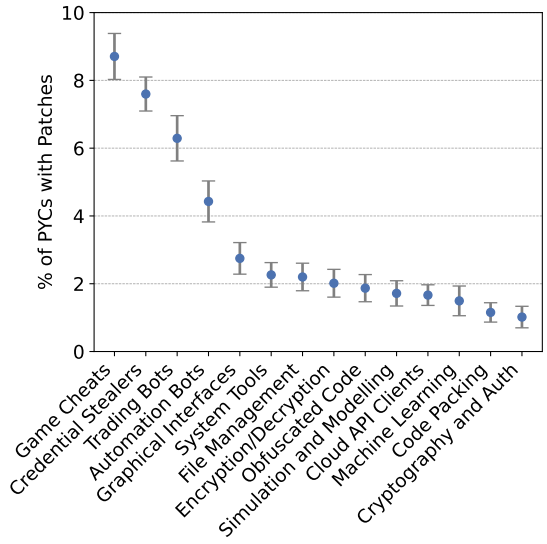


Figure 6: Observed patch rate in the wild by semantic cluster, with 95% confidence intervals. Only clusters where at least 1% of uploaded files are patched are shown. See Table 2 for all clusters.

all with zero patches uploaded. Clusters 19 and 3 are different parts of the open-source Plotly library that appear to have been decompiled in bulk by few unique IP addresses, most likely for testing purposes. Cluster 24 contains Oracle Cloud Infrastructure clients and definitions, which also appear to have been uploaded as part of some bulk decompilations. Overall, we see that users are more likely to patch files that are part of malicious software — these users are likely highly motivated to achieve decompilation so they can ensure they have correctly reverse-engineered these malware samples. The least motivated patchers are either testing the quality of the decompilation service or are decompiling large projects in bulk,

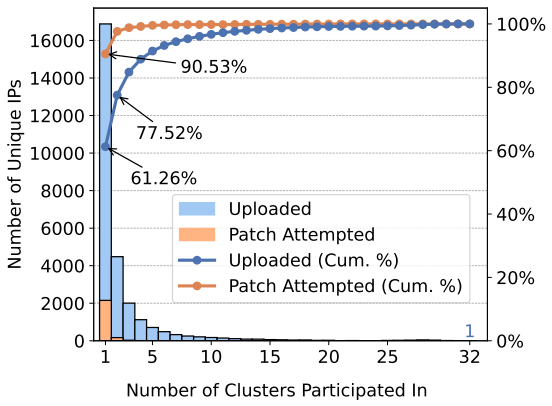
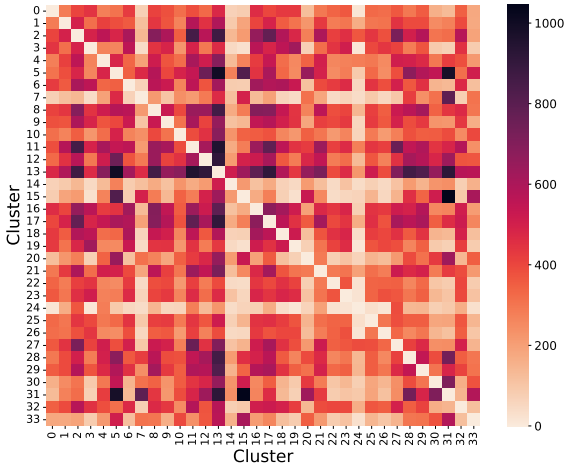
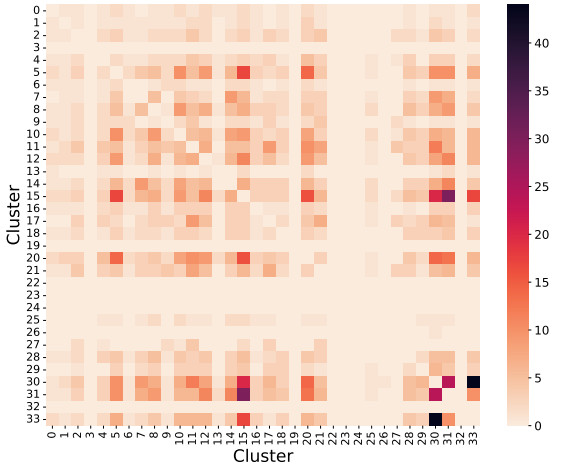


Figure 7: Semantic cluster participation by unique IPs in the wild.



(a) Heatmap of shared uploader counts for cluster pairs in the wild.



(b) Heatmap of shared patcher counts for cluster pairs in the wild.

Figure 8: Heatmaps of user activity across pairs of semantic clusters. For each pair, we show the number of users that uploaded/patched at least one file belonging to each cluster in the pair.

and are therefore less motivated to manually correct decompilation errors.

Semantic cluster co-occurrence analysis. Dividing the data into clusters helps reveal users' specific interests and demonstrates possible overlaps in user activities across clusters. To explore usage correlations between clusters, we measured the extent of user cross-involvement between pairs of semantic clusters. As shown in Figure 7, most users—61.26% of uploaders and 90.53% of patchers—participate in only one cluster. The cumulative percentage curve flattens exponentially beyond a single cluster, indicating that users' activity is often confined to a single area of interest.

Figure 8 is a heatmap of the number of users that showed activity in *at least* the pair of clusters associated with each cell. That is, a user who uploads files in three clusters will contribute to two cells above the main diagonal, one for each pair of cluster participation. In the upload activity, we found five pairs of frequently co-occurring clusters, as well as three cluster pairs that shared a common malicious cluster, cluster 31 (credential stealer) as a pair component.

Out of 34 clusters, 9 clusters contain less than 1% of malicious behavior, and can therefore programs reside in such clusters can be treated as either malicious or benign. Notably, cluster 5 (System Tools) commonly pairs with both malicious and benign clusters, acting as an intermediate cluster.

Cluster 5 (System Tools) and cluster 31 (Credential Stealers) pair shows the adversarial usage of cluster 5. Tags such as *keylogger* and *registry-modification* in cluster 5 suggest that it includes malicious programs that monitor user activity or manipulate the Windows registry to achieve persistence execution. Cluster 31 can leverage such behavior from cluster 5 to extract authentication tokens, personal information, and the key of cryptocurrency wallet from local applications and web browsers. This overlap highlights how certain system-level capabilities are exploited to facilitate undesired activity, revealing the types of behaviors adversaries are one of the most interested in combining.

Cluster 31 (Credential Stealers) also displays high co-occurrence with cluster 13 (Package Metadata) and cluster 15 (Game Cheats). Anomalous behaviors still require access to common Python libraries and utilities, often sourced from benign-looking package metadata to facilitate malicious activity. Interestingly, cluster 15 is frequently paired with cluster 31, suggesting that attackers often attempt to extract user credentials from gaming platforms. For instance, these programs typically spoof User-Agent headers to mimic legitimate game clients, then send crafted login requests to the target service to obtain access tokens from the response. This attack indicates a recurring pattern where credential stealers repurpose gaming-related utilities to infiltrate user accounts under the guise of legitimate traffic.

Figure 8b presents a heatmap highlighting co-occurrences of users *patching* files across cluster pairs. Since patches are significantly fewer than uploads, this heatmap is notably sparser than Figure 8a. Among the most prominent co-occurring pairs are cluster

31 (Credential Stealers) with cluster 5 (System Tools) and cluster 30 (Trading Bots), as well as the pairing of cluster 30 with cluster 33 (GUIs). These clusters represent security-relevant categories that offensive reverse engineers attempt to patch more frequently compared to other clusters, aiming to deploy attacks. Recall that each of these clusters independently attracts a disproportionately high number of patches; their intersection suggests a shared intent among users to modify or integrate malware-related functionality.

In cluster 30 (Trading Bots), malicious programs often include automated data exfiltration routines targeting system information, browser data, Discord tokens, and mechanisms that block access to security-related websites or task managers. We also observed that GUI front-ends (cluster 33) are frequently employed in ransomware to demand payments or to masquerade malicious operations behind a convincing user interface. Taken together, these behaviors underscore how adversaries integrate specialized components—such as trading bots, system utilities, and deceptive GUIs—into attacks. Their shared presence in patch activity describes attackers' continuous efforts to enhance success rate of their attacks by integrating multiple malicious features.

5.3 Classroom Study

To test the difficulty of correcting incomplete decompilation through user-submitted patches in isolation of hidden variables in the observational user study, such as user background, motivation, and technical expertise, we design and execute a classroom study. To respect the privacy of the 37 student participants, the only data collected in the study are anonymized patch histories and an exit survey. The form of the patch histories for analysis is the same as in the observational study, the key differences being that we control the challenge binaries and operate with a known population of patchers. The typical student participant has a strong technical background, but negligible experience with `PYC` bytecode or Python reverse engineering; the generalized conclusions from the classroom study assume that the capabilities of the student participants adequately approximate those of a typical junior reverse engineer.

Malware assignment summary. To emulate the conditions of real-world reverse engineers, we selected 40 `PYC` malware samples with decompilation errors, prioritizing those with a greater number of VirusTotal alerts and mitigating duplicates. These patching challenges, enumerated in our supplementary material[2], provide a practical view into how junior reversers approach `PYC` malware decompilation. Most of the challenges used in this study were stealers for cookies, credentials, and authentication tokens. Thirteen of these files have anti-debugging or anti-vm capabilities attempting to prevent reverse engineers from dynamically analyzing them. The remaining files were Remote Access Trojans (RAT) and keyloggers. Some of these files added themselves as startup programs to persist across sessions with Anti-Antivirus features keeping the host infected.

The challenge `PYC` malware samples were provided to the students through a shared sandbox machine to minimize the risk of harm to the students. This sandbox machine included `PYC` reverse engineering tools, such as `xdis` [35] and a control flow graph visualization script, which were advertised to the students. In the assignment, students received points for each malware sample they successfully patched. The scoring mechanism is as follows: the first 10 submitted challenges are worth 20 points each, for a total of 200 points; the next 10 submitted challenges are worth 10 points each, for a total of 100 points; the final 20 submitted challenges are worth 5 points each, for a total of 100 points. Full credit for the assignment was designated at 200 points, with the remaining 200 points for challenges beyond the 10th left as extra credit.

The incentive structure is the most significant difference between the classroom participants and the real-world users. The students approached `PYC`s from the wild using the same publicly available tools as real Python reversers, but were explicitly assigned perfect decompilation as a goal instead of regarding it as an intermediate step in a wider reversing process, similar to previous studies in this space [21]. Students seeking to minimize their workload while maximizing their score are incentivized to "scout" the challenge `PYC`s, selectively choosing the easiest ones to attempt. Incentivizing scouting is an intentional design choice made to amplify the signal from students' perception of difficulty as measured by the engagement rate of each challenge, where the engagement rate is the ratio of the number of students who attempted a challenge by submitting a patch versus the students who simply viewed a challenge without attempting a patch.

General patching workflow. The process of repairing a partially correct decompilation result can be conceptualized as several iterative improvements over the baseline decompiler output, each consisting of a simple process: *(1)* identify an error to fix; *(2)* locate the error in the source code; *(3)* modify the source code to fix the error; and *(4)* recompile the modified source code to verify that the error was fixed as intended. The form and execution of each step changes depending on the specific error in question, which makes automating the patching process difficult. The order in which errors are approached is ideally determined by the extent to which they obscure or distract from other errors. For example, syntax errors should be addressed first because they prevent the compilation step needed to identify and fix bytecode errors. Fortunately, syntax errors are easy to identify, locate, and fix due to helpful error messages from the compiler. Bytecode errors can be identified by analyzing the difference in instructions between the original `PYC` file and the candidate `PYC` file. We summarize common patching approaches and pain points identified in our exit survey in §5.4.

Patch difficulty and effort estimation. Figure 9 shows that participants were consistently able to accurately patch the challenge `PYC`s, with the median student successfully patching 10 challenges, achieving full credit for the assignment. 15 of the 37 participants completed more than 10 challenges, with the most prolific student patcher producing 28 successful patches out of the 40 available challenges. This trend is unsurprising given the incentive structure of the classroom study, and demonstrates that even novice reversers are able to repair incomplete decompilation results when appropriately motivated. Additionally, we find that the success rates of participants are generally high, with a median success rate of 63.16%, and 4 participants achieving a 100% success rate, eventually fully solving every challenge where they attempted a patch.

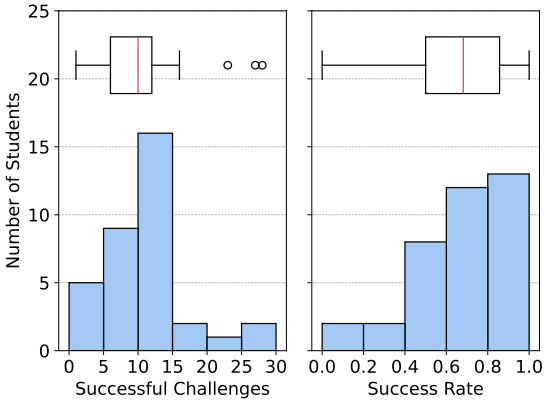


Figure 9: A histogram and box plot of the number of successful challenges and individual success rate of participants in the classroom study. $Success\ Rate = \frac{\#\ of\ Successful\ Challenges}{\#\ of\ Attempted\ Challenges}$

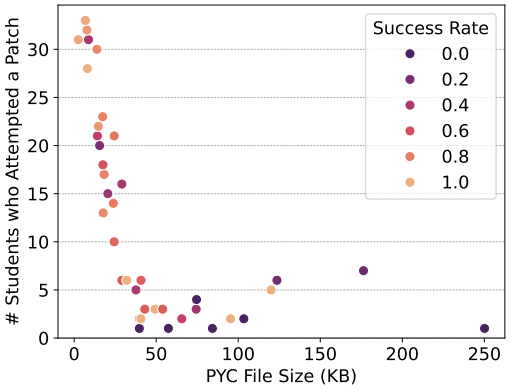


Figure 11: Small PYC files received disproportionately many attempts from classroom study participants and yielded higher success rates than larger PYC files.

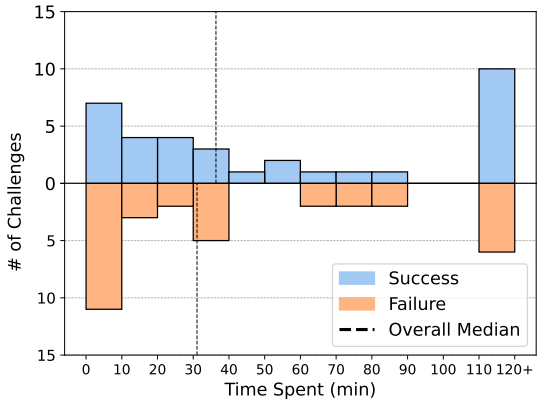


Figure 10: Distribution of the median time taken on each challenge during the classroom study. Median times were taken separately for successful and unsuccessful patches.

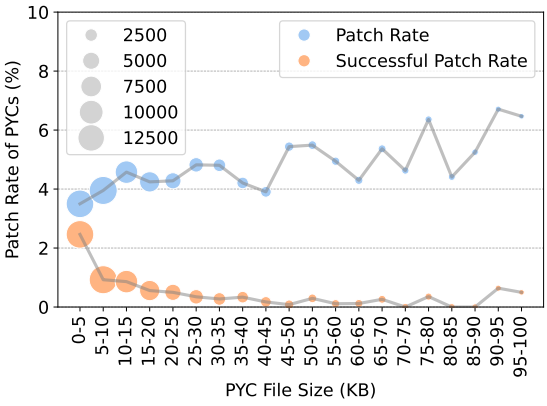


Figure 12: In the wild, file size strongly impacts patch success rates, but does not strongly impact patch attempt rates.

Further, in Figure 10, we find that participants did not typically spend excessive time on each challenge. Measuring the time between the first patch submission and last patch submission for each participant-challenge pair, we see that the median time to either succeed or abandon the challenge is approximately 30 minutes. We also see several high outliers caused by participants briefly working on a challenge on one day, then returning to complete it on a later date. Considering the time spent alongside the student success rates, we find that patching imperfect decompilations is neither prohibitively difficult nor prohibitively labor-intensive.

Challenge selection strategies. Surprisingly, despite the design of the assignment encouraging scouting — uploading many challenges and selectively choosing easy ones to attempt based on the decompilation error reports — 40% of participants created at least one patch for every challenge they uploaded to PyLingual. Further, each of the 4 participants with a 100% success rate attempted to patch every challenge they uploaded. Only 24% of participants rejected more than two files, which at first glance appears to indicate that the participants were not selective about which challenges to attempt. However, Figure 11 reveals a simple strategy employed by almost all of the participants: use the file size of the challenge PYCs as a proxy for difficulty.

Although naïve, the efficacy of this strategy is corroborated by patch success rates from the wild, shown in Figure 12. As the size of the PYC file increases, the number of instructions and code objects also tends to increase (with the notable exception of files containing large encrypted payloads). Intuitively, the more code a PYC contains, the more opportunities there are for decompilers to make mistakes; this tendency is amplified in neural decompilers. Despite the clear impact of file size on patching difficulty, it does not negatively impact reversers' willingness to attempt patches in the wild.

5.4 Classroom Study Exit Survey

Near the end of the classroom study period, we administered an optional exit survey consisting of 27 questions (available in our supplementary materials [2]) focused on how students approached perfect decompilation and the challenges they encountered during the process. We received 26 responses from a total of 37 study participants. We were unable to conduct a similar exit survey for the observational study participants because we did not collect identifiable information or perform interventions.

The core research question we asked was: "Did the process of PYC patching help students better understand the malware?" The majority of students agreed that the patching process was helpful (2 strongly agree, 13 agree, 7 somewhat agree, 1 somewhat disagree, and 1 strongly disagree). These responses suggest that the act of patching—aimed at achieving perfect decompilation—can be effective in deepening students' understanding of malware semantics, in addition to the inherent benefits of recovering recompilable source code for a given malware sample. Another conclusion from our survey results is that novice reversers tend to adopt very simple strategies to fix decompilation errors and heavily rely on large language models (LLMs) to translate bytecode sequences to Python statements.

When asked in what order they approached bytecode errors in an imperfectly decompiled PYC file, the majority of students reported simply starting at the top of the file and working downwards, as opposed to trying to prioritize code objects based on any heuristic. Anecdotally, we have found that the best order for approaching bytecode errors is *depth-first* within each control flow structure, starting from the entry point of each code object. By starting from the top of each code object, patchers can reduce the amount of distracting artifacts that are present in the bytecode diff view, and by fixing the innermost control flow elements first, patchers can more accurately compare jump targets between the original and candidate PYC files.

In Figure 13, students reported less frequently using more technical strategies to map bytecode errors to decompiled source. When asked how often they used tools other than PyLingual, students' responses were split approximately evenly between those who very frequently used tools and those who rarely used any at all. Out of the students who reported using tools, the vast majority of them reported using LLMs, with very few using tools specialized for PYC decompilation. Despite being instructed in class on how to use two useful external tools — `xdis` [35] and a tool to visualize control flow graphs — very few students reported using either of these tools. Instead, students largely relied on what was given to them in the PyLingual interface, with some help from LLMs for understanding bytecode instructions and unclear errors.

These strategies explain in part why students found Python decompilation to be a very tedious process. When asked to rate how often they felt stuck on a challenge, most students selected "Often" or "Very Often". Figure 14 shows how students rated the difficulty of various different decompilation errors, with control flow and exception handling errors being the most difficult. Since few students used the control flow visualization tool, it makes sense that control flow was difficult for them to handle. Similarly, exception handling in Python version 3.11+ uses an exception table

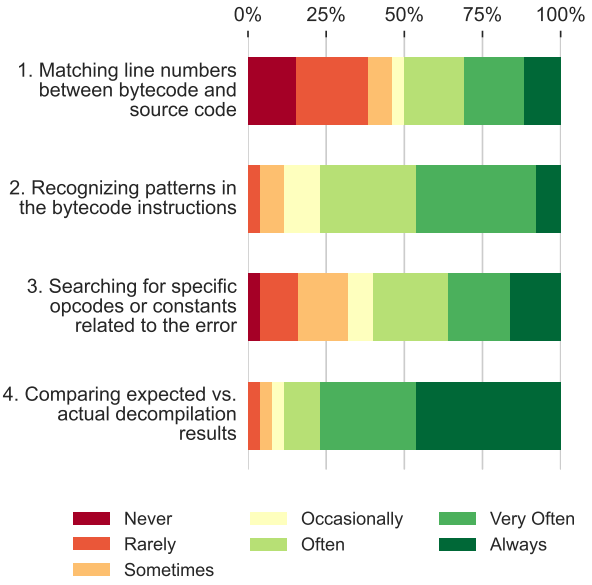


Figure 13: Classroom exit survey responses to "How often did you use each of the following strategies to locate the source code section responsible for a particular error?"

that is not shown in the PyLingual interface, but is shown by `xdis` [35] and the control flow visualization tool. In short answer responses, students overwhelmingly cited these issues as the most frustrating to deal with.

These results indicate a few potential improvements to the decompiler interface that would help novice reversers. Since students intuitively chose to approach files from the top down, it may be helpful to use the patching interface to nudge them towards control flow oriented approaches. The significant difficulty understanding control flow and the infrequent use of external tools indicates that the current decompilation interface does a poor job expressing how to handle control flow errors. The interface should integrate tools for control flow visualization so they are more accessible to novice reversers, as this would allow them to better understand errors that they find difficult to reason with when the code is only laid out linearly.

In summary, while participants agreed that patching incomplete decompilations helped them better understand the provided malware samples, they heavily relied on LLMs to fix statement-level errors. However, despite assistance from LLMs, many participants struggled with control-flow-related issues, which current LLMs are still unable to handle accurately. For future improvements to the web-based IDE and other decompiler interfaces, we propose two design directions: *(1)* integrating LLMs more seamlessly into the patching workflow, and *(2)* enhancing CFG visualization and its linkage to both bytecode and source code interfaces.

6 Related Work

There has been a growing body of research on user behavior in reversing and decompiling native binaries. We also review related

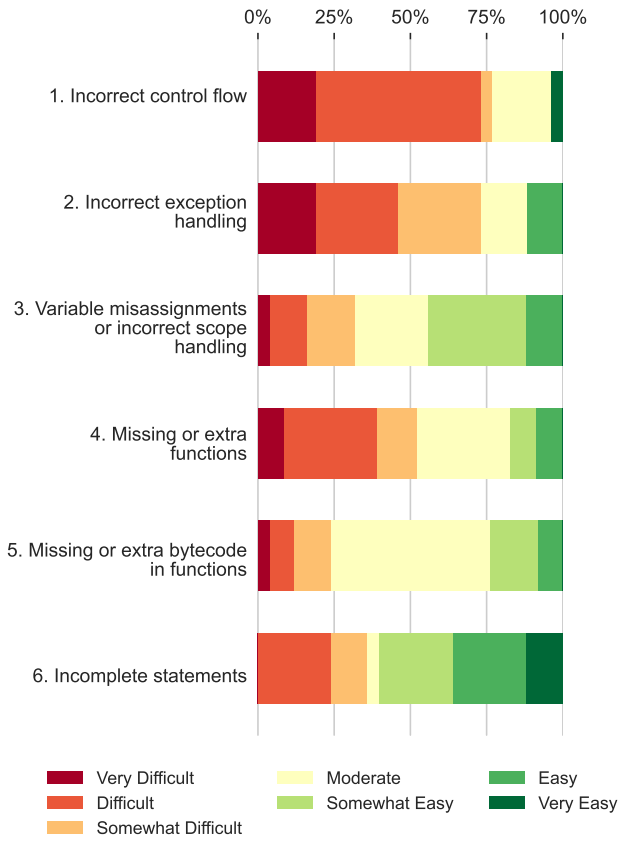


Figure 14: Classroom exit survey responses to "How difficult was correcting the following decompiler errors?"

work on native binary decompilation, as well as a separate line of research focused on the decompilation of HDL binaries.

6.1 Reversing User Study

Votipka et al. [36] interviewed sixteen reverse engineers as they reenacted a recent reverse engineering activity. Through these semi-structured interviews, the authors extracted common reverse engineering strategies. Mantovani et al. [37] studied how humans reverse engineer binary code, with an emphasis on the differences between novices and experts. Their approach used a Restricted Focus Viewer: basic blocks were blurred unless selected, allowing the authors to precisely track the subjects' attention. Most of these studies were — necessarily — small. They examined the reversing process using methods that were human-friendly but not amenable to automated analysis. As a result, the data was limited to what the researchers could analyze manually.

Burk et al. [21] introduced the concept of perfect decompilation as a tool for evaluating how users approach binary reverse engineering tasks. Their work demonstrated the practical viability of generating semantically equivalent source code and highlighted its potential to enhance both analysis accuracy and usability in reverse engineering workflows. In their study, the authors conducted a structured user study through a series of CTF-style binary reversing challenges. These challenges were explicitly designed to examine how users complete decompilation tasks when working with partial or incorrect outputs—an expected norm due to the inherent limitations of traditional decompilers, which fail to cover the original program semantics after compilation and optimization.

While prior research has focused on a curated set of native binaries, our study leverages the properties of Python bytecode to extand automated analysis of reversing success to the wild. We investigate the feasibility of perfect decompilation in this context using a large-scale real-world dataset that includes a wide range of Python malware samples. Our analysis provides actionable insights for advancing decompilation techniques in both Python-specific and general reverse engineering contexts.

6.2 Native 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 [38–45]. 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. [38] 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 [46], and *(2) structural analysis* to identify code blocks and restore control dependencies among them [38–41]. Structural analysis has more impact on the performance and usability of a decompiler, so it has been the primary focus of recent research [39–41].

A few studies have investigated the usability of RE tools. For instance, researchers have looked at improving the usability of decompilers [40, 41] showing that better variable naming and a reduced number of `GOTO`s affected positively the readability of the pseudocode.

6.3 HDL Decompilation

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 Cyber Physical System (CPS) computing sectors. In response, developers and malware authors alike have minimized external dependencies with architecture-neutral formats, standardized modules, and adaptable runtime components [5, 47, 48]. 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. Traditionally, `uncompyle6` [16] evolved from early attempts at creating a decompiler that leveraged the same strategies as traditional compilers covering Python 2.7 - 3.6.

`decompyle3` [17] 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 support Python 3.7 and 3.8, with the most recent release in 2025 still providing no public support to Python 3.9 or later.

`pycdc` [18] 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.

PyLingual [15] is an NLP-assisted Python bytecode decompiler that scales over the multiple Python versions with higher accuracy than the existing decompiler. Our research is based on a novel accuracy verification mechanism "perfect decompilation", where decompiled output is validated against the original binary for strict semantic equivalence thereby users can upload their patches to correct erroneous decompilation outputs.

Decompilers for other HDLs. `Soot` [49], 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 [50, 51] are also actively maintained. Golang and Rust do not have bytecode representation exposed to the user. Instead, users can directly produce native binaries for different architectures. The decompilation for such output binaries is more challenging, as they define proprietary formats and applies aggressive optimizations. We have seen malware written using Golang and Rust due to their conveniences and architecture coverage. While no reliable support to reverse such binaries, it is imperative to have stable decompilation support. Although niche and thus not actively maintained, decompilers also exist for other HDL families such as Ruby and Lua [52, 53]. 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.

7 Discussion and Future Work

Limitations. Being the first study to explore user responses to incomplete decompilation in the wild, our study still has several limitations. The core of the data comes from pylingual.io, which is, to our knowledge, the only online decompiler that exposes a patching interface with perfect decompilation verification. The file upload and user patch data from PyLingual is limited to CPython binaries. Identifiable information about the users is not collected, so we could not directly survey the observational study participants; our conclusions about their motivations are inferred from the observed file semantics and patching behaviors. The generalizability of difficulty estimations from the classroom study rests on the assumption that undergraduate ybersecurity students are similar to junior reverse engineers. The students in the controlled study were taught about perfect decompilation, the Python interpreter, and common Python reversing tools (*e.g.,* `xdis` [35]).

Implications of practical perfect decompilation. Our classroom study showed that even inexperienced reversers can achieve perfect decompilation with moderate effort, but our observational study showed that most decompiler users in the wild do not choose to pursue it. Practically, many decompiler use-cases do not require perfect decompilation, but there are some use-cases that see an outsized benefit, as we see in the semantic cluster analysis from the observational study. Successful perfect decompilation proves the correctness of a reverse engineering effort, enabling the result to be easily trusted by other reversers and supports source-level downstream tasks. Specifically, perfectly decompiling a low-level program enables it to be modified at the source level and then recompiled. In the context of security for closed-source Python applications, there is very little tool support for bytecode-level program modification, so the ability to patch at the source level is advantageous. Further, correct source code can be recompiled in a different Python version, allowing application consumers to upgrade the interpreter version used in packaged applications to benefit from security patches and efficiency improvements.

Human-in-the-Loop decompilers. Seeing the ability of humans to perform a wide range of decompilation subtasks, future decompiler works should explore human-centric interfaces that better enable users to contrast the semantics of the decompiled code with that of the original low-level program. While current studies of human-decompiler collaboration are limited by the separation of the automatic stage and human stage, there may be substantial gains in efficiency by finding better methods for humans to interact with the decompiler than simply editing the output. For example, an expert user could make changes to the intermediate form of the decompilation output between stages of the automatic decompiler, such as manually restructuring some small part of the control flow. Further, uncertainty heuristics could expose key decision points in the decompilation process where human reversers could apply domain-specific insights to improve decompilation outcomes.

Automatic decompiler error reports. Under the current "last-mile" user involvement paradigm, successfully repairing a failed decompilation results in a pair of a flawed automatic decompiler output and an exemplar perfect decompilation. Large sets of these pairs could be used to statistically narrow down the root cause of common decompiler errors by matching commonalities in the input low-level programs with commonalities in the patches required to repair the decompilation result. The primary roadblock in this direction is privacy. Reverse engineers are often privacy-conscious, and may be unwilling to share their input low-level programs and patches in order to help improve publicly available decompilers. Future work may investigate privacy-preserving aggregation of patch analytics to automatically produce usable error reports for decompiler developers.

Language extensions for fine-grained patch control. One of the major concerns with the difficulty of automatically verifiable decompilation is "fighting the compiler" – blindly iterating through many semantically equivalent source code candidates trying to generate the exact right low-level code sequence, typically by triggering the right sets of compiler optimizations. In our study, we

found three such cases that caused participants to fight with the decompiler before eventually stumbling into the right source code, asking for assistance, or giving up. These cases included: *(1)* a list made of constant f-strings instead of regular string constants to avoid the list being pre-built at compile time; *(2)* two identical lambdas in the same expression placed on separate source lines instead of one line to prevent the compiler from reusing the same lambda code object; and *(3)* an `assert` statement instead of `if ... raise AssertionError` to trigger the right grouping of compile-time boolean inversions. To alleviate such annoyances, we notice that the compiler used to produce the validation `PYC` candidate does not necessarily need to be the same compiler that produced the original `PYC`; it only needs to be *capable* of producing the original `PYC`. Future work may explore language extensions and compiler variants that allow the user to exert explicit control over compiler optimizations to reduce the impact of fighting with the compiler.

Large language models as human surrogates. In recent years, large language models have provided state-of-the-art performance across a wide range of text processing, coding, and reasoning tasks. While [54] found that naively applying language models to perform end-to-end decompilation was not effective, using language models to repair localized decompiler failures may still be beneficial. In this line, a foundation model could be fine-tuned using many pairs of failed and patched decompilation results so that it can learn to apply the most common fixes, reducing the workload required from humans. The prospect of leveraging large language models for targeted corrections becomes especially promising in the context of recent developments in reinforcement-learning-based reasoning improvements [55, 56] and large-context architectures [57, 58].

8 Conclusion

In this paper, we presented a comprehensive study of human-assisted decompilation of `PYC` binaries, with a focus on achieving perfect decompilation from erroneous outputs. Leveraging a unique dataset of real-world `PYC` binaries and user-submitted patches, we investigated how reverse engineers approach the task of correcting imperfect decompilation results. Our analysis combined both observational and controlled user studies, identifying key factors that influence the success of human-guided decompilation and offering actionable insights for improving decompiler design.

Our findings reveal that certain types of programs—such as unobfuscated malware and software with high levels of user interaction—are more likely to motivate users to pursue accurate decompilation. Notably, the overall difficulty of patching errors was not perceived as a significant barrier. By analyzing the strategies of successful reversers, we identified patterns and behaviors that can inform the development of more effective decompilation tools.

We also examined users' general workflows and tool preferences when correcting decompilation errors. Many participants reported frequent reliance on LLMs, particularly for addressing statement-level issues. However, challenges remained in resolving control flow errors, which current LLMs struggle to handle effectively.

Lastly, our research demonstrates both the feasibility and the usefulness of *perfect decompilation* in the context of Python binaries. We believe our findings open new avenues for benefiting other classes of decompilation tasks, including native binaries.

Acknowledgments

The authors would like to thank the anonymous reviewers for their valuable comments and suggestions. This work was supported by National Science Foundation grants 2321117 and 2331424, National Institute of Standards and Technology grant 14147259, and Eugene McDermott Graduate Fellowships 202208, 202504, and 202405. 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). This work is based upon the work supported by the National Center for Transportation Cybersecurity and Resiliency (TraCR) (a U.S. Department of Transportation National University Transportation Center) headquartered at Clemson University, Clemson, South Carolina, USA. Any opinions, findings, conclusions, and recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of TraCR, and the U.S. Government assumes no liability for the contents or use thereof.

References

[1] *PYPL PopularitY of Programming Language Index*. 2025. (Visited on 01/22/2025).

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

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

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

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

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

[7] Kotaro Ogino. *Unboxing Snake - Python Infostealer Lurking Through Messaging Services*. https://www.cybereason.com/blog/unboxing-snake-python-infostealer-lurking-through-messaging-service.

[8] Josh 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] 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.

[11] Warren Mercer. *PoetRAT: Python RAT uses COVID-19 lures to target Azerbaijan public and private sectors*. https://blog.talosintelligence.com/poetrat-covid-19-lures/.

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

[13] 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/.

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

[15] Joshua Wiedemeier et al. "PyLingual: Toward Perfect Decompilation of Evolving High-Level Languages". In: *IEEE Symposium on Security and Privacy (SP)*. IEEE Symposium on Security and Privacy (SP). May 2025.

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

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

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

[19] *syssec-utd/pylingual: Python decompiler for modern Python versions*. URL: https://github.com/syssec-utd/pylingual.

[20] *PyLingual*. https://pylingual.io/.

[21] 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.

[22] Eric Schulte et al. "Evolving Exact Decompilation". In: *Workshop on Binary Analysis Research (BAR)*. Workshop on Binary Analysis Research (BAR). ISBN: 1891562509. DOI: 10.14722/bar.2018.23008.

[23] *Status of Python versions*. URL: https://devguide.python.org/versions/.

[24] Jacob Devlin et al. "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, June 2019, pp. 4171–4186. DOI: 10.18653/v1/N19-1423. URL: https://aclanthology.org/N19-1423.

[25] 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.

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

[27] William M McKeeman. "Differential testing for software". In: *Digital Technical Journal* 10.1 (1998), pp. 100–107.

[28] *Monaco Editor*. URL: https://microsoft.github.io/monaco-editor/.

[29] Saed Alrabaee et al. "Compiler Provenance Attribution". In: *Binary Code Fingerprinting for Cybersecurity: Application to Malicious Code Fingerprinting*. Cham: Springer International Publishing, 2020, pp. 45–78. ISBN: 978-3-030-34238-8. DOI: 10.1007/978-3-030-34238-8_3. URL: https://doi.org/10.1007/978-3-030-34238-8_3.

[30] Mary L McHugh. "The chi-square test of independence". In: *Biochemia medica* 23.2 (2013), pp. 143–149.

[31] Michael Günther et al. *Jina Embeddings 2: 8192-Token General-Purpose Text Embeddings for Long Documents*. 2024. arXiv: 2310.19923 [cs.CL]. URL: https://arxiv.org/abs/2310.19923.

[32] dashingsoft. *pyarmor*. https://github.com/dashingsoft/pyarmor.

[33] *Oxyry Python Obfuscator*. 2025. (Visited on 01/22/2025).

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

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

[36] Daniel Votipka et al. "An Observational Investigation of Reverse Engineers' Processes". In: *USENIX Security Symposium (SEC)*. USENIX Security Symposium (SEC). USENIX Association, Oct. 2020, pp. 1875–1892. ISBN: 978-1-939133-17-5. URL: https://www.usenix.org/conference/usenixsecurity20/presentation/votipka-observational.

[37] Alessandro Mantovani et al. "RE-Mind: a First Look Inside the Mind of a Reverse Engineer". In: *USENIX Security Symposium (SEC)*. USENIX Security Symposium (SEC). Boston, MA: USENIX Association, Jan. 2022, pp. 2727–2745. ISBN: 978-1-939133-31-1. URL: https://www.usenix.org/conference/usenixsecurity22/presentation/mantovani.

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

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

[40] 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.

[41] Khaled Yakdan et al. "Helping Johnny to Analyze Malware: A Usability-Optimized Decompiler and Malware Analysis User Study". In: (May 2016), pp. 158–177.

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

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

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

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

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

[47] *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.

[48] *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/.

[49] 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.

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

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

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

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

[54] Josh Wiedemeier et al. " PYLINGUAL: Toward Perfect Decompilation of Evolving High-Level Languages ". In: *2025 IEEE Symposium on Security and Privacy (SP)*. Los Alamitos, CA, USA: IEEE Computer Society, May 2025, pp. 52–52. DOI: 10.1109/SP61157.2025.00052. URL: https://doi.ieeecomputersociety.org/10.1109/SP61157.2025.00052.

[55] Tianyang Zhong et al. *Evaluation of OpenAI o1: Opportunities and Challenges of AGI*. 2024. arXiv: 2409.18486 [cs.CL]. URL: https://arxiv.org/abs/2409.18486.

[56] DeepSeek-AI et al. *DeepSeek-R1: Incentivizing Reasoning Capability in LLMs via Reinforcement Learning*. 2025. arXiv: 2501.12948 [cs.CL]. URL: https://arxiv.org/abs/2501.12948.

[57] Albert Gu and Tri Dao. *Mamba: Linear-Time Sequence Modeling with Selective State Spaces*. 2024. arXiv: 2312.00752 [cs.LG]. URL: https://arxiv.org/abs/2312.00752.

[58] Ali Behrouz, Peilin Zhong, and Vahab Mirrokni. *Titans: Learning to Memorize at Test Time*. 2024. arXiv: 2501.00663 [cs.LG]. URL: https://arxiv.org/abs/2501.00663.