Walking The Last Mile: Studying Decompiler Output Correction in Practice

The increasing prevalence of Python as a language for both benign and malicious software has spurred interest in decompiling Python PYC bytecode back to its source code. This work presents a comprehensive study on human-assisted Python decompilation, leveraging extensive data collection from the PyLingual web service, comprising over 61,000 PYC binaries, 3,465 user-submitted patches, and 253 accuracy-verified patches. We investigate how reverse engineers respond to incomplete decompilation results and identify factors influencing their efforts to achieve accurate decompilation. Our study combines observational analysis of real-world data with a controlled user study involving graduate systems security students.

We reveal that the semantic content of programs significantly impact patching efforts in the wild, while the difficulty of patching is not a major barrier. By contrasting observed patching behavior between student participants and reversers in the wild, we gain insights into the incentive structures and accuracy sensitivity of real-world reversers. We provide actionable recommendations for improving PYC decompilation and integrating human intelligence into the decompilation workflow. Our findings contribute to enhancing the accuracy and performance of automatic decompilers and their user interfaces.

1 Introduction

Many malware payloads, as well as benign software, 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 time and resources while 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, 2, 30, 38–47]. 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 [3, 4] and the emergence of open-source projects [5, 6, 48] implementing decompilers for PYC binaries.

Of these recent endeavors, PyLingual [4] stands out as the most effective Python decompiler for modern Python versions (3.6-3.13). PyLingual has been able to keep up with Python's rapid development cycle by incorporating Natural Language Processing (NLP) models into the decompilation pipeline, which are prone to producing unpredictable decompilation failure modes. To restore trust in the system's outputs, PyLingual use a strict post-hoc decompilation accuracy test [7, 8] to verify the correctness of decompilation results. 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 January 2025, PyLingual has become the de facto framework for modern PYC decompilation, processing 61,382 PYC binaries and 3,465 user uploaded patches, including 253 verfiably accurate patches.

With the assistance of the PyLingual web service maintainers [4], our study received access to uploaded PYC binaries and user patches in full compliance with the legal and ethical guidelines established by the relevant Institutional Review Board (IRB). Leveraging this extensive dataset of PYC binaries and source-level patches that users submitted to the PyLingual web service, we study reverse engineers' responses to incomplete decompilation results. We specifically focus on factors that influence whether users will attempt to pursue verfiably accurate decompilation results in practice.

We conducted data analysis and user study in two stages: *(1)* we performed an observational study by analyzing PYC samples and user-submitted patches in the wild; *(2)* building on the observational study, we designed and conducted a controlled user study by selected 35 representative erroneous decompilations of varying difficulties to observe how students enrolled in a graduate-level system security class approached fixing these errors. In §4, we use these data points to illustrate that incentive structures and user objectives are stronger drivers of user-submitted patches than the magnitude of inaccuracy in the original decompilation result.

Based on the extensive dataset, our research derives valuable insights to improve PYC decompilation by addressing the following research questions: First, *what kind of programs are more accuracy-sensitive?* 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, music and animation software, and account management software. Second, *what are the major roadblocks preventing users from fixing decompilation errors?* 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 difficulty of patching is not a major barrier preventing users from attempting to fix erroneous decompilations. Third, *what elements contribute to successful patching of incomplete PYC decompilations?* 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.

Anonymous Submission to ACM CCS 2025, TBD
2024. ACM ISBN 978-x-xxxx-xxxx-x/YYYY/MM
https://doi.org/10.1145/nnnnnnn.nnnnnnn

Through our user study on PYC binaries and user-submitted patches, we investigate whether human-assisted Python decompilation can improve decompilation accuracy in practice. Furthermore, we examine efficient and effective methods for integrating human intelligence into the overall decompilation 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-facing interfaces. In summary, our study delivers the following contributions:

- **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.
- **Patching difficulty and effort estimation:** The study examines how human reverse engineers repair incomplete decompilations, showing that creating verifiably accurate corrections to inaccurate PYC decompilation results is a practical goal with a moderate amount of effort and knowledge.
- **Results-based analysis of reversers' motivations:** Through observational analysis, we determine that reversers' sensitivity to decompiler accuracy depends on the semantics of the target program, with patches disproprtionately being applied to programs where runnabilty and modifiability are relevant.

To benefit the research community, we publish the challenge dataset used for our in-class controlled study. [1]

2 Python Decompilation Challenges

2.1 Python Bytecode Decompilation

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

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. `Return` statements halt the execution of a function and return a value; however, if a `yield` statement is involved, the function can resume execution later. Exceptions can be raised at any point during execution, triggering a secondary control flow mechanism that activates exception handlers, executes cleanup code, and may exit the code object. 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.*

Python decompilation challenges. The most significant challenge in Python decompilation is 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 [9], 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.

By confirming that both the bytecode instructions and important metadata, like exception tables, match perfectly, PyLingual 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.

2.2 PyLingual Overview

PyLingual cleverly integrates NLP-models with programming language components to accurately decompile PYC binaries, seamlessly supporting evolving Python versions, including the most recent ones. The PyLingual framework comprises three components: *(1)* bytecode segmentation, *(2)* statement translation, and *(3)* control flow reconstruction.

Bytecode segmentation. Bytecode segmentation divides bytecode into smaller, independently digestible segments called "bytecode statements". These statements correspond to Python source code statements, facilitating accurate translation in later stages. Segmentation is crucial for preserving control flow and ensuring accurate mapping between bytecode and source code. PyLingual's BERT-based-model [10] predicts boundaries of bytecode statements deciding the beginning and end of a statement, facilitating statement translation and control flow reconstruction. While the segmentation step does not directly cause any errors, incorrect segmentation can negatively affect the statement translation and the control flow reconstruction step.

Statement translation. The statement translation module is responsible for converting segmented bytecodes into Python source code statements. Common errors resulting from statement translation are: format string junk, dictionary truncation, complex expression errors. Format string junk is a combination of both the segmentation step and translation step as the format value is confused for a string due to improper segmentation. Dictionary truncation is a result of large statements being fed into the model, this error can happen to any statement however it is most common with lists and dictionaries as those statements tend to be larger than others. Complex expression can overwhelm the model's limited complexity, additionally these cases are rare and not prevalent in training data.

Control flow reconstruction. After translating the individual statements, the control flow reconstruction module combines them

[1] https://anonymous.4open.science/r/pyc_user_study-7248

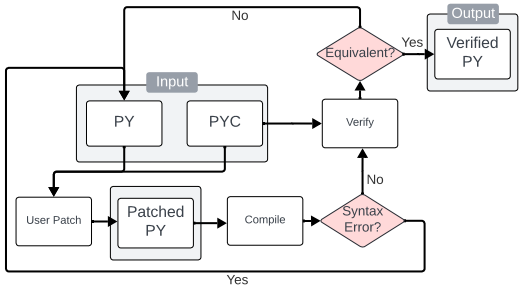


Figure 1: PyLingual patching process.

into a coherent Python program that attempts to accurately 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. Particularly, 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. For example, a `return` statement in the middle of a deeply nested series of `try-except` blocks will produce a copy of the exception handler cleanup code for every layer of `try-except`. When considering arbitrary combinations of `break`, `continue`, and `return` statements with arbitrary combinations of loops, exception handlers, conditional blocks, and context managers, it is easy to see how a manually curated set of heuristics fails to account for every case that comes up in practice.

3 PyLingual Web Service

In addition to offering an online decompilation service, PyLingual incorporates perfect decompilation to implement the efficient detection and localization of decompilation failures, which provides the foundation for a user-oriented feedback loop. PyLingual features a web-based IDE [11] that allows users to correct incomplete decompilation outputs by submitting user patches. The web interface enables users to upload PYC files and receive the decompiled source code, accompanied by an "equivalence report" that summarize the decompilation status of each code object along with the error types and their locations.

Figure 1 depicts the PyLingual patching process: *(1)* A user uploads a PYC binary (PYC) to the PyLingual web service, *(2)* PyLingual decompiles the uploaded PYC binary to generate the recovered source code. *(3)* The recovered source code is recompiled using the CPython compiler to produce a new PYC binary (PYC′), *(4)* The perfect decompilation evaluates the semantic equivalence between the input PYC binary (PYC) and the recompiled PYC binary (PYC′), *(5)* If compilation fails due to syntax errors in the recovered source or if equivalency verification detects discrepancies in any code object, the user can attempt to fix the errors using the web IDE, which highlights the error category and the specific error region. A user can repeat the processes of *(3)*, *(4)*, and *(5)* until PyLingual confirms the accurate decompilation. Verification of user patches incurs negligible computational overhead, as it only involves CPython compilation and static code equivalency verification.

While the current version of the user interface requires familiarity with Python bytecode to use effectively, perfect decompilation

```
168  try:
169      if hwid in r.text:
170          pass
171      print('[ERROR] HWID Not in database')
172      print(f'HWID: {hwid}')
173      time.sleep(5)
174      os._exit()
175  except:
176      print('[ERROR] Failed to connect to database')
177      time.sleep(5)
178      os._exit()
```

(a) Decompilation failure localization for the control flow difference.

```
168  try:
169      if hwid in r.text:
170          pass
171      else:
172          print('[ERROR] HWID Not in database')
173          print(f'HWID: {hwid}')
174          time.sleep(5)
175          os._exit()
176  except:
177      print('[ERROR] Failed to connect to database')
178      time.sleep(5)
179      os._exit()
```

(b) Result of a user patching the control flow difference.

Figure 2: PyLingual web interface.

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. Further, because verification does not involve the decompiler, the web interface can be generically extended to include other decompilers, which can directly benefit from the error localization and user patching layers.

When PyLingual produces incomplete decompilation output, and the verification module indicates that incorrect semantics were generated, the perfect decompilation 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.

PyLingual decompilation errors. As PyLingual decompiles code objects in uploaded PYC binaries, in case of incomplete decompilation, it fails with two different kinds of errors: *(1)* syntactic error, and *(2)* semantic error.

In many cases, recovered source code contains syntax errors that prevent it from being recompiled into a PYC binary for comparison with the original. These errors typically stem from issues such as incorrect indentation, missing colons, or mismatched parentheses. To apply perfect decompilation and generate an equivalency report, users must first resolve all syntax errors.

Fixing syntax errors may seem straightforward — for example, by deleting code statements or blocks with errors. However, such 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 observed instances where users effectively utilized AI-based code assistance tools for these. Perfect decompilation reports semantic

errors comparing the original and recompiled PYC binaries for any discrepancies. While both — original and recompiled binaries are valid Python programs, their static representations are different assuming both are compiled using the same CPython compiler.

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. Familiarity with NLP-based technologies and their limitations can also be beneficial. 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.

4 User Study

By analyzing uploaded binaries and user patches collected from the PyLingual web service, we conducted an observational study and designed in-class assignments for graduate-level students enrolled in a system security course. Specifically, we aimed to address the following research questions:

- **RQ1:** Would users want to fix incomplete decompilation outputs? If so, who seeks to achieve it, and for what types of Python binaries?
- **RQ2:** How difficult is it to fix incomplete decompilation?
- **RQ3:** What are the qualities and traits that define an effective patcher?
- **RQ4:** Can insights from this study be leveraged to improve automatic decompilers?

4.1 Python Decompilation In The Wild

With the assistance of the PyLingual web service maintainers, our study received access to uploaded PYC binaries and user patches. Since November 2023, pylingual.io has received 61,382 user-submitted PYC binaries for decompilation from 14,898 IP addresses, of which 33,642 PYC (≈ 55%) decompiled perfectly.

We focus on the remaining 27,740 PYC binaries that *did not* accurately decompile, and the subsequent efforts of users to repair the decompilation results. To this end, users have submitted 3,465 patches across 1,747 files and 1,248 IP addresses. These efforts have resulted in 253 additional files with semantically verified decompilations.

In Figure 3, it is immediately apparent that only a small fraction of incomplete decompilations receive patches from users, and only a small fraction of users engages 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 30% only including one bytecode error, and over 50% including just two or fewer

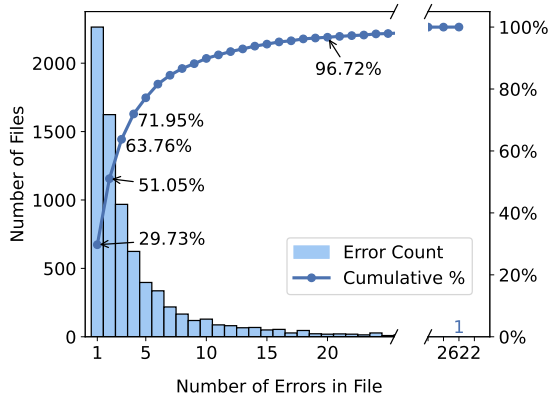


Figure 3: The number of *bytecode errors* in PYC files decompiled by PyLingual, including only files with one or more errors.

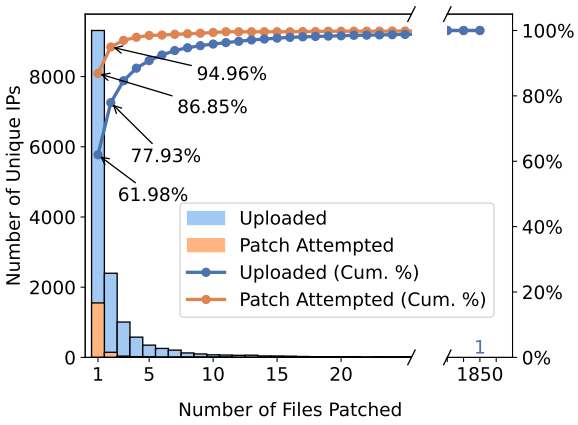


Figure 4: The number of *uploaded* and *patched* files by unique IPs.

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 (≈ 70%) of incomplete decompilations contain syntax errors, which prevents bytecode-level analysis.

In Figure 4, The number of uploaded and patched files heavily concentrated in a single instance, 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 a unique IP address for a user connect from different time and place, 61.98 % of file uploaders uploaded files only once and 86.85 % 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 the most users come to PyLingual to solely reverse single PYC file to Python file, then depart PyLingual after applying their patch.

Table 1: At $p \approx 3.6 \times 10^{-9}$ and $r \approx 0.176$, users whose first patch was successful were more likely to try patching additional files.

	First Patch Failed	First Patch Success
Only Patched Once	1,024	31
Patched Multiple Times	133	21

Moving onto the initial analysis, on Table 1, a χ^2 test [49] for independence indicates that the success of a user's first patched file positively correlates with that user returning to patch additional files weakly 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 $r \approx 0.176$ indicates a weak positive correlation at $p \approx 3.6 \times 10^{-9}$, which is well beyond the conventional threshold for stastistical signifcance of $p < 0.05$. In natural language, those users who experience an "early win" when patching files generally more likely to be among users who attempt to patch multiple files. 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 1-1 with human users. Although this assumption is unlikely to hold in general, we still find that it leads to a useful insight.

Given the low observed rates of patch attempts and successes, alongside the correlation between early successes and continued attempts, we next aim to 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.

4.2 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. The legal and ethical considerations for this study are discussed in §8. To protect the anonymity of the 24 student participants, the only data collected in the study is anonymized patch histories. The form of the data for analysis is the same as in the observational study, but here we control the challenge binaries, which were randomly sampled from unpatched files in the observational study, and we operate with a known population.

The assignment summary. To emulate the conditions of real-world reverse engineers, for the assignment in the study we chose 35 representative PYC binaries with decompilation errors. These files were manually assigned a difficulty category of easy, medium, or hard based on a simple visual inspection of the decompilation result compared against the original PYC bytecode. Students received points for the 7 hardest files that they successfully patched, with easy PYCs being worth 20 points, mediums worth 30 points, and hards worth 40 points. Full credit for the assignment was set at 140 points (patching 7 easy PYCs), with any additional points counting as extra credit. In total, there were 15 easy challenges, 14 medium challenges, and 7 hard challenges, containing a variety of syntactic and semantic decompilation errors. Interested readers can visit the repository that publicize the challenge dataset to further explore the details of the controlled user study.

The incentive structure is the most significant difference between the classroom participants and the real-world users. The students approached PYC from the wild using the same publicly available tools as real Python reversers, but were explicitly assigned accurate decompilation as a goal instead of regarding it as an intermediate step in a wider reversing process, similar to previous studies in this space [7]. Students seeking to minimize their workload while maximizing their score are incentivized to "scout" the challenge PYCs , selectively choosing the easiest ones in each difficulty category 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. 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.

Patch difficulty and effort estimation. Figure 5 shows that students were consistently able to accurately patch the challenge PYCs, with the median student successfully patching 9 challenges, just above what is required for full credit. Several students completed the requirements for full credit before moving on to harder challenges for extra credit, with the most prolific student patcher producing 22 successful patches out of the 35 available challenges. This trend is

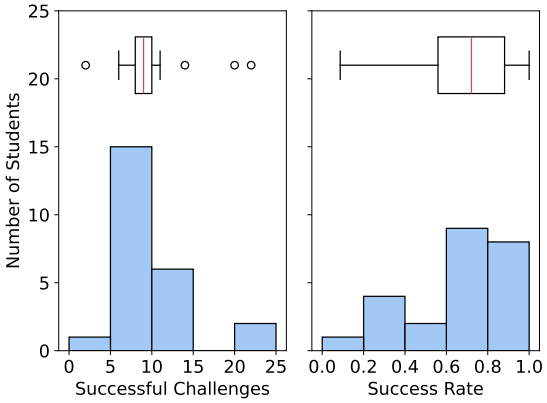


Figure 5: A histogram and box plot of the number of successful challenges and individual success rate of students in the classroom study. $Success\ Rate = \frac{\#\ of\ Successful\ Challenges}{\#\ of\ Attempted\ Challenges}$

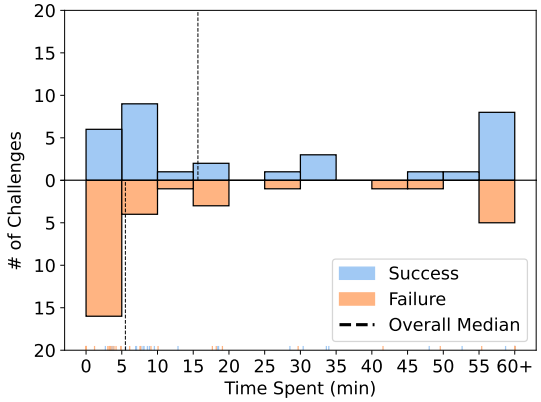


Figure 6: Distribution of the median time taken on each challenge during the classroom study. Median times were taken separately for successful and unsuccessful patches.

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 students are generally high, with a median success rate of 72 %, and three students achieving a 100 % success rate, eventually fully solving every challenge where they attempted a patch.

In the classroom study, the easiest challenge had an 86% engagement rate and a 100% success rate among students. This challenge had a single incorrect format string, where the `FORMAT_VALUE` argument was erroneously included in the format string at the source code level. The "FORMAT STRING JUNK" error results in recognizable artifacts at both the source code and bytecode level, and is trivial to patch by simply removing the additional character from the format string in the source code. In Figure 7, we find that among the `PYC`s in the wild that have just a single bytecode error, FORMAT STRING JUNK is the most common category. Despite its

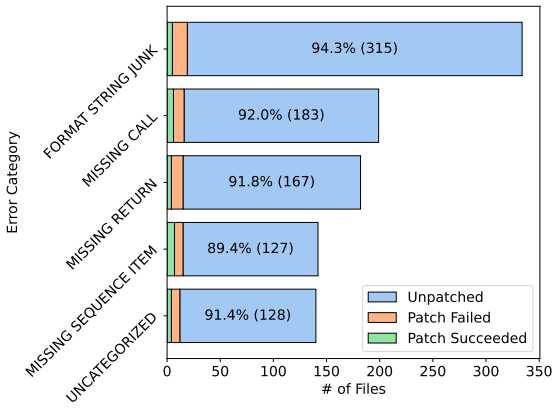


Figure 7: Patching outcomes for files with a single bytecode error by Internet users (top 5 error categories shown).

prevalence, recognizability, and simplicity to patch, over 94% of these instances remain unpatched, strongly indicating that ease of patching alone is not enough to encourage users in the wild to fix nearly correct decompilation results.

Table 2: Results of logistic regression of patch attempt rate on bytecode edit distance and presence of syntax errors in `PYC` binaries from the classroom study and the wild.

	Classroom		The Wild	
	Odds Ratio	p-value	Odds Ratio	p-value
Intercept	4.917	< 0.001	0.082	< 0.001
Bytecode Edit Distance	0.989	0.007	$1 + 6 \times 10^{-5}$	< 0.001
Syntax Error	0.332	< 0.001	1.317	< 0.001

To further illustrate the impact of the differing incentive structures between the classroom study and patchers in the wild, Table 2 summarizes the results of a linear regression analysis of the effect of two key patching difficulty indicators on the likelihood that a decompilation result will be patched by a user. While the baseline likelihood of patching was significantly higher in the classroom study than in the wild – as indicated by the much higher odds ratio of the intercept for the classroom regression model – we can compare the relative effects of the difficulty indicators between both environments. Bytecode edit distance quantifies the magnitude of the discrepancy between the decompilation result and the original `PYC` bytecode. In both the classroom study and the observational study, we find that *the bytecode edit distance* has a negligible impact on patch attempt rates – although it has a significant p-value, its odds ratio is very close to 1, indicating a very small effect from each bytecode error on the likelihood of patching attempts. Importantly, syntax errors prevent further accuracy analysis, which can obscure additional syntax and semantic errors behind them. In the classroom study, we find that students are strongly repelled by *syntax errors*, with the presence of a syntax error having an odds ratio of 0.332, indicating that files with syntax errors are significantly less likely to be patched than those without syntax errors, accounting

for the relative representation of syntax errors in the challenges. From the students' perspective, it is easier to simply move on and try a different challenge than to fix the syntax error to reveal further semantic errors underneath. However, in the wild, we observe the *opposite* effect: files containing syntax errors were significantly *more likely* to receive a patch than those without syntax errors. We believe that this reversal is caused by differences in objectives between student patchers and real-world reversers. In the classroom study, students have a quota of successful patches to fulfill and are not tied to any particular file within the assignment, rewarding students for engaging with easily appraisable decompilation errors. In the wild, reversers aim to understand a few target `PYC`s, and are therefore incentivized to repair syntax errors, which allows the accuracy analysis to proceed. Even when these patchers do not go on to fully repair the decompilation result, we conjecture that partial decompilation accuracy assessments are still useful to reversers in practice.

4.3 Program-Dependent Accuracy Sensitivity

Table 3: Logistic regression of patch attempt rate with clusters

	N	Odds Ratio	p-value
Intercept	-	0.070	< 0.001
Bytecode Edit Distance	-	$1 + 5 \times 10^{-5}$	< 0.001
Syntax Error	2825	1.313	< 0.001
Cluster			
1	1331	1.183	0.110
2	693	0.819	0.223
3	2085	1.704	< 0.001
4	1820	1.605	< 0.001
5	101	1.035	0.927
6	1845	0.554	< 0.001
7	1602	1.384	< 0.001
8	1300	1.245	0.036
9	788	1.131	0.364
10	1374	1.244	0.031
11	838	1.097	0.490
12	289	1.352	0.131
13	1271	1.676	< 0.001
14	1151	0.950	0.671
15	900	0.728	0.036
16	970	1.540	< 0.001
17	546	1.137	0.424
18	953	0.982	0.888
19	668	0.529	0.002
20	410	0.521	0.011
21	207	0.135	0.003
22	1004	1.226	0.082
23	442	2.850	< 0.001
24	291	1.656	0.007
25	609	0.884	0.460
26	642	0.340	< 0.001
27	156	1.422	0.183

We analyze the correlation between the contents of uploaded `PYC` files in the wild and Internet users' patching behavior. For this, we used the `jina-embeddings-v2-base-code` [12] 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 27 semantic clusters of files. We then labelled each cluster by randomly sampling 20 files for manual investigation and summarizing the sample by analyzing the structure, function, and context of each file. The full list of clusters and related program demographic statistics is available in the appendix in Table 4. 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 [13], variable and string scramblers like Oxyry [14], 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 [15].

Using logistic regression analysis on files that were inaccurately decompiled, we find in Table 3 that even after accounting for the influence of syntax errors and bytecode edit distance on user patching behavior, some semantic clusters have significant positive and negative influences on patching likelihoods. The semantic clusters were encoded in the model using effect coding, so their odds ratios can be interpreted as the ratio change in the odds of a file being patched compared to the unweighted grand mean of all clusters. Odds ratios above 1.0 indicate an increased chance of files being patched when the associated variable is high — a file belongs to a cluster, has a syntax error, or has many bytecode errors — while odds ratios below 1.0 indicate a decreased chance of patching with an increase in the variable. The odds ratio of the intercept represents the odds of a file being patched that has no bytecode or syntax errors, plus the mean effect of all clusters. The significantly *over-patched* clusters (at $p < 0.001$), in order of decreasing effect size, are: cluster 23, "multimedia software extensions"; cluster 3, "malware"; cluster 13, "game bots and desktop automation"; cluster 4, "file and account management"; cluster 16, "web servers and async task management"; and cluster 7, "database interfaces and excel scrapers".

For these over-patched clusters, there appears to be a focus on runnability, reusability, and modifiability. Cluster 23 disproportionately receives the most user-submitted patches, which we believe to be due to a focus on runnability; the audio editing and animation software in cluster 23 is often included in licensed extension packages, which may attract reversers who wish to redistribute copycat products. Cluster 3 contains a wide variety of unobfuscated malware, which may attract patches from reversers aiming to receive automatic verification of the correctness of the source code to more reliably share their findings, or may seek partial verification of key components to identify remediation and prevention procedures.

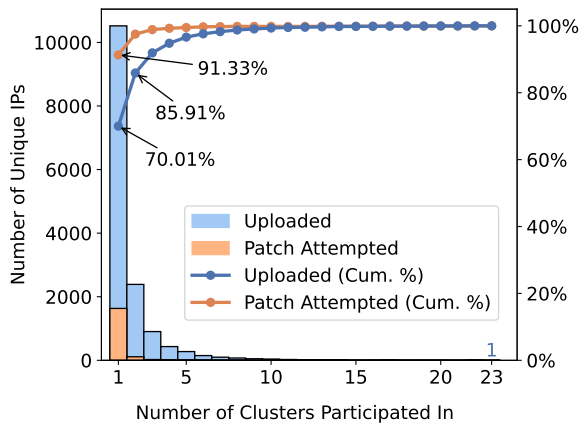
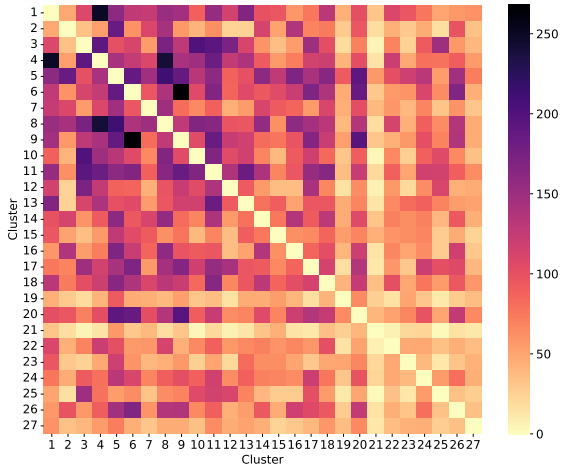
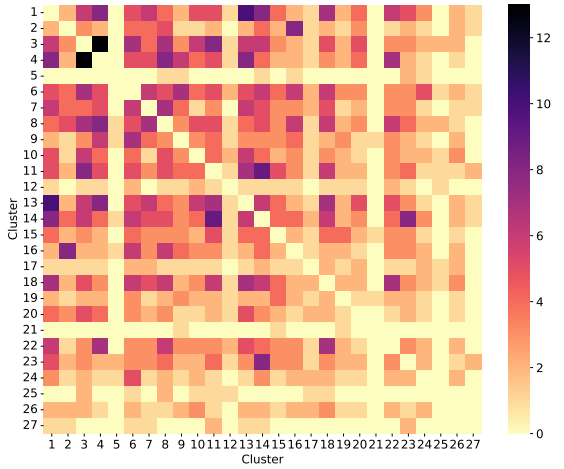


Figure 8: Semantic cluster participation by unique IP addresses.



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



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

Figure 9: Heatmaps representing user acitivities across cluster pairs.

The remaining over-patched clusters all focus on core business application logic, which serves as a rich resource for vulnerability researchers targeting critical interfaces in production code.

The significantly *under-patched* clusters (at $p < 0.001$), again in order of decreasing effect size, are: cluster 26, "socket and HTTP libraries"; and cluster 6, "Python standard library code". Both of these clusters contain code that is immediately recognizable for what it is, contain neat docstrings, and are typically openly available on github, locatable by searching for the internal filename. The explanation for why these files do not receive much interest in refining their decompilation results is twofold: *(1)* these files are often included in PyInstaller [15] bundles which are uploaded in bulk by project-scale reversers; *(2)* accurate decompilation is not necessary when the original source code is accessible online, so even when accurate code is needed, patching the decompilation result is more labor-intensive than simply finding the code on GitHub.

The strong influence of program semantics on the accuracy sensitivity of reverse engineers demonstrates that different reversers have different incentive structures depending on their individual reasons for decompiling a `PYC` file. User and file-dependent accuracy needs also presents an interesting decision point for decompiler designers making a tradeoff between computational efficiency and decompilation accuracy, where a user may not know what degree of accuracy they need before observing an initial coarse-grained decompilation result.

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 8, most users—70% of uploaders and 91% 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 9a 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 find three pairs of frequently co-occurring clusters, as well as two clusters that are relatively isolated.

Cluster 1 (menu GUIs developed with tools like tkinter and PyQT) and cluster 4 (file and account management software) have the greatest magnitude of shared user activity. This relationship implies that reversers frequently decompile account management projects with graphical interfaces spread across multiple files. GUI menus rarely exist in isolation in real software, so cluster 1 experiences moderate co-occurrences with a wide range of other clusters.

Cluster 4 is displays high co-occurrence with cluster 8 (encryption and authentication), sharing 230 uploaders. Account management is complemented by authentication, so such file pairs are likely to appear within the same project. Further, application login portals are an important point of investigation for vulnerability researchers, representing an network-facing and security-relevant

interface. Interestingly, because the authentication in cluster 8 tends to focus on internet-facing applications, cluster 8 only has moderate co-occurrence with cluster 1, desipte both clusters 1 and 8 having high co-occurrence with cluster 4.

Cluster 6 (Python internal standard libraries) and cluster 9 (module dependency management) also exhibit a strong overlap in uploader activity within cluster pairs. Cluster 6 deals with core functionalities of the Python standard library such as os and logging, whereas cluster 9 handles with managing Python standard library modules mostly utilizing frameworks like PyInstaller to bundle Python applications. Importantly, files in both clusters are nearly always included when a script is bundled as a standalone executable with packaging tools like PyInstaller. The high co-occurrence here is explained by reversers automatically uploading every PYC from extracted PyInstaller executables while searching for critical business logic.

Clusters that are dominated by small, focused groups of users engaging in large-scale reversing efforts showcase low co-occurrences across a wide range of other clusters. Cluster 19 (game code and Sims mods) is largely dominated by the large scale reversing of a popular modification for the Sims 4. Cluster 21 (simulation boilerplate) is dominated by a dedicated group of reversers aiming to decompile production-grade computational modelling software.

Figure 9b is a similar heatmap that showcases co-occurrences of users *patching* files across cluster pairs. There are significantly fewer patches than uploads, so this heatmap is notably more sparse than Figure 9a. Here, cluster 3 (malware) and cluster 4 (account management) stand out as security-relevant categories that are important for both offensive and defensive reverse engineers. Recall that both of these clusters independently attract disproportionately many patches; this patcher co-occurrence indicates that they are attracting patches from the same kinds of security-oriented reversers.

In addition to the clusters that were already isolated when considering the uploaders, clusters 5, 12, and 25 also share few patchers with other clusters; this apparent isolation is largely due to the small number of patchers that participate in those clusters, which present very few opportunities for co-occurence with other clusters.

5 Case Studies

Through manual review of patch histories in the classroom study, we find that recognizing how source-level patterns are reflected in the PYC bytecode is among the most critical skill for accurately localizing decompilation errors and crafting successful patches. Through illustrative case studies, we demonstrate how decompilation errors that are visible in the explicit bytecode implementation of implicit source code features can mislead novice reversers.

Out-of-order exception handling. Figure 10 shows a `try-except` structure that was decompiled verbatim in the same order as it appears in the bytecode; in Python 3.12, the contents of exception handler blocks were moved to the end of each code object to reduce jumping in the interpreter when there is no exception. This decompilation error results in incorrect control flow with unreachable code. The `if` statement belongs in the `except` block, which is only clear after careful observation of the original bytecode, including recognizing the exception setup and cleanup idioms,

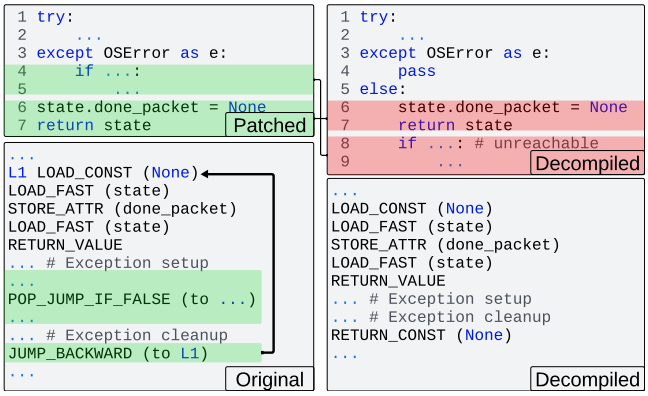


Figure 10: The patcher had to relocate code and delete the else.

which are implicit at the source level. Further, after the resolution of the `except` block in the original bytecode, control jumps to `sate.done_packet = None`. To accurately capture this control flow in the source code, the `else` block must be removed, placing the subsequent statments entirely outside of the `try-except` structure.

Unsuccessful student patchers often attempted to accomodate the `else` block rather than trying to change or remove it. We attribute their trust in the decompiler to a lack of deep understanding of the bytecode. Meanwhile, successful patchers did not have any reservations for preserving the else block, they would first attempt to change the else into a finally or outright delete it. While both unsuccessful and successful patchers were able to properly relocate the unreachable code, the distinguishing factor between the two was their willingness to modify the control flow structures produced by the decompiler. Successful patchers were more willing to make structural modifications, which enabled them to resolve the deceptive mistake made by the decompiler.

The key factor that contributed to the difficulty of this challenge was the depth of implict control flow understanding required to visualize the correct source code. Critically, it is not intuitive for a novice Python reverser that a backwards jump could indicate the *absence* of an `else` block in a `try-except` structure. Identifying this connection between the bytecode and source code requires either sufficient examples for the reverser to recognize the pattern, or a deep understanding of Python's bytecode generation process. Better tooling support could make these exception handling constructs more immediately visible, reducing the need for human reversers to be able to quickly recognize bytecode-only code generation idioms.

Improperly segmented conditional. Figure 11 shows a compound `if` statement that was incorrectly segmented, preventing the correct association between the conditional jumps and the else block. In the original bytecode, when jump associated with `not var.game_play_player_miss` is taken, the function immediately returns. However, in the decompiled source code, that condition is part of a larger `if` statement alongside other conditional jumps that are associated with an `else` block later in the function. Without splitting the conditional jumps into separate `if` statements, the original control flow is impossible to reproduce.

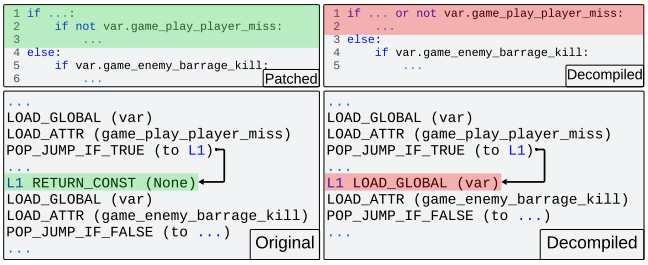


Figure 11: Though the jump target in the decompiled source is correct, it wrongly executes the else block instead of returning.

The true location of the decompilation failure is obscured by the fact that the final condition's jump target has the same offset in the incorrectly decompiled code as it does in the original. Unsuccessful patchers failed to correctly localize the issue, focusing their patching efforts on reformulating the conditions of the `if` statement or modifying code near the boundary between the `if` body and the `else` block. Successful patchers split the conditions of the compound `if` statement into multiple `if` statements, enabling them to correctly associate the `else` block and produce successful patches. The key distinction between successful and unsuccessful patchers was their ability to correctly identify errors that were not highlighted by the simple instruction diff mechanism.

Because the contents at the jump target changed rather than the target itself, from a novice's perspective, this case is indistinguishable from that of the code at the jump target location being incorrectly translated. Similar to the previous case studies, unsuccessful patchers relied too heavily on the code structure provided by the decompiler, and consequently failed to escape the trap that the decompiler had fallen into. Further, the fact that the jump target offsets were numerically equal despite being semantically misaligned caused the simple diff-based error localization tool to mislead patchers by highlighting a secondary effect of the true decompilation failure. Additional tooling support to create and visualize control flow graph differences between the original and candidate `PYCs` could better enable novice reversers to identify, localize, and address subtle control flow differences.

6 Related Works

6.1 Reversing User Study

Burk et al. [7] proposed to take advantage of the perfect decompilation to assess the users' approach to reversing the binaries. Mantovani et al. [16] 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. Votipka et al. [17] interviewed sixteen reverse engineers as they reenacted a recent reverse engineering activity. These interviews were also semi-structured, and the authors used them to extract common themes.

A few studies have investigated the usability of RE tools. For instance, researchers have looked at improving the usability of decompilers [18, 19] showing that better variable naming and a reduced number of `GOTOs` affected positively the readability of the pseudocode.

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 [18–23, 50, 51]. 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. [20] 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 [24], and *(2) structural analysis* to identify code blocks and restore control dependencies among them [18–21]. Structural analysis has more impact on the performance and usability of a decompiler, so it has been the primary focus of recent research [18, 19, 21].

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 requires strict syntax compliance and semantic accuracy, posing new challenges to natural language translation and complicating the generation of trustworthy results.

Wartell et al. [25] proposed one of the first ML-assisted binary analyses, using a predication by partial matching model to differentiate code from data in x86 binaries. Later, Shin et al. [26] built a multi-layer RNN network that consumes one byte at a time to predict a byte sequence that identifies function boundaries. Katz et al. [27] 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 programming language-specific features such as function and statement boundaries, number and type of instruction operands, *etc.* CODA [28] 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 [29] 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.

6.3 HDL Decompilers

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 [30–32]. 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.,* PYCfiles for Python and CIL files for .NET framework), reverse engineers often depend on incomplete or inaccurate solutions for analyzing malicious binaries in this intermediate form.

Python decompilers. The two most popular decompilers for Python binaries are uncompyle6 [5] and decompyle3 [6]. 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 [52] 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 [48] 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.

Decompilers for other HDLs. Soot [53], 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 [54, 55] 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 [56, 57]. 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

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 [33] 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 [34, 35] and large-context architectures [36, 37].

8 Conclusion

In this paper, we presented a comprehensive study on human-assisted decompilation of `PYC` binaries, leveraging a unique dataset of real-world `PYC` binaries and user-submitted patches. To explore how reverse engineers address incomplete decompilation results, we conducted both observational and controlled user studies, identified key factors influencing the success of human-assisted decompilation, and provided actionable recommendations for enhancing decompilation accuracy.

Our findings highlight that certain types of programs, such as unobfuscated malware and software with high user interaction, are more likely to motivate users to pursue accurate decompilation results. Additionally, we observed that the difficulty of patching decompilation errors is not a significant barrier for users, as even minimal bytecode differences can be effectively resolved by participants with appropriate incentives and objectives. By analyzing the patching behaviors and strategies of successful reverse engineers, we derived several insights that can enhance the design and functionality of decompilation tools. Our study underscores the importance of integrating human intelligence into the decompilation workflow to achieve higher accuracy and reliability in decompilation outputs.

Lastly, our research contributes to the field of decompilation by demonstrating the practical benefits of human-assisted decompilation and providing a robust knowledge base for future improvements in automatic decompilation systems. We hope that our findings will inspire further research and development in this area, ultimately leading to more effective and user-friendly decompilation tools.

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 user privacy perspective using their uploads for the 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 source out to various purposes. 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 that explicitly requests users' consent to share their `PYC` binaries and user patches. Furthermore, PyLingual offers users the option to delete their files and patches at any time.

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. 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.

Open Science Policy Compliance

In accordance with the Open Science Policy, we did our best to to ensure our research is accessible. While we cannot fully release the uploaded `PYC` files and user-submitted patches due to ethical considerations, we share the `PYC` binaries used in the classroom patching study, along with detailed descriptions of each challenge and guidance on how to solve them.

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

[4] 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.

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

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

[7] 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.

[8] Eric Schulte et al. "Evolving Exact Decompilation". In: *Workshop on Binary Analysis Research (BAR)*. Workshop on Binary Analysis Research (BAR). 18 2018. ISBN: 1891562509. DOI: 10.14722/bar.2018.23008.

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

[10] 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.

[11] *Monaco Editor*. [Online; accessed 2025-01-19]. URL: https://microsoft.github.io/monaco-editor/.

[12] 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.

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

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

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

[16] 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.

[17] 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.

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

[19] 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.

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

[21] 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.

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

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

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

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

[26] Eui Chul Richard Shin, Dawn Song, and Reza 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. URL: https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/shin.

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

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

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

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

[31] *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.

[32] *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/.

[33] 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.

[34] 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.

[35] 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.

[36] 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.

[37] 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.

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

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

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

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

[42] Josh Grunzweig. *Unit 42 Technical Analysis: Seaduke*. https://unit42.paloaltonetworks.com/unit-42-technical-analysis-seaduke/.

[43] jinye. *Necro Frequent Upgrades, New Version Begins Using PyInstaller and DGA*. https://blog.netlab.360.com/not-really-new-pyhton-ddos-bot-n3cr0m0rph-necromorph/.

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

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

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

[47] 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/.

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

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

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

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

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

[53] 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.

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

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

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

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

A Dataset Descriptions

Table 4 provides a comprehensive summary of all 27 clusters, detailing their descriptions and activity metrics. Each row corresponds to a distinct cluster with a description of its associated files and activities. The 'Total' column quantifies the total number of files in each cluster, derived as the sum of four patch-related categories: No Patch Needed, Patch Success, Patch Failed, and No Patch Attempt. Additionally, the table includes metrics on uploader and patcher activity. The 'Upload IPs' column indicates the number of unique IP addresses that uploaded files within the cluster, while the 'Patch IPs' column shows the number of unique IP addresses associated with patching activities.

Table 4: Summary of clusters

ID	Description	Total	No Patch Needed	Patch Success	Patch Failed	No Patch Attempt	Upload IPs	Patch IPs
1	Generic menu GUIs	2966	1635	6	103	1222	1166	80
2	Boilerplate CRUD methods	2721	2028	22	17	654	603	23
3	Malware	2463	378	4	241	1840	1655	196
4	File and account management	2461	641	6	192	1622	1490	152
5	Imports, environment setup	2345	2244	0	7	94	1044	6
6	Python standard library code, other general Python code	2335	490	13	61	1771	945	42
7	Database interfaces; web and Excel scrapers	2274	672	5	149	1448	1175	114
8	Automatic purchases, encryption, and authentication	2201	901	12	99	1189	1193	87
9	Module dependency and file path management utilities	2155	1367	11	49	728	1094	30
10	Social media bots	2031	657	9	111	1254	1195	92
11	External process management and program configuration	2016	1178	4	58	776	1380	53
12	PyArmor obfuscated files	1984	1695	0	27	262	1221	24
13	Game bots and desktop automation	1889	618	7	135	1129	1275	114
14	Computing management servers and byte-level memory parsing	1841	690	5	72	1074	852	68
15	Mathematics and ML files	1813	913	8	38	854	821	37
16	Web servers and async task managers	1799	829	64	35	871	537	25
17	Encryption and decryption	1782	1236	6	36	504	1158	34
18	File extraction and transfer	1684	731	6	58	889	1003	49
19	Game code, mostly Sims 4 mods	1564	896	1	24	643	418	19
20	Validation and type definitions	1544	1134	4	11	395	746	11
21	Simulation and computational model management boilerplate code	1530	1323	1	1	205	187	1
22	Video and audio streaming, downloading, and manipulation	1477	473	6	79	919	733	59
23	Multimedia software and its extensions, largely GUI code	1430	988	17	59	366	436	37
24	Toy examples and short standalone functions	1312	1021	9	22	260	975	24
25	Non-PyArmor obfuscated files, long unicode strings	1223	614	0	38	571	770	33
26	Socket and HTTP libraries, including some standard library code	1216	574	1	15	626	504	13
27	Configuration loaders and managers	621	465	0	15	141	550	13

Table 5: Class challenges

Hash	Version	Difficulty	Codeobj Count	Has Syntax Error	Error Categories
f18298da6b55ea3e...	3.6	Easy	6	No	PREMATURE EXCEPTION EXIT
00bcb3cb3fd5e8c2...	3.6	Medium	16	No	MISSING ELSE BLOCK
7d703b27e6ce7341...	3.7	Easy	2	No	MISSING FUNCTION ARGUMENT
cd384ab4b1f58ada...	3.7	Easy	5	No	INSERTED FUNCTION CALL WRONG NUMBER OF FUNCTION ARGUMENTS MISSING CALL
66dbc0fb1eaadd63...	3.7	Easy	26	No	MISSING IF BLOCK WRONG VARIABLE NAME WRONG CONSTANT MISSING ASSERT
2733c1d34d516d29...	3.7	Medium	32	No	WRONG CONSTANT WRONG EXPRESSION
14651d0f77743245...	3.7	Hard	22	No	MISSING ELSE BLOCK
92275159e9b89905...	3.8	Easy	26	No	MISSING CALL
c7dd5cf286039fb1...	3.8	Easy	30	No	INCORRECT EXCEPT BOUNDARY FORMAT STRING JUNK
3019c2c8b606f600...	3.8	Medium	25	Yes	-
76b36b7dedee7544...	3.8	Medium	32	Yes	-
81ebd7799a06bcc3...	3.8	Hard	66	Yes	-
26d0ed1f9f8de33a...	3.9	Easy	10	No	FORMAT STRING JUNK
092e0eba1d253fbb...	3.9	Easy	3	No	PREMATURE EXCEPTION EXIT
98f1c66d82270adc...	3.9	Hard	65	No	DELETED GENEXPR WRONG CONSTANT FORMAT STRING JUNK
ab65a335220b09c3...	3.9	Hard	8	Yes	-
9f2e243418eecfd5...	3.10	Easy	3	No	PREMATURE EXCEPTION EXIT
8e53c446a4853db4...	3.10	Easy	26	No	PREMATURE WITH BLOCK EXIT PREMATURE EXCEPTION EXIT WRONG EXPRESSION
fbcb33891ad43605...	3.10	Easy	21	No	PREMATURE RETURN FORMAT STRING JUNK
362bebf003368b39...	3.10	Easy	8	No	PREMATURE EXCEPTION EXIT MISSING RETURN WRONG CONSTANT WRONG FORMAT SPECIFIER INSERTED SYMBOL LOAD
04eca6aba3902b67...	3.10	Easy	19	Yes	-
24de5b16b80cd22e...	3.10	Medium	11	No	MISSING RETURN FORMAT STRING JUNK
7c16eab6f646f643...	3.11	Easy	24	Yes	-
08ee38583abdc6fc...	3.11	Medium	5	No	WRONG VARIABLE NAME MISSING EXPRESSION ITEMS INSERTED CALL MISSING SYMBOL LOAD MISSING SEQUENCE ITEMS INSERTED SYMBOL LOAD
dc211048df279579...	3.11	Medium	6	Yes	-
049c96249673f534...	3.11	Medium	25	No	MISSING UPDATE ASSIGNMENT PREMATURE RETURN INSERTED CHAINED COMPARISON EXTRA SEQUENCE NESTING PLUS STRING AS FORMAT STRING
7272a4e7d6e48e9b...	3.11	Medium	11	Yes	-
5a66eda21411eb62...	3.11	Hard	5	Yes	-
f5bcbf46937880b4...	3.11	Hard	116	Yes	-
ca8838668c9f4813...	3.12	Easy	4	No	PREMATURE EXCEPTION EXIT WRONG LIST CONSTRUCTION
a66a42988b4067b6...	3.12	Medium	2	No	MISSING RETURN PREMATURE RETURN
da3f02867ab7c586...	3.12	Medium	8	Yes	-
50dd1ae3f8195d19...	3.12	Medium	118	Yes	-
36d09041a07a331c...	3.12	Medium	7	Yes	-
c1936dfb9d39c9a5...	3.12	Hard	18	No	FORMAT STRING JUNK