PYFET: Forensically Equivalent Transformation for Python Binary Decompilation

Ali Ahad*, Chijung Jung*, Ammar Askar†, Doowon Kim‡, Taesoo Kim†, and Yonghwi Kwon*
*Department of Computer Science, University of Virginia, Charlottesville, VA, USA
†School of Computer Science, Georgia Institute of Technology, Atlanta, GA, USA
‡Department of Electrical Engineering and Computer Science, University of Tennessee, Knoxville, TN, USA
*{aliahad, cj5kd, yongkwon}@virginia.edu †{aaskar, taesoo}@gatech.ac.kr ‡doowon@utk.edu

***Abstract*—Decompilation is a crucial capability in forensic analysis, facilitating analysis of unknown binaries. The recent rise of Python malware has brought attention to Python decompilers that aim to obtain source code representation from a Python binary. However, Python decompilers fail to handle various binaries, limiting their capabilities in forensic analysis.**

This paper proposes a novel solution that transforms a decompilation error-inducing Python binary into a decompilable binary. Our key intuition is that we can resolve the decompilation errors by transforming error-inducing code blocks in the input binary into another form. The core of our approach is the concept of Forensically Equivalent Transformation (FET) which allows non-semantic preserving transformation in the context of forensic analysis. We carefully define the FETs to minimize their undesirable consequences while fixing various error-inducing instructions that are difficult to solve when preserving the exact semantics. We evaluate the prototype of our approach with 17,117 real-world Python malware samples causing decompilation errors in five popular decompilers. It successfully identifies and fixes 77,022 errors. Our approach also handles anti-analysis techniques, including opcode remapping, and helps migrate Python 3.9 binaries to 3.8 binaries.

1. Introduction

Malware such as viruses and ransomware is a key weapon leveraged by cyber attackers. Reverse-engineering malware is a crucial capability for investigating the details of cyberattacks. In particular, a decompiler is a highly desirable reverse-engineering tool that can translate a malware binary into human-readable source code representation. For many years, various reverse-engineering tools such as decompilers have been proposed and advanced [8–10, 21, 24, 25, 40, 42] to target C/C++ binary programs as these languages have been traditionally used to write the majority of malware.

We have recently seen that a large amount of malware has been written in emerging languages [29], particularly in Python [7]. Python is easy to learn and has significant community support, making it an attractive language for malware authors. Python malware is typically distributed as a compiled binary containing Python bytecode instructions that are challenging to analyze.

The community has responded to this problem with Python decompilers [16, 47]. Given a Python binary, decompilers translate bytecode instructions to the high-level representation (i.e., source code). Unfortunately, they have limitations and have difficulty handling various Python binaries. Specifically, when a decompiler encounters particular sequences of instructions, it fails the decompilation process. It either stops the decompilation process or generates incorrect outputs (i.e., incorrectly-decompiled code), thwarting and misleading consequent analyses. We have also seen Python binaries with customized opcode definitions that are incompatible with the standard Python environment and decompilers [5, 26], thwarting the decompilation process.

To overcome these challenges, a straightforward method is to fix the Python decompilers. However, multiple Python decompilers used in the wild are designed, implemented, and maintained by different groups. They are also written in different programming languages: Python [16, 46–48] and C++ [17]. Each decompiler has its own strengths (e.g., `Decompyle3` outperforms `Uncompyle6` in 3.7-8 Python binaries while `Decompyle3` does not support other Python versions), making it difficult to choose a decompiler to fix as there is no clear winner amongst them. Moreover, fixing a decompiler is time-consuming and error-prone. Based on our experiments on fixing three decompilers (Appendix 5.1), on average, it took an author *two days* to fix a single bug in a decompiler, while *2 of 3 fixes introduced new bugs*.

This paper proposes a program transformation-based solution to the decompilation failure problem. Our approach is based on the observation that failure-inducing instructions/structures in Python binaries can be transformed into other forms that decompilers can successfully handle. A challenge in such transformation is that we may not find solutions for various decompilation errors if we only allow semantically equivalent transformations. To this end, we define a concept of *Forensically Equivalent Transformation (FET)*, which relaxes the semantic preserving restriction as long as the transformation preserves sufficient semantics for forensic analysis (Details can be found in Section 3). We evaluate our approach on 17,117 Python malware binaries (that cannot be decompiled by five decompilers [16, 17, 46–48]) in the wild, showing that our technique resolves all the decompilation failures. Our contributions are as follows:

- We develop a system, PYFET, that helps Python decompilers by transforming Python bytecode binaries with decompilation failures to decompilable binaries.

- We propose the concept of Forensically Equivalent Transformation (FET) that relaxes the semantic preserving constraints from semantically equivalent transformation to resolve challenging decompilation errors.
- To the best of our knowledge, we are the first to systematically handle Python decompilers generating logically incorrect code, which we call *Implicit Errors*.
- We evaluate PYFET on 17,117 real-world Python malware samples with decompilation errors obtained by ReversingLabs [39]. PYFET identifies 77,022 decompilation errors across five decompilers and fixes all of them.
- We present two case studies to show that PYFET can (1) handle opcode remapping, a recent anti-analysis technique that is used by `DropBox` and `druva` binaries and (2) migrate Python 3.9 binaries to 3.8 so that they can be decompiled by `Uncompyle6` and `Decompyle3`.
- We publicly release all the code and data [37].

Scope and Limitations. This paper focuses on fixing errors in decompilers that happen during the process of reconstructing high-level semantics from low-level representations (e.g., instructions). Decompilation errors from before the semantic recovery process, such as errors due to incorrect file headers or formats, are not our focus. PYFET modifies the target binaries, meaning that decompiled program can be different from the original binary program. While we carefully design the transformation rules to minimize the side effects, this may affect the forensic analysis of the target binary. Source-level obfuscation is out of our scope. If a binary is created from obfuscated source code, we aim to help decompilers obtain the obfuscated source code.

2. Background and Preliminary Study

2.1. Bytecode, Disassemblers, and Decompilers

A Python program can be executed from either source code or binary. Besides running a Python source code program (i.e., `.py` file), one can also run a `.pyc` binary file containing Python bytecode (a set of instructions for Python's virtual machine). The binaries can be generated by the '`-m compileall`' command. The binary format is preferred by cyber attackers as it is challenging to analyze.

Challenges in Decompilation. At a high-level, decompilation is a process of *inferring a mapping between the original source code representations from the low-level bytecode instructions*. Unfortunately, during the compilation process, a high-level language construct can be transformed into multiple forms of low-level bytecode instructions. For example, there can be multiple ways to implement a `for` loop, and the compiler may choose one of the ways to generate a binary program. In addition, the compiler can further optimize the generated bytecode (e.g., reducing multiple instructions into a simpler instruction), making it hard to infer the original statements from the instructions. Since there can be multiple possible mappings, decompilation is often considered an undecidable problem. In practice, decompilers often leverage heuristics to choose one of the mappings based on typical code generation patterns' of the compilers.

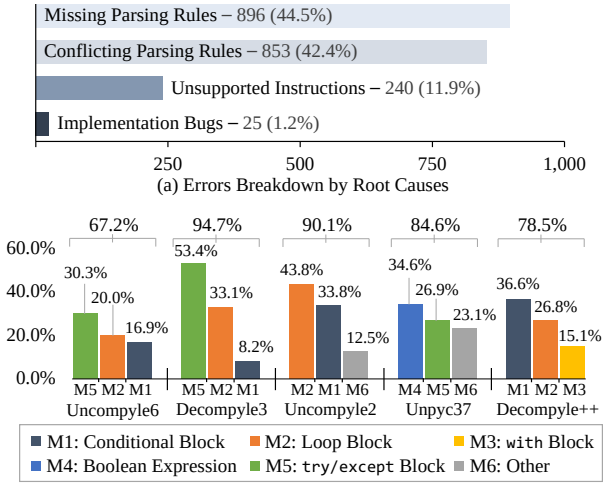


Figure 1. Causes of Decompilation Errors.

Python Binary File Structure. The `.pyc` file starts with a header containing the magic number representing the version of the Python compiler that generated the binary file. Then, it has a serialized code object containing the Python bytecode including instructions and data. The file and bytecode specifications vary between Python versions (e.g., instructions can be newly introduced or obsoleted).

Python Bytecode Disassemblers. Python disassemblers parse and extract the bytecode instructions while not aiming to recover the source code representation. Their key challenge is to handle the different instruction specifications.

Python Decompilers. `Uncompyle6` [47], `Decompyle3` [16], `Uncompyle2` [46], `Unpyc37` [48], and `Decompyle++` [17] are five popular Python decompilers. They take a Python binary program as input and aim to generate the input program's source code. They first disassemble the binary program to obtain bytecode instructions and then identify high-level language constructs from the instructions. For each high-level construct (e.g., a loop), they define *decompilation rules* that map patterns of instructions to high-level language constructs/structures. The decompilers repeat the process of applying decompilation rules on the input binary, until it recovers the entire source code of the target program.

2.2. Preliminary Study on Decompilation Failures

We conduct two preliminary studies. First, we analyze the causes of decompilation errors from real-world Python malware. Second, we study the correctness of decompilers' results (i.e., decompiled source code).

2.2.1. Causes of Decompilation Failures. We randomly select 150 samples from the real-world malware dataset for our evaluation in Section 5, obtaining 2,014 decompilation errors from the samples. We then investigate the causes of the decompilation errors in five decompilers: `Uncompyle6`, `Decompyle3`, `Uncompyle2`, `Unpyc37`, and `Decompyle++`. As a result, we identify that most decompilation failures stem from the following high-level causes:

1. *Missing Parsing Rules:* Decompilers do not have a proper parsing rule for a certain pattern in a binary. It is often a corner case of an already supported pattern.
2. *Conflicting Parsing Rules:* Decompilers fail to select a correct parsing rule when there are multiple applicable rules. Applying an incorrect rule may terminate the decompilation process or create a wrong result.
3. *Unsupported Instructions:* Decompilers do not support certain instructions (i.e., newly introduced instructions), leading to a decompilation failure.
4. *Implementation Bugs:* They are caused due to internal decompiler implementation bugs (e.g., segmentation fault and integer overflow causing infinite loop) that are not particularly related to the decompilers' design.

As shown in Fig. 1-(a), the missing and conflicting parsing rules are the two major causes. Decompilers maintain a large number of parsing rules (e.g., 1,408 rules in `Uncompyle6`, `Decompyle3`, and `Uncompyle2` on average), where many of them are inter-dependent. Hence, it is challenging to add new parsing rules or identify the problematic parsing rules to fix the error. Incorrectly added/fixed rules can introduce errors. We elaborate more in Appendix 5.1.

Handling Control Flows. Many errors are due to mishandling of control flow changing instructions. Specifically, Fig. 1-(b) shows the decompilers' modules/rules that are responsible for the errors. Major causes are conditional (36.6% in `Decompyle++`), loop (43.8% in `Uncompyle2`), and `try/except` (53.4% in `Decompyle3`) structures.

Undetected Conflicting Rules. We observe some errors caused by conflicting rules *incorrectly changing* parts of the decompiled result while not wrecking the entire decompilation process. We further investigate it in Section 2.2.2.

Representativeness of Decompilation Failures. We analyze all errors in Section 5 to infer the root causes. Our results show that errors in Section 5 follow similar distribution of that in Fig. 1. We elaborate in [35] (Section 7.1).

2.2.2. Correctness of Decompiled Source Code. Decompilers may produce an *incorrectly* decompiled source code. Hence, we empirically analyze the correctness of the decompilation results of 3,000 Python programs collected from diverse sources (e.g., CPython Standard Library [2] and popular projects in PyPI [4]). Specifically, we compile the source code programs to obtain `.pyc` binaries and use decompilers to get decompiled source code programs. Then, we compare the sources and binaries of both the original and decompiled programs illustrated in [35] (Section 7.2). The differences indicate potential decompilation failures. We observe the followings:

1. *Incorrectly Changing Semantics:* These errors essentially change the meaning of the program. They typically stem from overlapping parsing rules for the same set of instructions in binary.
2. *Harmless Additional Code:* Decompiled source code has additional code (e.g., additional `del` statements in an exception block for cleaning up variables) that do not impact the existing semantics of the program.
3. *Incorrectly Omitted Code:* Decompilers, particularly in `Decompyle++`, may produce a decompiled source code without certain parts of the program. We find that this is because of the incorrect parsing rules of the decompiler which skips the blocks and quits the parsing early.

Our work focuses on *the first case* (See Section 4.1.2). The second case is trivial and easy to detect. Hence, we detect and ignore them. The third case is out of our scope because it happens when the decompiler's implementation is *largely incomplete* for a certain pattern. We detect the third case by instrumenting every block in a binary and checking whether the instrumented code correctly appears in the decompiled source code. If it does not, we detect the third case and ignore it. We elaborate more in Appendix 1.

2.3. Observations and Challenges

More Failures on Newer Python Versions. We observe that decompilation failures are frequently observed on recent versions of Python binaries (e.g., 3.8 or later). This might imply that *code generation patterns* of the compilers are getting more complicated, imposing significant challenges.

Logically Different Decompiled Source Code. We observe that even if a decompiler successfully generates source code from the input binary, the decompiled source code may contain different logic. Since these errors happen silently without explicit error messages, detecting them is challenging. We call them *implicit errors* and discuss them in detail in Section 4.1.1. These errors can substantially affect all the downstream analyses that rely on the decompiled source code. Unfortunately, we have not seen active discussions about this problem, even though the severity of the problem is certainly concerning (e.g., we have observed 13,384 samples having such problems).

Anti Reverse-engineering Techniques. We observe the malicious techniques used by cyber attackers to thwart the decompilation processes. For example, attackers modify Python binaries to insert dummy/unusual bytecode instructions, breaking the patterns used by the decompilers (see Section 5.4.3). Also, a customized Python runtime with a randomized instruction set is used to prevent the decompilation of Python binaries [5, 26] (see Section 5.4.2).

3. Forensically Equivalent Transformation

Forensically Equivalent Transformation (FET) is a transformation aiming to preserve forensically critical semantics without strictly requiring preserving the semantics of the original program. Compared with the semantically equivalent transformation (SET), FET can conduct more diverse transformations, while some semantics might be lost.

Reasons for FETs. A salient reader may question why one needs to use FET instead of SET, because SET does not have concerns regarding losing semantics during the transformation. We define and use FETs, because we observed that using only SETs is insufficient to resolve various decompilation errors. In other words, fixing such errors requires *more aggressive transformations*, meaning that we may need to compromise the semantic preserving constraint.

TABLE 1. TRANSLATING KEYWORDS.

Keywords	Original Stmt.	Transformed Stmt.
Group 1: Keywords can be removed. Critical semantics preserved.		
await, async	`await task_fn`	`task_fn()`
Group 2: Keywords can be implemented by using other primitives.		

with, lambda	`with open(...) as f:`
	`  fn = lambda x:\`
	`  'y' if x%2==0\`
	`  else 'n'`
	`  f.write(fn(1))`
		`def FET_fn(x):`
		`  return 'y' if \`
		`  x%2==0 else 'n'`
		`f = open(...)`
		`f.write(FET_fn(1))`
		`f.close()`

Group 3: Keywords can be replaced with a function call.		
assert, del, raise	`assert(...)` `del obj`	`FET_assert(...)` `FET_del(obj)`
Group 4: Keywords for variable scopes can be removed.		
nonlocal, global	`global x` `x = ...`	`FET_global_x = ...`

* Blue and red represent keywords and transformed code by FET.

TABLE 2. TRANSLATING CONTROL FLOW STRUCTURES.

Keywords	Original Stmt.	Transformed Stmt.
Structure 1: Translating Exceptions to Conditionals.		
try-except	`try: s1` `except: s2`	`s1;` `if FET_error(): s2`
Structure 2: Breaking Chained Statements.		
if-elif, import-as	`if x: s1` `elif y: s2`	`if x: s1` `if not x and y: s2`
Structure 3: Translating Branching Statements to Conditionals in Loops.		

continue, break, return, yield	`while cond:`
	`  if s1:`
	`  continue`
	`  s2`
		`while cond:`
		`  if s1:`
		`  FET_continue = 1`
		`  if FET_continue == 0:`
		`  s2`

Structure 4: Taking Statements out of the Exception.		

return, continue in with	`with x as y:`
	`  return y.fn():`
		`with x as y:`
		`  FET_ret = y.fn()`
		`return FET_ret`

return, continue in try-except	`try:`
	`  return x`
	`except:`
	`  ...`
		`try:`
		`  FET_ret = x`
		`except:`
		`  ...`
		`if FET_ret is not None:`
		`  return FET_ret`

* Blue and red represent keywords and transformed code by FET.

For example, we observe that Python decompilers' supports for certain keywords (e.g., `async`, `await`, and `lambda`) are limited or lacking. After we apply a series of SETs on `async`, we find that none of the SETs we tried resolve the error since decompilers do not understand/implement the concept of these keywords. The closest transformation that works for the decompilers is practically removing the keyword (or transforming to a similar implementation without asynchronous instructions). This essentially motivates the development of FET, relaxing the concept of SET while not significantly affecting subsequent forensic analyses. FET is essentially a superset of SET, and FET includes any SETs.

Deriving FETs. We manually derive FETs by analyzing decompilation errors from 375 Python binaries collected from PyPI (90), CPython libraries (150), and open-source/malware samples (135), among the 3,000 samples we used in Section 2.2.2. Note that they are not parts of the dataset used in our evaluation.

Types. There are three types of FETs: (1) translating keywords (Section 3.1), (2) translating control structures (Section 3.2), and (3) inserting no-op instructions (Section 3.3).

3.1. Translating Keywords

Table 1 shows four groups of Python keywords that we transform. The first column shows the target keywords of the transformation, and the next two columns show examples of the original and transformed statements.

Group 1. FETs remove keywords in this group (e.g., `await` and `async`). Removing them has little to no change in the semantics for forensic analysis (i.e., it makes the program synchronous while all other semantics are preserved). Note that if there exist code snippets dependent on those keywords (e.g., registered callbacks), we do not apply this FET.

Group 2. Keywords in this group are translated into semantically *similar* implementations (i.e., FET) that do not use the keywords. The example in Table 1 shows a simplified version of the `with-as` implementation. Note that the semantics of `with-as` include an exception handler (i.e., `try-finally`). Hence, an example SET of '`with x() as f: f.write(y)`' is '`try: f = x(); f.write(y); finally: f.close()`'. In this example, there is a subtle difference between the SET and FET in how an exception is handled. Specifically, if an exception happens in the `with` block, the exception will be caught by the `try` block in the SET case, while it will be passed to an upper layer `try` block in the FET case.

Group 3. Keywords such as `break`, `continue`, and `return` are replaced with a function call. Transformations for this group change the semantic of the original statement, however, from the human reader (e.g., forensic analyst), the lost semantics have limited impact. Further, once successfully decompiled, those transformed functions can be replaced with the original form: '`FET_keyword()`'→'`keyword`'.

Group 4. Keywords defining the scope of variables are replaced with variables with a prefix of '`FET_`'. Similar to Group 2, Group 4 changes the semantics but it is not critical for forensic analysis, and they can be post-processed to recover the original semantics.

Other Keywords. There are a few remaining keywords that FET does not consider, as we have not observed decompilation errors caused by them. This is mainly because these keywords' semantics are straightforward or their usage pattern does not vary significantly, making it easier for decompilers to handle them. For example, there are constant values such as `True`, `False`, and `None`. Also, FET does not transform `pass` since it does not have a corresponding instruction in the binary. In other words, no code is generated for the `pass` statement. Lastly, there are keywords used to define objects and functions such as `class` and `def`.

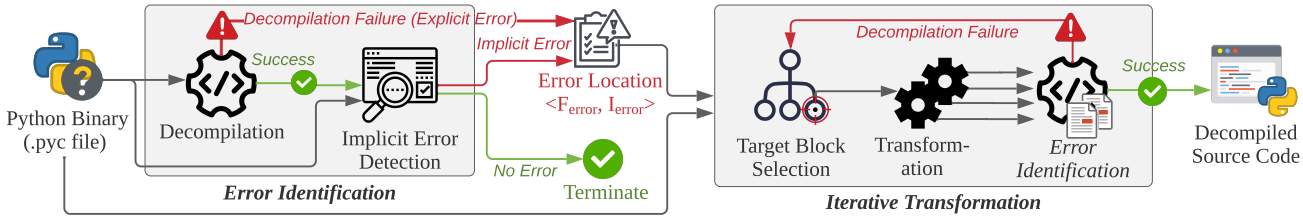


Figure 2. Overview of PYFET (Gray Boxes represent PYFET's Components).

3.2. Translating Control Structures

Table 2 shows the second type of FETs changing the control flow structure of the program.

Structure 1. This FET changes the program's control flow structure into a similar structure, while preserving all expression statements, including assignments, comparison, and function calls that are parts of the original control structure. For example, a `try-except` structure can be transformed into (1) a code block from the `try` and (2) another block from the `except`, guarded by an `if`.

Structure 2. Chained `if` statements are separated into multiple individual `if`s. For instance, `if-elif` can be translated into the two independent `if`s. Similarly, the `import` statement can have multiple imports with 'as' (e.g., `import x as a, y as b`), which can be separated into two `import` statements: '`import x as a; import y as b`'.

Structure 3. Branching statements (e.g., `continue`, `break`, `return`) in loops are translated to conditionals. For example, a `continue` with an assignment of a new variable `FET_continue`. The statement `s2` that is dependent on the original `continue` statement is now guarded by the '`if FET_continue == 0:`', emulating the original control flow structure. `break` and `return` can be similarly implemented by changing the loop condition (i.e., '`cond=0`').

Structure 4. FET can take control flow statements out of the exception handlers (e.g., `try-except` and `with`[1]). As shown in Table 2, a `return` in an exception is replaced with a `return` after the exception block.

Other Structures. Similar to the keywords that we do not consider in FET, structures not mentioned in this section are those that we have not observed in decompilation errors.

3.3. Adding No-operation Instructions

A FET can introduce new instructions if they do not affect the program's existing data and control flows. For example, adding no-operation instructions (i.e., instructions that do nothing meaningful in terms of data/control flow) to an existing basic block is a FET. There are multiple forms of no-op instructions: (1) calling an empty function '`FET_null()`' and (2) assigning a variable to itself (e.g., '`FET_null=FET_null`'). Note that Python has an existing `NOP` instruction (opcode 9) which we do not use because existing decompilers do not recognize and react to it.

1. '`with-as`' implements exception handling logic internally.

```
1 ...
2 def convert -- This code section failed: ---
3       ...
4     636  STORE_FAST               'converted_column'
5       ...
6 Parse error at or near `COME_FROM_LOOP' instruction at offset 638_1
```

(a) Extracting Error-inducing Instruction's Offset and Function

Error Location: <F_{error}, I_{error}> = <Ⓐ convert, Ⓑ638>

(b) Obtained Error Location

Figure 3. Example of Error Message Processing.

4. PYFET Design

Fig. 2 shows the overall procedure of PYFET, consisting of two phases: (1) Error Identification (Section 4.1) to detect decompilation errors and (2) Iterative Transformation (Section 4.2) to automatically resolve the detected errors.

4.1. Error Identification

Decompilation errors are detected by analyzing the execution outcome of the decompilers. For a detected error, we extract the error location, a tuple of a function containing the error and an error-inducing instruction's offset: (F_{error}, I_{error}). There are two types of errors: *explicit* and *implicit*. *Explicit errors* are are observed when a decompiler fails with an explicit error message. *Implicit errors* occur when a decompiler generates a decompiled code that is semantically different from the input binary without explicit errors.

4.1.1. Handling Explicit Errors. To detect explicit errors, we check for an error message from a decompiler. If a decompiler fails without any error message (e.g., the decompiler process hangs and does not terminate), we do not consider them explicit errors (Discussion in Section 6).

Obtaining Error Location. We parse error messages from the decompiler to obtain the error location. Fig. 3-(a) shows an example error message from `Uncompyle6` [47]. It contains the function name where the decompilation failed at line 2 (F_{error}), and the error-inducing instruction's offset (I_{error}) at line 6. We use a regular expression to extract the two pieces of information (Ⓐ and Ⓑ in Fig. 3-(b)). Note that if an error-inducing instruction's offset is not available, we use the first instruction's offset (i.e., the root of a CFG).

4.1.2. Handling Implicit Errors. Fig. 4 outlines PYFET's implicit error detection process, consisting of four steps.

1. *Implicit Error Pattern Searching:* We check whether the decompiled source code has a structure that matches a

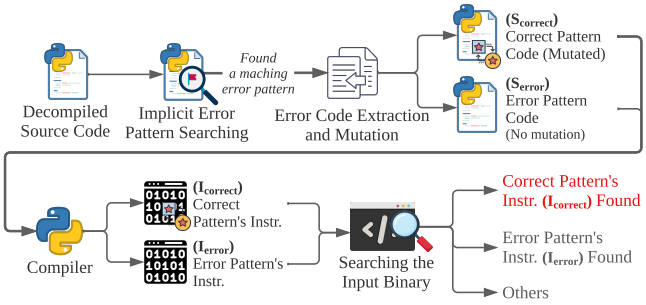


Figure 4. Implicit Error Detection Procedure.

predefined implicit error pattern[2]. Note that the matching happens at the source code level. If there is no matching implicit error pattern, we report no implicit errors.

2. *Error Code Extraction and Mutation:* If we identify a code snippet that matches an implicit error pattern, we extract the matching part of the source code, which we call "Error Pattern Code" (S_{error}) and its error location (i.e., <F_{error}, I_{error} >). Then, we mutate S_{error}'s code structure (as in the second column of Table 3) to its correct pattern's structure (as in the third column of Table 3), obtaining $S_{correct}$ (Correct Pattern Code).
3. *Compilation:* We compile $S_{correct}$ and S_{error} to obtain their bytecode instructions: Correct Pattern Code's instructions ($I_{correct}$) and Error Pattern Code's instructions (I_{error}). We use the same compiler version that generated the input binary, according to the binary's header.
4. *Searching the Input Binary:* We search the original input binary to check whether the matching instructions of I_{error} or $I_{correct}$ exist in the input binary at the *same error location*. Since we found S_{error} in the decompiled program in step 1, if the original input binary has $I_{correct}$ at the same location, it means the original binary and the decompiled program are *different*, suggesting an implicit error. Observe that we compare the code snippets in *two programs* (i.e., the decompiled program and the original binary) *at the same location*. If both have S_{error} and I_{error}, it means they are consistent hence no implicit errors are detected. Details of other outcomes of the search can be found in Appendix 2.2.

Implicit Error Patterns. Table 3 shows 6 selected implicit error patterns. The remaining patterns are in Appendix 2.1.

P1. A decompiler incorrectly includes statements (e.g., '`return s2`') after an `if` into the `if` block.
P2. A nested `if` with an inner `if` containing an `else` block is incorrectly decompiled to the outer `if` paired with the `else` (which should be paired with the inner `if`).
P3. When there are two consecutive `if`s, the second `if` was paired with the first `if`'s `else`.
P4. A decompiler introduces an `else` to `try` and includes the subsequent statements under the new `else`.

2. The patterns are obtained by studying various Python programs' decompilation results. Details of the study for predefined implicit error patterns is in Appendix 2.1, including examples shown in Table 3.

TABLE 3. IMPLICIT ERROR PATTERNS.

Pattern Name	Implicit Error Pattern	Correct Pattern

P1: `if` includes `with/return`	`if c1:`
	`␣␣s1`
	`␣␣return s2`
		`if c1:`
		`␣␣s1`
		`return s2`

P2: Incorrectly paired `else`	`if c1:`
	`␣␣if c2: s1`
	`else:`
	`␣␣s2`
		`if c1:`
		`␣␣if c2: s1`
		`␣␣else:`
		`␣␣␣␣s2`

P3: `if` introduced `else`	`if c1: s1`
	`else:`
	`␣␣if c2: s2`
		`if c1: s1`
		`if c2: s2`

P4: `try` introduced `else`	`try: s1`
	`except: s2`
	`else: s3`
		`try: s1`
		`except: s2`
		`s3`

P5: `for` introduced `else`	`for x in y:`
	`␣␣s1`
	`else:`
	`␣␣s2`
		`for x in y:`
		`␣␣s1`
		`s2`

P6: Removed the inner loop when nested	`for x in y:`
	`␣␣if c1:`
	`␣␣␣␣s1`
	`else:`
	`␣␣s2`
	`␣␣s3`
		`for x in y:`
		`␣␣while c1:`
		`␣␣␣␣s1`
		`␣␣s2`
		`s3`

* Blue text represents `keywords` and highlighted lines show the differences between the error and correct code.

P5. A decompiler introduces an `else` to `for` and includes the subsequent statements under the new `else`.
P6. When there is a `while` in a `for`, a decompiler completely removes the inner `while`. Further, it also incorrectly introduces an `else` and puts statements that should not belong to (`s2` and `s3` in the example).

Note that for `P4` and `P5`, the `else` block of `try` or `for` will only be executed when there is no exception raised or `break` executed respectively. The decompiler essentially mispresents the critical control flow logic of the program.

Obtaining Error Location. In the *Code Extraction and Mutation* process, we obtain the location of S_{error}.

4.2. Iterative Transformation

Outline. As illustrated in Algorithm 1, we iteratively transform the input binary via three steps. First, we obtain a control flow graph of the input binary (line 2) and choose the initial target block containing the error location (*ErrorLoc*) (line 3). Second, we transform instructions in the chosen target block (line 5). Note that there can be multiple transformations that are applicable to a block, represented as multiple lines between the transformation and error identification in Fig. 2 (also, the algorithm returns a set of transformations at line 5). For each transformed binary from each applied transformation, we run the error identification process to check whether the transformed binary does not have decompilation errors (lines 5~8). If it resolves the originally targeted error, we consider the transformation is successful (line 9). Otherwise, we try the same process iteratively on other blocks adjacent to the current target blocks (line 10).

Algorithm 1: Forensically Equivalent Transformation

```
Input : Bin: input Python binary.
        ErrorLoc: the location of the error-inducing instruction.
Output: DecResult: a decompiled source code.
1  procedure ForensicEquivTrans(Bin, ErrorLoc)
2      CFG ← GetCFG (Bin)
3      targetSet ← SelectInitialTarget
         (CFG.root, ErrorLoc)
4      while targetSet ≠ ∅ do
5          transformedSet ← TransformBin (CFG, targetSet)
6          for trBin ∈ transformedSet do
7              DecResult ← Decompile (trBin)
8              if IdentifyError (DecResult) = ∅ then
9                  return DecResult
10         targetSet ← SelectAdditionalTarget (targetSet)
11     return Failure
```

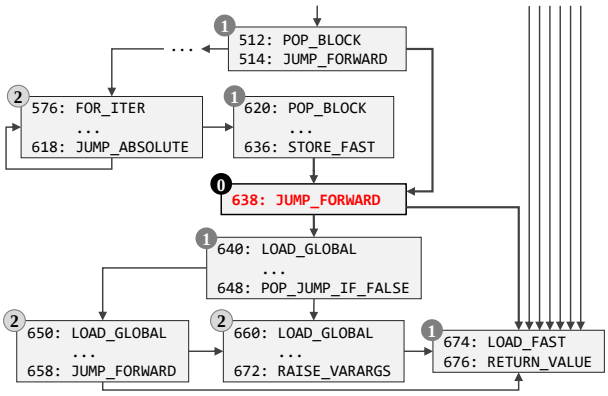


Figure 5. Control Flow Graph of `convert()` in Fig. 3.

The process terminates with a failure when no target blocks are left to process and no successful transformation is found. This essentially means that PYFET fails to find a solution. Note that if a binary contains multiple errors, we run the algorithm multiple times, fixing a single error at a time.

4.2.1. Obtaining CFG. Our transformation is applied per basic block. Hence, before the analysis, we obtain a control flow graph (CFG) of an input binary. We use a default Python disassembler to obtain all the instructions from the binary, and then identify control flow changing instructions.

4.2.2. Target Block Selection. The initial target block is essentially the basic block containing the error-inducing instruction. For example, the basic block ⓪ in Fig. 5 is the initial target block obtained from the example in Fig. 3. We then apply transformations to the identified target block.

Extending Target Blocks (Iterative Process). The initial block may not be the root cause of the error, meaning that transforming instructions in the initial target block (⓪) may not resolve the error. In such a case, PYFET selects additional target blocks for transformation incrementally. Specifically, PYFET first selects blocks that are *directly* connected to the initial block (i.e., reachable if we ignore the direction of the edges in the graph). Fig. 5 shows an example. ⓪ is the initial target block, where the next

```
([POP_JUMP_IF_FALSE,POP_JUMP_IF_TRUE].*)+
 [POP_JUMP_IF_FALSE,POP_JUMP_IF_TRUE]
```

(a) Regular Expression for Identifying Target Instructions

Ref.	Pattern	Transformation
RE-1	[POP_JUMP_IF_FALSE, POP_JUMP_IF_TRUE]	POP_JUMP_IF_FALSE → **JUMP_IF_FALSE_OR_POP** | POP_JUMP_IF_TRUE → **JUMP_IF_TRUE_OR_POP**
RE-2	.	No Change
RE-3	[POP_JUMP_IF_FALSE, POP_JUMP_IF_TRUE]	POP_JUMP_IF_FALSE → **(STORE_FAST, LOAD_FAST, POP_JUMP_IF_FALSE)** | POP_JUMP_IF_TRUE → **(STORE_FAST, LOAD_FAST, POP_JUMP_IF_TRUE)**

(b) Regular Expression Patterns and Transformation Rule Definitions

Figure 6. Regular Expression Example.

selectable blocks are marked by ①[3]. Observe that blocks 512~514 and 620~636 are connected to ⓪ via backward edges, and blocks 640~648 and 674~676 are connected via forward edges. If transforming all four additional blocks does not work, PYFET further extends the target blocks that are directly connected from all the selected blocks. Those blocks are marked by ②. We keep extending the target blocks until our transformed binary leads to successful decompilation or covers all the basic blocks in the function (without successful decompilation, implying that it fails).

Target Block Selection Algorithm. PYFET selects a target block for transformation as described in Algorithm 2. There are two functions SELECTINITIALTARGET() and SELECTADDITIONALTARGET(), for selecting the initial target block and additional target blocks respectively. SELECTINITIALTARGET() takes two inputs: *rootBlock* and *ErrorLoc*. *rootBlock* is the root node of the CFG of a function, and *ErrorLoc* is the error-inducing instruction's offset obtained in Section 4.1 (Error Identification). It initializes *targetSet* and *blockQueue* with the root block of the CFG, meaning that the root block is used as a default target block.

At lines 4~12, for each block (starting from the root block), it checks whether there is an instruction that has the same address as the error location (lines 6~7). Once found, it returns *targetSet* containing the current block with the error-inducing instruction. After processing the block, it enqueues all the successor blocks (i.e., children) to *blockQueue* so that they are processed in the subsequent loop iterations.

SELECTADDITIONALTARGET() selects additional target blocks if PYFET fails to resolve the decompilation errors after transforming all the current target blocks in *targetSet*. Additional target blocks are selected from the current target blocks' immediate predecessors and successors (lines 17~18 and 19~20). Finally, it returns the new set *newTargetSet*.

4.2.3. Transformation (FET). A FET consists of (1) regular expressions for searching target instructions to transform and (2) rules for code transformations.

Identifying Target Instructions. We use a regular expression consisting of Python bytecode's instruction mnemonics to describe target instructions to transform. Specifically, an element of our regular expression is instruction.

Transformation Rule Definition. It defines how to replace the matched instructions with new instructions. Fig. 6-(a)

3. The numbers indicate the processing order of target blocks (0 for initial, 1 for the next round, and so on).

TABLE 4. EXAMPLES OF PYFET'S TRANSFORMATION RULES.

	Name	Original Stmt.	Transformation	Description
R1	Dividing Logical Expressions	`(x or y) and z`	`t = x or y; t and z` `(x and z) or (y and z)`	Compound logical expressions in a (conditional) statement can be divided into smaller expressions in multiple steps.

R2	Simplifying Control Flow Structures	`if x:`
		` s1`
		`elif y:`
		` s2`
			`if x:`
			` s1`
			`if y and not x:`
			` s2`
				Complex/nested conditional statement structures often cause errors during the decompilation. Flattening means to transform the structures into a typical/ basic structure while preserving the semantic.

R3	Eliminating Unreacheable Code	`return x` `for x in y:`	`return x`	Unexpected/unusual code sequences (e.g., code after `return` can break the decompilers. They can be removed.

R4	Eliminating Control Flow Changes at the End of Loop	`while x:`
		` ...`
		` break`
		`...`
			`while x:`
			` ...`
			` break`
			` FET_null = FET_null`
				`break`/`continue` statements are implemented as jump instructions. If it is right next to the loop end, which also consists of multiple consecutive jumps, decompilers fail to parse which jump instructions are for the loop end.

R5	Detaching Control Flow Changing Stmt. in Exception Handler	`with x as y:`
		` return y.f()`
			`with x as y:`
			` FET_return = y.f()`
			`return FET_return`
			`y = x; z = y.f(); y.close();`
			`return z`
				`with` statement in Python runs the code inside an exception handler (i.e., `try`/`finally`). Control flow changing statements (e.g., `continue`/`return`) result in complicated code, leading to decompilation failures.

R6	Unsupported Language Features	`f = lambda x:\` ` x*2`	`def f(x):` ` return x*2`	Certain language features (e.g., `lambda` functions) may not be supported by decompilers.

* Blue and red represent `keywords` and `transformed code` by FET.

Algorithm 2: Target Block Selection Algorithm

```
Input : rootBlock: the root basic block of the binary's CFG.
        ErrorLoc: the location of the error-inducing instruction.
Output: targetSet: a set of selected target blocks.
procedure SelectInitialTarget(rootBlock, ErrorLoc)
    targetSet.push(rootBlock)
    blockQueue.enque(rootBlock)
    while blockQueue = ∅ do
        curBlock ← blockQueue.dequeue()
        for ins ∈ curBlock do
            if Address(ins) = ErrorLoc then
                targetSet ← ∅
                targetSet.push(curBlock)
                return targetSet
        for succBlock ∈ curBlock.successors do
            blockQueue.enqueue(succBlock)
    return targetSet
procedure SelectAdditionalTarget(targetSet)
    newTargetSet ← ∅
    for curBlock ∈ prevTargetSet do
        for predBlock ∈ curBlock.predecessors do
            newTargetSet.push(predBlock)
        for succBlock ∈ curBlock.successors do
            newTargetSet.push(succBlock)
    return newTargetSet
```

shows one of the regular expressions and transformation rules we used. It detects a chain of instructions consisting of `POP_JUMP_IF_FALSE` or `POP_JUMP_IF_TRUE` and a block of (any) instructions between the jump instructions. Fig. 6-(b) shows the notations we use in this paper to refer to matching patterns of the input. Specifically, we use `RE-1` to refer to instructions matched with the first pattern of the regular expression shown in (a). Similarly, `RE-2` and `RE-3` refer to the second and third patterns. The third column presents transformation rules that will be applied to the matched instructions. Specifically, for the instructions matched with `RE-1`, we replace `POP_JUMP_IF_FALSE` and `POP_JUMP_IF_TRUE` with `JUMP_IF_FALSE_OR_POP` and `JUMP_IF_TRUE_OR_POP` respectively. For the instructions matched with `RE-2`, we do not transform, meaning that they remain intact. `RE-3` adds two instructions: `STORE_FAST` and `LOAD_FAST`. We give an example of how the FET in Fig. 6 is applied on [35] (Section 8.1).

FET Rule Examples. Table 4 shows a few selected FETs including intuitive explanations of the rules. Note that the transformations are done at the binary level. We use the source code representation for the presentation purpose. The remaining rules are presented in Appendix 3.1.

4.2.4. Error Identification. After the transformation process, we may obtain multiple transformed binaries. For each binary, we go through the same error identification process (Section 4.1). If any of them can be decompiled without errors (both implicit and explicit), we conclude that the original error is resolved. Otherwise, we repeatedly try other target blocks until we succeed or exhaust all the blocks.

Binary with Multiple Decompilation Errors. A binary can have multiple error-inducing instructions at different locations. To facilitate discussion, we use E_i, L_i, and B_i to represent a decompilation error (E_i) in a binary (B_i) and its error location (L_i). Assume that we initially run PYFET to fix E_1 at L_1, if we observe another error E_2 at a different location L_2 ($\neq L_1$) on a new transformed binary B_n, it means that (1) we resolved the previous error E_1 and (2) the current binary contains two decompilation errors: E_1 and E_2. Hence, we reiterate the entire process to fix E_2 at L_2, on the transformed binary B_n. Note that when fixing multiple errors, we handle multiple transformed binaries that

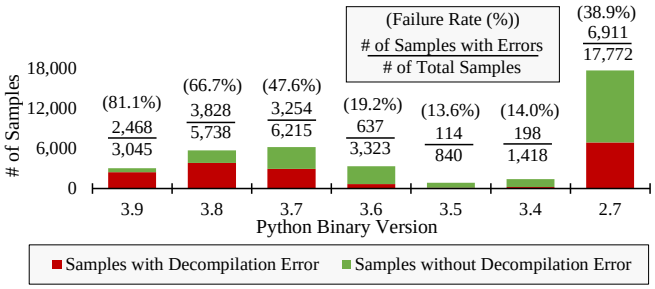


Figure 7. Python Malware with Decompilation Errors.

may have changed instruction addresses (or offsets) due to our transformations by maintaining a mapping of addresses between the input and transformed binaries.

5. Evaluation

Implementation. The prototype of PYFET is implemented in Python. The total SLOC is 6,243, including all the tools that we develop to assist PYFET: e.g., the control flow graph (CFG) generator and logging/result processing tools.

Environment Setup. All the experiments are conducted on a Ubuntu Linux 18.04 with an i9 3.70GHz CPU and 64GB RAM. We use Docker [30] with official Python images to handle individual Python versions independently.

Base Python Malware Samples. We collect Python malware binaries (i.e., `.pyc` files) through ReversingLabs [39] which is one of the largest threat intelligence research labs. Each sample from ReversingLabs is annotated with a threat level that represents the maliciousness, ranging from 0 to 5, where 0 indicates 'benign' and 5 means 'highly malicious'.

We search for two types of files with the threat level ranging from 1 to 5, between June 2016 and July 2021: (1) `.pyc` files and (2) binaries packed by popular Python packers such as [14]. We obtain 1,905 unique files of the first type (i.e., `.pyc` files), and 631 files of the second type (i.e., packed binaries) which contain 36,446 unique `.pyc` files when extracted. In total, we obtain 38,351 `.pyc` samples.

Identifying Malware Samples for Evaluation. To identify malware samples causing decompilation errors, We use five Python decompilers, `Uncompyle6`, `Decompyle3`, `Uncompyle2`, `Unpyc37`, and `Decompyle++` to decompile the 38,351 malware samples. We then identify 17,117 samples (45.6% of the total) causing decompilation failures. Fig. 7 shows the results by the Python versions with respect to the decompilation results (i.e., successful or failed).

Python 2.7 samples caused the most decompilation errors (6,911 samples). This is, in part, because our dataset contains significantly more 2.7 version samples than other versions. In terms of the failure rate, Python 3.7 and later versions have higher failure rates, 66.7% (for 3.8) and 81.1% (for 3.9), than Python 2.7 (38.9%). Note that since only `Decompyle++` officially supports Python 3.9 binaries, all the decompilation failures of 3.9 are from `Decompyle++`.

5.1. Effectiveness of PYFET

We use the 17,117 samples with decompilation errors to measure the effectiveness of our approach. The '# of Errors' column in Table 5 shows the number of implicit and explicit errors from the samples. As shown in the 'Fixed' column, PYFET successfully resolves *all the decompilation errors*.

 # Errors in File/Function. We find that most (62.2%, 10,653) erroneous samples have more than one error, and for functions, almost half of the erroneous functions (52.9%, 66,253) have multiple errors. On average, a single file has 4.1 errors, and a single function has 1.7 errors. `Decompyle++` has significantly more errors in a file/function than any other decompiler. The most number of errors in a single file is 91, which is from a Python 3.9 sample frequently using three instructions that are not supported by the decompiler: `JUMP_IF_NOT_EXC_MATCH`, `DICT_MERGE`, and `LIST_TO_TUPLE`. Our further inspection result reveals that `Decompyle++` does not support 45 opcodes across all Python versions, and we find 8,528 samples causing errors due to 23 unsupported opcodes. Details of the unsupported instructions can be found on [35] (Section 9.1).

Frequently Applied FETs. Table 5 shows the top 3 FETs that resolved the most number of decompilation errors for the samples from each Python version for each decompiler. The 'All' rows show the average of all the versions for each decompiler. We have significantly more errors in `Decompyle++` as compared to other decompilers (i.e., 73.4% of the total samples). Hence, two top FETs (`R11` and `R10`) are from `Decompyle++`. The top 3 FETs used are as follows:

- `R11`. Eliminating unpacking variables using asterisks.
- `R7`. Moving control flow changing instructions (e.g., `continue` and `return`) out of `try` or `except` block.
- `R10`. Simplifying `else` blocks of control structures by detaching statements into a separate conditional.

`R11` is commonly used by `Uncompyle6`, `Decompyle3`, and `Decompyle++`, handling 25% of all errors (4,279). The top 3 FETs handle 61.2% of errors (47,116), showing that a few FETs can handle a large number of errors. We also observe that a single FET handles errors caused by multiple causes. For instance, `R5` resolves errors caused by a missing grammar in `Uncompyle6` and an unsupported instruction `WITH_CLEANUP_START` in `Decompyle++`. In addition, we observe that FETs are not specific to a particular decompiler: (1) `R7` is effective for `Uncompyle6`, `Decompyle3`, and `Unpyc37` and (2) `R8` resolves errors for `Uncompyle6`, `Decompyle3`, and `Uncompyle2`. Many FETs (e.g., `R1`~`R3`, `R5`, `R7`, and `R8`) also work for multiple Python versions.

The 'Remains' column shows the number of the rest of the FETs that successfully resolved decompilations errors, with the accumulated percentages.

# of Blocks Attempted. The 'Blocks Attempted' column shows the number of basic blocks PYFET iterates and tries FETs throughout the process. It is dependent on the distance between the error location and the block containing the error-inducing instruction. In general, if there exist long and nested conditionals or loops along the path, it results in a large number of basic blocks attempted. The average number of blocks attempted is 6.5 (out of 19.6). Note that `Decompyle++` and `Unpyc37` do not provide the error location, hence we use the root node of a function as an error location. As a result, we observe more number of

TABLE 5. PYFET EVALUATION WITH THE 5 DECOMPILERS

Decomp.[1]	Version	# of Errors				# Errors in		Frequently Applied (Successful) FETs				Blocks Attempted			FETs Attempted		
								Top 3 FETs			Remains[3]	Min~Max	Average		Min~Max	Avg.	Per Block
		Explicit	Implicit	Total	Fixed	File	Fn.[2]	1st	2nd	3rd			Blocks	Layers			
Uncompyle6	3.8	5,464	1,500	6,964	100%	2.9	1.1	R7 (57.0%)	R8 (18.5%)	R5 (14.1%)	7 (10.4%)	1~10	5.4	2.7	1~14	4.1	1.5
	3.7	465	501	966	100%	2.4	1.1	R2 (47.9%)	R1 (47.2%)	R7 (2.1%)	3 (2.8%)	1~19	8.9	3.7	1~29	4.2	0.9
	3.6	39	11	50	100%	1.7	1.3	R2 (53.7%)	R5 (31.7%)	R9 (9.8%)	3 (4.8%)	2~15	5.4	2.6	1~8	3.8	1.4
	3.4	7	0	7	100%	1.0	1.0	R3 (42.9%)	R8 (28.6%)	R1 (14.3%)	1 (14.2%)	1~13	6.3	3.7	1~6	4.8	1.5
	2.7*	239	1	240	100%	1.4	1.4	R17 (98.7%)	R1 (1.3%)	-	-	1~5	1.2	1.0	1~4	1.1	1.1
	All	6,214	2,013	8,227	100%	2.7	1.1	R7 (50.2%)	R8 (19.9%)	R5 (10.2%)	10 (19.7%)	1~19	5.7	2.9	1~19	4.0	1.4
D3	3.8	5,621	1,064	6,685	100%	2.8	1.3	R7 (54.0%)	R8 (33.4%)	R3 (8.4%)	7 (4.2%)	1~13	5.5	2.8	1~17	4.4	1.6
	3.7	384	454	838	100%	2.0	1.5	R1 (37.5%)	R8 (35.4%)	R2 (12.0%)	5 (15.1%)	1~18	8.1	3.8	1~10	4.3	1.1
	All	6,005	1,518	7,523	100%	2.7	1.3	R7 (53.3%)	R8 (33.1%)	R3 (7.9%)	9 (5.7%)	1~13	5.8	2.9	1~17	4.4	1.5
U2	2.7*	16	2,144	2,160	100%	1.4	1.0	R2 (43.8%)	R8 (37.5%)	R4 (6.3%)	2 (12.4%)	3~9	5.7	2.8	2~8	4.5	0.8
U3	3.7	80	2,474	2,554	100%	2.0	1.0	R1 (34.6%)	R7 (26.9%)	R6 (14.1%)	4 (24.4%)	1~23	8.8	3.1	1~13	6.0	0.7
Decompyle++	3.9	9,288	1,214	10,502	100%	27.1	8.7	R4 (29.6%)	R11 (23.3%)	R13 (21.0%)	5 (26.1%)	1~161	10.1	4.9	2~204	12.8	1.3
	3.8	12,271	1,881	14,152	100%	16.0	7.5	R10 (42.1%)	R11 (34.4%)	R16 (6.3%)	8 (17.2%)	1~129	9.3	4.8	1~245	17.6	1.9
	3.7	5,335	2,924	8,259	100%	7.5	2.8	R11 (49.3%)	R5 (24.3%)	R9 (13.2%)	6 (13.2%)	1~65	5.8	2.2	2~121	11.8	2.0
	3.6	3,795	1,754	5,549	100%	9.1	3.2	R11 (47.5%)	R5 (25.4%)	R9 (10.9%)	6 (16.2%)	1~65	5.2	2.1	2~109	10.9	2.1
	3.5	430	187	617	100%	5.4	3.3	R5 (49.4%)	R16 (29.1%)	R14 (9.6%)	4 (11.9%)	1~34	5.9	2.8	2~48	8.9	1.5
	3.4	307	388	695	100%	3.6	1.8	R9 (70.7%)	R13 (10.9%)	R16 (8.4%)	2 (10.0%)	1~73	6.5	3.3	3~104	12.7	1.9
	2.7*	2,773	14,011	16,784	100%	3.2	1.2	R16 (77.8%)	R2 (14.6%)	R15 (4.0%)	1 (3.6%)	1~208	7.9	4.2	1~235	9.6	1.2
	All	34,199	22,359	56,558	100%	6.6	2.5	R11 (34.2%)	R10 (19.3%)	R16 (13.6%)	10 (32.9%)	1~208	7.7	4.7	1~245	12.1	1.6

1: Decompilers (D3: Decompyle3, U2: Uncompyle2, U3: Unpyc37). 2: Function. 3: The number of unique FETs successfully resolve decompilation errors. *: 2.7 or earlier.

blocks attempted than other decompilers, as PYFET has to traverse to the instruction causing errors from the root node. In particular, a sample makes `Decompyle++` traverse 208 blocks (out of 245 blocks) because in the sample's CFG, the error is very far from the root node. Overall, we find that PYFET explored 37.6% of the basic blocks (with an average of fewer than 11 blocks) from all the samples, showing that our algorithm is efficient.

# of FETs Attempted. Multiple FETs might be applicable to a single basic block. Hence, the number of FETs attempted reflects the effectiveness of PYFET in applying FETs for each block. The first two sub columns show the number of FETs attempted across all the basic blocks, which is 7.4 on average. Recall that PYFET explores 7 basic blocks on average. It means that for each block, there are typically one or two matching FETs. Observe that all the decompilers except `Decompyle++` have fewer FETs attempted than the number of blocks attempted, due to the blocks that do not have any applicable FETs. Except for `Decompyle++`, up to 6 FETs on average are attempted, which is a small number of attempts. Lastly, `Decompyle++` has substantially more FETs attempted than the number of blocks attempted. Our analysis on the cases shows that they go through many nested chains of conditionals consisting of multiple boolean expressions. When PYFET attempts to break down those expressions, it iteratively tries FETs (e.g., `R1`) on each boolean expression.

5.2. Correctness and Impact of PYFET

5.2.1. Correctness. We conduct the following experiment to understand the correctness of PYFET's transformations.

1. We collect 14,949 Python source code files from 100 popular Python programs (Details of the selection process can be found in Appendix 4.2).
2. From the step 1's dataset, we randomly select samples to obtain 40 decompilation errors for each of the 30 FETs[4], resulting in 1,200 decompilation errors.

3a. For each error sample from step 2, we manually apply a source-level FET to the original source and compile the transformed source[5].

3b. We compile the 1,200 samples from step 2, and then apply PYFET to the compiled binaries, obtaining 1,200 transformed binaries.

4. Compare transformed binaries from steps 3a and 3b to evaluate the *correctness of transformation by* PYFET.

The comparison results show that there is no bytecode differences between the binaries from the step 3a and 3b, meaning that all the transformations by PYFET are correct.

5.2.2. Impact of Transformation. We conduct another experiment to understand the impact of PYFET's transformations in the decompiled source code.

* *We borrow the steps 1, 2, and 3b from the correctness experiment in Section 5.2.1 for the steps 1,2, and 3.*

3. Same to 3b. We obtain 1,200 transformed binaries.
4. We use decompilers to obtain the decompiled source code from the 1,200 transformed binaries.
5. Compare the decompiled source from step 4 with the original source from step 2.

The results show that, at the source code level, PYFET's impact is less than 3 lines of source code on average across all FETs. Specifically, 14 FETs (`R17~R30`) do not affect the decompiled source code, 16 FETs (`R1~R16`) cause the changes of 3.5 lines of the source code, and 8 FETs

4. For rules that we do not have 40 errors, we manually inject errors *at the source code level* and repeat the steps 2~4.

5. If a FET is applicable for multiple Python versions, we try to use different compiler versions evenly.

(R4, R8~R11, R13, R15~R16) are reversible. The maximum number of lines changing we see is for R6 (Adding support for unsupported language features) with 8 lines of code change. We observe that it was because of converting dictionary comprehension into a separate function that takes additional lines of code for definition. However, we see only one line of code change in the original function (i.e., changing variable to be assigned to invoked function rather than dictionary comprehension). We also note that R2 can impact readability by generating complex `if` blocks from a large chain of `elifs`. While R5 removes implicit exception handlers from removing `with` block, potentially impacting other exception handlers. At the bytecode level, on average 2.9 instructions are changed, where all are intended for fixes. We observe that PYFET only changes the targeted statement/control structure but not others.

In terms of the control flow, the impact of our transformation is limited. Specifically, 22 rules (R3, R6, and R11~R30) have no change in the program flow. The remaining 8 rules (R1, R2, R4, R5, and R7~R10) have on average 2.3 changes (320 cases in total) in edges. However, they are all intended (e.g., R7 moving '`continue`' out of the loop). Due to the space, we provide code snippets of 60 example transformations by PYFET in real world applications on [35] (Section 12) to show the impact of PYFET's transformation. We provide 1,200 example transformations on [36] as well.

5.3. Performance of PYFET

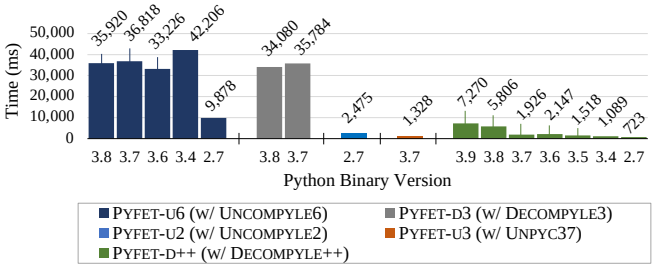


Figure 8. Performance of PYFET per Error.

Fig. 8 shows the overhead per error for PYFET across all samples. Note that PYFET is a framework that can choose which decompiler to use, resulting in 5 versions of PYFET in the result, each with a different decompiler. For decompilers that do not support certain Python versions, we do not report the results for the versions. PYFET-U6 and PYFET-D3 both take around 33~42 seconds to fix an error for Python versions except for 2.7 which takes around 10 seconds. Meanwhile, PYFET-U2, PYFET-U3, and PYFET-D++ take on average 2.7 seconds because they run faster than others. Note that most of the execution time is spent on running decompilers. The pattern matchings and transformations take less than 1.7 seconds (on average 84.2 ms).

5.4. Case Studies

5.4.1. Handling 3.9 Binaries. Among the five decompilers we evaluated, `Decompyle++` is the only decompiler supporting Python 3.9 binaries. However, it has a high failure rate (81.1%), as shown in Fig. 7. In this section, we use PYFET to transform 3.9 binaries to 3.8 to enable support of Python 3.9 binaries for `Uncompyle6` and `Decompyle3`.[6] In our base dataset, there are 3,045 Python 3.9 binaries.

Step 1. Changing the File Version. The first two bytes of `.pyc` files represent the magic number indicating the Python version. We modify it to '3413' which is the magic number for 3.8 version. This successfully handles 415 samples.

Step 2. Translating 3.9 Instructions to 3.8. 11 newly introduced instructions and 11 removed (i.e., obsolete) instructions in 3.9 are shown in [35] (Table 15). To accommodate the changes, we define 13 additional rules (more details in [35] (Section 10.1). Note that all the 13 transformation rules are SET (semantically equivalent transformation), meaning that the translated 3.8 binaries are semantically equivalent to their 3.9 binaries. The rules are straightforward, i.e., replacing a single instruction with another single instruction (e.g., `IS_OP` and `COMPARE_OP`). The most complicated rule is `RE-6` which translates a dictionary creation into a map creation. Note that coming up with the rules was fairly easy and only took 8 hours by one of the authors.

Decompilation Errors after Version Conversion. With the 13 rules (R18~R30), we successfully convert 3.9 binaries to 3.8 binaries. However, we observe that the converted 3.8 binaries suffer from similar decompilation errors we observed in the existing 3.8 binaries. Specifically, we show that the FETs used for 3.9 (except for the rules for version conversion) are similar to the FETs for 3.8 in Appendix 3.1.

Evaluation of Transforming Binaries. After changing the file version, we find a total of 214,831 errors across `Uncompyle6` and `Decompyle3`. PYFET resolves all errors making all binaries decompilable by the two decompilers. We give further details in [35] (Section 10.1).

5.4.2. Opcode Remapped Python Binaries. Opcode remapping is an anti-reverse-engineering technique against decompilers. It modifies the Python runtime environment with new opcode definitions that are not compatible with standard Python opcodes. As a result, any decompilation attempt fails because decompilers do not know the modified Python opcode definitions. In this case study, we use PYFET to handle opcode remapped Python binaries from two real-world products: `Dropbox` [3] and `druva inSync` [19].

Decompilation Attempt. We use `Uncompyle6` to decompile two Python binaries from `Dropbox` (Python 3.8 binary) and `druva inSync` (Python 2.7 binary). The decompiler hung and did not successfully terminate on both binaries, failing to decompile them. Note that while PYFET requires a decompiler to terminate and decompile or throw errors, we treat the hanging behavior here as an error and proceed.

Initial Analysis. We manually analyze both files and find that they have unknown magic numbers on the header. After we correct the magic numbers, however, we still observe decompilation failures. Then, we disassemble the binary. The result shows that some instructions have unrecognizable opcodes, indicating that the opcode might be remapped.

6. The remaining ones (`Unpyc37` and `Uncompyle2`) do not support 3.8.

Offset	Opcode	Arg
0	CALL_FUNCTION	1
2	STORE_ATTR	2
	...	
8	DELETE_GLOBAL	0
10	CALL_FUNCTION	2
12	<0>	
	...	
98	BEGIN_FINALLY	

(a) Opcode Remapped Program (Dropbox, Disassembled)

Offset	Opcode	Arg
0	LOAD_CONST	1
2	STORE_FAST	2
	...	
8	LOAD_GLOBAL	0
10	LOAD_CONST	2
12	CALL_FUNCTION	1
	...	
98	RETURN_VALUE	

(b) Original Program (Dropbox, Disassembled)

```
.*[#53]$
```

(c) Regular Expression Detecting Dropbox Opcode Remapped Binaries

Opcode mnemonic	Org. Opcode #	Modified Opcode #
RETURN_VALUE	83	53
BEGIN_FINALLY	53	83
CALL_FUNCTION	131	129
LOAD_CONST	100	131
LOAD_GLOBAL	116	98
STORE_FAST	125	95

(d) Opcode Remapping

Transformation Rule	[Original Opcode #] → [New Opcode #]
RETURN_VALUE	[#53]→[#83]
BEGIN_FINALLY	[#83]→[#53]
CALL_FUNCTION	[#129]→[#131]
LOAD_CONST	[#131]→[#100]
LOAD_GLOBAL	[#98]→[#116]
STORE_FAST	[#95]→[#125]

(e) Transformation Rules

Figure 9. Opcode Remapped Binary (Dropbox, Modified Python 3.8.12)

Extracting Python Opcode Remapping. To infer the opcode mapping between the original Python and the modified Python, we leverage the fact that (1) `Dropbox`'s modified Python is based on the 3.8.12 version and (2) they contain binary files of Python standard libraries. Hence, we obtain Python 3.8.12's standard libraries, compile them with a vanilla Python, and then compare the results with `Dropbox`'s binaries. We observe that whole bytecodes match except for their opcodes. For the `druva inSync` binaries, the `druva`'s Python environment provides `opcode.opmap` which we can compare with the original `opcode.opmap`. Note that `Dropbox` does not include the `opcode` module.

Handling Opcode Remapping in `Dropbox` (Python 3.8). Fig. 9-(a) shows a few lines of disassembled code of a `.pyc` binary from `Dropbox`. Note that Fig. 9-(b) shows the original program's disassembled code (for ground-truth). Observe that the instructions are abnormal. For example, it calls a function without specifying the target function (typically, `LOAD_GLOBAL` is used to specify the target function). Observe that at offset 12 it does not show a valid opcode mnemonic, meaning that the opcode number is not valid. Lastly, the function ends with `BEGIN_FINALLY` (at offset 98) while all normally compiled Python functions end with `RETURN_VALUE`. Hence, we use the regular expression in Fig. 9-(c), which finds a function ending with the opcode 53, to fingerprint a binary with the same opcode remapping.

Fig. 9-(d) shows how the opcodes are remapped. `RETURN_VALUE`'s opcode is changed from 83 to 53. `BEGIN_FINALLY`'s opcode is 83 in the `Dropbox` while it is 53 in a vanilla Python binary. Observe that `CALL_FUNCTION`'s opcode is changed from 131 to 129, while there is no opcode 129 in the vanilla Python. As a result, the disassembler fails at offset 12 in Fig. 9-(a).

Finally, Fig. 9-(e) shows a few transformation rules that change the remapped opcode back to the original opcode. For example, the first rule fixes the `RETURN_VALUE`'s opcode by changing the opcode 53 to 83 (i.e., `[#53]→[#83]`).

Offset	Opcode	Arg
0	SETUP_EXCEPT	48
3	<34>	
	...	
47	STORE_DEREF	255
50	POP_TOP	
51	LOAD_CONST	4
54	JUMP_ABSOLUTE	71
	...	

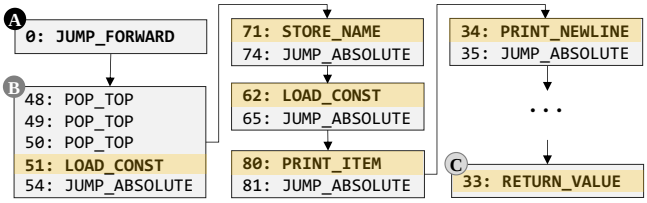
(a) **`PjOrion`** Obfuscated Program

Offset	Opcode	Arg
0	LOAD_CONST	4
3	STORE_NAME	2
6	LOAD_CONST	2
9	PRINT_ITEM	
10	PRINT_NEWLINE	
	...	
19	RETURN_VALUE	

(b) Original Program

```
^[SETUP_EXCEPT] → [JUMP_FORWARD]
```

(c) Transformation Rule for Updating Entry Point



(d) Control Flow Graph

```
^[JUMP_FORWARD].*[POP_TOP]+ → [].*[]
```

(e) Transformation Rule for Eliminating Entry Point Jump

```
[JUMP_FORWARD] → [], [JUMP_ABSOLUTE] → []
```
(Apply if the jump target instruction has *a single incoming edge*)

(f) Transformation Rules for Eliminating Unconditional Jumps

Figure 10. Control Flow Extraction for `PjOrion` Obfuscation.

Handling `inSync` Binaries (Python 2.7). Due to space, we present how we handle `inSync` binaries in [35] (Section 11.2).

5.4.3. PjOrion Obfuscated Binaries. `PjOrion` [43] is a popular advanced obfuscation technique against decompilers, leveraging four methods: (1) adding invalid instructions in binary, (2) adding exception blocks (using `SETUP_EXCEPT`), (3) hiding original opcodes in argument bytes, and (4) adding random jump instructions to shuffle the overall binary. In this section, we demonstrate how PYFET deobfuscates the `PjOrion` obfuscated binaries.

Decompilation Attempt. We use `Uncompyle6` to decompile a Python binary[7] obfuscated by `PjOrion`. The decompiler fails at the *invalid instruction* inserted by the obfuscator.

Initial Analysis. We manually analyze the binary and observe that the instructions are displaced. Specifically, the obfuscator exploits the multi-byte nature of binary to hide opcodes in bytes reserved for arguments. Fig. 10-(a) shows a few lines of disassembled obfuscated binary. We observe that (1) the binary starts with `SETUP_EXCEPT` followed by an invalid instruction (i.e., essentially raising an exception), (2) the `SETUP_EXCEPT` block ends at offset 48 that points to an argument of `STORE_DEREF` (i.e., hiding the next instruction to be executed in the argument of `STORE_DEREF` when an exception is raised), and (3) the first instruction of the original binary (offset 0 in Fig. 10-(b)) is at offset 51 of the obfuscated binary shown in Fig. 10-(a).

7. It is a simple program created ourselves and is a Python 2.7 binary as `PjOrion` only supports 2.7 binaries.

Extracting Control Flow. We first apply our rule shown in Fig. 10-(c), to eliminate the exception. We then use PYFET to generate a control flow graph. The graph is shown in Fig. 10-(d). Notice that A jumps to block B. In B we can see our first instruction at offset 51 following three `POP_TOP` instructions, two of which (offset 48 and 49) were initially hidden. Following the graph from B onwards, we can see the remaining instructions being revealed with eventually ending on C.

Applying FETs. We first apply `R3` from Table 4 to remove any unreachable code. Second, we apply Fig. 10-(e) to remove the entry point jump instruction and the corresponding `POP_TOP` instructions initially used by the previously eliminated `SETUP_EXCEPT`. We then use the rules in Fig. 10-(f) to remove the jump instructions added by `PjOrion`. Note that we only apply the rules when the jump target has a single incoming edge (e.g., instructions at 71, 62, 80, 34 and 33 to name a few). The rules essentially merge all the basic blocks in Fig. 10-(e) (from A to C). To this end, we successfully deobfuscate the `PjOrion` obfuscated binary.

6. Discussion

Source-level FET Presentation. We describe FETs at the source-level to ease the presentation, while these are *binary-level* transformations. Note that a source-level FET description means a specific implementation of the source (e.g., if a source-level FET can have multiple corresponding bytecodes, we mean one of those). Hence, all FETs in Table 4 and Table 9 mostly establishes 1:1 mapping between the source and binary representation, except for a few cases. For example, `while` in Python 3.7 has a `SETUP_LOOP` while 3.8's `while` does not have the instruction. On average, each example of FET is mapped to 1.07 binary code snippets.

Side-effects of FET. Since we derive FET rules from successfully decompiled programs (that are also free of implicit errors), FETs' transformations are safe for decompilers (i.e., they do not introduce known new errors). Regarding the possibility of FETs introducing errors in adjacent blocks, our results (in Section 5.2.2) show that the transformations do not have side effects apart from the expected semantic changes. We also did not observe such cases while experimenting with the tool (e.g., in Section 5.4).

Unhandled Decompilation Errors. There are errors that PYFET does not handle (i.e., out of the scope). First, if a decompiler does not terminate (hence there is no explicit error message), we do not handle the case. In practice, a decompiler might enter an infinite loop or an extremely slow execution path. In our paper, if decompilation does not finish within 5 minutes, we consider it does not terminate. While we rarely see such cases, one of such cases is presented in Section 5.4.2. Other examples of making decompilers not terminate are presented in Appendix 6.

Generality of PYFET. Many decompilation issues we observe are generic and can happen in other decompilers (e.g., C/C++ decompilers). For example, handling complex control flows and optimized binary code is a long-standing problem. Some decompilers may have implicit errors as well. To answer whether the technique can be expanded for other languages (e.g., C/C++), one can follow our experiments on decompilers. We leave it as future work.

7. Related Work

Binary Transformation/Rewriting. Dynamic binary translation techniques [23, 38, 49] are used in various tasks such as profiling and security analysis. They essentially conduct semantically equivalent transformations on machine instructions and add additional instructions for instrumentations. There are also binary rewriting/transformation techniques for optimizations [15, 53] and security [18, 20, 31, 34, 50, 51]. In particular, T-Fuzz [33] transforms (i.e., removes) conditional statements that are difficult to satisfy, increasing the coverage of fuzzing techniques. Tonder et al. [45] leverage program transformation (SET) at the source code level to improve the accuracy of static analysis techniques. PYFET differs from their work as our transformation is at the binary level and we use FET, not SET.

Decompilers. Decompilers [1, 11, 24] and decompilation techniques [12] are mainly for reverse-engineering. Popular decompilers such as Hex-Rays [22] and Radare2 [1] are built on top of disassemblers. They aim to recover the original source code representation from the given binary instructions, by identifying certain patterns of the instructions that match the high-level language constructs [13, 27, 32, 41].

Decompilation Errors. There is a line of research [6, 28, 44, 52] relating to testing, finding, and studying bugs in decompilers. [28] systematically tests popular C decompilers [24, 25] to evaluate the correctness. [6] analyze and discuss challenges and limitations of x86/64 disassembly techniques. While the specific challenges differ from Python decompilers, [6] acknowledge that decompilation errors can lead to misleading reverse-engineering processes.

8. Conclusion

We present PYFET that transforms a Python binary causing decompilation failures into another form without the errors, based on the novel concept of Forensically Equivalent Transformation. We evaluate our proof of concept system on 17,117 real-world Python malware samples in the wild, and PYFET successfully fixes 77,022 errors. Our case studies show that PYFET effectively solves various challenges, such as handling opcode remapping and obfuscation techniques. Moreover, we also use PYFET to migrate Python 3.9 binaries to 3.8 for decompilers that do not support the version.

Acknowledgment

We thank the anonymous referees for their constructive feedback and ReversingLabs for sharing the Python malware samples. The authors gratefully acknowledge the support of NSF (1916499, 1908021, 1850392, 1853374, 2210137, and 1924777), DARPA PA-20-02-07-FP-020, and ONR N00014-23-1-2095. This research was also supported by Science Alliance's StART program, and gifts from Google exploreCSR and Cisco Systems. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the sponsor.

References

[1] "Radare2," https://radare.org/r/, 2020.
[2] "cpython/Lib," https://github.com/python/cpython/tree/main/Lib, 2022.
[3] "Dropbox," https://www.dropbox.com/, 2022.
[4] "The Python Package Index," https://pypi.org/, 2022.
[5] M. Aguirre, "Protecting a Python codebase," https://bits.theoremone.co/protecting-a-python-codebase-part-3/, 2015.
[6] D. Andriesse, X. Chen, V. Van Der Veen, A. Slowinska, and H. Bos, "An in-depth analysis of disassembly on full-scale x86/x64 binaries," in *USENIX Security'16*.
[7] "Python Malware On The Rise, Cyborg Security," 2022, https://www.cyborgsecurity.com/cyborg_labs/python-malware-on-the-rise/.
[8] "REC Studio 4-Reverse Engineering Compiler," http://www.backerstreet.com/rec/rec.htm, 2015.
[9] "Binary Ninja," 2022, https://binary.ninja/.
[10] "Boomerang Decompiler," 2022, http://boomerang.sourceforge.net/index.php.
[11] "RetDec," 2022, https://github.com/avast/retdec.
[12] D. Brumley, J. Lee, E. J. Schwartz, and M. Woo, "Native x86 decompilation using Semantics-Preserving structural analysis and iterative Control-Flow structuring," in *USENIX Security'13*.
[13] Z. L. Chua, S. Shen, P. Saxena, and Z. Liang, "Neural nets can learn function type signatures from binaries," in *USENIX Security'17*.
[14] "PyInstaller," https://pyinstaller.org/, 2022.
[15] B. De Sutter, B. De Bus, and K. De Bosschere, "Link-time binary rewriting techniques for program compaction," *ACM Trans. Program. Lang. Syst.*, vol. 27, no. 5, p. 882–945, sep 2005.
[16] "Python decompiler for 3.7-3.8," 2022, https://github.com/rocky/python-decompile3.
[17] "Decompyle++," 2022, https://github.com/zrax/pycdc.
[18] S. Dinesh, N. Burow, D. Xu, and M. Payer, "Retrowrite: Statically instrumenting cots binaries for fuzzing and sanitization," in *SP'20*.
[19] "Druva inSync," https://www.druva.com/products/endpoints/, 2022.
[20] G. J. Duck, X. Gao, and A. Roychoudhury, "Binary rewriting without control flow recovery," in *PLDI'20*.
[21] "Ghidra," 2022, https://ghidra-sre.org/.
[22] I. Guilfanov, "Decompilers and beyond," *Black Hat USA*, vol. 9, p. 46, 2008.
[23] Intel, "Pin," https://www.intel.com/content/www/us/en/developer/articles/tool/pin-a-dynamic-binary-instrumentation-tool.html, 2022.
[24] "Hex Rays," 2022, https://www.hex-rays.com/ida-pro/.
[25] "JEB Decompiler by PNF Software," 2022, https://www.pnfsoftware.com/.
[26] D. Kholia and P. Wegrzyn, "Looking inside the (drop) box," in *WOOT'13*.
[27] J. Lee, T. Avgerinos, and D. Brumley, "TIE: principled reverse engineering of types in binary programs," in *NDSS'11*.
[28] Z. Liu and S. Wang, "How far we have come: Testing decompilation correctness of C decompilers," in *ISSTA'20*.
[29] "Malware Makers Using 'Exotic' Programming Languages," 2022, https://threatpost.com/malware-makers-using-exotic-programming-languages/.
[30] D. Merkel, "Docker: lightweight linux containers for consistent development and deployment," *Linux journal*, vol. 2014, no. 239, p. 2, 2014.
[31] K. Miller, Y. Kwon, Y. Sun, Z. Zhang, X. Zhang, and Z. Lin, "Probabilistic disassembly," in *ICSE'19*.
[32] M. Noonan, A. Loginov, and D. Cok, "Polymorphic type inference for machine code," *SIGPLAN Not.*, vol. 51, no. 6, p. 27–41, jun.
[33] H. Peng, Y. Shoshitaishvili, and M. Payer, "T-fuzz: Fuzzing by program transformation," in *IEEE S&P'18*.
[34] S. Priyadarshan, H. Nguyen, and R. Sekar, "Practical fine-grained binary code randomization," in *ACSAC'20*.
[35] "PyFET Supplementary Material," 2022, https://github.com/pyfet-pyc/src/blob/main/appx.pdf.
[36] "PyFET repository: Correctness and Impact of PyFET." 2022, https://github.com/pyfet-pyc/src/tree/main/Correctness_and_Impact_PyFET.
[37] "PyFET Repository," 2022, https://github.com/pyfet-pyc/src.
[38] "QEMU," https://www.qemu.org/, 2022.
[39] ReversingLabs, "Explainable Threat Intelligence, ReversingLabs," https://www.reversinglabs.com/, 2022.
[40] S. Sampath, "DisC-Decompiler for TurboC," https://www.debugmode.com/dcompile/disc.htm, 2001.
[41] E. J. Schwartz, C. F. Cohen, M. Duggan, J. Gennari, J. S. Havrilla, and C. Hines, "Using Logic Programming to Recover C++ Classes and Methods from Compiled Executables," in *CCS'18*.
[42] "Snowman," 2018, https://derevenets.com.
[43] "Project orion," 2014, https://kr.cm/f/t/15280/.
[44] C. Sun, V. Le, and Z. Su, "Finding compiler bugs via live code mutation," *SIGPLAN Not.*, vol. 51, no. 10, p. 849–863, oct.
[45] R. Tonder and C. Goues, "Tailoring programs for static analysis via program transformation," in *ICSE'20*.
[46] "Uncompyle2: A Python 2.7 byte-code decompiler," 2016, https://pypi.org/project/uncompyle2/.
[47] "Uncompyle6," 2022, https://github.com/rocky/python-uncompyle6.
[48] "Unpyc37: Decompiler for Python 3.7," 2022, https://github.com/greyblue9/unpyc37-3.10.
[49] Valgrind, "Valgrind," https://valgrind.org/, 2022.
[50] R. Wartell, V. Mohan, K. W. Hamlen, and Z. Lin, "Securing untrusted code via compiler-agnostic binary rewriting," in *ACSAC'12*.
[51] D. Williams-King, H. Kobayashi, K. Williams-King, G. Patterson, F. Spano, Y. J. Wu, J. Yang, and V. P. Kemerlis, "Egalito: Layout-agnostic binary recompilation," in *ASPLOS'20*.
[52] X. Yang, Y. Chen, E. Eide, and J. Regehr, "Finding and Understanding Bugs in C Compilers," *SIGPLAN*

Not., vol. 46, no. 6, p. 283–294, jun.

[53] X. Zhang, Q. Guo, Y. Chen, T. Chen, and W. Hu, "Hermes: A fast cross-isa binary translator with post-optimization," in *CGO'15*.

Appendix

1. Incorrectly Omitted Code

We inspect all samples in Section 5 for instances of incorrectly omitted code. We found 3,901 errors in 1,862 binaries across all Python versions. `Decompyle++` has the most number of such errors, with a total of 3,328 (85.3%). In terms of Python versions, Python 3.9 binaries suffer from this error most (0.25 errors per sample binary).

2. Implicit Errors

2.1. Implicit Error Patterns. As stated in Section 3 (**Deriving FETs**), we obtain implicit error patterns by analyzing the 375 samples' decompilation results. Specifically, we abstract the differences between their original source code and (incorrectly) decompiled source code to develop implicit error patterns and their correct pattern, as shown in Table 3 and Table 6 (showing the remaining implicit errors). Error patterns are obtained from the decompiled programs. Correct patterns are the error patterns' corresponding code from the original source code.

2.2. Implicit Error Detection's Outcomes. In Section 4.1.1, we present our implicit error detection process. This subsection further explains how we interpret the implicit error detection process's outcomes for each implicit error pattern found in decompiled binary. Table 7 shows four logically possible outcomes.

I. The first outcome case is that the input binary has $I_{correct}$ but does not have I_{error}. Note that the decompiled program has I_{error}. Hence, this means that the input program has a corrected version of the error code, meaning that the decompiled program has an implicit error. Intuitively, $I_{correct}$ is a solution of an implicit error. If the error does not exist, $I_{correct}$ should not exist in the input binary.

II. The input binary has I_{error} but does not have $I_{correct}$. This means the decompiled program and the input program have identical instructions (I_{error}). Hence no error is detected.

III. If none of them exist in the input binary, there can be two reasons. First, it may mean that the decompiled program is significantly different from the input program so we cannot make a meaningful comparison. Second, the compiler that we use to obtain $I_{correct}$ and I_{error} generated completely different code compared with the input binary. For both cases, we cannot trust the result; hence conservatively, we detect no errors.

IV. This result is impossible and may happen if our implementation of code matching has a bug. We detect no errors for this case, and we have not observed this case during our experiments.

TABLE 6. IMPLICIT ERROR PATTERNS (REMAINING).

Pattern Name	Implicit Error Pattern	Correct Pattern

P7: `if` removes `else`	`if c1:`
	`␣␣s1`
	`c2`
		`if c1:`
		`␣␣s1`
		`else: c2`

P8: Incorrectly removes `elif`	`if c1: s1`
	`else:`
	`␣␣if c2: s2`
	`if c3: s3`
		`if c1: s1`
		`elif c2: s2`
		`elif c3: s3`

P9: `for` incorrectly moves code outside	`if c1: s1`
	`else:`
	`␣␣for x in y:`
	`␣␣␣␣if c1: s1`
	`␣␣if c2: s2`
		`if c1: s1`
		`else:`
		`␣␣for x in y:`
		`␣␣␣␣if c1: s1`
		`␣␣␣␣if c2: s2`

P10: Code in `except` is moved in new `try` block	`except:`
	`␣␣try: s1`
	`␣␣finally: s2`
		`except:`
		`␣␣s1`

P11: `if` incorrectly adds `elif`	`if c1: s1` `elif c2: s2`	`if c1: s1` `if c2: s2`

P12: `if` converts `and` to nested `if`	`if c1:`
	`␣␣if c2:`
	`␣␣␣␣s1`
	`elif c3: s2`
		`if c1 and c2:`
		`␣␣s1`
		`elif c3: s2`

P13: `if` incorrectly adds `else` after `raise`	`if c1:`
	`␣␣s1`
	`␣␣raise x`
	`else: s2`
		`if c1:`
		`␣␣s1`
		`␣␣raise x`
		`s2`

P14: Incorrect indentation after `return`	`if c1:`
	`␣␣s1`
	`␣␣return s2`
	`␣␣s3`
		`if c1:`
		`␣␣s1`
		`␣␣return s2`
		`s3`

P15: `while` moves loop condition	`while c1:`
	`␣␣s1 or c2`
		`while c1\`
		`␣and not c2:`
		`␣␣s1`

P16: `if` incorrectly adds `else` inside `with`	`with x:`
	`␣␣if c1: s1`
	`␣␣else: s2`
		`with x:`
		`␣␣if c1: s1`
		`␣␣s2`

P17: `while` is converted to `if` with `False`	`if False:` `␣␣s1`	`while False:` `␣␣s1`

P18: `if` incorrectly introduces `else`	`if c1:`
	`␣␣s1`
	`else: s2`
		`if c1:`
		`␣␣s1`
		`s2`

P19: Duplicated `raise` outside `if`	`if c1:`
	`␣␣raise s1`
	`␣␣return s2`
	`raise s1`
		`if c1:`
		`␣␣raise s1`
		`␣␣return s2`

P20: Incorrect `or` statement	`if not c1: pass` `x = c2`	`x = c1 or c2`
P21: Incorrect `import` statement	`x = x` `import y`	`from x import y`

* Blue text represents keywords and highlighted lines show the differences between the error and correct code.

Note that the outcomes of III and IV can happen if (1) the implicit error patterns are not correct in the first place, (2) our correct patterns were wrong (e.g., error pattern and correct pattern both produce the same bytecode instructions), or (3) the compiler generates different instructions compared to the original input binary's instructions (e.g., we use a

TABLE 7. INPUT BINARY SEARCHING OUTCOMES.

Outcome	$I_{correct}$	I_{error}	Implicit Error Detected?
I	✔	✖	Error Detected
II	✖	✔	No Error
III	✖	✖	No Error (Result Not Trustable)
IV	✔	✔	No Error (Result Not Trustable)

* ✔: Found in the input binary, ✖: Not found in the input binary.

TABLE 8. UNIQUE FET RULES BREAKDOWN.

FET Rule	Python Version and Decompiler																
	3.9			3.8			3.7				3.6		3.5	3.4	2.7		
	U*	D*	D+	U	D	D+	U	D	D+	U3	U	D+	D+	U	D+	U	D+ U2
R1	☑	☑	☑	☑	☑	☑	☑	☑	☑	☑	☑	☑	☑	☑	☐	☑	☐ ☐
R2	☑	☐	☐	☑	☐	☐	☑	☐	☐	☑	☑	☐	☐	☑	☐	☐	☑ ☐
R3	☑	☑	☐	☑	☑	☐	☐	☐	☐	☐	☐	☐	☐	☑	☐	☐	☐ ☐
R4	☑	☑	☑	☐	☐	☑	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐ ☑
R5	☑	☑	☐	☑	☑	☐	☐	☐	☑	☐	☑	☑	☑	☐	☐	☐	☐ ☐
R6	☑	☐	☐	☑	☑	☑	☐	☑	☑	☑	☑	☑	☐	☐	☐	☐	☐ ☑
R7	☑	☑	☐	☑	☑	☑	☑	☑	☐	☑	☐	☐	☐	☐	☐	☐	☐ ☐
R8	☑	☑	☐	☑	☑	☐	☑	☑	☐	☑	☑	☐	☐	☑	☐	☐	☐ ☑
R9	☐	☐	☑	☐	☐	☑	☑	☑	☑	☑	☐	☑	☐	☐	☑	☐	☐ ☐
R10	☐	☐	☐	☑	☑	☑	☐	☑	☐	☐	☐	☐	☐	☐	☐	☐	☐ ☐
R11	☐	☐	☑	☑	☑	☑	☐	☐	☑	☐	☑	☑	☑	☐	☑	☐	☐ ☐
R12	☐	☐	☐	☑	☑	☐	☑	☑	☐	☑	☐	☐	☐	☐	☐	☐	☐ ☐
R13	☐	☐	☑	☐	☐	☑	☐	☐	☑	☐	☐	☑	☑	☐	☑	☐	☑ ☐
R14	☐	☐	☑	☐	☐	☑	☐	☐	☑	☐	☐	☑	☑	☐	☐	☐	☐ ☐
R15	☐	☐	☑	☐	☐	☑	☐	☐	☑	☐	☐	☑	☑	☐	☑	☐	☑ ☐
R16	☐	☐	☑	☐	☐	☑	☐	☐	☑	☐	☐	☑	☑	☐	☑	☐	☑ ☐
R17	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☑	☐ ☐
R18~30	☑	☑	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐ ☐

☑: FET rule is used at least once. U: `Uncompyle6`. D: `Decompyle3`.
D+: `Decompyle++`. U3: `Unpyc37`. U2: `Uncompyle2`. *: Migrated binaries.

different compiler since the input binary version header is wrong). Observe that we look for inconsistency between the two binaries where if both S_{error} (from decompiled code) and input binary have the error pattern code, it means there are no implicit errors since the two binaries are consistent.

3. Additional Details of FET Rules

3.1. Complete List of FET Rules. Table 9 shows the remaining FET rules used in our experiments except for the Python 3.9 to 3.8 migration which we present on our website [35] (Section 10.1). All rules for the migration are SET (semantically equivalent transformation).

3.2. FET Rules across Python Versions. Table 8 shows the FETs used by PYFET for the respective Python version. Observe that FETs used for 3.9 are similar to those for 3.8.

4. Additional Details for Evaluation

4.1. Decompilation Errors by Each Decompiler. We observe that there are different samples causing decompilation errors on different decompilers as shown in Fig. 11. Specifically, for `Uncompyle6`, `Decompyle3`, `Uncompyle2`, and `Unpyc37`, samples exclusively causing decompilation errors are 1,302, 715, 1,567, and 1,288, respectively. Meanwhile, 17,117 samples cause decompilation failures on `Decompyle++` only.

4.2. Application Selection Criteria for Section 5.2.1. We download top 100 Python projects based on popularity (# of stars) from GitHub (out of 2M+) for our ground truth study (14,949 Python source files). More details about 100 Python applications are on [35] (Section 9.2). We select the top 10 applications in terms of popularity and those have *at least 10 SLOC per function* and *more than 100 functions* (to prune out popular but trivial projects). Each sample is recompiled on the same Python versions used in Section 5.1 (i.e., 2.7, and 3.5~3.9). For each of the recompiled binary, we run the five decompilers to extract errors and eventually apply PYFET. We then filter out samples based on FETs applied such that we have 40 samples for each of the FET rules from `R1`~`R30` (1,200 samples). For rules we do not find sufficient candidates, we manually inject errors in randomly selected samples.

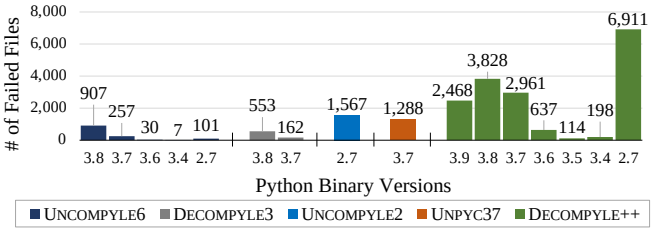


Figure 11. Dataset Breakdowns by Decompilers.

Selected Samples. On average, the evaluated samples have 11.3 functions, 8.8 conditionals (`if` conditions), 2.9 loops, 1.6 `with` blocks, and 0.6 `try-except` blocks per file.

5. Extended Case Studies

5.1. Manually Patching Decompilers vs PYFET. We manually debug three decompilers to resolve decompilation failures in order to demonstrate the effort required to fix the source code of decompilers. We choose `Uncompyle6`[8], `Unpyc37`, and `Decompyle++` as each of them has a unique design and implementation. Details of the chosen decompilers are on [35] (Section 9.3).

```
1 def error_1a(...):
2     return c1 or c2 and not c3
```

(a) Error Inducing Code

```
expr ::= LOAD_GLOBAL
expr ::= unary_not
unary_not ::= expr UNARY_NOT
ret_expr_or_cond ::= ret_expr
ret_and ::= expr JUMP_IF_FALSE_OR_POP
            ret_expr_or_cond COME_FROM
ret_or  ::= expr JUMP_IF_TRUE_OR_POP
            ret_expr_or_cond COME_FROM
ret_expr ::= expr
ret_expr ::= ret_and
ret_expr ::= ret_or
return ::= ret_expr RETURN_VALUE
sstmt ::= return RETURN_LAST
```

(b) Relevant Grammar in `Uncompyle6`

```
ret_expr  ::= ret_and_a
or        ::= expr POP_JUMP_IF_TRUE
              expr
and       ::= or JUMP_IF_FALSE_OR_POP
              COME_FROM expr
ret_and_a ::= and COME_FROM
```

(c) Corrected Grammar

```
3 def error_1a(...):
4     return c1 or c2 and not c3
```

(d) Decompiled Output with Corrected Grammar (Fixed)

```
5 def error_1b(...):
6     return c0 or c1 or c2 and
              not c3
```

(e) Error Inducing Input for Fixed Grammar (Implicit Error in (f))

```
7 def error_1b(...):
8     return c1 or c2 and
              not c3
```

(f) Incorrectly Decompiled Code ('`c0 or`' is omitted)

Figure 12. Decompilation Failure and Fixes for `Uncompyle6`

8. Since the design for `Uncompyle6`, `Decompyle3`, and `Uncompyle2` is similar, we choose `Uncompyle6` as it supports diverse Python versions.

TABLE 9. PYFET'S TRANSFORMATION RULES (REMAINING).

	Name	Original Stmt.	Transformation	Description

R7	Detaching Control Flow Changing Stmt. in `try/except` Block	`try: continue`
		`except: ...`
			`try: FET_continue = 1`
			`except: ...`
			`if FET_continue: continue`
				`try/except` statement in Python runs the code inside an exception handler. Control flow changing statements (e.g., `continue/break/return`) result in complicated code, leading to decompilation failures.

R8	Eliminating Control Flow Changes in Loop	`while x:`
		`  ...`
		`...`
			`while x:`
			`  ...`
			`  FET_null()`
			`...`
				Python compiler adds optimized jumps in between loops when part of the loop is only executed. These optimizations make it difficult for decompiler to parse. Adding nop instructions at the end of the loop fixes this.

R9	Extracting Logical Expressions from `while` Loop	`while x:`
		`  ...`
			`FET_cond = x`
			`while FET_cond:`
			`  ...`
			`  FET_cond = x`
				Having long logical expressions in `while` loop leads to complex loop structure. Separating the long logical expression from the loop makes it simpler for the decompiler to parse.

R10	Simplifying Control Flow Structures for `else` Block	`try: s1`
		`except: s2`
		`else: s3`
			`try: s1; FET_else = 1`
			`except: s2`
			`else: FET_null()`
			`if FET_else: s3`
				Having complex statements in `else` of any control flow block leads to complex structure. Separating those statements into separate conditional simplifies it for decompiler to parse since it will leverage grammar for conditionals instead.

R11	Eliminating Unpacking Variables	`class x(*arg,\` `  **kwargs)`	`class x(FET_one_star_arg,\` `  FET_two_star_kwargs)`	Unpacking variables in arguments under different contexts may not be supported by decompilers.
R12	Simplifying `Import`	`from x import\` `(a as b, c as d)`	`from x import a as b` `from x import c as d`	Chained imports in tuples lead to complex compiled code that may not be supported by decompilers.
R13	Datastructure initialization	`{x, y, z}`	`FET_set(x, y, z)`	Datastructure initializations (e.g., sets or dictionary) definition may not be supported by the decompiler.
R14	Expanding `yield` Stmt.	`yield from z`	`for i in z: yield i`	`yield from` may not be supported by the decompiler.
R15	Dictionary to List	`x = {a:b for \` `a,b in c}`	`t = [(a,b) for a, b in c]` `x = dict(t)`	Dictionary comprehension may not be supported by the decompiler.

R16	Separating Control Structure	`if x: s1`
		`for i in y: s2`
			`if x: s1`
			`FET_null()`
			`for i in y: s2`
				Having consective control structures can lead to difficulty in parsing especially when decompiler may perceive two control structures as one. Adding nop instructions fixes this.

R17	Converting `assert`'s Implementation	`assert x`	`assert x`	`assert`'s implementation without `CALL_FUNCTION` causes an error. Hence, we instrument `CALL_FUNCTION` in binary.

* Blue and red represent `keywords` and `transformed code` by FET.

Debugging `Uncompyle6`. We debug `Uncompyle6` to fix the decompilation error caused by the code shown in Fig. 12-(a).

1) Identifying Root Cause: We first run the decompiler with debugging flags "`-agT`" to trace back the root cause. Since `Uncompyle6` leverages the `SPARK` parser with 165 grammar rules, we dump the grammar rules associated with the error. From the error message and logs, we shortlist 11 grammar rules (out of 770) shown in Fig. 12-(b) that are related to boolean operators with `return`. Note that this process required substantial analysis and understanding of the decompiler's internals. For instance, we went through 60 functions, and five different files (6,520 SLOC in total) to shortlist the above rules, taking 7 hours of an author.

We begin with slightly modifying the shortlisted 11 rules to locate the problematic rules. Specifically, since the error message does not provide any detailed information on the failed term/grammar, we use '?' on each term to make it optional in the grammar as shown in Fig. 13-(a). As it essentially skips the term, we are trying to skip the problematic term. This process includes inspecting the parsing grammar used and reasoning the outcomes with the modified grammar rules. Doing so on all 11 rules took about 3~4 hours (each rule taking 20~30 minutes). After analyzing the results (i.e., inspecting rules used frequently near the error), we find that '`ret_expr`' is likely a problematic rule. We then start to modify the rule to narrow down terms causing the error by unfolding the rule and playing with each term. Note that changing a term in a rule often breaks the decompiler. After a handful of failed trials, Fig. 13-(b) shows the first successful trial of getting a source code output. However, the output misses the majority of the code.

As directly modifying `ret_expr` is failed, we try to extend `ret_and` by adding a new rule `ret_and_a` as shown in Fig. 13-(c). We change `unary_not` to `expr` so that the expression after `c1` can be parsed. The output now includes `c1`, `c2`, and `c3`. However, it misses `or` and `and` operators. We then attempt to break down the grammar in Fig. 13-(d) and Fig. 13-(e) to add '`or`' and '`and`' respectively. However, observe that Fig. 13-(e) misses `c2` and `c3` while recovering the operators. To recover the two operators in the same expression correctly, we had to inspect 10 additional rules and their contexts to craft them, taking more than 8 hours. Fig. 13-(f) shows a failed attempt that we tried to extend from step 4. Note that the entire process consisted of unfolding the initially successful grammar rule we identified in Fig. 12-(b). We finally craft a working rule shown in Fig. 12-(c) that results in a correct decompilation output. Note that the entire debugging process requires us to deeply understand the decompiler including classes/data structures

used and grammar rules. It took more than two days in total.

```
ret_and   ::= expr JUMP_IF_FALSE_OR_POP
              ret_expr_or_cond?
              COME_FROM
```
→ 1 | Parse error at or near 'None' instruction at offset -1

(a) Attempt 1 (Experimentally modifying existing rules)

```
ret_expr  ::= expr POP_JUMP_IF_TRUE
              expr JUMP_IF_FALSE_OR_POP
              COME_FROM unary_not
              COME_FROM
```
→ 2 | return c1

(b) Attempt 2 (Crafting new stand alone rule from analyzed bytecode)

```
ret_expr  ::= ret_and_a
ret_and_a ::= expr POP_JUMP_IF_TRUE
              expr JUMP_IF_FALSE_OR_POP
              COME_FROM expr COME_FROM
```
→ 3 | return c1 c2 (not c3)

(c) Attempt 3 (Relaxing rule to recover 'c1', 'c2', and, 'c3')

```
ret_expr  ::= ret_and_a
or        ::= expr POP_JUMP_IF_TRUE
              expr JUMP_IF_FALSE_OR_POP
              COME_FROM
ret_and_a ::= or expr COME_FROM
```
→ 4 | return c1 or (not c3)

(d) Attempt 4 (Adding rule to recover 'or')

```
ret_expr  ::= ret_and_a
or        ::= expr POP_JUMP_IF_TRUE
              expr JUMP_IF_FALSE_OR_POP
              COME_FROM
and       ::= or expr COME_FROM
ret_and_a ::= and
```
→ 5 | return c1 or and

(e) Attempt 5 (Adding rule to recover 'and')

```
ret_expr  ::= ret_and_a
or        ::= expr POP_JUMP_IF_TRUE
              expr
ret_and_a ::= or JUMP_IF_FALSE_OR_POP
              COME_FROM expr COME_FROM
```
→ 6 | return c1 or c2 \ (not c3)

(f) Attempt 6 (Relaxing rule further to recover 'c1', 'c2', and, 'c3')

Figure 13. Grammar Rules Failed in Fixing `Uncompyle6`.

2) New Bugs due to the Fix: We find that our fix causes multiple side-effects, leading to new bugs. For instance, it causes an implicit error as shown in Fig. 12-(f), where "`c0 or`" in the original code (Fig. 12-(e)) is incorrectly removed. We found this error after running 10 mutated test cases.

Debugging `Unpyc37` and `Decompyle++`. We debug `Unpyc37` and `Decompyle++` to fix decompilation errors in conditional blocks and exception blocks respectively. For the two decompilers we take upto 10 hours to fix. We outline the details for both of them in [35] (Section 10.2).

Fixing the Errors via PYFET. For all errors, PYFET fixes the errors by transforming less than or equal to 2 lines of source code (examples are presented on [35] in Section 10.3). Coming up with each FET takes 2.5 hours on average, including the time to analyze the error, craft the rules, and test on multiple examples. PYFET does not require one to understand and change the decompiler's internals.

PYFET's fixes essentially try to avoid the decompilers exercising the buggy rules or code. Specifically, we analyze how `Uncompyle6` works after the PYFET's transformation. We notice that it no longer uses the buggy rule, handling `JUMP_IF_FALSE_OR_POP`. We also find that there is another very similar rule handling `JUMP_IF_TRUE_OR_POP`, and by comparing the two rules, we notice the buggy rule is missing an `expr` term (which we added in our solution shown in Fig. 12-(c)'s rule '`and`'). Similarly, we analyze the execution of `Unpyc37` after PYFET's transformation and observe that the buggy parts of the code (lines 2509~2519 for `POP_JUMP_IF` function) are not executed anymore. Meanwhile, for `Decompyle++`, we observe that the buggy parts of code (i.e., extra invocation of `bc_next` function) impact only our instrumented null instruction (`FET_null()`) and not the original code (originally `SETUP_EXCEPT`).

6. Additional Discussion

Examples of Unterminated Decompilation Process. We see decompilers do not terminate even after 5 minutes, when the input binary has a particular data structure, such as a map or dictionary with many elements (e.g., more than 10,000 elements). Our manual inspection shows that the decompilers use a recursive logic to parse the consecutive elements in the map and dictionary. A potential solution is to split the large data structure into multiple small chunks.

Supporting New/Evolving Python Versions. As Python evolves, its opcode definitions change (e.g., changing/adding instruction opcodes). Since our FET rules (except for the opcode remapping rules only used in Section 5.4.2) are based on opcode mnemonics, changes in opcode numbers do not affect PYFET. Removed instructions do not significantly affect PYFET because compilers typically leverage other existing instructions to implement the removed instructions, where a working FET can be derived similarly. Handling new instructions is challenging. However, as shown in the Python 3.9 case study (Section 5.4.1), it can be done with a reasonable manual effort. Similarly, attackers may come up with new patterns that do not exist in PYFET. It can be handled by analyzing the obfuscated binary and crafting new rules, as shown in Section 5.4.3.

Intention of Error-inducing Instructions. While we find many malware samples causing decompilation errors, it does not necessarily mean that all the errors are intentionally or maliciously injected.

Limitations of Implicit Error Detection. In Section 4.1.2, we explain how we detect implicit errors. It is important to mention that our implicit error detection may have false negatives, while we are unlikely to suffer from false positives. This is because (1) our implicit error detection is based on the patterns obtained via our empirical observations, and (2) we use strict pattern matching rules to detect implicit error pattern code, leaving a small room for wrongly detecting and transforming code. Observe that all the detected implicit errors are correctly fixed (we manually verified all the fixes) in our evaluation, meaning that we did not observe any false-positive cases. There are two major sources of false negatives: (1) our implicit error patterns are incomplete and (2) our transformation has implementation bugs.

Obfuscation Techniques. We show in Section 5.4.2, Section 5.4.3 and [35] (Section 11.2) how PYFET supports various obfuscation techniques. From our experience, since an obfuscator is essentially a post-processing technique for a compiler, our approach is suitable for handling obfuscated binaries (i.e., PYFET is essentially a pre-processor for a decompiler). While handling new anti-analysis techniques requires new FET definitions, we believe our design is generic to handle them. Our approach prevents tinkering the decompiler to handle obfuscations that can potentially break overall parsing rules within the decompiler.