PYFET: Forensically Equivalent Transformation for Python Binary Decompilation

Ali Ahad*, Chijung Jung*, Ammar Askar†,
Doowon Kim‡, Taesoo Kim†, Yonghwi Kwon*

*{aliahad, cj5kd, yongkwon}@virginia.edu  †{aaskar, taesoo}@gatech.ac.kr  ‡doowon@utk.edu
Python Malware on the Rise

Growth of major programming languages
Based on Stack Overflow question views in World Bank high-income countries

Python malware is using a devious new technique
By Sead Fadljić published December 19, 2022
Crooks are using new tricks to keep their payloads hidden

PyPI malware creators are starting to employ Anti-Debug techniques
By Andrey Polkovnychenko | December 13, 2022
SHARE: 1 0 0

The JFrog Security Research team continuously monitors popular open-source software (OSS) repositories with our automated tools, and reports any vulnerabilities or malicious packages discovered to repository maintainers and the wider community.

Malicious Python Trojan Impersonates SentinelOne Security Client
A fully-functional SentinelOne client is actually a Trojan horse that hides malicious code within it was found lurking in the Python Package Index repository ecosystem.

Robert Lemos
Contributing Writer, Dark Reading
December 16, 2022
Python Bytecode, Decompilers, and Failures

- Python **malware** is compiled to Bytecode: Challenging to analyze.
- Decompilers come to the rescue!
- But they fail on many binaries.

![Depicting decompilation issues](image)
Challenges in Handling Decompilation Errors

1. **Asymmetric Warfare**: Decompilers are easy to break but hard to fix.
   - **Debugging** decompilers requires substantial expertise and effort.

   ![Diagram](image)

   - >2K Parsing Rules
   - >93K SLOC
   - >1.7K Functions

   Uncompyle6
Challenges in Handling Decompilation Errors

1. **Asymmetric Warfare:** Decompilers are easy to break but hard to fix.
   - Debugging decompilers **requires substantial expertise and effort.**
Challenges in Handling Decompilation Errors

1. **Asymmetric Warfare**: Decompilers are easy to break but hard to fix.
   - Debugging decompilers *requires substantial expertise* and *effort*.

2. **Multiple decompilers**: Not scalable to debug all.

---

<table>
<thead>
<tr>
<th>Graphic</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image.png" alt="Decompilers" /></td>
<td><img src="image.png" alt="Uses Grammar Parsing Rules" /> <img src="image.png" alt="Implemented in Python" /> <img src="image.png" alt="Uses Code Logic" /> <img src="image.png" alt="Implemented in C++" /> Uncompyle6 Unpyc37 Decompyle++</td>
</tr>
</tbody>
</table>
Challenges in Handling Decompilation Errors

1. **Asymmetric Warfare:** Decompilers are easy to break but hard to fix.
   - Debugging decompilers **requires substantial expertise** and **effort**.

2. **Multiple decompilers:** Not scalable to debug all.

   Fixing the **failure inducing file**, not decompilers.
Identifying Decompilation Failure
Detecting a typical Decompilation failure

• Decompilation fails with error messages.
• Specifies approximate functions and offset of failure.

**Explicit Error:** Decompilation failure with an explicit error message
Decompilers also **fail silently** (Implicit Error)!

- Decompiled successfully but generating a **wrong** code.

**Implicit Error**: Decompilation failure without any error message.

*Original Program (Ground truth)*

```python
def decrypt(data, key, block_size=8, key_size=16, num_rounds=32, padding=chr(0)):
    data = base64.b64decode(data)
    blocks = [
        data[chunk * block_size:((chunk + 1) * block_size)]
        for chunk in range(len(data) // block_size)
    ]
    vector = blocks[0]
    result = []
    for block in blocks[1:]:
        v0, v1 = struct.unpack("!2L", block)
        k0 = struct.unpack("!4L", key[0:4])
        delta, mask = 0x8e3779b9, 0xffffffff
        sum = (delta + num_rounds) & mask
        if num_rounds > 0:
            v1 = (v1 - ((v0 << 4 ^ v0 >> 5) + v0) & sum) & mask
            v0 = (v0 - ((v1 << 4 ^ v1 >> 5) + v1) & sum - k0(sum & 0xf)) & mask
        else:
            v0 = struct.pack("!2L", v0, v1)
        output = str().join(chr(ord(x) ^ ord(y)) for x, y in zip(vector, decode))
        vector = block
        result.append(output)
    return str().join(result).rstrip(padding)
```

*Decompiled Program with Implicit Error (the ‘else’ blocks)*

```python
def decrypt(data, key, block_size=8, key_size=16, num_rounds=32, padding=chr(0)):
    data = base64.b64decode(data)
    blocks = [
        data[chunk * block_size:((chunk + 1) * block_size)]
        for chunk in range(len(data) // block_size)
    ]
    vector = blocks[0]
    result = []
    for block in blocks[1:]:
        v0, v1 = struct.unpack("!2L", block)
        k0 = struct.unpack("!4L", key[0:4])
        delta, mask = 0x8e3779b9, 0xffffffff
        sum = (delta + num_rounds) & mask
        if num_rounds > 0:
            v1 = (v1 - ((v0 << 4 ^ v0 >> 5) + v0) & sum) & mask
            v0 = (v0 - ((v1 << 4 ^ v1 >> 5) + v1) & sum - k0(sum & 0xf)) & mask
        else:
            v0 = struct.pack("!2L", v0, v1)
        output = str().join(chr(ord(x) ^ ord(y)) for x, y in zip(vector, decode))
        vector = block
        result.append(output)
    return str().join(result).rstrip(padding)
```
Identifying (a few) Implicit Error Patterns

Correct Code Pattern

```
if c1:
    s1
    return s2
else:
    s2
```

Implicit Error

```
if c1:
    if c2:
        s1
    else:
        s2
else:
    s3
```

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

```
for x in y:
    if c1:
        s1
        s2
    s3
```

```
while block removed, new 'else:' introduced.
```

```
for x in y:
    if c1:
        s1
        s2
    s3
```

```
else: block wrongly coupled
```

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

```
new 'else:' introduced
```

```
for x in y:
    if c1:
        s1
        s2
    s3
```

```
while block removed, new 'else:' introduced.
```
Detecting: Implicit Errors with the Patterns

Correct Code

```
if c1:
    s1
return s2
```

Implicit Error

```
if c1:
    s1
    return s2
```

If the corrected instructions found at the same location, Implicit Error!

(If this is not an implicit error, the corrected/mutated code should not exist)

Detected 22,359 Implicit Errors from 5 Decompilers
Identifying Decompilation Failure
How to transform?

Identifying Decompilation Failure → Applying Transformation → Successfully Decompilable Binary
Fixing: Forensically Equivalent Transformation (FET)

- Forensically Equivalent Transformation
  - Careful extension of Semantically Equivalent Transformation
  - Preserving *forensically meaningful* semantics (manually defined)

Original Code

```python
with open("file", "r") as file:
    return file.read()
```

Forensically Equivalent Transformation

```python
with open("file", "r") as file:
    FET_return = file.read()
return FET_return
```

Semantically Equivalent Transformation

```python
try:
    file = open("file", "r")
    return file.read()
except:
    # default handler
finally:
    file.close()
```

Failure Condition: ‘return’ in ‘try-except’
### Fixing: (a few) Transformation Rules

<table>
<thead>
<tr>
<th>Original Code</th>
<th>Dividing logical expressions</th>
<th>Transformed Code</th>
<th>Reason</th>
</tr>
</thead>
<tbody>
<tr>
<td>(x or y) and z</td>
<td></td>
<td>t = x or y</td>
<td>Parsers fails to understand chains of logical expressions.</td>
</tr>
<tr>
<td>if c1:</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>s1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>elif c2:</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>s2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>f = lambda x: x*2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>while c1:</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>s1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>break</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>s2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reason</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Parser fails to parse chains of conditionals.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Language features like ‘lambda’ are not supported.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Parser fails to handle multiple consecutive control flow changes (e.g., loop back + break).</td>
</tr>
</tbody>
</table>
Applying Transformation

Identifying Decompilation Failure

Where to transform?

Successfully Decompilable Binary

Applying Transformation
Fixing: Iterative Transformation Process

- Based on Control Flow Graph of the program.
- Starting from the detected/reported error: e.g., 638: JUMP_FORWARD

1. Apply Transformation Rules at 638
2. Extend targets:
   All directly reachable nodes from 638.
3. Extend targets:
   All directly reachable nodes from already covered nodes.

Explores only 33% of the nodes on average
Resolving Real-world Decompilation Failures

- **38,351** real-world malware samples from ReversingLabs.
- **17,117** (44.6%) malware samples failed to be decompiled.
  - Proportionally **3.9 (81%)** failed most.
  - Followed by **3.8 (66%)** and **3.7 (47%)**

We resolved decompilation failures of **all** the malware samples (17,117)!
Case Study: Opcode Remapped & Obfuscated Binary

Python Source Code → Customized Python Compiler → Opcode Remapped Binary (not decompilable)

Python Binary → Obfuscator → Obfuscated Binary

We resolve all! More detail in paper.
Summary

- 17K Malware binaries’ decompilation errors resolved.
- 5 Different **Python Decompilers** handled. (Uncompyle6, Decompyle3, Uncompyle2, Uncompile3, Decompyle++)
- 3 Opcode remapping and **obfuscated** binaries handled. (DropBox and druva)
- 30 Transformation rules developed.