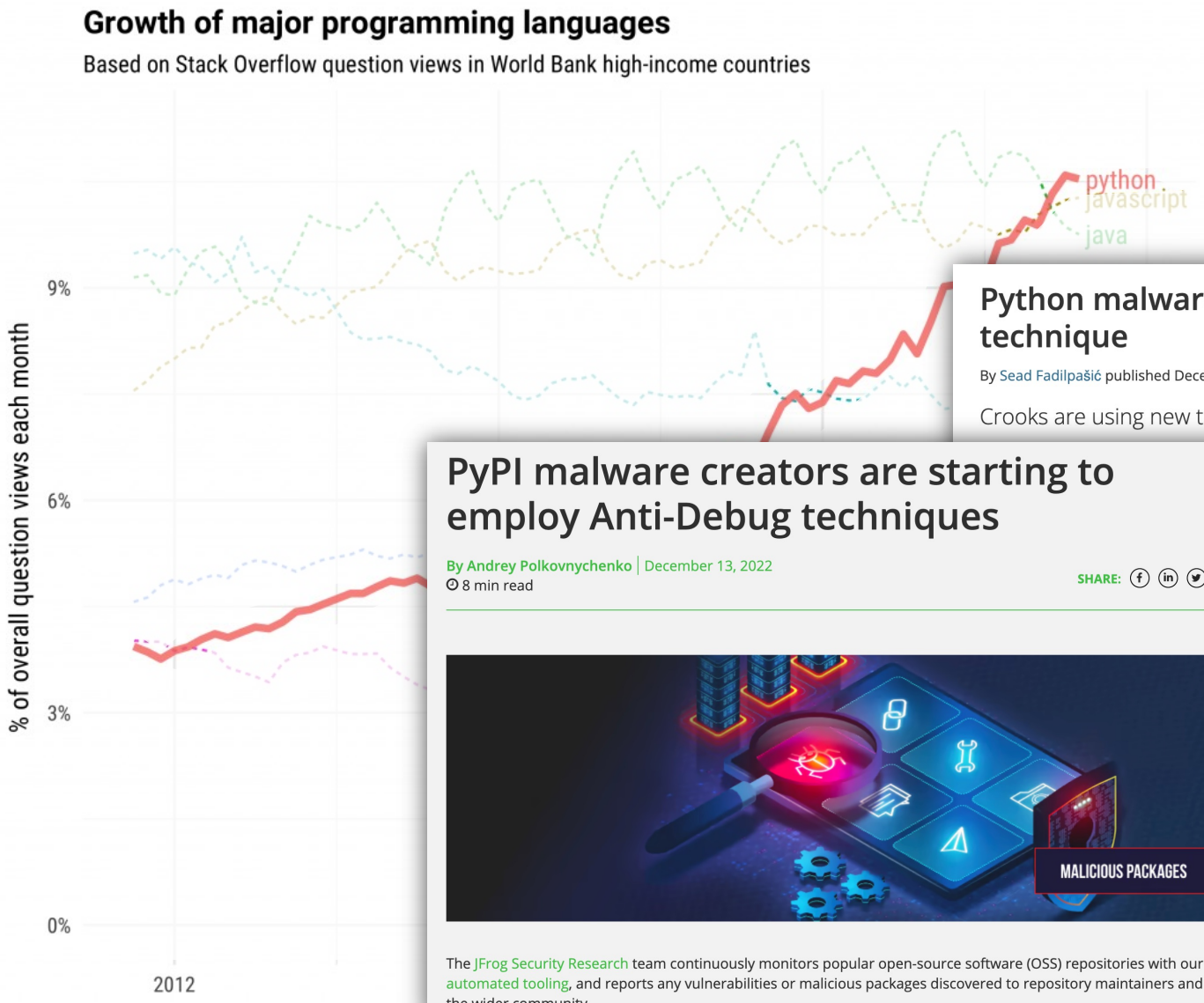
44th IEEE Symposium on Security and Privacy

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



Python Malware On The Rise

Cyborg Labs | July 14, 2020

serious malware over the past 30 years has been written in Assembly or
such as C, C++, and Delphi. However, ever-increasing over the past decade, a
ware has been written in interpreted languages, such as Python. The low barrier
rapid development process, and massive library collection has made Python

Python malware is using a devious new technique

By Sead Fadilpašić 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

8 min read

SHARE:

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

Malicious Python Trojan Impersonates SentinelOne Security Client

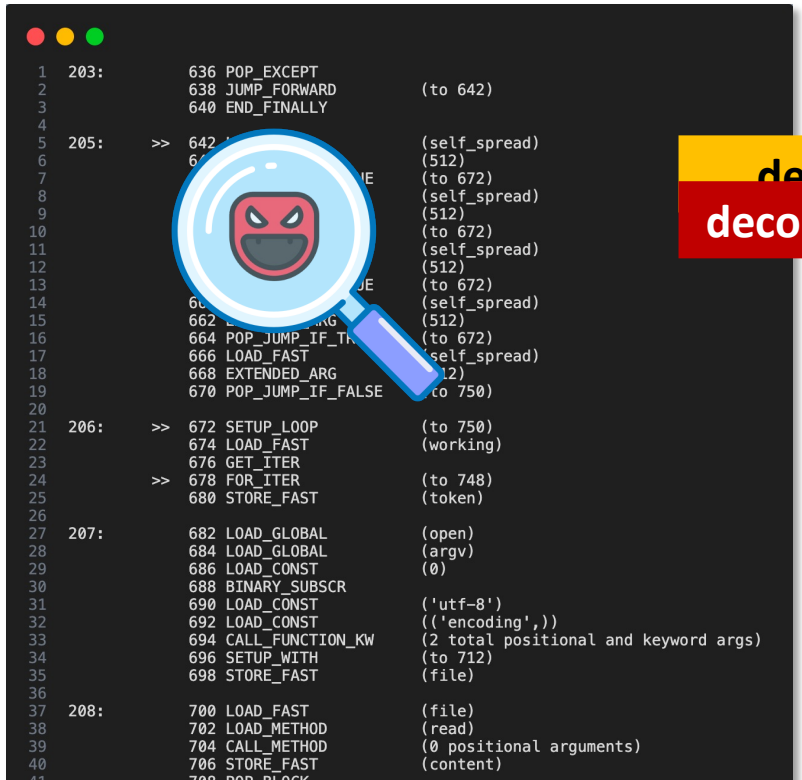
A fully functional SentinelOne client is actually a Trojan horse that hides malicious code within; it was found lurking in the Python Package Index repository ecosystem.

Robert Lemos
Contributing Writer, Dark Reading

December 19, 2022

Python Bytecode, Decompilers, and **Failures**

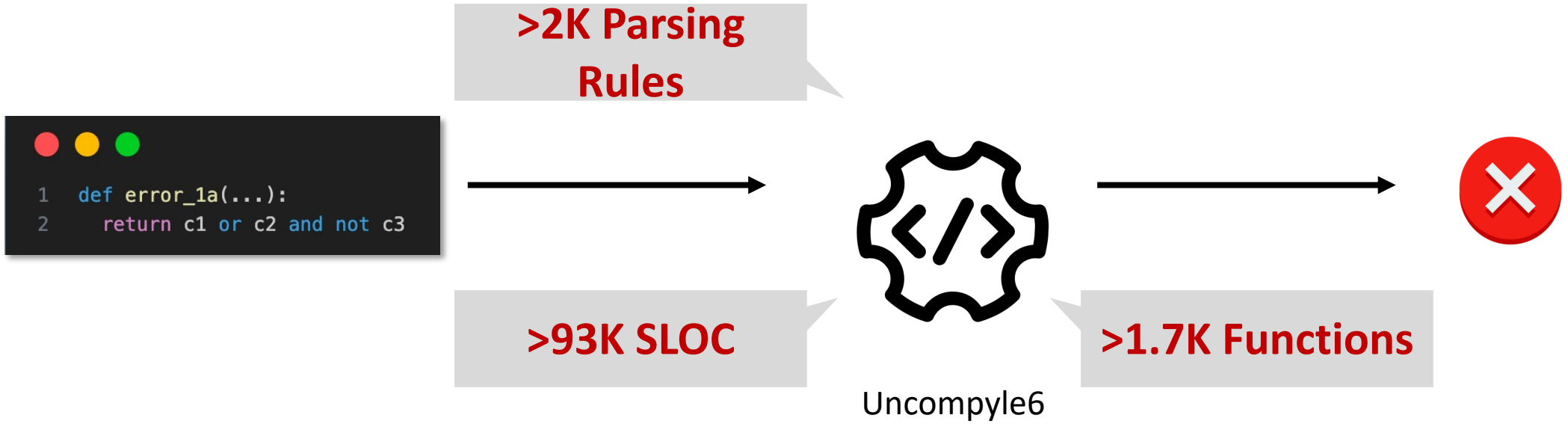
- Python **malware** is compiled to **Bytecode**: **Challenging** to **analyze**.
- **Decompilers** come to the rescue!
- But they **fail** on many binaries.



decompilation (fail)

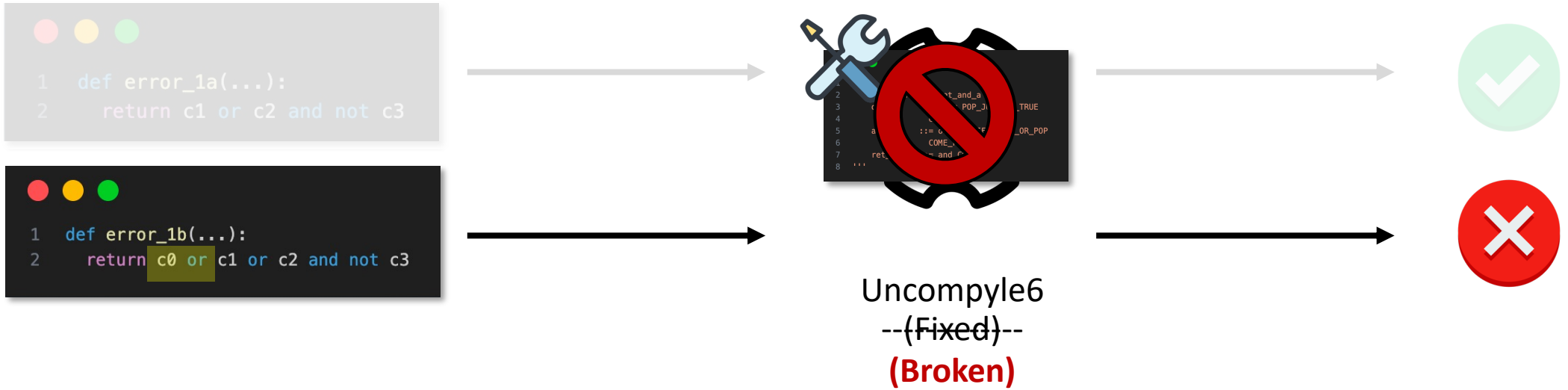
Challenges in Handling Decompilation Errors

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



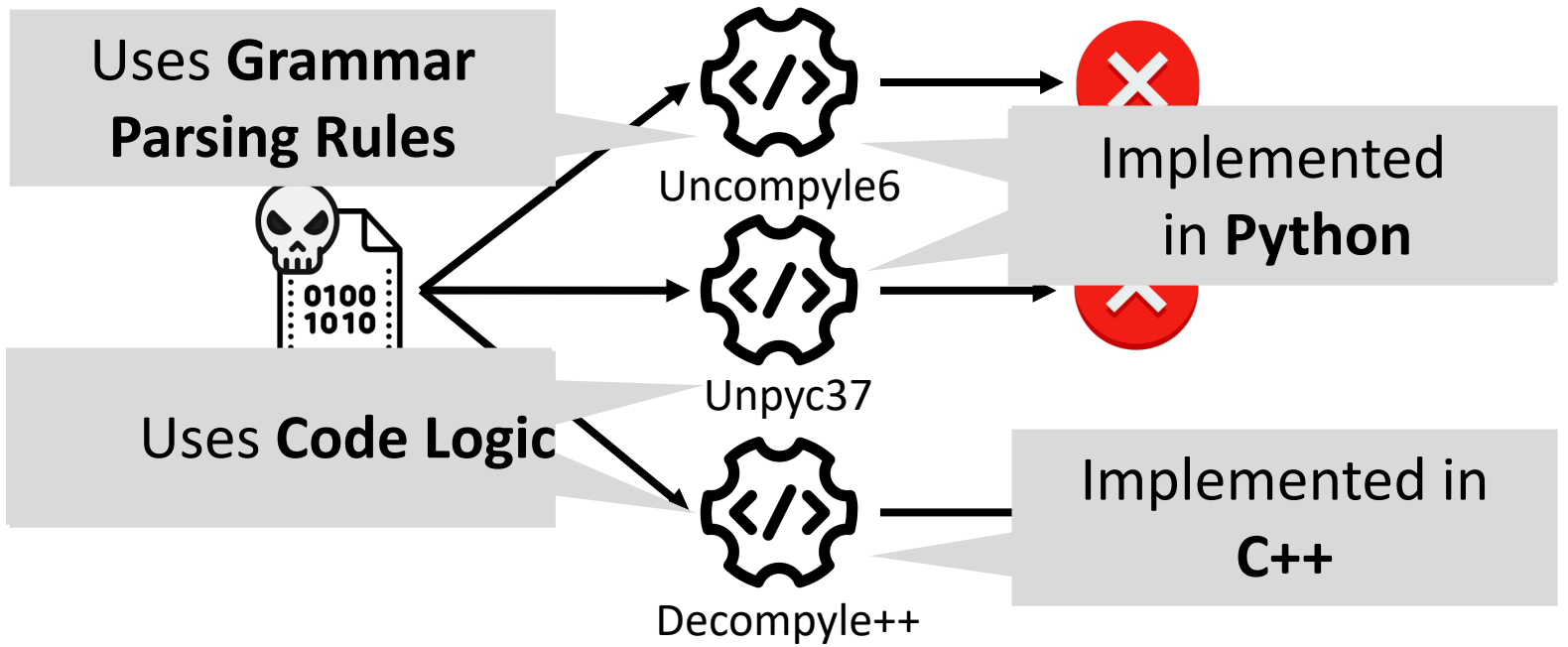
Challenges in Handling Decompilation Errors

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



Challenges in Handling Decompilation Errors

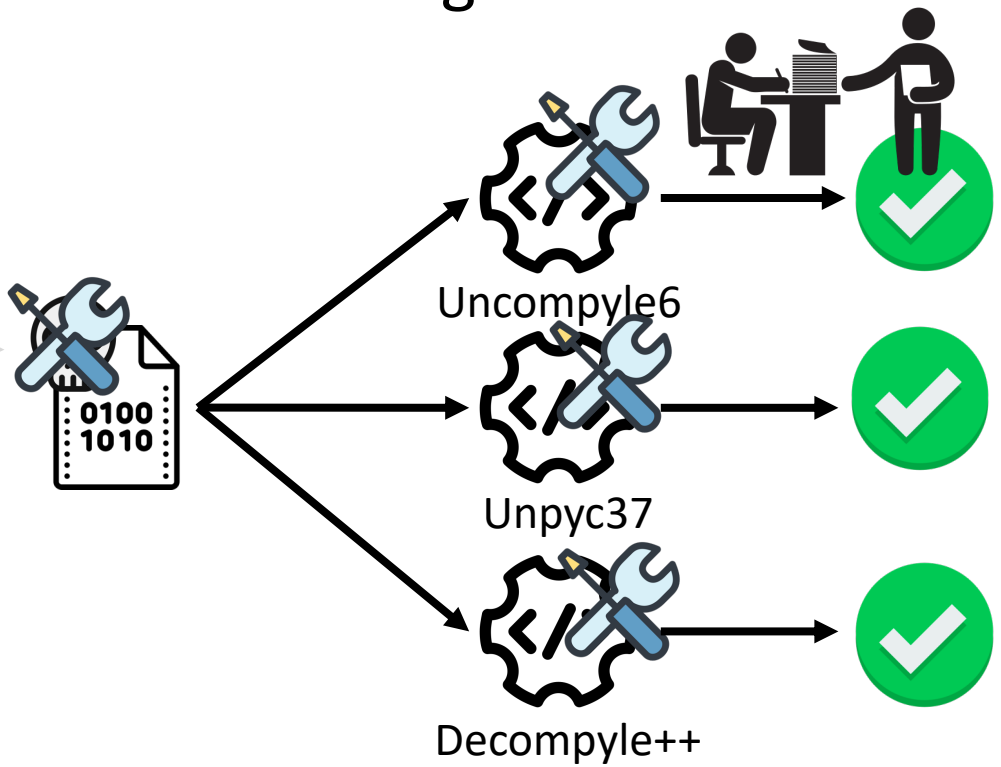
1. **Asymmetric Warfare:** Decompilers are easy to break but hard to fix.
   - **Debugging** decompilers **requires substantial expertise** and **effort.**
2. **Multiple decompilers:** Not scalable to debug all.



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**.**

```
203:        636 POP_EXCEPT
            638 JUMP_FORWARD         (to 642)
            640 END_FINALLY

205:    >>  642 LOAD_FAST            (self_spread)
            644 EXTENDED_ARG         (512)
            646 POP_JUMP_IF_TRUE     (to 672)
            648 LOAD_FAST            (self_spread)
            650 EXTENDED_ARG         (512)
            652 POP_JUMP_IF_TRUE     (to 672)
            654 LOAD_FAST            (self_spread)
            656 EXTENDED_ARG         (512)
            658 POP_JUMP_IF_TRUE     (to 672)
            660 LOAD_FAST            (self_spread)
            662 EXTENDED_ARG         (512)
            664 POP_JUMP_IF_TRUE     (to 672)
            666 LOAD_FAST            (self_spread)
            668 EXTENDED_ARG         (512)
            670 POP_JUMP_IF_FALSE    (to 750)

206:    >>  672 SETUP_LO
            674 LOAD_FAS
            676 GET_ITER
        >>  678 FOR_ITER
            680 STORE_FA

207:        682 LOAD_GL
            684 LOAD_GL
            686 LOAD_CON
            688 BINARY_S
            690 LOAD_CON
            692 LOAD_CON
            694 CALL_FUNCTION_KW     (2 total positional and keyword args)
            696 SETUP_WITH           (to 712)
            698 STORE_FAST           (file)

208:        700 LOAD_FAST            (file)
            702 LOAD_METHOD          (read)
            704 CALL_METHOD          (0 positional arguments)
            706 STORE_FAST           (content)
```

decompilation (fail)

```
def main--- This code section failed: ---

  L. 116         0  LOAD_GLOBAL              ROAMING
                 2  LOAD_STR                 '\\.cache~$'
                 4  BINARY_ADD
                 6  STORE_FAST               'cache_path'

...

  L. 203       636  POP_EXCEPT
               638  JUMP_FORWARD        642  'to 642'
               640  END_FINALLY
             642_0  COME_FROM           638  '638'
             642_1  COME_FROM           628  '628'

  L. 205       642  LOAD_FAST                'self_spread'
           644_646  POP_JUMP_IF_TRUE    672  'to 672'
               648  LOAD_FAST                'self_spread'
           650_652  POP_JUMP_IF_TRUE    672  'to 672'
               654  LOAD_FAST                'self_spread'
           656_658  POP_JUMP_IF_TRUE    672  'to 672'
               660  LOAD_FAST                'self_spread'
           662_664  POP_JUMP_IF_TRUE    672  'to 672'
               666  LOAD_FAST                'self_spread'
           668_670  POP_JUMP_IF_FALSE   750  'to 750'
             672_0  COME_FROM           662  '662'
             672_1  COME_FROM           656  '656'
             672_2  COME_FROM           650  '650'
             672_3  COME_FROM           644  '644'

           744_746  JUMP_BACK           678  'to 678'
               748  POP_BLOCK
             750_0  COME_FROM_LOOP      672  '672'
             750_1  COME_FROM           668  '668'

Parse error at or near `COME_FROM_LOOP' instruction at offset 750_0
```

Explicit Error: Decompilation failure with an explicit error message

Decompilers also **fail** **silently (Implicit Error)!**

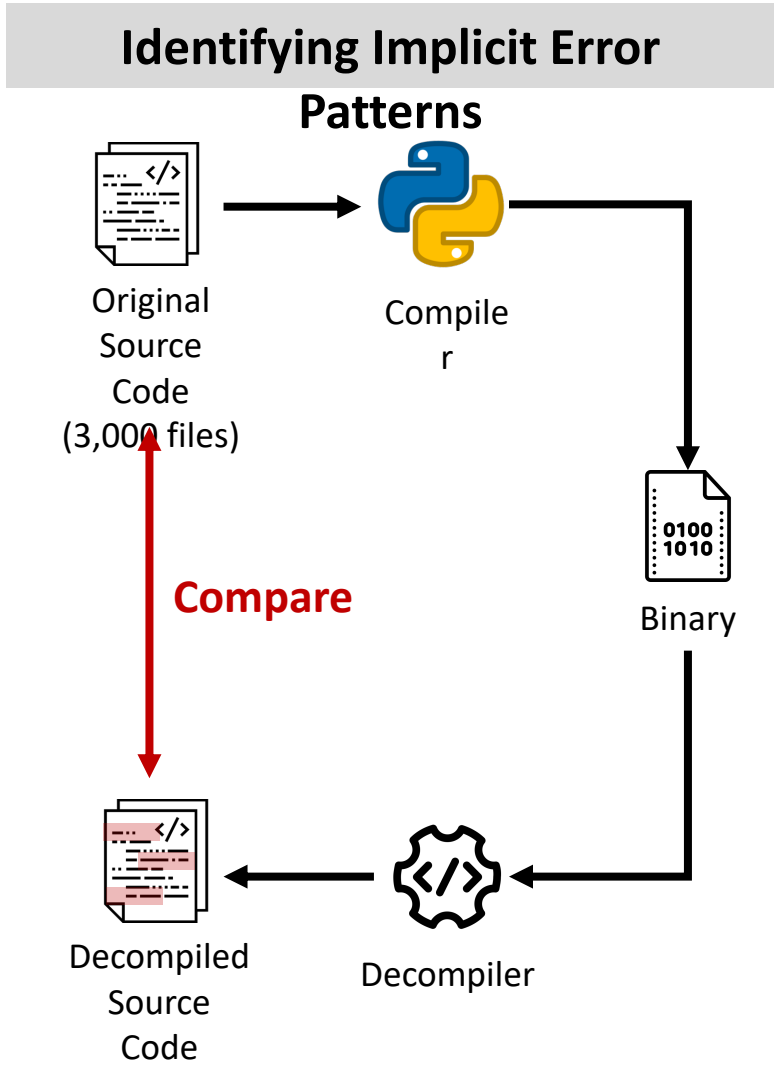
- Decompiled successfully but generating a **wrong** code.

Implicit Error: Decompilation failure without any error message

Original Program (Ground truth)

Decompiled Program with **Implicit Error** (the '**else:**' blocks)

Identifying (a few) Implicit Error Patterns



Correct Code Pattern

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

return **under** the wrong **if**.

Implicit Error Pattern

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

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

else block **wrongly coupled**

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

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

new '**else:**' introduced

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

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

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

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

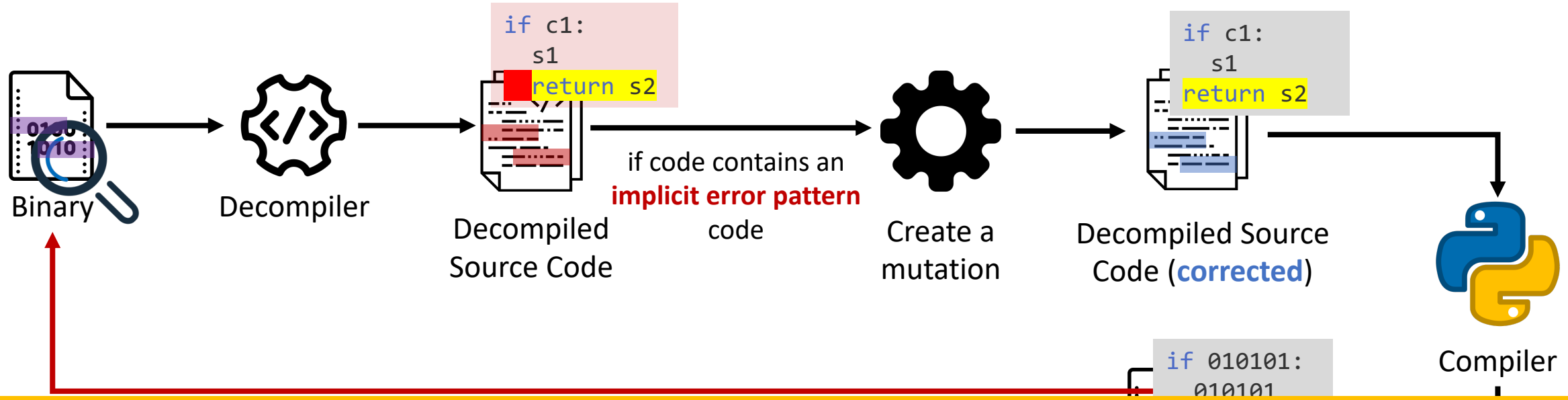
Detecting: Implicit Errors with the Patterns

Correct Code

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

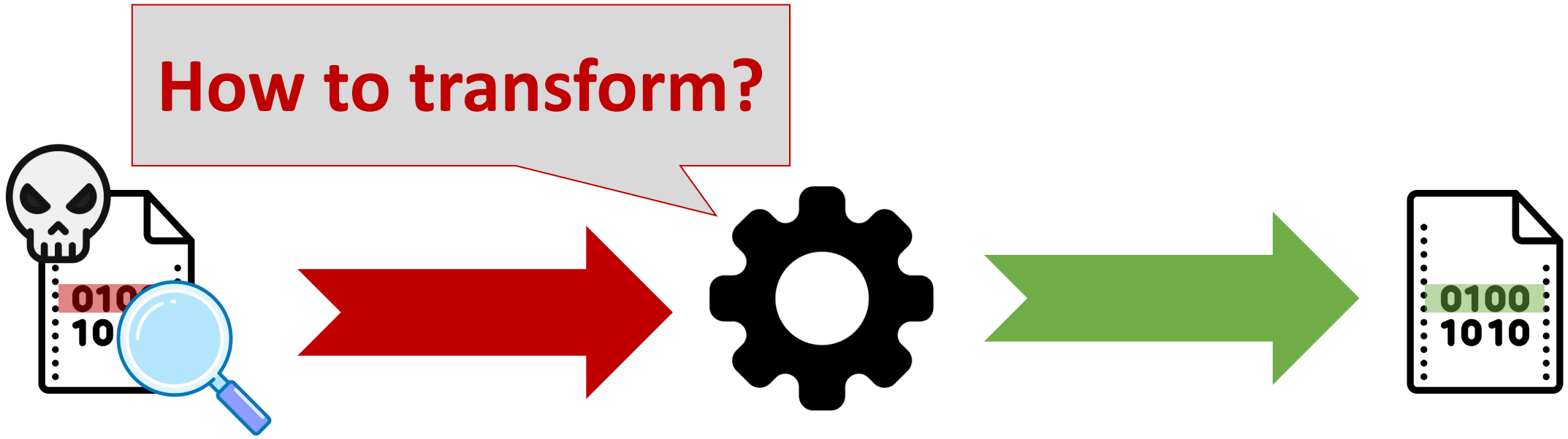
Implicit Error

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



Detected **22,359 Implicit Errors** from **5 Decompilers**

Identifying Decompilation Failure



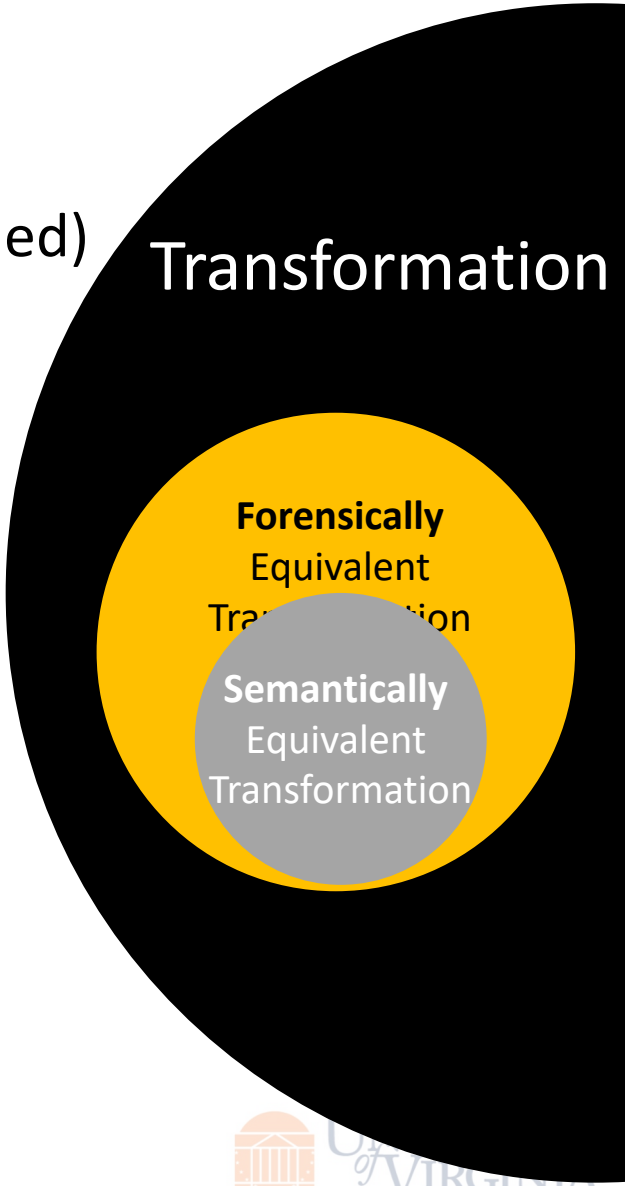
Identifying Decompilation Failure

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

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

Forensically Equivalent Transformation

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

Semantically Equivalent Transformation

```
try:
  file = open("file", "r")
return file.read()
except:
```

Failure Condition:
'return' in **'try-except'**

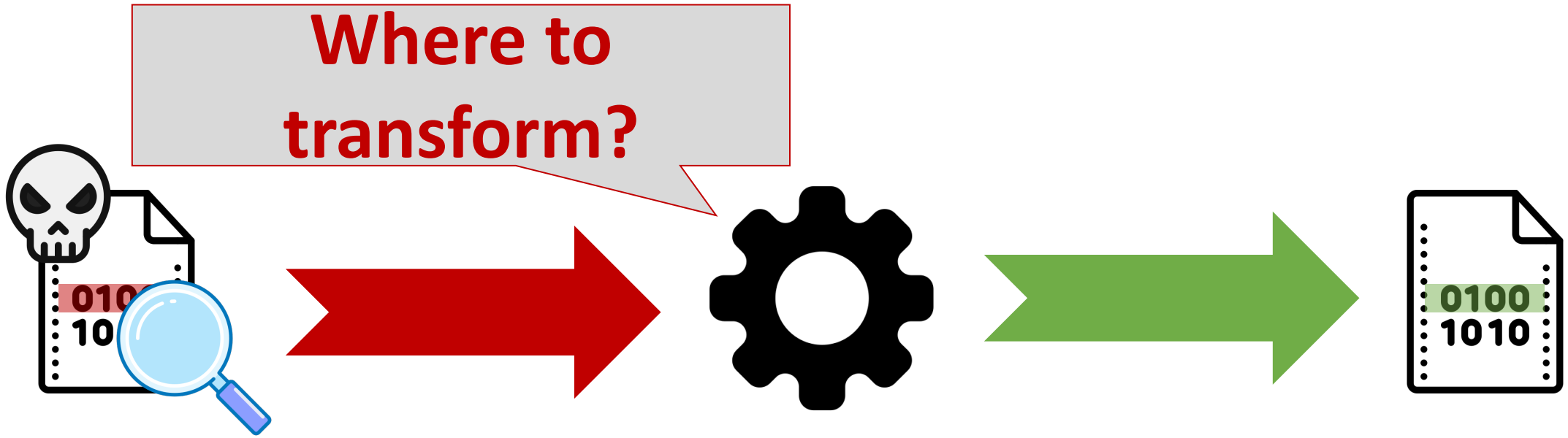
Fixing: (a few) **Transformation Rules**

Original Code		Transformed Code	Reason
`(x or y) and z`	**Dividing** logical expressions →	`t = x or y` `t and z`	Parsers fails to understand **chains of logical expressions**.

`if c1:`
`  s1`
`elif c2:`
`  s2`
	Simplifying/flattening conditionals →	`if c1:`
		`  s1`
		`if c2 and not c1:`
		`  s2`
			Parser fails to parse **chains of conditionals**.

`f = lambda x: x*2`	Converting **lambda** function →	`def f(x):` `  return x*2`	Language features like **'lambda'** are not supported.

`while c1:`
`  s1`
`  break`
`s2`
	Eliminating **consecutive control flow changes** →	`while c1:`
		`  s1`
		`  break`
		`  FET_null()`
		`s2`
			Parser fails to handle multiple **consecutive control flow changes** (e.g., loop back + break).



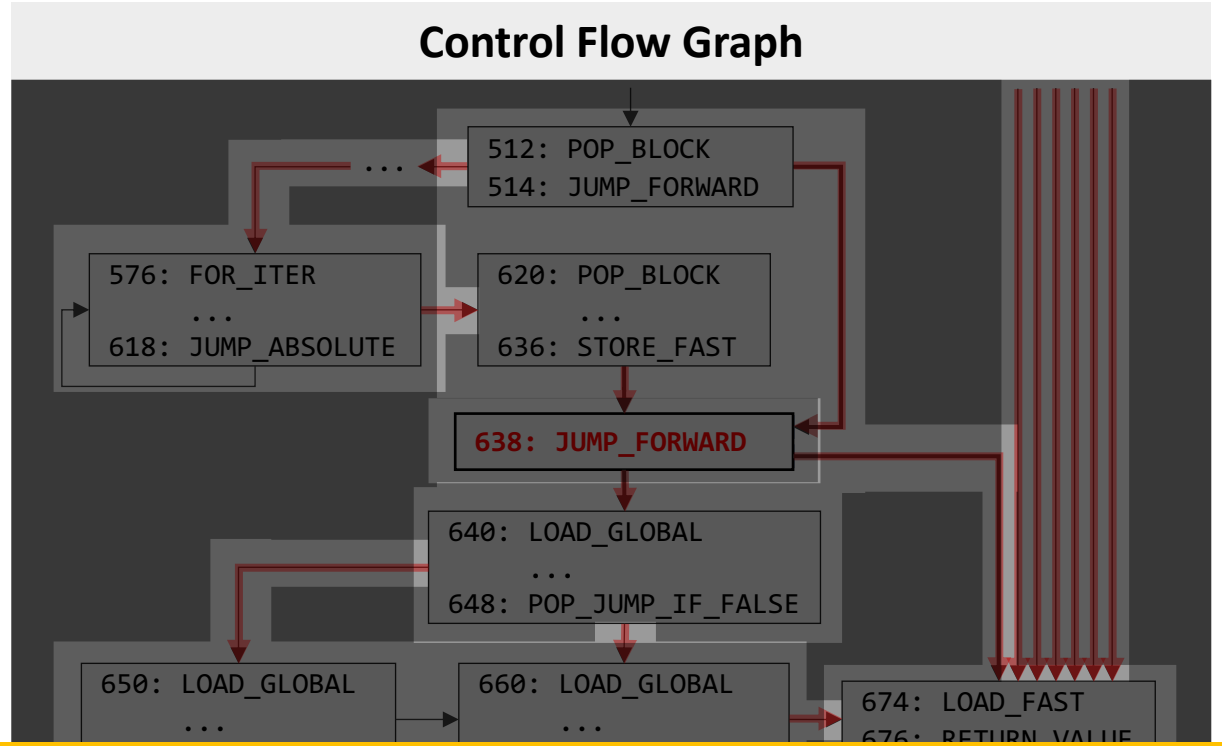
Identifying Decompilation Failure

Applying Transformation

Successfully Decompilable Binary

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

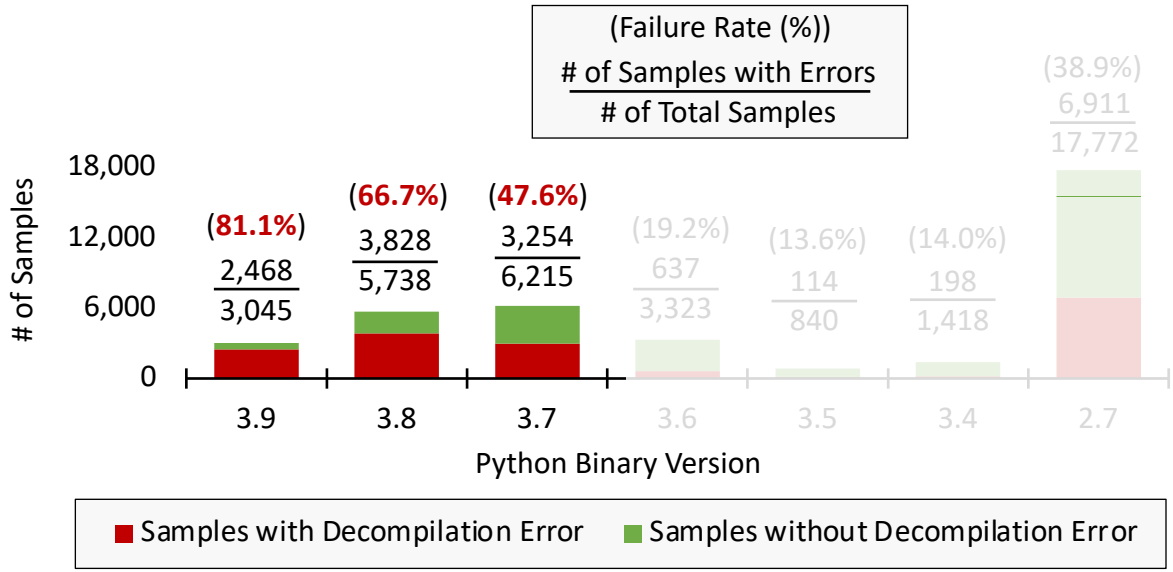
2. Extend targets:
   All directly reachable nodes from 638.

3. Extend targets:
   All directly reachable nodes from already covered nodes.

Explores only **33%** of the nodes on average

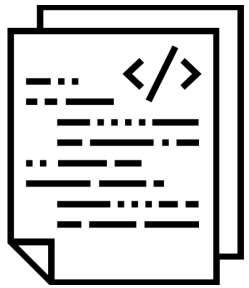
Resolving **Real-world Decompilation Failures**

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

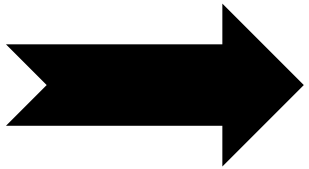


We **resolved** decompilation failures **of all the malware samples** (17,117)!

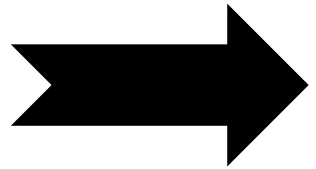
Case Study: Opcode Remapped & Obfuscated Binary

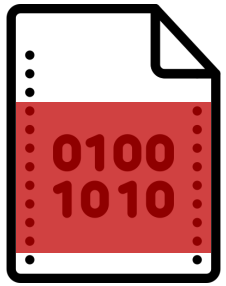


Python Source Code



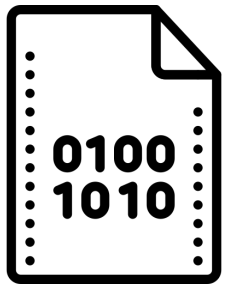
Customized Python Compiler



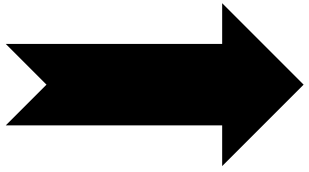


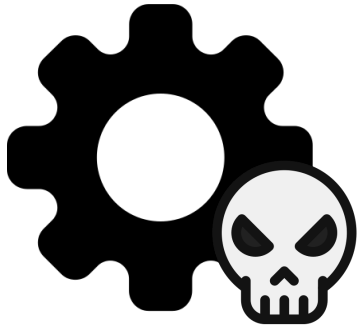
Opcode Remapped Binary

(not decompilable)

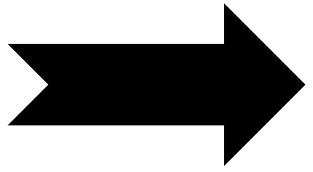


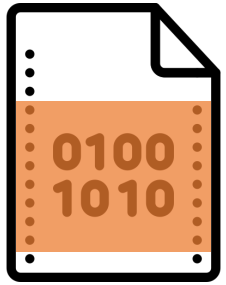
Python Binary





Obfuscator





Obfuscated Binary

We resolve all! More detail in paper.

Summary

17K **Malware binaries' decompilation errors resolv**

5 Different **Python Decompilers** handled.
(Uncompyle6, Decompyle3, Uncompyle2, Uncompile3, Decompyle++)

3 **Opcode remapping** and **obfuscated** binaries handl
(DropBox and druva)

30 **Transformation rules** developed.