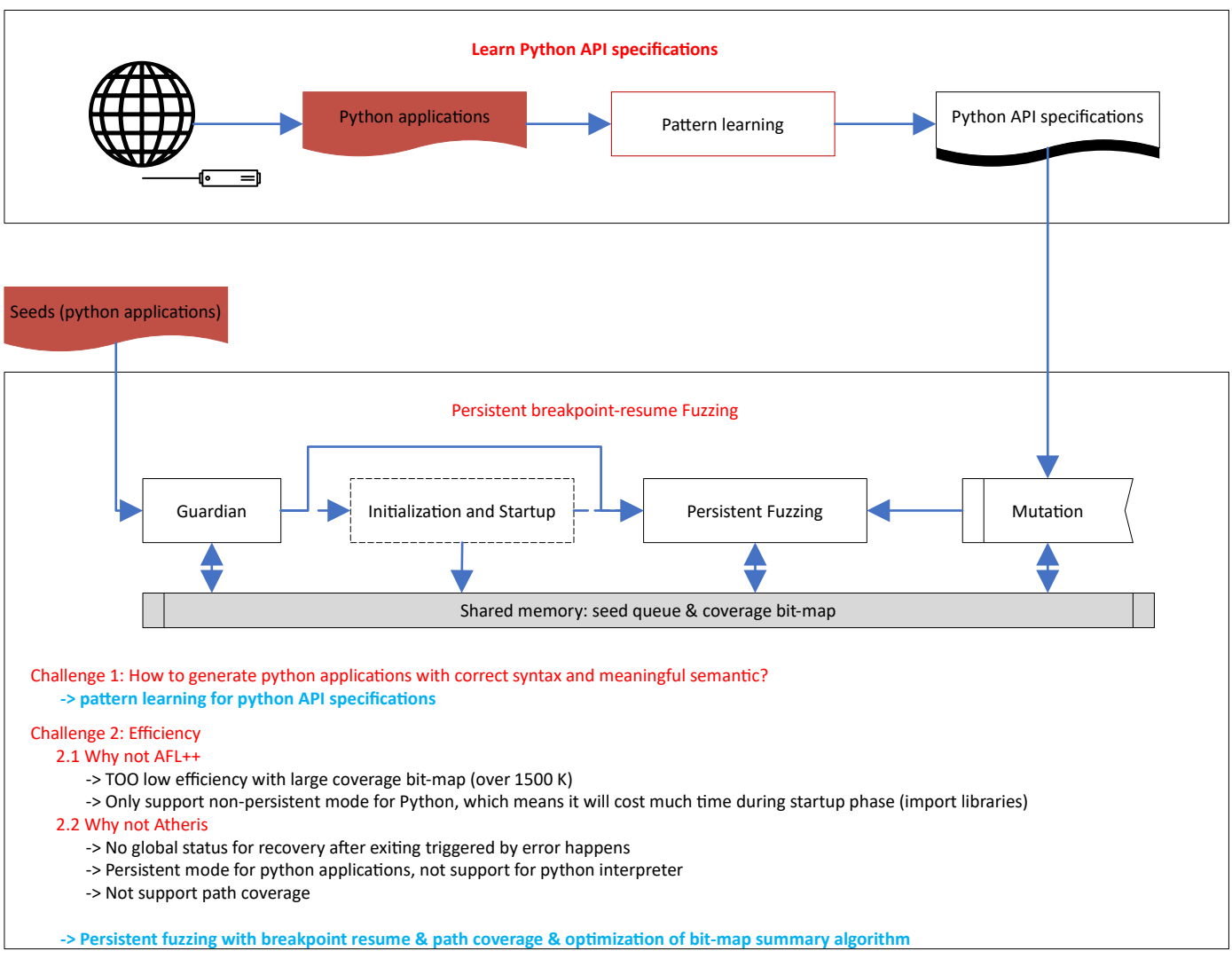
Overview



Challenge 1: How to generate python applications with correct syntax and meaningful semantic?

-> pattern learning for python API specifications

Challenge 2: Efficiency

2.1 Why not AFL++

-> TOO low efficiency with large coverage bit-map (over 1500 K)

-> Only support non-persistent mode for Python, which means it will cost much time during startup phase (import libraries)

2.2 Why not Atheris

-> No global status for recovery after exiting triggered by error happens

-> Persistent mode for python applications, not support for python interpreter

-> Not support path coverage

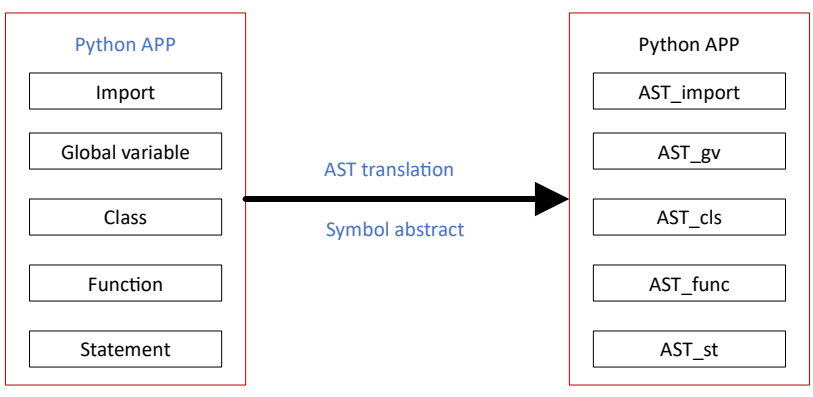
-> Persistent fuzzing with breakpoint resume & path coverage & optimization of bit-map summary algorithm

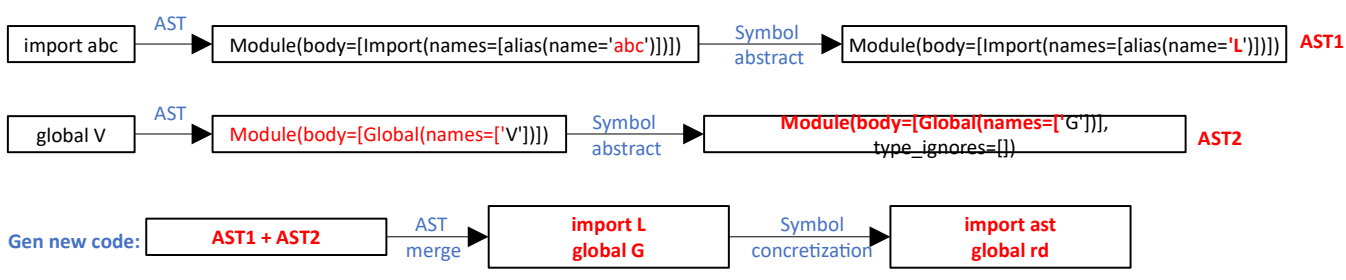
Python APP generation

1 **General python APP generation**

1. Learn the grammar from Python applications
   - translate APP into ASTs
   - for each type of element, **learn** the patterns and normalize the format through abstract symbol

2. Gen new APP through operation (ADD/DEL/MERGE/RECURSIVE...) of ASTs





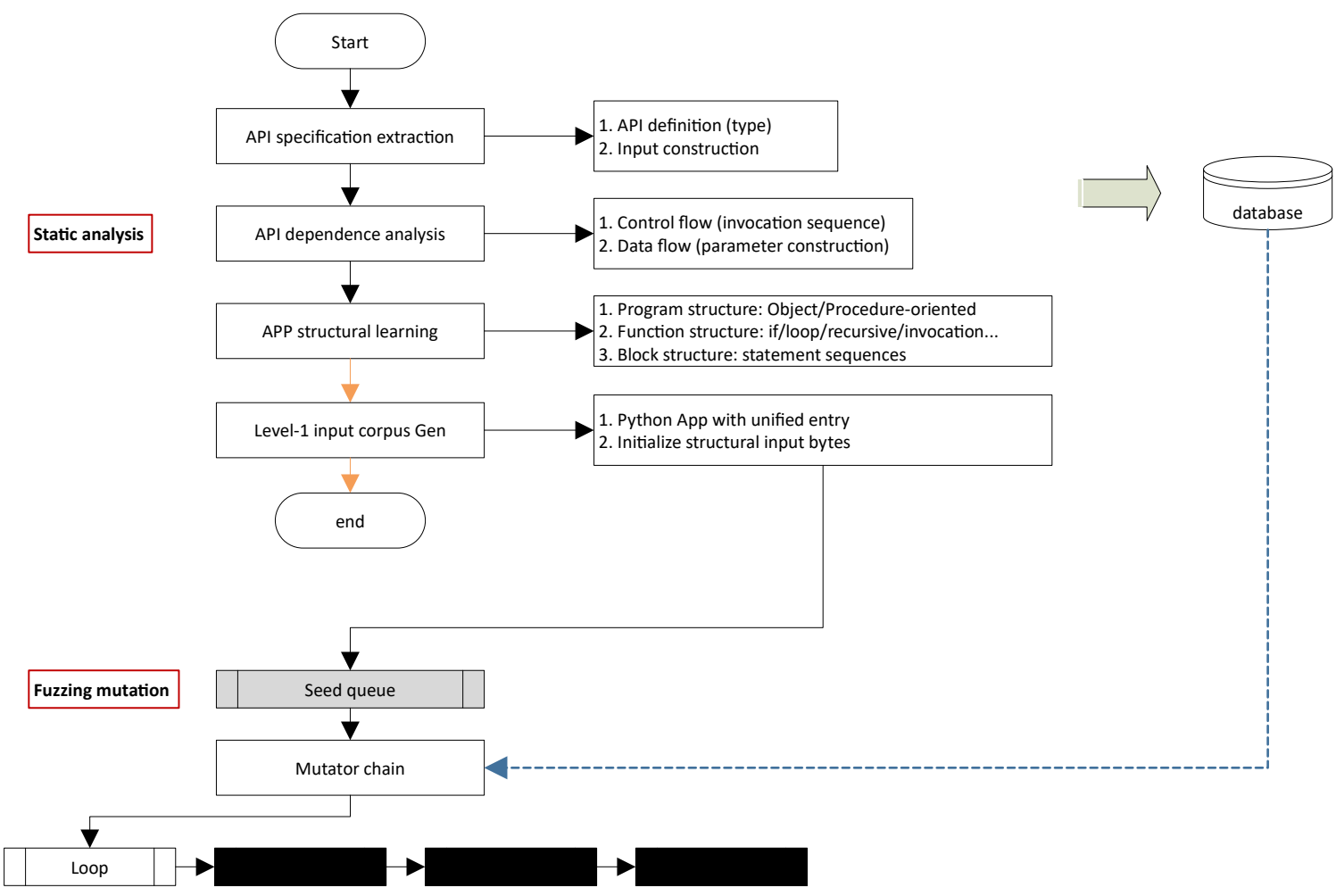
2 **Python APP generation focusing on usage of runtime libraries**

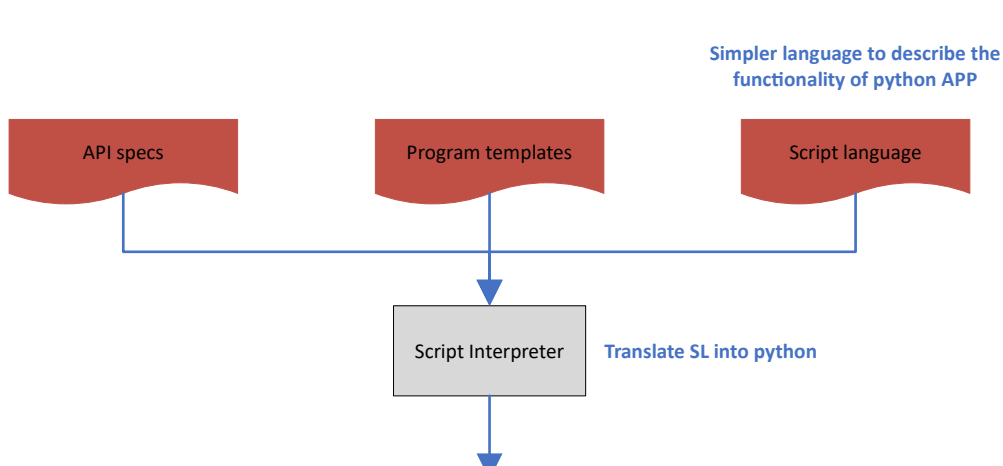
The usage of API is in form of call statement, hence we only need to learn all the possible call statements of these APIs and store them as ASTs

3 **Possible dependencies of runtime library APIs**

For possible dependencies among APIs, we can learn from the real world programs through static slicing

Software design





Level-1 Fuzzing type: Generation-based / Mutation-based → **Generation-based fuzzing**

Example

Templates

```
class demoCls:
  def __init__(self):
    pass
  def demoFunc0(self):
    pass
  def demoFunc1(self, PYF_ARG1):
    pass
def RunFuzzer (x):
  pass
```

```
def demoFunc0 ():
  pass

def demoFunc1 (PYF_ARG1):
  pass

def RunFuzzer (x):
  pass
```

```
<apisepc>
 <api>
  <module>cmath</module>
  <name>exp</name>
  <expr>ret = cmath.exp(x)</expr>
  <parameters>{'x':int}</parameters>
  <return>{'ret':int}</return>
 </api>
</apisepc>
```

CALL cmath.exp

```
class demoCls():
  def __init__(self):
    pass
  def demoFunc0(self):
    pass
  def demoFunc1(self, PYF_ARG1):
    ret = cmath.exp(PYF_ARG1)
def RunFuzzer(x):
  demoCls().demoFunc1(x)
```

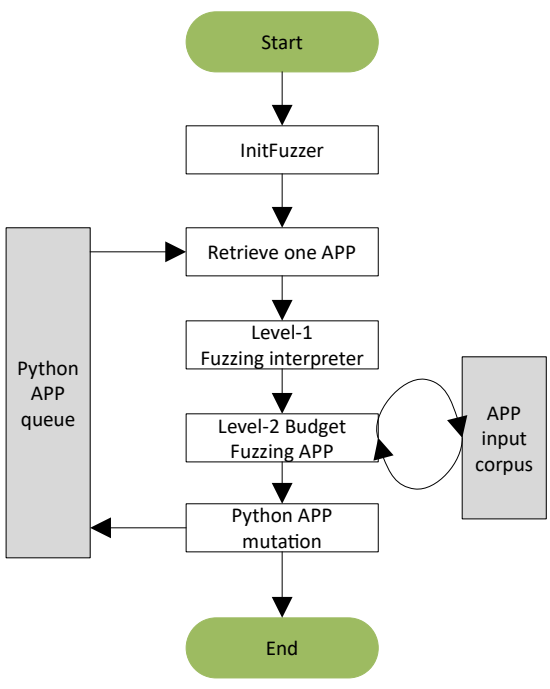
FOR_LOOP cmath.exp

```
class demoCls():
  def __init__(self):
    pass
  def demoFunc0(self):
    pass
  def demoFunc1(self, PYF_ARG1):
    for PYF_I in range(0, PYF_ARG1):
      ret = cmath.exp(PYF_I)
def RunFuzzer(x):
  demoCls().demoFunc1(x)
```

Recurve cmath.exp

```
class demoCls():
  def __init__(self):
    pass
  def demoFunc0(self):
    pass
  def demoFunc1(self, PYF_ARG1):
    for PYF_I in range(0, PYF_ARG1):
      ret = cmath.exp(PYF_I)
      self.demoFunc1(ret)
def RunFuzzer(x):
  demoCls().demoFunc1(x)
```

Two-level Fuzzing



Level-1 Fuzzing: Targeting interpreter Core. We use infinite budget at level-1.

Level-2 Fuzzing: Targeting runtime libraries. We use finite budget at level-2 until no favored path/block/feature found.

```
<apisepc>
  <library name="Email">
    <module name="charset">
      <class name="Charset">
        <api>
            <name>get_body_encoding</name>
            <expr>ret = get_body_encoding()</expr>
            <parameters>{}</parameters>
            <return>{'ret':base64|7bit}</return>
            <dependences>  </dependences>
        </api>

        <api>
            <name>header_encode</name>
            <expr>header_encode(str)</expr>
            <parameters>{'str':string}</parameters>
            <return>{}</return>
            <dependences>  </dependences>
        </api>
      </class>
    </module>

    <module name="contentmanager">
      <api>
        <name>header_encode</name>
        <expr>header_encode(str)</expr>
        <parameters>{'str':string}</parameters>
        <return>{}</return>
        <dependences>  </dependences>
      </api>
    </module >

    <errors>
      <exception>MessageError</exception>
      <exception>MessageParseError</exception>
      <exception>BoundaryError</exception>
      <exception>MultipartConversionError</exception>
    </errors>
  </library>
</apisepc>
```

Program structure

Object-oriented

```
class demoCls:
  def __init__(self):
     pass
  def demoFunc0(self):
     pass
  def demoFunc1(self, arg1):
     pass

def RunFuzzer (x):
  pass
```

Procedure-oriented

```
class demoCls:
  def __init__(self):
     pass
  def demoFunc0(self):
     pass
  def demoFunc1(self, arg1):
     pass

def RunFuzzer (x):
  pass
```

```
class demoCls (PY_LIB.CLS):
  def __init__(self):
     pass
  def CLS_func0(self):
     pass
  def CLS_func1(self, arg1):
     pass

def RunFuzzer (x):
  pass
```

Function structure

```
For intV in range (x, y):
     pass
```

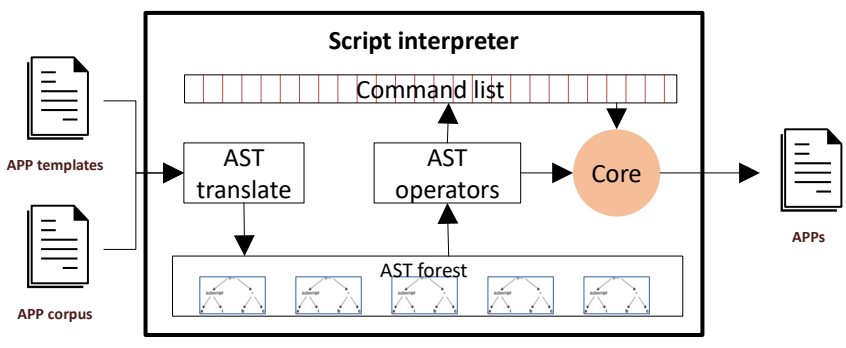
```
For obj in iterObj:
     pass
```

```
if x:
  pass
```

```
if x:
  pass
Else:
  pass
```

ScriptL: a simple script language for describing the semantic of python APP generation.

ScriptL Interpreter: interpreter the scripts and generate new python applications for various run-time APIs and python built-in functions.



Structural Command

[Command]: OO
[Description]: generate a new object-oriented program

[Command]: PO
[Description]: generate a new procedure-oriented program

[Command]: Inherit
[Description]: Inherit the class

Semantic abstract command

[Command]: FOR
[Description]: Wrap the code into a for loop

[Command]: While
[Description]: Wrap the code into a while loop

[Command]: With
[Description]: Wrap the code into a with block

[Command]: If
[Description]: Wrap the code into a if-else block

[Command]: ExcepNest
[Description]: Raise except in except block

[Command]: Callback
[Description]: Invoke a function through callback

[Command]: Recursive
[Description]: Recursive Invoke a function

[Command]: Pickle
[Description]: Store and load the python object

[Command]: ObjAdd (Magic Methods)
[Description]: Add two python object if support '__add__'

[Command]: ObjSub (Magic Methods)
[Description]: Add two python object if support '__sub__'

Custom command

Summarize the pattern from bugs
Target: **5**