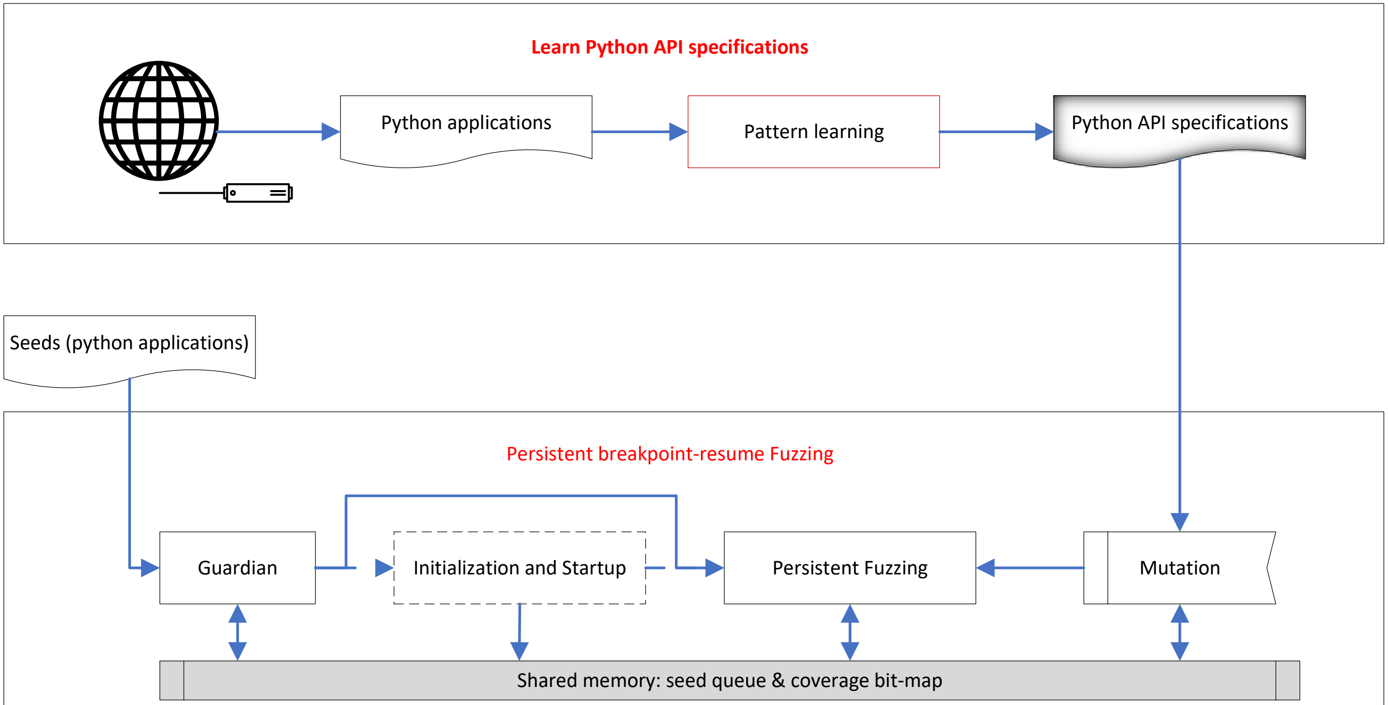
Overview



Learn Python API specifications
Python applications
Pattern learning
Python API specifications
Seeds (python applications)
Persistent breakpoint-resume Fuzzing
Guardian
Initialization and Startup
Persistent Fuzzing
Mutation
Shared memory: seed queue & coverage bit-map

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

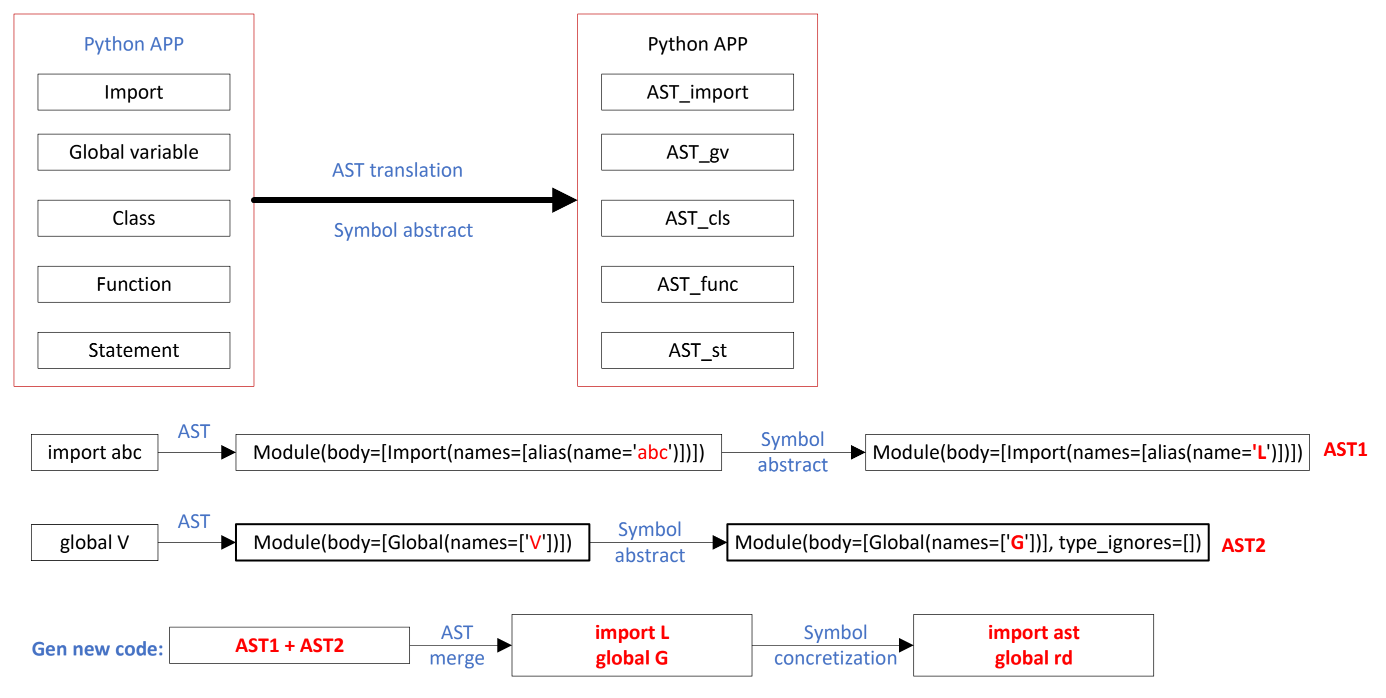
-> Persistent fuzzing with breakpoint resume & path coverage & optimization of bit-map summary algorithm

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



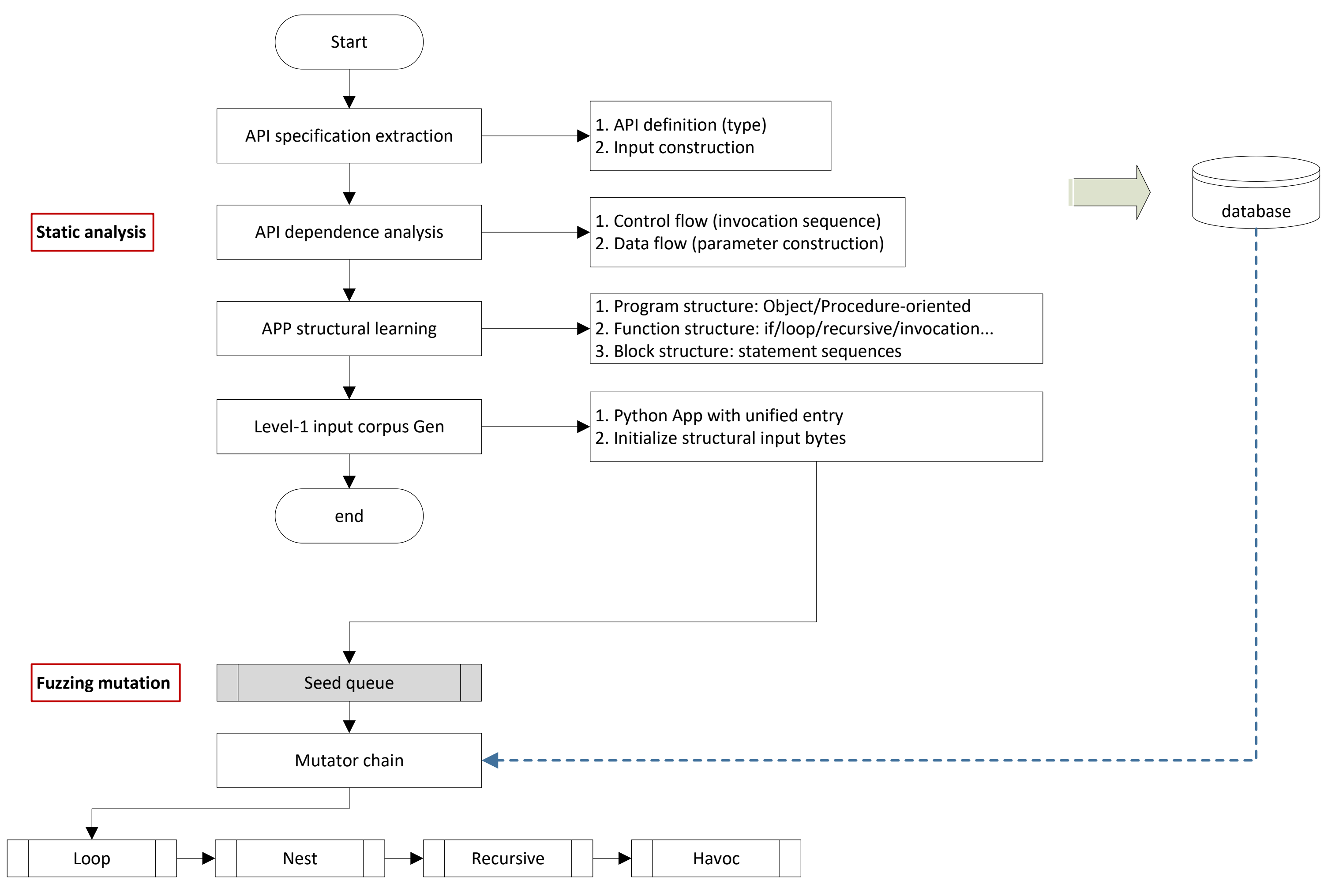
2 **Python APP generation focusing on usage of runtime libraries**

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

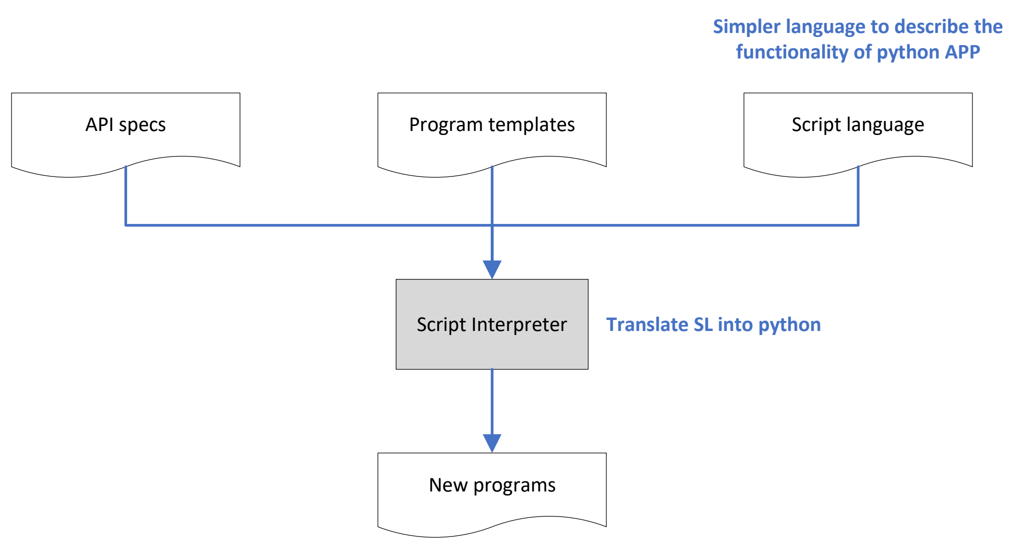
3 **Possible dependencies of runtime library APIs**

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

Software design

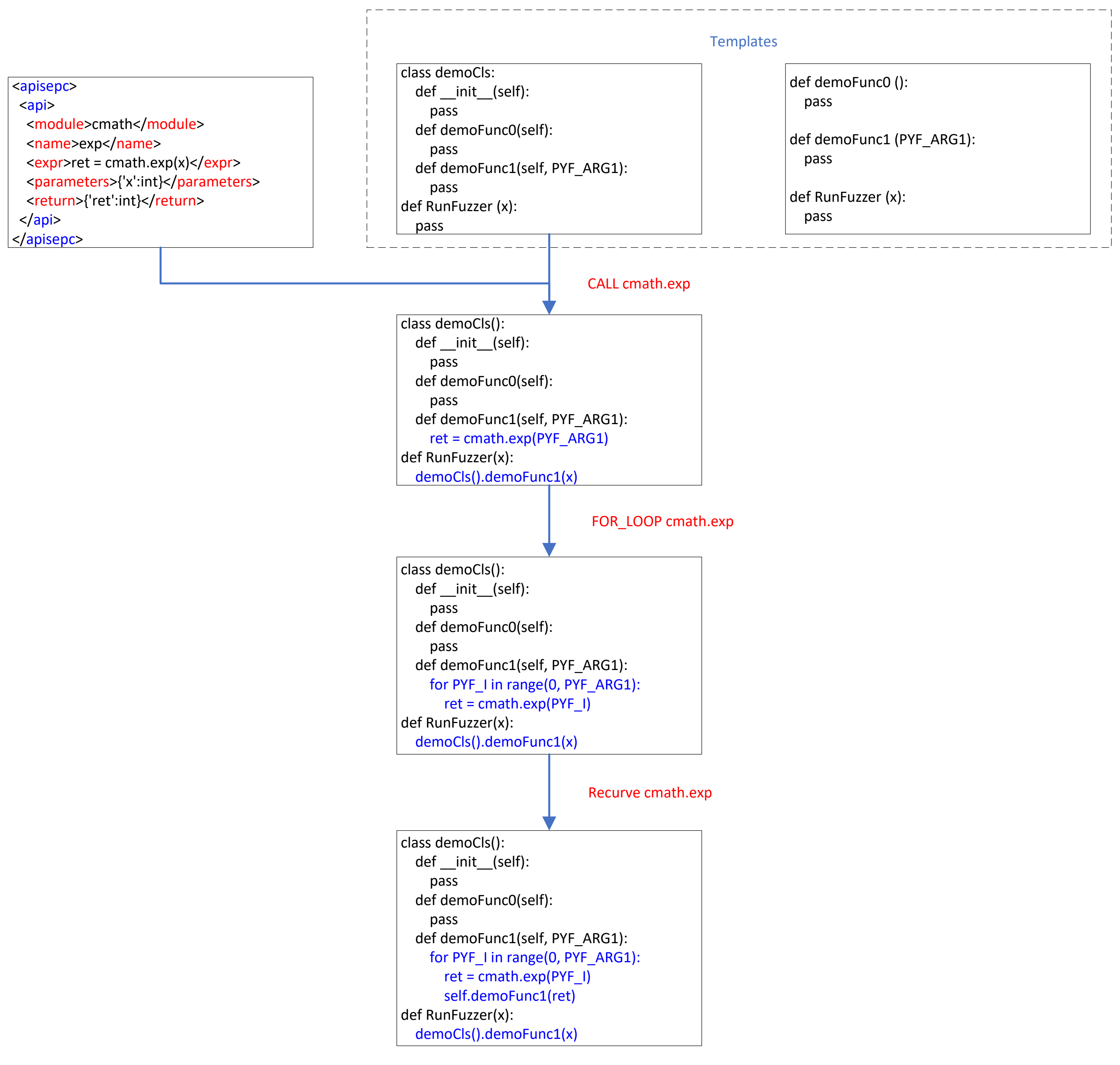


Start
API specification extraction
1. API definition (type)
2. Input construction
Static analysis
API dependence analysis
1. Control flow (invocation sequence)
2. Data flow (parameter construction)
database
APP structural learning
1. Program structure: Object/Procedure-oriented
2. Function structure: if/loop/recursive/invocation...
3. Block structure: statement sequences
Level-1 input corpus Gen
1. Python App with unified entry
2. Initialize structural input bytes
end
Fuzzing mutation
Seed queue
Mutator chain
Loop
Nest
Recursive
Havoc



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

Example



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

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

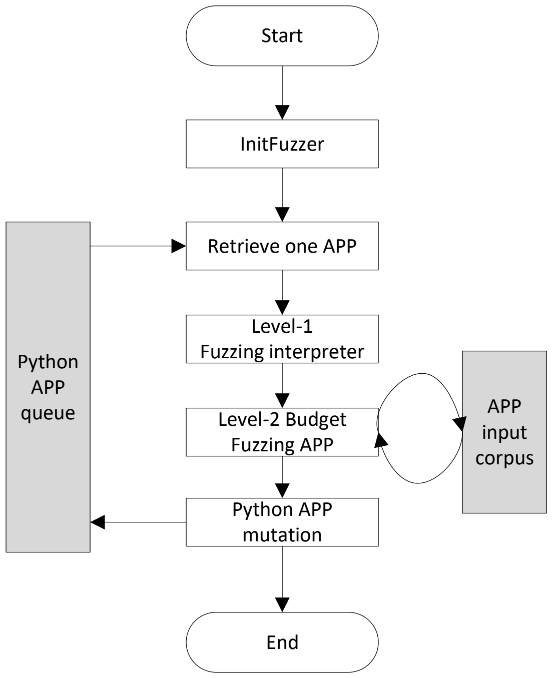
```
def demoFunc0 ():
   pass

def demoFunc1 (PYF_ARG1):
   pass

def RunFuzzer (x):
   pass
```

```
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)
```

```
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)
```

```
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.