Formal Verification of Python-based AI libraries

Bruno Farias, Dr. Youcheng Sun and Prof. Lucas Cordeiro

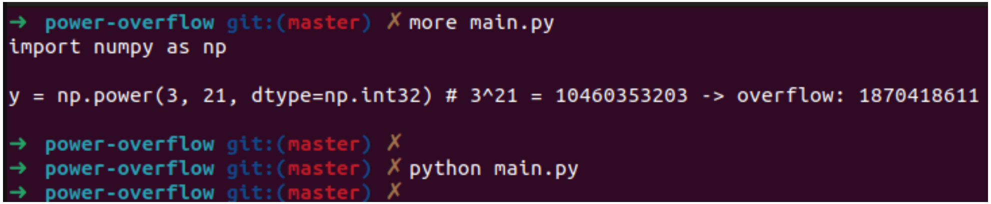
Department of Computer Science
University of Manchester

June 9, 2025

Motivating Example

main.py

```python
import numpy as np
y = np.power(3, 21, dtype=np.int32) # overflow: 1870418611
```



```
→  power-overflow git:(master) ✗ more main.py
import numpy as np

y = np.power(3, 21, dtype=np.int32) # 3^21 = 10460353203 -> overflow: 1870418611

→  power-overflow git:(master) ✗
→  power-overflow git:(master) ✗ python main.py
→  power-overflow git:(master) ✗
```

Challenges in Python Verification

- **Dynamic Features**
  - Dynamic typing, introspection, and runtime checks.
- **Optional Type System**
- **Limitations of Existing Approaches**
  - Requires user expertise or code annotations.
  - Cannot handle Python's dynamic aspects.
  - Produces false positives and fails to handle external libraries.
  - Often fails to guarantee full program coverage.

```python
a = "1"
a = 1
b = MyClass()
if isinstance(a, int):
  b.x = "str"
else:
  b.x = 2
c = b.x
d = c.upper()
```

Research Questions

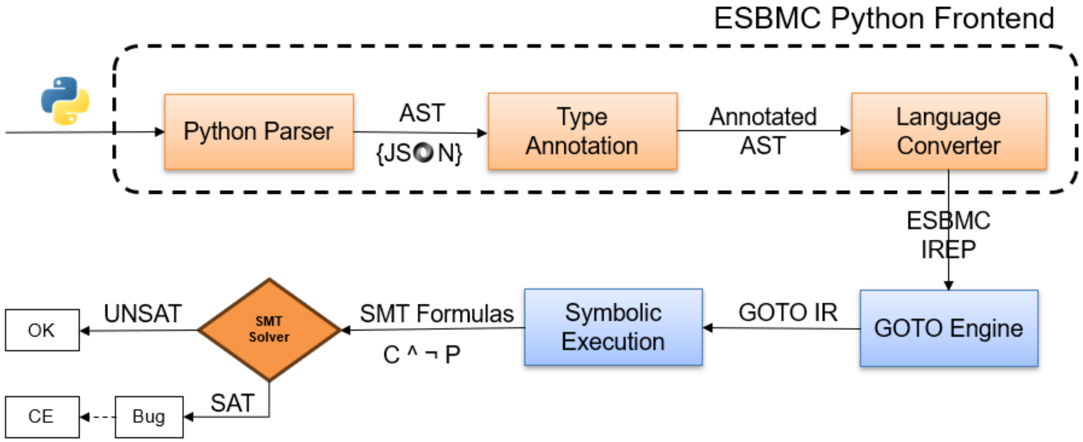
RQ1

Given the challenges of verifying Python programs and the limitations of current tools, is it possible to reuse a model checking framework to reduce time and effort while advancing the state of the art?

RQ2

Which verification properties are relevant for Python-based AI libraries, and how can a model checking framework be applied to verify them?

Our approach to verify Python Programs



JSON-Based Type Inference

- Constant Values

  `x = 10 ⇒ x:`**`int`**` = 10`

- Referred Variables

  `y = x ⇒ y:`**`int`**` = x`

- Class Instances

  `z:`**`MyClass`**` = MyClass()`

- Function Calls

  `x = foo() ⇒ x:`**`int`**` = foo`

```
{
 "_type": "Assign",
 "target": {
   "_type": "Name",
   "id": "x"
 },
 "value": {
   "_type": "Constant",
   "value": 10
 }
}
```

x = 10

```
{
  "_type": "AnnAssign",
  "target": {
    "_type": "Name",
    "id": "x"
  },
  "annotation": {
    "_type": "Name",
    "id": "int"
  },
  "value": {
    "_type":"Constant",
    "value": 10
  }
}
```

x:int = 10

Research Progress

- Python features
  - Language basics: Data types, conditionals, loops, functions.
  - OOP: classes, inheritance, polymorphism
  - Containers: Strings, Lists
- Non-determinism: Used to model unspecified inputs.
- Operational models: Simplified stubs to reduce external libraries complexity.
- Verification Properties
  - Arithmetic overflow, out-of-bounds access, division-by-zero, user assertions.
- Case Studies
  - Ethereum Consensus Verification: 1 bug confirmed; 7 bugs in analysis.
  - Numpy Verification: Arrays construction, math correctness.

Verification of Ethereum Specifications

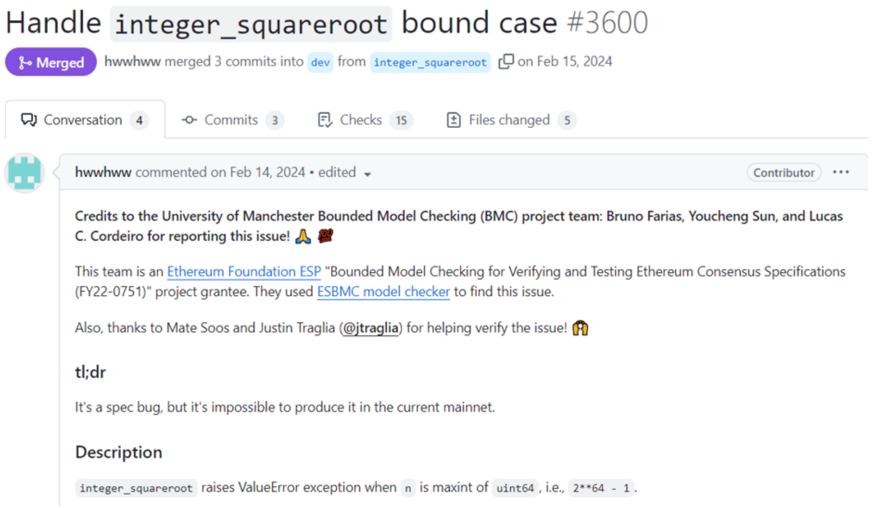
- **Consensus Specification**: A set of runnable specifications in Python.
- Detected overflow and div-by-zero in a function call.

main.py

```
1  def integer_squareroot(n: uint64)
       -> uint64:
2      x = n
3      y = (x + 1) // 2
4      while y < x:
5          x = y
6          y = (x + n // x) // 2
7      return x
```

```
[Counterexample]
State 1 main.py line 2
x = 0xFFFFFFFFFFFFFFFF
--------------------------------
State 2 main.py line 3
y = 0
--------------------------------
State 3 main.py line 5
x = 0
--------------------------------
State 4 main.py line 6
Violated property:
division by zero
x != 0
```

Verification of Ethereum Specifications



Handle `integer_squareroot` bound case #3600

Merged hwwhww merged 3 commits into `dev` from `integer_squareroot` on Feb 15, 2024

Conversation 4 Commits 3 Checks 15 Files changed 5

hwwhww commented on Feb 14, 2024 • edited Contributor

Credits to the University of Manchester Bounded Model Checking (BMC) project team: Bruno Farias, Youcheng Sun, and Lucas C. Cordeiro for reporting this issue! 🙏 💯

This team is an Ethereum Foundation ESP "Bounded Model Checking for Verifying and Testing Ethereum Consensus Specifications (FY22-0751)" project grantee. They used ESBMC model checker to find this issue.

Also, thanks to Mate Soos and Justin Traglia (@jtraglia) for helping verify the issue! 🙌

tl;dr

It's a spec bug, but it's impossible to produce it in the current mainnet.

Description

`integer_squareroot` raises ValueError exception when `n` is maxint of `uint64`, i.e., `2**64 - 1`.

Software Vulnerabilities in Python AI Libraries

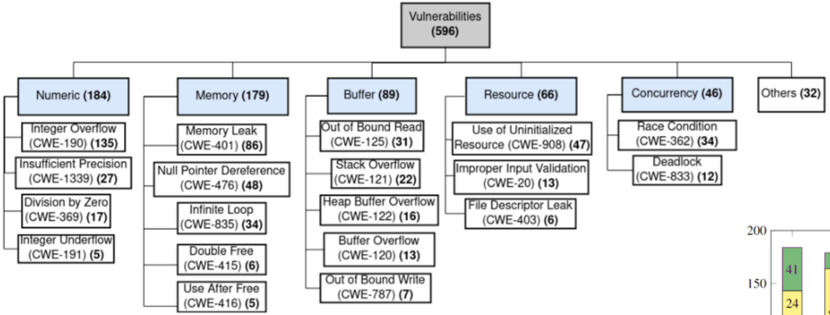

Vulnerabilities (596)
Numeric (184)
Integer Overflow (CWE-190) (135)
Insufficient Precision (CWE-1339) (27)
Division by Zero (CWE-369) (17)
Integer Underflow (CWE-191) (5)
Memory (179)
Memory Leak (CWE-401) (86)
Null Pointer Dereference (CWE-476) (48)
Infinite Loop (CWE-835) (34)
Double Free (CWE-415) (6)
Use After Free (CWE-416) (5)
Buffer (89)
Out of Bound Read (CWE-125) (31)
Stack Overflow (CWE-121) (22)
Heap Buffer Overflow (CWE-122) (16)
Buffer Overflow (CWE-120) (13)
Out of Bound Write (CWE-787) (7)
Resource (66)
Use of Uninitialized Resource (CWE-908) (47)
Improper Input Validation (CWE-20) (13)
File Descriptor Leak (CWE-403) (6)
Concurrency (46)
Race Condition (CWE-362) (34)
Deadlock (CWE-833) (12)
Others (32)

Figure 1: The taxonomy of vulnerability types studied in this work.

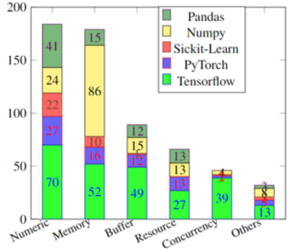


Figure 2: The distribution of software vulnerabilities in different ML libraries.

Harzevili et al. (2023). *Characterizing and Understanding Software Security Vulnerabilities in Machine Learning Libraries.*

NumPy Properties of Interest

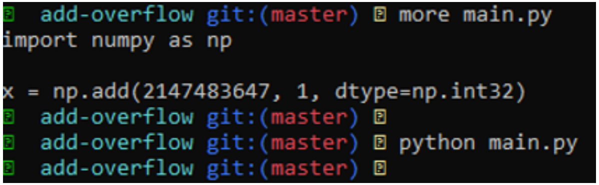
Property	Description	Check
Array shape consistency	Ensure operations on arrays are performed on arrays with compatible sizes.	Assert that shapes on input arrays match before performing operations.
Array element type consistency	Ensure array elements are of the expected type (e.g. all elements should be integers, float).	Use assertions or type checks to enforce element type consistency.
Index out of bounds	Ensure that all accesses to arrays are within bounds.	Assert that indices used for accessing arrays are within the bounds.
Mathematical Correctness	Ensure the correctness of numerical operations (addition, multiplication, etc)	Use assertions to check that the result of a calculation matches an expected value.
Abscence of Arithmetic Overflow	Ensure numeric operations do not cause overflow, particularly with custom bounded integer types.	Use symbolic execution to track integer values and ensure they stay within types bounds.

Verifying NumPy Programs with ESBMC

- Black-Box Verification with ESBMC
  - Analyze library behaviour via assertions from function calls.

main.py

```python
import numpy as np
x = np.add(2147483647, 1, dtype=np.int32)
```



```
▯  add-overflow git:(master) ▯ more main.py
import numpy as np

x = np.add(2147483647, 1, dtype=np.int32)
▯  add-overflow git:(master) ▯
▯  add-overflow git:(master) ▯ python main.py
▯  add-overflow git:(master) ▯
```

```
[Counterexample]
State 1 file main.py line 2
-----------------------------------
Violated property:
  file main.py line 2 column 0
  arithmetic overflow on add
  !overflow("+", 2147483647, 1)

VERIFICATION FAILED
```

Takeway messages

- Python presents several verification challenges, but we are already able to handle its static aspects and have validated our approach by reusing a BMC framework.
- NumPy and AI contain real-world issues, and our approach has the potential to uncover new ones.
- Our next steps focus on expanding language coverage, handling the dynamic aspects and integration with native libraries.

Thank you