Are a Static Analysis Tool Study's Findings Static? A Replication

David Liu, Jonathan Calver, Michelle Craig

Computer Science
UNIVERSITY OF TORONTO

Who we are

David Liu
david@cs.toronto.edu

Jonathan Calver
calver@cs.toronto.edu

Michelle Craig
mcraig@cs.toronto.edu

Background

- **Static analysis** is the analysis of code conducted without executing it
- We've been using an educational static analysis tool in our CS1/CS2 since 2016
- We present a conceptual replication of a 2017 study by Edwards et al.* that studied the errors reported by static analysis tools in an educational context

*Stephen H. Edwards, Nischel Kandru, and Mukund B.M. Rajagopal. 2017. Investigating Static Analysis Errors in Student Java Programs. In Proceedings of the 2017 ACM Conference on International Computing Education Research (Tacoma, Washington, USA) (ICER '17). Association for Computing Machinery, New York, NY, USA, 65–73. https://doi.org/10.1145/3105726.3106182

Research questions

- **RQ1** What are the most frequent static analysis errors in a set of student Python program submissions?
- **RQ2** Do the most frequent errors vary by student experience level?
- **RQ3** Which errors persist in students' final work?
- **RQ4** Are static analysis errors related to program grades?

Methods

CS1 course context

- Taught in Python
- 12-week semester
- 11 weekly programming exercises and 3 large programming assignments
- Static analysis tool incorporated into web autograder
  - worth 10-20%
- Students could run tool before the deadline
  - locally or through the web autograder

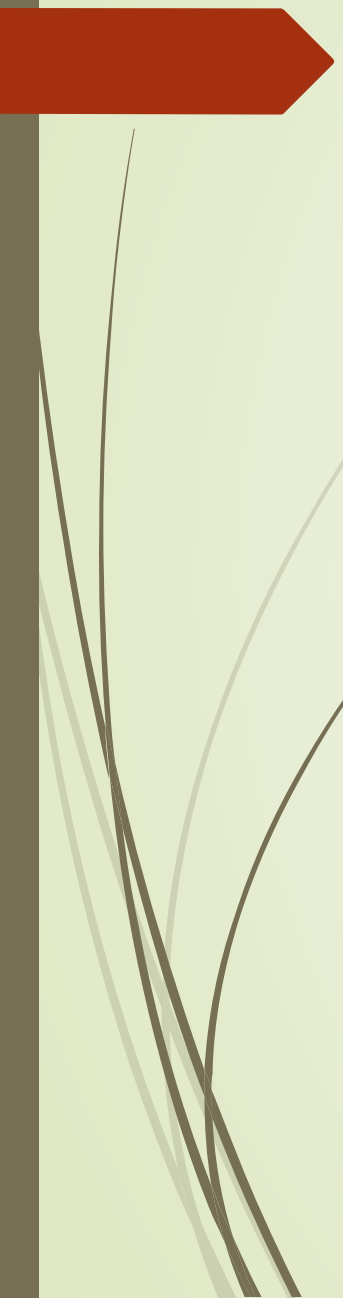
Offerings and Participants

- Study run in two CS1 offerings
- **1270** participants, **49,689** submissions, **161,012** errors

Prior experience	Participants	Interim submissions	Final submissions
All students	**1270**	**34,629**	**15,060**
No prior experience	662	17,677	7,588
A course before CS1	393	10,661	4790
Other prior experience	215	6291	2682

The PythonTA tool

- Free open-source Python package
- Wraps two professional-grade tools and implements custom checks
- Runnable in the terminal or through a Python API
- Customizable
  - Enable/disable specific checks
  - Set parameters (e.g., max line length)
  - Choose output format (text, HTML, JSON)
  - Override default error messages



```
PyTA Report for: C:\Users\David\Documents\my_file.py
= Code errors/forbidden usage (fix: high priority) =
E9996 (one-iteration)  Number of occurrences: 1.
  [Line 7] This loop will only ever run for one
       iteration
    5  def search(lst: list, item: Any) -> bool:
    6      """Return whether item is in lst."""
    7      for x in lst:
    8          if x == item:
    9              return True
   10          else:
   11              return False
   12
   13

= Style/convention errors (fix: before submission) =
No problems detected, good job!
```

Comparators to Edwards et al.

Dimension	Edwards et al.	Our study
Institution	R1, North American	R1, North American
Course	CS1, CS2, data structures	CS1
Programming language	Java	Python
Static analysis tools	pmd & checkstyle	PythonTA
Final (graded) submissions analysed?	Yes	Yes
Interim (ungraded) submissions analysed?	Yes	some

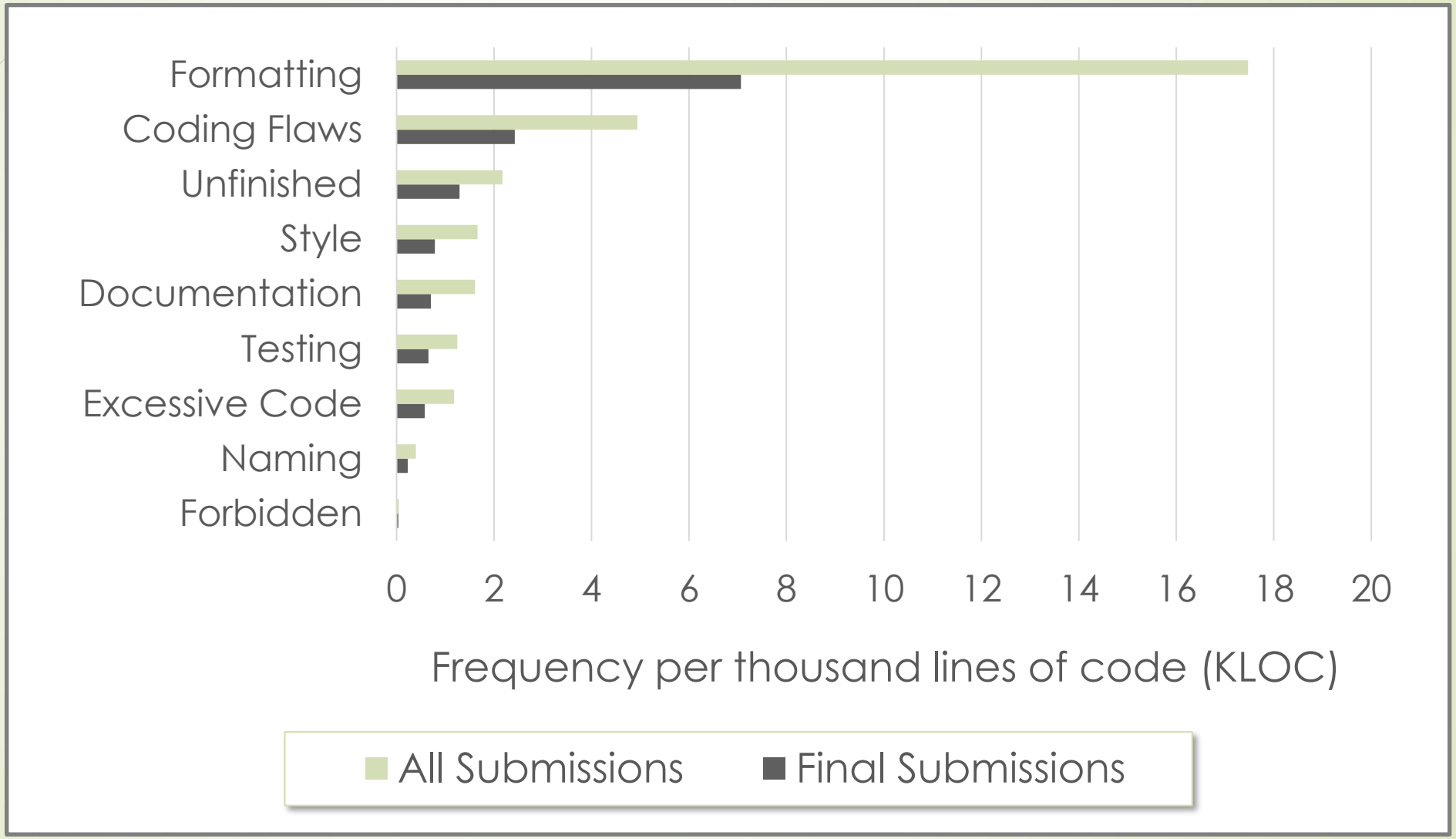
PythonTA error classification

Error category	Example error
Coding flaw	Undefined variable
Documentation	Missing function docstring
Excessive code	Too deeply-nested blocks
Forbidden*	Forbidden module imports
Formatting	Missing whitespace around operators
Naming	Naming convention violations
Style	Simplifiable if conditions
Testing	Formatting of doctest examples
Unfinished*	Unused function parameter

*category not present in Edwards et al.

Results

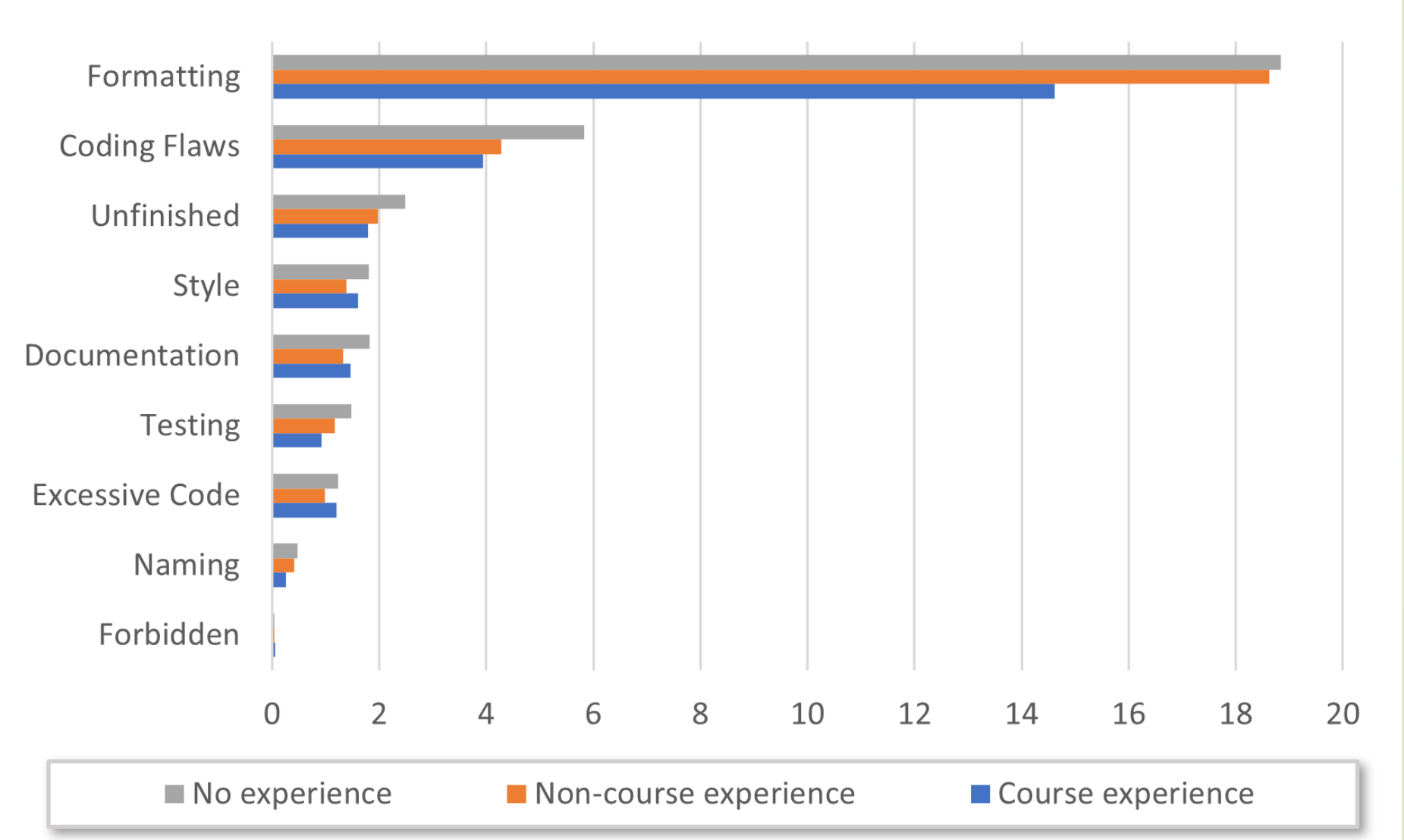
Error category frequencies (RQ1, RQ3)



Most frequent errors, frequencies per KLOC (RQ1, RQ3)

Category	Error	All subs.	Final subs
Formatting	**Formatting linter error**	**14.15**	**5.56**
Formatting	**Line too long**	**2.67**	**1.18**
Unfinished	Unused function parameter	1.66	1.17
Testing	Missing space in doctest	1.25	< 1.0
Coding Flaws	Undefined variable	1.20	< 1.0
Coding Flaws	Missing return statements	1.09	< 1.0
Excessive Code	Too many branches	< 1.0	< 1.0
Coding Flaws	Possibly undefined variable	< 1.0	< 1.0
Documentation	Missing docstring	< 1.0	< 1.0
Documentation	Missing type annotation	< 1.0	< 1.0
Style	Unnecessary indexing	< 1.0	< 1.0

Category frequencies by prior experience group (RQ3)



Relationship to correctness grades (RQ4)

- The number of PythonTA errors was negatively correlated with the percentage of correctness test cases passed.

- Presence of Coding Flaws is associated with differences in correctness measure—even when they only appear in interim submissions!

Group	Mean % tests passed
Never had a Coding Flaw	86.6
Final submission had no Coding Flaws, but an interim submission had a Coding Flaw	82.5
Final submission had a Coding Flaw	60.4

Takeaways and future work

Formatting errors dominate!

- Formatting errors were more frequent than all other errors **combined**
- Teaching how to use an autoformatter helped… somewhat

Other takeaways; limitations

- Coding flaws were second-most common error category
- (Most) students fixed (most) PythonTA errors in final submissions
  - **83%** final submissions with **0** errors
  - **95%** final submissions with **< 5** errors
- Novice programmers made more errors than programmers with prior experience
- Limitations:
  - Did not have access to local runs of PythonTA
  - Introduction of autoformatter was not a formal intervention

Conclusions and future work

- Static analysis tools can be used to detect a wide range of issues in student code, including code correctness.
- Formatting issues dominate!
  - Autoformatters can be useful—if students use them.
- More work to be done investigating the "tail" of final submissions.

Thank you!

David Liu
david@cs.toronto.edu

Jonathan Calver
calver@cs.toronto.edu

Michelle Craig
mcraig@cs.toronto.edu