



Python FFI 的陰暗角落

PyCon TW 2025

scc@scc.tw



SCC

SWE @ cycraft

C++ Lover





Facebook · 奧義智慧科技 CyCraft Technology

超過 10 個回應 · 2 年前

奧義冰淇淋也會出沒在會場，為大家的週末時光增
各大資訊社群活動中 出沒的「**奧義冰淇淋**」，也將和濁水溪以南最大的
一起強勢回歸啦 炎炎夏日，讓**奧義冰淇淋**陪您度過逛展會的美好時 ...



Facebook

<https://www.facebook.com> · photo

奧義智慧科技是世界頂尖的AI 資安公司，除了 ...
【來MOPCON 想吃奧義冰淇淋~】今晚，我想來點... **奧義智慧**贊助的
技是世界頂尖的AI 資安公司，除了以獲美國MITRE ATT&CK 評測最佳位



HackMD

<https://hackmd.io> ...

轉生到資安公司的我，竟開啟了遊戲人生？！

奧義智慧科技，一般人眼中的奧義，世界頂尖的台灣自研技術AI 資安公司
奧義，**冰淇淋**公司，各大conf 都能看到奧義的**冰淇淋**，遊戲公司，兩款桌遊



Instagram · mopcon.tw

1 年前

MOPCON 行動科技年會on Instagram: "👉 奧義智慧at ..."

奧義智慧再次出沒濁水溪以南最大的行動科技年會MOPCON 啦 今年很遺憾地沒辦法帶來經典的奧義**冰淇淋**，但聽說可以到**奧義智慧**攤位免費獲得大受好評的資安桌 ...



先別說冰淇淋了，
你聽過**奧義智慧**嗎？





Facebook · 奧義智慧科技 CyCraft Technology

超過 10 個回應 · 2 年前

奧義冰淇淋也會出沒在會場，為大家的週末時光增

各大資訊社群活動中 出沒的「奧義冰淇淋」，也將和濁水溪以南最大的
一起強勢回歸啦 炎炎夏日，讓奧義冰淇淋陪您度過逛展會的



Facebook

<https://www.facebook.com> photo

奧義智慧科技是世界頂尖的AI 資安公司， 【來MOPCON 想吃奧義冰淇淋～】今晚，我想來點... 奧義 技是世界頂尖的AI 資安公司，除了以獲美國MITRE ATT&C



HackMD

<https://hackmd.io> ...

轉生到資安公司的我，竟開啟了遊戲人生

奧義智慧科技. 一般人眼中的奧義. 世界頂尖的台灣自研技術
奧義. 冰淇淋公司. 各大conf 都能看到奧義的冰淇淋. 遊戲公



Instagram · mopcon.tw

1年前

MOPCON 行動科技年會on Instagram: "

奧義智慧再次出沒濁水溪以南最大的行動科技年會MOPCO
來經典的奧義冰淇淋，但聽說可以到奧義智慧攤位免費獲得



CYBERCANS



為了突出不同產業對於資產防禦的側重，
奧義智慧推出八張產業角色卡，分別為半
導體業、金融業、政府單位、能源產業、
黑色產業、遊戲業、電商產業、醫療業。

《Cybercans 2：紅色動員令》內除了產
業角色卡，還能結合第一代的卡牌擴充
出全新玩法，可以用更快速、刺激的方
式進行攻防體驗。



Download The Slide



Outline

- What is Python FFI
- The Hidden Corner
 - Benchmark ctypes, cffi, pybind11, PyO3
 - Why Free-Threaded Makes Them Slower?
 - Why PyO3 fast, but ctypes slow?
 - Racing FFI after No-GIL
 - Higher Risk for Arena Leakage in GLIBC

Background knowledge

- GIL
- Race conditions
- Memory Arena



Slides are heavy, listen first.



The hidden corner you might be aware

- Application level:
 - **GIL Semantics**, **Thread Pinning**, Refcount Edges & Ownership, Error Propagation, Reentrancy...
- OS level:
 - **Race Conditions**, **Dynamic Loading** / lazy binding, **System Libraries Features**, Loader Search Path, Permissions...
- Hardware level:
 - **ABI Compatibility**, **Cache Locality**, Modern CPU Affinity Binding, TLB, Branch Prediction, I/O Topology...

The hidden corner you might be aware

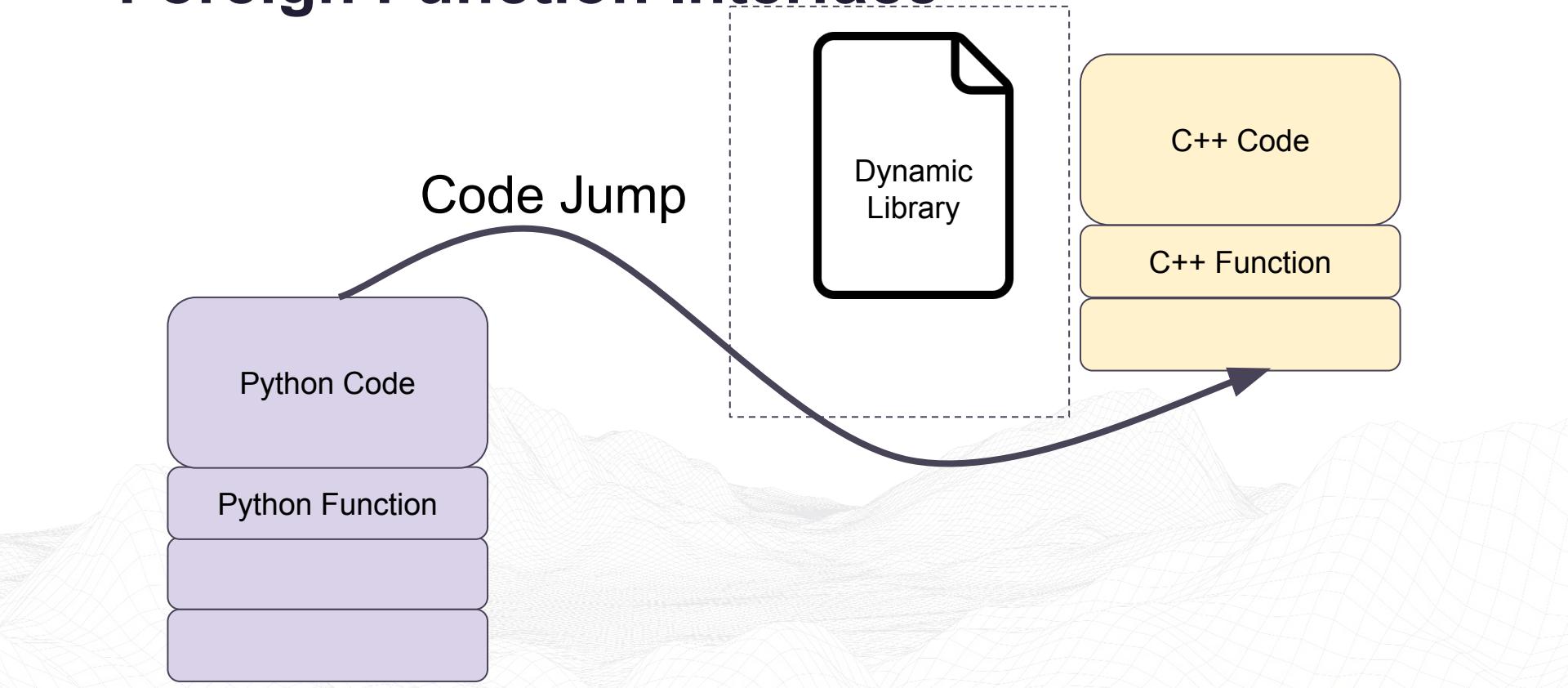
- 💪 Reliability Issues:
 - **GIL Semantics, Thread Pinning, Race Conditions, ABI Compatibility,**
- 🚀 Performance Issues:
 - **Dynamic Loading, System Libraries Features, Cache Locality,**



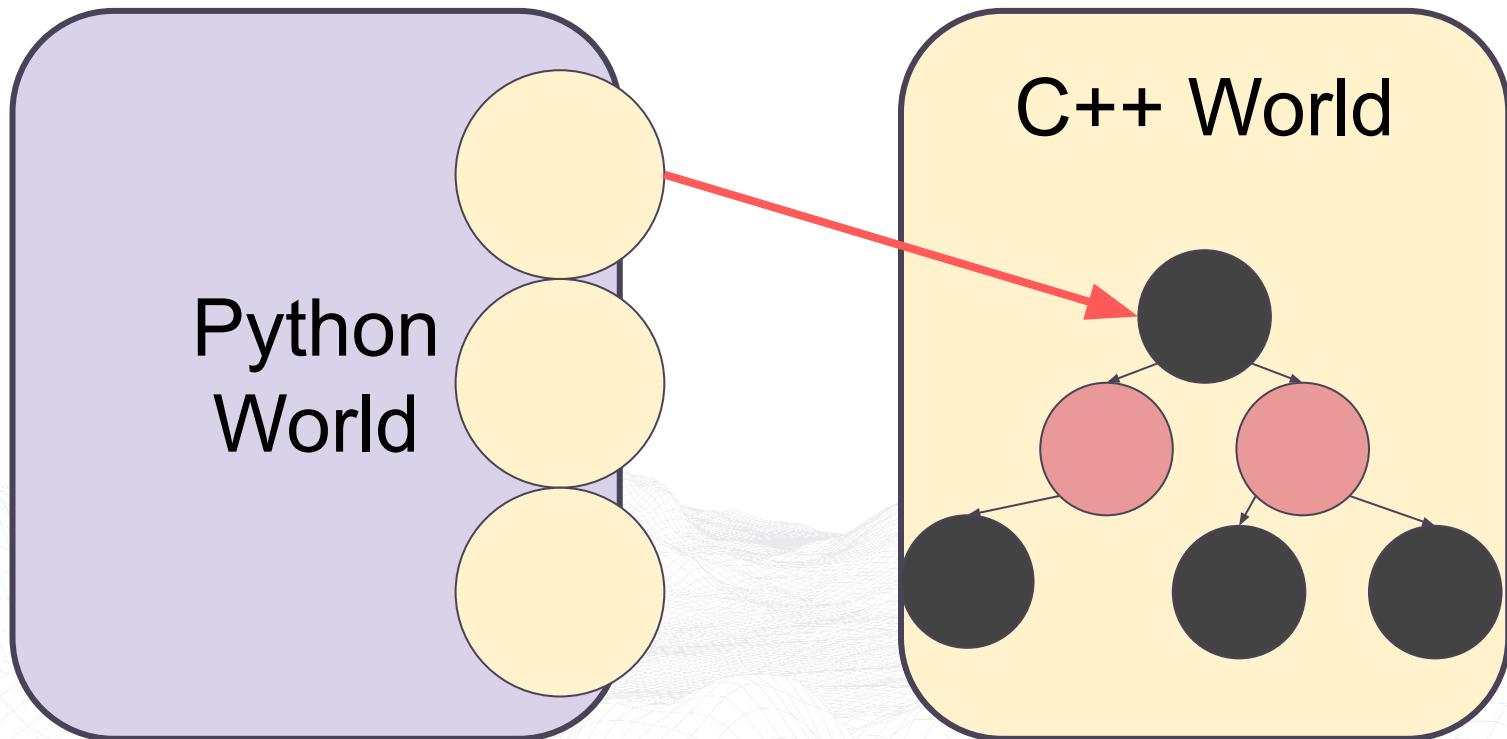
What is Python FFI



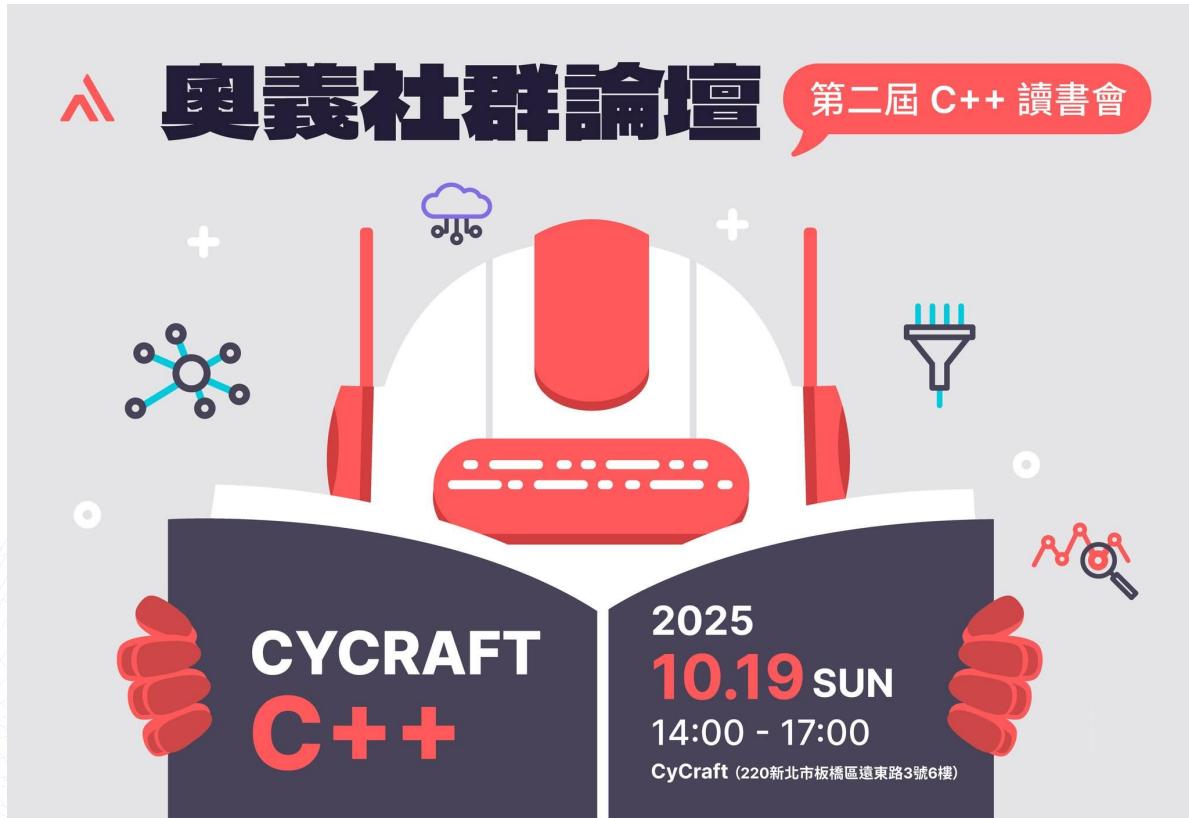
Foreign Function Interface



Why this issue



Why this issue





The Hidden Corner



Benchmark ctypes, cffi, pybind11, PyO3

Linux 6.15.9-arch1-1, isolcpus=2-7 nohz_full=2-7 rcu_nocbs=2-7

i5-13500 (2.5G lock freq), 32G DDR4-3200

Disable Turbo Boost, SMT/Hyper-Threading, ASLR

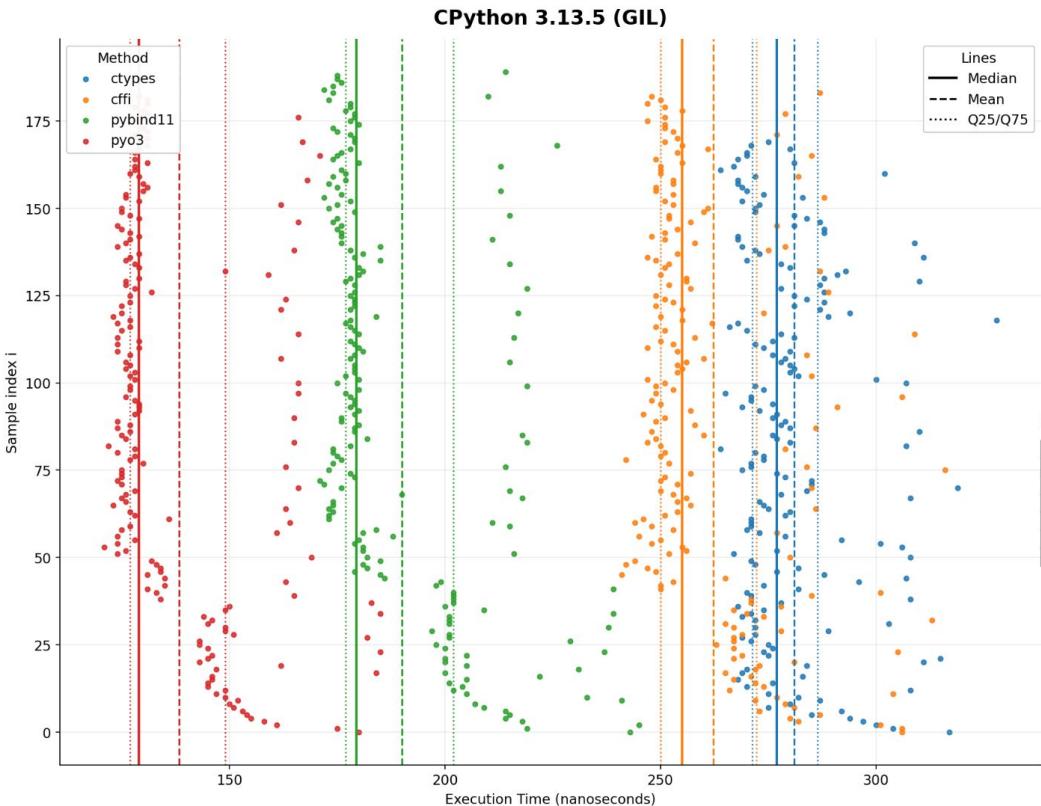
numactl --physcpubind=0-7 --membind=0

Huge pages 128k

GCC 15.1.1 20250729, Glibc 2.42, Rust 1.88.0

C_Python: 3.13.5, 3.14.0rc1

Benchmark ctypes, cffi, pybind11, PyO3



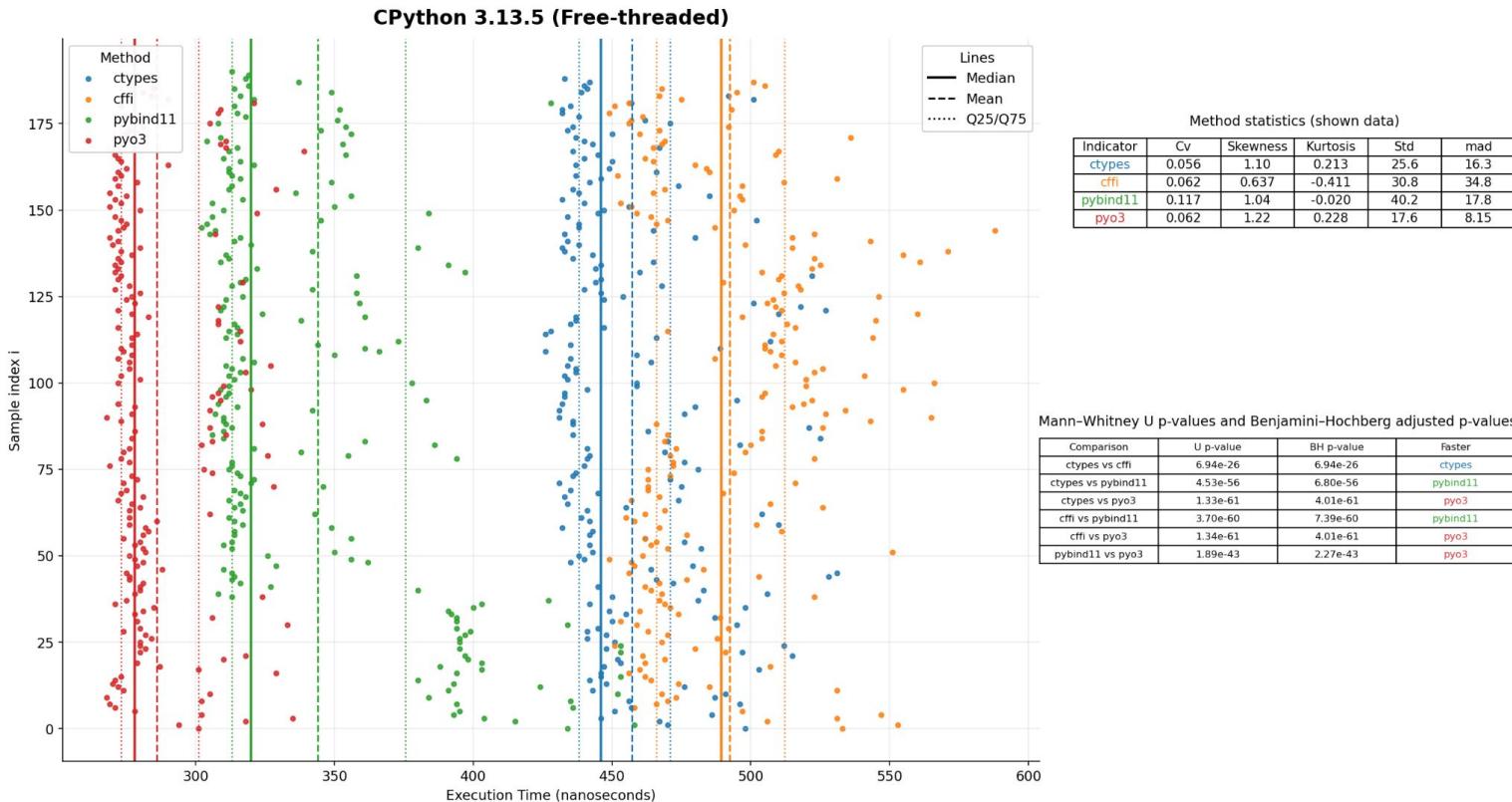
Method statistics (shown data)

Indicator	Cv	Skewness	Kurtosis	Std	mad
ctypes	0.047	1.30	1.01	13.1	8.90
cffi	0.063	1.22	0.782	16.4	8.90
pybind11	0.098	1.14	0.182	18.6	7.41
pyo3	0.119	1.15	0.121	16.5	5.93

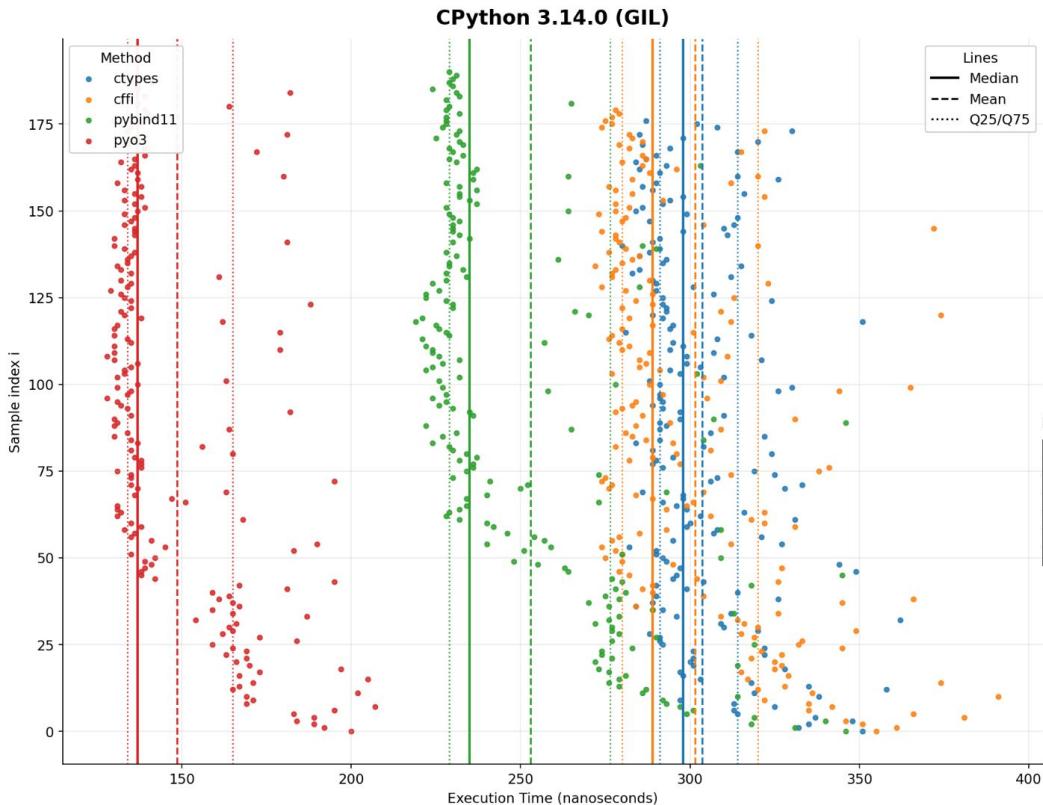
Mann-Whitney U p-values and Benjamini-Hochberg adjusted p-values

Comparison	U p-value	BH p-value	Faster
ctypes vs cffi	9.71e-25	9.71e-25	cffi
ctypes vs pybind11	1.30e-61	2.59e-61	pybind11
ctypes vs pyo3	3.55e-63	1.58e-62	pyo3
cffi vs pybind11	3.90e-60	5.84e-60	pybind11
cffi vs pyo3	5.28e-63	1.58e-62	pyo3
pybind11 vs pyo3	8.13e-49	9.75e-49	pyo3

Benchmark ctypes, cffi, pybind11, PyO3



Benchmark ctypes, cffi, pybind11, PyO3

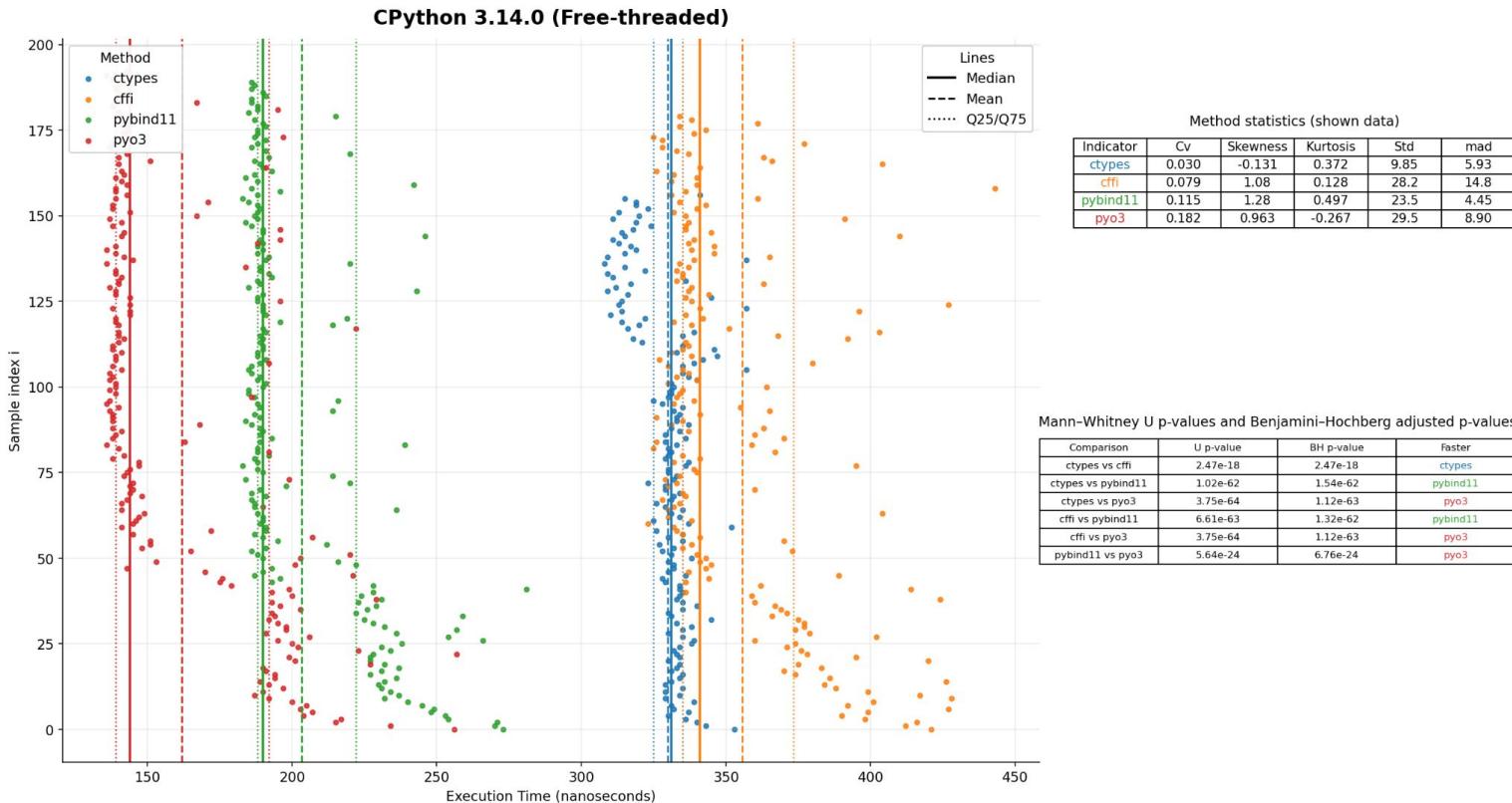


Method statistics (shown data)					
Indicator	Cv	Skewness	Kurtosis	Std	mad
ctypes	0.056	1.14	0.803	17.0	13.3
cffi	0.088	1.12	0.561	26.6	17.8
pybind11	0.123	1.04	0.149	31.0	16.3
pyo3	0.139	1.05	-0.165	20.6	7.41

Mann-Whitney U p-values and Benjamini-Hochberg adjusted p-values

Comparison	U p-value	BH p-value	Faster
ctypes vs cffi	1.18e-03	1.18e-03	cffi
ctypes vs pybind11	3.46e-42	5.18e-42	pybind11
ctypes vs pyo3	3.88e-64	1.17e-63	pyo3
cffi vs pybind11	1.12e-36	1.34e-36	pybind11
cffi vs pyo3	3.84e-64	1.17e-63	pyo3
pybind11 vs pyo3	1.24e-62	2.48e-62	pyo3

Benchmark ctypes, cffi, pybind11, PyO3



Benchmark ctypes, cffi, pybind11, PyO3



Why Free-Threaded Makes Them Slower?

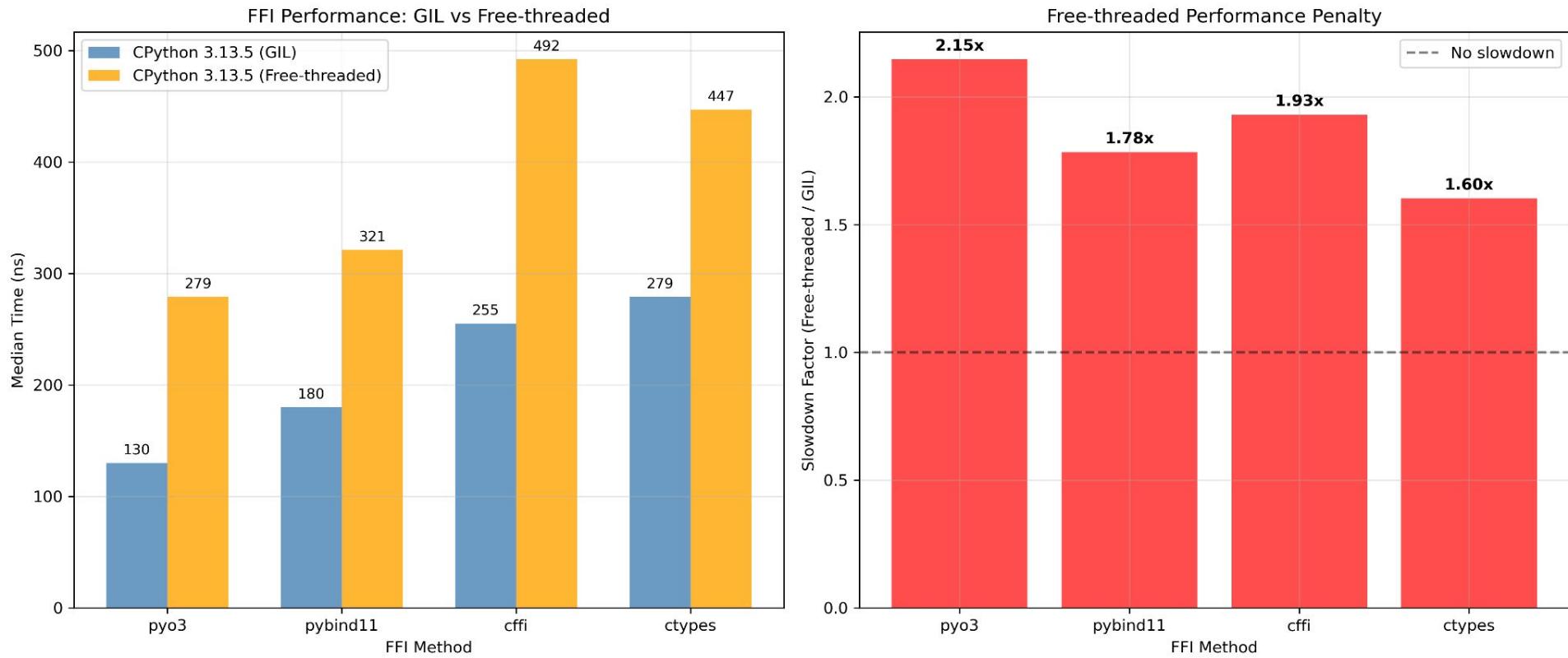


Why PyO3 fast, but ctypes slow?

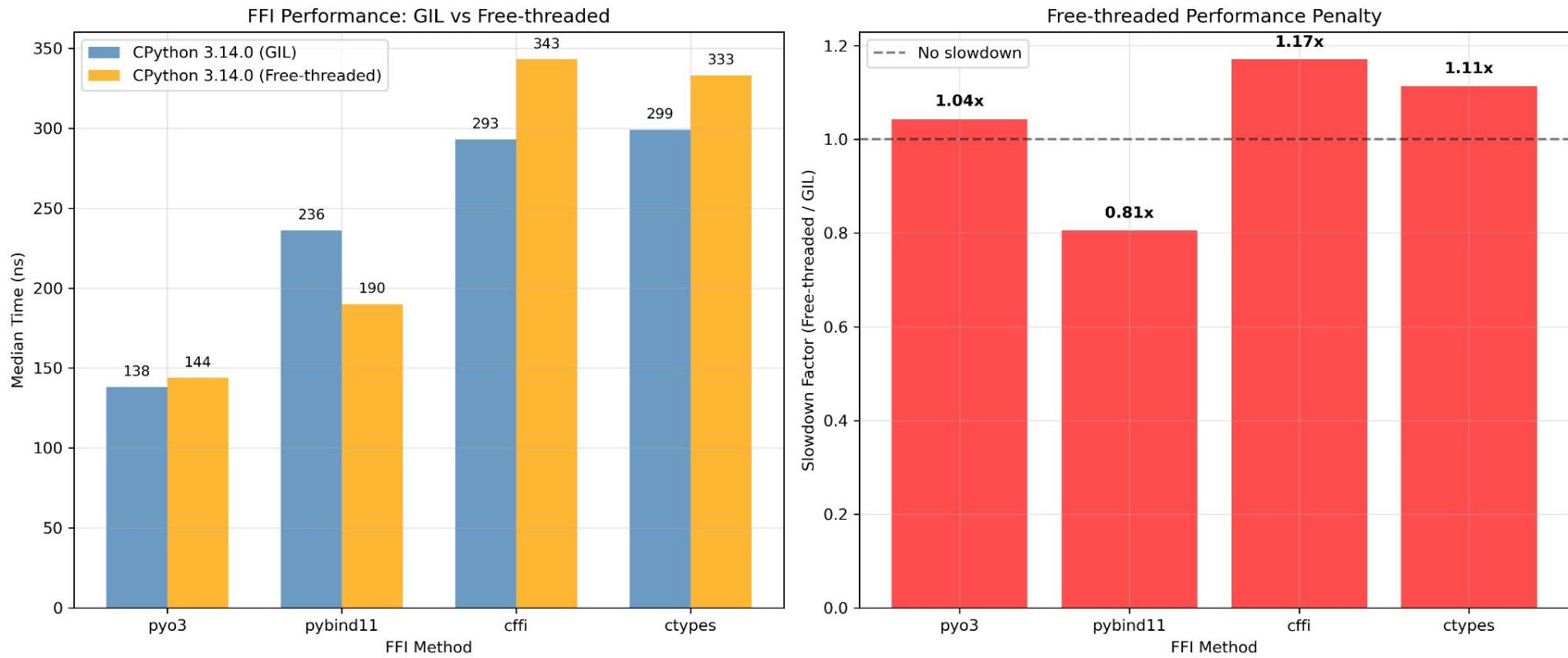
Why Free-Threaded Makes Them Slower?



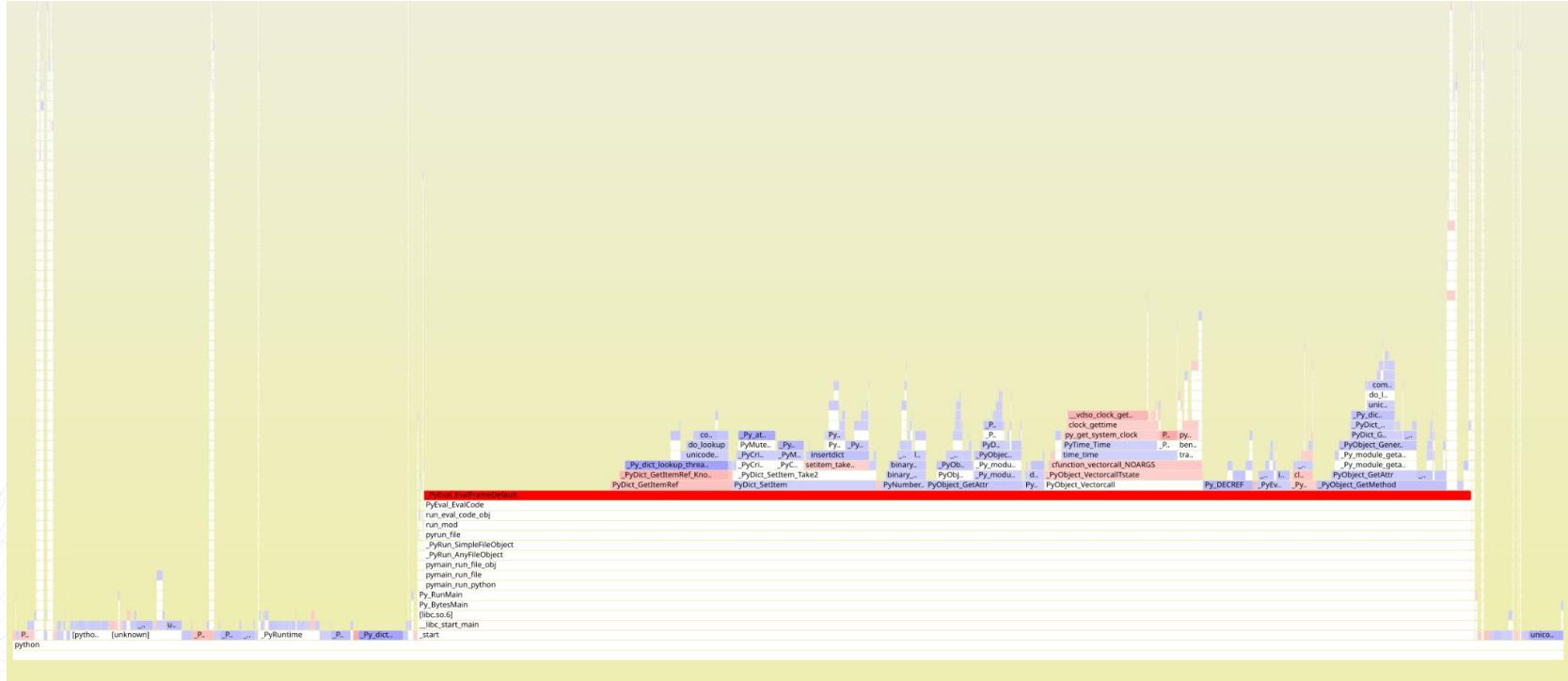
Why Free-Threaded Makes Them Slower?



Why Free-Threaded Makes Them Slower?



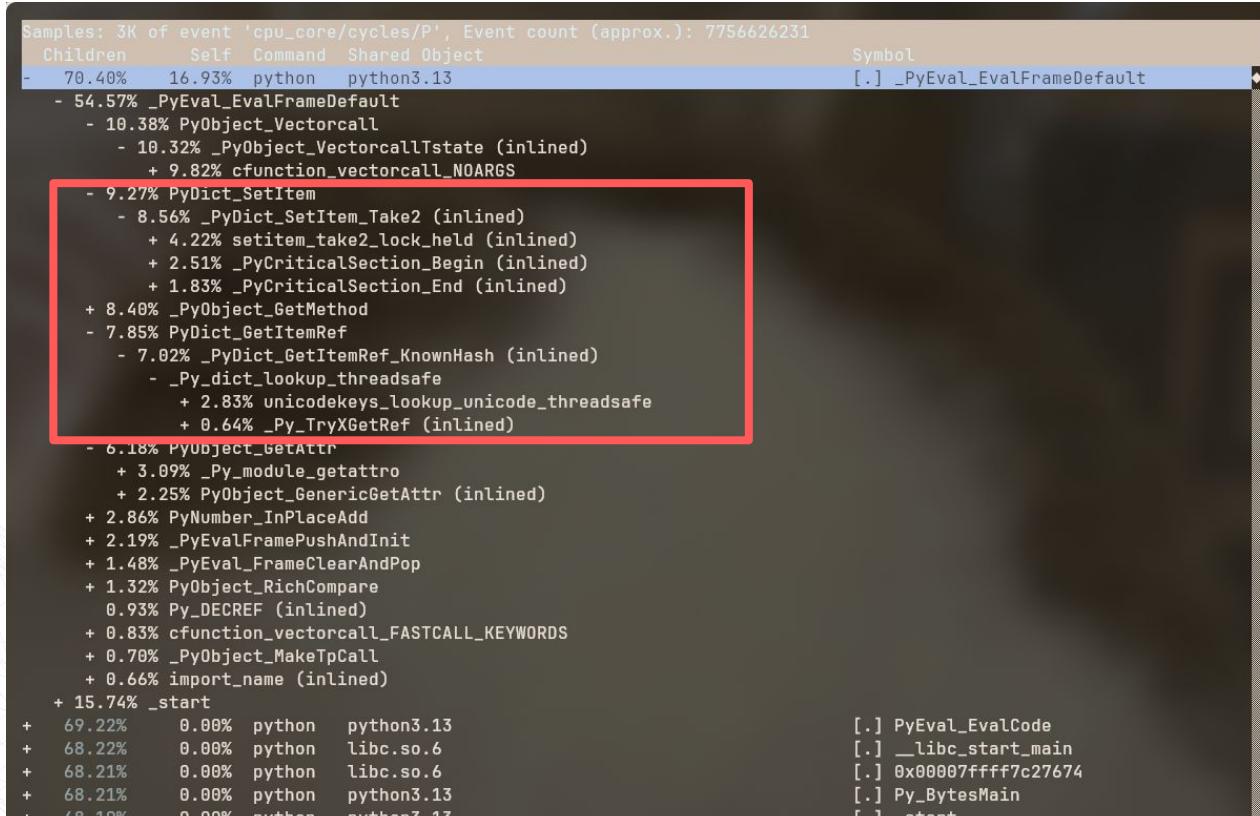
Why Free-Threaded Makes Them Slower?



Why Free-Threaded Makes Them Slower?

```
scc ▾ ▷ ~ /git/pycon2025-ffi-hidden-corner
❯❯❯ for F in "$P_GIL" "$P_NOGIL"; do
    echo "==== $(basename $F) ===="
    perf report --stdio --no-children -i "$F" --sort symbol \
        | grep -E 'futex|pthread_mutex|_PyCriticalSection|_Py_mutex|PyMutex|_Py_atomic|__x64_sys_|do_syscall|do_futex' \
        | sed 's/^/ /'
done | tee pyo3_lock_syscall_evidence.txt
==== pyo3_profile_1754689040.perf.data ====
==== pyo3_profile_1754689074.perf.data ====
    --0.77%--_Py_atomic_store_uint32_relaxed (inlined)
    --4.09%--_Py_atomic_compare_exchange_uint8 (inlined)
        |--2.41%--PyMutex_LockFast (inlined)
        |     _PyCriticalSection_BeginMutex (inlined)
        |     _PyCriticalSection_Begin (inlined)
        --1.67%--_PyMutex_Unlock (inlined)
            _PyCriticalSection_End (inlined)
    --0.61%--_Py_atomic_load_uint32_relaxed (inlined)
```

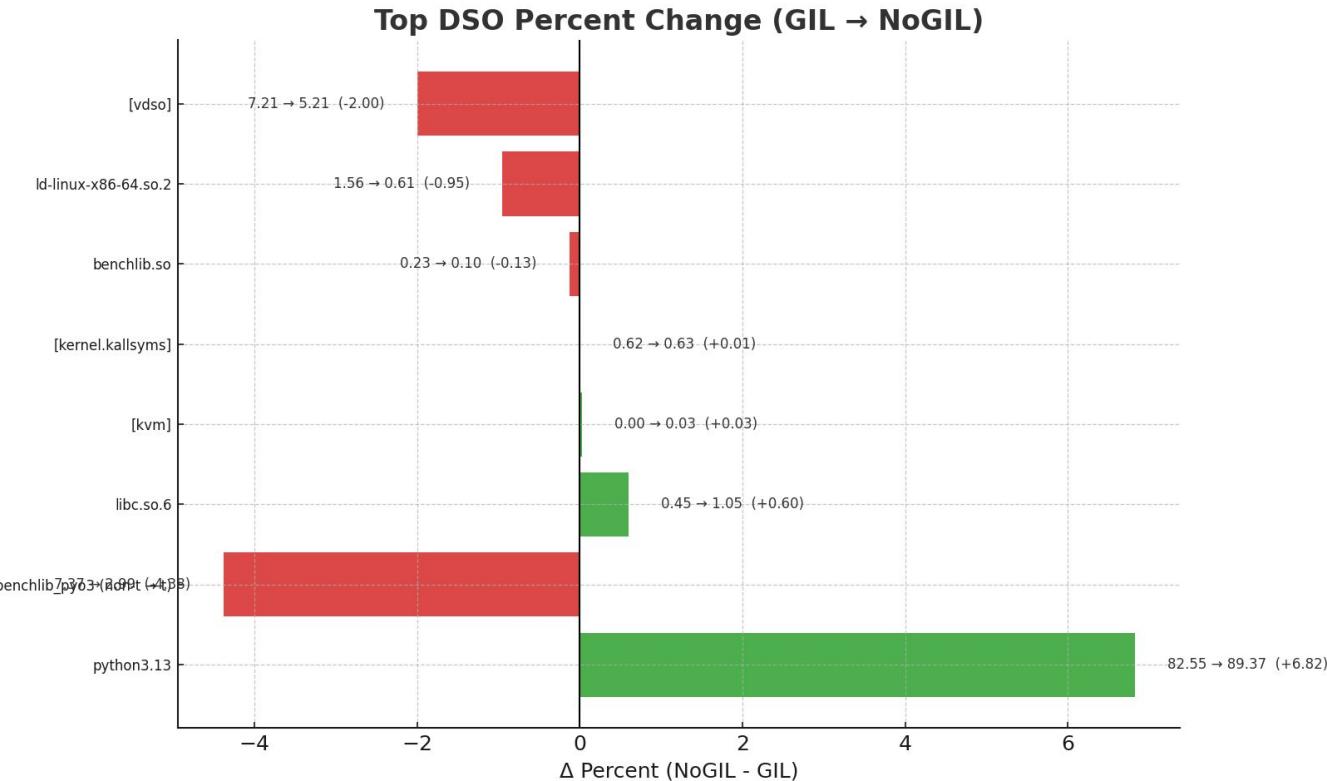
Why Free-Threaded Makes Them Slower?



Why Free-Threaded Makes Them Slower?

- 9.27% `PyDict_SetItem`
 - 8.56% `_PyDict_SetItem_Take2` (inlined)
 - + 4.22% `setitem_take2_lock_held` (inlined)
 - + 2.51% `_PyCriticalSection_Begin` (inlined)
 - + 1.83% `_PyCriticalSection_End` (inlined)
- + 8.40% `_PyObject_GetMethod`
- 7.85% `PyDict_GetItemRef`
 - 7.02% `_PyDict_GetItemRef_KnownHash` (inlined)
 - `_Py_dict_lookup_threadsafe`
 - + 2.83% `unicodekeys_lookup_unicode_threadsafe`
 - + 0.64% `_Py_TryXGetRef` (inlined)

Why Free-Threaded Makes Them Slower?



Recap

- Python 3.13-NoGIL shows higher overhead than 3.13-GIL when crossing FFI boundaries.
- Python 3.14-NoGIL achieves perf close to 3.14-GIL for FFI calls.
- Lock-free execution does not guarantee speedups
in some cases the locked implementation is faster.
- Unicode translation carries a significant overhead in NoGIL mode.

Benchmark ctypes, cffi, pybind11, PyO3



Why Free-Threaded Makes Them Slower?



Why PyO3 fast, but ctypes slow?

Why PyO3 fast, but ctypes slow?

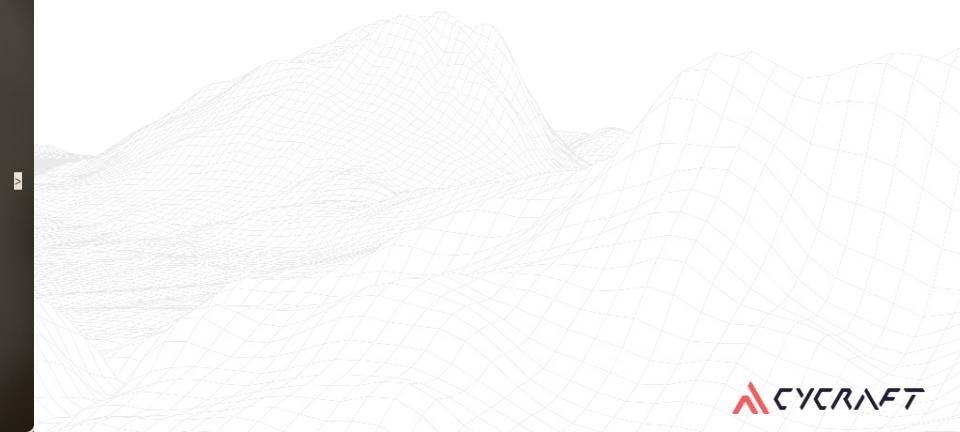


```
26.11% [.] _PyEval_EvalFrameDefault
| --22.56%--_PyEval_EvalFrameDefault
| |
| | --19.50%--_PyEval_EvalCode
| | |
| | | --19.11%--run_eval_code_obj
| | | run_mod
| | | pyrun_file
| | | _PyRun_SimpleFileObject
| | | _PyRun_AnyFileObject
| | | pymain_run_file_obj (inlined)
| | | pymain_run_file (inlined)
| | | pymain_run_python (inlined)
| | | Py_RunMain
| | | Py_BytesMain
| | | 0x7fffff7c27674
| | | __libc_start_main
| | |
| | | _start
| |
| | --0.54%--_PyFunction_Vectordcall (inlined)
| | | _PyObject_VectordcallDictTstate (inlined)
| | | _PyObject_Call_Prepnd
| |
| | --0.53%--_PyObject_VectordcallTstate (inlined)
|
| --1.42%--Py_INCREF (inlined)
| |
| | --1.25%--_PyEval_EvalFrameDefault
| | |
| | | --1.03%--_PyEval_EvalCode
| | |
| | | --0.96%--run_eval_code_obj
| | | run_mod
| | | pyrun_file
| | | _PyRun_SimpleFileObject
| | | _PyRun_AnyFileObject
| | | pymain_run_file_obj (inlined)
| | | pymain_run_file (inlined)
| | | pymain_run_python (inlined)
| | | Py_RunMain
| | | Py_BytesMain
| | | 0x7fffff7c27674
| | | __libc_start_main
| |
| | | _start
|
11.76% [.] _Py_dict_lookup
| --5.17%--_Py_dict_lookup
| |
| | --2.38%--_PyDict_GetItemRef_KnownHash
| | |
| | | --2.18%--PyDict_GetItemRef
| | | _PyEval_EvalFrameDefault
| | | PyEval_EvalCode
| |
| | | --2.16%--run_eval_code_obj
| | | run_mod
```

perf report -stdio -i "\$PYO3" --no-children --sort symbol

Why PyO3 fast, but ctypes slow?

PyO3



16.36%

```
[.] _PyEval_EvalFrameDefault  
|  
--14.26%--_PyEval_EvalFrameDefault  
|  
|   |--12.04%--_PyEval_EvalCode  
|   |  
|   |   |--11.74%--run_eval_code_obj  
|   |   |  
|   |   |   run_mod  
|   |   |   pyrun_file  
|   |   |   _PyRun_SimpleFileObject  
|   |   |   _PyRun_AnyFileObject  
|   |   |   pymain_run_file_obj (inlined)  
|   |   |   pymain_run_file (inlined)  
|   |   |   pymain_run_python (inlined)  
|   |   |   Py_RunMain  
|   |   |   Py_BytesMain  
|   |   |   0x7ffff7c27674  
|   |   |   __libc_start_main  
|   |   |  
|   |   |  
|   |   |   _start  
|  
|  
--0.63%--_PyObject_VectorcallTstate (inlined)  
  
--0.66%--Py_INCREF (inlined)
```

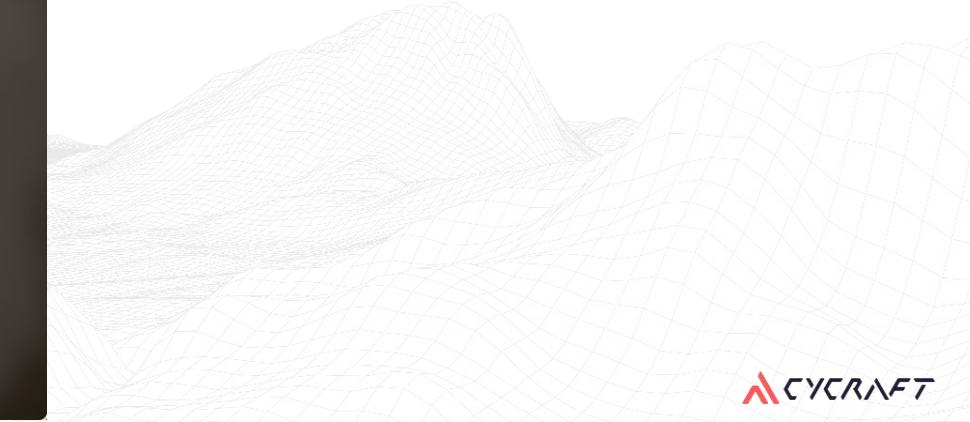
6.99% [.] pthread_mutex_lock

```
|  
--4.93%--take_gil  
|  
|   |--_PyThreadState_Attach  
|   |  
|   |   _call_function_pointer (inlined)  
|   |  
|   |   _ctypes_callproc (inlined)  
|   |  
|   |   PyCFuncPtr_call  
|   |  
|   |   _PyObject_MakeTpCall  
|   |  
|   |   _PyEval_EvalFrameDefault  
|   |  
|   |   PyEval_EvalCode  
|   |  
|   |   run_eval_code_obj  
|   |  
|   |   run_mod  
|   |  
|   |   pyrun_file  
|   |  
|   |   _PyRun_SimplefileObject  
|   |  
|   |   _PyRun_AnyfileObject  
|   |  
|   |   pymain_run_file_obj (inlined)  
|   |  
|   |   pymain_run_file (inlined)  
|   |  
|   |   pymain_run_python (inlined)  
|   |  
|   |   Py_RunMain  
|   |  
|   |   Py_BytesMain  
|   |  
|   |   0x7ffff7c27674  
|   |  
|   |   __libc_start_main  
|   |  
|  
--1.86%--drop_gil_impl (inlined)  
|  
|   |--drop_gil (inlined)  
|   |  
|   |   _PyEval_ReleaseLock  
|   |  
|   |   PyEval_SaveThread  
|  
|  
--1.84%--_call_function_pointer (inlined)  
|  
|   |--_ctypes_callproc (inlined)  
|   |  
|   |   PyCFuncPtr_call  
|   |  
|   |   _PyObject_MakeTpCall  
|   |  
|   |   _PyEval_EvalFrameDefault
```

perf report --stdio --no-children -i "\$CTYPES" --sort symbol

Why PyO3 fast, but ctypes slow?

ctypes



16.36% [.] _PyEval_EvalFrameDefault

```
--14.26%--_PyEval_EvalFrameDefault
|
|   |--12.04%--_PyEval_EvalCode
|   |
|   |   |--11.74%--run_eval_code_obj
|   |   |   run_mod
|   |   |   pyrun_file
|   |   |   _PyRun_SimpleFileObject
|   |   |   _PyRun_AnyFileObject
|   |   |   pymain_run_file_obj (inlined)
|   |   |   pymain_run_file (inlined)
|   |   |   pymain_run_python (inlined)
|   |   |   Py_RunMain
|   |   |   Py_BytesMain
|   |   |   0x7ffff7c27674
|   |   |   __libc_start_main
|   |   |   _start
|   |
|   |--0.63%--_PyObject_VectorcallTstate (inlined)
|
--0.66%--Py_INCREF (inlined)
```

6.99% [.] pthread_mutex_lock

```
--4.93%--take_gil
|
|   |--PyThreadState_Attach
|   |   _call_function_pointer (inlined)
|   |   _ctypes_callproc (inlined)
|   |   PyCFuncPtr_call
|   |   _PyObject_MakeTpCall
|   |   _PyEval_EvalFrameDefault
|   |
|   |   pyeval_EvalCode
|   |   run_eval_code_obj
|   |   run_mod
|   |   pyrun_file
|   |   _PyRun_SimpleFileObject
|   |   _PyRun_AnyFileObject
|   |   pymain_run_file_obj (inlined)
|   |   pymain_run_file (inlined)
|   |   pymain_run_python (inlined)
|   |   Py_RunMain
|   |   Py_BytesMain
|   |   0x7ffff7c27674
|   |   __libc_start_main
|   |   _start
|
--1.86%--drop_gil_impl (inlined)
|   drop_gil (inlined)
|   _PyEval_ReleaseLock
|   PyEval_SaveThread
|
|   |--1.84%--_call_function_pointer (inlined)
|   |   _ctypes_callproc (inlined)
|   |   PyCFuncPtr_call
|   |   _PyObject_MakeTpCall
|   |   _PyEval_EvalFrameDefault
```

Why PyO3 fast, but ctypes slow?

ctypes

6.99% [.] pthread_mutex_lock

```
--4.93%--take_gil
|
|   |--PyThreadState_Attach
|   |   _call_function_pointer (inlined)
|   |   _ctypes_callproc (inlined)
|   |   PyCFuncPtr_call
|   |   _PyObject_MakeTpCall
|   |   _PyEval_EvalFrameDefault
```



perf

Why PyO3 fast, but ctypes slow?

Loader	GIL Released?	Best for	Risk
CDLL	Yes	Long-running C code, multi-thread	Unsafe for Python C API calls
PyDLL	No	Short C calls, Python C API usage, single-thread	Blocks other threads, can deadlock

```
A = ctypes.CDLL(_lib_path)
```

```
B = ctypes.PyDLL(_lib_path)
```

Why PyO3 fast, but ctypes slow?

```
scc ▾ ▶ ⟲ /tmp
)~/git/pycon2025-ffi-hidden-corner/cpython3.14.0rc1-gil/bin/python3 ./cdll_vs_pydll.py
CDLL  getpid x2000000: 0.643s
PyDLL  getpid x2000000: 0.526s
Speedup (PyDLL vs CDLL): 1.222x
```

```
scc ▾ ▶ ⟲ /tmp
)~/git/pycon2025-ffi-hidden-corner/cpython3.14.0rc1-nogil/bin/python3 ./cdll_vs_pydll.py
CDLL  getpid x2000000: 0.581s
PyDLL  getpid x2000000: 0.533s
Speedup (PyDLL vs CDLL): 1.089x
```

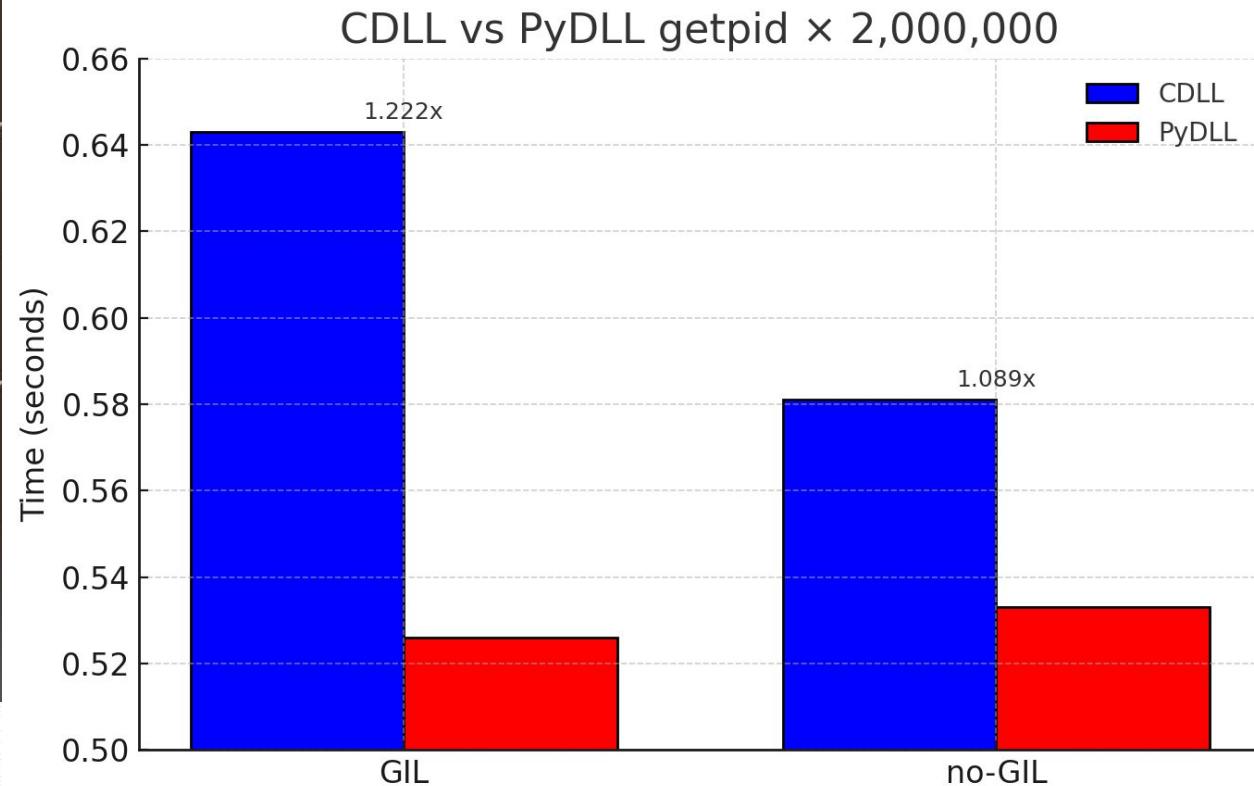
```
scc ▾ ▶ ⟲ /tmp
)~|
```

Why PyO3 fast, but ctypes slow?

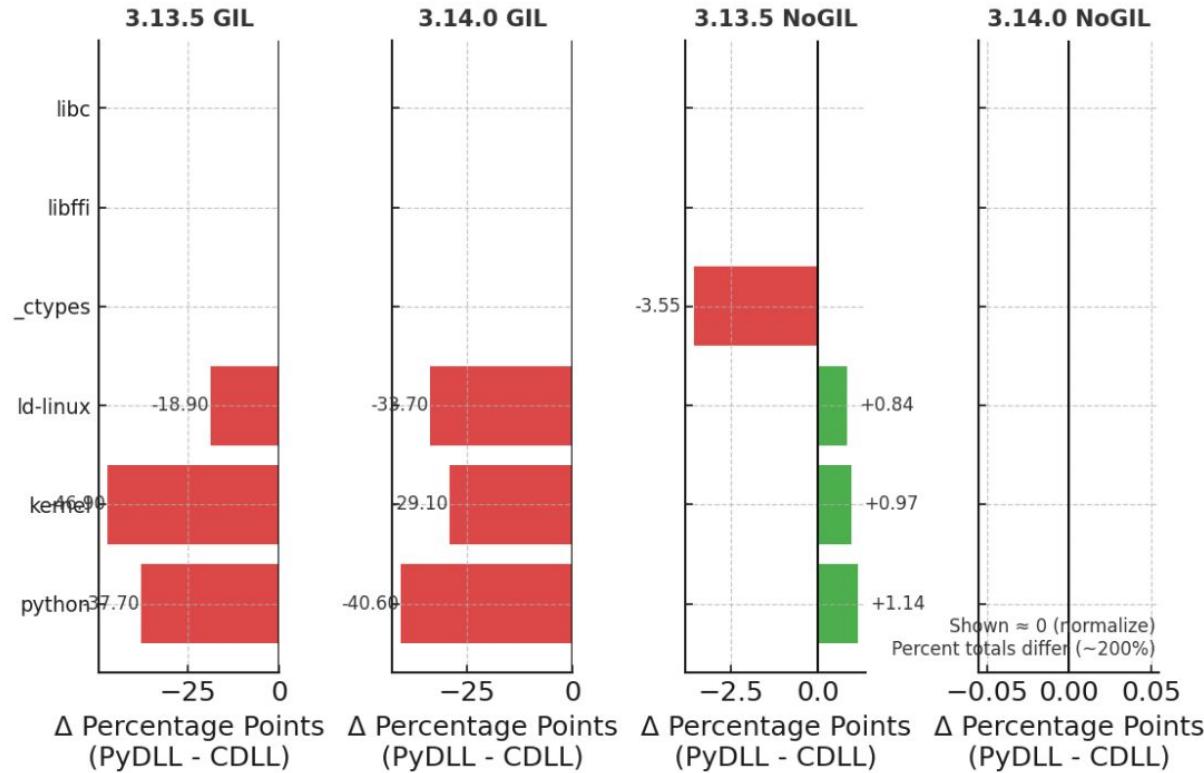
```
scc ~> /tmp
)) ~/git/pycon2025-ffi-hidden
CDLL  getpid x2000000: 0.643s
PyDLL  getpid x2000000: 0.526s
Speedup (PyDLL vs CDLL): 1.222x

scc ~> /tmp
)) ~/git/pycon2025-ffi-hidden
CDLL  getpid x2000000: 0.581s
PyDLL  getpid x2000000: 0.533s
Speedup (PyDLL vs CDLL): 1.089x

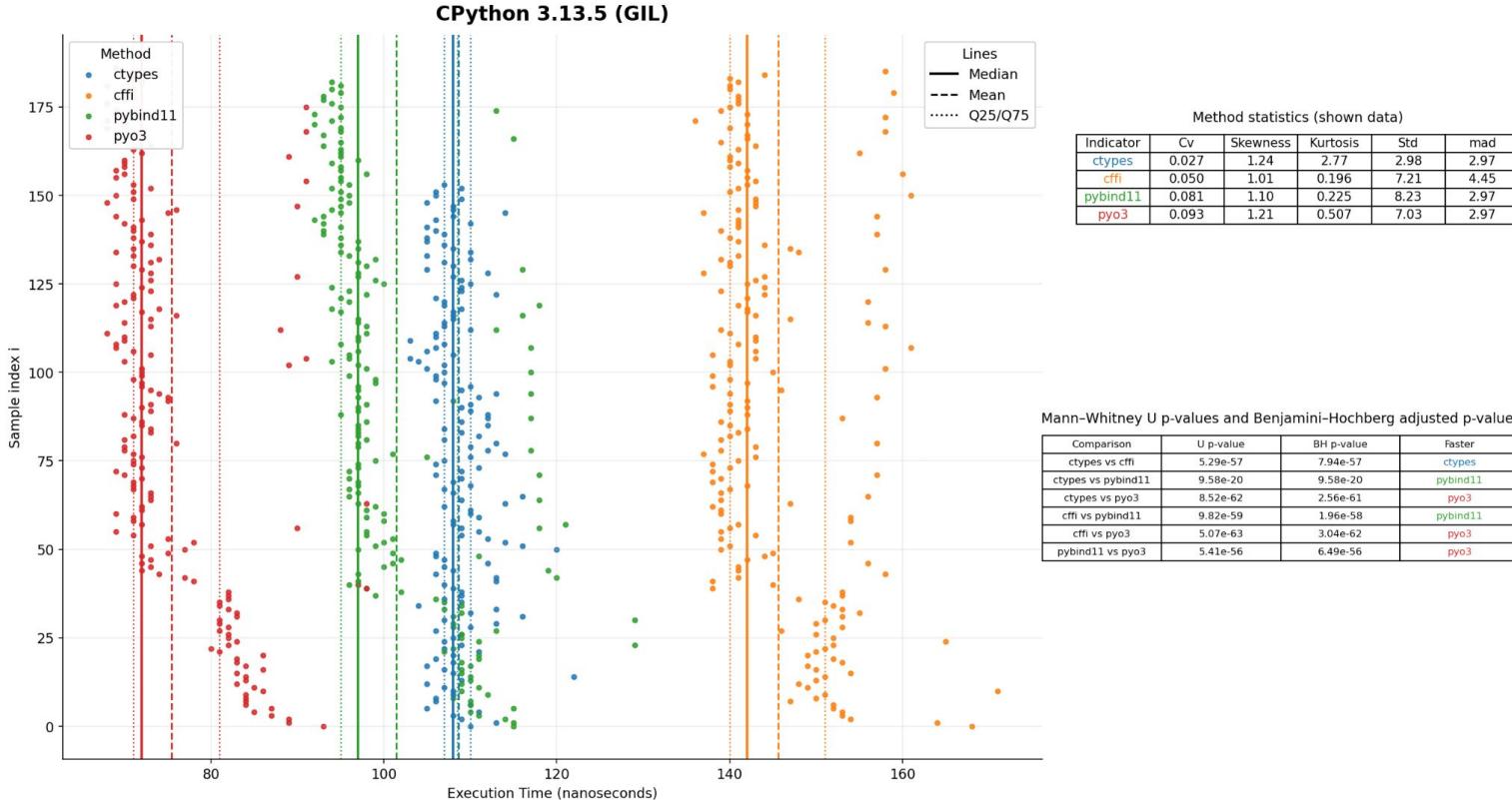
scc ~>
)) |
```



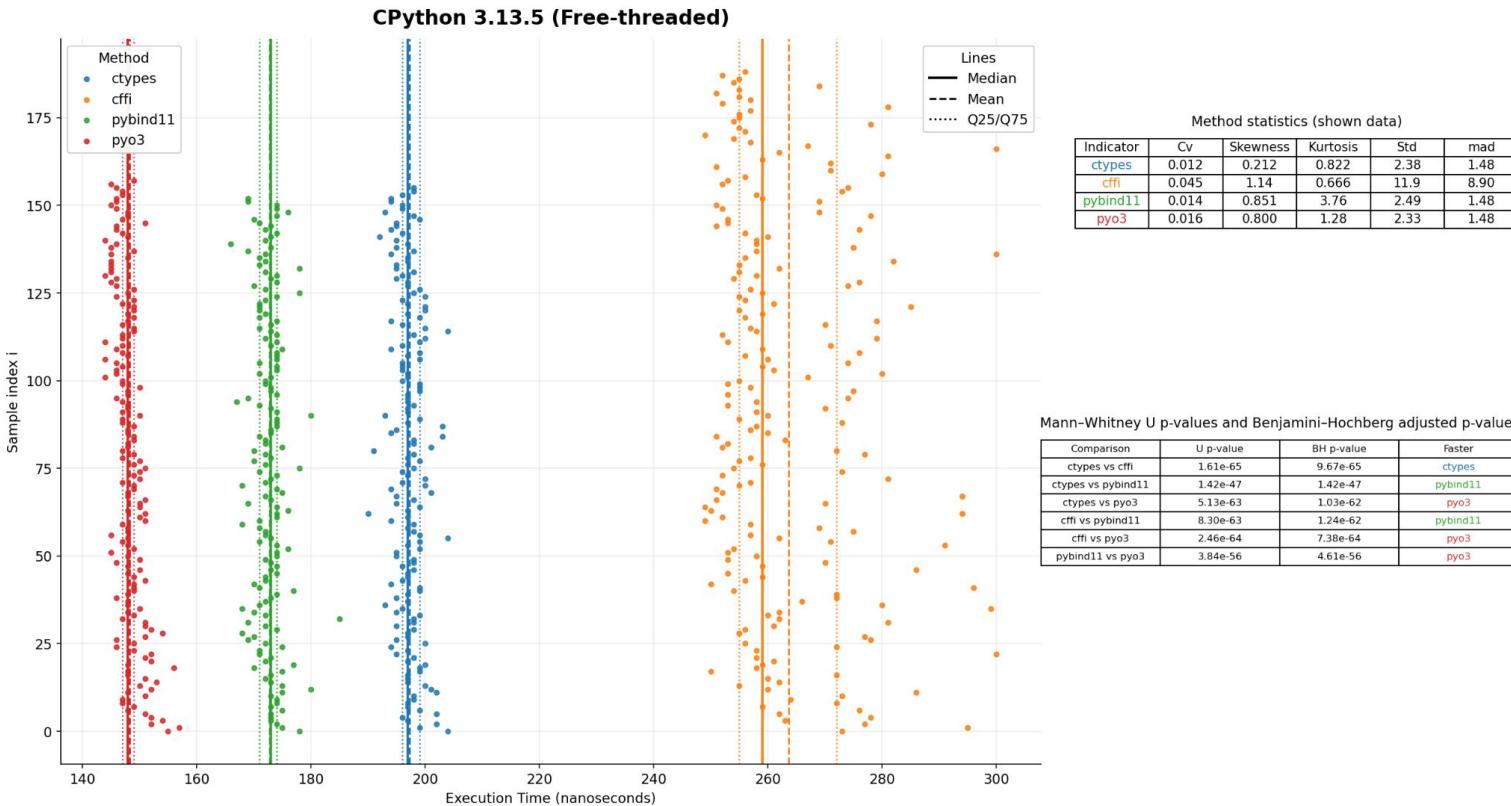
Why PyO3 fast, but ctypes slow?



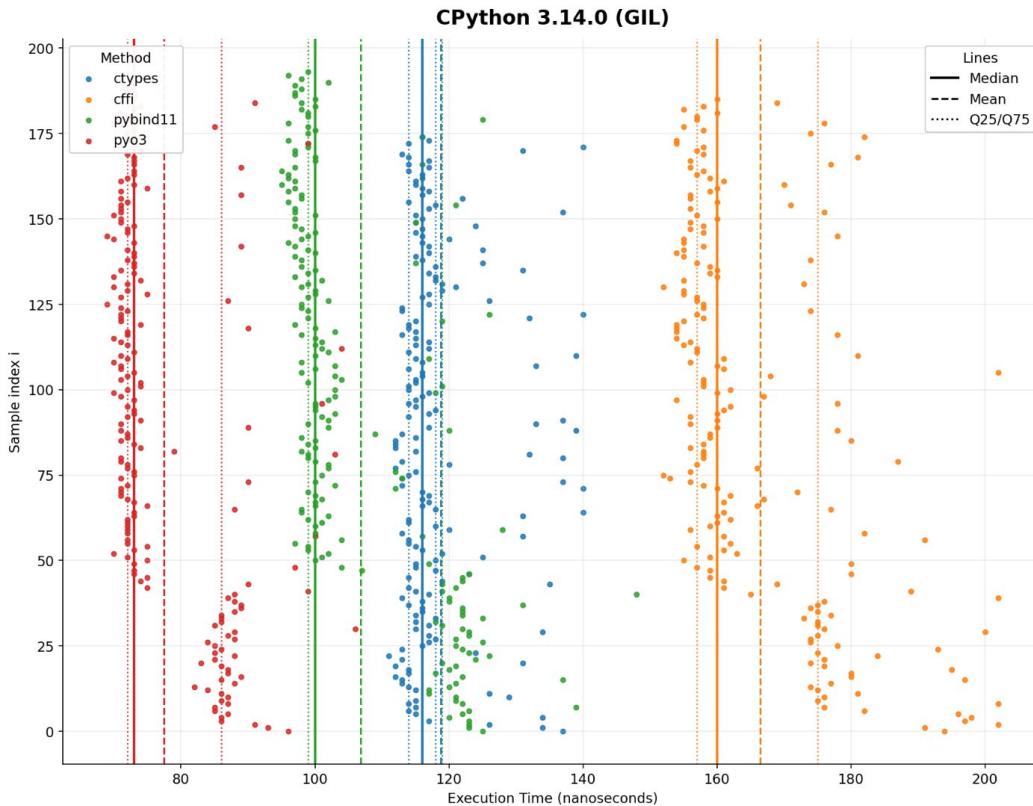
Why PyO3 fast, but ctypes slow?



Why PyO3 fast, but ctypes slow?



Why PyO3 fast, but ctypes slow?

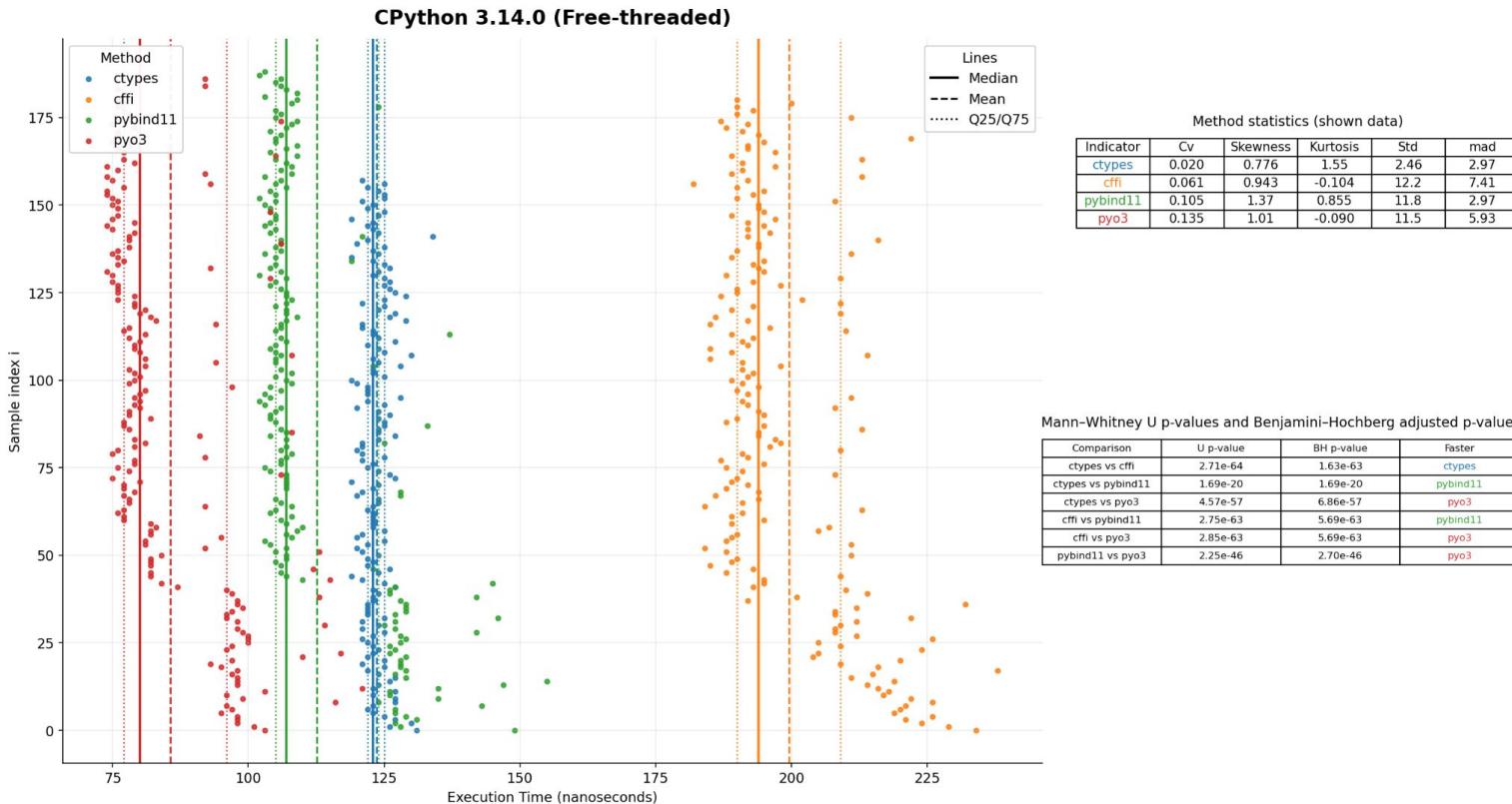


Method statistics (shown data)					
Indicator	Cv	Skewness	Kurtosis	Std	mad
ctypes	0.061	1.67	1.57	7.30	2.97
cffi	0.075	1.13	0.413	12.6	7.41
pybind11	0.104	0.963	-0.159	11.2	4.45
pyo3	0.110	1.23	0.537	8.53	2.97

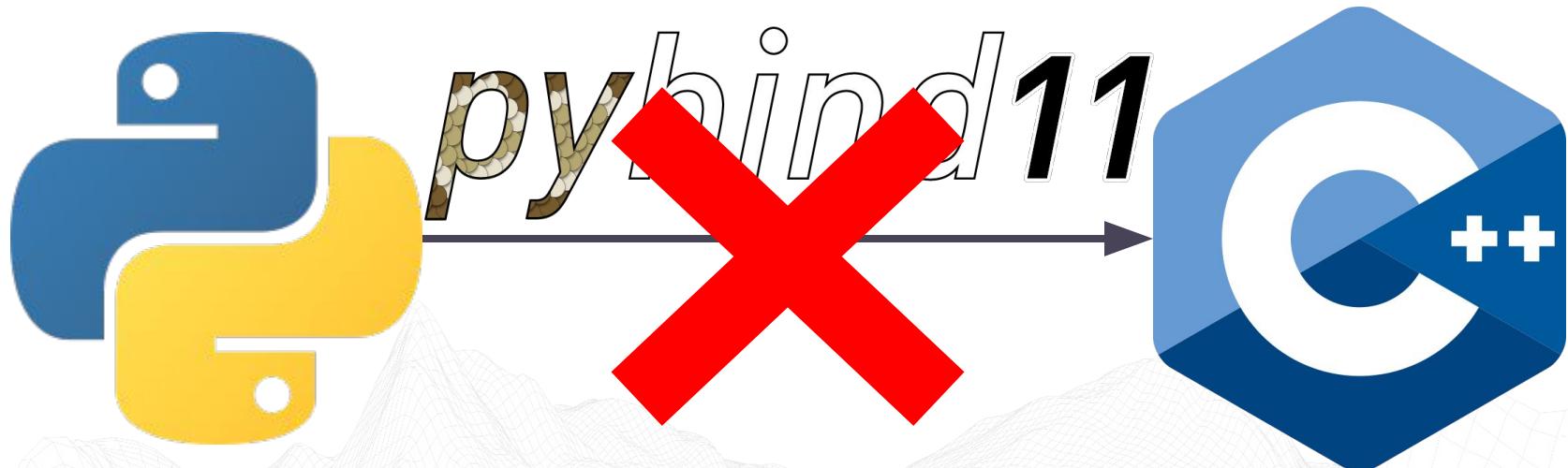
Mann-Whitney U p-values and Benjamini-Hochberg adjusted p-values

Comparison	U p-value	BH p-value	Faster
ctypes vs cffi	1.84e-65	1.10e-64	ctypes
ctypes vs pybind11	5.37e-21	5.37e-21	pybind11
ctypes vs pyo3	4.02e-61	6.02e-61	pyo3
cffi vs pybind11	3.42e-64	6.85e-64	pybind11
cffi vs pyo3	2.92e-64	6.85e-64	pyo3
pybind11 vs pyo3	3.73e-55	4.48e-55	pyo3

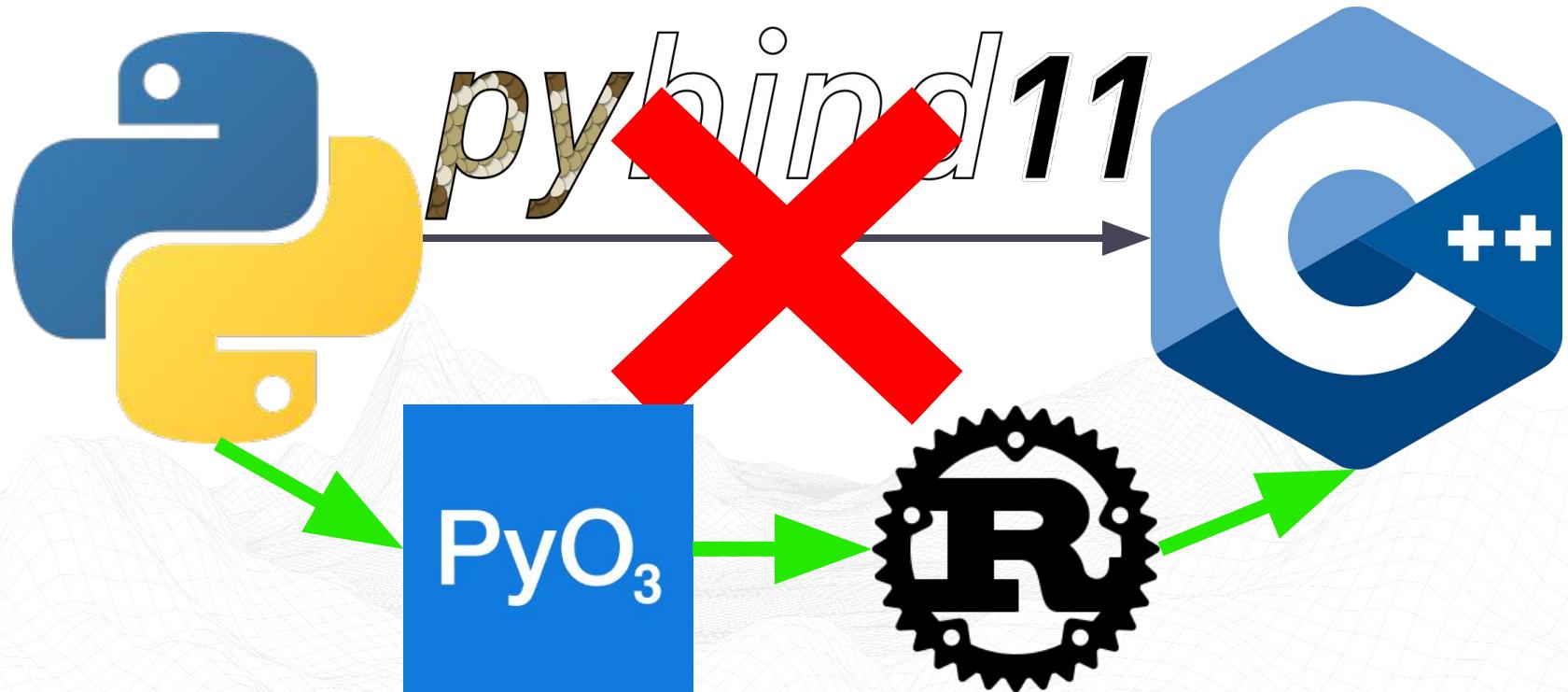
Why PyO3 fast, but ctypes slow?



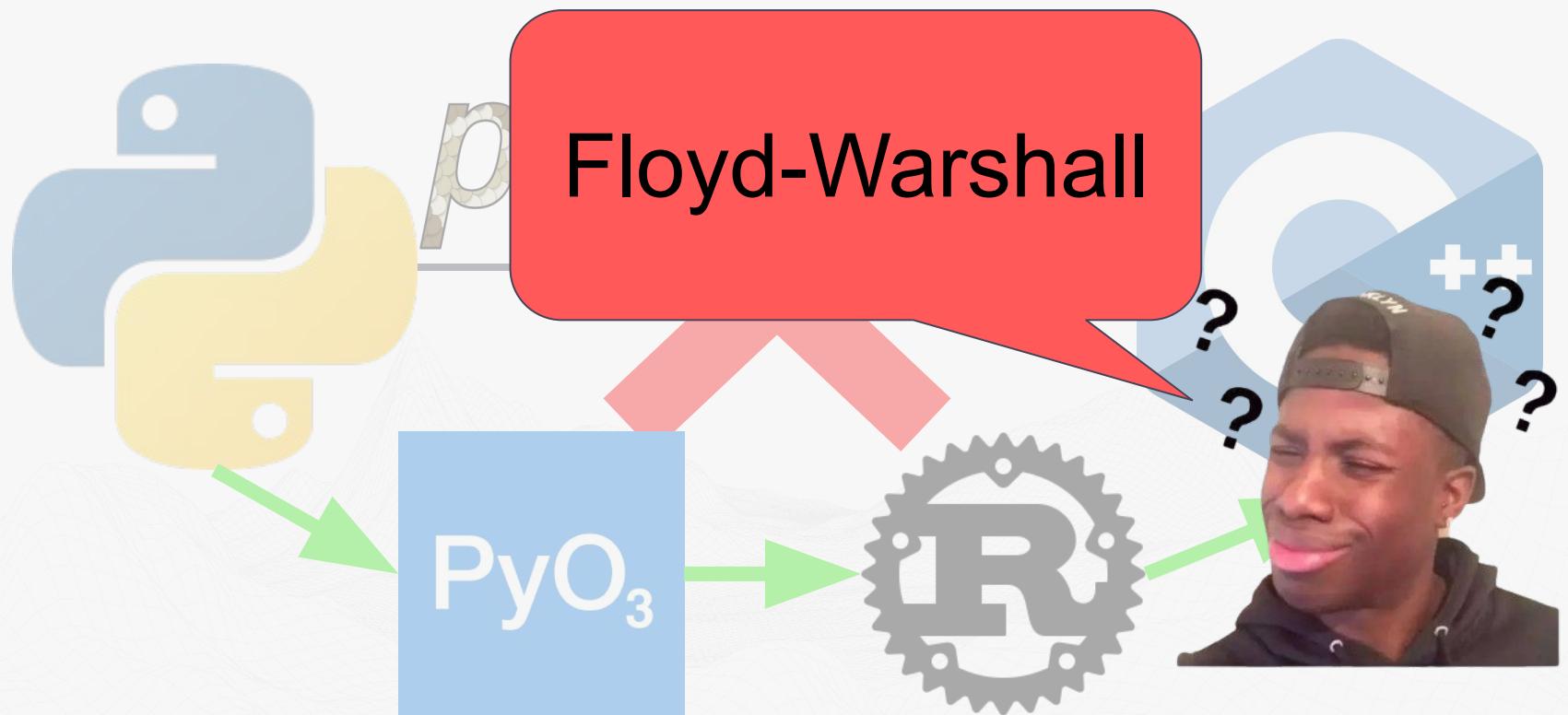
The Fastest Solution for C++ Binding



The Fastest Solution for C++ Binding



The Fastest Solution for C++ Binding



Not Release GIL, But Built with Free-Threaded

Loader	GIL Released?	Best for	Risk
CDLL	Yes	Long-running C code, multi-thread	Unsafe for Python C API calls
PyDLL	No	Short C calls, Python C API usage, single-thread	Blocks other threads, can deadlock

A = ctypes.**CDLL**(_lib_path)

B = ctypes.**PyDLL**(_lib_path)



Not Release GIL, But Built with Free-Threaded

```
scc ▾ ▷ /tmp
↳ ~/git/pycon2025-ffi-hidden-corner/cpython3.14.0rc1-gil/bin/python3 ./pydll_blocking.py
```

```
CASE: usleep holding call
Iterations during C sleep: 10,560,279
```

```
CASE: usleep holding call
Iterations during C sleep: 32,000
```

```
scc ▾ ▷ /tmp
↳ ~/git/pycon2025-ffi-hidden-corner/cpython3.14.0rc1-nogil/bin/python3 ./pydll_blocking.py
```

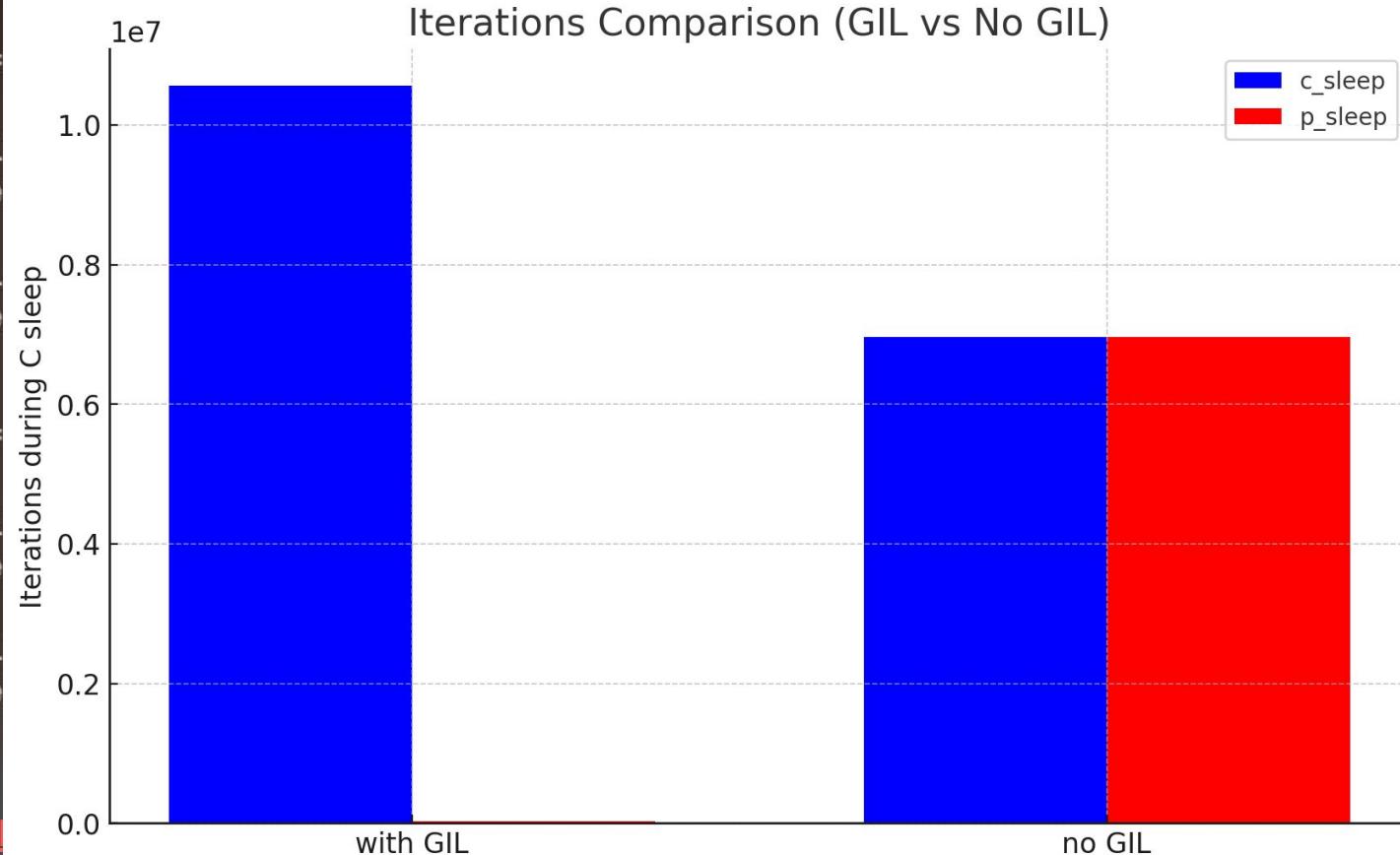
```
CASE: usleep holding call
Iterations during C sleep: 6,967,550
```

```
CASE: usleep holding call
Iterations during C sleep: 6,962,745
```

```
scc ▾ ▷ /tmp
↳ https://godbolt.org/z/dGxhsjhKT
```

Not Release GIL, But Built with Free-Threaded

```
scc A ▶ /tmp  
↳ ~/git/pycon2025-ff  
  
CASE: usleep holding call  
Iterations during C sleep  
  
CASE: usleep holding call  
Iterations during C sleep  
  
scc A ▶ /tmp  
↳ ~/git/pycon2025-ff  
  
CASE: usleep holding call  
Iterations during C sleep  
  
CASE: usleep holding call  
Iterations during C sleep  
  
scc A ▶ /tmp  
↳ https://godbolt.org/z/dGxhsjl
```



16.36% [.] _PyEval_EvalFrameDefault

```
--14.26%--_PyEval_EvalFrameDefault
|
|   |--12.04%--_PyEval_EvalCode
|   |
|   |   |--11.74%--run_eval_code_obj
|   |   |   run_mod
|   |   |   pyrun_file
|   |   |   _PyRun_SimpleFileObject
|   |   |   _PyRun_AnyFileObject
|   |   |   pymain_run_file_obj (inlined)
|   |   |   pymain_run_file (inlined)
|   |   |   pymain_run_python (inlined)
|   |   |   Py_RunMain
|   |   |   Py_BytesMain
|   |   |   0x7ffff7c27674
|   |   |   __libc_start_main
|   |   |   _start
|   |
|   |--0.63%--_PyObject_VectorcallTstate (inlined)
|
--0.66%--Py_INCREF (inlined)
```

6.99% [.] pthread_mutex_lock

```
--4.93%--take_gil
|   _PyThreadState_Attach
|   _call_function_pointer (inlined)
|   _ctypes_callproc (inlined)
|   PyCFuncPtr_call
|   _PyObject_MakeTpCall
|   _PyEval_EvalFrameDefault
|
|   pyeval_EvalCode
|   run_eval_code_obj
|   run_mod
|   pyrun_file
|   _PyRun_SimpleFileObject
|   _PyRun_AnyFileObject
|   pymain_run_file_obj (inlined)
|   pymain_run_file (inlined)
|   pymain_run_python (inlined)
|   Py_RunMain
|   Py_BytesMain
|   0x7ffff7c27674
|   __libc_start_main
|   _start
|
--1.86%--drop_gil_impl (inlined)
|   drop_gil (inlined)
|   _PyEval_ReleaseLock
|   PyEval_SaveThread
|
|   --1.84%--_call_function_pointer (inlined)
|       _ctypes_callproc (inlined)
|       PyCFuncPtr_call
|       _PyObject_MakeTpCall
|       _PyEval_EvalFrameDefault
```

Why PyO3 fast, but ctypes slow?

ctypes

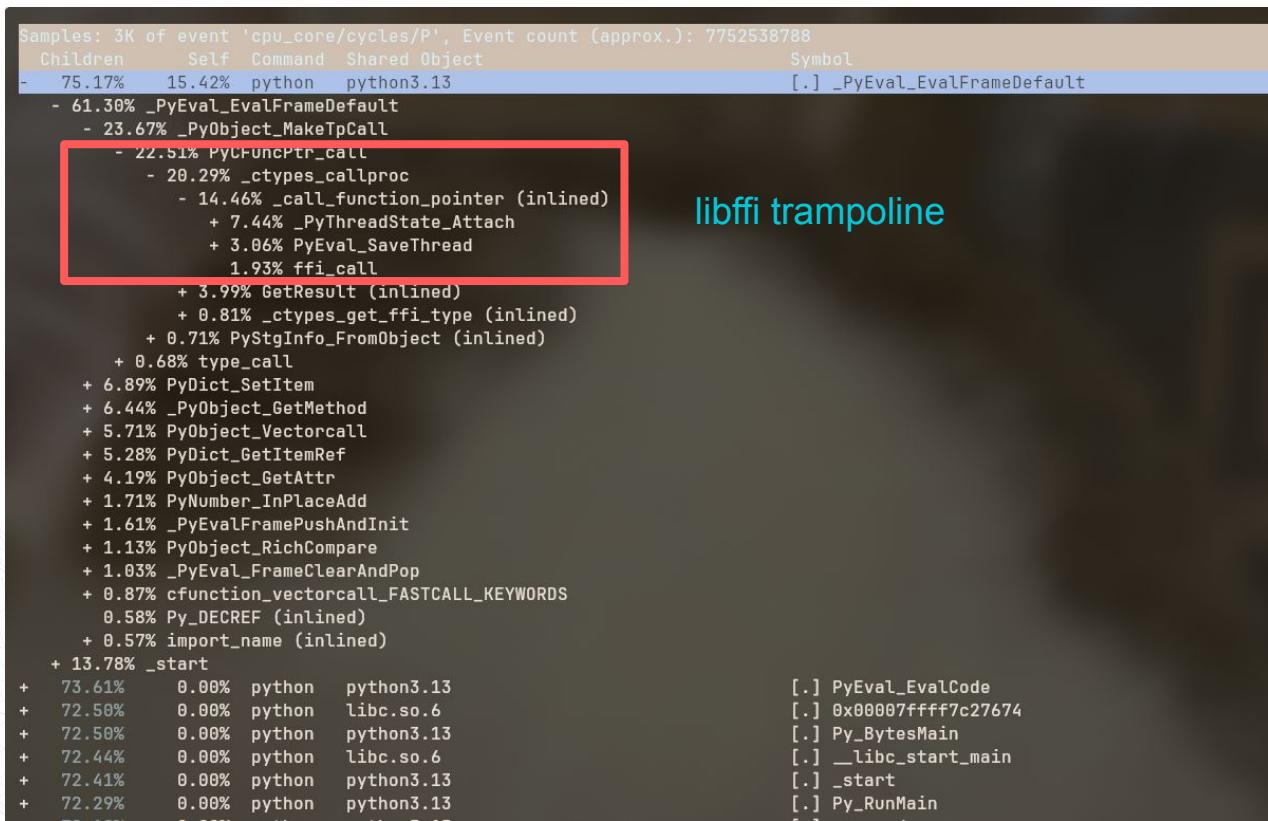
6.99% [.] pthread_mutex_lock

```
|   |--4.93%--take_gil
|   |   _PyThreadState_Attach
|   |   _call_function_pointer (inlined)
|   |   _ctypes_callproc (inlined)
|   |   PyCFuncPtr_call
|   |   _PyObject_MakeTpCall
|   |   _PyEval_EvalFrameDefault
```



perf

Why PyO3 fast, but ctypes slow?



Why PyO3 fast, but ctypes slow?

C function

Dynamic marshaling

pthread_mutex/futex

GIL

libffi trampoline

_ctypes_callproc

PyCFuncPtr_call

Rust function

Dynamic marshaling

pyo3::impl::trampoline
PyObject_Vector call

C function

Dynamic marshaling

_PyThreadState

libffi trampoline

_ctypes_callproc

PyCFuncPtr_call

Rust function

Dynamic marshaling

Py_dict_lookup_ threadsafe

pyo3::impl::trampoline

PyObject_Vector call

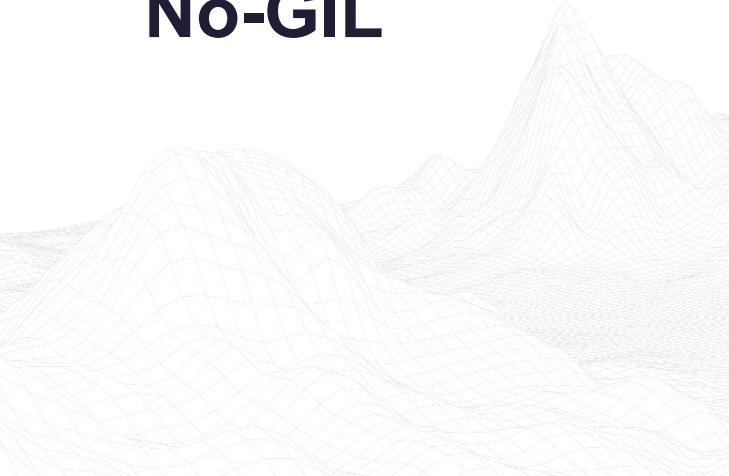
Recap

- PyDLL > CDLL over a 20% performance when NoGIL.
- PyO3 includes deep opt for vector calls without the GIL.
- The libffi trampoline emerges as the next bottleneck.

Racing FFI after No-GIL



Racing global states after No-GIL



Testing with GIL-enabled Python (Standard)

Running race condition tests...

Results with GIL:

Counter Test:

- Race conditions detected: 0 / 5
- Average accuracy: 100.0%

Fast Bank Test (no sleep - GIL protected):

- Race conditions detected: 0 / 5
- ✓ GIL is protecting against races in fast operations (as expected)

Testing with No-GIL Python (Experimental)

Running race condition tests...

Results without GIL:

Counter Test:

- Race conditions detected: 5 / 5
- Average accuracy: 63.7%
- Average lost increments: 181450

Fast Bank Test (no sleep - TRUE PARALLELISM):

- Race conditions detected: 2 / 5

▲ RACE CONDITIONS EXPOSED! (as expected without GIL)

Racing global states after No-GIL

```
lib = ctypes.PyDLL(str(lib_path))
threads = [threading.Thread(target=unsafe_worker) for _
in range(num_threads)]
```

Racing global states after No-GIL

```
lib = ctypes.PyDLL(str(lib_path))
threads = [threading.Thread(target=unsafe_worker) for _  
in range(num_threads)]
```

```
long unsafe_increment(int iterations) {  
    for (int _ = 0; _ < iterations; _++) {  
        long temp = global_counter; // Read - RACE condition!  
        COMPILER_BARRIER();  
        global_counter = temp + 1; // Write - RACE condition!  
        COMPILER_BARRIER();  
    }  
    return global_counter;  
}
```

Racing global states after No-GIL

```
lib = ctypes.PyDLL(str(lib path))
threads = [threading.Thread(target=unsafe worker) for _
in range(num_threads)]
```

It's safe when PyDLL + GIL.

```
long unsafe_increment(int it
    for (int _ = 0; _ < iterations; _++) {
        long temp = global_counter; // Read - RACE condition!
        COMPILER_BARRIER();
        global_counter = temp + 1; // Write - RACE condition!
        COMPILER_BARRIER();
    }
    return global_counter;
}
```

Racing global states after No-GIL

```
lib = ctypes.PyDLL(str(lib_path))

# Counter Increment Race Condition Demo
# Threads: 50
# Iterations per thread: 10,000
# Expected final value: 500,000

# Testing UNSAFE increment...
# Testing SAFE increment (mutex)...
# Testing ATOMIC increment...

# Counter Increment Comparison
# Method | Final Value | Expected | Lost Increments | Accuracy | Status
# UNSAFE | 500,000 | 500,000 | 0 | 100.0% | ✓ OK
# SAFE (mutex) | 500,000 | 500,000 | 0 | 100.0% | ✓ OK
# ATOMIC | 500,000 | 500,000 | 0 | 100.0% | ✓ OK

global_counter = temp + 1;
COMPILER_BARRIER();
}

return global_counter;
```

```
== Bank Withdrawal Race Condition Demo ==
Initial balance: $1000
Threads: 100
Withdrawal amount: $50
Attempts per thread: 3
Max possible withdrawals: 300 = $15000

Testing UNSAFE withdrawal (with race condition) ...
Testing SAFE withdrawal (thread-safe) ...

Race Condition Comparison
Method | Successful Withdrawals | Total Withdrawn | Final Balance | Balance + Withdrawn | Consistency | Time (s)
UNSAFE | 20 | $1000 | $0 | $1000 | ✓ OK | 0.010
SAFE | 20 | $1000 | $0 | $1000 | ✓ OK | 0.005

== Fast Bank Withdrawal Race (No Sleep) ==
Iterations: 5
Race conditions detected: 0/5
Average withdrawals: 50.0

✓ No race conditions detected (GIL protected)
```

```
uv_run --python
    ../../cpython3.13.5-gil/bin/python3 python
    demo_race_condition.py
```

Racing global states after No-GIL

```
lib = ctypes.PyDLL(str(lib_path))
== Counter Increment Race Condition Demo ==
Threads: 50
Iterations per thread: 10,000
Expected final value: 500,000

Testing UNSAFE increment...
Testing SAFE increment (mutex) ...
Testing ATOMIC increment...

1. Counter Increment Comparison
+-----+
| Method | Final Value | Expected | Lost Increments | Accuracy | Status |
+-----+
| UNSAFE | 400,766 | 500,000 | 99,234 | 80.2% | ✗ RACE |
| SAFE (mutex) | 500,000 | 500,000 | 0 | 100.0% | ✓ OK |
| ATOMIC | 500,000 | 500,000 | 0 | 100.0% | ✓ OK |
+-----+
⚠ RACE CONDITION: Lost 99,234 increments (19.8% data loss)!

global_counter = temp + 1;
COMPILE_BARRIER();
}

return global_counter;
```

```
= Bank Withdrawal Race Condition Demo =
Initial balance: $1000
Threads: 100
Withdrawal amount: $50
Attempts per thread: 3
Max possible withdrawals: 300 = $15000

Testing UNSAFE withdrawal (with race condition)...
Testing SAFE withdrawal (thread-safe) ...

2. Race Condition Comparison
+-----+
| Method | Successful Withdrawals | Total Withdrawn | Final Balance | Balance + Withdrawn | Consistency | Time (s) |
+-----+
| UNSAFE | 300 | $15000 | $300 | $15300 | ✗ BROKEN | 0.021 |
| SAFE | 20 | $1000 | $0 | $1000 | ✓ OK | 0.006 |
+-----+
⚠ RACE CONDITION DETECTED: Total is off by $14300!
```

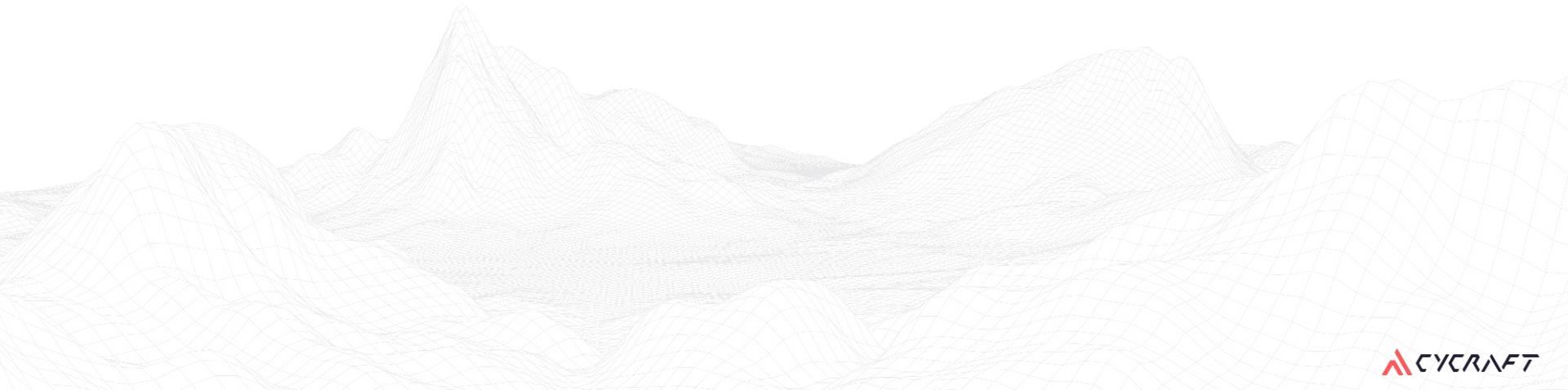
```
uv run --python
    ../../cpython3.13.5-nogil/bin/python3 python
    demo_race_condition.py
```

Racing global states after No-GIL

	GIL	No-GIL
Thread Safe FF	Safe	Safe
Non Thread Safe FF	Uncertain	Race!

Recap

- Be aware that the GIL protects unsafe global states.
- Not all FFI calls are protected by the GIL.
- Ensure your FFI code is thread-safe.



Higher Risk for Arena Leakage in GLIBC



Scenario	glibc malloc (ptmalloc2+tcache)	macOS libmalloc (zones+magazines+nano)	jemalloc	tcmalloc	Windows HeapAlloc (NT/Segment+LFH)
many short-lived threads	△ (cap <code>arena_max</code> , pool threads)	×	★ (many arenas, decay purge)	○ (per-thread caches)	×
Long-lived server, RSS-sensitive	△ (trim/mmap tuning)	×	★ (decay + background purge)	△ (raise release rate)	×
Latency-critical, multi-threaded	○ (tcache helps)	×	○	★ (lock-free fast paths)	△ (enable LFH/Segment Heap)
Cloud functions / short-lived processes	★ (simple, low setup)	×	○ (defaults fine)	○	×
Tight memory limit (containers, 256–512 MiB)	○ (tune thresholds)	×	★ (predictable footprint)	△ (cap thread caches)	×
Backgrounding app	×	★ (UI latency; pressure-relief)	△ (uncommon on macOS)	△	×
High thread-count backend	×	×	○ (if linked)	○ (if linked)	△ (enable LFH; consider Segment Heap)
Fork/exec heavy (prefork workers)	○	×	★ (copy-on-write friendly decay)	○	×
NUMA / many-core (>32 cores)	△ (no strong NUMA policy)	×	★ (arena sharding, pinning options)	○ (per-CPU caches)	△
Production leak/frag profiling needed	△ (basic)	○ (Instruments/stack logging)	★ (heap profiling, mallctl stats)	○ (heap profiler)	○ (ETW!heap/PageHeap)



Use GLIBC_TUNABLES when it is matter



Default tunable malloc

M_MMAP_THRESHOLD

For allocations greater than or equal to the limit specified (in bytes) by `M_MMAP_THRESHOLD` that can't be satisfied from the free list, the memory-allocation functions employ `mmap(2)` instead of increasing the program break using `sbrk(2)`.

Allocating memory using `mmap(2)` has the significant advantage that the allocated memory blocks can always be independently released back to the system. (By contrast, the heap can be trimmed only if memory is freed at the top end.) On the other hand, there are some disadvantages to the use of `mmap(2)`: deallocated space is not placed on the free list for reuse by later allocations; memory may be wasted because `mmap(2)` allocations must be page-aligned; and the kernel must perform the expensive task of zeroing out memory allocated via `mmap(2)`. Balancing these factors leads to a default setting of $128*1024$ for the `M_MMAP_THRESHOLD` parameter.

The lower limit for this parameter is `0`. The upper limit is `DEFAULT_MMAP_THRESHOLD_MAX`: $512*1024$ on 32-bit systems or `4*1024*1024*sizeof(long)` on 64-bit systems.

Note: Nowadays, glibc uses a dynamic `mmap threshold` by default. The initial value of the `threshold` is $128*1024$, but when blocks larger than the current `threshold` and less than or equal to `DEFAULT_MMAP_THRESHOLD_MAX` are freed, the `threshold` is adjusted upward to the size of the freed block. When dynamic `mmap thresholding` is in effect, the `threshold` for trimming the heap is also dynamically adjusted to be twice the dynamic `mmap threshold`. Dynamic adjustment of the `mmap threshold` is disabled if any of the `M_TRIM_THRESHOLD`, `M_TOP_PAD`, `M_MMAP_THRESHOLD`, or `M_MMAP_MAX` parameters is set.

M_TRIM_THRESHOLD

When the amount of contiguous free memory at the top of the heap grows sufficiently large, `free(3)` employs `sbrk(2)` to release this memory back to the system. (This can be useful in programs that continue to execute for a long period after freeing a significant amount of memory.) The `M_TRIM_THRESHOLD` parameter specifies the minimum size (in bytes) that this block of memory must reach before `sbrk(2)` is used to trim the heap.

The default value for this parameter is $128*1024$. Setting `M_TRIM_THRESHOLD` to `-1` disables trimming completely.

Modifying `M_TRIM_THRESHOLD` is a trade-off between increasing the number of system calls (when the parameter is set low) and wasting unused memory at the top of the heap (when the parameter is set high).

```
scc ▾ ➔ ~
)) /lib64/ld-linux-x86-64.so.2 --list-tunables | grep malloc
glibc.malloc.arena_max: 0x0 (min: 0x1, max: 0xffffffffffffffff)
glibc.malloc.arena_test: 0x0 (min: 0x1, max: 0xffffffffffffffff)
glibc.malloc.check: 0 (min: 0, max: 3)
glibc.malloc.hugetlb: 0x0 (min: 0x0, max: 0xffffffffffffffff)
glibc.malloc.mmap_max: 0 (min: 0, max: 2147483647)
glibc.malloc.mmap_threshold: 0x0 (min: 0x0, max: 0xffffffffffffffff)
glibc.malloc.mxfast: 0x0 (min: 0x0, max: 0xffffffffffffffff)
glibc.malloc.perturb: 0 (min: 0, max: 255)
glibc.malloc.tcache_count: 0x0 (min: 0x0, max: 0xffffffffffffffff)
glibc.malloc.tcache_max: 0x0 (min: 0x0, max: 0xffffffffffffffff)
glibc.malloc.tcache_unsorted_limit: 0x0 (min: 0x0, max: 0xffffffff)
glibc.malloc.top_pad: 0x20000 (min: 0x0, max: 0xffffffffffffffff)
glibc.malloc.trim_threshold: 0x0 (min: 0x0, max: 0xffffffffffffffff)
```

Suppose

```
def python_worker(worker_id):
    worker_memory_deltas = []
    for i in range(ITERATIONS_PER_THREAD):
        # Each call spawns RUST_THREADS_PER_CALL Rust threads
        initial, final = glibc_arena_poc.run_arena_test(RUST_THREADS_PER_CALL)
        worker_memory_deltas.append(final - initial)
        time.sleep(0.005) # Small delay

    return {
        "worker_id": worker_id,
        "total_memory_delta": sum(worker_memory_deltas),
        "memory_deltas": worker_memory_deltas
    }

initial_rss = glibc_arena_poc.get_rss_mib()
start_time = time.time()

worker_results = []
with concurrent.futures.ThreadPoolExecutor(max_workers=PYTHON_THREADS) as executor:
    futures = [executor.submit(python_worker, i) for i in range(PYTHON_THREADS)]

    for future in concurrent.futures.as_completed(futures):
        try:
            result = future.result()
            worker_results.append(result)
        except Exception as e:
            pass # Continue on errors

end_time = time.time()
final_rss = glibc_arena_poc.get_rss_mib()
```

Suppose

```
def python_worker(worker_id):
    worker_memory_deltas = []
    for i in range(ITERATIONS_PER_THREAD):
        # Each call spawns RUST_THREADS_PER_CALL Rust threads
        initial, final = glibcarena_poc.get_rss_mib(RUST_THREADS_PER_CALL)
        worker_memory_deltas.append(final - initial)
        time.sleep(0.005) # Small delay

    return {
        "worker_id": worker_id,
        "memory_deltas": worker_memory_deltas,
        "rss_mib": final_rss_mib()
    }

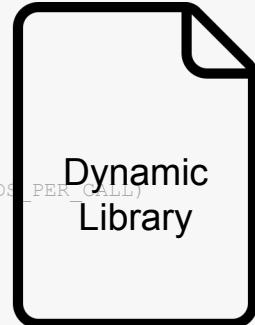
initial_start_time = time.time()
start_rss = glibcarena_poc.get_rss_mib()

for worker_id in range(WORKERS):
    with concurrent.futures.ThreadPoolExecutor(max_workers=PYTHON_THREADS) as executor:
        futures = [executor.submit(python_worker, i) for i in range(PYTHON_THREADS)]
        for future in concurrent.futures.as_completed(futures):
            result = future.result()
            rss = result["rss_mib"]

            if result["worker_id"] == worker_id:
                print(f"Worker {worker_id} RSS: {rss} MiB")

end_time = time.time()
final_rss = glibcarena_poc.get_rss_mib()
```

Code Jump



Rust Code

Rust Function

Python Code

Python Function

Suppose

```
def python_worker(worker_id):
    worker_memory_deltas = []
    for i in range(ITERATIONS_PER_THREAD):
        # Each call spawns RUST_THREADS_PER_CALL Rust threads
        initial, final = glibc_arena_poc.get_rss_mib(RUST_THREADS_PER_CALL)
        worker_memory_deltas.append(final - initial)
        time.sleep(0.005)  Small delay

    return {
        "worker_id": worker_id,
        "memory_deltas": worker_memory_deltas,
        "rss_mib": final_rss_mib()
    }

if __name__ == "__main__":
    init_start_time = time.time()
    start_time = time.time()

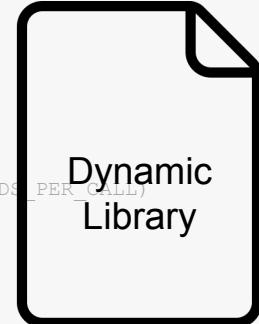
    workers = []
    for i in range(PYTHON_THREADS):
        workers.append(Thread(target=python_worker, args=(i,)))

    executor(max_workers=PYTHON_THREADS) as executor:
        futures = [executor.submit(python_worker, i) for i in range(PYTHON_THREADS)]
        for future in futures:
            result = future.result()
            print(result)

    end_time = time.time()
    final_rss = glibc_arena_poc.get_rss_mib()
```

M Python threads
Python Code

Code Jump



N Rust threads



Rust Function

```
scc A ▶ ~ /git/pycon2025-ffi-hidden-corner/tests/arena/rust
  p (main) +3 🐍?1 3.13.5
└─ » python3 arena_test.py benchmark --arena-risk --python-threads 8 --rust-threads 50
  🔥 Running Arena Risk Analysis...
    Python Threads (N): 8
    Rust Threads per call (M): 50
    Expected complexity: O(N*M) = O(400)
[2025-08-15 21:55:46] [INFO] 🔮 Comparing Arena Risk - Python 3.13.5: GIL vs no-GIL
[2025-08-15 21:55:46] [INFO] ✅ Arena risk test completed for 3.13.5-gil: 0.56 MiB delta, 0.0x parallelism
[2025-08-15 21:55:46] [INFO] ✅ Arena risk test completed for 3.13.5-nogil: 16.57 MiB delta, 0.0x parallelism
[2025-08-15 21:55:46] [INFO]     GIL memory delta: 0.56 MiB
[2025-08-15 21:55:46] [INFO]     no-GIL memory delta: 16.57 MiB
[2025-08-15 21:55:46] [INFO]     Risk amplification: 29.46x
[2025-08-15 21:55:46] [INFO] 🔮 Comparing Arena Risk - Python 3.14.0rc1: GIL vs no-GIL
[2025-08-15 21:55:46] [INFO] ✅ Arena risk test completed for 3.14.0rc1-gil: 0.59 MiB delta, 0.0x parallelism
[2025-08-15 21:55:46] [INFO] ✅ Arena risk test completed for 3.14.0rc1-nogil: 32.60 MiB delta, 0.0x parallelism
[2025-08-15 21:55:46] [INFO]     GIL memory delta: 0.59 MiB
[2025-08-15 21:55:46] [INFO]     no-GIL memory delta: 32.60 MiB
[2025-08-15 21:55:46] [INFO]     Risk amplification: 55.26x
```

🔴 Arena Leakage Risk Analysis Results:

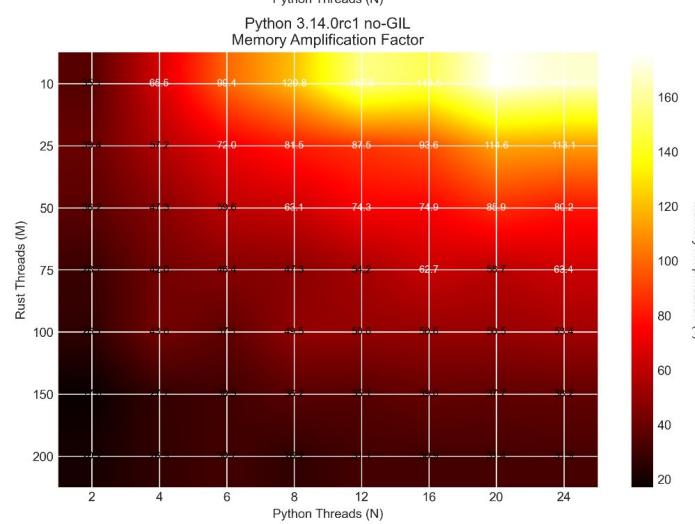
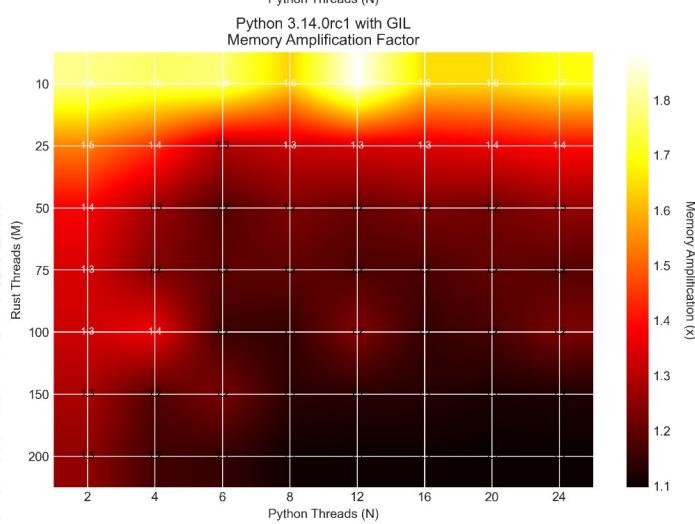
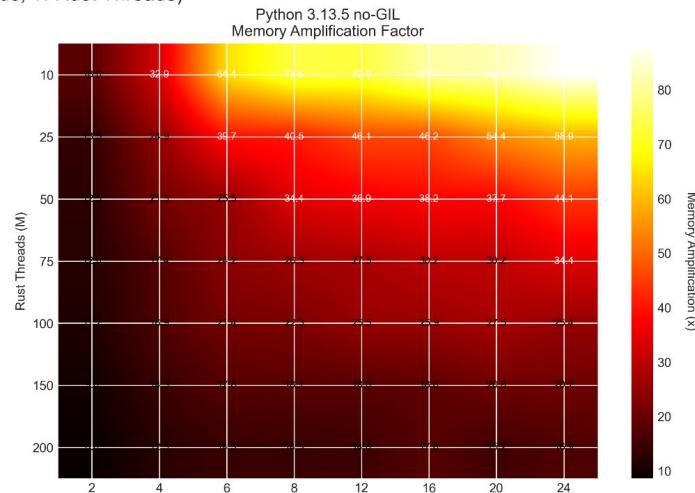
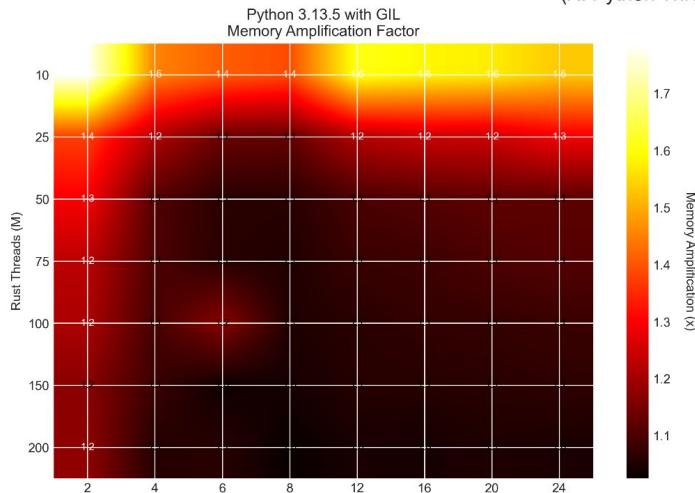
Python 3.13.5 - Arena Risk Assessment:

🔒 GIL Memory Delta:	0.56 MiB (amplification: 1.09x)
🔓 no-GIL Memory Delta:	16.57 MiB (amplification: 32.38x)
⚠️ Risk Amplification:	29.46x
🌐 Risk Level:	HIGH
🔄 Parallelism Increase:	0.00x
⚠️ HIGH RISK: no-GIL shows significant arena leakage amplification!	

Python 3.14.0rc1 - Arena Risk Assessment:

🔒 GIL Memory Delta:	0.59 MiB (amplification: 1.18x)
🔓 no-GIL Memory Delta:	32.60 MiB (amplification: 62.28x)
⚠️ Risk Amplification:	55.26x
🌐 Risk Level:	HIGH
🔄 Parallelism Increase:	0.00x
⚠️ HIGH RISK: no-GIL shows significant arena leakage amplification!	

Arena Leakage Risk Matrix: GIL vs no-GIL
(X: Python Threads, Y: Rust Threads)





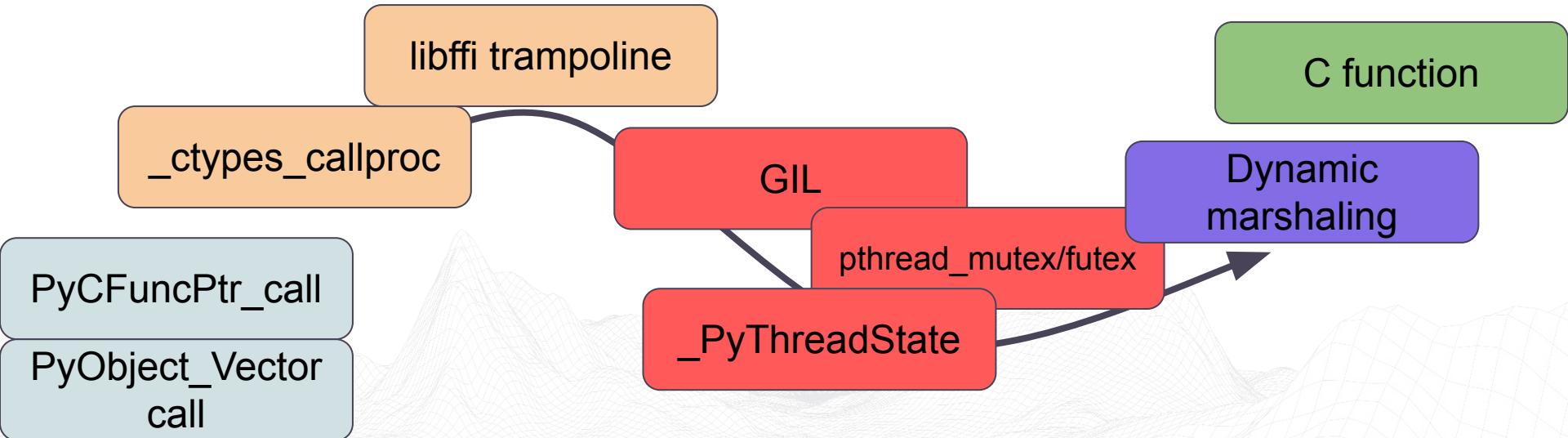
GDB!



Best Practice

- Pin the thread
- Aware multi-threading bugs
- Thread safe for each component

The hidden corner you might be aware



The hidden corner you might be aware

- Application level:
 - **GIL Semantics**, **Thread Pinning**, Refcount Edges & Ownership, Error Propagation, Reentrancy...
- OS level:
 - **Race Conditions**, **Dynamic Loading** / lazy binding, **System Libraries Features**, Loader Search Path, Permissions...
- Hardware level:
 - **ABI Compatibility**, **Cache Locality**, Modern CPU Affinity Binding, TLB, Branch Prediction, I/O Topology...

Takeaway

- Python 3.14t is Impressive
- Canary or Shadow Deployment
- Make Sure All Components are Thread-Safe

Thank you!

scc@scc.tw



Q&A

Empower cybersecurity with innovative AI technology



py-free-threading.github.io

Free-Threaded Wheels

ft-checker.com

