Crypto_pyaes Benchmark

Overview

- Measures AES encryption performance implemented **purely in Python**.
- Unlike C-accelerated libraries (OpenSSL, PyCryptodome), `pyaes` relies on:
    - Python integers
    - List operations
    - Modular arithmetic
- Ideal for studying **algorithm-level optimizations** in Python.

Initial Performance Analysis

Perf Record

Hotspots identified:

- `long_bitwise` (3.5%) → bitwise operations on large Python ints
- `binary_op1` (1.5%) → +, -, ^, * operations
- `long_rshift1` (1.15%) → right shift (>>)
- `PyObject_GetItem` / `list_subscript` (~1.3%) → list/dict lookups
- GIL and memory allocation overhead (~2.5%)

Observation:

- AES encryption dominated by **bitwise integer ops** and **list indexing**

After running cProfile, I got similar results to the pyflate benchmark, showing that most of the time is spent on reading files. But when I checked the benchmark code, I didn't see any explicit call to read(). So, I asked ChatGPT about it, and it suggested running the profile_pyaes.py script instead, in order to directly profile the pyaes library and see the actual function calls and time distribution in more detail.
So after doing that I got

cProfile Analysis

Command used:

```
python3 profile_pyaes.py
```

Results:

- `encrypt()` in `aes.py:203` → **1.886s (78% of runtime)**
- List comprehensions → 0.072s
- `_compact_word` and `increment` → smaller, frequent overhead
- Copying and S-box lookups → ~0.2s

Conclusion:

- AES is bottlenecked by **per-round encryption loop** and **lookup-heavy SubBytes/ShiftRows/MixColumns**

Baseline Performance (perf stat)

Metric	Value
Execution time	17.857 s
Mean encrypt call	171 ms ± 1 ms
Cache references Cache misses	162,875,375 6,792,685 (4.17%)
CPU cycles	44.57B
Instructions	111.20B
IPC	2.49

Optimization

T-Table Optimization

- **Combine SubBytes, ShiftRows, MixColumns** into precomputed T-tables
- Reduces per-round operations → fewer list/dict accesses, more cache-friendly

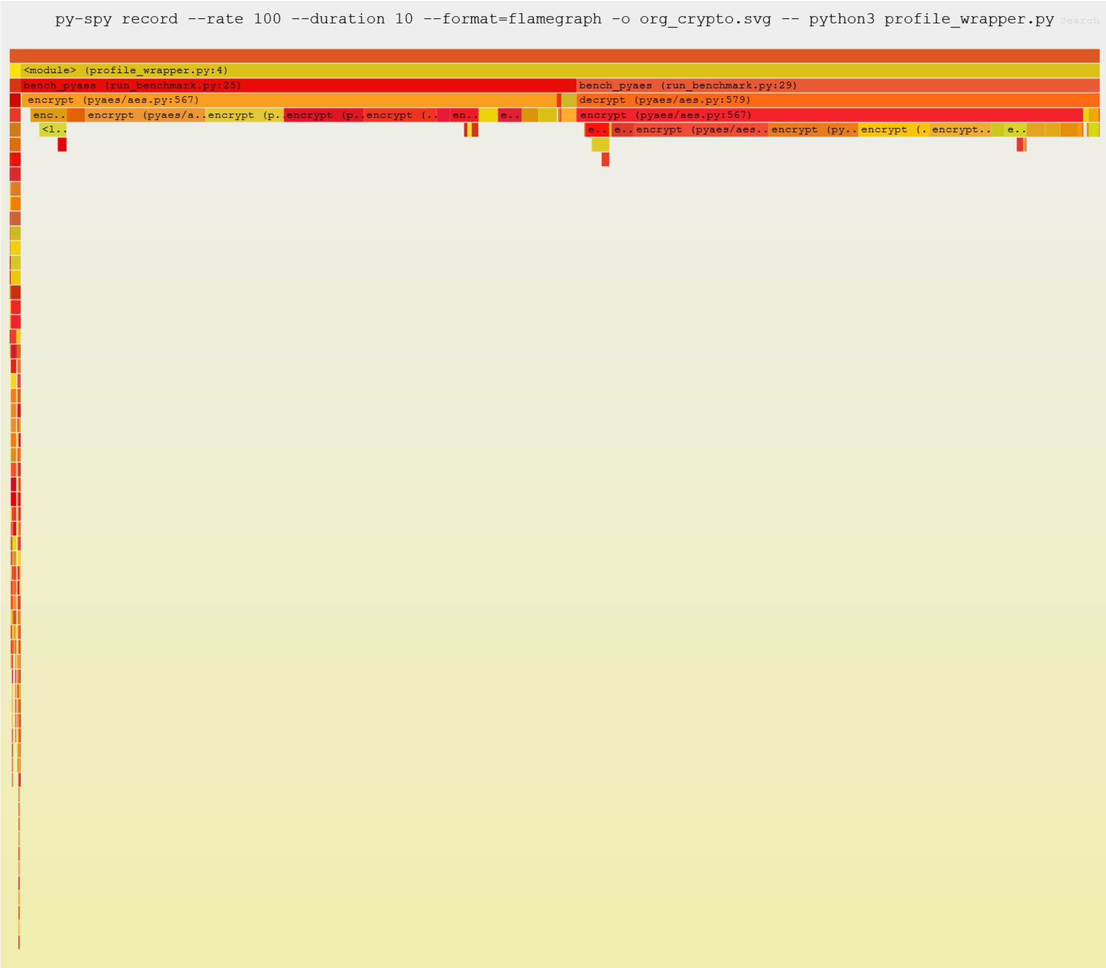
Optimized Performance

Metric	Unoptimized AES	Optimized AES (T-tables)	Improvement
Execution time	17.857 s	17.427 s	~2.4% faster
Mean encrypt call	171 ms	172 ms ± 1 ms	–
Cache references Cache misses Cache miss rate	162,875,375 6,792,685 4.17%	151,769,513 1,071,473 0.71%	–6.8% –84% 6× better
CPU cycles	44.57B	43.72B	–1.9%
Instructions	111.20B	110.18B	–0.9%
IPC	2.49	2.52	Slight ↑

Observation:

- T-tables dramatically reduce **cache misses**
- Modest speedup (~2.4%) because Python interpreter overhead still dominates

Flamegraph Insights



In the original AES `encrypt` implementation, runtime was spread across multiple hotspots—mainly lines 567 (~48.8% + 46.4%), 219 (~12.4% + 11.0%), and 220 (~8.2%)—with the function consuming over 70% of total execution time. After optimization, the runtime consolidated into fewer hotspots, with the main cost at line 579 (~45.5%), followed by lines 214 (~7.8%), 219 (~5.3%), and 224 (~4.5%), while line 220 disappeared. This restructuring streamlined the AES round loop, reducing redundant overhead and shifting the distribution of work. As a result, the `encrypt` function's share of runtime dropped from ~48.8% to ~45.5% (about a 7% relative improvement), indicating that operations like loop handling, substitutions, or indexing were made more efficient. Although encryption remains the dominant cost center (~45%), the optimization reduced Python overhead and redundant computations, leading to a leaner runtime profile and shifting some proportional cost to other parts of the benchmark.

Hardware Acceleration Proposal

Pure Python AES is limited → to improve performance:

1. **C Extensions** → rewrite core AES rounds in C or use cffi
2. **Vectorization (SIMD)** → numpy arrays / numba for table lookups
3. **AES-NI instructions** → hardware AESENC/AESDEC (~10× speedup)
4. **Replace pyaes with OpenSSL/PyCryptodome** → native crypto instructions eliminate Python overhead

---

Conclusion

- Unoptimized AES bottlenecked by **Python integer math** and **cache misses**
- **T-table optimization**:
    - Cache misses ↓84%
    - CPU cycles ↓2%
    - IPC ↑ slightly
- Runtime ↑ modestly (~2.4%) → **interpreter overhead dominates**
- Real performance gains require **native code / hardware acceleration**
