Crypto PyAES report

Overview

Advanced Encryption Standard (AES) 128 is a symmetric block cipher standard. It should be noted that for the algorithm, we will sometimes view the 128-bit block we want to cipher as a $4x4$ matrix, in which each cell is a byte. It is comprised from 3 subfunctions for key expansion:

1. **RotWord**: Rotate a word left by 1 byte.
2. **SubWord**: Apply S-box to each byte in a word.
3. **RconXOR**: XOR with $[RC(i), 00, 00, 00]$, in which RC is a fixed vector for our purposes, and i is the key number.

And 4 functions for the actual encryption:

1. **AddRoundKey**: XORs the state matrix with the round key.
2. **SubBytes**: Each byte in the state matrix is replaced with a corresponding value from a fixed substitution box.
3. **ShiftRows:** Cyclically shifts the rows of the state matrix by different offsets.
4. **MixColumns**: Modifies the location of bits in a fixed way inside each column of the matrix.

The way those functions are handled is in “rounds”. Before the algorithm runs, we can precompute the 11 round-keys:

1. Start with the original key split into 4 words (w[0], w[1], w[2], w[3]).
2. **Generate new words**:
    - **First word of each new round key**:
      Take last word of previous key → rotate 1 byte left → apply S-box → XOR with Rcon → XOR with first word of previous key
    - **Next 3 words**:
      Each word = previous word of this round XOR word from previous key in the same position
3. **Repeat** until you have **11 round keys** (for rounds 0–10).

There are 11 rounds total, that perform:

1. **Initial Round:** AddRoundKey
2. **Rounds 1–9:** SubBytes → ShiftRows → MixColumns → AddRoundKey
3. **Final Round (10th):** SubBytes → ShiftRows → AddRoundKey (no MixColumns)

The “crypto pyaes” benchmark evaluates the performance of a pure-Python implementation of the Advanced Encryption Standard (AES) block cipher in Counter (CTR) mode using the PyAES module. Its workload is to encrypt and decrypt data using AES in a loop to simulate real-world usage (such as in VPN apllications). In the pyperformance test, the key is 16 bytes in length (AES128), and the plain-text data is about 23,000 bytes long ("This is a test. What could possibly go wrong? " ˙500). The mode is CTR, but it differs from the traditional cascaded Initialization vector (IV) and a counter. Instead of having 12 bytes of unique IV and a counter for each block,

the PyAES implementation in pyperformance uses a 16-byte counter starting at 1. For each block, the counter is incremented to have a unique value. That unique value is encrypted using AES128, and then we use a XOR operation between that and the plaintext data block.

It is important to note that for a given plaintext, the operation is highly parallel, since for each block, the increment, AES, and XOR operations can run with no dependency on other blocks and their results.

The only library used is PyAES, which is inherently pure Python.

The key and plaintext are both the Python type "bytes", which represent raw byte data that are immutable and efficient for low-level operations. Internally, Python uses a C array of unsigned 8-bit integers (uint8_t) to store the data and some metadata.

During the analysis, we used different profiling tools for different purposes. For example, perf gives detailed information about C-level functions but lacks clarity for Python-level functions. Py-Spy is a different profiling tool that shines on Python analysis but struggles to give an in-depth view of C-level functions.

Initial Analysis

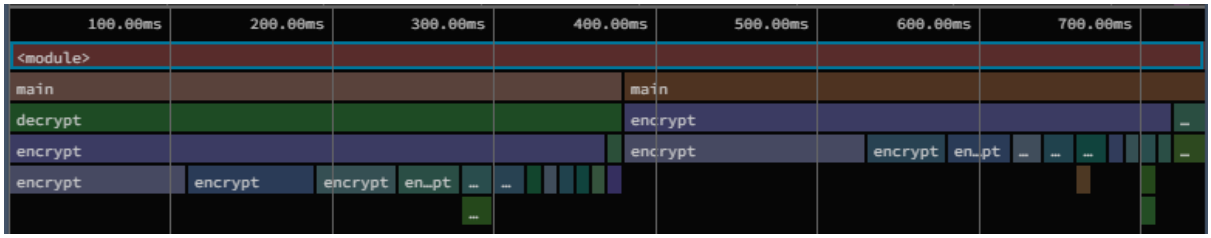

Figure 1 PyAES single run flamegraph- generated using py-spy

For analysing this benchmark, we used py-spy. Py-Spy works more natively with Python than perf and allows a clearer view.

Looking at the single-run PyAES flamegraph, as seen in Figure 1, we can notice that the functions that run for the longest time are the encrypt and decrypt functions inside PyAES.

The encrypt function, as seen in Figure 2 does a lot of memory accesses. It implements both the sub-bytes and the mix-columns functions by using tables, in which the given byte is treated as an index in the table, and the value after the two subfunctions is read (An example is seen with the use of T1 in the code shown in Figure 2). The other important function is the XOR ("^" operator).

Removing the loops by parallelization can help with the performance, as well as doing base mathematical operations at a lower level instead of pure Python.

Using the flamegraph in sandwich mode, we can also tell that the key expansion is an extremely small percentage in the overall work (which is why we do not see it in the flamegraph), and the encryption/decryption takes almost all the time.

```
# Convert plaintext to (ints ^ key)
t = [(_compact_word(plaintext[4 * i:4 * i + 4]) ^ self._Ke[0][i]) for i in xrange(0, 4)]

# Apply round transforms
for r in xrange(1, rounds):
    for i in xrange(0, 4):
        a[i] = (self.T1[(t[ i          ] >> 24) & 0xFF] ^
                self.T2[(t[(i + s1) % 4] >> 16) & 0xFF] ^
                self.T3[(t[(i + s2) % 4] >>  8) & 0xFF] ^
                self.T4[ t[(i + s3) % 4]        & 0xFF] ^
                self._Ke[r][i])
    t = copy.copy(a)

# The last round is special
result = [ ]
for i in xrange(0, 4):
    tt = self._Ke[rounds][i]
    result.append((self.S[(t[ i           ] >> 24) & 0xFF] ^ (tt >> 24)) & 0xFF)
    result.append((self.S[(t[(i + s1) % 4] >> 16) & 0xFF] ^ (tt >> 16)) & 0xFF)
    result.append((self.S[(t[(i + s2) % 4] >>  8) & 0xFF] ^ (tt >>  8)) & 0xFF)
    result.append((self.S[ t[(i + s3) % 4]  & 0xFF] ^  tt ) & 0xFF)
```

Figure 2 The arithmetic code in the encrypt function inside pyaes.py

Optimizations

In this report, we will show multiple approaches to optimizing the AES CTR algorithm. The base mean running time for PyAES is 597 ms.

We will note that for some use cases, optimizations can focus on other aspects such as memory usage \ energy, etc, and optimizations will be different (like in tiny-AES-c).

We will examine Python extensions because many of them smartly use HW, using different techniques to give better performance without leaving the Python environment or making it harder on the developer.

We should also mention that the first implementation is extremely trivial and uses an existing library, and we took it as a challenge to outperform it. For each optimization, there is also a benchmark file and a validation file against PyAES.

PyCryptodome

First, we show an implementation using an existing efficient AES library called “PyCryptodome”, a Python package of low-level cryptographic primitives. It uses the Intel AES-NI (Intel Advanced Encryption Standard New Instructions) for its computation, which is a hardware accelerator accessed as instructions (like an instruction for performing a whole AES round), implemented in x86 architecture, enabling a single dedicated instruction to perform the calculations, instead of multiple generic arithmetic and logical instructions.

Numpy & Numba

Secondly, we decided to implement the algorithm more manually using existing popular libraries. As mentioned before, AES CTR is extremely parallel, so we decided to use libraries efficient in parallel computing of numeric calculations. We decided to use NumPy because of familiarity. To further improve performance we also used the Numba package, which gives us a Just In Time (JIT) compiler that accelerates performance by compiling the Python functions to machine code instead of using the traditional bytecode. We should also mention that it is tailored for arithmetic operations and works seamlessly with NumPy. Numba also optimizes the compilation per CPU, including the use of AVX if it's available.

Main changes we made to improve performance:

@NJIT decorator

For almost all functions, we used the Numba “@njit” decorator for maximal performance. By also mentioning that $cache = True$ in the decorator, we kept the compiled version of functions in the memory so they wouldn’t need to be recompiled. We are interested in the case of multiple runs, not the first run, so this gives us a better approximation of real life use.

We also tried $parallel = True$ to take advantage of multiple threads, if possible, but since the QEMU only supports one processor, it created overhead and was not beneficial.

We also implemented some functions that receive and return integers as type uint8, instead of regular int, for better memory usage and better cache locality (more data in the same cache line).

NumPy vectorized operations

The NumPy library is designed for heavy arithmetic calculations by processing vector data types instead of Python arrays. This means that it can use existing optimized C libraries and CPU SIMD vector processing hardware (like AVX). It can also reduce Python object overhead (type tags, pointers, etc.).
Since NumPy arrays are contiguous blocks of raw numbers in memory, cache misses are lessened, and using C loops instead of Python loops removes a lot of overhead.
We should also mention that some integer operations, like incrementing the counter, were changed to bit-wise operations to remove branch prediction and use simple instructions.

Minimal memory allocation

In this implementation, we allocate most of the needed memory at the start of the run, utilizing memory view so there are no copies of the data, and all processing is in-place, so there is no creation of unnecessary objects.

C-based AES NI

The main difference we noticed between PyCryptodome and our NumPy implementation was the use of AES-NI, so we tried utilizing it.

We implemented the entire AES CTR algorithm in C, calling the AES-NI hardware directly. This has removed some Python overhead that existed while using the wrapper.

Unlike the NumPy implementation, here there are very few explicit uses of SIMD instructions as the AES-NI commands already handle the whole 128 bits present in keys, counters, and data. There is still use of a SIMD instruction for 128-bit XOR operations.

In this implementation, the Python overhead is even smaller than when using NumPy, and similarly, the memory allocation is minimal.

Cython-based AES NI

Cython is a language that compiles Python-like code into optimized C code. It aims to allow writing c-level data types and functions while also enjoying Python's ease of use.
Our Cython-based solution is extremely like the C-based solution, as they both utilize the speed of compiled language and the basic AES-NI functions to achieve better performance.

Performance Comparison

We should note that all performance benchmarks and comparisons were done with the command python3-dbg.

PyCryptodome

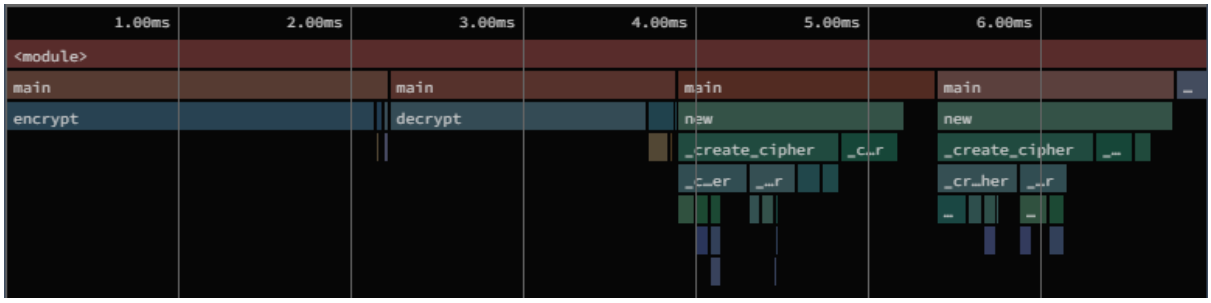

Figure 3 PyCryptodome flamegraph– created using py-spy

In PyCryptodome, we can see that the creation of the AES objects, which includes the key expansion algorithm, takes a much bigger percentage of the overall workload. This happens because the encryption and decryption portions are much shorter than before.

Unfortunately, we can't view the use of AEs-NI and the C-level function because it is an external library.

It has an average running time is $204\ us$, meaning a $x2{,}926$ speedup in comparison to PyAES.

Numpy & Numba

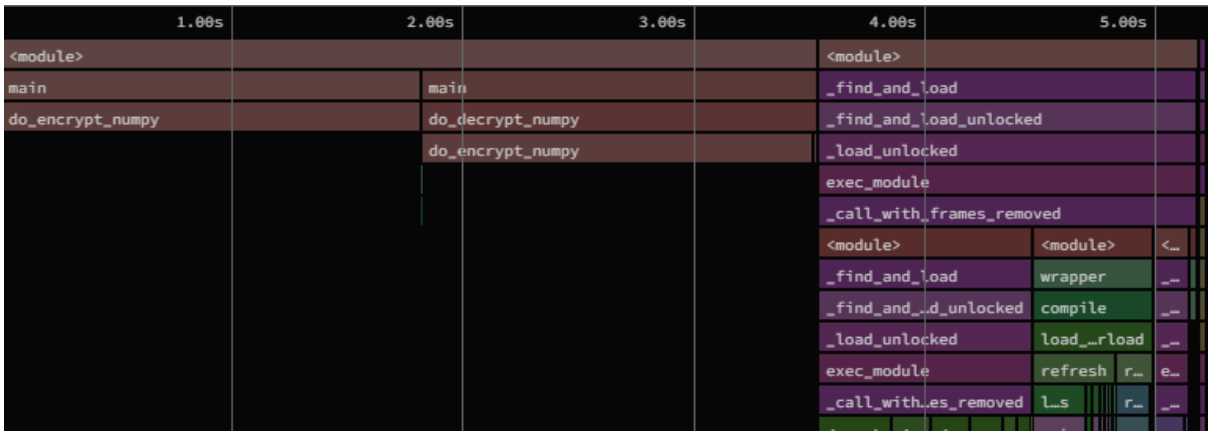

Figure 4 Numpy& Numba flamegraph - created using py-spy

Because of the Numba @NJIT decorator, we can see in the Figure 4 Numpy& Numba that a big chunk of the runtime is dedicated to the compilation section.

The use of NumPy works to our advantage by making the encryption and decryption functions much faster, but because all those functions are compiled, we cannot see how each sub-function is affected.

The average running time of $3.75\ ms$ meaning a $x159.2$ speedup in comparison to PyAES, which is nice but not nearly as good as PyCryptodome.

C-based AES NI

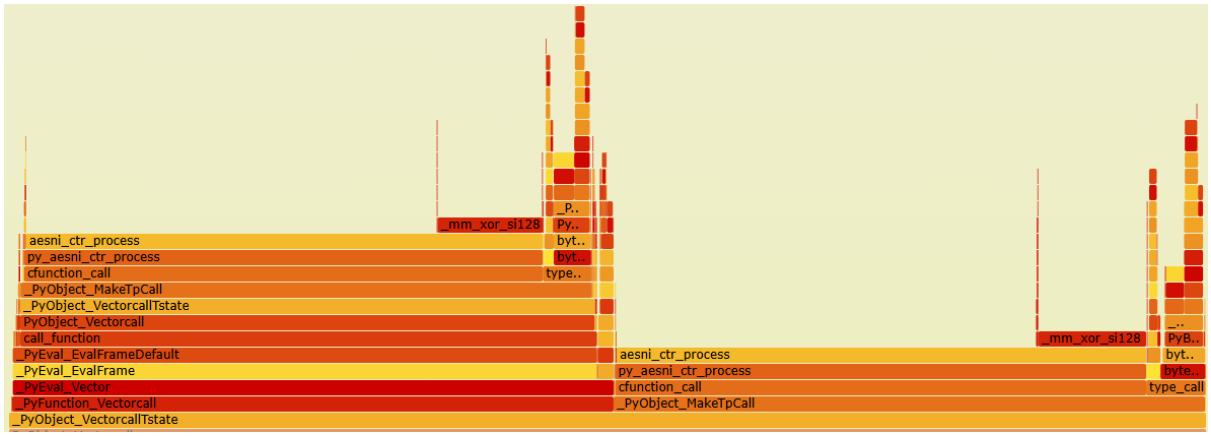

Figure 5 C-based AES NI implementation – created using perf

This implementation has an average running time of $127\ us$, meaning a $x4{,}700.8$ speedup in comparison to PyAES, and a $x1.6$ speedup in comparison to PyCryptodome.

The pure C implementation worked faster, as expected. While PyCryptodome utilizes AES-NI as well, in this, the whole “encrypt” \” decrypt” function is implemented in C, including the CTR part of the algorithm. This helped to improve performance over the PyCryptodome method, in which handling the modes is done in the Python code, creating more transfers from Python to compiled C and thus more overhead.

It is also interesting to note that while multiple AVX function calls were used explicitly, as well as AES-NI, only the XOR operations are visible in the flamegraph, because they take the most time, so they are easier to sample. We have attempted to use a 256 XOR operation instead of 128 (512-bit operations are not available on the HW setup), but it was slightly slower, probably due to overhead of concatenating bytes together.

Cython-based AES NI

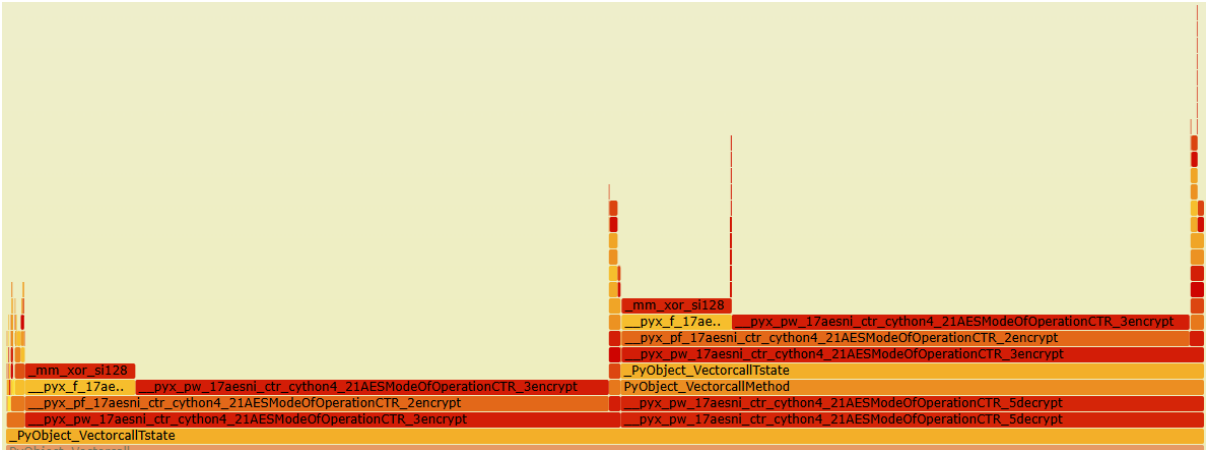

Figure 6 Cython-based AES NI implementation

It has an average running time of $109\ us$, meaning a $x5477$ speedup in comparison to pyaes, and a $x1.87$ speedup in comparison to PyCryptodome. This was the fastest implementation.

We have seen a 16.5% speedup in comparison to the C-based implementation. We suspect that the difference is in the Python overhead of the Python wrapper. This is supported by the flamegraphs, which show that the call stack for the C implementation is significantly deeper than for the Cython one, which negatively impacts performance.

It is very possible that when handling Python objects, Cython is much faster than the custom C extension. They both need to unwrap objects from the Python portion and wrap objects back up. Cython is designed to interface with Python and can handle both Python objects and C objects, switching between them more efficiently. It also utilizes Numpy, so under the hood, some other optimizations could be made.

Hardware Acceleration Proposal

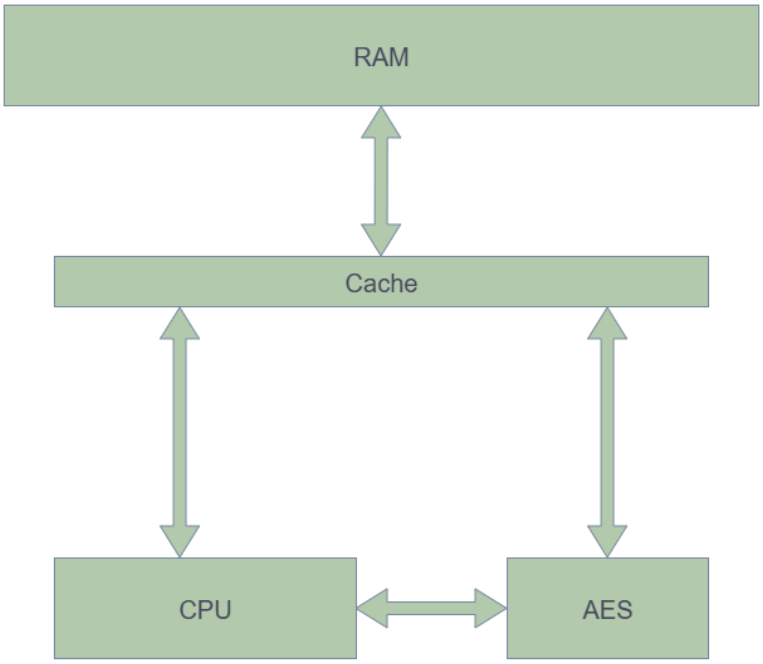

Figure 7 Our Hardware accelerator overview

After observing some papers with Processing in Memory ideas for implementing AES, we suggest processing the data "On the way" to the memory. We are suggesting a peripheral AES block that handles the process fully. The CPU will have memory-mapped access to the core, and the core will have Memory-Mapped access to the cache/Memory.

For encryption, the core will set the AES with parameters: the initial address to start writing from, the size of the plain text, the key, the mode of operation (Like CTR), and the number of rounds. After which the CPU will write a stream of data to the AES (We assume that the AES is pipelined for processing the data). The AES block will encrypt the data according to the parameters and write it to the cache/Memory, almost as a second core. If necessary, it will mark the addresses in the cache before encryption as dirty, so the CPU won't read old data, creating consistency in a similar way to how L1 caches update each other. When data is not dirty in the cache or is only found in the memory, the CPU can read it.

For decryption, the process will be the same but reversed. The CPU will set the parameters, ask for the AES block for the decrypted data as a burst, the AES will read the data from the memory, decrypt it, and send it to the CPU.

Another mode of operation for situations in which the CPU already has the plaintext data in the memory, especially assuming that the plaintext is large and can't fit in the CPU (like a text file). The CPU will give the AES parameters, and it will read and write the data itself.

It could be a little slower in comparison to pure in-memory computation, especially if the cache isn't utilized, since the AES block is far from the data, but if smaller chunks of data need to be ciphered, the first mode could be better, and even more secure since the plaintext is never on the external RAM.

It is important to note one key element that has both advantages and disadvantages: hardcoding different elements like the AES modes (CTR, for example), in the AES block itself. This will remove the overhead managed in the CPU using current hardware accelerators like AES-NI, but will also make improvements and changes to the algorithm not possible to implement (like a new mode of operation).

We believe that it is necessary in order to maximize performance, and that the inflexibility is tolerable, since AES hasn't changed much in recent years, at least officially by NIST, and the modes are commonly used as standard.

Conclusions

While Python is a very popular, useful language, it is not always the best choice for performance-intensive applications. Using external extensions with low-level functions can be extremely beneficial in reducing the overhead.

We explored 4 kinds of optimized implementations:

PyCryptoDom – An existing Python library made exactly for this use, built to give both the easiest interface and the best results. It enables the usage of different AES accelerators in CPUs. It was not optimized for each AES mode, in our case, CTR, which was still handled in Python and not at low level.

Numpy & Numba implementation – An implementation of the algorithm using popular Python libraries, such as the NumPy library, including NumPy variables, and Numba, which allows utilizing a JIT compiler for NumPy functions for efficient hardware and memory management, with no use of dedicated AES low-level extension.

C Implementation – Creating our own Python C-extension. Having lower-level control over the full CTR AES algorithm in C, directly controlling memory, and most importantly, using the AES-NI commands for calculations, resulted in an extreme speedup of 3-4 orders of magnitude.

Cython – Cython is often used to allow non-C developers to enjoy the advantages of C as a compiled language, but Cython was also built as a superset of Python, making integration extremely efficient. Cython creates an optimized C file, usually already better optimized than a manually written C code, and compiles it. This, the built-in integration, coupled with similar benefits to the ones from the C extension, gave us the best results.

For HW acceleration, we propose to use an accelerator that will handle all the overhead of the AES implementation and modes, to further improve performance and reduce the work needed to be done by the CPU, while making sure to keep access to the caches to not make a compute-bound problem into a memory-bound problem.

Sources

Tiny-aec-c

kokke/tiny-AES-c: Small portable AES128/192/256 in C

Cryptodome docs

Welcome to PyCryptodome's documentation — PyCryptodome 3.23.0 documentation

AES-NI explained

https://www.intel.com/content/www/us/en/developer/articles/technical/advanced-encryption-standard-instructions-aes-ni.html

Cython docs

Welcome to Cython's Documentation — Cython 3.2.0a0 documentation

Repository

YuvalMandel/PyperformancBenchmarkOptimization: Final Project of the Technion course 00460882 - Improving the performance of the pyaescrypto & gc_collect