Multithreaded File Encryption Using Libsodium and Python

Kush Patel

A report submitted as a part
of the requirements for
the investigatory project
of Class 12
at the Kokilaben Dhirubhai Ambani
Reliance Foundation School

November 2023

Supervisor Surjyakanta Bebortha

Preface

In an age where the digital landscape has become an integral part of our daily lives, the importance of privacy and data security has never been more paramount. With the growing concerns over data breaches, cyber-attacks, and unauthorized access to sensitive information, robust cryptography and encryption techniques have become the need of the hour. The task of safeguarding our digital assets and communications has fallen upon innovative technologies that can withstand the ever-evolving threats posed by malicious actors.

The report addresses one such aspect of these pressing concerns by exploring file encryption in Python, an interpreted language that is not conventionally known for its performance capabilities. This project delves into the realm of concurrency and aims to demonstrate how significant improvements in encryption performance can be achieved by leveraging the power of multithreading.

With the advent of powerful hardware and the increasing demand for processing vast amounts of data, the single-threaded approach may not always suffice. By employing multithreading, we aim to harness the potential of parallel execution, enabling faster encryption and decryption processes for large files.

This project report is an integral part of our investigatory project, a requisite for the completion of the Central Board of Secondary Education (CBSE) class 12 curriculum. It represents our collective effort to explore the fusion of modern cryptographic techniques and the capabilities of Python, offering readers an insight into the world of multithreaded I/O applications.

Acknowledgement

I would like to take this opportunity to express my heartfelt gratitude to all those who have been instrumental in the successful completion of this project. First and foremost, I extend my sincere thanks to our respected Principal for fostering an environment that encourages academic exploration and innovation. I am deeply indebted to my Computer Science teacher, Mr. Surjyakanta Bebortha, whose expertise and enthusiasm for the subject have been an immense source of inspiration. Your guidance, patient explanations, and insightful feedback have played a pivotal role in shaping the direction of this project. I would also like to extend my heartfelt appreciation to my family and friends for their unwavering support and understanding throughout this endeavor.

Finally, I express my gratitude to all the individuals who have contributed directly or indirectly to the completion of this project. Their support and encouragement have been invaluable in making this project a reality.

Contents

Chapter 1

Introduction

In this project report, we explore multithreaded cryptographic operations in Python, where we try to implement essential Python concurrency techniques to improve performance. Our focus lies in harnessing the capabilities of Python's *concurrent.futurers* module, which offers ability to launch parrellel tasks, and use it for encryption using the *PyNaCl* module, which provides bindings for the C-based Sodium library. By integrating PyNaCl and using concurrency, we aim to demonstrate performance gains over similar single-threaded Python implementations.

1.1 Background

During the initial phases of this project, we observed that a simple Python encryption program suffered from the under-utilization of hardware resources. Python, being an interpreted language, inherently exhibits slower performance compared to low-level compiled languages like C or C++. Consequently, when attempting to sequentially encrypt file chunks of substantial sizes, particularly files exceeding 500 megabytes, the process became exceptionally sluggish.

The sluggishness prompted us to explore alternative approaches to enhance the performance of our encryption program. We recognized that harnessing concurrency features in Python could hold the key to significant performance improvements. By introducing concurrent execution, we aimed to maximize the utilization of available hardware resources and reduce encryption and decryption times dramatically.

Through the implementation of multithreading, we sought a parallel encryption process, allowing multiple chunks of files to be encrypted simultaneously. This approach was expected to reduce the overall encryption time at the cost of increased usage of hardware

resources.

In this report, we present our findings and outcomes from the project, showcasing the remarkable impact of using concurrency in Python along with the PyNaCl library. We will also present a complete implementation in the form of CryptBuddy — a command-line file encryption software.

1.2 Chapter List

The following list provides a brief summary of each chapter in this report.

Chapter 2 XSalsa20Poly1305 and PyNaCl. This chapter deals with the introduction of PyNaCl and the usage of cryptographic algorithm XSalsa20Poly1305 for symmetric encryption.

Chapter 3 Concurrency in Python. This chapter explores concurrency features provided by Python and its advantages as well as drawbacks.

Chapter 4 Implementation. This chapter deals with the implementation of a multi-threaded encryption program with optimized performance.

Chapter 5 Evaluation & Testing. This chapter presents the statistics and performance benchmarks for the program.

Chapter 6 Conclusion. The conclusions of the report.

Chapter 2

XSalsa20Poly1305 and PyNaCl

PyNaCl is a library that provides Python bindings to the C library Libsodium. The Sodium cryptographic library, by Denis 2013 is a modern cross-compatible and packageable fork of the Network and Cryptography library (`https://nacl.cr.yp.to/`) with extended API. In the context of this report, the encryption algorithm used by Libsodium for it's authenticated secret-key (Symmetric-key) encryption will be investigated, namely, the XSalsa20 stream cipher and Poly1305 authentication algorithms. Symmetric-key ciphers use the same cryptographic key for encryption of plaintext and decryption of the ciphertext, so produced.

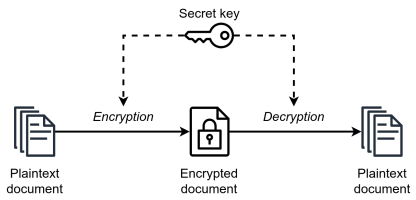

Figure 2.1: Simple Symmetric-key Encryption, By MarcT0K (icons by JGraph) - Own work, CC BY-SA 4.0

2.1 Specifications of XSalsa20 Cipher

Similar to a simple symmetric-key algorithm as shown in Figure 2.1, The XSalsa20 stream cipher needs a 192-bit unique nonce value in addition to the secret key. The XSalsa20 cipher is the successor of the Salsa20 cipher which required a 64-bit nonce value, as explained in the specifications, by Bernstein 2011. The XSalsa20 is a fast and secure encryption algorithm and safe against most attacks.

As per the context of encryption of large files, the details of the following must be

researched:

1. The PRG period of the cipher which will determine the maximum safe size of plaintext that can be encrypted without risk.

2. The performance and resource usage for encrypting long plaintext messages.

2.2 Limitations of XSalsa20Poly1305

The maximum safe length of the plaintext is virtually infinite for XSalsa20, as Salsa20 uses a 64-bit block counter and blocks of 64 bytes, limiting it's PRG period to 2^{73} bits. This implies that mathematically, any file can directly be encrypted using the XSalsa20 cipher regardless of it's size. However, the time required for computation of such large files, as well as the amount of memory allocated for it at once must be considered by the user. As per the claim of this report, time required for encryption of large files should be greatly reduced by dividing the data into chunks and encrypting them parallelly. The authentication algorithm Poly1305 will also not limit the file size, because the ciphertext length field in the construction of the buffer on which Poly1305 runs, limits the ciphertext (and hence, the plaintext) size to 2^{64} bytes, as described in the article by Nir and Angeley 2018

However, there is one significant limitation of the XSalsa20 cipher that must be taken into serious consideration; the same nonce value with the same key must not be used for encrypting different plaintexts. If multiple plaintexts are encrypted using the same keystream, an adversary can use these messages to infer information about the plaintexts. Therefore, XSalsa20 generates random keystrings for every message using a randomly generated nonce along with the key, allowing the sender to preserve the same encryption key among messages. Hence, when encrypting the chunks parallelly usage of a unique nonce for encryption of each chunk must be ensured.

2.3 PyNaCl and CFFI

PyNaCl (`https://pynacl.readthedocs.io/en/1.4.0/`) provides Python bindings to Libsodium. PyNaCl uses C Foreign Function Interface for Python (CFFI) to provide bindings to the C functions in Libsodium. One important thing to note here, is that the C function calls are done with Python's Global Interpreter Lock released. This is specified in the CFFI manual by Rigo 2023. In CPython, the Global Interpreter Lock, or GIL, is a mutex that protects access to Python objects, preventing multiple threads from executing Python bytecodes at once. Without the GIL being released, it would not be possible to achieve parallelization through multithreading.

Chapter 3

Concurrency in Python

Python provides both, multithreading and multiprocessing features for parallel applications. However, the limitations when working with the CPython interpreter must be evaluated.

3.1 Multithreading in Python

Python has built-in support for threads. The Python Software Foundation 2023b describes threads as light-weight tasks that can share a global data space. The low level module to achieve fine-grained control of threads in Python is *_threads*. This report will be using an abstract class *concurrent.futures.ThreadPoolExecutor()* on top of *_threads* which will execute parallel tasks in a thread pool.

The major drawback of using threads is that the GIL does not allow parallel execution among threads and only one thread can run Python bytecode at a time. This makes threading practically useless for most Python applications. However, certain libraries that use bindings to C functions can overcome this issue by releasing the GIL. As already discussed in the Section 2.3, C function calls done using CFFI will be executed with the GIL released, which will allow true parallelization of tasks in encryption using PyNaCl.

Another limitation of multithreading in Python is that the threads will not necessarily divide the tasks among different CPU cores, which means that most of the CPU cores will remain unutilized by the parallel tasks.

```
1 from concurrent.futures import ThreadPoolExecutor
2 values = [2,3,4,5]
3 def square(n):
4    return n * n
```

```
def main():
   with ThreadPoolExecutor(max_workers = 3) as executor:
      results = executor.map(square, values)
for result in results:
      print(result)
if __name__ == '__main__':
   main()
```

Listing 3.1: An Example of a Thread Pool Implementation using ThreadPoolExecutor()

Listing 3.1 provides a simple example of usage of the thread pool executor. It is important to note here that this will not improve performance due to GIL.

3.2 Multiprocessing in Python

Multiprocessing spawns multiple, entirely separate processes for executing tasks parallelly. As mentioned by the Python Software Foundation 2023a, this will effectively side-step the Global Interpreter Lock and make the tasks run in parallel. It will also use multiple CPU cores as specified, and divide the tasks among them. Python has a built-in *multiprocessing* module for spawning processes similar to *threading*. However, *concurrent.futures.ProcessPoolExecutor()* will be used here.

There is significant overhead in execution time for spawning multiple processes, and serialization & deserialization of return values using pickle. It also adds Inter-Process Communication (IPC) overhead. Therefore, multiprocessing should only be used for highly CPU intensive tasks that have low data footprint passed to each function call. A simple implementation of *concurrent.futures.ProcessPoolExecutor()* is provided in Listing 3.2:

```
from concurrent.futures import ProcessPoolExecutor
values = [2,3,4,5]
def square(n):
   return n * n
def main():
   with ProcessPoolExecutor(max_workers = 3) as executor:
      results = executor.map(square, values)
for result in results:
      print(result)
if __name__ == '__main__':
   main()
```

Listing 3.2: An Example of a Process Pool Implementation using ProcessPoolExecutor()

Chapter 4

Implementation

After trying both the concurrency techniques mentioned in Chapter 3, experiments showed that multiprocessing was extremely slow as the encryption task has a high data footprint passed to the function call. This made it even slower than direct encryption. Hence, we settled on the multithreaded approach.

4.1 Basic Multithreaded Encryption

The implementation of a simple encryption program using thread pool is straightforward. First of all, a function for encrypting a single chunk of data can be defined as shown in Listing 4.1.

```
1 def encrypt_chunk(args):
2     chunk, box, nonce = args
3     return box.encrypt(chunk, nonce).ciphertext
```

Listing 4.1: The encrypt chunk function

The arguments are deliberately kept in a single tuple and destructured later, as it will be easier to pass them as a single tuple from the parent function from which this function will be called. The box is PyNaCl's *SecretBox()* class which can create authenticated ciphertext from the plaintext. The chunk and nonce are python byte objects.

Now, this function can be called for each chunk asynchronously using a thread pool. For this, *ThreadPoolExecutor()* from *concurrent.futures* module is used. It must be ensured that each chunk gets a unique nonce value for encryption, as stated in Section 2.2. For this, all the tuples (arguments) for the chunks are constructed at once, incrementing the nonce for each subsequent chunk, as shown in Listing 4.2.

```
from nacl.bindings import sodium_increment
from nacl.secret import SecretBox

KEYSIZE = SecretBox.KEY_SIZE
NONCESIZE = SecretBox.NONCE_SIZE
key = urandom(KEYSIZE)
nonce = urandom(NONCESIZE)
data = urandom(300 * 1024 * 1024)

box = SecretBox(key)
args = []
total = len(data)
i = 0
while i < total:
    chunk = data[i : i + chunksize]
    nonce = sodium_increment(nonce)
    args.append((chunk, box, nonce))
    i += chunksize
```

Listing 4.2: Creating arguments for each chunk

Hence, The list of all arguments to be passed to the *encrypt_chunk()* function is obtained as the *args* variable.

Finally, the thread pool executor is used to map the arguments to the *encrypt_chunk()* function to get the list of output chunks, encrypted in parallel threads, as shown in Listing 4.3.

```
with ThreadPoolExecutor(max_workers=4) as executor:
    out = executor.map(encrypt_chunk, args)
```

Listing 4.3: Multithreaded Encryption

To assemble the code in useful functions, the complete implementation for multi-threaded encryption is given in Listing 4.4

```
from concurrent.futures import ThreadPoolExecutor
from os import urandom

from nacl.bindings import sodium_increment
from nacl.secret import SecretBox


def encrypt_chunk(args):
    chunk, box, nonce = args
    return box.encrypt(chunk, nonce).ciphertext

```

```
12
13 def encrypt(
14     data: bytes,
15     key: bytes,
16     nonce: bytes,
17     chunksize: int
18 ):
19     box = SecretBox(key)
20     args = []
21     total = len(data)
22     i = 0
23     while i < total:
24         chunk = data[i : i + chunksize]
25         nonce = sodium_increment(nonce)
26         args.append((chunk, box, nonce))
27         i += chunksize
28     with ThreadPoolExecutor(max_workers=4) as executor:
29         out = executor.map(encrypt_chunk, args)
30     return out
```

Listing 4.4: Complete Multithreaded Encryption Implementation

And, the corresponding functions for decryption is given in Listing 4.5

```
1  from concurrent.futures import ThreadPoolExecutor
2  from os import urandom
3
4  from nacl.bindings import sodium_increment
5  from nacl.secret import SecretBox
6
7  def decrypt_chunk(args):
8      chunk, box, nonce = args
9      return box.decrypt(chunk, nonce)
10
11 def decrypt(data: bytes, key: bytes, nonce: bytes, chunksize: int, macsize: int):
12     box = SecretBox(key)
13     args = []
14     total = len(data)
15     i = 0
16     while i < total:
17         chunk = data[i : i + chunksize + macsize]
18         nonce = sodium_increment(nonce)
19         args.append((chunk, box, nonce))
20         i += chunksize + macsize
21     with ThreadPoolExecutor(max_workers=4) as executor:
22         out = executor.map(decrypt_chunk, args)
23     return out
```

Listing 4.5: Complete Multithreaded Decryption Implementation

4.2 Working With The Filesystem

As the basic implementation for encryption and decryption is completed, large files can be read and written in binary mode using Python's *open()* function. As the files are very large and the data cannot be directly loaded into memory, they can be sequentially read in parts and encrypted part-by-part and the parts will also be written to output file sequentially. Listing 4.6 depicts the code required for this solution.

```
1  inp = "secret_message.dat"
2  out = "encrypted_message.dat"
3
4  KEYSIZE = SecretBox.KEY_SIZE
5  NONCESIZE = SecretBox.NONCE_SIZE
6  key = urandom(KEYSIZE)
7  nonce = urandom(NONCESIZE)
8  chunksize = 5 * 1024 * 1024
9  maxpartsize = 1 * 1024 * 1024 * 1024 # Limit the part size as per available memory
10
11 with open(inp, "rb") as infile:
12     with open(out, "wb") as outfile:
13         while infile.tell() != -1:
14             partdata = infile.read(maxpartsize)
15             encypted_part = encrypt(partdata, key, nonce, chunksize)
16             outfile.write(encypted_part)
```

Listing 4.6: I/O Implementation With Large Files

Chapter 5

Evaluation & Testing

Several benchmarks on the implementation provided in Listings 4.4 and 4.5 were performed using the *pyperf* (`https://pyperf.readthedocs.io/en/latest/`) toolkit on a machine with 7.8GB usable RAM, and 11th Gen Intel(R) Core(TM) i3-1115G4 CPU. Python v3.9.13 was used on Windows 11 Home build 22621.2134. The results are presented here.

5.1 Benchmarking Direct Encryption

Direct encryption of 300MB of random data takes $729ms \pm 37ms$ on average. Table 5.1 is the summary of 40 runs with 120 values, each taken with 5 loop iterations.

Result	**Time** (milliseconds)
Minimum	704
Median	724
MAD	6
Mean	729
Standard Deviation	37
Maximum	993

Table 5.1: Direct encryption of 300MB of data

The results are summarized as through a histogram in Figure 5.1.

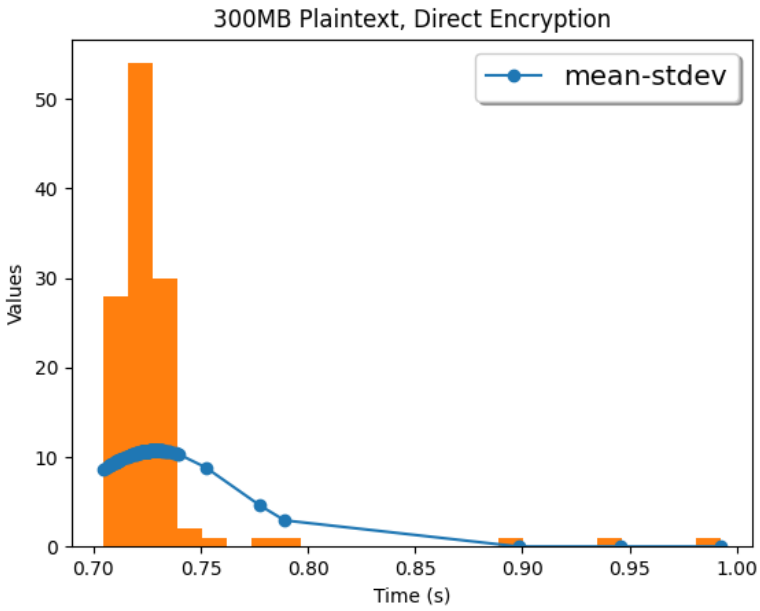

Figure 5.1: Direct Encryption Statistics

5.2 Benchmarking Multithreaded Encryption

The benchmarks for the multithreaded encryption implementation from Listing 4.4 were taken by dividing 300MB of random data into chunks of different sizes. Figure 5.2 shows the graph for execution time for using different chunk sizes.

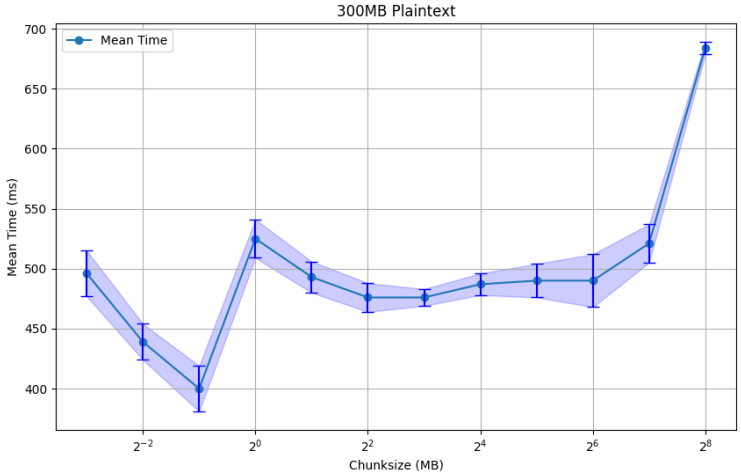

Figure 5.2: Multithreaded encryption times for various chunk sizes.

Encrypting chunks of smaller sizes upto 64MB has relatively low execution times, whereas, for larger chunks of over 64MB, the execution time increase significantly. This is because the entire data will not be distributed among all four workers, resulting in some workers remaining unused. Using chunks of sizes under 1MB perform significantly better. While chunks of 128KB perform slower, as the number of tasks created will be responsible for increasing the load on the thread pool, using chunks of 512KB results in the lowest execution time, averaging at $400ms \pm 19ms$. This is because the small size of chunks can fit in the L3 cache of the processor, and will also not create too many tasks for the four workers.

5.3 Optimum Performance with Chunks of 512KB

As 512KB has the least execution time experimentally, further benchmarking was done with 120 values, similar to Section 5.1, to get the results as summarized in Figure 5.3.

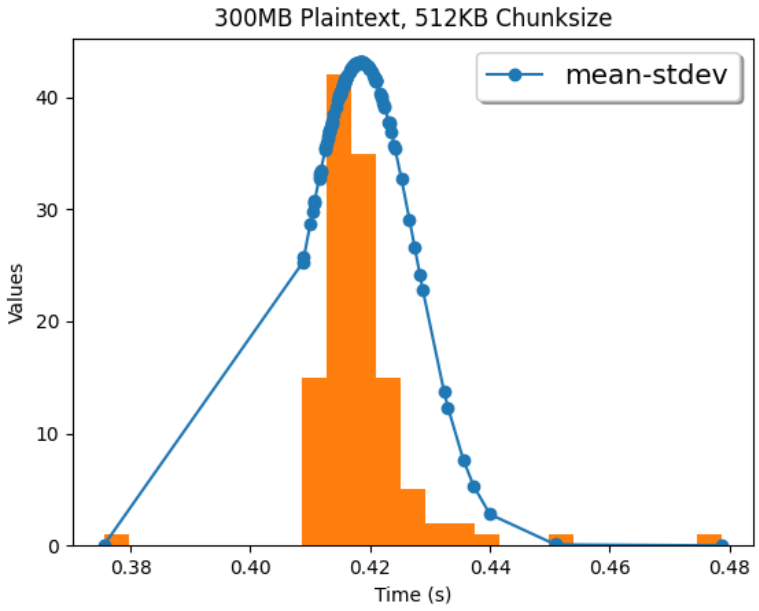

Figure 5.3: Statistics for chunks of size 512KB

5.4 Conclusions

The encryption time for most chunk sizes from 1MB to 64MB is approximately 1.53 times faster than direct encryption. However, the peak performance attained by 512KB chunks is 1.82 times faster than direct encryption. This proves the claim of the report, as the benchmarks show that multithreading has successfully reduced execution time.

Chapter 6

Conclusion

The Section 5.4 summarizes the performance gains through the detailed benchmarks on 300MB of random data. The implementation provided in Listing 4.6 can be used to create the foundation of a robust encryption suite in Python. CryptBuddy (https://github.com/Quat3rnion/cryptbuddy) has been created by the authors of this report, as a fully featured command-line tool for multithreaded encryption and has several features for security of large files along with fast encryption and decryption. CryptBuddy consistently outperforms the old implementation with over 40% decrease in execution time for both encryption as well as decryption of files over 1GB.

6.1 Conclusions

Even though Python is a single-threaded interpreted language not particularly known for performance, by the usage of multithreading, the performance of encryption using a state of the art stream cipher, XSalsa20. Encryption of large files spanning across multiple GBs of data can be encrypted within seconds using parallelism. CryptBuddy (https://github.com/Quat3rnion/cryptbuddy) is a performant CLI encryption suite written in Python by using the techniques described in this report.

6.2 Future Work

To improve the performance even further, we can optimize and create a custom multiprocessing module that can manage the high amounts of data across multiple processes through efficient IPC. This can potentially harness the power of multiple CPU cores to boost performance more significantly than the multithreaded approach.

Bibliography

Rigo, Armin (2023). *CFFI Documentation*. URL: https://cffi.readthedocs.io/_/downloads/en/release-1.15/pdf/.

Bernstein, Daniel J. (2011). "Extending the Salsa20 nonce". In: *cr.yp.to*. URL: https://cr.yp.to/snuffle/xsalsa-20110204.pdf.

Denis, Frank (June 2013). *Sodium Cryptographic Library*. URL: https://doc.libsodium.org/.

Nir, Y. and A. Angeley (June 2018). "ChaCha20 and Poly1305 for IETF Protocols". In: *RFC Series*. URL: https://www.rfc-editor.org/rfc/rfc8439.

Python Software Foundation (July 2023a). *Process-based parallelism*. URL: https://docs.python.org/3.9/library/multiprocessing.html.

– (July 2023b). *Thread-based parallelism*. URL: https://docs.python.org/3.9/library/threading.html.

Appendix A

Project Specification

Summary of the project outline.

A.1 Software Specifications

All the implementations and programs provided in the report used the following software:

1. Python v3.9.13
2. PyNaCl v1.5.0
3. pyperf v2.6.1

The operating system used is Microsoft Windows 11 Home build 22621.2134.

A.2 Hardware Specifications

The following are the specifications of the hardware used to run the programs provided in the report:

1. 11th Gen Intel(R) Core(TM) i3-1115G4 @ 3.00GHz 3.00 GHz processor
2. 8.00 GB (7.80 GB usable) RAM
3. 64-bit operating system, x64-based processor