Implementing the BLAKE3 algoritm

Karl Hannes Veskus
Institute of Computer Science, University of Tartu

University of Tartu

Introduction

BLAKE3 [2] is a cryptographic hash function, released in 2020, that is designed to be:

- Consistenly fast on all platforms
- Easy to use
- Easy to implement
- Secure

The goal of this project was to implement BLAKE3 in Python, while retaining the speed and security promised by it's designers.
Implementing a cryptographic hash function properly is non-trivial as the algorithm works at a very low-level and the security properties should still hold. This means that any language peculiarities and deviations from the original algorithm have to be thought through carefully and implemented with precision. Additionally, as the function is recent and state-of-art, it is reasonable to assume that even understanding the algorithm might present challenges.

Preliminaries

Before talking about BLAKE, one needs to know what cryptographic hash functions are in general. We can informally define them as follows:
A **hash function** is a function that takes potentially long bitstrings as inputs and outputs a fixed length bitstring. [3]
A **cryptographic hash function** is a hash function also satisfies a number of security properties, like collision resistance and preimage resistance.
Collision resistance means that it should be hard to find two inputs that give the same output. Preimage resistance means that given an output of a hash function it should be hard to find the corresponding input.

Origins

The predecessor of BLAKE3, named BLAKE, was born between 2007-2008 as a submission for a contest held by the United States National Institute of Standards and Technology (NIST) to create a new cryptographic hash function standard, SHA-3. Even though the BLAKE algorithm did not win the contest, it was among the 5 finalist. [1]
The immidiate successor, BLAKE2, was created in 2012, soon after the winner of the competition for SHA-3 was announced. The goal behind BLAKE2 was to optimize the BLAKE algoritm for modern systems, while removing redundant competition requirements set by NIST. BLAKE2 quickly raised in popularity and today it is widely adopted, including in, for example, WinRAR software and both Python and Go standard libraries. [1]
In early 2020 BLAKE3 was presented as the state-of-art update to the BLAKE family. The goal of BLAKE3 was to unify all versions of BLAKE2 into one single algorithm, offer further performance boosts by utilizing more parallelisation, and extend the possible uses for the hash function.[2]

BLAKE3

The BLAKE3 algorithm takes an input of any byte length and hashes it into an output of any specified byte length. The algorithm can be broadly split into 3 separate phases: initialization, hashing, and finalizing.
In the initialization phase, the input data is split into chunks of 1024 bytes. Each chunk is processed independently in parallel. The hash values of each chunk are kept in a binary Merkle tree. The structure of tree comes from the following two rules:

- "Left subtrees are full. Each left subtree is a complete binary tree, with all its chunks at the same depth, and a number of chunks that is a power of 2." [2]
- "Left subtrees are big. Each left subtree contains a number of chunks greater than or equal to the number of chunks in its sibling right subtree." [2]

An example of the tree structures with 1 to 4 chunks are given on Figure 1. Additionally, as the input can be of any length, the last chunk may be shorter than 1024 bytes, but never empty (unless the input is empty, in which case there is just the root node).

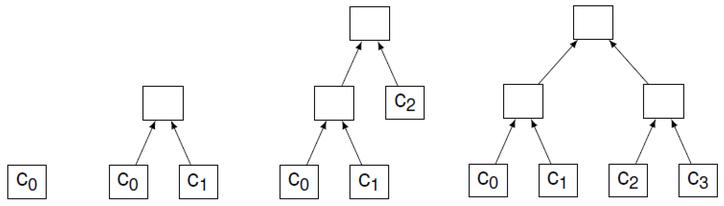

Fig. 1: Tree structure [2]

The hashing phase consist of splitting the chunks into blocks of 64 bytes (padding the last block if necessary) and iteratively hashing each of the blocks one after another using the previous hash as an input to the compression function.
The compression function acts as the core of the algorithm. It generates an internal state V which it then 'scrambles' using 7 rounds of bitwise operations G_i. Both V and G_i are shown on Figure 2. A single round in the compression consists of 8 instances of the function G_i:

$$G_0(v_0, v_4, v_8, v_{12}) \quad G_1(v_1, v_5, v_9, v_{13}) \quad G_2(v_2, v_6, v_{10}, v_{14}) \quad G_3(v_3, v_7, v_{11}, v_{15})$$
$$G_4(v_0, v_5, v_{10}, v_{15}) \quad G_5(v_1, v_6, v_{11}, v_{12}) \quad G_6(v_2, v_7, v_8, v_{13}) \quad G_7(v_3, v_4, v_9, v_{14})$$

$$V = \begin{pmatrix} v_0 & v_1 & v_2 & v_3 \\ v_4 & v_5 & v_6 & v_7 \\ v_8 & v_9 & v_{10} & v_{11} \\ v_{12} & v_{13} & v_{14} & v_{15} \end{pmatrix} \qquad G_i(a,b,c,d) = \begin{Bmatrix} a \leftarrow & a+b+m_{2i} \\ d \leftarrow & (d \otimes a) \ggg 16 \\ c \leftarrow & c+d \\ b \leftarrow & (b \otimes c) \ggg 12 \\ a \leftarrow & a+b+m_{2i+1} \\ d \leftarrow & (d \otimes a) \ggg 8 \\ c \leftarrow & c+d \\ b \leftarrow & (b \otimes c) \ggg 7 \end{Bmatrix}$$

Fig. 2: The internal state V, and the mixing function $G_i(a,b,c,d)$ [2]

Finally the hash values of each separate chunk are combined into their parents (using the same compression function) until reaching the root node. The final user-specified length hash value is received by re-hashing the value in the root node and appending it to the output in 64 byte increments. If the output is not a multiple of 64, the remainder is turncated. This kind of rehashing and adding allows for shorter outputs to be prefixes of longer outputs of the same input.

Results

From BLAKE3's 3 modes of operation, only the hashing function was implemented. While the project implementation hashes any length inputs to any length outputs as intended, it does not give the same hash values as the original BLAKE3 algorithm, indicating possible issues in the security of the implementation. Furthermore, the project implementation is roughly 10^3 times slower than the original implementation on small inputs ($n < 1000$), and 10^4 times slower on large inputs ($n > 10^5$). This is due to a severe lack of optimization and the inherent slowness of a purely Python implementation.

Comparison

As the original Rust implementation of BLAKE3 has also been ported to Python, along with several other well known hash algorithms, we can easily compare the real performance of BLAKE3 versus various hash algorithms on different length inputs. The comparison is graphed on Figure 3.

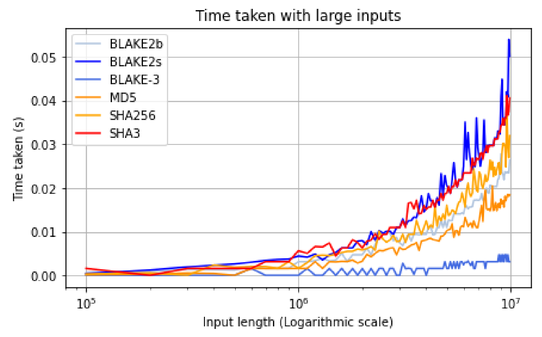

Fig. 3: Blake3 performance vs. other hash functions

With short inputs BLAKE 3 is just slightly faster than SHA-256, but still slower than MD5. However for longer inputs ($10^5 < n < 10^7$) BLAKE3 is almost an order of magnitude faster than the next fastest function, MD5. The comparison of average speeds is visible on Figure 4.

Hash function	Avr. time ($n < 10^3$)	Avr. time ($10^5 < n < 10^7$)
BLAKE2b	0.0000026 s	0.01259
BLAKE2s	0.0000030596 s	0.02015
MD5	0.0000019932 s	0.00952
SHA256	0.0000026333 s	0.01476
SHA3	0.0000031135 s	0.01911
BLAKE3	0.0000025966 s	0.002
Project BLAKE3	0.0045635849 s	2.8197479

Fig. 4: Blake3 performance vs. other hash functions

References

[1] Jean-Philippe Aumasson et al. *The Hash Function BLAKE*. Information Security and Cryptography. Springer, 2014. ISBN: 9783662447567.
[2] Jack O'Connor et al. *BLAKE3 one function, fast everywhere*. URL: https://blake3.io.
[3] Prof. Dr. Dominique Unruh. *Cryptology I - Short notes*. 2018.