

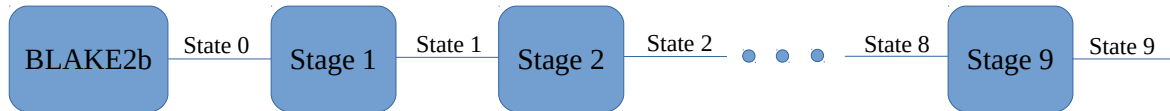
The following outlines xenoncat's implementation of Equihash (n=200, k=9) solver.

Original Equihash document: <https://www.internetsociety.org/sites/default/files/blogs-media/equihash-asymmetric-proof-of-work-based-generalized-birthday-problem.pdf>

1. Overview

Stage: The procedure that finds collision and execute XORs. For k=9, there are 9 stages.

State: Refers to the memory state, which is input and output of Stages. In most cases, State refers to Pairs and Xorwork.



Pairs: Tells which two items in the previous state are XORed together to create leading zeroes

Xorwork: The intermediate XOR value, created by XORing two items in the previous state's Xorwork.

Generally speaking, each Stage inputs about 2 million items and outputs about 2 million items. To minimize discarding of results, the design allows up to 2965504 items.

Pairs is an array of 2965504 items. Each item is 32 bits with pairs compression (advanced topic).

Xorwork is an array of 2965504 items. Each item is 200 bits at the beginning, and shrinks by 20 bits per Stage until 40 bits at the end.

1:1 item correspondence between Pairs and Xorwork. SoA instead of AoS for efficiency.

A new Pairs array is created for each Stage and is preserved until the end of algorithm for back tracking

Xorwork from previous Stages can be discarded and memory reused.

2. Binary Tree

No need to store a long list of Indices

Each pair is just a reference of 2 items in the previous State.

Stage 1 finds collisions in Xorwork 0, producing Xorwork 1 and Pairs 1.

Stage 2 finds collisions in Xorwork 1, producing Xorwork 2 and Pairs 2. Instead of having 4 Indices, each item in Pairs 1 is a reference to two items in Pairs 1.

Stage 8 finds collisions in Xorwork 7, producing Xorwork 8 and Pairs 8. Each item in Pairs 8 is a reference to two items in Pairs 7.

Stage 9 finds collisions in Xorwork 8, producing two candidate solutions. Each candidate solution references two items in Pairs 8.

For each candidate solution, to get the 512 Indices,

Lookup two items in Pairs 8, giving four items in Pairs 7.

Lookup four items in Pairs 7, giving eight items in Pairs 6.

...

Lookup 256 items in Pairs 1, giving 512 Indices for the solution.

3. Straightforward method

Each stage does 2 things: Finding collisions and executing the XOR.

The collision finding part examines approximately 2 million inputs for matching 20-bit pattern. The expected output is 2 million pairs.

If 3 inputs have the same 20-bit pattern, all combinations are output: 1-2 1-3 2-3

If 4 inputs have the same 20-bit pattern, all combinations are output: 1-2 1-3 2-3 1-4 2-4 3-4

This follows the "n choose k" rule.

hashtab: where it was most recently seen and how many times seen so far

A simple and direct way of doing the collision search is to have an array of 2^{20} elements which remembers where each of the 20-bit pattern was last seen.

Also, create an output buffer called workingpairs.

Then walk through the 2 million inputs, lookup the hashtab (most recently seen table) using the 20-bit pattern:

- case never seen before: update the hashtab with the current walking index
- case seen once: output the pair 1-2 (1 comes from the hashtab, 2 comes from current walking index). Update the hashtab to point to the pairs 1-2 we have just written.
- case seen twice: the hashtab is pointing to 1-2, the current walking index is 3; so output 1-3 2-3. Update the hashtab to point to the pairs 1-3 we have just written.

- case seen thrice: the hashtable is pointing to 1-3 2-3, the current walking index is 4; so output 1-4 2-4 3-4. Update the hashtable to point to pairs 1-4 we have just written.

Implementation detail:

Each hashtable element can be implemented as a uint32, with bitfields for "count" and "where".

When "count" is 0, "where" is ignored.

When "count" is 1, "where" refers to index within the 2 million inputs.

When "count" is >1, "where" refers to index within the workingpairs.

4. Buckets

Coarse level sorting: 256 buckets

At the output of BLAKE2b, there are exactly 2097152 items of 200 bits each. The items are distributed into buckets in State 0 according to bit[199:192].

In 256-bucket implementation, the Xorwork items are reduced by 8 bits because they are implied by being distributed to that particular bucket.

basemap is used to map the bucket distribution in State 0 to the correct Indices used in solution.

With buckets, the collision search become finding approximately 8200 inputs for matching 12-bit pattern. The expected output is 8200 pairs.

The collision search is repeated for each of the 256 buckets, in other words, hashtable is initialized when advancing to the next bucket.

Earlier implementations experimented with 2048 buckets, 1024 buckets and 512 buckets. Performance is best with 256 buckets.

[199:192]		Selects bucket in State 0
	[191:180]	Collision search in Stage 1
[179:172]		Selects bucket in State 1
	[171:160]	Collision search in Stage 2
[159:152]		Selects bucket in State 2
	[151:140]	Collision search in Stage 3
[139:132]		Selects bucket in State 3
	[131:120]	Collision search in Stage 4
[119:112]		Selects bucket in State 4
	[111:100]	Collision search in Stage 5
[99:92]		Selects bucket in State 5
	[91:80]	Collision search in Stage 6
[79:72]		Selects bucket in State 6
	[71:60]	Collision search in Stage 7
[59:52]		Selects bucket in State 7
	[51:40]	Collision search in Stage 8
[39:32]		Selects bucket in State 8
	[31:20]	Collision search in Stage 9
[19:0]		Check for equal for pairs produced in Stage 9

	Bits in Xorwork	Bytes, rounded up	Bytes, rounded up to multiple of 4	Method to extract 12 bits for collision
State 0	192	24	24	[22]>>4
State 1	172	22	24	[20]
State 2	152	19	20	[17]>>4
State 3	132	17	20	[15]
State 4	112	14	16	[12]>>4
State 5	92	12	12	[10]
State 6	72	9	12	[7]>>4
State 7	52	7	8	[5]
State 8	32	4	4	[2]>>4
State 9	0	0	0	

The following shows the memory layout of Pairs and Xorwork.

Pairs from each stage need to be preserved. Xorwork from previous stages can be overwritten.

Each column represents one unit of memory: left is low address, right is high address.

Each row represents progress through time: top is the beginning of the algorithm, bottom is the end of algorithm.

One unit of memory is the space occupied by one Pairs array: 2965504 items * 4 bytes = 11862016 bytes.





For State 0, the Xorwork of each item is 192bits=24bytes, therefore 6 units of memory is needed.

For State 8, the Xorwork of each item is 32bits=4bytes, therefore 1 unit of memory is needed.

It can be seen that 14 units of memory is needed for Pairs and Xorwork. There is also 1 unit of memory needed by basemap. So Equihash (200,9) needs about 15*11862016 bytes = 178MB.

This layout is implemented in the first version of solver. However, while writing this document, an alternative layout is discovered.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13
State 0									Red	Red	Red	Red	Red	Red
State 1	Green		Red	Red	Red	Red	Red	Red						
State 2	Light Green	Light Green								Red	Red	Red	Red	Red
State 3	Light Green	Light Green	Light Green		Red	Red	Red	Red	Red					
State 4	Light Green	Light Green	Light Green	Light Green	Light Green					Red	Red	Red	Red	Red
State 5	Light Green	Light Green	Light Green	Light Green	Light Green	Light Green		Red	Red	Red				
State 6	Light Green	Light Green	Light Green	Light Green	Light Green	Light Green	Light Green				Red	Red	Red	Red
State 7	Light Green	Light Green	Light Green	Light Green	Light Green	Light Green	Light Green	Light Green	Red	Red				
State 8	Light Green	Light Green	Light Green	Light Green	Light Green	Light Green	Light Green	Light Green	Light Green	Light Green	Red			
State 9	Light Green	Light Green	Light Green	Light Green	Light Green	Light Green	Light Green	Light Green	Light Green	Light Green	Yellow			

-  Newly written pairs
-  Pairs from previous stages preserved
-  Xorwork area
-  Uncompressed pairs output of Stage 9

Alternative memory layout which is more straightforward. The main idea is to avoid overlaps when transitioning from one state to the next state.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13
State 0		Red	Red	Red	Red	Red	Red							
State 1	Green								Red	Red	Red	Red	Red	Red
State 2	Green	Green		Red	Red	Red	Red	Red						
State 3	Green	Green	Green							Red	Red	Red	Red	Red
State 4	Green	Green	Green	Green		Red	Red	Red	Red					
State 5	Green	Green	Green	Green	Green						Red	Red	Red	Red
State 6	Green	Green	Green	Green	Green	Green		Red	Red	Red				
State 7	Green	Green	Green	Green	Green	Green	Green				Red	Red		
State 8	Green	Green	Green	Green	Green	Green	Green	Green		Red				
State 9	Green	Green	Green	Green	Green	Green	Green	Green	Yellow					

5. Partitions

For multithreading and Pairs compression.

A bucket has slots for 11584 items. With partitioning, it is organized as 4 partitions * 2896 items.

Partitioning does not affect collision search of finding among 8200 inputs for matching 12-bit. Hashtab is maintained when crossing partition boundary.

Usage of partition is illustrated through an example: Stage 2 processes State 1 into State 2:

- When reading out Buckets 0-63 of State 1, the pairs formed can only go into Partition 0 of State 2.
- When reading out Buckets 64-127 of State 1, the pairs formed can only go into Partition 1 of State 2.
- Similarly Buckets 128-191 of State 1, go into Partition 2 of State 2.
- And Buckets 192-255 of State 1, go into Partition 3 of State 2.
- Note that the decision of which Bucket in State 2 to write depends on the XOR result of making pairs (match leading 12 bits, XOR the pairs and get the next 8 bits).

To summarize, a State contains up to 2965504 items: 256 buckets * 4 partitions * 2896 items.

To decide which bucket to write: XOR the pairs and get the next 8 bits.

To decide which partition to write: The source Bucket ID. In multithreading scenario, the thread ID can come into play.

To decide which item to write: An array of counters (256*4) to remember number of items in each bucket*partition.

6. Pairs compression

A pair is a reference to two items in the previous State. As there are up to 2965504 items in each State, the uncompressed method of storing requires 22+22=44 bits.

Pairs compression allows a pair to be stored in 32 bits.

Putting aside Partitioning, Bucket ID ranges from 0-255, Item ID ranges from 0-11583.

To uniquely identify an item, we store both Bucket ID and Item ID.

For a pair, it becomes {(bucket0, item0) (bucket1, item1)}. But bucket0=bucket1, because items have to be in the same bucket to collide. So we store {bucket (item0, item1)}.

For the two Item IDs, the order does not need to be preserved, Indices will be sorted when forming the solution.

One item ID will be bigger than the other, we call the item IDs b (for big) and s (for small) respectively.

To compress: $x = b(b-1)/2 + s$;

To uncompress:

$$b = \text{round_to_nearest}(\sqrt{2x+1})$$

$$s = x - b(b-1)/2$$

We see that x fits in 26 bits. We have remaining 6 bits used for bucket ID.

We have 256 buckets, requiring 8 bits of bucket ID, this is where Partitioning by 4 comes in. The two missing bits can be recovered by checking which partition the pair is stored at.

Additional notes on Partition: Partitioning makes Item ID have gaps in between. The first item in Partition 0 is given the Item ID 0. The first item in Partition 1 is given the Item ID 2896.

In this way, Item ID 2895 is usually invalid (contains junk) because the bucket is not usually filled until full; but Item ID 2896 is valid.

7. Minor details

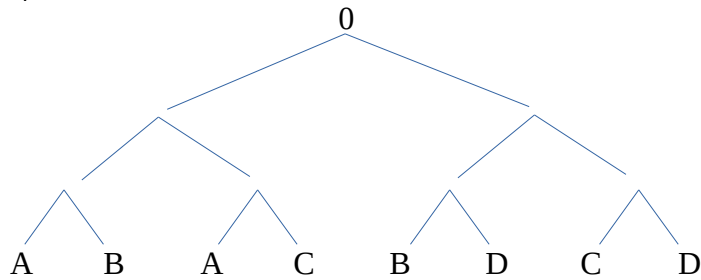
Cache locality

Pairs: $4 * 2896 * 256 * 4\text{byte}$

Xorwork: $256 * 4 * 2896 * 24\text{byte}$

Duplicate removal and trivial solutions

Example of trivial solution:



A duplicate is invalid XORing of pairs for example (A B) xor (A C).

It is likely not worth the time to detect duplicates during Stage 1 to 9.

Trivial solution will be detected and discarded from Stage 4 to 8. Not doing so will cause buckets to be filled up with many trivial solutions.

After Stage 9, the GetSolutions routine will detect duplicates when forming solutions.