Brute-Force Cryptanalysis with Aging Hardware: Controlling Half the Output of SHA-256

Mellila Bouam, Charles Bouillaguet, Claire Delaplace

▶ **To cite this version:**

Mellila Bouam, Charles Bouillaguet, Claire Delaplace. Brute-Force Cryptanalysis with Aging Hardware: Controlling Half the Output of SHA-256. 2019. hal-02306904v1

HAL Id: hal-02306904

https://hal.science/hal-02306904v1

Preprint submitted on 7 Oct 2019 (v1), last revised 26 Jun 2021 (v3)

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Brute-Force Cryptanalysis with Aging Hardware: Controlling Half the Output of SHA-256

Mellila Bouam[1], Charles Bouillaguet[2], Claire Delaplace[3]

[1] Ecole Superieure d'Informatique, Alger, Algeria
em_bouam@esi.dz
[2] Univ. Lille, CNRS, Centrale Lille, UMR 9189 - CRIStAL - Centre de Recherche en Informatique Signal et Automatique de Lille, F-59000 Lille, France
charles.bouillaguet@univ-lille1.fr
[3] Horst Görtz Institute for IT Security
Ruhr University Bochum, Germany
claire.delaplace@rub.de

Abstract. This paper describes a "three-way collision" on SHA-256 truncated to 128 bits. More precisely, it gives three random-looking bit strings whose hashes by SHA-256 maintain a non-trivial relation: their XOR starts with 128 zero bits. They have been found by brute-force, without exploiting any cryptographic weakness in the hash function itself. This shows that birthday-like computations on 128 bits are becoming increasingly feasible, even for academic teams without substantial means.
These bit strings have been obtained by solving a large instance of the three-list generalized birthday problem, a difficult case known as the 3XOR problem. The whole computation consisted of two equally challenging phases: assembling the 3XOR instance and solving it.
It was made possible by the combination of: 1) recent progress on algorithms for the 3XOR problem, 2) creative use of "dedicated" hardware accelerators, 3) adapted implementations of 3XOR algorithms that could run on massively parallel machines.
Building the three lists required $2^{67.6}$ evaluations of the compression function of SHA-256. They were performed in 7 calendar months by two obsolete second-hand bitcoin mining devices, which can now be acquired on eBay for about 80€. The actual instance of the 3XOR problem was solved in 300 CPU years on a 7-year old IBM Bluegene/Q computer, a few weeks before it was scrapped.
To the best of our knowledge, this is the first explicit 128-bit collision-like result for SHA-256. It is the first bitcoin-accelerated cryptanalytic computation and it is also one of the largest public ones.

1 Introduction

This paper reports on the computation of a three-way collision for SHA-256 truncated to 128 bits. SHA-256 is presently considered to be a secure hash function, and it is widely used. It was designed by the NSA in 2001, and it is one of the few hash functions standardized by the government of the United States of America for its own use [23].

We emphasize that the results presented in this paper do not undermine its security. We did not discover nor exploit any new cryptographic weakness. Instead, we make the practical demonstration that brute-force "birthday" attacks on 128 bits can be practically feasible in some circumstances.

In symmetric cryptology, where key and output sizes of actual functions are fixed (to 128 bits, 256 bits, etc.), it is usual to analyze weakened variants of secure cryptographic constructions. When considering functions where a "round function" is iterated, weakened versions can be obtained by reducing the number of rounds. One of the best cryptographic attacks [18] on SHA-256 is a practical collision on 28 rounds out of 64. This collision attack can be extended to 31 rounds with a computation of 2^{64} operations (which, to the best of our knowledge, has not been carried out).

We looked at a weakened version of SHA-256, obtained not by reducing the number of rounds but instead by truncating its output to 128 bits. This 128-bit variant could be denoted as $H :=$ SHA-256-128, according to the current terminology. This yields a 128-bit hash function which is presumably "secure", in the sense that no attack faster than brute-force is presently known. Of course, 128-bit hashes only offer "64-bit security" and this is way too low by today's standards. A collision, or collision-like results, could be obtained after 2^{64} evaluations of the function, even if it were perfectly secure, by the birthday "paradox". This level of effort is on the verge of practicality.

Indeed, we found a three-way collision by brute-force. More precisely, we obtained three bit strings x, y and z such that $H(x) \oplus H(y) \oplus H(z) = 0$, where $\oplus$ denotes the XOR operation. They are shown in figure 1. This was accomplished by solving an instance of the 3XOR problem, which is a generalized birthday problem [28] with three "lists". From an algorithmic point of view, it is an annoying case: the best algorithms require time $\tilde{\Theta}\left(2^{n/2}\right)$ versus $\mathcal{O}\left(2^{n/3}\right)$ for the four-list case (where n denotes the number of bits of the hash function to attack).

Finding this "three-way collision" necessitated a somewhat large computational effort which took place between October, 2018 and May, 2019. Because of the scarcity of our research funding, we had to make do with aging hardware, but this turned out to be sufficient. In total, we evaluated the compression function of SHA-256 $2^{67.6}$ times. To the best of our knowledge, this is the biggest reported cryptanalytic computation reported in a research work at this time (larger computations have been carried out in other fields; in addition, the bitcoin network does a lot more evaluations of SHA-256 every minute, but with a different purpose). As we discuss later, it is not the most expensive cryptanalytic computation, nor the hardest to carry out in practice.

The main ingredient that allowed us to go through with such a large computation is that it was ideally suited to exploit unconventional but (nearly) off-the-shelf hardware accelerators: bitcoin mining devices. These inexpensive and power-efficient devices contain dedicated ASICs devoted to the evaluation of SHA-256. A single bitcoin mining device is capable of evaluating SHA-256 at least one million times faster than a CPU core, but it only do so in a restricted way. Finding a three-way collision happened to be one of the few attacks we

Consider the three 80-byte ASCII strings given below :

$$a = \texttt{"FOO-0x000000003B1BD2039"} + \texttt{"␣"} \times 53 + \texttt{dd3ff46f},$$
$$b = \texttt{"BAR-0x00000000307238E22"} + \texttt{"␣"} \times 53 + \texttt{a80da323},$$
$$c = \texttt{"FOOBAR-0x000000001BB6C4C9F"} + \texttt{"␣"} \times 50 + \texttt{b01d7c21}.$$

Notice how the 64-bit hexadecimal counters take 17 characters; this is the result of an embarrassing off-by-one error.

Let $x \leftarrow \mathsf{SHA\text{-}256}(a), y \leftarrow \mathsf{SHA\text{-}256}(b)$ and $z \leftarrow \mathsf{SHA\text{-}256}(c)$. These hashes are random-looking bit strings:

$$\begin{aligned} x = {} & \texttt{2cf9b0f0 8cf86175 1f3faad0 4fee9fec} \\ & \texttt{99ac4305 69a48c7c 49d779d8 c4d34321}, \\ y = {} & \texttt{b9c9240a 4295ff73 fcd53d9b 559ff454} \\ & \texttt{64e9feb2 2d954f9c c7f12d5c 7910bbc0}, \\ z = {} & \texttt{d0f7153e e6ceb465 01583208 603423b5} \\ & \texttt{f0e2221b 81ccce79 5b0189d5 671bdcca}. \end{aligned}$$

Let us feed these values into SHA-256 again. This time, the results are special: seen as integers, they are less than 2^{224}.

$$\begin{aligned} \mathsf{SHA\text{-}256}(x) = {} & \texttt{00000000 1a3d266d 0cce284a 21ee2b70} \\ & \texttt{730f8603 62b84219 9af220b9 bdaee2a7}, \\ \mathsf{SHA\text{-}256}(y) = {} & \texttt{00000000 1a2dea9c 30f58ff7 24f4533a} \\ & \texttt{e2485711 a143b883 0db5cd0a efa96f60}, \\ \mathsf{SHA\text{-}256}(z) = {} & \texttt{00000000 0010ccf1 3c3ba7bd 051a784a} \\ & \texttt{efb83f87 a5a87be7 51873c64 aac9340b}. \end{aligned}$$

Let $\Delta \leftarrow \mathsf{SHA\text{-}256}(x) \oplus \mathsf{SHA\text{-}256}(y) \oplus \mathsf{SHA\text{-}256}(z)$, where $\oplus$ denotes the XOR operation, we finally obtain:

$$\begin{aligned} \Delta = {} & \texttt{00000000 00000000 00000000 00000000} \\ & \texttt{7effee95 6653817d c6c0d1d7 f8ceb9cc}. \end{aligned}$$

Fig. 1. The result: a 128-bit 3XOR on SHA-256.

knew of that could potentially be miner-accelerated. The two other ingredients are fast algorithms for the 3XOR problem and efficient parallel implementations thereof.

We spent 7 calendar months "mining" in order to build an instance of the 3XOR problem, using two bitcoin mining devices. In a second step, we solved this instance of the 3XOR problem in three calendar days using 65536 cores simultaneously, on an IBM BlueGene/Q computer, a massively parallel machine. This required about $2^{64.2}$ CPU cycles.

Related Work. Cryptographic attacks using dedicated ASICs are rare. The only example know to us is the "Deep Crack" machine designed in 1998 to break the DES by exhaustive search. It had 1856 custom ASICs. We note that we have not designed nor implemented any custom hardware, but we found a way to use "almost dedicated" ASICs for our own purposes. Many cryptographic algortihms and actual cryptanalytic attacks targeting real cryptographic constructions have been run on GPUs (see [25] for instance). Several cryptanalytic algorithms have also been run on FPGAs, for instance using the COPACOBANA machine [15].

The 3XOR problem has recently seen a renewed interest. Joux proposed an incremental improvement on the naive $2^{n/2}$ algorithm in [10]. Motivated by a generic attack [20] against the COPA [4] mode of operation for authenticated encryption, several new algorithms were discovered [21, 8]. A recent attack [17] against the two-round single-key Even-Mansour cipher works by reducing it to a 3XOR computation.

Several other cryptographic constructions can be attacked by solving instances of the generalized birthday problem with more than three lists. An example is [7], which describes an actual implementation of an attack against (reduced version of) the FSB hash function.

Outline. The algorithms we used to solve the 3XOR problem are described in section 2. Section 3 provides some background on bitcoin mining and how (aging) bitcoin mining devices can be exploited for cryptanalytic purposes. Section 4 dwells into the detail of our parallel implementation of a 3XOR algorithm and its tuning to the (aging) parallel computer at hand. Section 5 discusses the outcome of the computation and gives a summary of the whole experience.

2 Algorithms for the 3XOR Problem

Most of the algorithmic groundwork had been done in [8]. In this section, we give a high-level view of the algorithmic techniques used to carry out the computation. The precise details of tuning these algorithms to the actual hardware we used are given in section 4.

We seek a 3XOR triplet on SHA-256 truncated to $n = 128$ bits. For this, we use a two-step process: first we build three arrays of hashes `A`, `B` and `C` (of respective sizes A, B and C), then in a second time we look for 3XOR triplets

$(x, y, z) \in \mathtt{A} \times \mathtt{B} \times \mathtt{C}$ such that $x \oplus y \oplus z = 0$. We are free to choose the sizes of the arrays, and the existence of a solution is guaranteed with high probability as soon as $ABC \geq 2^{128}$.

To actually find these triplets, we considered the algorithms listed in Table 1 ; we deliberately disregarded the algorithm of Nikolić and Sasaki [21] because it is inferior to Joux's.

Algorithm	Ref.	Asymptotic running time	Size of input arrays
Quadratic		$AB + C$	any
Joux	[10]	$\sqrt{2^n/n}$	$A = B = \sqrt{2^n/n}, C = n/2$
Generalized Joux	[8]	$\frac{(A+B)C}{n - \log_2 \min\{A, B\}}$	any

Table 1. Generic algorithms for the 3XOR problem.

Data Volume and Clamping. Dealing with lists of size 2^{64} is intractable: it would require at least 128 exabyte, which much more space than even the biggest computer installation in the world could offer. This disqualifies Joux's algorithm.

Instead, we used a well-known trick: we "clamped" the first 32 bits of all hashes to zero. More precisely, to build each of the input arrays $\mathtt{A}, \mathtt{B}$ and $\mathtt{C}$, we evaluated $\mathsf{SHA\text{-}256} \approx 2^{64}$ times on arbitrary inputs and kept only the preimages whose hashes begin with 32 zero bits (this is visible on x, y, z in fig. 1). This results in three arrays of size 2^{32} containing a single 3XOR triplet on average.

This reduces our storage requirements to 96GByte, a much more practical amount. We exploited *ad hoc* hardware accelerators (bitcoin miners) to speed-up this "data-collection" phase, as discussed in section 3.

Making the Problem Parallel. Finding the 3XOR triplets in lists of size 2^{32} is going to require $\approx 2^{64}$ operations, which is a large computation that can only be carried out on a parallel computing infrastructure. This in turn requires a parallel algorithm. We considered implementing a parallel version of algorithm G (see fig. 2): this would require distributing $\mathtt{A}$, $\mathtt{B}$ and $\mathtt{C}$ amongst many compute nodes, and we believe that this would have made partitioning (step J2) very impractical because of the cost of communications between nodes that this would require. In addition, This step has to be repeated many times.

Instead, we used a simple divide-and-conquer approach: we split each input array in two according to the most significant bit of the hashes. This splits the original problem into four sub-problems of half the size: a 3XOR triplet in $(\mathtt{A}, \mathtt{B}, \mathtt{C})$ belongs to either $(\mathtt{A}_0, \mathtt{B}_0, \mathtt{C}_0), (\mathtt{A}_0, \mathtt{B}_1, \mathtt{C}_1), (\mathtt{A}_1, \mathtt{B}_0, \mathtt{C}_1)$ or $(\mathtt{A}_1, \mathtt{B}_1, \mathtt{C}_0)$. Note that this is equivalent to the quadratic algorithm, because doing this recursively results in a time complexity governed by $T(n) = 4T(n/2)$, which yields $T(n) = \mathcal{O}\left(n^2\right)$.

Doing this divide-and-conquer step k times recursively splits each input array into 2^k smaller arrays and splits the input problem into 2^{2k} independent subproblems. This makes computing the 3XOR triplets embarrassingly parallel, which is exactly what we wanted. We chose $k = 15$ according to the hardware specifications of the compute nodes that we have access to. This resulted in a billion independent subproblems in which each input array has expected size 131,072.

Vector Length. Clamping and parallelizing yields instances of the 3XOR problem with 80-bit hashes. This is not very practical because most computers have registers whose size is a power of two. Therefore, we adopted the following strategy: we compute all the 2^{17} triplets that are 3XOR on the first 64 bits. Then, amongst all these, we expect to find one which is in fact 3XOR on the whole 80 bits.

As such, the actual 3XOR subproblems we need to solve are the following: three input arrays of size 2^{17} each containing 64-bit random values; the expected number of 3XOR triplets in each instance is 2^{-13} and they all have to be found. Of course, the most likely outcome is that a single subproblem does not contain any 3XOR triplet on 64 bits.

Solving the Independent Subproblems. To solve a subproblem confined in a single compute node, we essentially have the choice between the quadratic algorithm and the generalized Joux algorithm. Asymptotically, the latter should be more efficient ; its designers found it to be roughly 3 times faster than a well-optimized implementation of the quadratic algorithm. It is the algorithm we used in practice. We recall it in fig. 2.

A side advantage of only solving small subproblems is that it increases the advantage of algorithm G compared to the quadratic algorithm: the number of iterations it has to do is reduced when A, B are small compared to n. If the subproblems become too small, on the other hand, the "administrative overhead" is going to dominate.

The main algorithmic trick of algorithm G, which is due to Joux [10], consists in doing a linear change of variable (in step G3) which in turn reduces finding all the 3XOR triplets in $\mathtt{A} \times B \times \mathtt{C}_i$ to a (linear) join computation, done by algorithm J. To reduce the number of iterations in algorithm G, the partitioning step should produce the smallest possible number of partitions of $\mathtt{C}$, along with the corresponding changes of variables.

In theory, the optimal value of k is $\lceil \log_2 \min(A, B) \rceil$, which in our case should be 17. In practice we used $k = 19$: this results in (slightly) more iterations, but it makes algorithm J faster and easier to implement (it divides by 4 the number of times step J6 is executed).

The value of ℓ used in algorithm J should be tuned to the underlying hardware. We used $\ell = 10$. We used cuckoo hashing as a static hash table to hold $\mathtt{C}$ and linear probing for H. The matrix-matrix product is done using the "four Russians" trick.

Algorithm G (*3XOR by linear changes of variables*). Given $\mathtt{A}, \mathtt{B}$ and $\mathtt{C}$, return all 3XOR triplets $(x, y, z) \in \mathtt{A} \times \mathtt{B} \times \mathtt{C}$. Let $0 < k < n$ be a parameter of the algorithm.

G1. [Partition $\mathtt{C}$.] Using algorithm P (below), partition $\mathtt{C}$ into $\mathtt{C}_1, \ldots, \mathtt{C}_m$ with associated matrices $M_1, \ldots, M_m$ such that:

$$\mathtt{C}_i M_i = \underbrace{\begin{pmatrix} 0 \ldots 0 & \star \ldots \star \\ \vdots \quad\quad \vdots & \vdots \quad\quad \vdots \\ 0 \ldots 0 & \star \ldots \star \end{pmatrix}}_{\substack{\longleftarrow k \longrightarrow \\ k}} \quad \underbrace{\qquad\qquad}_{\substack{\longleftarrow n-k \longrightarrow \\ n-k}} \quad \Big\updownarrow \geq n-k$$

G2. [Main loop.] For all $1 \leq i \leq m$, perform steps G3–G4 then stop.

G3. [Matrix product.] Set $\mathtt{A}' \leftarrow \mathtt{A}M_i$, $\mathtt{B}' \leftarrow \mathtt{B}M_i$ and $\mathtt{C}' \leftarrow \mathtt{C}_i M_i$.

G4. [Join] Use algorithm J (below) to find all 3XOR triplets (x', y', z') in $\mathtt{A}' \times \mathtt{B}' \times \mathtt{C}'$ such that $x'[0..k] = y'[0..k]$. For each such triplet, emit $\left(x'M_i^{-1}, y'M_i^{-1}, z'M_i^{-1}\right)$, which is a solution of the original problem. ∎

Algorithm P (*Partitioning by simultaneous linear approximations*). Partition $\mathtt{C}$ into slices $\mathtt{C}_1, \ldots, \mathtt{C}_m$. In each slice, all vectors satisfy k simultaneous linear equations, given by matrices $M_1, \ldots, M_m$.

P1. [Setup.] Let $i \leftarrow 0$.

P2. [All finished?] If $\mathtt{C}$ contains less than $n-k$ elements, then do: $\mathtt{C}_i \leftarrow \mathtt{C}$; find M_i by solving a linear system; terminate the algorithm.

P3. [Start new slice.] Initialize $j \leftarrow 0, S \leftarrow \mathtt{C}$ and let M_i be the zero matrix.

P4. [Decoding.] Find a low-weight linear combination of the columns of T using an Information Set Decoding (ISD) algorithm ($y = \sum_{k=1}^{n} x_k S_k^t$, where y has low hamming weight).

P5. [Update Slice.] Remove from T all rows where y is non-zero. Store $x_1, \ldots, x_n$ in the j-th column of M_j.

P6. [Finalize slice?] If $j < k$, return to step P4. Otherwise, pad M_j with linearly independent columns; $C_i \leftarrow T$; $\mathtt{C} \leftarrow \mathtt{C} - T$; increment i and go back to step P2. ∎

Algorithm J (*3XOR with special $\mathtt{C}$ using a partitioned hash join*). Given three arrays $\mathtt{A}, \mathtt{B}, \mathtt{C}$, where all $z \in \mathtt{C}$ are such that $z[0..k] = 0$, returns all 3XOR triplets in $\mathtt{A} \times \mathtt{B} \times \mathtt{C}$. The algorithm is given k and also another parameter $0 \leq \ell < k$.

J1. [Setup.] Initialize a (static) hash table with the content of $\mathtt{C}$.

J2. [Partition input.] Partition $\mathtt{A}$ and $\mathtt{B}$ according to their first ℓ bits into $\mathtt{A}_0, \ldots, \mathtt{A}_{2^\ell - 1}$ and $\mathtt{B}_0, \ldots, \mathtt{B}_{2^\ell - 1}$ ($\mathtt{A}_i$ contains all $x \in \mathtt{A}$ such that $x[0..\ell] = i$).

J3. [Loop on partitions.] For all $0 \leq i < 2^\ell$, do steps J4–J6 then stop.

J4. [Build H.] Let H be an empty hash table. For all $x \in \mathtt{A}_i$, append $x[k..n]$ to $H[x[\ell..k]]$.

J5. [Probe H.] For all $y \in \mathtt{B}_i$, do: probe $y[\ell..k]$ in H and retrieve all associated x's.

J6. [Probe $\mathtt{C}$.] For each matching x, do: $z \leftarrow x \oplus y$; if $z \in \mathtt{C}$, then emit (x, y, z). ∎

Fig. 2. Algorithm used to solve the actual 3XOR sub-problems.

Slicing $\mathtt{C}$**.** In each partition $\mathtt{C}_i$, all vectors satisfy k simultaneous linear equations (given by the first k columns of M_i). Finding such a partition of $\mathtt{C}$ is done greedily: the largest possible $\mathtt{C}_1$ is found, then the algorithm tries to partition $\mathtt{C} - \mathtt{C}_1$, and so on.

Finding such a slice of $\mathtt{C}$ is also done greedily: a single equations satisfied by as many vectors of $\mathtt{C}$ as possible is found (this is a decoding problem), and the process is iterated on vectors satisfying this equation.

We used the Lee-Brickell [16] algorithm to find low-weight codewords. It is not the most efficient decoding algorithm, but it had the advantage of being relatively easy to implement. It is an iterative procedure in which each iteration has some chance of finding low-weights linear combinations. We could precisely control the running time of the procedure by choosing the number of iterations. This does not result in the best possible result — but decoding up to the Gilbert-Varshamov bound is computationally intractable given the problem sizes.

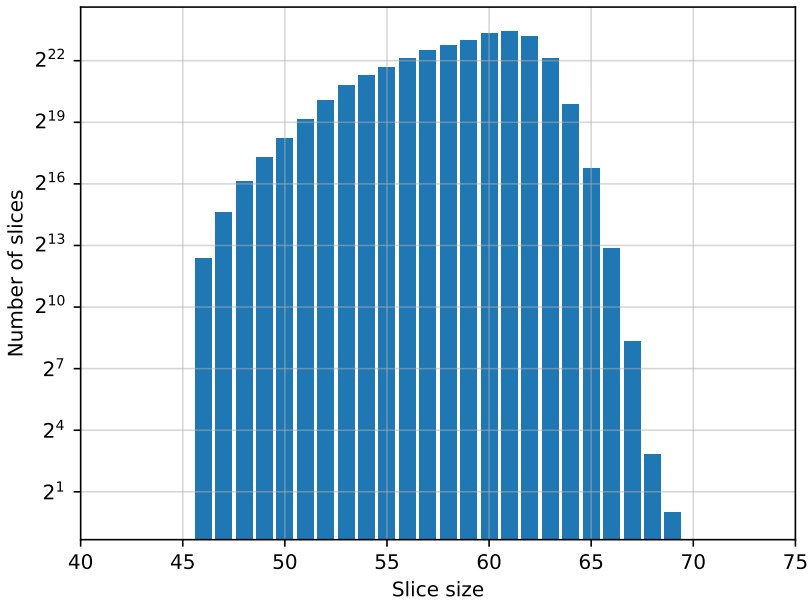

Fig. 3. Slice sizes after running algorithm P for 4 hours on each subproblem.

Fig. 3 shows the distribution of the sizes of partitions resulting from running algorithm P for 4 hours on each subproblem. The average size of slices obtained this way was 59. If only simple linear algebra was used to compute the change-of-basis matrices, then all slices would have had size 45. Using algorithm P instead

of a more naive procedure therefore results in a $\approx 30\%$ reduction in the number of iterations of algorithm G (and therefore on the time needed to solve the 3XOR problem). Less than 3.5% of the total running time was spent in algorithm P.

We refer to [8] for more algorithmic details.

3 "Dedicated" Hardware Accelerators

As mentioned in section 2, we chose to compute a 3XOR on 128 bits by accumulating three lists of 2^{32} bit strings each, whose hash starts with 32 zero bits. This presumably requires 3×2^{64} evaluations of SHA-256. The speed benchmark of OpenSSL tells us that an average current CPU core (an Intel Core i7 6600U at 2.6Ghz) can do about 5 millions SHA-256 per second. Thus, building the lists would take $\approx$ 350,000 CPU years on a single core. This was nevertheless accomplished in 7 calendar months using two inexpensive (and aging) bitcoin mining devices.

3.1 The SHA-256 Hash Function

The SHA-256 hash function is fully specified in [22]. We only describe the aspects relevant to our work. In particular, we do not describe the compression function. SHA-256 follows the Merkle-Damgård construction: the message is split into 512-bit blocks that are hashed interatively. If the last block is incomplete, it is padded to a multiple of 512 bits. The final hash value is obtained by iterating the compression function: with $h_{-1} = \texttt{IV}$ (given by the specification), let $h_i = F(h_{i-1}, m_i)$, where F is the compression function, m_i is the i-th message block and h_i is the i-th intermediate chaining value. The last chaining value is the hash.

In particular, evaluating SHA-256 over an 80-byte message requires two invocations of the compression function; evaluating SHA-256 over a 32-byte message requires a single compression function call.

3.2 Generalities on Bitcoin Mining

Without going too much into details (interested readers are referred to the original bitcoin paper [19]), the bitcoin network maintain a public secure ledger (the *blockchain*) that records all transactions: this allows the network to check the validity of transactions and prevent double-spending.

Bitcoin Puzzle. This relies on a proof-of-work system similar to HashCash [5]: to incorporate pending transactions to the blockchain, *miners* must solve the current bitcoin puzzle. This essentially amounts to finding a partial preimage on the SHA-256d hash function, which is defined as $\mathsf{SHA\text{-}256d}(x) = \mathsf{SHA\text{-}256}(\mathsf{SHA\text{-}256}(x))$. More precisely, to extend the blockchain, miners must produce a valid 80-byte *transaction header block*, depicted in figure 4.

Byte range	Length (bits)	Content
0:4	32	Version
4:36	256	hash of the previous block header
36:68	256	hash of the transactions in this block
68:72	32	current network time
72:76	32	(encoded) current network difficulty
76:80	32	arbitrary nonce

Fig. 4. Format of a bitcoin transaction header block.

A block is *valid* if its hash by SHA-256d, seen as an integer, is less than $2^{224}/D$, where D is the current network difficulty. Essentially, this forces the hash of the block to begin with many zero bits. Solving the bitcoin puzzle is done by brute force. Doing it naively would require three invocations of the compression function of SHA-256 per trial, but this can be reduced to two. Indeed, let `header` = `payload` $\|$ `nonce`. Recall that the goal consists in finding the `nonce` (if it exists) that solves the bitcoin puzzle, given the `payload`. We find that

$$\begin{aligned}
\mathsf{SHA\text{-}256d}(\texttt{header}) &= \mathsf{SHA\text{-}256}(\mathsf{SHA\text{-}256}(\texttt{header})) \\
&= F(\texttt{IV}, \mathsf{SHA\text{-}256}(\texttt{header}) \parallel \mathit{padding}) \\
\mathsf{SHA\text{-}256}(\texttt{header}) &= F(\texttt{midstate}, \texttt{payload[64:76]} \parallel \texttt{nonce} \parallel \mathit{padding}) \\
\texttt{midstate} &= F(\texttt{IV}, \texttt{payload[0:64]})
\end{aligned}$$

The point is that the `midstate` is independent from the `nonce`: for each `nonce`, computing SHA-256d(`header`) requires only two compression function invocations (the `midstate` does not have to be recomputed).

A a new bitcoin header block is found every 10 minutes or so, and the current network difficulty is adjusted every 2016 blocks (two weeks) to enforce this rule. On September 21st, 2019, $D = 2^{43.4}$, so that finding a new valid block by brute force presently requires $2^{75.4}$ evaluation of SHA-256d. This roughly happens every 10 minutes, so all the miners in the bitcoin network perform 85 exa-hashes per second ($2^{66.2}$ / s) at this time.

The transactions are hashed using a Merkle hash tree, with a single application of SHA-256d at each node. Miners are incentivized: the first transaction of each block is a *coinbase transaction*. It has no origin (it creates currency) and the miner is free to direct to its own bitcoin wallet.

To find a valid block, miners are free to choose the 32-bit nonce at the end, and they can choose arbitrarily an uninterpreted value in the coinbase transaction.

Mining Pools. Miners are essentially playing a game of chance: they invest resources (equipment, electricity, ...); if and only if they succeed in extending the

blockchain, they gain newly minted bitcoins and transaction fees. Their chance of success is proportional to their computing power.

To reduce risk, miners can join forces and form a *mining pool*. In a pool, miners are remunerated much more regularly (typically, daily) and proportionally to their hashrate. Concretely, this means that in a pool, when a miner solves the current bitcoin puzzle, it reports the solution to the pool manager; the pool manager uses it to extend the blockchain, and cashes in the reward. Pool operators take the risk of having to remunerate the miners participating in the pool regularly, while they themselves only receive the benefits of extending the blockchain irregularly (they typically perceive a fee). Pooled mining requires a communication protocol between a pool server that dispatches work, and the miners who actually perform it.

Miners receive work and submit *shares*. A share is a solution to the current bitcoin puzzle, which is valid for a lower difficulty than the network difficulty. Each share stands some chance of being a full solution to the current bitcoin puzzle. Shares with a higher difficulty stand more chance. By sending shares to the pool manager, the miners simultaneously prove that they are working on the problem, and potentially reveal its solution to the pool manager. Miners are typically remunerated a fixed amount per share of a given difficulty. If a miner finds a solution to the current bitcoin puzzle, she cannot claim the reward for herself, because the pool server does not reveal enough information (in particular, it does not reveal the pending transactions).

In early protocols, a "job" would consist of the 76-byte of a transaction block header, and the miners would return a valid 32-bit nonce that complete the block. As the hashing power of miners increased, this system ceased to be practical.

The Stratum Protocol. The most widely used pool mining protocol at this time seems to be the *stratum pool mining protocol* [2], and it is designed to work around this issue. The idea is to let miners choose freely the uninterpreted part of the coinbase transaction.

To let miners choose freely one specific transactions without revealing them all the pending transactions included in the block, the stratum protocol uses partially-evaluated merkle trees (cf. fig. 5). A job sent by the pool server to the miners is composed of an incomplete transaction header block in which both the nonce and the hash of all transactions is missing. In addition, a *coinbase prefix* and a *coinbase suffix* are given, as well as the so-called "Merkle hash roots", which are values $h_1, h_2, h_3, \ldots$ in fig. 5.

The miner may choose the *hole* in the coinbase transaction arbitrarily. Once a value is chosen, the hash of all transactions can be computed quickly by finishing the evaluation of the Merkle tree. The miner then has to find a nonce valid at the expected difficulty. If no such value exists, a new value of the *hole* can be tried. A share is composed of the value of the *hole* in the coinbase transaction and the nonce.

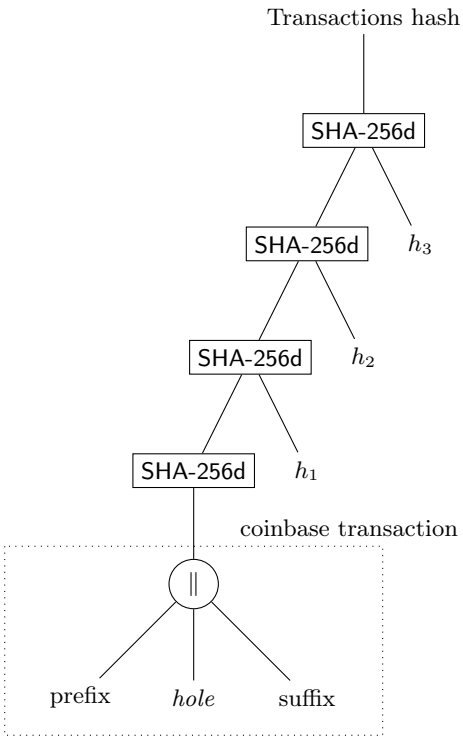

Fig. 5. Partially-evaluated Merkle trees to assemble the hash of all transactions without knowing them, while choosing a part of the coinbase transaction.

3.3 Bitcoin Mining Rigs

At the time of this writing, commercial high-end bitcoin mining devices implement the stratum protocol. They present themselves as standalone devices, which only require a power cable and Ethernet connectivity. They have some kind of configuration interface in which the URL of a stratum pool server can be specified (as well as credentials on said server). Operations are mostly automatic and require almost no supervision.

In June 2017, we bought a second-hand bitcoin mining device, an Antminer S7 from Bitmain, which completely fits the above description (see Fig. 6). It contains three boards (see Fig. 8), each holding 45 custom "BM1385" 28nm ASICs clocked at 600MHz. The whole device is capable of doing 4.7×10^9 evaluations of SHA-256d per second (meaning $2^{33.13}$ SHA-256 compression function evaluation per second). It was no longer top-notch when we bought it, and it presently stands absolutely no chance of potentially turning in a profit in bitcoin mining.

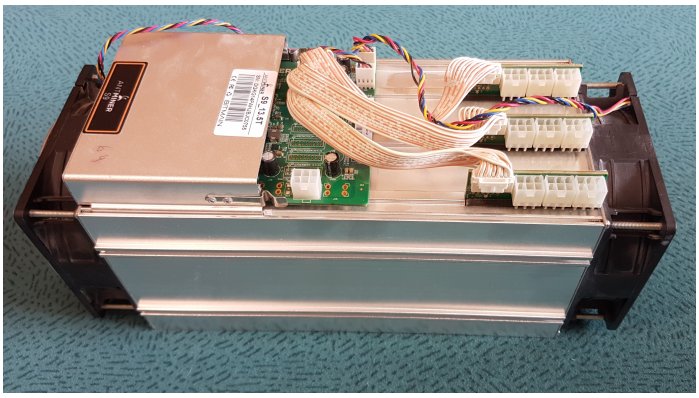

Fig. 6. An AntMiner S7 from Bitmain. The device is made of a small ARM computer (visible on top), and contains three "hashboards" inside an aluminium casing.

We gather from their very partial specification [1] that the ASICs in each board seem to be linked in a chain. They communicate with their controller using a standard serial protocol. We gather that the ASICs take as input: the chaining value after the first invocation of the compression function (the "midstate"), bytes 64–76 of the block and a zero-bit count. They likely return nonces such that SHA-256d($block \parallel nonce$) starts with the prescribed number of zero bits. Thus, trying a new nonce requires two compression function evaluations.

The AntMiner S7 also contains a small ARM cortex-A8 CPU with 512MB of RAM running Linux. It hosts the web interface and the mining program. There is also an FPGA which is presumably an interface between the mining program and the mining ASICs. We measured a power consumption of 1450W at the plug.

Fig. 7. An AntMiner S7 from Bitmain (side view without the fan). The three hashboards are visible.

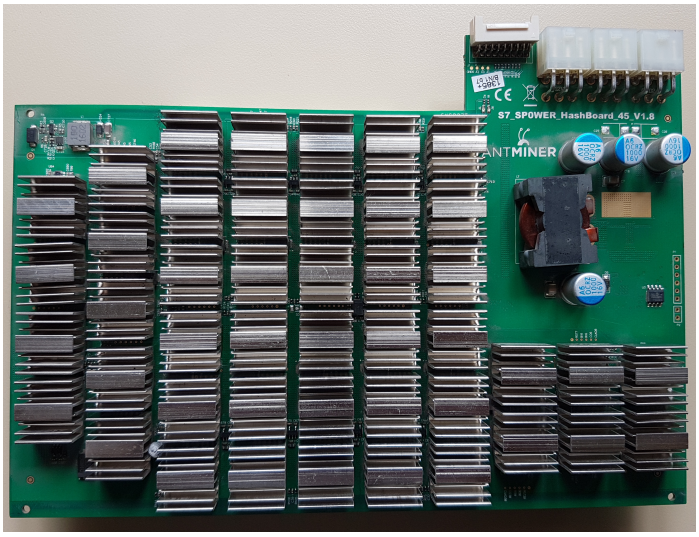

Fig. 8. A "Hashboard" containing 45 custom ASICs. The mining device contains three of those.

Given the hashing power of this mining rig, in the best case we could expect to find ≈ 1094 valid blocks per second. If this were true, three lists of size 2^{32} could potentially be assembled in only 136 days.

3.4 A Cryptanalytic Mining Pool

We first attempted to put the computing power of this bitcoin mining rig to good use by creating a custom mining pool. A custom pool server sends special work to miners; miners submit shares; each share yields an item in one of the three lists A, B or C, whose hash starts with at least 32 zero bits.

This had the obvious advantage that we could use any bitcoin mining device without modification, even remotely. In the stratum mining protocol, the pool server actually has a lot of control over the work done by the miners. Indeed, byte ranges [0:36] and [68:76] are directly given by the pool server. The hash of transactions and the nonce will unpredictably depends on the miner's implementation.

We thus implemented a functional stratum server in Python, using the twisted asynchronous network library. This required about 1000 lines of (verbose) code. The whole setup (server+miner) was fully functional. Our code is available here:

https://github.com/cbouilla/3sum-pool

Results were disappointing, as illustrated by fig. 9. The rate at which shares are produced is in principle a decreasing function of the difficulty. But other

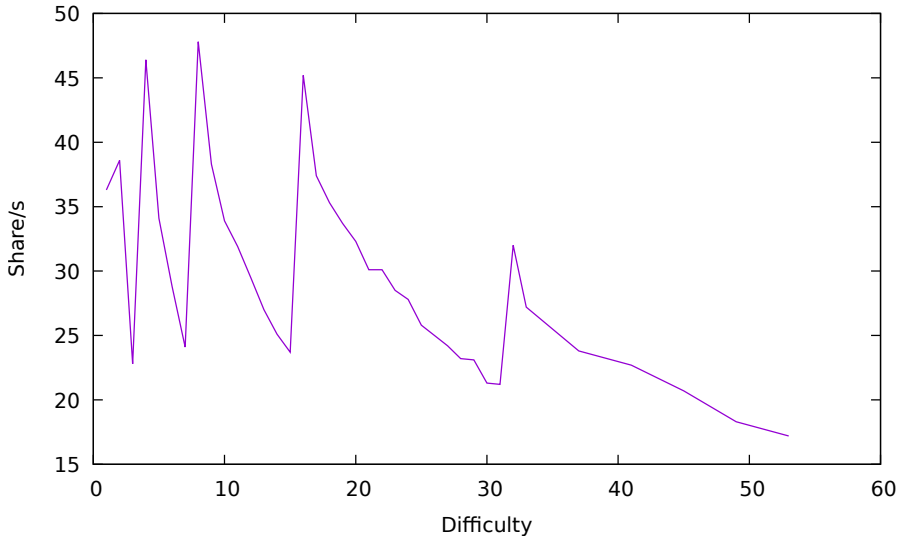

Fig. 9. Rate of share production according to the chosen difficulty on an AntMiner S7. It shows that some components of the miner are badly implemented, and performs poorly for non-power of two difficulties.

phenomenons were clearly at play (including an apparent implementation error for non-power of two difficulties...). All-in-all, the best we could obtain was roughly 45 share/s at difficulty 16, a far cry from the 1000+ share/s that we hoped for. In particular, shares are produced at a higher rate at difficulty 16 than at difficulty one, even though it requires 16 times more hashing work!

This lead us to speculate that the stratum protocol incurred a lot of overhead, and that the low-cost ARM CPU was not powerful enough to deal with it at high share rate. Indeed, we could check that the mining program controlling the ASICs had 100% CPU utilization at low difficulty.

We nevertheless started a first mining campaign in December 2017, mining at difficulty 16. A 116-bit 3XOR was computed in February 2018. While we initially hoped that we could use some funding to buy more mining devices and carry our 128-bit project using this simple solution, this turned out to be impossible, if only for administrative reasons. We eventually gave up on the process in May 2018. At that point, 780 million blocks had been mined, enough to obtain a 3XOR on 119–120 bits.

3.5 Digging Deeper

In order to circumvent the bottleneck that the stratum protocol imposed upon us, we tried to examine the inner workings of the mining device. The SSH access that it exposes with the usual `root / admin` credentials turned out to be very useful.

We found out that it runs a well-known mining program, `cgminer`, whose source code (in C) is available at:

`https://github.com/bitmaintech/cgminer`

The code running on the S7 corresponds to the `bitmain_fan-ctrl` git branch. This code could be cross-compiled for ARM devices and the resulting binary could be copied to the device and run. We were therefore capable of altering the control software of the mining operations. Subsequent mining devices such as the S9 use a different program and work differently.

After reading its code, we realized that the original `cgminer` program works in a somewhat modular way, centered around a concurrent (blocking) *work queue* of finite capacity. A work item in this queue is essentially composed of a 76-byte transaction header block missing its nonce.

- The "stratum component" interacts with the pool server: it receives updates about the current bitcoin puzzle which changes every 10 minutes (it then flushes the work queue) and is responsible for sending back shares to the pool.
- The program runs into an infinite loop that generates new work items for the current bitcoin puzzle and pushes them into the work queue.
- Each active "device component" pops work items from the work queue and feed them to the mining hardware it controls. A callback from the stratum component is invoked for each solution found.

Of course, each component has several threads, several internal queues (both in software and hardware for the device), etc.

We essentially scrapped the stratum component and replaced both its work generator and networking code with our own. We used a simple protocol: the miner connects to a server; the server answers with a prefix (`FOO`, `BAR` or `FOOBAR`) and a 64-bit counter value. Instead of receiving stratum work and generating the corresponding blocks as described above, the miner generates simple blocks by assembling the prefix and the counter value in hexadecimal. The counter is incremented for each new block. When a valid nonce is found, the pair (counter, nonce) is reported back to the server. We tried to make the code path as direct as possible (for instance we removed all correctness checks in all components).

To deal with networking, we initially used the ∅MQ library [9]. It is a C++ library; cross-compiling it for the embedded Linux on the mining device was complicated, because it had an old `libc`. Compiling it *on* the embedded Linux was not possible because it does not have enough RAM. In any case, we ditched ∅MQ because it did not handle network disconnects gracefully (even though it is supposed to be possible, we have not been able to make it re-establish the connection automatically). We instead used the nanomsg [26] library, which offers comparable features (and is written in plain C). It also works around networks problem more gracefully.

The corresponding server is a 220-line C program that mostly stores the (counter, nonce) pairs it receives in a file, and keep the current value of the counter up to date.

Using this setup, we were able to collect about 550 share/s at difficulty 2. We could not "mine" at difficulty 1 to obtain the 1000+ share/s we hoped for; this was crashing the hardware for some reason. We ran a second mining campaign between October 15th, 2018 and May 15th, 2019. This resulted in three lists of 2^{32} blocks whose hash begin with 33 zero bits. Using a single AntMiner S7, the process would have taken 10 full months, but on 15th February, 2019 we put a second one in production to speed things up. We bought it on eBay for 80€.

Accumulating the "clamped" hashes finished on May 16th: three lists of 2^{32} (counter, nonce) pairs had been accumulated. This was enough to expect two solutions on 128 bits (and made us confident that at least one would be found). In total, $2^{67.6}$ (this is $2 \times 3 \times 2^{32} \times 2^{33}$) compression function evaluations have been performed by these two mining devices.

4 Computing the 3XOR

This section details how we have managed the computation of the solution to the large instance of the 3XOR problem accumulated by our bitcoin mining devices.

4.1 Hardware

The computationally expensive parts of the whole computation have been run on a massively parallel (and aging) IBM BlueGene/Q computer, named `turing` and located at IDRIS ("Institut du développement et des ressources en informatique scientifique"), a french national computation center. The machine had been in production since 2012 and it has been has been decommissioned on October 1st, 2019. We had been granted 10 million CPU hours. The whole computation took 5 million CPU hours in May 2019.

This machine is a "small" installation, made of only 6 racks, each with 32 boards, each with 32 nodes. The nodes are connected by a very fast custom 5D torus network. Each node contains a special 64-bit PowerPC A2 processor running at 1.6Ghz and 16GB of RAM. The processor has 18 cores, but only 16 are available for applications (one is dedicated to the OS, one is a spare). The cores run a lightweight variant of Linux with only partial support for the POSIX specification (for instance, `fork` is not supported). The programs may use the usual POSIX functions to read and write files. They may also communicate using MPI functions (the MPI environment is the only way to run programs).

The processors do not support paging nor virtual memory: all memory addresses are physical. An advantage is that TLB faults and page walks cannot happen. The CPU cores are *in-order* and support 4 hardware threads: a core can execute at most one instruction per CPU cycle, taken amongst the four threads. Therefore, actually executing one Instruction per Cycle (IPC= 1) may potentially be challenging.

Each core has 16Kb of L1 cache (256 cache lines of 64 bytes), shared between the threads. The L1 cache is *write-through* (writes do not cause cache miss) and has a 4-cycle latency (in case of a cache hit), therefore using more than one

hardware thread may help keeping the CPU busy. It is in fact required to reach near-peak performance. A 32MB L2 cache (with an 80-cycle latency) is shared between the 16 cores. It was therefore critical to exploit the L1 cache as much as possible.

These CPUs are simple and relatively slow, which is to some extent an advantage: it is quite difficult to saturate the memory bandwidth, for instance (which is roughly 25GB/s). We used up to 4096 nodes (65536 cores) simultaneously. The specificities of this platform dictated some algorithmic choices and tuning, as discussed below.

The only supported programming languages are C and Fortran, via IBM's `xlc` compilers. We used C... We tried to keep the programs running on this machine as simple as possible and avoided relying on external libraries (we initially used M4RI [3] to do linear algebra over $\mathbb{F}_2$ but eventually recoded several functions because it was simpler). Because we had to write multi-thread code, we used OpenMP.

In order to make sure that everything was going according to plan, we tried our code on instances of increasing size, and checked that the results were consistent with our expectations.

We ran algorithm P on the $k = 32768$ independent parts of `C` using 32768 cores simultaneously for 4 hours. The corresponding self-contained C program is 750 lines long. We then ran algorithm G using four 20-hours jobs on 65536 cores. The corresponding self-contained C program is 1100 lines long, with only a few dozens concentrating all the performance issues.

4.2 Preprocessing and Data Management

The mining server program generates *preimage files* composed of 12-byte (nonce, counter) pairs. At 550 share/s, this means roughly 550 Mbyte per day of fresh data. Every few days, we restarted the server so that a new preimage file would be created.

The preimage files must be checked (i.e. the corresponding hashes must actually have 33 zero bits). The bitcoin miners occasionally produced incorrect results (one of the bitcoin mining device produced a proportion $\approx 10^{-8}$ of wrong results, while the second one errs more frequently with a proportion $\approx 10^{-6}$ of incorrect results).

Each preimage is associated to a 256-bit *full hash* (by applying SHA-256d), in which bits [0:33] are zero. From this full hash, we extract a subset of 64 (uniformly distributed) bits (bits [33:97]) that we call the *hash*. Furthermore, we extract a k-bit *partitioning key* from the full hash by taking bits [97:97+k].

For various reasons, we enforce that the 64-bit hashes are unique in `A`, `B` and `C`. For each list, we actually maintain a dictionary `hash` $\rightarrow$ `preimage`. A *dictionnary file* is simply a list of pair. Because the 3XOR code actually produces the three hashes that XOR to zero, the dictionaries are useful to retrieve the corresponding preimages.

1. When a new preimage file is ready, we split it into 2^k *sub-dictionaries* using the partitioning key. Potential duplicate preimages are now confined into a single sub-dictionary.
2. We sort the sub-dictionaries by hash. This is easily done, since they are quite small and fit in RAM.
3. For each partitioning key, we perform a multi-way merge on all sub-dictionaries. Duplicate hashes are detected and removed (they were mostly the results of "human errors"). We write down the unique hashes in order into a single *hash file* per partitioning key.
4. For the `C` list, we ran algorithm P (cf. fig. 2) on all the 2^k parts, which results in as many *slice files*. A slice file contains a sequence of slices of `C`, along with the corresponding M matrices and their inverse (therefore the actual 3XOR code does not have to compute the inverses).

All these tasks were accomplished by a collection of small (and inasmuch as possible, simple) `C` programs to split, sort, merge all these files. More importantly, we tried to have another collection of programs to check the correctness of each step. All-in-all, this represent ≈ 2300 additional lines of `C` code.

4.3 Tuning the Algorithms to the Underlying Hardware

From a performance point of view, we quickly identified three critical sections in algorithm G: matrix multiplication (step G3), partitioning (step J2) and subjoins (steps J4–J6). These represent 39%, 18% and 41% of the whole running time, respectively.

Matrix Multiplication. Recall that we have to multiply a $m \times 64$ matrix (with large m) by a 64×64 matrix M. We use a variant of the "four Russians trick". Given M, we precompute 8 tables of 256 entries, after which the vector-matrix product $x \cdot M$ can be computed as:

```
static inline u64 gemv(u64 x, const struct matmul_table_t *M)
{
        u64 r = 0;
        r ^= M->tables[0][x & 0x00ff];
        r ^= M->tables[1][(x >> 8) & 0x00ff];
        r ^= M->tables[2][(x >> 16) & 0x00ff];
        r ^= M->tables[3][(x >> 24) & 0x00ff];
        r ^= M->tables[4][(x >> 32) & 0x00ff];
        r ^= M->tables[5][(x >> 40) & 0x00ff];
        r ^= M->tables[6][(x >> 48) & 0x00ff];
        r ^= M->tables[7][(x >> 56) & 0x00ff];
        return r;
}
```

The tables fit exactly into the available 16KB of L1 cache. Each access to the tables entails a latency of 4 cycles, which can only be partially overlapped by

other instructions. Running this function using a single hardware thread results in only 0.5 instructions per cycle. Fortunately, using more hardware threads allows to overcome the latency of the L1 cache without much programming effort. With 2 threads, we obtain IPC= 0.8, and with the whole 4 threads we obtain IPC=0.95. The matrxi multiplication step is thus CPU-bound.

Doing the vector-matrix product requires 34 cycles, so the throughput is 46.6 millions products per second. Amongst these 34 cycles, 28 instructions seem mandatory: 8 loads, 8 XORs, 8 combined rotations/mask (`rlwinm`), 4 additional shifts (`rldicl`), so this is not bad.

Partitioning. This operations essentially consists in moving data around in memory, and it is memory-bound. This is essentially equivalent as doing one pass of radix-sort / bucket-sort. It runs the following code:

```
for (u32 i = 0; i < n; i++) {
        u64 x = L[i];
        u64 h = x >> (64 - l);
        u32 idx = count[h]++;
        scratch[idx] = x;
}
```

Before this loop runs, the `count` array is correctly initialized. To avoid using an additional counting pass to count the size of each partition beforehand, we use Chernoff bounds to over-allocate individual partitions (the input lists are hashes, so they are fairly "random").

The hardware characteristics of the BlueGene/Q processor are actually quite helpful here: a) because the cache is write-through, writing in `scratch` does not pollute it and b) because there is no paging, there are no TLB faults. TLB faults are usually a problem in this kind of code in databases servers (see for instance [6]) and special techniques are deployed to avoid them, such as software write-combining buffers. Here, this is simply not necessary.

The cache will only contain (useless) chunks of `L` and the `count` array that requires $2^{\ell+2}$ bytes. If ℓ is too large, then the `count` array will no longer fit in cache. In addition, to amortize the 4-cycle latencies of reading `L` and `count`, we need to run at least two hardware threads, each with its own `count` array. Indeed, the threads write in distinct locations inside the `scratch` array: this way we avoid the need to synchronize them, which would be expensive.

A few experiments led us to use two hardware threads per core with $\ell = 10$. On the whole processor (16 cores), this reaches a memory bandwidth of 23.5GB/s, which is very close to the maximum (this makes 16 cycles per item, with IPC=0.8 instructions per cycle, or nearly 100 million items dispatched per second per core). Again, this is close to the peak performance of the hardware.

Using more threads would be detrimental: since memory bandwidth is the problem, it would likely not improve anything; in addition the threads would be competing for the L1 cache. The two `count` occupy half of the L1 cache; because the cache uses a LRU replacement strategy, when a cache line has to be evicted

to make room for the next `L[i]`, there is little chance that a portion of `count` will be evicted.

It follows that partitions of `A` and `B` in algorithm J are 1024 times smaller than the input subproblems. By choosing the size of the input subproblems, we can tune the expected size of partitions in algorithm J.

Subjoins. This involves building and probing hash tables containing 64-bit integers. We considered cuckoo hash tables and "classic" linear probing: we found that probing a cuckoo table is about 67% faster than linear probing, when the fill ratio is 40%. However, inserting a new value into a cuckoo table is slower and more complicated (it may fail). Therefore we used linear probing for the H table in steps J4–J5. The main parameter affecting performance is the fill ratio, which depends on partition size and on the number of concurrent hardware threads we want to run.

We settled for 3 hardware threads, each with a 4KB table (512 entries of 64 bits), which leaves some leg room in the L1 cache for fetching inputs. This means that we have to extract a 9-bit hash from x to insert x in a table. This mandates that $k = \ell + 9$, hence $k = 19$, which is the value we used.

Because the input subproblems have size 2^{17}, this means that the hash tables are only 25% full (smaller fill ratios are expected to result in faster probes/insertions). We verified experimentally that input problems of size 2^{16} and 2^{18} do not give better results.

In steps J5–J6, two probes to two different hash tables are chained. The size of `C` is quite small, but L1 cache is very scarce. As a result, it turned out to be faster to "materialize" the output of the join in memory, instead of probing `C` on the fly. In other terms, it is faster to write all the x resulting from step J5 to memory instead of doing J6 right away. Then, step J6 can be done for all elements of this array, for all partitions $\mathtt{C}_i$ in a second time. This results in a 15% speedup.

All-in-all, building *and* probing H requires about 32 cycles per item inserted and probed with IPC=0.75 instructions per cycle (47.7 millions items/s). Then, probing `C` runs at ≈ 20 cycles per item probed; because the number of concerned items is smaller, it requires $10\times$ less wall-clock time.

Comparison with the Quadratic Algorithm. In average, the whole computations runs at 0.85 instructions per cycle, which is close to optimal. When the code runs on 65536 cores at 1.6Ghz, this make a not-very-impressing "90 TFLOPS" (except that these were not floating point but only integers instructions). After all, this is aging hardware. All-in-all, the computation required about $2^{64.2}$ CPU cycles in total.

The tuned implementation of algorithm G "solves" a subproblem with input size 2^{17} in about 33.5s on a single PowerPC A2 core. For the sake of comparison, we ported the well-optimized implementation of the quadratic algorithm used in [8] to the BlueGene/Q and ran it on a few subproblems. With a bit of tuning

and using 4 hardware threads, we obtain a performance of 178s per subproblem (there are 2^{34} pairs to process, and each pair requires 16 cycles).

The optimized implementation of algorithm G is thus found to be 5.3× faster than the quadratic algorithm on the BlueGene/Q — in fact, the quadratic algorithm benefits from vectorized instructions available on more conventional CPUs but not on this particular PowerPC.

4.4 Managing the Computation

We had to organize the computation of one billion independent subproblems which each require their own input data (roughly 5MB) and take about 30s. We arranged tasks in a 2D grid: the "task" (u, v) consists in solving the subproblem $(\mathtt{A}_u, \mathtt{B}_v, \mathtt{C}_{u \oplus v})$. Because 30s is too short, we grouped these tasks into 2D *task groups* of 8×8 tasks. Each task group requires about 35 minutes of wall-clock time and 41.5MB of data. There is therefore a 2D grid of 4096×4096 tasks group to solve.

Each core loads the data required for a task group, finds all the solutions of all tasks in the task group and communicate the results to a master processor (using MPI functions) which writes it in a file. Therefore, if anything fails, we lose 30 minutes of computation in the worst case. All task groups on the same row/column/"diagonal" require the same portions of A/B/C; we therefore avoided loading the same data multiple time: in each row/column/diagonal only one processor loads the data and broadcasts it to the other ones (again using MPI functions).

It must be noted that in this shared computing environment, we have to commit to a maximum wall-clock running time (of up to 20h), after which computing jobs are forcefully stopped. Therefore we were careful to voluntarily stop before the deadline. Our implementation of algorithm P explicitly manages the remaining running time. When running algorithm G, we calibrated the jobs to make sure that they would not extend past the deadline.

We used the biggest possible jobs allowed by the computing center (we found out that the bigger the jobs are, the higher their priority in the task scheduler...): 4096 nodes (65536 cores) for 20 hours. In each job, each core is affected 30 task groups. After three such jobs, a 128-bit 3XOR was found. We actually ran a fourth job. About 46.875% of the search space was explored, which is consistent with our expectation to have two potential solutions.

5 The Solutions

The main result of the computation is shown in fig. 1. It has a striking feature: while the hashes of x and y begin with 35 zero bits (as expected), the hash of z begins with 43 zero bits. This seems to be a coincidence. In any case, we have not been able to detect any significant bias, either in the input list or in the other solution triplets.

A total of 62,006 3XOR triplets on at least 112 bits have been found, versus an expected number of 61,440 after exploring 46.875% of the search space. As such, we feel confident that our code actually finds all solutions — at the very minimum, it does not "miss" a significant fraction thereof. The numbers of 3XOR triplets of each size are shown in table 2.

The most interesting 3XOR triplets are completely described by table 3. The complete list of all triplets can be obtained from the authors.

3XOR bits	# triplets found	# expected
112	31011	30720
113	15523	15360
114	7752	7680
115	3835	3840
116	1959	1920
117	974	960
118	478	480
119	244	240
120	118	120
121	50	60
122	34	30
123	14	15
124	7	7.5
125	2	3.75
126	3	1.88
127	1	0.9375
128	0	0.47
129	1	0.23

Table 2. Number of 3XOR triplets on k bits.

Cost and Energy. The whole computation (mining + solving) required $\approx$ 40MWh of energy, which is about 144GJ. This is enough energy to boil 430m^3 of water (a $25 \times 10 \times 1.72$ swimming pool) starting from 20°C. The bitcoin miners ate 10.8MWh over 7 months, while the BlueGene/Q consumed the remaining 30MWh over three days. Given the electricity rates at our institution, this makes roughly 3000€ of electricity. Given the electricity mix of the country where the computation took place, about 2 tons of CO_2 have been released into the atmosphere just because of us.

In addition, the computation center that granted us 10 million core hours on the IBM BlueGene/Q machine told us that this was worth 50 000€. In total, the cost of the attack can be estimated to $\approx 55,000$€ (including hardware and a few other expenses).

bits	FOO		BAR		FOOBAR	
	Counter	Nonce	Counter	Nonce	Counter	Nonce
129	0x000000003B1BD2039	dd3ff46f	0x00000000307238E22	a80da323	0x000000001BB6C4C9F	b01d7c21
127	0x000000001AB6DDDFF	7afc826e	0x0000000010ABABA5E	b6672ea4	0x000000000F8F3875C	3a0a14c6
126	0x0000000025250647B	8df9eed3	0x0000000031597C736	261e4a9d	0x000000003145B7B70	763631e3
	0x00000000381486C57	34b306a5	0x0000000011949F2D5	0bc08b4d	0x00000000325A69F32	283c42cd
	0x000000002553FC59C	9ca5b9d5	0x0000000019621F89B	88b9abab	0x0000000002FBEC230	dce8e58d
125	0x000000002C0EB67D9	6f8c288b	0x0000000004698FDA4	da53d324	0x000000001725E4711	d7f0b552
	0x00000000298CFE96D	d21c6e19	0x000000007A25D587B	472e9a07	0x000000003367F04FD	972896a1
124	0x000000004075FBBB7	527b4fd5	0x000000001E8172DE2	6a871455	0x000000003DE71816A	c66d621d
	0x000000003664BCFAE	c8de11aa	0x000000001B6BAB522	d7be4e48	0x000000001DF323250	d22bd184
	0x0000000004A83643B	dd7bf7c3	0x000000001ED54A5B8	88a9b853	0x0000000011E3CFD59	11cba187
	0x0000000019DA7AF6C	6ed71499	0x000000002B5CCB0C2	41b7f0f0	0x000000001D7BAB763	52740071
	0x00000000033EE88AA	5ab7d84e	0x0000000009CD025FE	394db949	0x00000000395231517	1ca5e9e5
	0x0000000023F38452B	c641562a	0x00000000035DBA65D	aac3f8f1	0x00000000052A54276	b76a496b
	0x000000002ABED77A3	cdb244c8	0x0000000002012ADB2D	c330a430	0x000000000D41BF9A0	2f58802f

Table 3. Most interesting 3XOR triplets. The "counter" is the string that appears at the beginning of a, b and c in fig. 1 (the letters have to be in uppercase). The "nonce" is the 32-bit value appended at the end of a, b and c.

Comparison with Other Practical Attacks. The 2017 practical collision attack on SHA-1 [25] required an effort equivalent to $2^{63.1}$ and compression function evaluations. It used GPUs to speed things up. The attack required about 6500 core years, plus an additional ≈ 100 high-end GPU years. Running the attack on the Amazon public cloud would have cost $560,000.

The 2016 computation of a 768-bit discrete logarithm [12] required between 4000 and 5300 years on a single core of a 2.2GHz Xeon E5 2660. This is estimated to be equivalent to $2^{60.6}$ SHA-1 calls. Before that, the 2009 factorization of a 768 bits RSA modulus [11] required about 2000 core years on a 2.2GHz AMD Opteron processor, with about 2^{67} instructions executed. This required about 500MWh of energy.

Evaluating the compression function of SHA-256 takes approximately twice as long as evaluating that of SHA-1, and we did evaluate it $2^{67.6}$ times. In total, we thus did about 45 times more "total work" than [25] and about 250 times more than [12].

In reality, we got away with it using a *much* smaller amount of computational resources, as well as *much* less money and energy, thanks to the efficient dedicated (and aging) ASICs available in bitcoin mining devices. The two other attacks discussed above [25, 12, 11] are definitely *much* more difficult to carry out in practice than the computation described in this paper.

Comparison with a Simple Collision Search. Finding a 128-bit collision on SHA-256 would require about 2^{64} compression function evaluations (less than we did), and could be done memory-less using the rho method. It can be parallelized

using the algorithm of van Oorshot and Wiener [27]. However, it is most likely *much* more difficult in practice than the computation described in this paper: we do not see how bitcoin miners could be used to accelerate it.

Summary. This computation is the result of a two-year effort. Now that we have reached the result we were targeting, we may ask: "*what have we gained by doing this?*". From a personal and professional point of view, the answer is mostly a clear sharpening of our algorithmic and programming skills.

From an algorithmic point of view, "*The Art of Computer Programming*" has been the single most useful reference for us (especially volume 3, "*Searching and Sorting*" [14]).

We occasionally used literate programming [13]: we found it helpful to write some of our C programs by slicing them down into small chunks and focusing on one single chunk at a time. We notably used this to write the "mining server" described in section 3.5 and all the "preprocessing" programs described in section 4.2. We found noweb [24] to be the friendliest literate programming tool available.

Lastly, trying to actually implement several 3XOR algorithms and run a large-scale computation reinforced our awareness of practicality issues: some nice algorithms had to be summarily dispatched because of their space requirements, for instance. We now have a clearer understanding of the kind of obstructions that may fail to make an attack "practical".

References

1. BM1385 datasheet v2.0. Available online. https://bits.media/images/asic-miner-antminer-s7/BM1385_Datasheet_v2.0.pdf.
2. Stratum pool mining protocol. Available online. https://slushpool.com/help/stratum-protocol.
3. Martin Albrecht and Gregory Bard. *The M4RI Library – Version 20121224*. The M4RI Team, 2012.
4. Elena Andreeva, Andrey Bogdanov, Atul Luykx, Bart Mennink, Elmar Tischhauser, and Kan Yasuda. Parallelizable and authenticated online ciphers. In Kazue Sako and Palash Sarkar, editors, *Advances in Cryptology - ASIACRYPT 2013*, pages 424–443, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.
5. Adam Back. Hashcash - a denial of service counter-measure, 2002.
6. Cagri Balkesen, Gustavo Alonso, Jens Teubner, and M. Tamer Özsu. Multi-core, main-memory joins: Sort vs. hash revisited. *PVLDB*, 7(1):85–96, 2013.
7. Daniel J Bernstein, Tanja Lange, Ruben Niederhagen, Christiane Peters, and Peter Schwabe. FSBday: Implementing Wagner's Generalized Birthday Attack. In *INDOCRYPT*, pages 18–38, 2009.
8. Charles Bouillaguet, Claire Delaplace, and Pierre-Alain Fouque. Revisiting and improving algorithms for the 3XOR problem. *IACR Trans. Symmetric Cryptol.*, 2018(1):254–276, 2018.
9. Pieter Hintjens. *ZeroMQ: messaging for many applications*. O'Reilly, Sebastopol, CA, 2013.
10. Antoine Joux. *Algorithmic cryptanalysis*. CRC Press, 2009.

11. Thorsten Kleinjung, Kazumaro Aoki, Jens Franke, Arjen K. Lenstra, Emmanuel Thomé, Joppe W. Bos, Pierrick Gaudry, Alexander Kruppa, Peter L. Montgomery, Dag Arne Osvik, Herman J. J. te Riele, Andrey Timofeev, and Paul Zimmermann. Factorization of a 768-bit RSA modulus. In Tal Rabin, editor, *Advances in Cryptology - CRYPTO 2010, 30th Annual Cryptology Conference, Santa Barbara, CA, USA, August 15-19, 2010. Proceedings*, volume 6223 of *Lecture Notes in Computer Science*, pages 333–350. Springer, 2010.
12. Thorsten Kleinjung, Claus Diem, Arjen K. Lenstra, Christine Priplata, and Colin Stahlke. Computation of a 768-bit prime field discrete logarithm. In Jean-Sébastien Coron and Jesper Buus Nielsen, editors, *Advances in Cryptology - EUROCRYPT 2017 - 36th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Paris, France, April 30 - May 4, 2017, Proceedings, Part I*, volume 10210 of *Lecture Notes in Computer Science*, pages 185–201, 2017.
13. Donald E. Knuth. *Literate programming*, volume 27 of *CSLI lecture notes series*. Center for the Study of Language and Information, 1992.
14. Donald E. Knuth. *Searching and sorting*, volume 3 of *The Art of Computer Programming*. Addison-Wesley, Reading, Massachusetts, second edition, 10 January 1998. This is a full BOOK entry.
15. Sandeep S. Kumar, Christof Paar, Jan Pelzl, Gerd Pfeiffer, and Manfred Schimmler. Breaking ciphers with COPACOBANA - A cost-optimized parallel code breaker. In Louis Goubin and Mitsuru Matsui, editors, *Cryptographic Hardware and Embedded Systems - CHES 2006, 8th International Workshop, Yokohama, Japan, October 10-13, 2006, Proceedings*, volume 4249 of *Lecture Notes in Computer Science*, pages 101–118. Springer, 2006.
16. Pil Joong Lee and Ernest F Brickell. An observation on the security of McEliece's public-key cryptosystem. In *Workshop on the Theory and Application of of Cryptographic Techniques*, pages 275–280. Springer, 1988.
17. Gaëtan Leurent and Ferdinand Sibleyras. Low-memory attacks against two-round even-mansour using the 3-xor problem. In Alexandra Boldyreva and Daniele Micciancio, editors, *Advances in Cryptology - CRYPTO 2019 - 39th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 18-22, 2019, Proceedings, Part II*, volume 11693 of *Lecture Notes in Computer Science*, pages 210–235. Springer, 2019.
18. Florian Mendel, Tomislav Nad, and Martin Schläffer. Improving local collisions: New attacks on reduced SHA-256. In Thomas Johansson and Phong Q. Nguyen, editors, *Advances in Cryptology - EUROCRYPT 2013, 32nd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Athens, Greece, May 26-30, 2013. Proceedings*, volume 7881 of *Lecture Notes in Computer Science*, pages 262–278. Springer, 2013.
19. Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. Availale online, Dec 2008. https://bitcoin.org/bitcoin.pdf.
20. Mridul Nandi. Revisiting Security Claims of XLS and COPA. *IACR Cryptology ePrint Archive*, 2015:444, 2015.
21. Ivica Nikolić and Yu Sasaki. Refinements of the k-tree Algorithm for the Generalized Birthday Problem. In *ASIACRYPT*, pages 683–703. Springer, 2014.
22. U.S. Department of Commerce, National Institute of Standards, and Technology. *Secure Hash Standard - SHS: Federal Information Processing Standards Publication 180-4*. CreateSpace Independent Publishing Platform, USA, 2012.
23. National Institute of Standards and Technology. NIST policy on hash functions, 2015.

24. Norman Ramsey. Literate programming simplified. *IEEE Software*, 11(5):97–105, 1994.
25. Marc Stevens, Elie Bursztein, Pierre Karpman, Ange Albertini, and Yarik Markov. The first collision for full sha-1. In Jonathan Katz and Hovav Shacham, editors, *Advances in Cryptology – CRYPTO 2017*, pages 570–596, Cham, 2017. Springer International Publishing.
26. Martin Sustrik. nanomsg. https://nanomsg.org.
27. Paul C. van Oorschot and Michael J. Wiener. Parallel collision search with cryptanalytic applications. *J. Cryptology*, 12(1):1–28, 1999.
28. David Wagner. A generalized birthday problem. In *CRYPTO*, pages 288–304, 2002.

A Python Script To Verify the Result

```
from hashlib import sha256
from binascii import hexlify

inputs = [
  b"FOO-0x000000003B1BD2039" + b" " * 53 + b"\xdd\x3f\xf4\x6f",
  b"BAR-0x00000000307238E22" + b" " * 53 + b"\xa8\x0d\xa3\x23",
  b"FOOBAR-0x000000001BB6C4C9F" + b" " * 50 + b"\xb0\x1d\x7c\x21"
]

h = [sha256(sha256(x).digest()).digest() for x in inputs]
xor = bytes([h[0][i] ^ h[1][i] ^ h[2][i] for i in range(32)])

print('  sha256({})'.format(sha256(inputs[0]).hexdigest()))
print('^ sha256({})'.format(sha256(inputs[1]).hexdigest()))
print('^ sha256({})'.format(sha256(inputs[2]).hexdigest()))
print('  ================================================')
print('         {}'.format(hexlify(xor).decode()))
```