Computational Records with Aging Hardware: Controlling Half the Output of SHA-256

Mellila Bouam, Charles Bouillaguet, Claire Delaplace, Camille Noûs

Computational Records with Aging Hardware: Controlling Half the Output of SHA-256

Mellila Bouam[a], Charles Bouillaguet[b,*], Claire Delaplace[c], Camille Noûs[d]

[a]*Ecole Superieure d'Informatique, Alger, Algeria*
[b]*Sorbonne Université, CNRS, LIP6, F-75005 Paris, France*
[c]*MIS Laboratory, Université de Picardie Jules Verne, 14 quai de la Somme, 80080 Amiens, France*
[d]*Laboratoire Cogitamus; https://www.cogitamus.fr/*

Abstract

SHA-256 is a secure cryptographic hash function. As such, its output should not have any detectable property. This paper describes three bit strings whose hashes by SHA-256 are nevertheless correlated in a non-trivial way: the first half of their hashes XORs to zero. They were found by "brute-force", without exploiting any cryptographic weakness in the hash function itself. This does not threaten the security of the hash function and does not have any cryptographic implication.

This is an example of a large "combinatorial" computation in which at least 8.7×10^{22} integer operations have been performed. This was made possible by the combination of: 1) recent progress on algorithms for the underlying problem, 2) creative use of "dedicated" hardware accelerators, 3) adapted implementations of the relevant algorithms that could run on massively parallel machines.

The actual computation was done on aging hardware. It required seven calendar months using two obsolete second-hand bitcoin mining devices converted into "useful" computational devices. A second step required 570 CPU-years on an 8-year old IBM BlueGene/Q computer, a few weeks before it was scrapped.

To the best of our knowledge, this is the first practical 128-bit collision-like result obtained by brute-force, and it is the first bitcoin miner-accelerated computation.

Keywords: 3XOR, Generalized Birthday Paradox, Brute-force, Implementation, Hardware, ASIC, bitcoin hardware

1. Introduction

Cryptography is an ubiquitous component of information security. Cryptographic algorithms are used to enforce security properties such as secrecy, integrity and authenticity. A few cryptographic algorithms are widely deployed, and ensure the secure operations of several key economic sectors. The RSA signature algorithm guarantees the authenticity of all European credit cards and enables end-user to digitally sign electronic mail. Both RSA and variants of the Diffie-Hellman key-exchange protocol (DH) are present in the majority of "secure" network connection in the world, because they are the main components of the TLS secure network layer. They secure connections to emails accounts, allow web-browsers to authenticate banking, e-commerce or governmental websites, enable remote connections to "Virtual Private Networks", etc.

Public-key cryptographic algorithms rely on the hardness of well-defined computational problems. Concretely, the security of RSA relies on the hardness of factoring large integers, while that of the Diffie-Hellman key-exchange relies on the hardness of computing discrete logarithms in some groups (usually the multiplicative group of integers modulo a large prime number or the group of points on an elliptic curve).

Cryptanalysis is the art of breaking the security properties that are supposed to be guaranteed by cryptographic schemes. Research in cryptanalysis is both of a theoretical and practical nature; on the theoretical side, weaknesses in some cryptographic schemes can be found "on paper": given enough resources, an algorithm could potentially break the security properties offered by a cryptographic scheme faster than expected. On the other hand, when the break is practical, consequences are usually more dramatic, especially if it is widely deployed; this puts more pressure on industrial actors to update their products.

Cryptographic hash function play an important yet special role in cryptology: as opposed to encryption or data authentication schemes, their security does not depend on the confidentiality of any secret data (such as an encryption key). While formalizing the expected security properties of hash function *families* is fairly straightforward, precisely defining the security of *fixed* and *public* cryptographic hash function such as SHA-256 is a long-standing problem (see e.g. [Rog06]). Informally speaking, a cryptographic hash function is a fixed, public function *without structure*: given an arbitrary input, its output should appear indistinguishable from a random bit string, i.e. it should appear completely decorrelated from the input. It follows that it should

*Corresponding author
Email addresses: em_bouam@esi.dz (Mellila Bouam), charles.bouillaguet@lip6.fr (Charles Bouillaguet), claire.delaplace@u-picardie.Fr (Claire Delaplace), camille.nous@cogitamus.fr (Camille Noûs)

not be possible to construct inputs such that their hashes exhibit a detectable correlation. At the very minimum, a cryptographic hash function should be one-way and collision-resistant — a collision for a function f is a pair $x \neq y$ such that $f(x) = f(y)$, and f is collision-resistant if finding such a pair is intractable.

SHA-256 is presently considered to be a secure cryptographic hash function, and it is widely used. It was designed by the NSA in 2001, and it is one of the few cryptographic hash functions standardized by the government of the United States of America for its own use [oST15].

1.1. Context of this Work

In the cryptographic community, the "3XOR problem" consists in finding three n-bit strings x, y and z such that $f(x) \oplus g(y) \oplus h(z) = 0$, where $\oplus$ denotes the XOR operation and f, g, h are random functions from $\{0,1\}^n$ to $\{0,1\}^n$. This problem has recently seen a renewed interest because of its potential use in cryptanalysis: it is used as a building block in a generic attack [Nan15] against the COPA [ABL+13] mode of operation for authenticated encryption. Another recent attack [LS19] against the two-round single-key Even-Mansour cipher [EM91] also works by reducing it to a 3XOR computation. An instance of the 3XOR problem can be solved by several algorithms summarized in Table 1. The first two, which are described below, are folklore and serve as a meaningful baseline to compare against.

The Quadratic Algorithm. It is possible to build three arrays $\mathtt{A}, \mathtt{B}$ and $\mathtt{C}$ such that $\mathtt{A}[i] = f(i)$, $\mathtt{B}[i] = g(i)$ and $\mathtt{C}[i] = h(i)$; the task then consists in finding $(x, y, z) \in \mathtt{A} \times \mathtt{B} \times \mathtt{C}$ such that $x \oplus y \oplus z = 0$. This can be done by trying all pairs $(x, y) \in \mathtt{A} \times \mathtt{B}$ and checking if $x \oplus y$ belongs to $\mathtt{C}$. Because there are 2^n triplets in total, we expect to find one such that the n-bit equality holds.

This is the *quadratic algorithm*. In this form, it requires $n \cdot 2^{n/3}$ bits of memory, and performs $O\left(2^{2n/3}\right)$ operations. Note that the random functions f, g, h only have to be evaluated $2^{n/3}$ times. The inputs to the random functions can be arbitrary, as long as they are all distinct.

Collision-Finding. Alternatively, one could fix $z \leftarrow 0$ and set $g'(y) \leftarrow g(y) \oplus h(0)$. The task then becomes finding x, y such that $f(x) = g'(y)$, (i.e. finding a *collision* between two random functions). This can be done sequentially without any memory using a slightly modified cycle-finding algorithm (this is called the "ρ algorithm" in cryptology); this requires on average $\sqrt{\pi} 2^{n/2}$ evaluations of f and g plus an equivalent number of bookkeeping operations. It can be parallelized with essentially linear speed-up and a controllable amount of communication using the algorithm of van Oorshot and Wiener [vOW99] for parallel collision search.

This is the *collision-finding* approach. It requires the ability to evaluate the random functions f and g "adaptively", namely on inputs determined during the course of the computation and not known in advance.

Recent Improvements. Advances on the 3XOR problem in the cryptographic community aimed at reducing the total computational load while ignoring potential memory and hardware constraints. Joux proposed an incremental improvement in [Jou09], which reduces the computational load by $\sqrt{n}$ at the expense of using an exponential amount of memory and increasing the number of queries to the random functions. Motivated by the potential cryptanalytic applications, several new algorithms were discovered by Nikolić and Sasaki [NS14] and then later by Bouillaguet, Delaplace and Fouque [BDF18].

1.2. Objectives and Results

The attacks that use 3XOR computations as a sub-component have not been implemented. Previous work on the 3XOR problem is also of a mostly theoretical nature. What is the practical efficiency of the five algorithms listed in Table 1? If someone actually wanted to solve an instance of the 3XOR problem in practice, what would they do? Seeking to answer these questions, we set up a large instance of the 3XOR problem and tried to solve it.

We settled for $n = 128$, a significant milestone: it it the smallest power of two for which the computation is not obviously easy at the present time. In addition, this size is cryptographically meaningful: the MD5 cryptographic hash function produces 128-bit digests and was once vastly deployed[1] ; 128-bit is currently a standard key size for symmetric encryption. We used a contemporary cryptographic hash function (SHA-256), truncated to 128 bits, as f, g and h (using different input prefixes to distinguish the three functions): because it is considered secure, it should mimic reasonably well the behavior of random functions.

Solving this instance of 3XOR would mean being able to "control" the first half of the output of SHA-256. This would have no immediate cryptographic consequences and would not threaten the security of any known cryptographic protocol. But this has not been done before and it is assumed to be difficult.

We were able to solve this large instance of the 3XOR problem. This necessitated a somewhat large computational effort that spanned over seven months. In total, we evaluated the "compression function" of SHA-256 $2^{67.6}$ times, which makes it a large cryptanalytic computation. This demonstrates that the algorithms we used are practical, and that brute-force cryptographic "birthday attacks" on 128 bits can be feasible in some circumstances.

The main ingredient that allowed us to go through with such a large computation (with a modest budget) is that it was ideally suited to exploit unconventional but nearly off-the-shelf hardware accelerators: *bitcoin mining devices*. These inexpensive and power-efficient machines contain ASICs dedicated to the evaluation of SHA-256. A single bitcoin mining device is capable of evaluating SHA-256 at least one million times faster

[1] MD5 was designed in 1992 and was first broken in 2005 [WY05]. Its weaknesses were exploited, among others, by the Flame malware to fake a Microsoft digital signature in 2012 [Ste13]. It was deprecated by Microsoft in its own products in 2014.

Algorithm	Ref.	Running time	Memory	Inputs
Quadratic	folklore	$2^{2n/3}$	$2^{n/3}$	Arbitrary
Collision	folklore	$2^{n/2}$	1	Chosen adaptively
Nikolić and Sasaki	[NS14]	$2^{n/2}/\sqrt{n/\ln n}$	$2^{n/2}/\sqrt{n/\ln n}$	Arbitrary
Joux	[Jou09]	$2^{n/2}/\sqrt{n}$	$2^{n/2}/\sqrt{n}$	Arbitrary
Generalized Joux	[BDF18]	$2^{2n/3}/n$	$2^{n/3}$	Arbitrary

Table 1: Generic algorithms for the 3XOR problem. Quantities are asymptotic. Inputs denote the values fed to the random functions f, g and h.

than a CPU core, but it only does so in a restricted way. Solving our instance of the 3XOR problem with SHA-256 happened to be one of the few computations that could potentially be bitcoin miner-accelerated. The two other ingredients are fast algorithms for the 3XOR problem and efficient parallel implementations thereof.

We used the following approach to make the computation tractable. Instead of dealing with an unwieldy instance of the 3XOR problem on 128 bits, we invested quite a lot of time in assembling a special, smaller instance of the problem: in order to find (x, y, z) such that $f(x) \oplus g(y) \oplus h(z) = 0$, we restricted our attention to inputs x, y, z such that the first 32 bits of $f(x), g(y)$ and $h(z)$ are equal to zero. This reduces the problem to finding a 3XOR on a much smaller instance with $n' = 96$ bits. On the other hand, the inputs to the random functions are now imposed upon us, and we lose the freedom to choose them.

In more detail, we built three lists `A`, `B` and `C`, each containing 2^{32} values $f(x), g(y)$ and $h(z)$ beginning with 32 zero bits. Each list requires 52GB, which is much more manageable. Finding by brute force an x such that $f(x)$ begins with 32 zero bits requires in expectation 2^{32} random trials. Building the lists therefore requires 3×2^{64} evaluations of the random functions. Once this is done, $\mathtt{A} \times \mathtt{B} \times \mathtt{C}$ contains 2^{96} triplets $f(x), g(y), h(z)$ on 96 bits, and we expected one of these to be a full "3XOR triplet" with $f(x) \oplus g(y) \oplus h(z) = 0$.

It turns out that bitcoin mining devices are ideally suited to the task of building these three lists. We spent seven calendar months "mining" `A`, `B` and `C`, using two second-hand bitcoin mining devices.

After the three lists had been built, it remained to find the actual 3XOR triplet hidden in $\mathtt{A} \times \mathtt{B} \times \mathtt{C}$. The lists have size $2^{n'/3}$ with $n' = 96$, therefore using the quadratic algorithm would yield a solution with 2^{64} probes in a hash table. In practice, we used the slightly improved algorithm of [BDF18].

We solved this smaller instance of the 3XOR problem (with $n' = 96$ bits) in four calendar days using 65536 cores simultaneously on an aging IBM BlueGene/Q parallel computer. This required about $2^{64.6}$ CPU cycles on its PowerPC A2 CPUs running at 1.6Ghz, or equivalently 570 CPU-years.

The actual results can be seen in Fig. 1. We give three innocuous bit strings whose digest by SHA-256 are strongly correlated (they XOR to zero on the first 128 bits). We emphasize that these three special bit strings have been found by "brute-force"; the results presented here do not undermine the security of SHA-256. We did not discover nor exploit any new cryptographic weakness. The `pyhton` script in Appendix A checks that the solutions are correct.

Trying to actually implement and run several 3XOR algorithms in order to solve a large-scale problem led us to an unavoidable conclusion: many good "theoretical" algorithms have little practical value as-is, either because of their unrealistic memory requirements or because of their unmanageable communication complexity. Evaluating an algorithm in the "Random Access Machine" abstract computer model, using the number of "elementary operations" as a sole metric leads to impractical results. This is probably obvious to the HPC world, but other more theoretical research communities seem to be oblivious to this issue or voluntarily disregarding it. Looking at the five algorithms summarized in table 1, it appears that the requirement for about $2^{n/2}$ "memory cells" is completely impractical for the range of n that we consider. The corresponding algorithms had to be summarily rejected for this work.

All the programs used to produce the result of Fig. 1 represent 3700 lines of C code. They are publicly available at:

`https://github.com/cbouilla/3XOR-mining`

1.3. Cost and Energy

The whole computation (mining + solving) required about 40MWh of energy, which is about 144GJ. This is enough energy to boil 430m^3 of water (a $25m \times 10m \times 1.72m$ swimming pool) starting from 20°C. The bitcoin miners ate 10.8MWh over seven months, while 65536 cores of the BlueGene/Q consumed the remaining 30MWh over four days. Given the electricity rates and the energy mix in France, where the computation took place, this makes roughly 3000€ of electricity and two tons of CO_2 released into the atmosphere.

In addition, the computation center that granted us 10 million CPU-hours on a BlueGene/Q 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).

For the sake of comparison, the 2017 practical collision attack on SHA-1 [SBK+17] required an effort equivalent to $2^{63.1}$ evaluations of SHA-1. 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 according to its authors. Evaluating SHA-256 requires approximately twice as many operations than evaluating SHA-1, and thus we did about 45 times more total "hashing work".

In reality, we used a *much* smaller amount of computational resources, as well as *much* less money and energy, thanks to the

Consider the three 80-byte ASCII strings given below:

$a =$ "FOO-0x000000003B1BD2039" + "␣" × 53 + dd3ff46f,
$b =$ "BAR-0x00000000307238E22" + "␣" × 53 + a80da323,
$c =$ "FOOBAR-0x000000001BB6C4C9F" + "␣" × 50 + b01d7c21.

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

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

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

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

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

$\Delta =$ 00000000 00000000 00000000 00000000
7effee95 6653817d c6c0d1d7 f8ceb9cc.

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

efficient dedicated ASICs available in bitcoin mining devices. The attack on full SHA-1 is definitely *much* more challenging than the computation described in this paper (and more critical from a cryptographic point of view).

1.4. Comparison with a Simple Collision Search on 128 Bits

The algorithmic strategy outlined above requires $2^{n/2}$ evaluations of the random functions to build a special instance of the problem, then spends $2^{n/2}/n$ operations to solve it, using $2^{n/4}$ memory. Given this figures, why not use the folklore and well-known "collision-search" approach?

We claim that finding a single 128-bit collision on SHA-256 truncated to 128 bits is actually *much more difficult* than obtaining the result shown in Fig. 1.

Recall that this would mean finding $x \neq y$ such that $f(x) = f(y)$, where f is SHA-256 truncated to 128 bits. As argued above, this would require about to evaluate SHA-256 on $2^{64.8}$ *adaptively chosen* inputs to succeed. This requirement is the actual problem: we do not see how bitcoin miners nor other special equipment could be used to accelerate it. Evaluating SHA-256 requires at least 1496 arithmetic operations on 32-bit integers (additions, XORs, ANDs, ORs). As such, finding the collision seems to require 8.7×10^{22} integer operations on "generic" computational hardware, and this looks quite difficult, at least not without a large budget and/or access to a very large computational facility.

1.5. Related Work

The 3XOR problem is a specific (and difficult) case of the "generalized birthday problem" [Wag02], which requires finding $x_1, \ldots, x_k$ such that $f_1(x_1) \oplus \cdots \oplus f_k(x_k) = 0$. This also has interesting cryptographic applications, and several cryptographic constructions can be attacked by solving instances of the generalized birthday problem with $k > 3$. An example is [BLN+09], which describes an actual implementation of an attack against (reduced version of) the FSB hash function [AFS05].

Cryptographic attacks using ASICs are rare. The only example known to us is the "Deep Crack" machine [Fou98] designed in 1998 to break the DES block-cipher 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 algorithms and actual cryptanalytic attacks targeting real cryptographic constructions have been run on GPUs; this is for instance the case of the first actual collision on SHA-1 [SBK+17]. Several cryptanalytic algorithms have also been run on FPGAs for instance using the COPACOBANA machine [KPP+06], including exhaustive key-search for DES or solving systems of multivariate quadratic equations modulo 2 [BCC+13].

2. A Brief Description of SHA-256

SHA-256 is one of four hash functions defined in the Federal Information Processing Standard (FIPS-180-2) [oCoST12]. It was designed by the National Security Agency (NSA) and issued by NIST in 2002. It outputs a 256-bit hash given an (almost) arbitrary quantity of input data. The input is usually called the *message* in the cryptographic community.

The hash is obtained by an iterated process: the input is split into *message blocks* of a fixed size (512 bits) that are processed one at a time using the *compression function* F, an inner hash function that hashes 768 bits into 256. The input message is therefore padded to a multiple of 512 bits and we have:

$$\begin{aligned} h_{-1} &= IV, \\ h_i &= F\left(h_{i-1}, m_i\right). \end{aligned}$$

The *Initialization Vector* IV is a fixed 256-bit constant, the h_i's are 256-bit successive *chaining values* of the hash function and the m_i's are the 512-bit message blocks. The hash of the input data is the last chaining value. It follows that evaluating SHA-256 on a (say) 80-byte message requires two invocations of the compression function.

The bulk of the hashing process happens in the compression function F; it takes as inputs a 256-bit chaining value, a 512-bit message block and yields a new chaining value. It operates on 32-bit words.

Computational Cost. Evaluating the compression function of SHA-256 requires 600 additions, 192 ANDs, 128 ORs, 576 rotations, 96 shifts and 576 XORs, which makes a total of 2168 operations on 32-bit integers. If we assume that rotations and shifts are "free" (they sometimes can be realized quite cheaply in ad hoc hardware), we are left with 1496 unavoidable arithmetic operations.

It must be noted that SHA-256 can be evaluated several times in parallel using SIMD instructions (Intel provides such optimized implementations [GYG12]). In addition, some recent CPUs, starting with Intel "Ice Lake" and AMD Zen processors, have hardware instructions to perform both the message expansion (`sha256msg1`, `sha256msg2`) and the step update function (`sha256rnds2`), which is the bulk of the compression function.

Note that even with these tricks, evaluating the compression function about 2^{64} times is still a large workload.

3. Computational Hardware

In this section, we describe the aging hardware we used to solve the large 3XOR instance described in the introduction. We used bitcoin mining devices to assemble an easier subproblem and a more usual parallel machine to solve it.

3.1. Bitcoin Mining Devices

As mentioned in the introduction, we chose to compute a 3XOR on 128 bits by accumulating three lists of 2^{32} bit strings each, whose hash start with 32 zero bits (looking at Fig. 1 shows that the hashes of x, y, z have this special property). This requires 3×2^{64} expected evaluations of SHA-256.

Using the `cpuminer` program[2], which uses hand-written assembly implementations of SHA-256 using AVX2 instructions, we measured that a single core of an Intel Xeon Gold 6130 CPU at 2.10GHz can evaluate SHA-256 about 10 million times per second in the best case. Performing the 3×2^{64} evaluations of SHA-256 thus requires more than 175'000 CPU-years on these CPUs. It took seven calendar months using two inexpensive bitcoin mining devices.

These machines contain circuits devoted to the fast evaluation of the function:

$$\Phi : M \in \{0,1\}^{608} \mapsto \Big\{ x \in \{0,1\}^{32} \;\Big|\; \exists y \in \{0,1\}^{224} : \mathsf{SHA\text{-}256d}(M \parallel x) = \mathtt{0x00000000} \parallel y \Big\},$$

where $\parallel$ denotes concatenation and the SHA-256d hash function is defined as $\mathsf{SHA\text{-}256d}(x) = \mathsf{SHA\text{-}256}(\mathsf{SHA\text{-}256}(x))$. It follows that we just had to pick arbitrary M's and evaluate $\Phi(M)$. This yields sets of x's such that $\mathsf{SHA\text{-}256}(\mathsf{SHA\text{-}256}(M \| x))$ begins with 32 zero bits, which is exactly what we need. Indeed, looking again at Fig. 1, the double evaluation of SHA-256 on a, b and c is visible, as well as the special structure of a, b, c: the prefixes (`FOO`, `BAR`, `FOOBAR` and the counters) were supplied by us to the miners (in M), while the 4 random-looking bytes at the end were found as a result of evaluating Φ.

[2] `https://github.com/pooler/cpuminer`

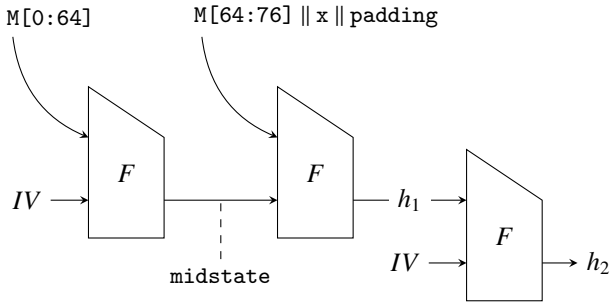

Figure 2: Evaluation of SHA-256d on an 76-byte prefix M and a 4-byte suffix x.

In bitcoin mining devices, the Φ function is evaluated by brute-force, namely by trying all the possible 2^{32} possibles values of x. Fig. 2 shows the hashing process. The compression function of SHA-256 needs only be evaluated twice, because the prefix M is the same for all possible x's and thus the "`midstate`" is independent of the choice of x. It follows that computing $\Phi(M)$ essentially requires 2^{33} compression function evaluations.

At the time of this writing, commercial high-end bitcoin mining devices present themselves as standalone devices that only require a power cable and Ethernet connectivity. They usually expose a web interface for configuration purposes. Once they have been set up, operations are mostly automatic and require almost no supervision.

The actual device we used is an AntMiner S7 from Bitmain (acquired for 500€). It contains three boards, each holding 45 custom "BM1385" 28nm ASICs clocked at 600MHz. The whole device is capable of doing 4.7×10^{12} evaluations of SHA-256d per second (meaning $2^{43.1}$ SHA-256 compression function evaluations per second).

We gather from their very partial specification [BM] that the ASICs in each board seem to be linked in a chain. They communicate with their controller using a standard serial protocol. We understand that the ASICs take as input: the chaining value after the first invocation of the compression function (the "`midstate`"), bytes 64–76 of M, and a zero-bit count. They likely return the possible x's such that $\mathsf{SHA\text{-}256d}(M \parallel x)$ starts with the prescribed number of zero bits.

The AntMiner S7 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.

Given the hashing power of this mining device, in the best case we could expect to evaluate the Φ function $\approx$ 1094 times per second. If this were true, three lists of size 2^{32} could potentially be assembled in only 136 days.

3.2. A Parallel Computer

Once the special instance of the 3XOR problem with 32 leading zero bits would be assembled, it would remain to solve it. This happened on an 8-year old massively parallel IBM

BlueGene/Q computer named `turing` and located at IDRIS ("Institut du Développement et des Ressources en Informatique Scientifique"), a national computing center. The whole computation took 5 million CPU-hours over 4 calendar days.

This machine is a small BlueGene/Q installation, made of only 6 racks, each with 32 compute boards, each with 32 compute nodes. The nodes are connected by a custom 5D torus network. Each node contains a 64-bit PowerPC A2 processor running at 1.6Ghz with 16 available cores and 16GB of RAM.

The processors do not support paging nor virtual memory: all memory addresses are physical. An advantage is that there are no TLB faults nor page walks. The CPU cores are in-order and support 4 simultaneous hardware threads. A core can execute at most two instructions per clock cycle (one integer, one floating-point), taken amongst any of the four threads. We are only doing integer computations, therefore actually executing one Instruction per Cycle (IPC= 1) is potentially challenging.

Each core has 16KB of L1 cache (256 cache lines of 64 bytes), shared between the threads. The L1 cache is write-through and has a 4-cycle hit latency, therefore using more than one hardware thread is required to keep the CPU busy. A 32MB L2 cache with an 80-cycle hit latency is shared between the 16 cores. It was therefore critical to exploit the L1 cache as much as possible.

These CPU cores are simple and relatively slow, which is to some extent an advantage: for instance it is quite difficult to saturate the memory bandwidth, which is roughly 25GB/s. We used 4096 nodes (65536 cores) simultaneously. The specificities of this platform dictated some algorithmic choices and parameters tuning, as discussed in section 5.6.

The only supported programming languages are C and Fortran, via IBM's `xlc` compilers. We tried to keep the programs running on this machine as simple as possible and avoided relying on external libraries (we initially used the M4RI [AB12] library for linear algebra over $\mathbb{Z}_2$ but eventually recoded several functions because it was simpler). Because we had to write multi-thread code, we used OpenMP.

4. Converting Bitcoin Miners into Compute Accelerators

The global algorithmic strategy described in the introduction consists in accumulating lists of hashes beginning with Z zero bits, then solving this smaller instance of the 3XOR problem. We used $Z = 33$, and we now explain why.

Three lists of $(128 - Z)$-bit vectors, of length $N = 2^{(128-Z)/3}$, contain on average one 3XOR triplet. Finding by brute-force a single hash beginning with Z zero bits requires 2^Z trials. Therefore, the cost of building the lists is essentially $2^Z N = 2^{42.7+2Z/3}$. Finding all 3XOR triplets in these lists using the quadratic algorithm or the improved algorithm from [BDF18] takes time $N^2 = 2^{85.3-2Z/3}$. Balancing the cost of the two phases suggests to use $Z = 32$, which is, in any case, the minimum value for which bitcoin mining devices are efficient.

Recent Bitcoin mining devices implement an *ad hoc* communication protocol to connect to a "pool server", receive work and submit results. The *stratum pool mining protocol* [str] is widely implemented at this time. Interacting with bitcoin miners trough the stratum protocol, while feasible, was unfit for our purposes. In "normal use" (to mine bitcoins), a valid block is found and reported every few seconds. Our particular use case consisted in finding ≈ 1000 valid blocks per second. The stratum protocol incurred too much overhead and the low-cost ARM cortex-A8 running the mining program could not keep up. It saturated at less than 50 block/s.

To circumvent this problem, we had to change the program controlling the device. An undocumented shell access is exposed through `ssh`. The device 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 a specific branch in this git repository, the `bitmain_fan-ctrl` branch. This code could be cross-compiled for ARM devices and the resulting binary could be copied to the device and run. It is therefore possible to alter the control software of the mining operations.

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 block M missing its 4-byte suffix x, using the notations of section 3.

- The "stratum component" interacts with the mining pool server: it receives updates about the current work to do (which changes every 10 minutes; the work queue is then flushed) and is responsible for sending back results to the pool.
- The program runs into an infinite loop that generates new work items 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. This evaluates the Φ function on the current work item. A callback from the stratum component is invoked for each suffix x found.

Each component has several threads, manages several internal queues (both in software and in hardware for the device components parts), etc. We essentially scrapped the stratum component and replaced both its work generator and networking code with our own. To avoid overloading the low-cost ARM CPU of the device, we used the simplest possible network 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 work and generating the corresponding 76-byte blocks as prescribed by the stratum protocol, 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 suffix x is found, the pair (counter, x) 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.

We used the nanomsg [Sus] message-passing library to address networking issues. It is written in plain C, is easy to

cross-compile to the ARM device and works around network connectivity problems gracefully, while offering well-defined semantics. The corresponding home-made server is a small C program that mostly stores the (counter, x) pairs it receives from the miners in a file and keeps the current value of the counter up to date. The storage requirement is then 12 bytes per submitted block.

Using this setup, we were able to collect about 550 block/s, each yielding a hash that starts with 33 zero bits. We could not "mine" the 1000+ block/s we hoped for with only 32 zero bits (this was crashing the miners for some reason). Using a single AntMiner S7, the process would have taken 10 full months; after four months, we acquired another used AntMiner S7 for 80€ and put it in production to speed things up.

After a seven-month mining campaign, three lists of 2^{32} (counter, x) pairs had been accumulated, each yielding a hash starting with 33 zero bits. This was enough to expect two solutions on 128 bits (and made us confident that at least one would be found). In total, $2{\times}3{\times}2^{32}{\times}2^{33} = 2^{67.6}$ compression function evaluations have been performed by these two mining devices. The total volume of data accumulated was 155GB.

5. Solving the 3XOR Instance

After having accumulated three lists A, B and C each containing of 2^{32} hashes on 95 bits using our bitcoin miners, it remained to find $(x, y, z) \in \mathtt{A} \times \mathtt{B} \times \mathtt{C}$ such that $x \oplus y \oplus z = 0$. This section details how we have managed the computation of the solution to this large instance of the 3XOR problem.

5.1. 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 only had access to the BlueGene/Q computer described above. Note that each input lists weights 51.5 GByte, while compute nodes only have 16 Gbyte of RAM.

We considered implementing a distributed version of the algorithm of [BDF18], but we believed that this would have been impractical because of the communication complexity that this would have generated.

Instead, we used a simple divide-and-conquer approach: we partitioned 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 the recurrence $T(N) = 4T(N/2)$, which yields $T(N) = O\left(N^2\right)$.

Repeating this divide-and-conquer step p times recursively splits each input array into 2^p smaller arrays and splits the input problem into 4^p independent subproblems. This makes solving our original 3XOR instance embarrassingly parallel, which is exactly what we wanted. We chose $p = 15$ (for reasons discussed below): this resulted in a billion independent subproblems in which each input array has expected size $2^{32}/2^{15} = 2^{17}$.

5.2. Vector Length

This parallelization scheme yields a billion instances of the 3XOR problem with 80-bit hashes (33 bits are known to be zero and 15 more bits are dealt with automatically by the divide-and-conquer steps). Dealing with 80 bits is not very practical because usual CPUs have 64-bit registers. Therefore, we adopted the following strategy: we compute all triplets that are 3XOR on the first 64 bits (such that $x[0..64] \oplus y[0..64] \oplus z[0..64] = 0$).

Each subproblem contain $\left(2^{17}\right)^3$ triplets; the expected number of triplets "matching" on 64 bits is therefore 2^{-13} (of course, the most likely outcome is that a single subproblem does not contain any 3XOR triplet on 64 bits). There are 2^{30} subproblems, and we expect to find 2^{17} triplets matching on 64 bits in total. By chance, we expect 2 of them to match on 16 additional bits to reach 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. We used a specially tuned implementation of the algorithm of [BDF18], which requires frequent access to two of the three arrays. Each array requires 1MB, and with 16 cores on a PowerPC A2 CPUs each running an independent copy of the algorithm, the 32MB of L2 cache are enough to hold all the frequently accessed data. This is partly why we chose $p = 15$ above.

5.3. Preprocessing and Data Management

The mining server program generates *preimage files* composed of 12-byte (counter, x) pairs. At 550 block/s, this means roughly 550 Mbyte of fresh data per day. 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 erred 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 p-bit *partitioning key* from the full hash by taking bits [97..97+p].

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 $\mathtt{hash} \rightarrow \mathtt{preimage}$. A *dictionary 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.

When a new preimage file is ready, we split it into 2^p *sub-dictionaries* using the partitioning key. Potential duplicate preimages are now confined into a single sub-dictionary. We sort the sub-dictionaries by hash. This is easily done, since they are quite small and fit in RAM.

For each partitioning key, we perform a multi-way merge on all sub-dictionaries. Duplicate hashes are detected and removed

(they were the result of "human errors"). We write down the sorted unique hashes into a single *hash file* per partitioning key.

For the C list, we run algorithm S (cf. Fig. 3) on all the 2^p hash files, which results in as many *slice files*. A slice file contains a sequence of slices of C, along with some additional information (see below).

All these tasks were accomplished by a collection of small (and in as much 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 this preprocessing was run on a desktop PC.

5.4. Solving the Independent Subproblems

At this point, all that is left is to write an efficient sequential program capable of finding all the solutions of small instances of the 3XOR problem on 64-bit vectors. We essentially had the choice between the folklore quadratic algorithm and the generalized Joux algorithm of [BDF18] to do so. The latter is better asymptotically; we knew it to be roughly 3 times faster than a well-optimized implementation of the quadratic algorithm on x86 CPUs, and therefore we opted to use it. We recall this algorithm in Fig. 3, along with a few brief comments below.

The main algorithmic idea, which is due to Joux [Jou09], consists in finding the "change of basis" matrices M_i and the "slices" $\mathtt{C}_i$ such that M_i cancels out the first k coordinates of all rows of $\mathtt{C}_i$. Because the slices $\mathtt{C}_i$ form a partition C, solving all subinstances $\mathtt{A} \times \mathtt{B} \times \mathtt{C}_i$ guarantees to find all solutions. The main loop of algorithm G tries all $\mathtt{C}_i$ one by one. To reduce the number of iterations, the slicing step (G1) should produce the smallest possible number of slices of C — *i.e.* the biggest possible slices.

The point of these changes of variables is that if there is a solution $x \oplus y \oplus z = 0$ with $(x, y, z) \in \mathtt{A} \times \mathtt{B} \times C_i$, then by linearity $xM_i \oplus yM_i \oplus zM_i = 0$. Set $x' \leftarrow xM_i, y' \leftarrow yM_i$ and $z' \leftarrow zM_i$; it follows that (x', y', z') is also a 3XOR triplet in $\mathtt{A}' \times \mathtt{B}' \times \mathtt{C}'_i$. Thus, searching all the solutions in $\mathtt{A}' \times \mathtt{B}' \times \mathtt{C}'_i$ is sufficient, but it is easier.

Indeed, because all vectors in $\mathtt{C}'_i$ have their first k coordinates equal to zero, it follows that x' and y' must necessarily *match* on their first k coordinates. Finding all $(x', y', z') \in \mathtt{A}' \times \mathtt{B}' \times \mathtt{C}'_i$ is therefore akin to computing the *join* of A and B on the first k bits: find all $(x', y') \in \mathtt{A}' \times B'$ such that $x'[0..k] = y'[0..k]$, then for each pair, check if $x' \oplus y' \in \mathtt{C}'_i$. This is what algorithm J does.

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 $\log_2 |\mathtt{A}|, \log_2 |\mathtt{B}|$ are small compared to 64. However, if the subproblems become too small, on the other hand, the "administrative overhead" is going to dominate.

All three algorithms depend on two parameters $0 \leq \ell < k \leq 64$. Choosing their best possible values is a delicate balancing act. The first one k, specifies the number of coordinates that must be cancelled by the changes of variables. The other one controls the number of sub-partitions in algorithm J, which does a partitioned hash join. A small k leads to bigger slices of C, and therefore reduces the number of iterations of the main loop of algorithm G. On the other hand, a large k speeds up each individual iteration by making algorithm J faster. In theory, the optimal value of k is $\lceil \log_2 \min(|\mathtt{A}|, |\mathtt{B}|) \rceil$, which in our case should be 17. Other practical considerations (discussed below) led us to use $k = 19$.

5.5. Slicing C

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

Finding such a slice of $\mathtt{C}_i$ is also done greedily: a single equation satisfied by as many vectors of $\mathtt{C}_i$ as possible is found (this is the classic NP-hard problem of decoding an arbitrary linear code), and the process is iterated on vectors satisfying this equation.

We used the Lee-Brickell [LB88] algorithm to find low-weight codewords. It is not the best 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 yield the best possible result — but decoding up to the optimal bound is computationally intractable given the problem sizes.

Fig. 4 shows the distribution of the sizes of slices resulting from running algorithm S 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 M_i, then all slices would have had size $64 - k = 45$. Using algorithm S 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).

As a preprocessing step, we ran algorithm S on the $2^p = 32768$ independent parts of C using 32768 cores simultaneously for 4 hours. Less than 3.5% of the total computation time was spent in algorithm S. The algorithm outputs the actual slices $\mathtt{C}_i$, along with the corresponding matrices M_i and their inverse. This is so that the actual 3XOR code (algorithms G and J) does not have to compute the inverses "online", mostly for the sake of keeping them as simple as possible.

We refer to [BDF18] for more algorithmic details.

5.6. Tuning the Algorithms to the Underlying Hardware

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

5.6.1. Matrix Multiplication over $\mathbb{Z}_2$ (step G3)

We have to multiply a $m \times 64$ matrix (with large m) by a 64×64 matrix M, over $\mathbb{Z}_2$. We use a variant of the "four Russians

Algorithm G (*3XOR by linear changes of variables*). Given $\mathtt{A}, \mathtt{B}$ and $\mathtt{C}$, return all 3XOR triplets $(x, y, z) \in \mathtt{A} \times \mathtt{B} \times \mathtt{C}$. The algorithm also takes a parameter $0 < k < 64$.

G1. [Slice $\mathtt{C}$.] Using algorithm S (below), partition $\mathtt{C}$ into *slices* $\mathtt{C}_1, \ldots, \mathtt{C}_m$ with associated 64×64 matrices $M_1, \ldots, M_m$ such that:

$$\mathtt{C}_i \times M_i = \begin{pmatrix} 0 & \ldots & 0 & \star & \ldots & \star \\ \vdots & & \vdots & \vdots & & \vdots \\ 0 & \ldots & 0 & \star & \ldots & \star \end{pmatrix} \quad \updownarrow \geq 64 - k$$
$$\underbrace{\qquad\qquad}_{k} \quad \underbrace{\qquad\qquad}_{64-k}$$

G2. [Loop over slices.] For all $1 \leq i \leq m$, perform steps G3–G4 (this finds solutions in $\mathtt{A} \times B \times C_i$) 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 $\left(x'M_i^{-1}, y'M_i^{-1}, z'M_i^{-1}\right)$ as a solution of the original problem. ∎

Algorithm S (*Slicing 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 $M_1, \ldots, M_m$.

S1. [Setup.] Let $i \leftarrow 0$.

S2. [All finished?] If $\mathtt{C}$ contains strictly more than $64 - k$ elements, then go to step S3. Otherwise, set $\mathtt{C}_i \leftarrow \mathtt{C}$; find M_i by solving a linear system; set $m \leftarrow i$ and terminate the algorithm.

S3. [Start new slice.] Initialize $j \leftarrow 0, T \leftarrow \mathtt{C}$ and let M_i be the 64×64 zero matrix.

S4. [Decoding.] Find a low-weight linear combination y of the columns of T ($y = \sum_{k=1}^{64} x_k T_k^t$, where y has low hamming weight). This is done using an Information Set Decoding algorithm such as Lee-Brickell.

S5. [Update Slice.] Remove from T all rows where y is non-zero. Store $x_1, \ldots, x_{64}$ in the j-th column of M_j.

S6. [Slice done?] If $j < k$, return to step S4.

S7. [Finalize slice.] Pad M_j with linearly independent columns; set $C_i \leftarrow T$; $\mathtt{C} \leftarrow \mathtt{C} - T$.

S8. [Move to next slice.] Increment i and go back to step S2. ∎

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 $(x, y, z) \in \mathtt{A} \times \mathtt{B} \times \mathtt{C}$ (it follows that $x[0..k] = y[0..k]$). The algorithm is given k and also another parameter $0 \leq \ell < k$.

J1. [Setup.] Initialize a (static) hash table H 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 $x[0..\ell] = i$. A potential 3XOR triplet necessarily belongs in $\mathtt{A}_i \times \mathtt{B}_i \times \mathtt{C}$ for some i).

J3. [Loop on partitions.] For all $0 \leq i < 2^\ell$, do steps J4–J6 (this finds solutions in $\mathtt{A}_i \times \mathtt{B}_i \times \mathtt{C}$), then stop.

J4. [Build H'.] Let H' be a fresh, empty hash table. For all $x \in \mathtt{A}_i$, store the binding $x[\ell..k] \mapsto x$ in H'.

J5. [Probe H'.] For all $y \in \mathtt{B}_i$, do: retrieve all x's bound to the key $y[\ell..k]$ in H'. For each one, run step J6.

J6. [Probe H.] Set $z \leftarrow x \oplus y$; if $z \in H$, then emit (x, y, z). ∎

Figure 3: Algorithm used to solve the actual 3XOR sub-problems.

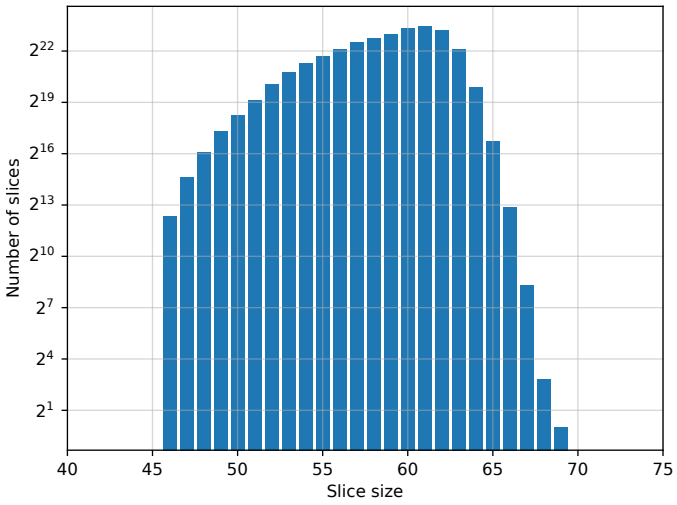

Figure 4: Slice sizes after running algorithm S for 4 hours on each subproblem.

trick" [KADF70] which consists in precomputing all possible linear combinations of subsets of the rows of M.

Given M, we precompute 8 tables of 256 entries, where the i-th table contains all the possible linear combinations of the rows of $M[8i : 8(i + 1)]$, after which the vector-matrix product xM 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 running `gemv` in parallel, we obtain 0.8 instructions per cycle, and with the whole 4 threads we reach 0.95 instructions per cycle. The matrix multiplication step is thus CPU-bound.

Doing the vector-matrix product requires 34 cycles, so the throughput is 46.6 millions vector-matrix products per second on a single core of a PowerPC A2 at 1.6Ghz. Amongst these 34 cycles, 28 instructions seem mandatory: 8 loads, 8 XORs, 8 combined rotations/mask (`rlwinm`) and 4 additional shifts (`rldicl`). In addition, some bookkeeping operations are necessary to perform this in a loop. The `xlc` compiler thus did a fairly good job, and it seemed unlikely that hand-written assembly code would have been much better.

5.6.2. *Partitioning (step J2)*

This operations essentially consists in moving data around in memory, and it is memory-bound. This is essentially equivalent to doing one pass of radix-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;
}
```

The `count` array must be correctly initialized before this loop is run. In general, this requires an additional counting pass beforehand, but in our case we could avoid it entirely by exploiting the fact that the input lists are cryptographic hashes, and therefore they are fairly "random". If the N input items are randomly dispatched into 2^ℓ partitions, then the number of items in each partition is a random variable X following a binomial distribution with mean $\mu := N/2^\ell$. The probability that X deviates far away from its mean is extremely small. More precisely, a standard Chernoff-type bound states that:

$$\Pr(X \geq (1+\delta)\mu) \leq e^{-\delta^2\mu/3}, \qquad 0 \leq \delta \leq 1.$$

This guarantees that $X > \mu + \sqrt{210\mu}$ only happens with probability smaller than 2^{-100}. Instead of performing a counting pass to allocate space and set the `count` array correctly, we simply over-allocate each partition to hold $\mu + \sqrt{210\mu}$ elements and set `count` accordingly. This avoids inspecting the input. The procedure was performed more than 2^{41} times, and it never failed.

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 [BATÖ13]) 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 L1 cache. However, we are compelled to use the largest possible ℓ because this speeds up algorithm J by reducing the number of times step J6 is executed.

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. These threads write in distinct locations inside the `scratch` array, and therefore do not need to be explicitly synchronized.

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 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` arrays 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 $\mathtt{A}_i$ and $\mathtt{B}_i$ in algorithm J are 1024 times smaller than the input subproblems `A` and `B`, on average. By choosing the size of the input subproblems (i.e. by choosing p), we can tune the expected size of partitions in algorithm J.

5.6.3. *Join Computations (steps J4–J6)*

Techniques for efficiently performing database joins have been abundantly studied. Some of these techniques can be used in our setting, most notably the idea of using a partitioned hash join. We just discussed the partitioning step, and it now remains to discuss the actual joins.

This involves building and probing two hash tables H and H' each containing 64-bit integers. We considered cuckoo hash tables [PR01] and classic open-addressing with 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 leads to complications: it may fail and force a full rebuild of the table with new hash functions; using keyed hash functions is slower than using fixed ones.

The hash table containing `C` is built once (in J1) and probed many times (in J6). In addition, `C` is small: it has the size of a slice (see Fig. 4). Therefore, we decided to use a cuckoo hash table and invest more time to build H in order to speed up step J6.

An interesting implementation detail popped up when we tried to optimize our implementation of cuckoo hash tables. Our initial code was the following:

```
/* Does x belong to H? */
static inline bool
cuckoo_lookup(const u64 *H, const u64 x)
{
    const u64 probe1 = H[x & 511];
    const u64 probe2 = H[(x >> 16) & 511];
    return (probe1 == x) || (probe2 == x);
}
```

The semantic of the C language mandates that the right side of the "logical or" shall not be evaluated if the left side is true. We checked that all compilers we could use on the BlueGene/Q (`gcc`, `clang` and `xlc`) compile it using a conditional jump. We tried to get rid of the branch by replacing the "logical or" operator `||` with the "bitwise or" operator `|`, which is not subject to the "early-abort" policy. The resulting compiled code was indeed branchless, and has the same number of instructions but it is ≈ 4 times slower, which seemed counter-intuitive.

We hypothesize that the problem is the following. The semantics of the equality operator mandates that it returns a 0/1 value. Both `gcc` and `xlc` compute `probe1 - x` and check this value against zero, using a clever combination of two PowerPC instructions: `cntlzd` ("count leading zero doubleword") and `rldicl` instructions ("Rotate Left Double Word Immediate then Clear Left"). To summarize, the number of leading zeros in the difference is counted, then divided by 64, which yields the desired 0/1 value. However, on the PowerPC A2 processor, the `cntlzd` instruction is microcoded and thus may perform quite poorly compared to other instructions (its latency is suspiciously not given in the CPU manual). In addition, the conditional branch was reliably not taken, and could therefore be easily predicted. All-in-all, it should have been possible to hand-write better assembly code to probe a cuckoo hash table on this particular CPU, especially by bypassing idiosyncrasies of the C language. In our case however, this represented a small fraction of the total computation time. It therefore did not seem very profitable to invest a large amount of time into optimizing this particular task.

Going back to our join computations, we used linear probing for H' in steps J4–J5 because this hash table must be built quickly and is not probed much. The main parameter affecting performance of linear probing is the fill ratio of the hash table. This in turn depends on partition size (dictated by p and ℓ) and on the size allocated to the each table. Everything must fit in L1 cache, so the number of concurrent hardware threads we want to run also comes into play. With $p = 15$ and $\ell = 10$, the expected size of the partitions $\mathtt{A}_i$ and $\mathtt{B}_i$ are 128 elements.

Using a table H' of 512 entries (4KB with entries of 64 bits) leads to a relatively low fill ratio of 25% full (smaller fill ratios are expected to result in faster hash probes/insertions). We verified experimentally that adjusting the input problem size by using $p + 1$ and $p - 1$ does not lead to better results.

Using hash tables H' with 512 entries means that we have to extract a 9-bit hash from x to insert x in H'. This mandates that $k = \ell + 9$, hence $k = 19$, which is the value we used. Recall that in principle, the optimal value of k was 17. This slightly reduce the size of slices computed by algorithm S.

After a few experiments, we settled for 3 hardware threads, each with its own H'. This consumes 12KB of L1 cache and leaves some leg room for fetching inputs.

In steps J5–J6, two probes to two different hash tables are chained. L1 cache is very scarce in this step. 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 resulted in a 15% speedup.

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

5.6.4. *Summary*

In average, the whole computations (algorithms G and J) runs at 0.85 instructions per cycle, which we found to be satisfactory. When the code runs on 65536 cores at 1.6Ghz, this

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.

makes a not-very-impressive "90TIntOPS" (Integer OPerations). After all, this is aging hardware. It has a peak performance of 839TFLOPS, that we are unable to exploit because we do only integer operations; the peak FLOPS performance comes from the floating-point-only SIMD units of the BlueGene/Q processor — that lack the ability to perform a bitwise XOR between two 256-bit SIMD registers, and therefore are mostly useless to us. All-in-all, the computation required about $2^{64.6}$ CPU cycles in total.

5.7. Comparison with the Quadratic Algorithm

The tuned implementation of algorithm G "solves" a subproblem with input size 2^{17} in about 33.5s on a single BlueGene/Q core. For the sake of completeness, we compared it with our well-optimized implementation of the quadratic algorithm, that we ported to the BlueGene/Q for the occasion. 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 5.3× faster than the quadratic algorithm on the BlueGene/Q — in fact, the quadratic algorithm benefits from integer SIMD instructions available on more conventional CPUs but not on the BlueGene/Q.

5.8. Managing the Computation

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 had to organize the computation of one billion independent subproblems which each require their own input data (roughly 5MB) and run for 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, 30 minutes of computation are lost 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 the computing center to which we had access, submitted jobs must be given 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 S 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.

Because our tasks are highly moldable, we initially considered submitting a large number of small and short tasks to backfill the parallel machine, however the support staff of the computing center advised against it. We therefore used the biggest possible jobs allowed by the computing center, namely 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.

6. 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 33 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 (with the most "colliding bits") are completely described by table 3.

Acknowledgment. We thank Richard Parker for useful discussions. We also thank the support staff at IDRIS, and Alexandra Assanovna Elbakyan for helping scientists to access the works of their colleagues. This work was granted access to the HPC resources of IDRIS under the allocation number 2019-A0060610749 made by GENCI.

bits	FOO		BAR		FOOBAR	
	Counter	x	Counter	x	Counter	x
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 "x" is the 32-bit value appended at the end of a, b and c.

References

[AB12] Martin Albrecht and Gregory Bard. *The M4RI Library – Version 20121224*. The M4RI Team, 2012.

[ABL+13] 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.

[AFS05] Daniel Augot, Matthieu Finiasz, and Nicolas Sendrier. A family of fast syndrome based cryptographic hash functions. In Ed Dawson and Serge Vaudenay, editors, *Progress in Cryptology - Mycrypt 2005, First International Conference on Cryptology in Malaysia, Kuala Lumpur, Malaysia, September 28-30, 2005, Proceedings*, volume 3715 of *Lecture Notes in Computer Science*, pages 64–83. Springer, 2005.

[BATÖ13] 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.

[BCC+13] Charles Bouillaguet, Chen-Mou Cheng, Tung Chou, Ruben Niederhagen, and Bo-Yin Yang. Fast exhaustive search for quadratic systems in $\mathbb{F}_2$ on FPGAs. In Tanja Lange, Kristin E. Lauter, and Petr Lisonek, editors, *Selected Areas in Cryptography - SAC 2013 - 20th International Conference, Burnaby, BC, Canada, August 14-16, 2013, Revised Selected Papers*, volume 8282 of *Lecture Notes in Computer Science*, pages 205–222. Springer, 2013.

[BDF18] 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.

[BLN+09] 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.

[BM] BM1385 datasheet v2.0. Available online. `https://bits.media/images/asic-miner-antminer-s7/BM1385_Datasheet_v2.0.pdf`.

[EM91] Shimon Even and Yishay Mansour. A construction of a cipher from a single pseudorandom permutation. In Hideki Imai, Ronald L. Rivest, and Tsutomu Matsumoto, editors, *Advances in Cryptology - ASIACRYPT '91, International Conference on the Theory and Applications of Cryptology, Fujiyoshida, Japan, November 11-14, 1991, Proceedings*, volume 739 of *Lecture Notes in Computer Science*, pages 210–224. Springer, 1991.

[Fou98] Electronic Frontier Foundation. *Cracking DES: Secrets of Encryption Research, Wiretap Politics & Chip Design*. O'Reilly Series. Electronic Frontier Foundation, 1998.

[GYG12] Jim Guilford, Kirk Yap, and Vinodh Gopal. Fast SHA-256 implementations on intel architecture processors, May 2012. Available online: `https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/sha-256-implementations-paper.pdf`.

[Jou09] Antoine Joux. *Algorithmic cryptanalysis*. CRC Press, 2009.

[KADF70] M Kronrod, V Arlazarov, E Dinic, and I Faradzev. On economic construction of the transitive closure of a direct graph. In *Sov. Math (Doklady)*, volume 11, pages 1209–1210, 1970.

[KPP+06] 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.

[LB88] 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.

[LS19] 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.

[Nan15] Mridul Nandi. Revisiting Security Claims of XLS and COPA. *IACR Cryptology ePrint Archive*, 2015:444, 2015.

[NS14] Ivica Nikolić and Yu Sasaki. Refinements of the k-tree Algorithm for the Generalized Birthday Problem. In *ASIACRYPT*, pages 683–703. Springer, 2014.

[oCoST12] 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.

[oST15] National Institute of Standards and Technology. NIST policy on hash functions, 2015.

[PR01] Rasmus Pagh and Flemming Friche Rodler. Cuckoo hashing. In

European Symposium on Algorithms, pages 121–133. Springer, 2001.

[Rog06] Phillip Rogaway. Formalizing human ignorance. In Phong Q. Nguyen, editor, *Progressin Cryptology - VIETCRYPT 2006, First International Conference on Cryptology in Vietnam, Hanoi, Vietnam, September 25-28, 2006, Revised Selected Papers*, volume 4341 of *Lecture Notes in Computer Science*, pages 211–228. Springer, 2006.

[SBK+17] 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.

[Ste13] Marc Stevens. Counter-cryptanalysis. In Ran Canetti and Juan A. Garay, editors, *Advances in Cryptology – CRYPTO 2013*, pages 129–146, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[str] Stratum pool mining protocol. Available online. `https://slushpool.com/help/stratum-protocol`.

[Sus] Martin Sustrik. nanomsg. `https://nanomsg.org`.

[vOW99] Paul C. van Oorschot and Michael J. Wiener. Parallel collision search with cryptanalytic applications. *J. Cryptology*, 12(1):1–28, 1999.

[Wag02] David Wagner. A generalized birthday problem. In *CRYPTO*, pages 288–304, 2002.

[WY05] Xiaoyun Wang and Hongbo Yu. How to break MD5 and other hash functions. In Ronald Cramer, editor, *Advances in Cryptology - EUROCRYPT 2005, 24th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Aarhus, Denmark, May 22-26, 2005, Proceedings*, volume 3494 of *Lecture Notes in Computer Science*, pages 19–35. Springer, 2005.

Appendix 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()))
```