IACR Transactions on Symmetric Cryptology
ISSN 2519-173X, Vol. 2018, No. 1, pp. 254–276. DOI:10.13154/tosc.v2018.i1.254-276

Revisiting and Improving Algorithms for the 3XOR Problem

Charles Bouillaguet[1], Claire Delaplace[1,2] and Pierre-Alain Fouque[2]

[1] University of Lille-1, Lille, France
charles.bouillaguet@univ-lille1.fr
[2] Univ Rennes 1, Centre National de la Recherche Scientifique (CNRS) , Institut de Recherche en Informatique et Systèmes Aléatoires (IRISA), Rennes, France
{claire.delaplace,pierre-alain.fouque}@irisa.fr

Abstract. The 3SUM problem is a well-known problem in computer science and many geometric problems have been reduced to it. We study the 3XOR variant which is more cryptologically relevant. In this problem, the attacker is given black-box access to three random functions F, G and H and she has to find three inputs x, y and z such that $F(x) \oplus G(y) \oplus H(z) = 0$. The 3XOR problem is a difficult case of the more-general k-list birthday problem.

Wagner's celebrated k-list birthday algorithm, and the ones inspired by it, work by querying the functions more than strictly necessary from an information-theoretic point of view. This gives some leeway to target a solution of a specific form, at the expense of processing a huge amount of data.

However, to handle such a huge amount of data can be very difficult in practice. This is why we first restricted our attention to solving the 3XOR problem for which the total number of queries to F, G and H is minimal. If they are n-bit random functions, it is possible to solve the problem with roughly $\mathcal{O}\big(2^{n/3}\big)$ queries. In this setting, the folklore quadratic algorithm finds a solution after $\mathcal{O}\big(2^{2n/3}\big)$ operations. We present a 3XOR algorithm that generalizes an idea of Joux, with complexity $\mathcal{O}\big(2^{2n/3}/n\big)$ in times and $\mathcal{O}\big(2^{n/3}\big)$ in space. This algorithm is practical: it is up to $3\times$ faster than the quadratic algorithm. Furthermore, using Bernstein's "clamping trick", we show that it is possible to adapt this algorithm to any number of queries, so that it will always be at least as good as, if not better than, Wagner's descendants in the same settings.

We also revisit a 3SUM algorithm by Baran-Demaine-Pătraşcu which is asymptotically $n^2/\log^2 n$ times faster than the quadratic algorithm when adapted to the 3XOR problem, but is otherwise completely impractical.

To gain a deeper understanding of these problems, we embarked on a project to solve actual 3XOR instances for the SHA256 hash function. We believe that this was very beneficial and we present practical remarks, along with a 96-bit 3XOR for SHA256.

Keywords: 3XOR problem · Wagner's algorithm · generalized birthday

1 Introduction

The *birthday problem* is a widely used cryptanalytical tool : given two lists $\mathcal{L}_1, \mathcal{L}_2$ of bitstrings drawn uniformly at random from $\{0,1\}^n$, find $x_1 \in \mathcal{L}_1$ and $x_2 \in \mathcal{L}_2$ such that $x_1 \oplus x_2 = 0$ (the $\oplus$ notation denotes the bitwise exclusive-or operation). This problem is well understood: a solution exists with constant probability as soon as $|\mathcal{L}_1| \times |\mathcal{L}_2| \gg 2^n$ holds, and it can be found in $\mathcal{O}\big(2^{n/2}\big)$ time by simple algorithms (sorting then scanning $\mathcal{L}_1$ and $\mathcal{L}_2$ is a possibility).

Received: 2017-11-23, Accepted: 2018-01-23, Published: 2018-03-01

Wagner studied in [Wag02], the k-XOR problem and showed that for the 4-XOR problem if we increase the size of the lists from $2^{n/4}$, which is the minimal size for having a solution with good probability, to $2^{n/3}$, then the number of solutions increases to $2^{n/3}$ and the problem becomes easier since we do not have to find a needle in a haystack but find one out of many needles. He showed that it is possible to find one solution in time $\mathcal{O}\big(2^{n/3}\big)$, by searching for a solution that satisfy some property.

In this paper, we are concerned with the *3XOR problem* in which there are three lists containing arbitrarily many random bit strings, and where the goal is to find $(x_1, x_2, x_3) \in \mathcal{L}_1 \times \mathcal{L}_2 \times \mathcal{L}_3$ such that $x_1 \oplus x_2 \oplus x_3 = 0$. In this case, the idea behind Wagner's algorithm does not pay off, and does not yield anything better than the following folklore quadratic algorithm : for all pairs (a, b) in $\mathcal{L}_1 \times \mathcal{L}_2$, check if $a \oplus b \in \mathcal{L}_3$.

The (decisional) 3SUM problem over $(\mathbb{Z}, +)$ with three lists of size N can be solved by the same simple quadratic algorithm. 3SUM has been widely studied because many geometric problems can be reduced to it in subquadratic time, and thus are said to be 3SUM-hard [GO95]. Testing whether a set of points in the plane contains three collinear points is a notable example thereof. A $\Omega\left(N^2\right)$ lower-bounds is known when the input numbers are arbitrary reals [Eri95]. It has also been shown that a $\mathcal{O}\big(N^{2-\epsilon}\big)$ algorithm for the 3XOR problem would imply a faster-than-expected algorithm for listing triangles in a graph [JV13, Vio12].

Not only is the 3XOR problem interesting for searching parity check relations in fast correlation attacks with $k = 3$ – Chose, Joux and Mitton [CJM02] could only take $k \geq 4$ – but it also has important applications in cryptanalysis of some authenticated encryption scheme. Nandi [Nan15] exhibited a forgery attack against the COPA mode of operation for authenticated encryption requiring only $2^{n/3}$ encryption queries and about $2^{2n/3}$ time. This attack works by reducing the problem of forging a valid ciphertext to that of solving an instance of the 3XOR problem. This attack was later refined by Nikolić and Sasaki [NS14], using an improved 3XOR algorithm, to $2^{n/2-\varepsilon}$ queries and $\mathcal{O}\big(2^{n/2-\varepsilon}\big)$ operations, for a small value of ε.

Nikolić and Sasaki's algorithm is a variant of Wagner's algorithm: they reduced $\mathcal{L}_3$ to the elements that have the most frequently-occurring pattern in the ℓ first bits, with $\ell \simeq n/2$. Using a probabilistic analysis, notably the well-known maximal bin load when N balls are randomly thrown into N bins [Mit96], they obtained a speedup of $\sqrt{(n/2)/\ln(n/2)}$ compared to Wagner's algorithm.

However, Joux had already proposed a better algorithm five years before Nikolić and Sasaki in [Jou09, §8.3.3.1], which is the best algorithm for the 3XOR problem to this day. The idea is to set $\mathcal{L}_i' := \{x\mathbf{M},\ x \in L_i\}$ where $\mathbf{M}$ is a well-chosen invertible matrix over $\mathbb{F}_2$, and then solve the 3XOR problem over $\mathcal{L}_1', \mathcal{L}_2'$ and $\mathcal{L}_3'$. This results in a solution of the original instance, but the change of coordinates $\mathbf{M}$ allows a few tricks. Joux's idea was to choose $\mathbf{M}$ such that $n/2$ vectors of $\mathcal{L}_3'$ have their last $n/2$ bits equal to 0, compute $\mathcal{L}_{12}' = \mathcal{L}_1' \bowtie_{n/2} \mathcal{L}_2'$, the list of the XOR of pairs $\mathcal{L}_1'$ and $\mathcal{L}_2'$ that match on the $n/2$ last bits, and check for collisions between $\mathcal{L}_{12}'$ and $\mathcal{L}_3'$. With correctly chosen list sizes, this yields a $\sqrt{n/2}$ speedup over Wagner's algorithm. It seems very hard to combine these two improvements in a new algorithm.

Independently, Bernstein [Ber07] proposed a simple data-memory tradeoff that he called "clamping". Adapted to our context, the idea is that when more queries than actually needed are performed, we may choose to keep only the entries that are zeroes on the first ℓ bits, with ℓ chosen so that the product of the sizes of the three lists would be exactly $2^{n-\ell}$. This reduces the dimension of the problem, along with the amount of data that has to be stored.

Moving back in time again, four years before Joux, an algorithm for the 3SUM problem (over the integers) had been described by Baran, Demaine and Patrascu in [BDP05]. This algorithm, which was the first generic subquadratic algorithm for the 3SUM problem, has a better complexity than all the previously mentioned ones, and it can be applied *mutatis mutandis* to the 3XOR problem with lists of size $2^{n/3}$, yielding an expected speedup of $n^2/\log^2 n$ compared to Wagner's algorithm. On the other hand, the FFT algorithm for 3SUM over the integers given as an exercice in [CLRS01, §30.1-4] cannot easily be adapted to the 3XOR problem.

Our contributions. We first give a formal definition of the computational model that we consider, which is slightly non-standard... but was apparently adopted by nearly all the previous literature on generalized birthday algorithms [Wag02, NS14]. We recall several known results about sorting, hashing, etc. which are sometimes implicitly used by previous work.

Armed with these tools, we present a new algorithm that generalizes Joux's algorithm to input lists with arbitrary size. Joux first described the technique consisting in multiplying the lists by a matrix to force the existence of specific solutions. He only applied it once, which forces the lists $\mathcal{L}_1$ and $\mathcal{L}_2$ to be of size $2^{n/2-\epsilon}$. Our generalization of the technique consists in re-iterating this method several times. It can then work with input lists of any sizes.

Basically, we select a subset of entries z in the third list, and find an invertible matrix M such that zM starts by a fixed number k of zeroes, for all z in this set. Then we search for pairs of elements (x, y) in the two other lists such that $(x \oplus y)M$ starts by k zeroes. For each such pairs, we search if $x \oplus y$ matches an element z of the selected subset. We iterate the procedure all over again until all entries of the third list have been considered. This procedure is always $\mathcal{O}(n)$ times faster than the quadratic algorithm. We also propose a constant-factor improvement of this procedure that reduces the number of iterations that have to be performed. Moreover, we show that, when combined with "clamping", the resulting algorithm is faster than Nikolić and Sasaki's, while requiring less storage and less queries.

We implemented this new algorithm and ran it with input vectors of size $n = 96$ on a 64-bit machine. We used the following strategy: First find all solutions to the 3XOR problem on the first 64 bits of each entries, using our algorithm, then for all triplets of partial solutions thus found, check the last 32 bits of their XOR. In the same conditions, our algorithm runs about 2 and 3 times faster than the quadratic algorithm.

We also present an adaptation of the BDP algorithm that we tuned for the 3XOR problem. We show that this algorithm is $\mathcal{O}\big(n^2/\log^2 n\big)$ faster than the quadratic algorithm, when n is asymptotically large. However, it cannot be used in practice on reasonably small values of n and is mostly of theoretical interest.

We raise several practical considerations that are not really discussed in previous work. For instance, the algorithms designed in [Jou09, NS14] need at least two of the input lists to have size $2^{n/2-\varepsilon}$. We claim that, in practice, working with input lists whose size is reduced is better. For instance, to solve the 3XOR problem with 96-bit input vectors, Joux's algorithm handles about 0.86 Petabytes of data while Nikolić-Sasaki's algorithm handles about 2.55 Petabytes. In this case, even sorting the lists is not easy. This makes these algorithms quite impractical.

We illustrate our result with the computation of a 96-bit 3XOR on the SHA256 hash function. This only required 384 Megabytes of memory. The code used to perform this computation is available at

https://github.com/cbouilla/3XOR

2 Preliminary

Notations. Given a list $\mathcal{L}$, we denote by $\mathcal{L}_i$ the i-th element of the list, by $\mathcal{L}[i..j]$ the sublist $\mathcal{L}_i, \ldots, \mathcal{L}_{j-1}$ and by $\mathcal{L}[i..]$ the sublist $\mathcal{L}_i, \mathcal{L}_{i+1}, \ldots$ — this is similar to the "slice" notation in Python. We denote by $\mathcal{L}^{[p]}$ the sublist composed by all vectors whose first bits are p (the size of this prefix is usually clear given the context). For a given vector x, $x[i..j]$ denotes the subvector formed by taking the coordinates from i to $j-1$. Given two lists $\mathcal{A}, \mathcal{B}$ and an integer k, we use the following notation to denote the "join" of $\mathcal{A}$ and $\mathcal{B}$ on the first k bits :

$$\mathcal{A} \bowtie_k \mathcal{B} = \{(a,b) \in \mathcal{A} \times \mathcal{B}, \text{ s.t. } (a \oplus b)[0..k] = 00\ldots0\}$$

We denote by L (resp. A, B, C) the size of the list $\mathcal{L}$ (resp. $\mathcal{A}$, $\mathcal{B}$, $\mathcal{C}$). The log function representes logarithm in base 2.

2.1 Computational Model and Assumptions

We consider that all computations take place inside a RAM (Random Access Machine) in which each "memory cell" contains a w-bit word, with $w = \Theta(n)$ — in other words, the machine is "large enough" to accommodate the problem. We assume that the usual arithmetic and bitwise operations on w bits, as well as memory accesses with w-bit addresses, are elementary operations. This computational model, which is implicit in previous work on the generalized birthday paradox, is sometimes called the transdichotomous model [FW93].

We assume that our machine has black-box access to three oracles **A**, **B** and **C**. When queried with an n-bit integer i, the oracle **A** returns the n-bit value $\mathcal{A}_i$ (the i-th element of the list $\mathcal{A}$). The same goes for **B** and **C**. It is understood that, in most cryptographic applications of the 3XOR problem, querying the oracles actually corresponds to evaluating a cryptographic primitive. As such, we assume that the oracles implement random functions.

The machine on which the actual algorithms run is allowed to query the oracles, to store anything in its own memory and to perform elementary operations on w-bit words. An algorithm running on this machine solves the 3XOR problem if it produces a triplet (i, j, k) such that $\mathcal{A}_i \oplus \mathcal{B}_j \oplus \mathcal{C}_k = 0$.

The relevant performance metrics of these algorithms are the amount of memory they need (M bits), the number of elementary operations they perform (T) and the number of queries they make to each oracle.

Most algorithms for the 3XOR problem begin by querying the oracle **A** on consecutive integers $0, 1, \ldots, A-1$ (the same goes for **B** and **C** with respective upper-bounds of B and C) and storing the results in memory. We therefore assume that algorithms start their execution with a "local copy" of the lists, obtained by querying the oracles (except when explicitly stated otherwise).

It is usually simpler in practice to implement algorithms that produce the colliding *values* $(\mathcal{A}_i, \mathcal{B}_j, \mathcal{C}_k)$ instead of the colliding *inputs* (i, j, k). The reason for that is that most algorithms sort at least one of the lists, so that $\mathcal{L}_i$ is not longer at index i in the array holding $\mathcal{L}$. It would be possible to store pairs $(\mathcal{A}_i, i)$ in memory, sorting on $\mathcal{A}_i$ and retaining the association with i at the expense of an increased memory consumption. In practice it is much simpler to find the colliding values, then re-query the oracles again to find the corresponding inputs, at the expense of doubling the total number of queries.

To avoid degenerate cases, we assume that the number of queries to the oracles are exponentially smaller than 2^n. More precisely, we assume that there is a constant $\varepsilon > 0$ such that $\max\{A, B, C\} < 2^{(1-\varepsilon)n}$.

2.2 Algorithmic Tools

Algorithms for the 3XOR problem process exponentially-long lists of random n-bit vectors. In this section, we review the algorithmic techniques needed to perform three reoccurring operations on such lists: sorting, testing membership and right-multiplication by an $n \times n$ matrix. We recall that these operations can be performed in linear time.

Membership testing. The simplest algorithm for the 3XOR problem works by considering all pairs $(a, b) \in \mathcal{A} \times \mathcal{B}$ and checking if $a \oplus b \in \mathcal{C}$. This is guaranteed to find all solutions.

It has long been known that hashing allows for dictionaries with $\mathcal{O}(1)$ access-time and constant multiplicative space overhead [FKS84]. This is enough to make the quadratic algorithm run in time $\mathcal{O}(AB + C)$. In this case, it is thus beneficial to let $\mathcal{C}$ be the largest of the three lists.

A simple hash table of size $2C$ with *linear probing* allows to test membership with 2.5 probes on average [Knu98, §6.4]. We recall that linear probing is a scheme used to resolve collisions in hash tables. Let us consider a hash table T, with hash function h. To insert an element x in T, check if $T[h(x)]$ is empty. If so, x is inserted at index $h(x)$. If not, probe index $h(x) + 1$, then $h(x) + 2$, and so on, until an empty cell is found. To search for a given x in T, first probe $h(x)$. If $T[h(x)]$ contains something else than x, then probe $T[h(x) + 1]$, and so on, until finding either an empty cell (x does not belong to the table), or a cell whose stored key is x. Because the hashed elements are uniformly random, taking some lowest-significant bits yields a very good hash.

Linear probing is already good, but *Cuckoo Hashing* improves the situation to 2 probes in the worst case [PR01]. In this case two arrays of size C, T_1 and T_2, with two hash functions h_1 and h_2 are considered. If x belongs to the hash table, it is either in $T_1[h_1(x)]$ or in $T_2[h_2(x)]$. To insert an element x in the table, set $T_1[h_1(x)]$ to x. If $T_1[h(x)]$ was not empty, then there was a key y stored in this cell before. The insertion removed y from table T_1, so try to insert it in $T_2[h_2(y)]$. If there was already a key z stored in $T_2[h_2(y)]$, then z is removed, so try to insert it in $T_1[h_1(z)]$, and so on. This process may fail because of cycles, in which case the two arrays have to be rebuilt with new hash functions. Insertion requires an expected constant number of operations. In principle, Cuckoo hashing needs a family of universal hash functions, which is a drawback in our case. We may use the usual universal hash functions $h_{a,b} : x \mapsto (ax + b) \bmod p$, where p is a prime.

We found that Cuckoo hashing was about 2× faster than linear probing. It allows to make the quadratic algorithm quite efficient, with only a small constant hidden in the $\mathcal{O}$ notation: around 10 CPU cycles are enough to process a pair.

Sorting and joining. The evaluation of the "join" $\mathcal{A} \bowtie_k \mathcal{B}$ of two lists $\mathcal{A}$ and $\mathcal{B}$ of size $2^{\alpha n}$ is ubiquitous in generalized birthday algorithms following Wagner's ideas. All these algorithms rely on a common idea: they target a solution $a \oplus b \oplus c = 0$ such that $c[0..k] = 0$ (for some value of k), which implies that $a[0..k] = b[0..k]$. These special solutions can be found efficiently by performing a linear-time join operation between $\mathcal{A}$ and $\mathcal{B}$. The following algorithm is thus the workhorse of these techniques.

Algorithm M (*3XOR with a sort join*). Given three lists $\mathcal{A}, \mathcal{B}$ and $\mathcal{C}$, where all $c \in \mathcal{C}$ are such that $c[0..k] = 0$, returns all triplets $(a, b, c) \in \mathcal{A} \times \mathcal{B} \times \mathcal{C}$ such that $a \oplus b \oplus c = 0$.

```
M1. [Prepare input]     Sort $\mathcal{A}$ and $\mathcal{B}$ according to their first $k$ bits.
                        Initialize a hash table with the entries of $\mathcal{C}$.
M2. [Main loop.]        Set $i \leftarrow 0, j \leftarrow 0$.
                        While $i < A$ and $j < B$ do:
M3. [Compare prefix.]       If $\mathcal{A}_i[0..k] < \mathcal{B}_j[0..k]$ then increment $i$
                            Else if $\mathcal{A}_i[0..k] > \mathcal{B}_j[0..k]$ then increment $j$
                            Else:
M4. [Save prefix.]              Set $p \leftarrow \mathcal{A}_i[0..k]$ and $j_0 \leftarrow j$.
M5. [Check pairs.]              While $i < A$ and $\mathcal{A}_i[0..k] = p$ do:
                                    While $j < B$ and $\mathcal{B}_i[0..k] = p$ do:
                                        If $\mathcal{A}_i \oplus \mathcal{B}_j \in \mathcal{C}$ then
                                            emit $(\mathcal{A}_i, \mathcal{B}_j)$.
                                        Increment $j$.
                                    Increment $i$.   ▮
```

It is well-known that an array of N random k-bit vectors can be sorted in *linear* time [Knu98, §5.2.5]: the randomness of the input allows to beat the $\Omega(N \log N)$ lower-bound of comparison-based sorting. Here is a way to do it using $\mathcal{O}(\sqrt{N})$ extra storage: perform two passes of *radix sort* on $0.5 \log N$ bits, then finish sorting with insertion sort. Morally, for each pass of radix sort, we initialize $\sqrt{N}$ counters corresponding to the possible prefixes, then we scan through the list, pick up the $0.5 \log N$-bit prefix of each vector, increment the corresponding counter. Then the entries can directly be dispatched in the right output bucket.

Each pass of radix sort requires $\sqrt{N}$ words to store the counters. Besides that, the input array can in principle be permuted in-place [Knu98, §5.2.1]. The two passes of radix sort guarantee that the array is sorted according to its first $\log N$ bits. This reduces the expected number of inversions from $\approx N^2/4$ to $\approx N/4$. Thus, the expected running time of the insertion sort will be linear.

This tells us that step M1 runs in time linear in $A + B$. Steps M2–M4 are repeated at most $A + B$ times and require a constant number of operations on w-bit words. The test of step M5 is executed once for each pair $(a, b) \in \mathcal{A} \times \mathcal{B}$ with $a[0..k] = b[0..k]$. The expected number of such pairs is $AB/2^k$. Considering that we also have to initialize a hash table with the elements of C, the total expected running time of the algorithm is thus $\mathcal{O}\big(A + B + C + AB/2^k\big)$.

The algorithms designed by Wagner [Wag02], Nikolić–Sasaki [NS14] and Joux [Jou09] use this procedure but differ in how they filter or modify their inputs to ensure that a solution of the correct form exists with high probability.

Matrix multiplication. Given a $n \times n$ matrix $\mathbf{M}$ over $\mathbb{F}_2$, we need to compute the matrix-matrix product $\mathbf{LM}$, where $\mathbf{L}$ is seen as a $2^{\alpha n} \times n$ matrix. Performing the product naively would require $\mathcal{O}(n 2^{\alpha n})$ operations. This can improved to $\mathcal{O}(2^{\alpha n})$ operations, using a trick similar to the "method of Four Russians" [KADF70].

The idea is to divide $\mathbf{M}$ into slices of $n\frac{\alpha}{1+\varepsilon}$ rows (for any $\varepsilon > 0$); for each slice, we precompute all the $2^{\frac{\alpha}{1+\varepsilon}n}$ linear combinations of the rows and store these in a table. This would take $2^{\frac{\alpha}{1+\varepsilon}n}$ word operations.

Then, the vector-matrix product $x\mathbf{M}$ can then be evaluated by dividing x in slices of size $n\frac{\alpha}{1+\varepsilon}$, looking up the precomputed linear combination of the rows of the corresponding slices of $\mathbf{M}$, and XORing together the $\frac{1+\varepsilon}{\alpha}$ resulting vectors. This step takes only a constant number of operations and we will need to perform it $2^{\alpha \cdot n}$ times. Hence, the whole complexity of the procedure is $\mathcal{O}(2^{\alpha n})$ word operations.

2.3 Clamping

This idea, which is due to Bernstein [Ber07], is that the amount of data can be reduced if we are allowed to perform more queries than strictly required.

We assume that $2^{\alpha n}$ (resp. $2^{\beta n}$, $2^{\gamma n}$) queries are made to oracle $\mathbf{A}$ (resp. $\mathbf{B}$, $\mathbf{C}$), and that $\alpha + \beta + \gamma > 1$. Let us denote $\kappa = (\alpha + \beta + \gamma - 1)/2$. The idea is that when querying the oracles, we may reject values that are not zero on the first κn bits — naturally, this only works when $\kappa \geq \min(\alpha, \beta, \gamma)$.

Doing this "clamping", we come up with three lists $\mathcal{A}, \mathcal{B}, \mathcal{C}$ of $(1 - \kappa)n$-bit entries. We have $A = 2^{(\alpha-\kappa)n}$ $B = 2^{(\beta-\kappa)n}$ and $C = 2^{(\gamma-\kappa)n}$ so that $ABC = 2^{(1-\kappa)n}$

For instance, if $\mathcal{O}\big(2^{n/2}\big)$ queries to the oracles are allowed, then, setting κ to $1/4$, solving the 3XOR problem can be done in $\mathcal{O}\big(2^{n/2}\big)$ operations, but using only $\mathcal{O}\big(2^{n/4}\big)$ words of memory. The three lists $\mathcal{A}$, $\mathcal{B}$ and $\mathcal{C}$ obtaing by clamping contain roughly $2^{n/4}$ entries of $n' = 3 \cdot n/4$ bits. A solution to the 3XOR problem can then be found by any algorithm that is able to recover a solution to the problem over n' bits when the size of the lists is $2^{n'/3}$. Using the quadratic algorithm, it requires about $\mathcal{O}\big(2^{n/2}\big)$ operations.

However, as discussed in section 5, querying the oracles can sometimes require a lot of time. So, in practice, there is a tradeoff between the number of queries and the time/data complexity.

3 Finding 3XOR by Linear Changes of Variables

In this section, we propose a new algorithm which is asymptotically n times faster than the quadratic algorithm. It is a generalization of an earlier idea of Joux [Jou09]. While it is asymptotically inferior to the BDP algorithm discussed later, it is faster in practice than the quadratic algorithm for relevant parameter sizes. On the other hand, the BDP algorithm is not practical.

3.1 Algorithm Description

This algorithm exploits the fact that when $\mathbf{M}$ is an $n \times n$ invertible matrix over $\mathbb{F}_2$, then $a \oplus b \oplus c = 0$ if and only if $a\mathbf{M} \oplus b\mathbf{M} \oplus c\mathbf{M} = 0$. The sets of solutions in $\mathcal{A} \times \mathcal{B} \times \mathcal{C}$ and $\mathcal{A}\mathbf{M} \times \mathcal{B}\mathbf{M} \times \mathcal{C}\mathbf{M}$ are the same.

The idea of our algorithm is to select a slice of $n-k$ vectors from $\mathcal{C}$ and to choose a matrix $\mathbf{M}$ such that the first k entries of the slice become zero. Algorithm M then finds all the triplets $a \oplus b \oplus c = 0$ where c belongs to the slice. Repeating this procedure for all slices yields all the possible solutions.

Algorithm G (*3XOR by linear changes of variables*). Given $\mathcal{A}, \mathcal{B}$ and $\mathcal{C}$, return all $(a, b, c) \in \mathcal{A} \times \mathcal{B} \times \mathcal{C}$ such that $a \oplus b \oplus c = 0$. Let $k \leftarrow \lceil \log_2 \min(A, B) \rceil$.

```
G1. [Main loop.]           For all $0 \le i < C/(n-k)$ do:
G2. [Change of basis.]         Let $u \leftarrow i(n-k)$ and $v \leftarrow \min\{(i+1)(n-k), C\}$.
                               Compute an $n \times n$ matrix $\mathbf{M}$ such that:
                               $\mathcal{C}[u..v]\mathbf{M} = \begin{array}{cl} \begin{pmatrix} 0 & \dots & 0 & \star & \dots & \star \\ \vdots & & \vdots & \vdots & & \vdots \\ 0 & \dots & 0 & \star & \dots & \star \end{pmatrix} & \Big\updownarrow n-k \\ \underset{k}{\longleftrightarrow} \quad \underset{n-k}{\longleftrightarrow} & \end{array}$
G3. [Matrix product.]          Set $\mathcal{A}' \leftarrow \mathcal{A}\mathbf{M}$, $\mathcal{B}' \leftarrow \mathcal{B}\mathbf{M}$ and $\mathcal{C}' \leftarrow \mathcal{C}[u..v]\mathbf{M}$.
G4. [Join]                     Run algorithm M to find all pairs $(a', b')$ in $\mathcal{A}' \times \mathcal{B}'$
                               such that $a' \oplus b' \in \mathcal{C}'$ and $a'[0..k] = b'[0..k]$.
                               For each such pair, emit $\left(a'\mathbf{M}^{-1}, b'\mathbf{M}^{-1}\right)$.  ▮
```

Theorem 1. *Algorithm G finds all 3XORs in $\mathcal{A} \times \mathcal{B} \times \mathcal{C}$ in expected time*

$$T = \mathcal{O}((A + B)C/n)$$

and space linear in $A + B + C$.

Proof. We first observe that the algorithm only needs to store two of copies of the input lists (this could be brought down to a single copy, updated in-place). This establishes the linear space claim. Step G2 can be done in time $\mathcal{O}(n^3)$, for instance using the PLUQ factorization $\mathcal{C}[u..v]^{\mathrm{T}} = \mathbf{PLUQ}$, where $\mathbf{P}$ and $\mathbf{Q}$ are permutation matrices, $\mathbf{L}$ is lower-triangular with unit diagonal and $\mathbf{U}$ is upper-trapezoidal (i.e. all coefficients bellow the diagonal are zeroes). Only the first r rows of $\mathbf{U}$ are non-zero, where r denotes the rank of $\mathcal{C}[u..v]$. It follows that $\mathbf{M} = \mathbf{P}\left(\mathbf{L}^{\mathrm{T}}\right)^{-1}$ is a suitable choice.

Let us assume, without loss of generality, that $A \le B$. Using the techniques described in section 2.2, we know that step G3 costs $\mathcal{O}(A + B + (n-k))$. The expected cost of step

G4 is $\mathcal{O}\big(A + B + AB/2^k\big)$. The choice of k at the beginning of the algorithm ensures that this is in fact $\mathcal{O}(A + B)$.

Steps G2–G4 are repeated $C/(n - \log_2 A)$ times, therefore the total complexity of the algorithm is then

$$\mathcal{O}\left(\frac{(A + B)C}{n - \log_2 A}\right).$$

The hypothesis that $\log_2 A < (1 - \varepsilon)n$ guarantees that the denominator is $\Omega(n)$, and yields the claimed complexity. □

Using algorithm G, it is beneficial to choose C as the smallest of the three lists.

Improvement using the clamping trick. As discussed in section 2.3, it is possible to reduce both the amount of data and the time complexity if more queries are allowed.

If we assume that we can make $2^{\alpha \cdot n}$ queries to each oracle, then using clamping, we can reduce the problem to the one of finding a 3XOR over $(1 - \kappa)n$ bits, with $\kappa = (3\alpha - 1)/2$. The size of the three input lists is then about $2^{1/2 \cdot (1-\alpha) \cdot n}$, and processing them with algorithm G requires $\mathcal{O}\big(2^{(1-\alpha) \cdot n}/n\big)$ operations.

This shows that algorithm G, combined with clamping, is always at least as efficient as Joux's algorithm, and always more efficient that Nikolić-Sasaki's algorithm.

In fact, if $2^{n/2-\epsilon}$ queries to the oracles are allowed, performing the clamping over $k = (n/2 - 3\epsilon)/2$, we will obtain lists of size around $2^{1/2 \cdot (n/2+\epsilon)}$, and processing them with algorithm G will require $\mathcal{O}\big(2^{n/2+\epsilon}/n\big)$ operations. If $\epsilon \simeq 1/2 \cdot \log((n/2)/\ln(n/2))$ as in Nikolić and Sasaki's algorithm, then the time complexity of algorithm G is more or less $\mathcal{O}\Big(2^{n/2}/\sqrt{n \cdot \ln n}\Big)$ instead of the $\mathcal{O}\Big(2^{n/2}/\sqrt{n/\ln n}\Big)$ obtained by Nikolić and Sasaki's algorithm.

3.2 Constant-Factor Improvements

In step G2, an invertible matrix $\mathbf{M}$ is found that sends the first k bits of $n - k$ random vectors to zero. In this section, we discuss means to find matrices that have the same effect on *more* than $n - k$ vectors. The result would be that each iteration of steps G2–G4 would process larger slices of $\mathcal{C}$. As such, less iterations would be needed.

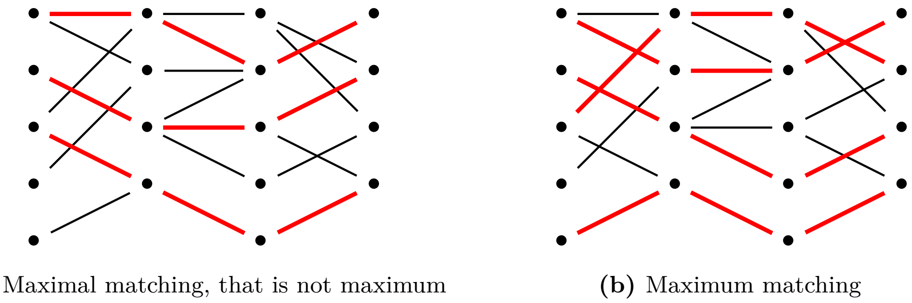
3.2.1 Finding the Coordinate Change

Given a large collection $\mathcal{C}$ of independent and uniformly random n-bit vectors x, we are facing the problem of finding a matrix $\mathbf{M}$ that maximizes (or at least increases) the number of $x \in \mathcal{C}$ such that $x\mathbf{M}[0..k] = 0$. In more algebraic terms, let $\mathtt{V}_0$ be the $(n-k)$-dimensional subspace of $\{0, 1\}^n$ containing all vectors whose first k coordinates are zeroes. The matrix $\mathbf{M}$ should be chosen such that it sends the largest number of input vectors from $\mathcal{C}$ to $\mathtt{V}_0$. Alternatively, let $\mathtt{V}$ be the pre-image of $\mathtt{V}_0$ through $\mathbf{M}$: $x\mathbf{M}$ belongs to $\mathtt{V}_0$ if and only if x belongs to $\mathtt{V}$.

Finding a subspace $\mathtt{V}$ having a large intersection with the input list is the actual difficult task. Indeed, once a basis $(b_1, \ldots, b_{n-k})$ of $\mathtt{V}$ has been found, building an invertible matrix $\mathbf{M}$ that sends $\mathtt{V}$ to $\mathtt{V}_0$ is easy. We therefore face the following computational problem:

Problem 1. *Given a list $\mathcal{C}$ of uniformly random vectors in $\{0, 1\}^n$, find a $(n - k)$ dimensional subspace $\mathtt{V}$ of $\{0, 1\}^n$ such that $\mathtt{V} \cap \mathcal{C}$ is as large as possible.*

In other terms, we are looking for multiple simultaneous linear approximations of the random vectors in $\mathcal{C}$. Note that we have to find not only one, but a large number such subspaces (one per iteration of algorithm G).


(a) Maximal matching, that is not maximum

(b) Maximum matching

Figure 1: A maximal matching is shown in sub-figure 1a. This matching is not maximum. A larger can be found as shown in sub-figure 1b

In the rest of this section, we discuss two ways to tackle this problem. The first approach consists in finding all these subspaces at once, during a pre-computation step, then permuting $\mathcal{C}$ so that for known parameters $N_0, N_1, \ldots$ the first N_0 entries of $\mathcal{C}$ span the first subspace, the following N_1 span the second subspace, and so on.

The second approach is iterative: at each iteration of the procedure, we consider the list $\mathcal{C}_{left}$ of the elements of $\mathcal{C}$ that we have not treated yet, we search a solution to problem 1 with $\mathcal{C}_{left}$ as input list.

In both cases, we want this extra-computation to be much faster than the actual 3XOR algorithm that it is supposed to speed-up. As such, we give up on finding optimal solutions and instead look for heuristics that produce quick results. We propose two approaches to obtain better results.

3.2.2 All-at-Once Approach Using Wagner's Algorithm

The first method we propose precomputes sets of linearly dependent vectors in $\mathcal{C}$, and then permutes the list, so these linearly dependent vectors are stacked together. In the interesting case where $C = 2^{n/2}$, this enables us to reduce the number of iterations in algorithm G by 25% in exchange for a one-time pre-computation of negligible complexity. We discuss a situation where this happens naturally in section 5.3.

Wagner's celebrated 4-tree algorithm can be used to find quadruplets of distinct indices (x, y, z, t) such that $\mathcal{C}_x \oplus \mathcal{C}_y \oplus \mathcal{C}_z \oplus \mathcal{C}_t = 0$. These four entries of $\mathcal{C}$ are then linearly dependent.

The main idea of this procedure is to find all pairs $x < y$ (resp. $z < t$) such that $\mathcal{C}_x \neq \mathcal{C}_y$ and $\mathcal{C}_z \oplus \mathcal{C}_t$ starts with k zeroes. The expected number of such pairs is $\mathcal{O}\big(2^k\big)$. Then, the expected number of distinct couples $(x, y) < (z, t)$, such that $\mathcal{C}_x \oplus \mathcal{C}_y \oplus \mathcal{C}_z \oplus \mathcal{C}_t = 0$ is $\mathcal{O}\big(2^{3k-n}\big)$. For instance, if $C = 2^{n/2}$, we are in the interesting case where we will find about $\mathcal{O}\big(2^{n/2}\big)$ such quadruplets.

We then need to isolate a (large) subset of pairwise disjoint quadruplets. Finding the biggest possible subset is the 4D MATCHING problem. Given four finite disjoint sets X, Y, Z, T. Let S be a subset of $X \times Y \times Z \times T$, and let $M \subseteq S$ such that if $(x_1, y_1, z_1, t_1) \in M$ and $(x_2, y_2, z_2, t_2) \in M$ then $x_1 \neq x_2$, $y_1 \neq y_2$, $z_1 \neq z_2$, $t_1 \neq t_2$.

M is said to be *maximal* if there is no matching M' such that $M \subsetneq M' \subseteq S$.

M is said to be *maximum* if there is no matching $M' \subseteq S$ such that $|M'| > |M|$. These definition are illustrated by figure 1.

Finding a *maximum* 4D matching is NP-complete. However, any *maximal* 4D matching is a 4-approximation to a maximum 4D matching, and it can be found efficiently by a greedy algorithm. Let $\mathcal{M}$ denote a 4D matching, and r denote its cardinality.

We build a permuted list $\mathcal{D}$ as follows: start from an empty list; for each $\{x, y, z, t\} \in \mathcal{M}$, append $\mathcal{C}_x, \mathcal{C}_y, \mathcal{C}_z$ and $\mathcal{C}_t$ to $\mathcal{D}$. Finally, append $\mathcal{C} - \mathcal{D}$ to $\mathcal{D}$, in any order. Algorithm G can be updated to exploit $\mathcal{D}$, with a running time reduced by 25%.

Algorithm G' (*3XOR by linear change of variables with 4SUM precomputation*). Given $\mathcal{A}, \mathcal{B}$ and $\mathcal{D}$ (as computed above), return all $(a, b, d) \in \mathcal{A} \times \mathcal{B} \times \mathcal{D}$ such that $a \oplus b \oplus d = 0$. Let $k \leftarrow \lceil \log_2 \min(A, B) \rceil$.

```
G'1. [Main loop.]          For all $0 \leq i < 3r/(n-k)$ do:
G'2. [Change of basis]         Set $u \leftarrow 4i(n-k)/3$ and $v \leftarrow 4(i+1)(n-k)/3$
                               (note that $\mathcal{D}[u..v]$ has rank $n-k$ even though it has
                               $4(n-k)/3$ rows). Compute an $n \times n$ matrix $\mathbf{M}$ such
                               that
                               $$\mathcal{D}[u..v]\mathbf{M} = \begin{array}{c}\left(\begin{array}{cccccc} 0 & \dots & 0 & \star & \dots & \star \\ \vdots & & \vdots & \vdots & & \vdots \\ 0 & \dots & 0 & \star & \dots & \star \end{array}\right) \\ \overset{\longleftrightarrow}{k} \qquad \overset{\longleftrightarrow}{n-k} \end{array} \Big\updownarrow \tfrac{4}{3}(n-k)$$
G'3. [Matrix product.]         Set $\mathcal{A}' \leftarrow \mathcal{A}\mathbf{M}$, $\mathcal{B}' \leftarrow \mathcal{B}\mathbf{M}$ and $\mathcal{D}' \leftarrow \mathcal{D}[u..v]\mathbf{M}$.
G'4. [Join]                    Run algorithm M to find all pairs $(a', b')$ in $\mathcal{A}' \times \mathcal{B}'$
                               such that $a'[0..k] = b'[0..k]$ and $a' \oplus b' \in \mathcal{D}'$.
                               For each such triplet, emit $\big(a'\mathbf{M}^{-1}, b'\mathbf{M}^{-1}\big)$.
G'5. [Finish]              Run algorithm G with input lists $\mathcal{A}, \mathcal{B}$ and $\mathcal{D}[4r..]$. ▮
```

3.2.3 One-at-a-Time Approach Using Decoding Algorithms

The second method we present is an iterative way of finding these vectorial subspaces. We discuss how to find *one* hopefully large sublist of the initial list $\mathcal{C}$, that spans a $(n-k)$ dimensional subspace, knowing that this procedure will have to be repeated many times with decreasing list sizes.

Recall that an $(n-k)$-dimensional subspace of $\{0, 1\}^n$ is defined by a system of k linear equations in n variables. Finding a vector space $\mathtt{V}$ having a large intersection with $\mathcal{C}$ amounts to finding k linear equations that are simultaneously biased over $\mathcal{C}$ (i.e. more often simultaneously true than the contrary).

We propose a greedy approach to find these equations: first initialize a list $\overline{\mathcal{C}}$ to $\mathcal{C}$, find a biased equation E_1 over $\overline{\mathcal{C}}$, then remove from $\overline{\mathcal{C}}$ all vectors that do not satisfy E_1, and re-iterate the method $k-1$ times. This reduces the problem to that of finding a single biased equation over $\mathcal{C}$.

Finding *the* most biased equation over $\overline{\mathcal{C}}$ is tempting but too expensive: an exhaustive search would require $\mathcal{O}(2^{n+k})$ operations and FFT-like methods such as the Walsh transform still have a workload of order $\mathcal{O}(k \cdot 2^n)$. We have to settle for a "good", if not optimal bias.

Finding a biased linear equation over $\mathcal{C}$ is a *decoding problem.* Consider the binary linear code spanned by the columns of $\mathcal{C}$, and let y be a low-weight codeword: there is a vector x such that $y = \mathcal{C}x$ and y has a low Hamming weight (amongst all possible vectors y). This means that x describes the coefficient of one of the most biased linear equation over $\mathcal{C}$.

Information Set Decoding. A *linear binary (error-correcting) code* of *length* n and dimension d is a subspace of $\mathbb{F}_2^n$ of dimension d. It contains 2^d codewords of n bits. A linear code can be seen as the linear span of the rows of an $d \times n$ matrix G called the *generator matrix* of the code. The *weight* of a codeword is its Hamming weight. The *minimum-distance* of the code is the minimum weight of non-zero codewords. Determining the minimum-distance of a linear code is NP-complete in general. The first algorithm for the *minimum-weight codeword* problem has been first introduced by McEliece [McE78] in the security analysis of his cryptosystem, and has been widely studied since. One of the simplest way to recover a minimum-weight codeword is to use the Las Vegas randomized *Information Set Decoding* (ISD) procedure. Although this algorithm has been improved several times [BJMM12, MMT11, MO15], we found out that in our particular case, the Lee-Brickell algorithm [LB88] is nearly as efficient as its later optimizations, while being easier to implement.

Let $\mathtt{C}$ be a random binary linear code of dimension k and length N. We denote by d the minimum distance of $\mathtt{C}$ and by $\delta := d/N$ its relative distance. Let $R := k/N$ be the rate of $\mathtt{C}$, and let $\mathcal{H}$ denote the binary entropy function. The time complexity of the Lee-Brickell algorithm has been analyzed in [CG90] and is about $\mathcal{O}\big(2^{N \cdot F(R)}\big)$ with:

$$F(R) := (1 - R) \cdot (1 - \mathcal{H}(\delta/(1 - R))). \tag{1}$$

As $\mathtt{C}$ is a random code, its minimum distance d is close to the *Gilbert-Varshamov bound* [Gil52, Var57], and thus we assume that $\mathcal{H}(\delta) = 1 - R$.

The function F reaches a maximum of 0.1207 when $R \simeq 0.454$. The further improvements of the original Lee-Brickell algorithm mostly aim at reducing its worst case complexity. However, in our case, we will have to find minimum-weight words in codes of very low rate. As such the values of $F(R)$ we encounter are much smaller.

Using ISD, one can find codewords c of Hamming weight $wt(c) = d$, where d is the minimum distance of the code. However, the complexity of such algorithms is exponential in the length of the code (the number of vectors in the input list), and therefore, considering our time budget of $\mathcal{O}(C)$, we have to set an upper bound N_{max} on the size of the list that we consider during the procedure. The value of N_{max} is determined below.

Hence, before using decoding algorithms, we have to reduce the size of the input list below N_{max}. To this end, we focus the search on the vectors of $\mathcal{C}$ whose $\lceil \log C/N_{max} \rceil$ first bits are zero: their expected number is approximately N_{max}, and they already satisfy $\lceil \log C/N_{max} \rceil$ linear equations...

Procedure description. Let N_{max} and ε be fixed parameters such that N_{max} represents an upper bound on the length of the initial code $\mathtt{C}_0$, and threshold parameter ε. The following procedure finds a set of k biased equations:

Algorithm F (*Greedily find simultaneous linear approximations*). Given a list $\mathcal{C}$ (sorted in ascending order), two parameters N_{max} and ε, return a (hopefully large) sublist $\overline{\mathcal{C}}$ such that all entries of $\overline{\mathcal{C}}$ satisfy k linear equations.

```
F1. [Initialization.]      Set $\ell \leftarrow \lceil \log C/N_{max} \rceil$.
                           Let $j$ be the biggest integer such that $\mathcal{C}_j[0..\ell] = 0$.
                           Set $\overline{\mathcal{C}} \leftarrow \mathcal{C}[0..j]$ and $t \leftarrow \ell$.
    ($t$ is the number of equations currently found).
F2. [Main loop.]           While $t < k$ do:
F3. [Find equation.]           Estimate $d$, the minimum distance of the code C
                               spanned by $\overline{\mathcal{C}}^T$ (i.e. the matrix whose columns are
                               the entries of $\overline{\mathcal{C}}$).
                               Run the Lee-Brickell ISD algorithm to find a non-zero
                               codeword $c$ in C such that $wt(c) \leq d + \varepsilon$.
                               Increment $t$.
F4. [Filter.]                  For all $i$ such that $c_i = 1$ do:
                                   remove $\overline{\mathcal{C}}_i$ from $\overline{\mathcal{C}}$.
```

Theorem 2. *Algorithm F returns a subset of $\mathcal{C}$ that spans a $(n - k)$-dimensional subspace of $\{0, 1\}^n$.*

Proof. We prove the following invariant: each time step F3 begins, $\overline{\mathcal{C}}$ is contained inside a subspace E_t of dimension $n-t$.

At the end of step F1, $\overline{\mathcal{C}}$ belongs to the subspace E_ℓ composed by all the vectors whose first ℓ coordinates are zeroes, so the invariant holds the first time step F3 is performed.

Because c is a codeword that belongs to the code spanned by the columns of $\overline{\mathcal{C}}$, there exists x such that $c = \overline{\mathcal{C}}x$. Step F4 removes from $\overline{\mathcal{C}}$ the vectors that are not orthogonal to x. It follows that after step F4, $\overline{\mathcal{C}}$ is contained into the subspace $E_{t+1} = E_t \cap \{0, x\}^\perp$. Because c is non-zero, at least one vector is removed from the list. This vector belonged to E_t, and does not belong to E_{t+1}. As such, the dimension of E_{t+1} is (at least) one less than that of E_t. □

Estimation of N_{max}. Knowing C the size of the initial list $\mathcal{C}$ it is possible to estimate the upper bound N_{max} over the length of the code that the ISD algorithm can process within our time budget.

Let us denote by f the function $f(x, y) := x \cdot F(y/x)$ (for $0 < y < x$), where F is the function given by equation 1. A minimum-weight codeword in a code $\mathtt{C}$ of length N_{max} and dimension $n-\ell$ can be found in time $\mathcal{O}\big(2^{f(N_{max}, n-\ell)}\big)$.

If we denote by T_{ISD}, the total time spent in the $k-\ell$ iterations of step F3, we claim that

$$T_{ISD} \leq (k-\ell) \cdot 2^{f(N_{max}, n-\ell)}. \tag{2}$$

The function $f(x, y)$ grows with x, and also grows with y when $y < 0.1207x$. This is enough to prove inequality (2), assuming that N_{max} is much larger than $n-\ell$.

To be sure that the complexity of algorithm F does not exceed $\mathcal{O}(C)$, we want to find the maximum value of N_{max} so that $T_{ISD} < C$. This can be done numerically (for instance using the bisection algorithm). Table 1 gives an estimation of the parameters for some values of n and k. With k a little lower than $n/2$, we can hope to find at least n vectors of $\mathcal{C}$ in the first subspace (this is to be compared with $n/2$ without this technique).

4 Adaptation of BDP Algorithm

In this section, we revisit an algorithm that was introduced by Baran, Demaine and Pătraşcu (BPD) in [BDP05]. Initially this algorithm was designed for the 3SUM problem over $(\mathbb{Z}, +)$, where the size of the three lists is bounded by some parameter N. Their goal was to determine whether a 3SUM exists in the input lists or not and to return it, when appropriate. Their algorithm is subquadratic in N.

The adaptation described here is asymptotically $n/\log^2 n$ times faster than the algorithm we present in section 3, but is hardly practical. We consider three lists $\mathcal{A}$, $\mathcal{B}$ and $\mathcal{C}$ of size $2^{n/3}$ (this can be generalized to different size of input lists). We recall that $\mathcal{A}^{[i]}$ represents the sublist of $\mathcal{A}$ that starts with the prefix i, where i is a bit-string. The high level idea is the following:

1. Dispatch $\mathcal{A}$, $\mathcal{B}$ and $\mathcal{C}$ in buckets according to their first k bits, and let m be the expected number of elements in each bucket ($m = 2^{n/3-k}$).

Table 1: Parameter estimation of the ISD-based algorithm for some value of n and k

n	k	expected N_0	expected $\overline{C}$
64	28	382	81
128	60	1996	154
256	123	9894	285
512	252	60271	545

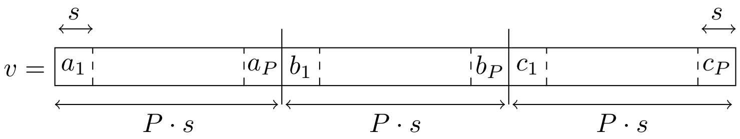


Figure 2: Representation of an index v of the table T

2. For each triplets of buckets $(\mathcal{A}^{[i]}, \mathcal{B}^{[j]}, \mathcal{C}^{[i\oplus j]})$, perform a preliminary *constant time* test. The test returns *false* when it is certain that no solution will be found in this triplet.

3. For each triplet that has not been rejected, use the quadratic algorithm on this reduced instance.

A full description of the procedure is given in appendix A.

Basically for each triplet of bucket, this test consists in checking if there is a partial collision on s bit, with s a well chosen parameter. To do that, we have previously precomputed a table T. Each triplet corresponds to an index v of T. If $T[v] = 1$ there is a partial collision. If $T[v] = 0$, then, the test fails, and we know for sure there is no collision. We describe this test in more details bellow.

4.1 Preliminary Test

Let $w = \Theta(n)$ denote the size of a word. Let s be such that $s = \kappa_1 \cdot \log(w)$, with κ_1 a constant to be determined. Let $k < n/3$ be the parameter defined above, and $P \geq 1$ a parameter to be determined, such that $3 \cdot P \cdot s \leq w$.

Let T be a table of $2^{3 \cdot P \cdot s}$ elements. For each index v, that can be represented by figure 2,

$$T[v] = \begin{cases} 1 & \text{if } \exists\ i, j, l \text{ s.t. } a_i \oplus b_j \oplus c_l = 0 \\ 0 & \text{otherwise.} \end{cases} \tag{3}$$

Then, for all three sets of s-bit vectors, that contain each at most P elements, one can check if a solution to the 3XOR problem exists by a constant look-up in T.

By construction, $|T| = 2^{3 \cdot P \cdot s}$. We do not want the additional space to exceed the space required to store the lists. We set:

$$3 \cdot P \cdot s = \min(1/3 \cdot n, w). \tag{4}$$

For all $x \in \mathbb{F}_2^n$, let h be the function that returns s consecutive bits of x starting from bit k, or formally $h : x \to x[k..k+s]$. For all $0 \leq \ell < 2^k$ and we denote by $h(\mathcal{A}^{[\ell]})$ the list

$$h(\mathcal{A}^{[\ell]}) = \{a = h(x), \text{s.t.} x \in \mathcal{A}^{[\ell]}\}.$$

We define $h(\mathcal{B}^{[\ell]})$ and $h(\mathcal{C}^{[\ell]})$ accordingly for all values of ℓ.

Note that a triplet of buckets $(\mathcal{A}^{[\ell]}, \mathcal{B}^{[t]}, \mathcal{C}^{[r]})$ may contain a solution of the 3XOR problem, *only if* $r = \ell \oplus t$. For all triplets of buckets $(\mathcal{A}^{[\ell]}, \mathcal{B}^{[t]}, \mathcal{C}^{[\ell\oplus t]})$, the idea of the algorithm is to check if a solution may exist or not by first checking if there is a solution to the instance $(h(\mathcal{A}^{[\ell]}), h(\mathcal{B}^{[t]}), h(\mathcal{C}^{[\ell\oplus t]}))$.

If this test fails, we know for sure that there is no solution in $(\mathcal{A}^{[\ell]}, \mathcal{B}^{[t]}, \mathcal{C}^{[\ell\oplus t]})$, and we can go to the next instance, without having to perform any other operation. On the other hand, if the test passes, we only know that there is a solution to the 3XOR problem with probability $(1/2)^{n-k-s}$.

Solving each of the 2^{2k} small instances $(h(\mathcal{A}^{[\ell]}), h(\mathcal{B}^{[t]}), h(\mathcal{C}^{[\ell\oplus t]}))$ can be done in constant time, using table T.

Remark 1. One can notice that if a bucket contains more than P vectors then these additional vectors are not considered during the preliminary test. Thus, we have to treat them separately afterward. However, if we choose P to be equal to $\kappa_2 \cdot m$, with a well chosen κ_2, we can ensure that the number of buckets that will contain more than P elements is very small.

Using Chernoff bounds, we figured out that, if we choose κ_2 to be around 4.162, the probability that a bucket will contain more than P elements will be $2^{-4 \cdot m}$.

Remark 2. Taking this into account, we have:

$$m = 2^{n/3-k} = \Theta(n/\log n). \tag{5}$$

This is a direct consequence of the definition of P and s and of equation 4.

4.2 Complexity Analysis

Theorem 3. *With input lists of size $2^{n/3}$, the time complexity of the BDP algorithm in our model is*

$$\mathcal{O}\left(\frac{2^{2n/3} \cdot \log^2(n)}{n^2} + N_{test} \cdot \frac{n^2}{\log^2 n}\right),$$

where N_{test} is the number of triplets that pass the test. The expected value of N_{test} is:

$$\mathbb{E}[N_{test}] = \left(\frac{2^{2 \cdot n/3}}{m} - 1\right) \cdot \left(1 - \left(1 - \frac{1}{w^{\kappa_1}} \cdot \left(1 - \frac{1}{w^{\kappa_1}}\right)\right)^{\Theta(n^3/\log^3(w))}\right) + 1.$$

When n grows to infinity, N_{test} is equivalent to 1. Thus, the asymptotic complexity of this algorithm is

$$\mathcal{O}\left(\frac{2^{2n/3} \cdot \log^2 n}{n^2}\right).$$

Before we prove this theorem, we need first to introduce some intermediate results.

Lemma 1. *Let us denote by N_{test} the number of triplets $(\mathcal{A}^{[t]}, \mathcal{B}^{[t \oplus \ell]}, \mathcal{C}^{[\ell]})$ The time complexity of the algorithm 1 is:*

$$T_{BDP} = \mathcal{O}\left(\frac{2^{2n/3} \cdot \log^2 n}{n^2} + N_{test} \cdot \frac{n^2}{\log^2 n}\right).$$

Proof. The time complexity of the full procedure is:

$$T_{BDP} = T_{dispatch} + 2^{2 \cdot k} \cdot T_{test} + N_{test} \cdot T_{solve} + T_{additional},$$

where: $T_{dispatch}$ is the time it takes to dispatch the elements of the three lists according to their k first bits. As discussed in section 2.2,

$$T_{dispatch} = \mathcal{O}(A + B + C) = \mathcal{O}\left(2^{n/3}\right).$$

T_{test} is the time complexity of testing one sub-instance $(\mathcal{A}^{[t]}, \mathcal{B}^{[\ell \oplus t]}, \mathcal{C}^{[\ell]})$. This is constant time, by a lookup in a precomputed table. We will have to perform this test for all triplets $(A^{[t]}, B^{[\ell \oplus t]}, C^{[\ell]})$, that means $2^{2 \cdot k}$ times.

T_{solve} is the time required to solve one small instance $(\mathcal{A}^{[t]}, \mathcal{B}^{[\ell \oplus t]}, \mathcal{C}^{[\ell]})$.

$$T_{solve} = \mathcal{O}\Big((C^{[\ell]} + A^{[t]} \cdot B^{[\ell \oplus b]})\Big).$$

Furthermore, as $C^{[\ell]}$, $A^{[t]}$ and $B^{[\ell\oplus b]}$ are all $\mathcal{O}(m)$,

$$T_{solve} = \mathcal{O}\big(m^2\big)\,.$$

$T_{additional}$ is the time required for additional search, when a bucket contains more than P elements. As we have chosen P so that this event is very unlikely, this can be neglected.

All in all we obtain:

$$T_{BDP} = \mathcal{O}\Big(2^{n/3} + 2^{2\cdot k} + N_{test}\cdot m^2\Big)\,.$$

In addition, we have $2^k = 2^{n/3}/m$ and $m = \Theta(n/\log(n))$. This is enough to conclude the proof of this lemma. □

Lemma 2. *The expected number of triplets that pass the test is:*

$$\mathbb{E}[N_{test}] = \big(2^{2\cdot k}-1\big)\cdot\left(1-\left(1-\frac{1}{w^{\kappa_1}}\cdot\left(1-\frac{1}{w^{n-\kappa_1}}\right)\right)^{\Theta(n^3/\log^3(w))}\right)+1.$$

Proof. Let us consider a sub-instance $(\mathcal{A}^{[t]},\mathcal{B}^{[\ell\oplus t]},\mathcal{C}^{[\ell]})$. Let $(a,b,c)\in\mathcal{A}^{[t]}\times\mathcal{B}^{[\ell\oplus t]}\times\mathcal{C}^{[\ell]}$. Recalling that $h(a)$ (resp. $h(b)$, $h(c)$) represents s arbitrary chosen bits of a (resp. b, c), that are uniformly random, the probability p that $h(a)\oplus h(b) = h(c)$, knowing that $a\oplus b\neq c$ is:

$$p = \frac{1}{2^s}\cdot\left(1-\frac{1}{2^{n-s}}\right).$$

Using Chernoff bounds, we know that the size of the small lists $h(\mathcal{A}^{[t]})$, $h(\mathcal{B}^{[\ell\oplus t]})$ and $h(\mathcal{C}^{[\ell]})$ are $\Theta(m)$ with high probability. Then, we can estimate the probability that a given instance is a false positive to the preliminary test is:

$$\mathbb{P}[\text{pass while should not}] = \left(1-(1-p)^{\Theta(m^3)}\right),$$

with $m = \Theta(n/\log(w))$.

Furthermore, given the size of the list, we expect only to find one solution. Thus, we expect only one true positive in the preliminary test. Then, there should be only *one* triplet among $2^{2\cdot k}$ that contains a solution. The $2^{2\cdot k}-1$ other are expected to contain none.

From here, we can estimate that N_{test} is:

$$N_{test} = (2^{2\cdot k}-1)\cdot\left(1-(1-p)^{\Theta(n^3/\log^3(w))}\right)+1.$$

□

Lemma 3. *If $w(n)\in\Theta(n)$, and β is a constant, then Choosing $\kappa_1 = 3$, $N_{test} \underset{0^+}{\sim} 1$.*

Proof. We set:

$$F(n) = \left(1 - \frac{1}{w(n)^{\kappa_1}}\left(1 - \frac{1}{w(n)^{n-\kappa_1}}\right)\right)^{\beta \cdot n^3 / \log^3(w(n))}.$$

Then

$$\ln(F(n)) = \beta \cdot \frac{n^3}{\log^3(w(n))} \cdot \ln\left(1 - \frac{1}{w(n)^{\kappa_1}}\left(1 - \frac{1}{w(n)^{n-\kappa_1}}\right)\right).$$

As $w(n) \in \Theta(n)$, with $\kappa_1 > 0$:

$$\lim_{n \to +\infty} \frac{1}{w(n)^{\kappa_1}} - \frac{1}{w(n)^n} = 0.$$

Thus,

$$\ln(1 - (\frac{1}{w(n)^{\kappa_1}} - \frac{1}{w(n)^n})) \underset{+\infty}{=} -\left(\frac{1}{w(n)^{\kappa_1}} - \frac{1}{w(n)^n}\right) + \mathcal{O}\left(\frac{1}{n^{2\kappa_1}} + \frac{1}{n^n}\right) \underset{+\infty}{=} -\frac{1}{w(n)^{\kappa_1}} + \mathcal{O}\left(\frac{1}{n^{2\kappa_1}}\right).$$

Then,

$$\ln(F(n)) \underset{+\infty}{=} -\beta \cdot \frac{n^3}{\log^3(w(n))} \cdot \frac{1}{w(n)^{\kappa_1}} + \mathcal{O}\left(\frac{1}{n^3 \log^3(n)}\right).$$

As $w(n) \in \Theta(n)$, there are two constants β_1, β_2 strictly greater than 0, and a certain n_0 such that for all $n \geq n_0$,

$$-\beta_1 \cdot \frac{n^{3-\kappa_1}}{\log^3(w(n))} + \mathcal{O}\left(\frac{n^{3-2\kappa_1}}{\log^3(n)}\right) \leq \ln(F(n)) \leq -\beta_2 \cdot \frac{1}{\log^3(w(n))} + \mathcal{O}\left(\frac{n^{3-2\kappa_1}}{\log^3(n)}\right).$$

Choosing $\kappa_1 \geq 3$ will ensure $\ln(F(n)) \to 0$ when n grows up to infinity. We choose then $\kappa_1 = 3$.

From here we can deduce the following equation:

$$\lim_{n \to +\infty} \left(1 - \frac{1}{w(n)^3}\left(1 - \frac{1}{w(n)^{n-3}}\right)\right)^{\beta \cdot n^3 / \log^3(w(n))} = 1, \tag{6}$$

or in a simpler way:

$$(1 - 1/p)^{\Theta(n^3 / \log^3(w))} = 1$$

The proof lemma 3, is trivial from here. □

Theorem 3 is a direct consequence of all these results.

5 Practical Considerations and Experimental Results

5.1 An Academic Exercise

During the preparation of this work, we implemented and compared the relative performances of several algorithms to solve the 3XOR problem. We tried to answer the question: "*if someone actually wanted to solve a concrete instance of the 3XOR problem, what would she do?*". This is in fact ill-formulated ; in all known applications of generalized birthday algorithms, we not only have to solve the instance, but also to create it in the first place. Often, some tradeoffs can be made. This is the case in the aforementioned application to the COPA mode of operation for authenticated encryption: we may either assemble three lists of size $2^{n/3}$ (and solve the 3XOR instance in time $\approx 2^{2n/3}$) or assemble three lists of size $2^{n/2}$ (and solve the 3XOR instance in time $\approx 2^{n/2}$).

To better understand these tradeoffs, we chose to tackle an academic "practical" problem: computing a 3XOR on the SHA256 hash function reduced to n bits, for the largest possible value of n.

Most of the literature devoted to the generalized birthday problem is mostly theoretical, in particular because the exponential space requirement of these algorithms makes them quite impractical. One notable exception is [BLN+09], which provides the source code of a high-quality implementation of Wagner's k-list algorithm. Studying this code was enlightening for us.

We performed our tests on a "Haswell" Core i5 CPU. Our implementation of the quadratic algorithm takes 340 CPU hours, while our implementation of algorithm G, described in section 3 takes 105 CPU hours.

Space/Data constraints are the hardest. The algorithms of Joux and Nikolić-Sasaki compute the join of two lists of size $2^{n/2-\epsilon}$. Storing and/or moving around such a massive amount of data is the main limiting factor. For instance, with $n = 96$, each list of 2^{48} 12-byte entries requires 3 Petabytes of storage. Even by exploiting the fact that $\mathcal{L}_i$ can be re-computed on-the-fly and does not strictly need to be stored, we came short of a way to compute the join in reasonable time.

However, this obstacle can be sidestepped with clamping. With $n = 96$ and clamping on 24 bits, about $2^{49.6}$ evaluations of SHA256 are required to create three lists of 2^{24} entries with 24 leading zeroes each. Then, the 3XOR can be computed in about 2^{48} operations using 576 Megabytes of storage. This computation was carried out in practice.

The BDP Algorithm is completely impractical. While it is asymptotically more efficient, the BDP algorithm fails in practice for reasonable values of n (e.g. $n = 96$). In fact, for $n = 96$, and $w = 64$, from equation 4, we have:

$$3 \cdot P \cdot s = 3 \cdot \kappa_1 \log(w) \cdot \kappa_2 m = 32.$$

Choosing $\kappa_1 = 3$ as in lemma 3, and $\kappa_2 = 4.162$, as in remark 1, we obtain:

$$m = \frac{32}{3 \cdot 3 \cdot 4.162 \cdot \log 64} = \frac{32}{224.748} \simeq 0.142.$$

The best we could hope, in that case, is to process "batches" composed of... a tenth of an entry!

Creating the lists cannot be neglected. A single core of a modern CPU is capable of evaluating SHA256 on single-block inputs about 20 million times per second, so that creating the lists takes 11,851 CPU hours sequentially. Put another way, generating the input lists is 100× slower than processing them!

It would have been smarter to do clamping on 22 bits instead of 24. This would have reduced the total running time by 2.5×, at the expense of making the lists 4 times bigger. This is ultimately dependent on the speed at which the "oracles" can be evaluated.

5.2 Clusters, Cache, Registers and Other Gory Details

All our implementations are written in plain C, and are relatively concise, totaling 1000 lines of code (not including one-shot tools to generate the hashes, check for potential collisions, etc. which add 1600 additional lines). This section details some of our choices.

$n = 96$ on 64-bit machines. The transdichotomous RAM [FW93] model is practical to think about the complexities of algorithms, but it hits limitations when one tries to actually implement things. We used the following two-step strategy to cope with values of n larger than 64: first, find and store all triplets (i, j, k) such that $\mathcal{A}_i[0..64] \oplus \mathcal{B}_j[0..64] \oplus \mathcal{C}_k[0..64] = 0$. This means running the quadratic algorithm or algorithm G, and only dealing with 64-bit quantities. This allows for simple and efficient implementations.

In a second step, check if these partial solutions extend to full solutions. This necessitates to deal with the full n bits, but is comparatively much simpler than the first step. In addition, the expected number of partial solutions is small.

The quadratic algorithm. The summary description given in section 2.2 suggests to allocate a hash table that holds the whole $\mathcal{C}$ list. Most accesses to this hash table are likely to incur the penalty of a cache miss, and this can be avoided at no cost. We dispatch the entries of $\mathcal{A}, \mathcal{B}$ and $\mathcal{C}$ into buckets of small expected size using their most-significant bits. An eventual 3XOR must lie in $A^{[i]} \times B^{[j]} \times C^{[i \oplus j]}$ for some i, j. The idea is to process buckets of C one-by-one, storing them in a small hash table. Then, we consider all the pairs from the corresponding bins of A and B. This yields the following algorithm

Algorithm Q (*More practical quadratic algorithm*). Find all triplets (a, b, c) from $\mathcal{A} \times \mathcal{B} \times \mathcal{C}$ such that $a \oplus b \oplus c = 0$. The algorithm takes two input parameters k and ℓ.

```
Q1. [Dispatch.]            Dispatch $\mathcal{A}$, $\mathcal{B}$ and $\mathcal{C}$ in buckets according to their first
                           $k$ bits (this essentially requires sorting the lists on their
                           first $k$ bits).
Q2. [Outer loop.]          For all $0 \leq u, v < 2^{\ell}$ do:
Q3. [L3-loop.]                 For all $u2^{k-\ell} \leq i < (u+1)2^{k-\ell}$ do:
Q4. [Hash $C^{[i]}$.]              Initialize a hash table with the entries of $\mathcal{C}^{[i]}$.
Q5. [L1-loop.]                     For all $v2^{k-\ell} \leq j < (v+1)2^{k-\ell}$ do:
Q6. [Check $\mathcal{A}^{[j]} \times \mathcal{B}^{[i \oplus j]}$.]            For all $x \in \mathcal{A}^{[j]}$ and all $y \in \mathcal{B}^{[i \oplus j]}$ do:
                                           If $x \oplus y \in \mathcal{C}^{[i]}$ then emit $(x, y)$.   ▮
```

k should be chosen such that the hash table created in step Q4 fits inside L1 cache. Then all iterations of the L1-loop of steps Q5–Q6 works with data available in the fastest cache. Parallelization is easy: all iterations of the outer loops on u, v can be done concurrently on several machines. The three-level loop structure guarantees that one iteration of steps Q3–Q6 only needs to read $2^{n-\ell}$ entries of each list. ℓ is ideally chosen so that the corresponding portions of both $\mathcal{A}$ and $\mathcal{B}$ fit in L3 cache. 2^{18} entries of $\mathcal{A}$ and $\mathcal{B}$ fit in 2 megabyte of L3 cache, and a few minutes will be necessary to process the corresponding 2^{36} pairs. Memory bandwidth is not a problem, and the quadratic algorithm scales well on multi-core machines. The algorithm can also be run on machines with limited memory. Algorithm Q runs at 10–11 cycles per pair processed on a "Haswell" Core i5 CPU.

Algorithm G. The efficient implementation of joins is a non-trivial problem, which has been studied for a while by the database community. We used a reasonably efficient sort-join, but it turns out that hash-joins have the favors of the experts (for now).

The longest operation of each iteration is the sorting step of Algorithm M, which accounts for 50% of the total running-time. We use three passes of radix-256 sort for the case of $n = 96$ (where the value of the k parameter is 24). The out-of-place version is 2–3× faster than the in-place version (but requires twice more memory); the technique of "multi-histogramming" helped a little. We use the M4RI library [AB12] to compute the PLUQ factorization. To solve our $n = 96$ problem, a single iteration of algorithm G runs at 75 CPU cycles per list item processed on the same "Haswell" Core i5 CPU.

It is easy to parallelize the loop on i (each iteration takes slightly less than 1s for $n = 96$). The problem is that both the full $\mathcal{A}$ and $\mathcal{B}$ must fit in RAM, as they are entirely read in each iteration. When the lists only have 2^{24} entries (as it is the case for $n = 96$), they only require 256 Megabytes. On a multi-core machine, one iterations can be run concurrently per core. One potential problem is that this may saturate the memory bandwidth: each iteration reads 1.25 Gigabytes from memory in about 1s, so on a large chip with 18 cores/36 threads, up to 22.5 Gigabytes/s of memory bandwidth would be required.

A further problem is that, for larger values of n, it becomes impossible to holds many independent copies of the lists in memory. In that case, the sorting and merging operations themselves have to be parallelized, and this is a bit less obvious. If a single copy of the lists does not fit in memory, a distributed sort/merge will be required, and it is likely that the communication overhead will make this less efficient than the quadratic algorithm.

Distributed computation. It is easy to run these algorithms on clusters of loosely-connected machines. We used a simple master-slave approach, in which the iterations to be parallelized are numbered. When a slave node becomes ready, it requests an iteration number to the master node. When it has finished processing it, it sends the partial solutions found in the iteration back to the master. The master stores a log of all accomplished iterations. Communications were handled by the ∅MQ library [Hin13]. We actually ran the algorithms on several hundred cores concurrently.

5.3 Results and Further Work

Consider the three ASCII strings:

$$x = \texttt{FOO-0x0000B70947f064A1}$$
$$y = \texttt{BAR-0x000013f9e450df0b}$$
$$z = \texttt{FOOBAR-0x0000e9b2cf21d70a}$$

The reader can readily check that

```
  SHA256(x) = 000000a9 4fc67b35 beed47fc addb8253 911bb4fa ecaee2d9 f46f7f10 5c7ba78c
^ SHA256(y) = 00000017 d29b29eb a0ef2522 db22d0cc 5d48d2f9 36149197 6430685b 1266ee76
^ SHA256(z) = 000000be 9d5d52de 1e0262de e51c1119 edff081d 868fe419 879932ab bbcfe66e
=====================================================================================
            = 00000000 00000000 00000000 93e54386 21ac6e1e 5c359757 17c625e0 f5d2af94
```

After completing this 96-bit 3XOR, we embarked on a project to compute a 128-bit 3XOR on SHA256. To this end, we found a way to use off-the-shelf bitcoin miners, which evaluate SHA256 about one million times faster than a CPU core. Bitcoin miners naturally produce bitstrings whose SHA256 is zero on (at least) the 32 most significant bits. We plan to accumulate three lists of 2^{32} entries, and then to use algorithm G to find 2^{32} partial solutions on 64 bits, amongst which one should lead to a 128-bit 3XOR (thanks to the extra 32 bits of clamping). Note that this setting naturally enables the use of the technique described in section 3.2.2.

We presently accumulated 10% of the input lists, and we keep on mining.

Acknowledgement

Claire Delaplace was supported by the french ANR under the Brutus project. We thank the anonymous reviewers for their comments.

References

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

[BDP05] Ilya Baran, Erik D Demaine, and Mihai Pătraşcu. Subquadratic algorithms for 3SUM. In *Workshop on Algorithms and Data Structures*, pages 409–421. Springer, 2005.

[Ber07] Daniel J. Bernstein. Better price-performance ratios for generalized birthday attacks, 2007.

[BJMM12] Anja Becker, Antoine Joux, Alexander May, and Alexander Meurer. Decoding random binary linear codes in $2^{n/20}$: How $1 + 1 = 0$ improves information set decoding. In *EUROCRYPT*, pages 520–536. Springer, 2012.

[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.

[CG90] John T Coffey and Rodney M Goodman. The complexity of information set decoding. *IEEE Transactions on Information Theory*, 36(5):1031–1037, 1990.

[CJM02] Philippe Chose, Antoine Joux, and Michel Mitton. Fast correlation attacks: An algorithmic point of view. In *EUROCRYPT*, pages 209–221. Springer, 2002.

[CLRS01] Thomas Cormen, Charles Eric Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to algorithms*, volume 6. MIT press Cambridge, 2001.

[Eri95] Jeff Erickson. Lower Bounds for Linear Satisfiability Problems. In *SODA*, pages 388–395, 1995.

[FKS84] Michael L. Fredman, János Komlós, and Endre Szemerédi. Storing a sparse table with 0(1) worst case access time. *J. ACM*, 31(3):538–544, June 1984.

[FW93] Michael L Fredman and Dan E Willard. Surpassing the information theoretic bound with fusion trees. *Journal of computer and system sciences*, 47(3):424–436, 1993.

[Gil52] Edgar N Gilbert. A comparison of signalling alphabets. *Bell System Technical Journal*, 31(3):504–522, 1952.

[GO95] Anka Gajentaan and Mark H Overmars. On a class of $\mathcal{O}(n^2)$ problems in computational geometry. *Computational geometry*, 5(3):165–185, 1995.

[Hin13] Pieter Hintjens. *ZeroMQ: messaging for many applications.* O'Reilly, Sebastopol, CA, 2013.

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

[JV13] Zahra Jafargholi and Emanuele Viola. 3sum, 3xor, triangles. *CoRR*, abs/1305.3827, 2013.

[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.

[Knu98] 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.

[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.

[McE78] Robert J McEliece. A public-key cryptosystem based on algebraic. *Coding Theory*, 4244:114–116, 1978.

[Mit96] Michael David Mitzenmacher. *The power of two random choices in randomized load balancing*. PhD thesis, PhD thesis, Graduate Division of the University of California at Berkley, 1996.

[MMT11] Alexander May, Alexander Meurer, and Enrico Thomae. Decoding Random Linear Codes in $\tilde{\mathcal{O}}(2^{0.054n})$. In *EUROCRYPT*, pages 107–124. Springer, 2011.

[MO15] Alexander May and Ilya Ozerov. On computing nearest neighbors with applications to decoding of binary linear codes. In *EUROCRYPT*, pages 203–228, 2015.

[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.

[PR01] Rasmus Pagh and Flemming Friche Rodler. Cuckoo hashing. In *European Symposium on Algorithms*, pages 121–133. Springer, 2001.

[Var57] RR Varshamov. Estimate of the number of signals in error correcting codes, 1957.

[Vio12] Emanuele Viola. Reducing 3xor to listing triangles, an exposition. Technical report, Northeastern University, College of Computer and Information Science, May 2012. Available at http://www.ccs.neu.edu/home/viola/papers/xxx.pdf.

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

A Baran Demaine and Pătrașcu Algorithm

Let k be a parameter such that $\mathcal{A}$, $\mathcal{B}$ and $\mathcal{C}$ are dispatched into 2^k buckets according to their first k bits. For each small set $\mathcal{A}^{[\ell]}$ (resp. $\mathcal{B}^{[t]}$ and $\mathcal{C}^{[\ell \oplus t]}$), we keep a Ps-bit vector $v_A[\ell]$ (resp. $v_B[t]$ and $v_C[\ell \oplus t]$), in which the elements of $h(\mathcal{A}^{[\ell]})$ (resp. $h(\mathcal{B}^{[t]})$ and $h(\mathcal{C}^{[\ell \oplus t]})$) are stored. With this construction, assuming that the number of elements of each buckets does not exceed P, a solution to the instance $(h(\mathcal{A}^{[\ell]}), h(\mathcal{B}^{[t]}), h(\mathcal{C}^{\ell \oplus t}))$ exists if and only if $T[v_A[\ell] \parallel v_B[t] \parallel v_C[\ell \oplus t]] = 1$. The following procedure is used to initialize the tables v_A, v_B and v_C:

Algorithm V (*Initialise vectors for preliminary test*). Given a list $\mathcal{A}$ dispatched into 2^k buckets, create the table v_A.

```
V1. [Iterate on buckets.]     For all $0 \leq \ell < 2^k$ do:
V2. [Initialize.]                 Set $v_A[\ell] \leftarrow \varepsilon$.
V3. [Iterate on $A^{[\ell]}$.]    For all $0 \leq i < \max(P, |A^{[\ell]}|)$ do:
V4. [Update.]                         $v_A[\ell] \leftarrow v_A[\ell] \parallel h\left(A^{[\ell]}[i]\right)$. ▮
```

If there are exceeding elements in the buckets they are to be treated independently. All in all, this leads to Algorithm 1.

```
Data: Three lists $\mathcal{A}$, $\mathcal{B}$ and $\mathcal{C}$ and the precomputed table $T$
Result: All couples $(\mathcal{A}_i, \mathcal{B}_j)$ such that $\mathcal{A}_i \oplus \mathcal{B}_j$ is an element of $\mathcal{C}$
Dispatch $\mathcal{A}, \mathcal{B}$ and $\mathcal{C}$, according to their first $k$ bits.
Use Algorithm V to create the tables $v_A$, $v_B$ and $v_C$;
for $0 \leq \ell, t < 2^k$ do
    $v \leftarrow v_A[t] | v_B[\ell \oplus t] | v_C[\ell]$;
    if $T[v] = 1$ ;          // There may be a solution in this sub-instance
     then
        for all couples $(\mathcal{A}_i, \mathcal{B}_j)$ in $\mathcal{A}^{[t]} \times \mathcal{B}^{[\ell \oplus t]}$ do
            if $\mathcal{A}_i \oplus \mathcal{B}_j$ is in $\mathcal{C}[id_C[\ell]..id_C[\ell+1]]$ then
                emit $(\mathcal{A}_i, \mathcal{B}_j)$;
            end
        end
    end
    else if $A^{[t]} > P$ ; // There is more than $P$ elements of $\mathcal{A}$ that start
     with the prefix $t$
     then
        for all couples $(\mathcal{A}_i, \mathcal{B}_j)$ in $\mathcal{A}^{[t]}[P..] \times \mathcal{B}^{[\ell \oplus t]}$ do
            if $\mathcal{A}_i \oplus \mathcal{B}_j$ is in $\mathcal{C}[id_C[\ell]..id_C[\ell+1]]$ then
                emit $(\mathcal{A}_i, \mathcal{B}_j)$;
            end
        end
    end
    else if $B^{[\ell \oplus t]} > P$ ;       // There is more than $P$ elements of $B$ that
     start with the prefix $t \oplus \ell$
     then
        for all couples $(\mathcal{A}_i, \mathcal{B}_j)$ in $\mathcal{A}^{[t]} \times \mathcal{B}^{[\ell \oplus t]}[P..]$ do
            if $\mathcal{A}_i \oplus \mathcal{B}_j$ is in $\mathcal{C}[id_C[\ell]]..id_C[\ell+1]]$ then
                emit $(\mathcal{A}_i, \mathcal{B}_j)$;
            end
        end
    end
    else if $C^{[\ell]} > P$ ; // There is more than $P$ elements of $C$ that start
     with the prefix $\ell$
     then
        for all couples $(\mathcal{A}_i, \mathcal{B}_j)$ in $\mathcal{A}^{[t]} \times \mathcal{B}^{[\ell \oplus t]}$ do
            if $\mathcal{A}_i \oplus \mathcal{B}_j$ is in $\mathcal{C}[id_C[\ell] + P..id_C[\ell+1]]$ then
                emit $(\mathcal{A}_i, \mathcal{B}_j)$;
            end
        end
    end
end
```

Algorithm 1: Adaptation of BDP algorithm to our problem