Another Look at the Cost of Cryptographic Attacks

Charles Bouillaguet

▶ To cite this version:

Charles Bouillaguet. Another Look at the Cost of Cryptographic Attacks. 2019. hal-02306912v1

HAL Id: hal-02306912
https://hal.science/hal-02306912v1

Preprint submitted on 7 Oct 2019 (v1), last revised 17 Feb 2022 (v2)

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

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

Another Look
at the Cost of Cryptographic Attacks

Charles Bouillaguet

Univ. Lille, CNRS, Centrale Lille, UMR 9189 - CRIStAL - Centre de Recherche en Informatique Signal et Automatique de Lille, F-59000 Lille, France
`charles.bouillaguet@univ-lille1.fr`

Abstract. This paper makes the case for considering the cost of cryptographic attacks as the main measure of their efficiency, instead of their time complexity. This allows, in our opinion, a more realistic assessment of the "risk" these attacks represent. This is half-and-half a position and a technical paper.

Cryptographic attacks described in the literature are rarely implemented. Most exist only "on paper", and their main characteristic is that their estimated time complexity is small enough to break a given security property. However, when a cryptanalyst actually considers implementing an attack, she soon realizes that there is more to the story than time complexity. For instance, Wiener has shown that breaking the double-DES costs $2^{6n/5}$, asymptotically more than exhaustive search on n bits.

We put forward the asymptotic cost of cryptographic attacks as a measure of their practicality. We discuss the shortcomings of the usual computational model and propose a simple abstract cryptographic machine on which it is easy to estimate the cost.

We then study the asymptotic cost of several relevant algorithm: collision search, the three-list birthday problem (3XOR) and solving multivariate quadratic polynomial equations. We find that some smart algorithms cost much more than what their time complexity suggest, while naive and simple algorithms may cost less. Some algorithms can be tuned to reduce their cost (this increases their time complexity).

Foreword

A celebrated High Performance Computing paper entitled "*Hitting the Memory Wall: Implications of the Obvious*" [47] opens with these words:

> This brief note points out something obvious — something the authors "knew" without really understanding. With apologies to those who did understand, we offer it to those others who, like us, missed the point.

We would like to do the same — but this note is not so short.

1 Introduction

We all know that the total number of "elementary operations" required to perform any kind of computation (including cryptographic "attacks") is a *lower-bound* on the difficulty of actually carrying out this computation. Very often this lower-bound is not tight.

This fact is used, implicitly or explicitly by the designers of cryptographic primitives, in particular in the public-key setting where security properties can always be broken by successfully running a large computation to solve the underlying hard problem. Designers estimate the cost of the best attack and choose presumably secure parameters in order to make it intractable.

To give an example chosen (almost) at random: the submitters of the GeMSS signature scheme [14] discuss the complexity of several algorithms against the underlying hard problem (solving systems of multivariate quadratic equations); they discuss the case of the elegant and "provable" algorithm of Lokshtanov, Paturi, Tamaki, Williams and Yu [31], which runs faster than exhaustive search in $\widetilde{\mathcal{O}}\left(2^{0.8765n}\right)$ steps. Conservatively, the designers of GeMSS assume that the polynomial factor hidden in the exponential complexity may be small, and they explicitly state that "*[they] will estimate the cost of this attack by the lower bound* $2^{0.8765n}$".

In this paper, we discuss the relationship between the "cost" of running an attack and the number of elementary operations the attack is supposed to require. These are not the same thing. For instance, we argue in section 8 that the above algorithm is extremely unlikely to be asymptotically more "cost-effective" than exhaustive search. It is therefore likely *less dangerous* in an asymptotic sense, even though it requires a smaller total number of elementary operations.

Using the time complexity of an attack as a lower-bound on the actual hardness of running it is all right. But using it as an upper-bound is not.

1.1 Simple Encryption vs. Double Encryption

To illustrate the discrepancy between the time complexity of an attack and the actual difficulty of running it, let us consider double encryption (*e.g.*, the double-DES). This technique doubles the key length of a block cipher by encrypting twice. Given $E : \{0,1\}^n \times \{0,1\}^n \rightarrow \{0,1\}^n$, we build $E^2_{k_1,k_2}(x) = E(k_1, E(k_2, x))$.

Double-encryption is widely considered to only offer "n-bit security" (see for instance Wikipedia [46]) because of a well-known meet-in-the-middle attack that require only two known plaintext-ciphertext pairs and 2^n sequential steps. The attack works as follows: for all possible k_1, store the $(E_{k_1}(P_1), k_1)$ pair in a table; then for all k_2, look up $E^{-1}_{k_2}(C_1)$ in the table and retrieve the associated k_1's. Check all (k_1, k_2) pairs against the second plaintext-ciphertext pair.

Double encryption seemingly offers no advantage compared to single encryption: breaking it is "as fast" as breaking single encryption. However, we believe that the situation is actually more complicated. In this form, this simple attack is in fact *much more difficult* to actually carry out that n-bit exhaustive search,

because it requires (at least) $n2^n$ bits of space. If we consider the DES ($n = 56$), *ad hoc* parallel machines have been built to perform exhaustive search, such as the "Deep Crack" machine (1856 custom ASICs, 1998). Some low-cost machines are capable of doing it in a few days, such as the COPACOBANA [27] (128 FPGAs, 2006). It is estimated that performing a DES key exhaustive-search on the Amazon cloud requires 97 "cloud-CPU years"; doing it as fast as possible would cost $450,000 as of 2018 [15].

Now, the simple meet-in-the middle attack would require ≈ 512 Petabyte of fast storage, which is not practical. At the time of this writing, the most powerful computer in the world ("Summit" at Oak Ridge National Laboratory in the USA) only has 3.2Pbyte of RAM and 250Pbyte of external storage.

There are several ways to reduce the space requirements of the meet-in-the-middle attack, most notably the parallel collision search algorithm of van Oorschot and Wiener [41]. Using nw bits of memory, the running time would be of order $2^{3n/2}/\sqrt{w}$ blockcipher evaluations. In any case, using less than $n2^n$ bits of memory will require more than 2^n time. As such, breaking the double-DES on the cloud is going to *cost more* than doing a DES exhaustive search: it will require more energy, more computational resources, more communications, more storage, more intellectual effort, etc. All this is not reflected by just counting the number of elementary steps in the algorithm.

In fact, breaking double encryption on the cloud is going to cost *asymptotically* more money than breaking single encryption, as pointed out by Wiener [45], who estimated the cost to be of order $2^{6n/5}$ using optimal parallelization. Breaking the double-DES *in practice* is potentially within the reach of academic teams, but it is certainly more challenging than breaking the simple-DES.

Other Examples. It is easy to come up with artificial problems that require 2^n computation steps but that are much more costly that n-bit exhaustive key search. For instance, consider the MEDIAN problem: if $F_k : \{0,1\}^n \rightarrow \{0,1\}^n$ is a pseudo-random function (indexed by a key k), consider the problem of finding the median of its range given the key.

Precisely assessing the "cost" of solving MEDIAN is not completely obvious; if we had to make a wild guess, we would say that solving an instance of MEDIAN of size n is asymptotically as costly as inverting a $\frac{4}{3}n$-bit random function (this estimate is justified in section 3.2). In any case, we would be quite surprised if anyone could solve an instance of this problem with $n = 56$ during the next few years.

1.2 Main Ideas and Contributions

We first discuss common assumptions about abstract computational models and how cryptographic attacks are compared.

- In section 2, we argue in detail why comparing the time complexity of cryptographic attacks is sufficient as long as the attacks remain purely theoretical.

In practice, it fails to address the practical difficulty of running the attack. We argue that the cost of an attack is a better metric.
- In section 3, we put forward the idea of asymptotically estimating the *cost* of running cryptographic attacks. We also argue that some common operations (multiplication, FFT, sorting, ...) cost asymptotically more than their time complexity. A lot of published cryptographic attacks cost more than simple algorithms such as exhaustive search.
- In section 4, we restate the (well-known) argument that the usual assumption that an unbounded amount of memory can be accessed in constant time violates both the law of physics and empirical observations. This means that the "running time" and the "time complexity" of exponentially large computations are decorrelated.
- In section 5, based on ideas of Wiener and Vitányi, we propose a simple abstract model to discuss the cost of cryptographic attacks: build a parallel machines using any number of fixed-size computer nodes — and 1-meter long network cables.

We then apply the previous ideas to several algorithms from both symmetric and asymmetric cryptanalysis.

- In section 6, as a warm-up, we present parallel implementations (in our proposed model) of parallel collision search, including the meet-in-the-middle attack against double encryption. Its cost matches the known result of Wiener [45].
- In section 7, we study several recent algorithms for the 3XOR problem (*i.e.* for the 3-list birthday problem). We argue that the "smart" algorithms cost asymptotically more than what their time complexity suggests. We also show that a simple, brute-force algorithm can be implemented with optimal cost, and is therefore the algorithm of choice.
- Lastly, in section 8, we turn our attention to several algorithms for the MQ problem (solving multivariate quadratic equations over $\mathbb{F}_2$), which is relevant for some post-quantum schemes. Here the situation is more nuanced: we disqualify a smart algorithm as more costly than exhaustive search, while another one can be tuned and implemented in a way that costs less. Its cost is still higher than its time complexity.

To summarize, we advocate a more practice-oriented approach to cryptanalysis, even when the attacks remain purely theoretical and are analyzed asymptotically.

All complexities given in this paper are asymptotic. We usually dispense with writing the "big O", for the sake of lighter notations. When we say that an algorithm costs 2^{xn} or has time complexity 2^{xn}, we mean $\widetilde{\mathcal{O}}\left(2^{xn}\right)$ (with the usual meaning that there is polynomial in n hidden in the $\tilde{\mathcal{O}}$).

2 What is a "Better Attack"?

Amongst all cryptographic attacks that have been published, only a minority is "practical", in the sense that it could potentially be implemented and run on

existing hardware. Some practical attacks have had a large impact, most notably in forcing the rest of the world to retire bad cryptographic algorithms (SFLASH by [18], MD5 by [37], SHA-1 by [36], not to mention RC4). At the same time, new records in factoring [24] or discrete logs [25] help adjust recommended key sizes.

Some researchers tried specifically to search for practical attacks. For instance, instead of trying to break as many rounds of the AES as possible faster than exhaustive search, Bar-On, Dunkelman, Keller, Ronen and Shamir [1] tried to break as many rounds as possible *in practice* (and verified some of their attacks by reportedly implementing them).

However, most cryptographic attacks are never implemented and are not meant to be. For instance, the nice attack discovered by Isobe [20] against GOST requires time equivalent to 2^{224} evaluations of the block cipher. Further work by Dinur, Dunkeman and Shamir [17] reduces this to 2^{192}. Both attacks are impractical to the point that it makes little sense to try to implement them.

These attacks are theoretical objects that exist only "on paper", and their relative merits are compared in theory (without resorting to practical experiments): in this example, the second attack is better because it achieves the same result while using less resources than the first one[1].

To some extent, cryptanalysis is in part a theoretical game where computer scientists describe (often unimplementable) algorithms and compare their properties in some abstract model. There, if it is good enough, the algorithm "breaks" some security property. Asymptotic and sometimes "concrete" complexities are estimated and compared for computation that will most likely never ever be tractable.

In this theoretical realm, the principal measure of comparison between attacks is their time complexity. Most of us computer scientists are addicted to making algorithms as fast as possible, *i.e.* to minimize their total number of elementary operations. It is the main metric against which algorithms are evaluated, not to mention the most familiar.

If attack A runs in time 2^n using negligible memory, and attack B runs in time $2^{0.875n}$ but also requires $2^{0.875n}$ bytes of memory, which one is better? In other terms, which one is more dangerous? As a community, we cryptographers behave as if attack B was better. The algorithm of [31] for multivariate equations mentioned in the introduction is considered to be better than exhaustive search ("*beating brute force*" as the title of the paper says) because it runs in time $2^{0.8765n}$ instead of 2^n.

The (obvious) point of this article is that judging attacks by time complexity alone is fine... as long as these attacks remain theoretical and that no one ever tries to implement them. Let us look again at attacks A and B above with $n = 64$. Attacks requiring 2^{64} operations and negligible memory have been carried out in practice by academic teams having access to large computational resources [36,

[1] In fact, it is a bit more complicated: [17] present one attack with less memory than [20], other things being equal (it is strictly better). There is also an other another attack in [17] with more data, less memory and less time (is it "better"?).

24]. On the other hand, the memory requirement of attack B (several Exabytes) is a show-stopper.

Practitioners know that space constraints are much more difficult to deal with: one can always wait for a program to stop, but if the data does not fit in memory, it won't even start. Algorithms with large space complexities have to be adapted and made *slower* to accommodate the space constraints (see [6] for an example in the context of generalized birthday attacks).

We stress that, for large computations, time complexity alone is only weakly correlated with the practical hardness of running the computation — it is a lower bound. Looking again at the "better" attack B above, it is obvious that it is going to be more difficult to run than attack A in most scenarios. Attack A should therefore be considered "more dangerous" than attack B, even though it is "slower".

Ultimately, the security of a cryptographic construction depends on the practical hardness of running attacks against it. The main thesis of this paper is that discussing the *cost* of running an attack is more realistic than their time complexity for two reasons: a) it gives a better understanding of what would happen if somebody ever tried to implement it and b) the *cost* is better correlated to the "concrete hardness" of running a computation.

Recommendation 1 : Cost Metric

> Cryptanalysts should consider the *cost* of running the attacks and should compare the relative efficiencies of attacks according to the *cost* (possibly in addition to other metrics such as the total number of instructions executed and the total amount of memory required).

3 The "Cost" of Running a Cryptographic Attack

Several authors in the cryptographic community, most notably Dan Bernstein (in several papers including [3, 5, 4]), Michael Wiener [45], Kleinjung, Lenstra, Page and Smart (in [26] and subsequent updates [15]), proposed to consider the *cost* of cryptographic attacks.

Cryptographers who have run large computations often discuss their cost. For instance, Stevens, Bursztein, Karpman, Albertini and Markov [36] estimated that if their their recent collision on full SHA-1 had to be run on the Amazon public cloud, it would have cost $560,000. Before that, the authors of the 2009 factorization record of a 768 bits RSA modulus [24] estimated that the computation required about 500MWh of energy. This is enough to boil two Olympic swimming pools starting from 20°C (this is another "cost measure").

The "Amazon cost", which consists in measuring the cost in US dollar spent on a public cloud to actually run a given computation is sometimes promoted as a way of estimating the amount of resources needed to run it. Kleinjung *et al.* discuss the Amazon cost of several attacks, including exhaustive key search, factoring, etc. in [26, 15]. Kuo, Schneider, Dagdelen, Reichelt, Buchmann, Cheng and Yang [28] discuss the Amazon cost of solving several instances of the Shortest Vector Problem in euclidean lattices.

The "Amazon cost" evolves in time, which makes it somewhat impractical as a complexity metric (and Amazon itself like all living things has a finite life expectancy). Because we want to focus on timeless truths, we consider a more abstract notion of cost: the product of the size of the machine on which it is run by its (wall-clock) running time :

$$\text{Cost} = \text{Machine size} \times \text{Running time.}$$

Bernstein calls it the *Price-Performance ratio* ; Wiener calls it the *full cost* ; Lenstra, Shamir, Tomlinson and Tromer [29] calls it the *throughput cost*. We just call it the *cost*.

The number of sequential operations of the best known sequential algorithm gives a lower-bound on the cost of the best known parallel implementation: just assume that the sequential algorithm uses $\mathcal{O}(1)$ memory, so that the size of the machine on which the algorithm runs is $\mathcal{O}(1)$.

Sometimes, it is possible to design a parallel implementation whose cost matches the number of sequential operations. We say that such an implementation is *cost-optimal*. Note that if the algorithm uses memory, then any cost-optimal implementation *has* to be parallel. In sections 6 and 7 we provide two examples of cost-optimal implementations. Not all algorithms admit cost-optimal implementations.

3.1 Memory Costs

The cost "charges" the programmer for the use of memory: to hold n bits of memory, a machine must have size $\Omega(n)$. A sequential algorithm running in time T and using M bits of space has cost TM.

There are well-known examples where parallel machines achieve a lower cost than sequential machines. This was put forward by Bernstein [3] with the example of sorting: a well-known "systolic array" (a two-dimensional mesh of $n \times n$ nodes, each with a constant amount of memory), can sort n^2 numbers in time $\mathcal{O}(n)$ [39]. This costs $\mathcal{O}\left(n^3\right)$, compared to $\Omega\left(n^4\right)$ for a single processor connected to a memory of size n^2. Another example is matrix multiplication: another well-known "systolic array" can multiply two $n \times n$ matrices in time $\mathcal{O}(n)$, with improved cost compared to a sequential machine.

The intuitive explanation is that in sequential machines, only a constant number of memory bits can be accessed at each time step, so most of them are "inactive" all the time, yet they contribute to the cost. In parallel machines, each piece of silicon gets more chance to contribute to the actual result at each time step.

The algorithm of [31] to solve quadratic boolean equations in n variables is an interesting example. It has time and space complexity $2^{0.88n}$, so running it would requires a machine of size $2^{0.88n}$. But if such a big machine could be built, it would be much more interesting to use it for... exhaustive search! Instead of taking time $2^{0.88n}$, exhaustive search on n bits using $2^{0.88n}$ processors in parallel would take $2^{0.12n}$ time steps, which is much less. The cost of the algorithm of [31] is discussed in detail in section 8.2.

3.2 Venerable Cost Lower-Bounds

The same notion of cost was in favor in the early 1980's, when the Very Large Scale Integration (VLSI) community proved numerous lower-bounds on the Area-Time product of any VLSI circuit performing specific computations, including cyclic shift and convolution [44], sorting [7], integer multiplication [13], discrete Fourier transform [38], matrix product [35], etc. These bounds have generally been matched by corresponding designs, and hence are tight.

These bounds relate the area A of the circuit and its time complexity T; they are usually of the form "$AT \geq n^{1.5}$" or "$AT^2 \geq n^2$" (the former derives from the latter when the area must be greater than n, which is often the case because the input have to be memorized). They inherently rely on the fact that the circuits are *planar*. One of the proof techniques consists in showing that if the chip is arbitrarily cut in two, then a given amount of information must flow across the cut. This in turn requires a minimum number of wires to be cut if the flow is fast enough, and hence a minimum area for these wires. This way of thinking about computation gives a major role to the "cost of communications" — moving data around is not free.

We are not aware of specific lower-bounds for the MEDIAN problem; nevertheless, the results cited above suggest that trying to solve MEDIAN by sorting costs at least $\Omega\left(2^{1.5n}\right)$ if only flat chips are used — *asymptotically* much more than exhaustive search on n bits.

We are also not aware of generalization of these lower bounds to three dimensions. But it is easy to conjecture that they generalize to "Volume $\times$ Time $= \Omega\left(n^{4/3}\right)$". For instance, a d-dimensional mesh of size n can sort n numbers in time $\mathcal{O}\left(n^{1/d}\right)$ [39]. For $d = 2$, this achieves the lower-bound $AT = n^{3/2}$, while for $d = 3$ this achieves $VT = n^{4/3}$). Going to 3 dimensions would bring down the cost of solving MEDIAN by sorting to $2^{4n/3}$: $2^{n/3}$ time steps with a machine of size 2^n.

We feel confident in assuming that all the problems with VLSI lower-bound also admit (slightly weaker) cost lower-bounds in three dimensions.

Reality Check. Sorting is an important building blocks in many cryptographic attacks. Because sorting requires communications (or, equivalently, access to a large shared memory), we have just claimed that it costs $\Omega\left(n^{4/3}\right)$, hence asymptotically more than its sequential number of operations. Therefore, sorting couldn't possibly be cost-optimal. Is that only theoretical, or does it verifies in practice?

Just to be sure, and to validate our claims, we ran a little experiment on actual hardware. Sorting a large array distributed amongst many cluster nodes can be done as follows: first, each node locally sorts its portion of the distributed array; second, nodes communicate and move portions of the sorted array to their destination nodes (this is an "all-to-all" shuffle); lastly, nodes merge the incoming sorted portions of the array and store them locally.

Using an IBM BluGene/Q parallel computer, we used up to 4096 nodes connected using a high-performance network with a 5D-torus topology. Each node

generated $\approx$ 7Gbyte of random junk and we measured the wall-clock time needed to complete the "all-to-all" shuffle that would happen in sorting between n nodes: each node sends/receives $\approx 7/n$ Gbyte to/from each other node on the network. This was done by running the `MPI_Alltoall` function of the MPI library provided with the machine. Results are shown on Table 1.

# nodes	Data	Time (s)	Mesh / torus dimension
2	13.5 G	2.7	2
4	27 G	2.9	2×2
8	54 G	7.0	$2 \times 2 \times 2$
16	108 G	9.5	$4 \times 2 \times 2$
32	215 G	12.45	$2 \times 4 \times 2 \times 2$
64	430 G	4.1	$2 \times 2 \times 4 \times 2 \times 2$
128	861 G	8.9	$4 \times 4 \times 4 \times 2$
256	1.7 T	13.0	$2 \times 4 \times 4 \times 4 \times 2$
512	3.4 T	13.7	$4 \times 4 \times 4 \times 4 \times 2$
1024	6.9 T	15.9	$4 \times 4 \times 4 \times 4 \times 2$
2048	13.4 T	17.0	$4 \times 4 \times 8 \times 8 \times 2$
4096	27.5 T	18.7	$8 \times 4 \times 8 \times 8 \times 2$

Table 1. Time needed to complete the communication phase of a simulated sort. "Data" gives the total amount of data shuffled around on the network. The bad times obtained for $8 \leq n \leq 32$ may be caused by the fact that these are "small jobs" unsuited to the network topology. The network is always a torus on the last dimension; on the other dimensions, it it a torus if there are at least four ranks, otherwise it is a mesh.

We observed that the running time of the communication phase with n is essentially $6.1n^{0.135}$ seconds (excluding the bad small jobs). On this machine, the cost of sorting would then be $\mathcal{O}\left(n^{1.135}\right)$. Because this machine has a 5D torus network, we could expect it to be able to sort n numbers in time $n^{1/5}$ — it is indeed the case, and the machine does even better. We argue in section 5 that the 5D torus "cheats" because it has "long wires": its size would scale super-linearly with the number of nodes.

In any case, we were able to observe the asymptotic effect of communications on the cost of sorting, which is *super-linear*.

3.3 Is the "Cost" an Better Metric ?

We believe that the cost defined above is a better metric than the total number of elementary operations to appreciate the difficulty of actually running a cryptographic attack. The "exhaustive search vs. MEDIAN" example shows that the cost correlates well with an intuitive notion of difficulty.

The cost metric is annoying because it reverts "attacks" (algorithms that are fast enough to break something) to "non-attacks" (algorithms that are too costly to break anything).

To take another example (almost) at random, consider the two-rounds single-key Even-Mansour construction. This acts as an n-bit block, n-bit key block cipher. Exhaustive search takes time equivalent to 2^n evaluations of the construction and requires no memory to recover the secret key. A series of work by Nikolic, Wang, and Wu [34], Dinur, Dunkelman, Keller and Shamir [16], Isobe and Shibutani [21], Leurent and Sibleyras [30] present attacks against this construction. All these attacks have a total number of operations greater than $2^n/n$ and strive to minimize it using as little queries to the encryption oracle as possible. Their complexity is summarized in table 2. They all use exponentially more than n memory words, so *as they are described by their authors* they all have cost exponentially greater than exhaustive search.

To be fair, these attacks have been designed with other goals than being cost-efficient (minimizing total number of queries to a public oracle or the total number of sequential operations, etc.). Furthermore, they are described as sequential algorithms using a large memory. It is possible that their cost could be reduced by parallelizing them, but it is potentially a non-trivial effort.

In any case, we gather that none of this attacks have been implemented. If a concrete instance with (say) $n = 56$ or $n = 64$ were to be broken in practice, would any of these attacks be more practical than exhaustive search? Some of these articles discuss this "practical" case in detail and give precise "concrete" complexities... which —we dare to think— probably do not mean much in practice. We believe that those of [16, 30] stands a chance to "beat brute force" because of their reduced space complexity, but the global situation is unclear, and again a non-trivial effort would be required to find out.

Ref.	Data	Time	Space	Cost
trivial	2 KP	2^n	1	$= 2^n$
[34]	$2^n \ln n/n$ KP	$2^n \ln n/n$	$2^n \ln n/n$	$\geq 2^{2n}/n^2$
[16]	$2^n/\lambda n$ CP	$2^n/\lambda n$	$2^{\lambda n}$	$\geq 2^{(1+\lambda)n}/n$
[21]	$2^{\lambda n}$ CP	$2^n \ln n/n$	$2^n \ln n/n$	$\geq 2^{2n}/n^2$
[30]	λn KP	$2^n/\lambda n$	$2^{\lambda n}$	$\geq 2^{(1+\lambda)n}/n$

Table 2. Comparison of published attacks against the two-rounds single-key Even-Mansour cipher. $0 < \lambda < 1$. KP: Known plaintext; CP: Chosen plaintext.

Other interesting cases arise when algorithms have parameters. Very often, cryptanalysts try to find the values of these parameters that minimize total number of elementary operations. And very often, *different* values of the parameters minimize the cost. An example is provided in section 8.3.

One could argue that there are situations where the cost metric does not make much sense for the cryptanalyst who wishes to actually run an attack. If she has a given machine at her disposal and shes wants to break stuff for fun and profit, then what matters to her is the actual running time of her code on her machine and not the price-performance ratio. Here, the realism of the metric

does not have to be debated: it holds by definition. Nevertheless, we note that, if the given machine is a (somewhat) large parallel computer, then minimizing the actual running time of an attack is quite similar to minimizing the cost: all the "hidden costs" now have to be accounted for: communications, I/O, memory latency, etc.

In any case, cryptanalysts most often cannot build parallel machines that are exactly as they would like (to minimize the cost), and they have to make do with what they have. The pretty sorting meshes described by Bernstein [3] have most likely not seen much practical realization.

Most large cryptographic computations known to us typically happened on machines to which the cryptanalysts did not have exclusive access. Either the machines belongs to a public cloud operator (as in [36]) or to a campus/national computing infrastructure (as in [24]), shared between many users (often from different fields) — of course there are exceptions [6, 37]. In this setting where the machines are shared, the *cloud cost* ("number of nodes $\times$ running time") seems relevant, but it is very similar to the cost discussed above. It is adapted to situations where the cryptanalyst can still choose the amount of parallelism and trade a notion of machine size for time, but cannot build a fully customized machine.

In any case, the cryptanalyst will generally have resource constraints: a budget (in public cloud, the "cloud cost" translates quite directly to a financial cost, which is the argument used by [26, 15]), or a fixed number of allocated "nodes $\times$ hours" on a shared machine. So the relevant question is: "*what is the largest instance I can break on this machine with this many node $\times$ hours?*". In other terms, what is the (cloud) cost of breaking an instance of size n? We end up with the idea that minimizing the (cloud) cost is the actual objective.

This is reinforced by the fact that both the cost as defined above and the cloud cost correlate with the energy consumption of the computation, thus with an actual cost in dollars (that someone has to pay eventually).

4 What is the Problem With the Usual Computational Model?

The usual computational model in the study of algorithms is the "Random Access Machine" (RAM). Without going into very precise details, it is a von Neumann architecture where a processor can access an (unbounded) memory divided in cells containing w-bit words. The machine can do the usual logic and arithmetic operations between registers in constant time, as well as read/write from its memory in constant time.

This model is simple and practical, arguably more practical than (even multi-tapes) Turing machines or lambda-calculus ; it is reasonably close to how (small) computer appear to a programmer.

It has a parallel counterpart, the "Parallel Random Access Machine" (PRAM) model, in which an arbitrary number of processors operate synchronously and have access (in constant time) to an unbounded shared memory. Several flavors

exist, with various behaviors in case of concurrent Read/Write access to the memory. The most commonly accepted, the CREW PRAM allows Concurrent Reads but Exclusive Write (to the same memory location). Both models suffers from several problems:

- The RAM model is, obviously, a sequential model of computation. Cryptographic attacks, when they are actually implemented, run on parallel computers.
- Attaching a single processor to a very large memory would be a tremendously inefficient use of resources. Such machines do not exist in the real world.
- If we look at parallel computational models, we find that the PRAM model is unrealistic: large parallel machine do not have a shared memory. Individual nodes have their own memory and communicate through a network. More realistic models of parallel computation take communication costs and delay into account.
- In both case, the assumption that an arbitrarily large memory can be accessed in constant time breaks down asymptotically because of speed-of-light delays. The *latency* of memory accesses increases with the size of the memory. The models ignore it.

It follows that if both the time and space complexity of a computation are exponential, then its running time is no longer proportional to the number of "elementary" operations for the above reason.

Looking at the same problem from another angle, reading 64-bit from the memory (with a cache miss) of a modern computer requires about 10 times more energy than performing a double-precision floating-point multiplication [42, appendix A]. This also only gets worse if data has to be moved further from another computing node. Thus, running code that accesses a lot of memory *costs* more than code that "just computes", because eventually someone has to pay that power bill.

All this could be summarized by saying that: a) accessing a lot of data is *not* free, and b) accessing a large shared memory entails *hidden communication costs* which are not accounted for by the usual computational models.

Recommendation 2 : Memory is not Free

> Because it asymptotically contradicts both the laws of physics and practical observations, cryptanalysts should not blindly assume the existence of a shared, unbounded memory with constant-time access.

Reality Check. We argued that speed-of-light delays affects the running time of computations using a large memory. Is this only a theoretical argument? Does it only have an "asymptotic" value or does it mean something in practice?

We believe that the argument is not only theoretical. Recent multi-die CPUs can be large enough that light takes at least one cycle to go from one core to the furthest core on the chip (on the AMD EPYC "rome", two cores can be up to 6cm apart). Actual bits of information take presumably much more time

than light to travel. For this reason, distinct processor core on the same machine do not always have a consistent view of their shared memory (it would be too expensive to keep them sequentially consistent).

It is well-known from the HPC community that the *latency* of memory accesses increases quickly when reaching memory that is further and further away from the processor core. Just to be sure and to validate our claims, we ran another little experiment on actual hardware: we measured the latency of memory accesses on a modern cluster node with Xeon Gold 6130 processors (skylake) and fast RAM. We measured the time needed to do $x \leftarrow A[x]$ repeatedly, where A is an array of size n initialized with a random permutation. The results are shown in table 3.

size of A	Latency	Note
32Kb	5	L1 cache hit
1Mb	25	L2 cache hit
22Mb	50	L3 cache hit
32Mb	130	
128Mb	250	
8Gb	300	
192Gb	340	

Table 3. Latency (in CPU cycles) of random memory accesses in a contemporary (high-end) computer. 2Mb "huge pages" have been used to alleviate page fault issues and reduce the latencies.

This shows that randomly probing a few hundred megabytes of memory is $\approx 60\times$ more expensive than accessing a few kilobytes in the fastest cache. We feel comforted in the conviction that, the more memory an algorithm needs, the more it has to wait to access it (if it accesses it randomly).

An interesting other data point is the following: on the same CPUs, the latency of a single evaluation of the AES-128 is about 86 cycle (using AES-NI instructions, with precomputed subkeys). A good thing to take home is the following: evaluating the AES is faster than reading a single integer from RAM.

We note that these facts are well-known from the cryptographic community: special *memory-bound* functions have been designed to explicitly rely on the fact that random memory accesses are slow. These functions can be used to provide proof-of-work (as in [19]) and are recommended to hash passwords (as in Argon2 [8]).

5 What is the "Size" of a Machine?

Most of the problems we discuss revolve around what happens when processors access memory. We argued that this involves a "hidden" interconnection network,

which incurs non-trivial costs and delays the computation. Wiener made the same point in a striking way with the following theorem.

Theorem 1 ([45], rephrased by us). *In a machine where each of p processors uniformly random access to m memory elements at a memory access rate r, the total length of wires is $\Omega\left((pr)^{1.5}\right)$. This bound is tight.*

This requires no assumption on the relative locations of processors and memory elements. However this assumes that the computation is in "steady state" for a sufficiently long time.

In other terms, if many processors must have fast access to a common memory, then there is a hidden cost (the communication network) which can even asymptotically dominate the size of the machine if $r > p^{-1/3}$.

Here is another example of hidden communication costs. Assume 2^k simple processors are connected together in a hypercube (a classical network topology in parallel computing classes; each node is connected to k neighbors). Bitonic sorting is easy to implement on such a machine: it sorts 2^k items (one per node) in $k^2/2$ steps, where each steps involves the parallel exchange of one element along a network link, without contention. So, sorting 2^k items would take time $\mathcal{O}\left(k^2\right)$ on a machine of size $\mathcal{O}\left(2^k\right)$ — beating the cost lower bound announced in section 3.2.

However, Vitányi [43] has shown that the total length of wires needed to connect the nodes of a hypercube in our three-dimensional world is $\Omega\left(2^{4n/3}\right)$. As such, the "size of the machine" is in fact greater than assumed earlier. And this is assuming that the wires have zero volume! If the wires occupy some space, then they will push the nodes further apart, necessitating even longer wires again... It is known that a hypercube variant (the "cube-connected cycles") can be laid out on a three dimensional grid with volume $2^{3n/2}$, and it seems that it is the best possible.

In light of the theorem above, and to avoid nasty surprises, we believe that if cryptanalysts want to discuss the cost of their attacks, they have two options. The first consists in actually implementing them on real machines, when this makes sense. The other option is to discuss the asymptotic cost on some abstract "cryptanalytic" machine.

This machine can be a public cloud: assume that an unbounded number N of identical nodes with a fixed amount of memory M per node is available. The cost of a computation with running time T is then NTM. The slight problem is that the interconnection network remains to be specified: communication between nodes takes time and congestion has to be taken into account. We do not know the precise details of network connectivity in actual public clouds.

This could be fixed by assuming a well-specified network topology that leads itself to a reasonable embedding in our material world. Two-dimensional or three-dimensional meshes (or tori) of identical nodes with a fixed amount of local memory seem to fit this description: the total length of wire is linear in the number of processors, the number of links per node is constant, and each link has constant length.

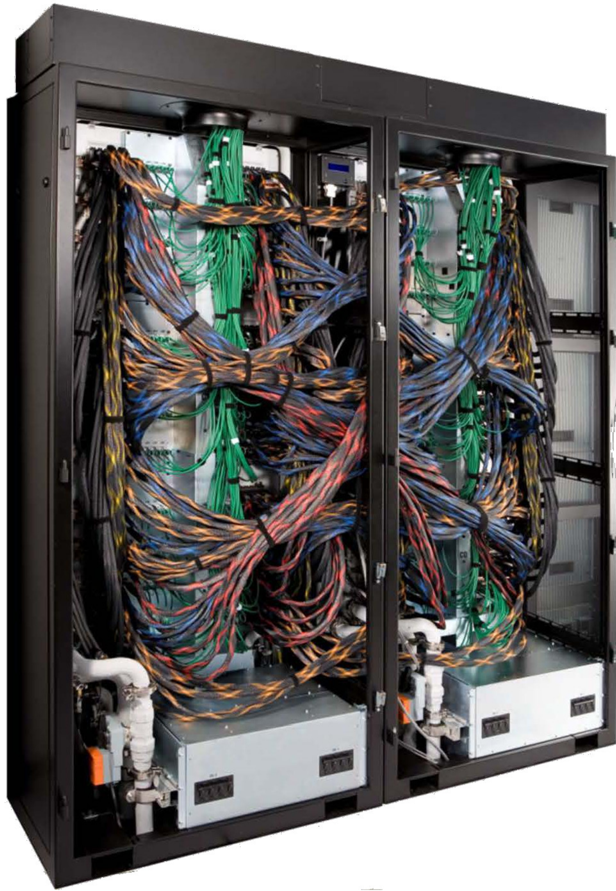


Fig. 1. Theorem 1 illustrated. Wiring is the hidden cost of parallel memory accesses. (Image: Cray XC40 cabinet).

Recommendation 3 : Abstract Parallel Machine for Cryptanalysts

Cryptanalysts should evaluate the asymptotic cost of their attacks on machines built assuming an unbounded supply of:

a) Identical cluster nodes with constant amount of CPU cores and RAM.
b) Constant-length network cables.

1D, 2D or 3D meshes of processors with a constant amount of local memory fit this description.

One could argue that we propose to replace an abstract and inadequate model of computation (the RAM model) by another abstract model of computation which does not actually correspond to any actual available machine (we really don't want to discuss the issues of power delivery and heat dissipation in a large 3D mesh...).

We still claim that the "2D/3D mesh" model of computation is *more realistic* than either the RAM or the PRAM model. In fact, some existing large computers have a comparable network topology. The "Sequoia" supercomputer (Lawrence Livermore National Laboratory, USA) is a 98,304-node IBM BlueGene/Q. It is made of 96 racks, each with 32 boards, each with 32 processor nodes (with 16 cores each). The nodes are connected by a custom 5D torus network of dimension $16 \times 12 \times 16 \times 16 \times 2$ (the last dimension connects two adjacent processors in the same board). Another machine, the "K Computer" (Riken Advanced Institute for Computational Science, Japan) has 88,128 nodes (864 cabinets of 96 nodes) with a custom "Tofu" interconnect. The nodes are arranged in a large $24 \times 18 \times 17$ torus of small $2 \times 3 \times 2$ tori.

6 Application #1: Collision Search and Double Encryption

We now provide several examples where discussing the cost of several attacks provides a better understanding of the actual security of cryptographic constructions.

As a warm-up, and to exercise the (rather constrained) model we recommended at the end of section 5, we describe machines for parallel collision search. We first describe a parallel machine that finds a single collision on a random n-bit function with optimal cost $2^{n/2}$, using $2^{3n/8}$ processors for $2^{n/8}$ units of time (less processors can be linearly traded for more time). This corresponds, for instance, to the computation of a discrete logarithm in a generic group.

We then describe a parallel machine that breaks double encryption in time $2^{3n/5}$ using $2^{3n/5}$ processors, and therefore cost $2^{6n/5}$. This demonstrates that the results announced in [45] also hold in our model.

6.1 Common Setting

Let us consider a function $f : \{0,1\}^n \to \{0,1\}^n$ for which collisions are to be found. Let $\mathcal{D} \subset \{0,1\}^n$ be a set of "distinguished points", with $|\mathcal{D}| = \theta 2^n$ (for

instance, the bitstrings beginning with $-\log_2\theta$ zero bits). The algorithm works by iterating the function f starting from a distinguished point and until another distinguished point is reached. The expected number of iterations is $1/\theta$. If no distinguished point has been reached after (say), $20/\theta$ iterations, then we have probably entered a cycle and need to restart from a fresh starting point. This induces a mapping $f^* : \mathcal{D} \to \mathcal{D} \cup \{\perp\}$. Finding collisions on f is reduced to finding collisions on f^*: from a collision $f^*(x) = f^*(y)$ (with $x \neq y$), it is possible to extract a collision on f by "walking the trails". The point is that f^* is defined over a smaller domain.

In order to obtain a small cost, the parallel machine should have the following characteristics: maximum parallelism and minimum memory per processor. We use P processors each with a constant amount of memory, connected in a 3D mesh. This machine has enough memory to hold P distinguished points, but this memory is distributed amongst all processors. Each distinguished point is associated to a processor which is "responsible" for it. This processor can for instance be identified by taking the distinguished point modulo P.

Each time a distinguished point is computed by a processor (of rank i), it has to be stored in the local memory of the responsible processor (of rank j). To accomplish this, processor i sends a message to processor j through the mesh network. Routing this single message to its destination requires $3P^{1/3}$ routing steps in the worst case. If all processors simultaneously send a message to a random destination, then a simple routing algorithm can route all the packets in time $\widetilde{\mathcal{O}}\left(P^{1/3}\right)$ with high probability [40]. The routing algorithm moves all messages to the correct coordinate on the x plane first, then on the y plane, and lastly on the z plane. The expected maximum number of messages piling-up on a single node is $\mathcal{O}(\log P)$ (this follows from the fact that the expected maximum bin load when n balls are randomly thrown into n bins is $\log n$).

We will enforce that routing a message through the mesh takes less time than generating a new distinguished point. This guarantees that the network is not saturated. We therefore want to choose θ such that $1/\theta = \Omega\left(P^{1/3}\right)$.

The machines work as follows. Each processor picks a random distinguished point x_0 (in a way to be specified below), iterates the function f by computing $x_{i+1} = f(x_i)$ until another distinguished point x_ℓ is found. It then sends a message through the mesh consisting of (x_0, ℓ, x_ℓ) to the processor responsible for x_ℓ. Upon reception of a message (x, y, z), two situations may occur:

- Either the processor (which is responsible for z) already holds a triplet with the same z and a different x. In this case a collision is detected. Both trails are to be walked again to precisely locate the collision.
- Otherwise, the incoming triplet is simply stored in the local single memory cell (it replaces anything previously stored).

6.2 Finding a Single Collision

Let $0 < \alpha < \frac{3}{8}$. We use $P = 2^{\alpha n}$ processors, with the following strategy: we adjust θ such that each processor only has to compute a single distinguished point.

It is known that the total number of iterations required to obtain a collision is $\sqrt{\frac{\pi}{2} 2^n} \approx 2^{n/2}$. This means that $\theta 2^{n/2}$ distinguished point will be computed in total; we want one distinguished point per processor, so $P = \theta 2^{n/2}$, from which we obtain $\theta = 2^{(\alpha - \frac{1}{2})n}$.

The machine works as follows: each processor (say its rank is i), starting from $x_0 = i$ (this is a distinguished point), iterates f until it reaches a distinguished point ; then it sends it through the mesh and stops. Once all distinguished points have been routed through the network, the collision should be identified.

Computing a distinguished point requires on average $1/\theta = 2^{(\frac{1}{2} - \alpha)n}$ iterations. Routing the distinguished points on the network requires $2^{\alpha n/3}$ routing steps. Because $\alpha < \frac{3}{8}$, we find that $1/\theta$ is always larger than $2^{\alpha n/3}$. The total time needed to find the collision by this parallel machine is then $T = 2^{(\frac{1}{2} - \alpha)n}$.

The cost of the machine is $2^{n/2}$. This matches the cost of sequential memoryless collision search (the "rho" method), and is thus optimal. Had we used a 2D mesh, we would have had to restrict $\alpha < \frac{1}{3}$ to enforce that computation takes as least as long as communications. Using a linear processor array, we would have had to enforce $\alpha < \frac{1}{4}$. In both cases, the optimal cost can still be attained, but with restricted parallelism.

6.3 Finding All Collisions: Application to Double Encryption

This corresponds to the case of meet-in-the middle attacks. Let us be given two functions $F, G : \{0,1\}^{\ell} \to \{0,1\}^n$. We want to find all collisions between F and G. For instance, to break double encryption with two known plaintext-ciphertext pairs (P_1, C_1) and (P_2, C_2), let $F(x) = E(x, P_1)$ and $G(y) = E^{-1}(y, C_1)$. Each collision $F(k_1) = G(k_2)$ suggests a potential key pair (k_1, k_2) for the double-encryption. We expect 2^n such collisions to exist, only one of which is the "golden collision" corresponding to the actual key. Checking if a suggested key is correct can be done by checking it against the second known pair.

To find a collision between F and G, we would like to iterate the following:

$$H(b \| x) = \begin{cases} F(x) & \text{if } b = 0, \\ G(x) & \text{otherwise} \end{cases} .$$

The problem is that H maps $n+1$ bits to n. To be able to iterate it, we compose it with a reasonably random predicate P (for instance we could use $P(x) = a \cdot x$ for some non-zero vector a). Therefore we define $\pi(x) = P(x) \| x$ and we iterate $H' = H \circ \pi$. A collision $H'(x) = H'(y)$ reveals a collision between F and G with probability 50%.

The (heuristic) analysis in [41] shows that the expected total number of iterations required per collision detected is $2^n \theta / w + 2/\theta$ using w words of memory; to find the golden collision, this has to be multiplied by 2^n. The value of θ that minimizes the total number of iterations is $\theta \approx \sqrt{w/2^n}$.

We use $P = 2^{\alpha n}$ processors with $\mathcal{O}(1)$ memory, with $0 < \alpha < \frac{3}{5}$. Just like in section 6.2, the upper-bound on α enforces that computing a distinguished

point is at least as long as routing it on the network. Each processor starts from a random distinguished point, iterates f until reaching a new distinguished point, routes the result through the mesh and repeats. The machine stops when the golden collision has been found (some technical details are hidden under the rug — the predicate P has to be changed regularly for instance. See [41]).

The expected running time is $2^{\frac{3}{2}(1-\alpha)n}$, and the machine has size $2^{\alpha n}$. Thus, it breaks double-encryption with cost $2^{\frac{3-\alpha}{2}n}$. Increasing the number of processors *reduces* the total number of sequential operations (because it increases the total amount of memory and reduces useless computations). However, after a certain threshold ($P = 2^{3n/5}$), increasing the number of processors stops being productive because the communication network cannot keep up. With $P = 2^{3n/5}$ the cost is $2^{6n/5}$: each processor computes $2^{2n/5}$ distinguished points; computing a single distinguished point requires $2^{n/5}$ sequential operations, and routing it through the 3D mesh requires as many routing steps.

If we restricted ourselves to 2D meshes, we would have had to pick $\alpha < \frac{1}{2}$ and the best cost would be $2^{5n/4}$. If only a linear array of processors were available, we would have had to pick $\alpha < \frac{1}{3}$ and the best cost would be $2^{4n/3}$. This example shows that communications can play a crucial role in cryptanalytic attacks, and that different communications models result in different costs. Thus, plainly assuming that communications are "free" (as in free beer) obviously yields more optimistic results. But things are likely to present themselves differently in reality.

Inferior Designs. We briefly consider other more costly possibilities. Using M memory cells per processor divides the running time by $\sqrt{M}$ but it multiplies the size of the machine by M — therefore it increases the cost.

The search for a "golden collision" with limited memory wastes some distinguished points that are overwritten without contributing to the collision search. To avoid this phenomenon (*i.e.* never "forgetting" a distinguished point), we could increase the number of processors to $P = \theta 2^n$. This, in turn, will make routing in the mesh slower, and requires us to decrease θ to maintain the balance between computation and communication. This leads to $\theta = 2^{-n/4}$ and $P = 2^{3n/4}$ processors. To find the "golden collision", 2^n collisions have to be found, and therefore (at least) 2^n distinguished points have to be computed. This takes time $2^{n/2}$, and therefore costs $2^{5n/4}$. It turns out that being forgetful and wasting computation enables the use of a smaller machine and allows reaching a better compromise.

7 Application #2: 3XOR Computation

The 3XOR problem has recently seen a renewed interest. It is a difficult case of the generalized birthday problem (with only three "lists"), and it can be used as a building block in more sophisticated attacks, for instance against the 2-round Even-Mansour construction mentioned above [30] or against the COPA authen-

ticated encryption mechanism [32]. A series of algorithms have been developed by Joux[22], Nikolić and Sasaki[33], Bouillaguet, Delaplace and Fouque [12].

7.1 3XOR on Pseudo-Random Functions

The problem comes in several flavors that are not completely equivalent. One of them is the following. Let $F_k, G_k, H_k : \{0,1\}^n \rightarrow \{0,1\}^n$ be three families of pseudo-random functions indexed by n-bit keys. We consider the problem of computing a 3XOR on (F, G, H):

- For a random key k, find x, y and z such that $F_k(x) \oplus G_k(y) \oplus H_k(z) = 0$.

A possible obvious sequential algorithm (the "quadratic algorithm") works as follows:

1. Prepare a hash table of size $2^{n/3}$ then for all $0 \leq i < 2^{n/3}$ do: $A[H_k(i)] \leftarrow i$.
2. Tabulate the other functions. For all $0 \leq i < 2^{n/3}$ do: $B[i] \leftarrow G_k(i)$ and $C[i] \leftarrow H_k(i)$.
3. Look for a match. For all $0 \leq j, k < 2^{n/3}$ do: set $u \leftarrow B[i] \oplus C[j]$. Probe u in the hash table. If there is a match, report $(i, j, A[u])$ as a solution.

It requires $2^{n/3}$ memory and $2^{2n/3}$ time — thus it costs 2^n. It is possible to do better with a simple idea called "clamping" by Bernstein in [4]: evaluate each function $2^{n/2}$ times, but discard all results that do not start with $n/4$ zero bits. This yields three "lists" of $\frac{3n}{4}$-bit strings of size $2^{n/4}$. These lists contain on average a single solution. This solution can be found in time $2^{n/2}$ and space $2^{n/4}$, thus with improved cost $2^{3n/4}$. This technique was used in [12] for an actual computation with $n = 96$.

But it is possible to do even better. First let $t \leftarrow H_k(0)$, then find a collision between F_k and $G_k \oplus t$: this yields a 3XOR for (F, G, H). The collision can be found by memory-less collision search in time $2^{n/2}$ and constant space, thus with cost $2^{n/2}$ — this seems hard to beat. The collision search can be parallelized with the same cost using the parallel architecture described in section 6.2.

7.2 3XOR on Arbitrary Arrays

Now, consider a different, more general flavor of the problem:

- Given three arrays of n-bit strings $\mathtt{A}, \mathtt{B}$ and $\mathtt{C}$ of respective size A, B and C, find a single (alternatively, find every) triple (i, j, k) such that $\mathtt{A}[i] \oplus \mathtt{B}[j] \oplus \mathtt{C}[k] = 0$.

All published 3XOR algorithms actually solve this particular problem — they start with the three "lists" $\mathtt{A}, \mathtt{B}$ and $\mathtt{C}$ in memory. In this setting, the question of the cost presents itself differently: because the machine has to be big enough to contain the lists, the costs are going to be much higher.

Smart Algorithms. Joux's algorithm [22] works when $A = B = 2^{n/2}/\sqrt{n/2}$ and $C = n/2$. It solves the problem in linear time and space, therefore it costs $2^n/n$. This improves upon the quadratic algorithm; it uses more data, more space and less time. This is a sequential algorithm, so its cost could be reduced by parallelization. However, there is a non-trivial obstruction: the algorithm requires the computation of the *join* of A and B: more precisely it needs to be able to enumerate the set

$$A \bowtie B = \left\{ x \oplus y \mid x \in A, y \in B, x \text{ and } y \text{ match on the first } n/2 \text{ bits} \right\}.$$

One way of computing this join is by sorting A and B using the first $n/2$ bits as keys, after which a single pass over the sorted arrays identifies the matching pairs. Recall from section 3.2 that sorting an array of size n costs $\mathcal{O}\left(n^{4/3}\right)$ using a 3D mesh. In our "mesh cost model" of computation, just this step in Joux's algorithm therefore costs at least $2^{2n/3}$. We therefore conjecture that it is impossible to implement Joux's algorithm with optimal cost.

The algorithm of Bouillaguet, Delaplace and Fouque [12] finds all the solutions in time $(A + B)C/n$, regardless of the sizes of the lists. This requires C/n (possibly parallel) join computations between A and B. Assuming that $A = B = C = 2^{n/3}$, all these joins costs at least $2^{7n/9}$. Again, we conjecture that this algorithm cannot be implemented with optimal cost. Note that it costs more than Joux's algorithm, but this is because the amount of data required is smaller so the problem is more difficult. In fact, with $A = B = C = 2^{n/2}$, clamping on $n/4$ bits produces an instance with lists of size $2^{n/4}$, so the sorting lower-bound on the cost of the algorithm of [12] drops to $2^{7n/12}$.

Naive Algorithms. We now describe a cost-optimal implementation of the quadratic algorithm, namely a machine of size N and running time N which finds all the solutions when $A = B = C = N$.

The problem lends itself well to a divide-and-conquer approach (already used in [12]): split the input arrays according to the most-significant bit of each entry, then solve the 4 sub-instances $(A_0, B_0, C_0), (A_1, B_0, C_1), (A_0, B_1, C_1)$ and (A_1, B_1, C_0). This could be done recursively, leading to $\mathcal{O}\left(N^2\right)$ sub-instances of expected constant size that could be solved on a single processor. Therefore, a very large, very parallel machine could potentially be built, which would solve the problem in constant time. The problem is that getting the input data to each processor in this very large machine would *not* require a constant amount of time — and it would be much longer than the actual computation.

Therefore, a balance has to be kept between the size of the machine (*i.e.* the amount of parallelism) and the time needed to propagate the input data inside it.

We proceed inductively. Assume we have a 2D mesh M_N of size $\sqrt{N} \times \sqrt{N}$ capable of solving instances of size N. Each M_N is capable of either ingesting or relaying the data it is fed on the top/left to the bottom/right after $\sqrt{N}$ steps. M_N needs $\sqrt{N}$ time steps to be fed the data (using $\sqrt{N}$ wires) and N computation steps to solve the problem. So it costs $\mathcal{O}\left(N^2\right)$.

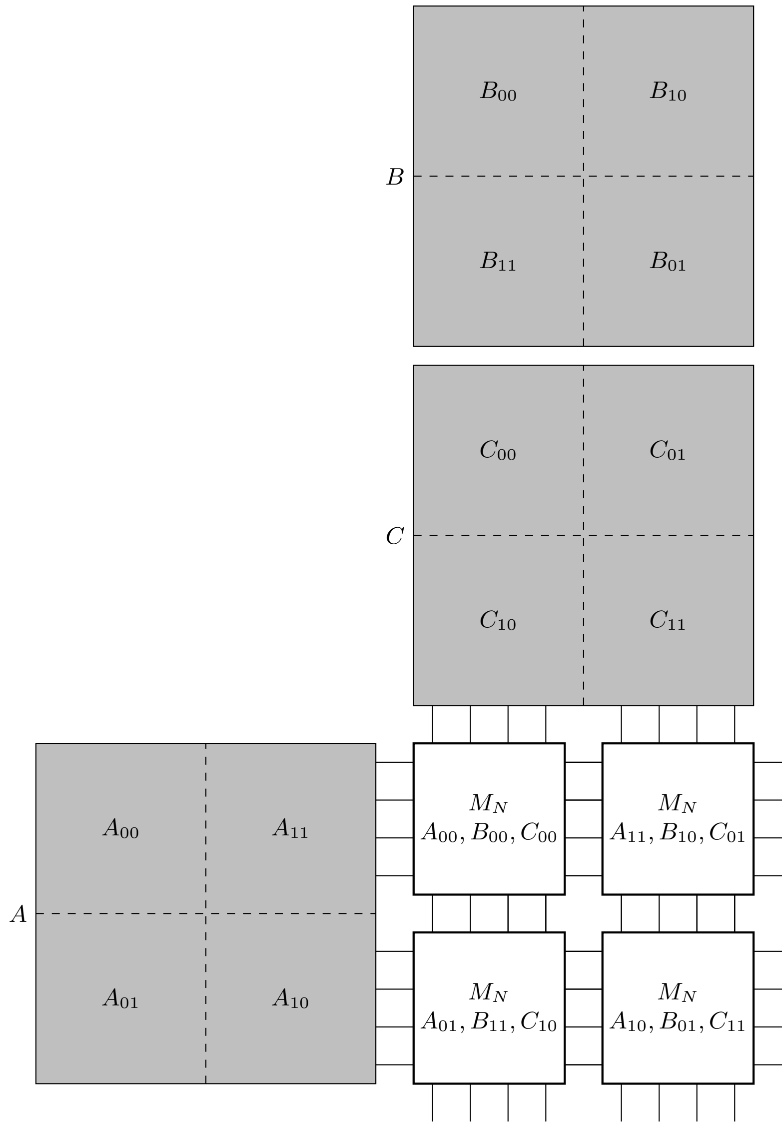
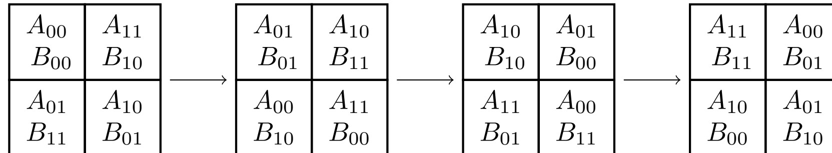


Fig. 2. Construction of M_{4N} from four copies of M_N: a 2D mesh for the quadratic algorithm.

We build M_{4N} as shown in fig. 2. The whole machine ingests A, B and C in time $\mathcal{O}\left(\sqrt{N}\right)$. The lists A and B have to be presented in a (recursive) snakelike order. In any case, the required ordering can be obtained by sorting in a preprocessing step of negligible cost. Once the sublists shown in fig 2 are loaded into the four copies of M_N, then four sub-instances out of 16 are solved in time N. To solve the 12 remaining sub-instances, some data juggling has to take place between the four copies of M_N:



The time needed to move the data at each step is upper-bounded by $\mathcal{O}\left(\sqrt{N}\right)$. So the total time needed by this machine is dominated by the computation — not by the data moving. In addition, with some buffering and at most twice more wires, it could be done in parallel with the computation, and thus not require any extra time.

Discussion. A simple and naive algorithm seems more cost-effective than sophisticated techniques. Because it does not rely on sorting (or equivalently, join computation), the quadratic algorithm require less expensive communications. We believe that, in this particular example, this in fact quite consistent with the practical work and observations of actual implementers. For instance, the "practical work" section of [12] considers parallelizing the join computations over several machines, and concludes that "*it is likely that the communication overhead will make this less efficient than the quadratic algorithm*".

The same authors [12] found their algorithm to be faster on a single machine than the quadratic algorithm. Combining these observations leads to an implementation strategy consistent with our cost analysis: to solve a large instance of the 3XOR problem, apply the cost-optimal algorithm (divide-and-conquer) until data fits inside a single compute node, then use the fastest possible algorithm.

8 Application #3: Multivariate Quadratic Equations

The MQ problem consists in solving systems of multivariate quadratic equations. It is well-known that this problem is NP-complete over finite fields; as such, it is one of the main hard problems on which "post-quantum" public-key cryptography relies. For the sake of simplicity, we only consider the case where the variables are binary, *i.e.* multivariate quadratic equations over $\mathbb{F}_2$. As such, we assume that $x^2 = x$ for all x. Given a list of m quadratic polynomials in n variables:

$$f_k(\mathbf{x}) = \sum_{i=1}^{n} \sum_{j=1}^{n} \alpha_{ijk} x_i x_j + \sum_{i=1}^{n} \beta_{ik} x_i + \gamma_k,$$

the problem consists in finding a vector $\mathbf{x} \in \mathbb{F}_2{}^n$ such that $f_1(\mathbf{x}) = \cdots = f_k(\mathbf{x}) = 0$. We assume that the input system is generic, meaning that it does not exhibit any special structure. It is widely believed that $m = n$ is the worst case, therefore we focus on this setting.

We discuss the cost of several algorithms: exhaustive search (that costs 2^n) and two "smart" algorithms. We argue that the algorithm of Lokshtanov, Paturi, Tamaki, Williams and Yu [31], with running time $2^{0.88n}$, in fact costs at least $2^{1.15n}$ (more than exhaustive search). We then turn our attention the BooleanSolve algorithm of Bardet, Faugere, Salvy and Spaenlehauer [2], with running time $2^{0.792n}$. We propose a proper tuning and a parallel implementation which, while not cost-optimal, costs $2^{0.837n}$. We thereby conclude that BooleanSolve can be asymptotically more cost-effective than exhaustive search.

We haven't been able to apply the same process to the CrossBred algorithm of Joux and Vitse [23], because a precise complexity analysis is not yet available. This algorithm is practically faster than exhaustive search as soon as $n \geq 40$, so we expect it be very cost-effective, but we cannot assert it yet.

8.1 Exhaustive Search

In its simplest form, exhaustive search tries all the 2^n possible solutions $\mathbf{x}$. Evaluating a single quadratic polynomial on an arbitrary vector requires $\mathcal{O}\left(n^2\right)$ operations. A possible strategy consists in finding all the solutions satisfying the first $\log n$ polynomials: there should be $2^n/n$ of them; Then, in a second stage, these partial solutions are checked against the remaining polynomials — which should take an asymptotically equivalent time. In total, solving the system should require $\mathcal{O}\left(n^2 \log n 2^n\right)$ operations and space $\mathcal{O}\left(n^3\right)$ (to store the coefficients of the polynomials). Therefore this simple strategy costs $n^5 \log n 2^n$. It is easily parallelizable at will.

An improved exhaustive search algorithm is described in [10]. It can enumerate the values of a polynomial on all possible input by doing only $\mathcal{O}(1)$ operations per trial, at the expense of $\mathcal{O}(n)$ additional space. Using it on the first $\log n$ polynomials reduces the time complexity to $\mathcal{O}(\log n 2^n)$ operations, and thus reduces the cost by n^2.

When one actually tries to implement algorithms, things always get more complicated. These two exhaustive search algorithms have been considered for hardware implementations on FPGAs [11]. It turns out that they both have the same (asymptotic) area complexity (they need to store the polynomial system), and they both can be implemented in a fully pipelined way. Thus, in hardware they would be almost equivalent — the notion of cost introduced above does not take pipelining into account. In the end, it is only because of concrete "details" (number of slices of each kind available on the given model of FPGA, etc.) that the "improved" algorithm was eventually a bit better than the very naive one.

8.2 The Algorithm of Lokshtanov, Paturi, Tamaki, Williams and Yu [31]

This probabilistic algorithm requires $\widetilde{\mathcal{O}}\left(2^{0.877n}\right)$ elementary operations, which is less than exhaustive search. It also has the interesting characteristic that, as opposed to other algebraic techniques, its complexity can be established without relying on hypothetical algebraic properties of the input polynomials, such semi-genericity (which only hold heuristically and have not been proved to exist).

We show that it is unlikely that a parallel implementation of this algorithm may cost less than exhaustive search, because it hits several of the obstructions we identified in section 3.2.

Let $0 < \delta < 1$ be a parameter to be determined later. The algorithm relies on the simple observation that $\mathbf{x} = (x_1, \ldots, x_n)$ is a solution of the input system if and only if $\mathbf{y} = (x_{\delta n+1}, \ldots, x_n)$ is a solution of the equation:

$$\prod_{\mathbf{a} \in {\mathbb{F}_2}^{\delta n}} \left(1 - \prod_{i=1}^{n} \left(1 - f_i(\mathbf{a}, \mathbf{y})\right)\right) = 0.$$

The main idea of the algorithm is to avoid working with this unwieldy polynomial (it has up to 2^n terms), but instead to use the following low-degree approximation:

$$R(\mathbf{y}) = \sum_{\mathbf{a} \in {\mathbb{F}_2}^{\delta n}} t_a \left(1 - \prod_{i=1}^{\delta n+2} \left(1 - \sum_{j=1}^{n} s_{aij} f_j(\mathbf{a}, \mathbf{y})\right)\right),$$

where the t_a and s_{aij} coefficients are chosen uniformly at random in $\mathbb{F}_2$. If $\mathbf{y}$ is the last part of an actual solution of the input system, then $R(\mathbf{y})$ is uniformly distributed in $\mathbb{F}_2$; otherwise, $R(\mathbf{y}) = 0$ with probability at least $3/4$.

The algorithm works as follows. Repeat $100n$ times: choose at random the t_a and s_{aij} coefficients; compute the polynomial R; evaluate $R(\mathbf{y})$ for all $\mathbf{y} \in {\mathbb{F}_2}^{(1-\delta)n}$. For each $\mathbf{y}$, keep a counter of the number of times this procedure results in $R(\mathbf{y}) \neq 0$. Any $\mathbf{y}$ for which this counter exceeds $40n$ is a solution of the input system with overwhelming probability.

Minimizing the Running Time. Enumerating all the values of R can be done in time (and space) $\widetilde{\mathcal{O}}\left(2^{(1-\delta)n}\right)$ using for instance the FFT-like algorithm called the Moebius transform by Joux [22]. Larger values of δ make this step faster but may potentially increase the size of the R: it is a polynomial in $(1-\delta)n$ variables of degree $2\delta n + \mathcal{O}(1)$. It therefore has $\binom{(1-\delta)n}{2\delta n}$ terms of degree $2\delta n$. The number of these terms asymptotically dominate the number of terms of all smaller degrees as long as $\delta < 1/5$. Taking approximations using Stirling's formula, the size of R is about $2^{n(1-\delta)H\left(\frac{2\delta}{1-\delta}\right)}$ terms, where H denotes the binary entropy function.

This R polynomial is the sum of $2^{\delta n}$ polynomials, themselves the product of δn quadratic polynomials in $(1-\delta)n$ variables. The complexity of computing

this multiplication is (at best) linear in the size of its output, Therefore, the time needed to compute R is about $2^{n[\delta+(1-\delta)H\left(\frac{2\delta}{1-\delta}\right)]}$.

Minimizing the running time means balancing the number of operations in the computation of R and its evaluation. The optimal value of δ is then the solution of $\delta + (1-\delta)H\left(\frac{2\delta}{1-\delta}\right) = 1-\delta$, which yields $\delta = 0.12375....$ This results in the announced time complexity of $2^{0.877n}$. The space complexity is the same.

Cost Analysis. In its sequential presentation, the algorithm costs more than $2^{7n/4}$. The problem of this algorithm is its huge space complexity: if a machine of size $2^{0.88n}$ could be built, then using it to run exhaustive search would require $2^{0.12n}$ time steps, instead of $2^{0.88n}$ time steps to run the algorithm of [31].

Can the cost of this algorithm be improved below 2^n by parallelization? We now argue that this seems quite unlikely. The problem is that the two dominating operations (polynomial multiplication and Moebius transform, which is in fact an FFT) are subject to a VLSI lower-bound of $AT = n^{3/2}$, as mentioned in section 3.2. It is therefore implausible that they can be implemented cost-optimally. Using 3D machines, we assume that both operations cost $n^{4/3}$.

The cost of computing R is then lower-bounded by $2^{\delta n + \frac{4}{3}n(1-\delta)H\left(\frac{2\delta}{1-\delta}\right)}$, and the cost of enumerating all the values of R is also lower-bounded by $2^{\frac{4}{3}n(1-\delta)}$. The balancing act between the cost of the two operations is then:

$$\delta + \frac{4}{3}(1-\delta)H\left(\frac{2\delta}{1-\delta}\right) = \frac{4}{3}(1-\delta),$$

and the optimal value of δ is now $\delta = 0.1319....$ The balanced cost is $2^{1.157n}$. Note that this is a lower-bound on the cost of the full algorithm.

We conclude that, in our opinion, this algorithm cannot possibly be relevant from a cryptographic point of view: running any concrete implementation will cost more than exhaustive search.

8.3 The BooleanSolve Algorithm of Bardet, Faugere, Salvy and Spaenlehauer [2]

This hybrid algorithm combines exhaustive search and algebraic manipulations in the style of Gröbner basis computations. Its time complexity is $2^{0.792n}$. We present a parallel implementation with cost $2^{0.837n}$. While not cost-optimal, it improves on the direct use of the sequential implementation and it "beats brute force".

The algorithm depends on a parameter γ, to be specified later. Again, let $\mathbf{y} = (x_{(1-\gamma)n+1}, \ldots, x_n)$. From a high level, the algorithm works as follows: for each $\mathbf{a} \in \mathbb{F}_2{}^{(1-\gamma)n}$, check if 1 can be written as a polynomial combination of $f_1(\mathbf{a}, \mathbf{y}), \ldots, f_n(\mathbf{a}, \mathbf{y})$. If so, the input polynomials "specialized in $\mathbf{a}$" have no solution. Otherwise, perform an exhaustive search to find $\mathbf{y}$.

Determining whether 1 can be written as a polynomial combination of given polynomials $f_1, \ldots, f_n$ can be decided by solving a linear system over $\mathbb{F}_2$: there

exists polynomials $g_1, \ldots, g_n$ of total degree less than d such that $f_1 g_1 + \cdots + f_n g_n = 1$ if and only if there exists a *linear* combination of $M \cdot f_i$ in which all monomials vanish except 1, where M ranges over all monomials of degree $\leq d$.

The problem is to choose the right value of d. Under the usual semi-regularity assumptions on the input polynomials, then it is enough to choose this d to be the index of the first negative coefficient of the power series $\frac{(1+t)^n}{(1-t)(1+t^2)^n}$. Using complex analysis techniques, an asymptotic estimation can be obtained:

$$d \sim n\gamma M\left(\frac{1}{\gamma}\right), \quad M(x) = -x + \frac{1}{2} + \frac{1}{2}\sqrt{2x^2 - 10x + 1 + 2(x+2)\sqrt{x(x+2)}}.$$

For the right value of d, this linear system can be written as $Ax = 0$, where the largest dimension of the matrix is $N = \widetilde{\mathcal{O}}\left(2^{n\gamma H(M(1/\gamma))}\right)$. This matrix is sparse, with at most n^2 non-zero coefficients per row. Solving such a linear system using naive dense gaussian elimination has time complexity $\mathcal{O}\left(N^3\right)$; because the matrix is so sparse, using a sparse iterative method such as Wiedemann's algorithm allows the system to be solved in time $\widetilde{\mathcal{O}}\left(N^2\right)$. We note that the matrix A admits a compact representation: the coefficients of each row can be easily recomputed from the input polynomials and d. Therefore, running the Wiedemann algorithm mostly requires the storage of a few vectors of size N.

Under additional genericity assumptions on the input system ("strong semi-regularity"), the exhaustive search phase will never be performed in vain (it will always yield a solution). We thus assume that it will be done only once.

The time complexity of the algorithm is then $2^{n[1-\gamma+2\gamma H(M(1/\gamma))]} + 2^{\gamma n}$.

Minimizing the Running Time. It is enough to find the minimum of $1 - \gamma + 2\gamma H(M(1/\gamma)$. This expression reaches a minimum of 0.792 when $\gamma = 0.55...$, and the exhaustive search phase is asymptotically negligible. This establishes the announced running time. Note that this amounts to doing an exhaustive search on the first $0.45n$ variables.

Cost Analysis. Using the same value of the parameter $\delta = 0.55...$, we find that the algorithm requires $2^{0.171n}$ bits of memory. So the cost of running the algorithm on a sequential computer is $2^{0.963n}$, which is slightly better than exhaustive search.

Solving the linear systems dominate the overall complexity. Running the Wiedemann algorithm sequentially costs $\widetilde{\mathcal{O}}\left(N^3\right)$, but this can be improved. Bernstein has shown [3] that a 2D mesh of total size $\widetilde{\mathcal{O}}\left(N\right)$ can perform the matrix-vector product in time $\widetilde{\mathcal{O}}\left(N^{1/2}\right)$ by reducing it to sorting. As pointed out by Wiener [45], a 3D mesh of the same size can sort a bit faster in time $\widetilde{\mathcal{O}}\left(N^{1/3}\right)$. It follows that solving one linear system using Wiedemann's algorithm and doing the matrix-vector products with this 3D mesh in fact costs $\widetilde{\mathcal{O}}\left(N^{7/3}\right)$.

Minimizing the cost of this parallel implementation of $\mathsf{BooleanSolve}$ means finding the minimum of $1 - \gamma + \frac{7}{3}\gamma H(M(1/\gamma)$, which is 0.836 when $\gamma = 0.416...$.

In this case, the space complexity drops to $2^{0.109n}$ and the cost decreases to $2^{0.837n}$. The total number of elementary operations *increases* to $2^{0.801n}$.

This amounts to doing an exhaustive search on the first $0.58n$ variables (again, improving the cost means doing more exhaustive search). Let $0 < \alpha \leq 0.58$. With $2^{(0.109+\alpha)n}$ processors, the input polynomial system can be solved in time $2^{(0.692-\alpha)n}$ (by exhaustively searching αn variables in parallel).

References

1. Bar-On, A., Dunkelman, O., Keller, N., Ronen, E., Shamir, A.: Improved key recovery attacks on reduced-round AES with practical data and memory complexities. In Shacham, H., Boldyreva, A., eds.: Advances in Cryptology - CRYPTO 2018 - 38th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2018, Proceedings, Part II. Volume 10992 of Lecture Notes in Computer Science., Springer (2018) 185–212
2. Bardet, M., Faugère, J., Salvy, B., Spaenlehauer, P.: On the complexity of solving quadratic boolean systems. J. Complexity **29**(1) (2013) 53–75
3. Bernstein, D.J.: Circuits for integer factorization: a proposal (2001) URL: `http://cr.yp.to/papers.html`.
4. Bernstein, D.J.: Better price-performance ratios for generalized birthday attacks (2007)
5. Bernstein, D.J., Lange, T.: Batch NFS. In Joux, A., Youssef, A.M., eds.: Selected Areas in Cryptography - SAC 2014 - 21st International Conference, Montreal, QC, Canada, August 14-15, 2014, Revised Selected Papers. Volume 8781 of Lecture Notes in Computer Science., Springer (2014) 38–58
6. Bernstein, D.J., Lange, T., Niederhagen, R., Peters, C., Schwabe, P.: FSBday: Implementing Wagner's Generalized Birthday Attack. In: INDOCRYPT. (2009) 18–38
7. Bilardi, G., Preparata, F.P.: Area-time lower-bound techniques with applications to sorting. Algorithmica **1**(1) (1986) 65–91
8. Biryukov, A., Dinu, D., Khovratovich, D.: Argon2: New generation of memory-hard functions for password hashing and other applications. In: IEEE European Symposium on Security and Privacy, EuroS&P 2016, Saarbrücken, Germany, March 21-24, 2016, IEEE (2016) 292–302
9. Boneh, D., ed.: Advances in Cryptology - CRYPTO 2003, 23rd Annual International Cryptology Conference, Santa Barbara, California, USA, August 17-21, 2003, Proceedings. Volume 2729 of Lecture Notes in Computer Science., Springer (2003)
10. Bouillaguet, C., Chen, H., Cheng, C., Chou, T., Niederhagen, R., Shamir, A., Yang, B.: Fast exhaustive search for polynomial systems in F_2. In Mangard, S., Standaert, F., eds.: Cryptographic Hardware and Embedded Systems, CHES 2010, 12th International Workshop, Santa Barbara, CA, USA, August 17-20, 2010. Proceedings. Volume 6225 of Lecture Notes in Computer Science., Springer (2010) 203–218
11. Bouillaguet, C., Cheng, C., Chou, T., Niederhagen, R., Yang, B.: Fast exhaustive search for quadratic systems in $$\mathbb {F}_{2}$$ on fpgas. In Lange, T., Lauter, K.E., Lisonek, P., eds.: 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., Springer (2013) 205–222
12. Bouillaguet, C., Delaplace, C., Fouque, P.: Revisiting and improving algorithms for the 3xor problem. IACR Trans. Symmetric Cryptol. **2018**(1) (2018) 254–276
13. Brent, R.P., Kung, H.T.: The chip complexity of binary arithmetic. In: Proceedings of the Twelfth Annual ACM Symposium on Theory of Computing. STOC '80, New York, NY, USA, ACM (1980) 190–200
14. Casanova, A., Faugère, J.C., Macario-Rat, G., Patarin, J., Perret, L., Ryckeghem, J.: GeMSS: A Great Multivariate Short Signature. Research report, UPMC - Paris 6 Sorbonne Universités ; INRIA Paris Research Centre, MAMBA Team, F-75012, Paris, France ; LIP6 - Laboratoire d'Informatique de Paris 6 (December 2017)
15. Delcourt, M., Kleinjung, T., Lenstra, A., Nath, S., Page, D., Smart, N.: Using the cloud to determine key strengths – triennial update. Cryptology ePrint Archive, Report 2018/1221 (2018) `https://eprint.iacr.org/2018/1221`.
16. Dinur, I., Dunkelman, O., Keller, N., Shamir, A.: Key recovery attacks on iterated even-mansour encryption schemes. J. Cryptology **29**(4) (2016) 697–728
17. Dinur, I., Dunkelman, O., Shamir, A.: Improved attacks on full GOST. In Canteaut, A., ed.: Fast Software Encryption - 19th International Workshop, FSE 2012, Washington, DC, USA, March 19-21, 2012. Revised Selected Papers. Volume 7549 of Lecture Notes in Computer Science., Springer (2012) 9–28
18. Dubois, V., Fouque, P., Shamir, A., Stern, J.: Practical cryptanalysis of SFLASH. In Menezes, A., ed.: Advances in Cryptology - CRYPTO 2007, 27th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2007, Proceedings. Volume 4622 of Lecture Notes in Computer Science., Springer (2007) 1–12
19. Dwork, C., Goldberg, A.V., Naor, M.: On memory-bound functions for fighting spam. [9] 426–444
20. Isobe, T.: A single-key attack on the full GOST block cipher. In Joux, A., ed.: Fast Software Encryption - 18th International Workshop, FSE 2011, Lyngby, Denmark, February 13-16, 2011, Revised Selected Papers. Volume 6733 of Lecture Notes in Computer Science., Springer (2011) 290–305
21. Isobe, T., Shibutani, K.: New key recovery attacks on minimal two-round even-mansour ciphers. In Takagi, T., Peyrin, T., eds.: Advances in Cryptology - ASIACRYPT 2017 - 23rd International Conference on the Theory and Applications of Cryptology and Information Security, Hong Kong, China, December 3-7, 2017, Proceedings, Part I. Volume 10624 of Lecture Notes in Computer Science., Springer (2017) 244–263
22. Joux, A.: Algorithmic cryptanalysis. CRC Press (2009)
23. Joux, A., Vitse, V.: A crossbred algorithm for solving boolean polynomial systems. In Kaczorowski, J., Pieprzyk, J., Pomykala, J., eds.: Number-Theoretic Methods in Cryptology - First International Conference, NuTMiC 2017, Warsaw, Poland, September 11-13, 2017, Revised Selected Papers. Volume 10737 of Lecture Notes in Computer Science., Springer (2017) 3–21
24. Kleinjung, T., Aoki, K., Franke, J., Lenstra, A.K., Thomé, E., Bos, J.W., Gaudry, P., Kruppa, A., Montgomery, P.L., Osvik, D.A., te Riele, H.J.J., Timofeev, A., Zimmermann, P.: Factorization of a 768-bit RSA modulus. In Rabin, T., ed.: Advances in Cryptology - CRYPTO 2010, 30th Annual Cryptology Conference, Santa Barbara, CA, USA, August 15-19, 2010. Proceedings. Volume 6223 of Lecture Notes in Computer Science., Springer (2010) 333–350

25. Kleinjung, T., Diem, C., Lenstra, A.K., Priplata, C., Stahlke, C.: Computation of a 768-bit prime field discrete logarithm. In Coron, J., Nielsen, J.B., eds.: Advances in Cryptology - EUROCRYPT 2017 - 36th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Paris, France, April 30 - May 4, 2017, Proceedings, Part I. Volume 10210 of Lecture Notes in Computer Science. (2017) 185–201
26. Kleinjung, T., Lenstra, A.K., Page, D., Smart, N.P.: Using the cloud to determine key strengths. In Galbraith, S., Nandi, M., eds.: Progress in Cryptology - INDOCRYPT 2012, Berlin, Heidelberg, Springer Berlin Heidelberg (2012) 17–39
27. Kumar, S.S., Paar, C., Pelzl, J., Pfeiffer, G., Schimmler, M.: Breaking ciphers with COPACOBANA - A cost-optimized parallel code breaker. In Goubin, L., Matsui, M., eds.: 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., Springer (2006) 101–118
28. Kuo, P., Schneider, M., Dagdelen, Ö., Reichelt, J., Buchmann, J.A., Cheng, C., Yang, B.: Extreme enumeration on GPU and in clouds - - how many dollars you need to break SVP challenges -. In Preneel, B., Takagi, T., eds.: Cryptographic Hardware and Embedded Systems - CHES 2011 - 13th International Workshop, Nara, Japan, September 28 - October 1, 2011. Proceedings. Volume 6917 of Lecture Notes in Computer Science., Springer (2011) 176–191
29. Lenstra, A.K., Shamir, A., Tomlinson, J., Tromer, E.: Analysis of bernstein's factorization circuit. In Zheng, Y., ed.: Advances in Cryptology - ASIACRYPT 2002, 8th International Conference on the Theory and Application of Cryptology and Information Security, Queenstown, New Zealand, December 1-5, 2002, Proceedings. Volume 2501 of Lecture Notes in Computer Science., Springer (2002) 1–26
30. Leurent, G., Sibleyras, F.: Low-memory attacks against two-round even-mansour using the 3-xor problem. In Boldyreva, A., Micciancio, D., eds.: 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., Springer (2019) 210–235
31. Lokshtanov, D., Paturi, R., Tamaki, S., Williams, R.R., Yu, H.: Beating brute force for systems of polynomial equations over finite fields. In Klein, P.N., ed.: Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2017, Barcelona, Spain, Hotel Porta Fira, January 16-19, SIAM (2017) 2190–2202
32. Nandi, M.: Revisiting Security Claims of XLS and COPA. IACR Cryptology ePrint Archive **2015** (2015) 444
33. Nikolić, I., Sasaki, Y.: Refinements of the k-tree Algorithm for the Generalized Birthday Problem. In: ASIACRYPT, Springer (2014) 683–703
34. Nikolic, I., Wang, L., Wu, S.: Cryptanalysis of round-reduced `led`. In Moriai, S., ed.: Fast Software Encryption - 20th International Workshop, FSE 2013, Singapore, March 11-13, 2013. Revised Selected Papers. Volume 8424 of Lecture Notes in Computer Science., Springer (2013) 112–129
35. Savage, J.E.: Area—time tradeoffs for matrix multiplication and related problems in vlsi models. Journal of Computer and System Sciences **22**(2) (1981) 230 – 242
36. Stevens, M., Bursztein, E., Karpman, P., Albertini, A., Markov, Y.: The first collision for full sha-1. In Katz, J., Shacham, H., eds.: Advances in Cryptology – CRYPTO 2017, Cham, Springer International Publishing (2017) 570–596
37. Stevens, M., Lenstra, A.K., de Weger, B.: Chosen-prefix collisions for MD5 and colliding X.509 certificates for different identities. In Naor, M., ed.: Advances

in Cryptology - EUROCRYPT 2007, 26th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Barcelona, Spain, May 20-24, 2007, Proceedings. Volume 4515 of Lecture Notes in Computer Science., Springer (2007) 1–22

38. Thompson, C.D.: Area-time complexity for vlsi. In: Proceedings of the Eleventh Annual ACM Symposium on Theory of Computing. STOC '79, New York, NY, USA, ACM (1979) 81–88
39. Thompson, C.D., Kung, H.T.: Sorting on a mesh-connected parallel computer. Commun. ACM **20**(4) (April 1977) 263–271
40. Valiant, L.G., Brebner, G.J.: Universal schemes for parallel communication. In: Proceedings of the Thirteenth Annual ACM Symposium on Theory of Computing. STOC '81, New York, NY, USA, ACM (1981) 263–277
41. van Oorschot, P.C., Wiener, M.J.: Parallel collision search with cryptanalytic applications. J. Cryptology **12**(1) (1999) 1–28
42. Vasilakis, E.: An instruction level energy characterization of ARM processors. Technical Report MSU-CSE-06-2, Computer Architecture and VLSI Systems (CARV) Laboratory, Institute of Computer Science (ICS), Foundation of Research and Technology Hellas (FORTH) (March 2015) Technical Report FORTH-ICS/TR-450.
43. Vitányi, P.: Locality, communication, and interconnect length in multicomputers. SIAM Journal on Computing **17**(4) (1988) 659–672
44. Vuillemin, J.: A combinatorial limit to the computing power of vlsi circuits. IEEE Trans. Comput. **32**(3) (March 1983) 294–300
45. Wiener, M.J.: The full cost of cryptanalytic attacks. J. Cryptology **17**(2) (2004) 105–124
46. Wikipedia contributors: Triple des — Wikipedia, the free encyclopedia (2019) [Online; accessed 11-September-2019].
47. Wulf, W.A., McKee, S.A.: Hitting the memory wall: implications of the obvious. SIGARCH Computer Architecture News **23**(1) (1995) 20–24