Using the Cloud to Determine Key Strengths

T. Kleinjung[1], A.K. Lenstra[1], D. Page[2], and N.P. Smart[2]

[1] EPFL IC LACAL, Station 14, CH-1015 Lausanne, Switzerland.
[2] Dept. Computer Science, University of Bristol, Merchant Venturers Building, Woodland Road, Bristol, BS8 1UB, United Kingdom.

Abstract. We develop a new methodology to assess cryptographic key strength using cloud computing, by calculating the true economic cost of (symmetric- or private-) key retrieval for the most common cryptographic primitives. Although the present paper gives the current costs, more importantly it provides the tools and infrastructure to derive new data points at any time in the future, while allowing for improvements such as of new algorithmic approaches. Over time the resulting data points will provide valuable insight in the selection of cryptographic key sizes. [3]

1 Introduction

An important task for cryptographers is the analysis and recommendation of parameters, crucially including key size and thus implying key strength, for cryptographic primitives; clearly this is of theoretic and practical interest, relating to the study and the deployment of said primitives respectively. As a result, considerable effort has been and is being expended with the goal of providing meaningful data on which such recommendations can be based. Roughly speaking, two main approaches dominate: use of special-purpose hardware designs, including proposals such as [14, 15, 37, 38] (some of which have even been realised), and use of more software-oriented (or at least less bespoke) record setting computations such as [4, 5, 17]. The resulting data can then be extrapolated using complexity estimates for the underlying algorithms, and appropriate versions of Moore's Law, in an attempt to assess the longevity of associated keys. In [19] this results in key size recommendations for public-key cryptosystems that offer security comparable to popular symmetric cryptosystems; in [23] it leads to security estimates in terms of hardware cost or execution time. Existing work for estimating symmetric key strengths (as well as other matters) is discussed in [39].

It is not hard to highlight disadvantages in these approaches. Some special-purpose hardware designs are highly optimistic; they all suffer from substantial upfront costs and, as far as we have been able to observe, are always harder to get to work and use than expected. On the other hand, even highly speculative designs may be useful in exploring new ideas and providing insightful lower bounds. Despite being more pragmatic, software-oriented estimates do not necessarily cater adequately for the form or performance of future generations of general-purpose processors (although so far straightforward application of Moore's Law is remarkably reliable, with various dire prophecies, such as the "memory wall", not materialising yet). For some algorithms, scaling up effort (e.g., focusing on larger keys) requires no more than organisational skill combined with patience. Thus, record setting computation that does not involve new ideas may have little or no scientific value: for the purposes of assessing key strength, a partial calculation is equally valuable. For some other algorithms, only the full computation adequately prepares for problems one may encounter when scaling up, and overall (in)feasibility may thus yield useful information. Finally, for neither the hardware- nor software-oriented approach, is there a uniform, or even well understood, metric by which "cost" should be estimated. For example, one often overlooks the cost of providing power and cooling, a foremost concern for modern, large-scale installations.

Despite these potential disadvantages and the fact that papers such as [19] and [23] are already a decade old, their results have proved to be quite resilient and are widely used. This can be explained by the fact

[3] We see this as a living document, which will be updated as algorithms, the Amazon pricing model, and other factors change. The current version is V1.0 of May 2011; updates will be posted on `http://www.cs.bris.ac.uk/~nigel/Cloud-Keys/`.

that standardisation of key size requires some sort of long term extrapolation: there is no choice but to take the inherent uncertainty and potential unreliability for granted. In this paper we propose to complement traditional approaches, using an alternative that avoids reliance on special-purpose hardware, one time experiments, or record calculations, and that adopts a business-driven and thus economically relevant cost model. Although extrapolations can never be avoided, our approach minimises their uncertainty because whenever anyone sees fit, he/she can use commodity hardware to repeat the experiments and verify and update the cost estimates.

The current focus on cloud computing is widely described as a significant long-term shift in how computing services are delivered. In short, cloud computing enables any party to rent a combination of computational and storage resources that exist within managed data centers; the provider we use is the Amazon Elastic Compute Cloud [3], however others are available. The crux of our approach is the use of cloud computing to assess key strength (for a specific cryptographic primitive) in a way that provides a useful relationship to a true economic cost. Crucially, we rely on the fact that cloud computing providers operate as businesses: assuming they behave rationally, the pricing model for each of their services takes into account the associated purchase, maintenance, power and replacement costs. In order to balance reliability of revenue against utilisation, it is common for such a pricing model to incorporate both long-term and supply and demand driven components. We return to the issue of supply and demand below, but for the moment assume this has a negligible effect on the longer-term pricing structure of Amazon in particular. As a result, the Amazon pricing model provides a valid way to attach a monetary cost to key strength. A provider may clearly be expected to update their infrastructures and pricing model as technology and economic conditions dictate. However, by ensuring our approach is repeatable, for example using commodity cloud computing services and platform agnostic software, we are able to track results as they evolve over time. We suggest this, and the approach as a whole, should therefore provide a robust understanding of how key size recommendations should be made.

In Section 2 we briefly explain our approach and those aspects of the Amazon Elastic Compute Cloud that we depend on. In Section 3 we describe our analysis applied to a number of cryptographic primitives, namely DES, AES, SHA-2, RSA and ECC. Sections 4 and 5 contain concluding remarks. Throughout, all monetary quantities are given in US dollars.

2 The Amazon Elastic Compute Cloud

Amazon Elastic Compute Cloud (EC2) is a web-service that provides computational and storage resource "in the cloud" (i.e., on the Internet) to suit the specific needs of each user. In this section we describe the current (per February 2011) EC2 hardware platform and pricing model, from a point of view that is relevant for our purposes, and then our approach.

EC2 Compute Units. At a high level, EC2 consists of numerous installations (or data centers) each housing numerous processing nodes (a processor core with some memory and storage) which can be rented by users. In an attempt to qualify how powerful a node is, EC2 uses the notion of an *EC2 Compute Unit*, or *ECU* for short. One ECU provides the equivalent computational capacity of a 1.0-1.2 GHz Opteron or Xeon processor circa 2007, which were roughly state of the art when EC2 launched. When new processors are deployed within EC2, they are given an ECU-rating; currently there are at least four different types of core, rated at 1, 2, 2.5 and 3.25 ECUs. The lack of rigour in the definition of an ECU (e.g., identically clocked Xeon and Opteron processors do not have equivalent computational capacity) is not a concern for our purposes.

Instances. An *instance* refers to a specified amount of dedicated compute capacity that can be purchased: it depends on the processor type (e.g., 32- or 64-bit), the number of (virtualised) cores (1, 2, 4 or 8), the ECU-rating per core, memory and storage capacity, and network performance. There are currently eleven different instances, partitioned into the *instance types* termed "standard", "micro", "high-memory", "high-CPU", "cluster compute", and "cluster GPU". The later two cluster types are intended for High Performance Computing (HPC) applications, and come with 10 Gb ethernet connections; for all other instance types except micro, there are two or three subinstances of different sizes, indicated by "Large (L)", "Extra Large

(EL)" and so on. Use of instances can be supported by a variety of operating systems, ranging from different versions of Unix and Linux through to Windows.

Pricing model. Instances are charged per instance-hour depending on their capacity and the operating system used. In 2007 this was done at a flat rate of $ 0.10 per hour on a 1.7 GHz processor with 1.75 GB of memory [1]; in 2008 the pricing model used ECUs charged at the same flat rate of $ 0.10 per hour per ECU [2]. Since then the pricing model has evolved [3]. Currently, instances can be purchased at three different price bands, "on-demand", "reserved" and "spot", charged differently according to which of four different geographic locations one is using (US east coast, US west coast, Ireland or Singapore).

On-demand pricing allows purchase of instance-hours as and when they are needed. After a fixed annual (or higher triennial) payment per instance, reserved pricing is significantly cheaper per hour than on-demand pricing: it is intended for parties that know their requirements, and hence can reserve them, ahead of when they are used. Spot pricing is a short-term, market determined price used by EC2 to sell off unused capacity.

For all instances and price bands, Windows usage is more expensive than Linux/Unix and is therefore not considered. Similarly, 32-bit instances are not considered, because for all large computing efforts and price bands 32-bit cores are at least as expensive as 64-bit ones. Table 1 lists the current pricing of the relevant remaining instances, of which k-fold multiples can be purchased at k times the price listed, for any positive integer k. Although of course there is a clear upper bound on k due to the size of the installation of the cloud service. To simplify our cost estimate we ignore this upper bound, and assume that the provider will provide more capacity (at the same cost) as demand increases. Separate charges are made for data transfer and so on.

Table 1. February 2011 US east coast instance pricing, in US dollars, using 64-bit Linux/Unix.

Instance	cores	ECUs per core	total ECUs	M2050 GPUs	RAM GB	on-demand per hour rate δ	reserved fixed payment 1 yr α	reserved fixed payment 3 yr τ	reserved per hour rate ϵ
standard L	2	2	4	0	7.5	$0.34	$910	$1400	$0.12
high-memory EL	2	3.25	6.5	0	17.1	$0.50	$1325	$2000	$0.17
high-CPU EL	8	2.5	20	0	7	$0.68	$1820	$2800	$0.24
cluster compute	(not specified)		33.5	0	23	$1.60	$4290	$6590	$0.56
cluster GPU			33.5	2	22	$2.10	$5630	$8650	$0.74

With, for a given instance, δ, α, τ, and ϵ the four pricing parameters as indicated in Table 1 and using Y for the number of hours per year, it turns out that

$$1.962(\tau + 3\epsilon Y) < 3\delta Y < 2.032(\tau + 3\epsilon Y). \tag{1}$$

That is, for any instance using on-demand pricing continuously for a triennial period is approximately twice as expensive as using reserved pricing with a triennial term for the entire triennial period. Furthermore, for all instances reserved pricing for three consecutive (or parallel) annual periods is approximately 1.3 times as expensive as a single triennial period:

$$1.292(\tau + 3\epsilon Y) < 3(\alpha + \epsilon Y) < 1.306(\tau + 3\epsilon Y). \tag{2}$$

There are more pricing similarities between the various instances. Suppose that one copy of a given instance is used for a fixed number of hours h. We assume that h is known in advance and that the prices do not change during this period (cf. remark below on future developments). Which pricing band(s) should be used to obtain the lowest overall price depends on h in the following way. For small h use on-demand pricing, if h is larger than a first cross-over point γ_α but at most one year, reserved pricing with one year term must be used instead. Between one year and a second cross-over point γ_τ one should use reserved pricing with one year term for a year followed by on-demand pricing for the remaining $h - Y$ hours, but for longer periods

up to three years one should use just reserved pricing with a triennial term. After that the pattern repeats. This holds for all instances, with the cross-over values varying little among them, as shown below.

The first cross-over value γ_α satisfies $\delta\gamma_\alpha = \alpha + \epsilon\gamma_\alpha$ and thus $\gamma_\alpha = \frac{\alpha}{\delta-\epsilon}$. The second satisfies $\alpha + \epsilon Y + \delta(\gamma_\tau - Y) = \tau + \epsilon\gamma_\tau$ and thus $\gamma_\tau = Y + \frac{\tau-\alpha}{\delta-\epsilon}$. We find that $4015 < \gamma_\alpha < 4140$ and $Y + 2045 < \gamma_\tau < Y + 2228$. It follows that it is always the case that γ_α is somewhat less than half a year and that γ_τ is approximately a year and a quarter.

Our requirements and approach. For each cryptographic primitive studied, our approach hinges on the use of EC2 to carry out a negligible yet representative fraction of a certain computation. Said computation, when carried to completion, should result in a (symmetric- or private-) key or a collision depending on the type of primitive. For DES, AES, and SHA-2 this consists of a fraction of the symmetric-key or collision search, for RSA of small parts of the sieving step and the matrix step of the Number Field Sieve (NFS) integer factorisation method, and for ECC a small number of iterations of Pollard's rho method for the calculation of a certain discrete logarithm.

Note that in each case, the full computation would require at least many thousands of years when executed on a single core[4]. With the exception of the NFS matrix step and disregarding details, each case allows embarrassing parallelisation with only occasional communication with a central server (for distribution of inputs and collection of outputs). With the exception of the NFS sieving and matrix steps, memory requirements are negligible. Substantial storage is required only at a single location. As such, storage needs are thus not further discussed. Thus, modulo details and with one exception, anything we could compute on a single core in y years, can be calculated on n such cores in y/n years, for any $n > 0$.

Implementing a software component to perform each partial computation on EC2 requires relatively little upfront cost with respect to development time. In addition, execution of said software also requires relatively little time (compared to the full computation), and thus the partial computation can be performed using EC2's most appropriate pricing band for short-term use; this is the only actual cost incurred. Crucially, it results in a reliable estimate of the number of ECU years, the best instance(s) for the full computation, and least total cost (as charged by EC2) to do so (depending on the desired completion time). Obviously the latter cost(s) will be derived using the most appropriate applicable long-term pricing band.

Minimal average price. Let $\mu(h)$ denote the minimal average price per hour for a calculation that requires h hours using a certain fixed instance. Then we have

$$\mu(h) = \begin{cases} \delta & \text{for } 0 < h \le \gamma_\alpha \quad \text{(constant global maximum)} \\ \frac{\alpha}{h} + \epsilon & \text{for } \gamma_\alpha < h \le Y \quad \text{(with a local minimum at } h = Y\text{)} \\ \frac{\alpha + \epsilon Y + (h-Y)\delta}{h} & \text{for } Y < h \le \gamma_\tau \quad \text{(with a local maximum at } h = \gamma_\tau\text{)} \\ \frac{\tau}{h} + \epsilon & \text{for } \gamma_\tau < h \le 3Y \text{ (with the global minimum at } h = 3Y\text{).} \end{cases}$$

For $h > 3Y$ the pattern repeats, with each triennial period consisting of four segments, reaching decreasing local maxima at $h = 3kY + \gamma_\alpha$, decreasing local minima at $h = 3kY + Y$, decreasing local maxima at $h = 3kY + \gamma_\tau$, and the global minimum at $h = 3kY + 3Y$, for $k = 1, 2, 3, \ldots$. For all relevant instances, Figure 1 depicts the graphs of the minimal average prices for periods of up to six years.

Consequences. Given the embarrassingly parallel nature and huge projected execution time of each full computation, the above implies that we can always reach lowest (projected) cost by settling for a three year (projected) completion time. Faster completion would become gradually more expensive until completion time $\frac{\tau}{\alpha}Y > Y$ is reached[5], at which point one should switch right away to a shorter completion time of one year; faster than one year again becomes gradually more expensive until γ_α is reached, at which point the cost has reached its (constant) maximum and the completion time is only limited (from below) by the number of available on-demand copies of the required instance. We stress yet again that this all refers to projected

[4] The processor type can be left unspecified but quantum processors are excluded; to the best of our knowledge and consistent with all estimates of the last two decades, it will always take at least another decade before such processors are operational.

[5] Note that $1.509 < \frac{\tau}{\alpha} < 1.539$, so $\frac{\tau}{\alpha}Y$ is about a year and a half for all instances.

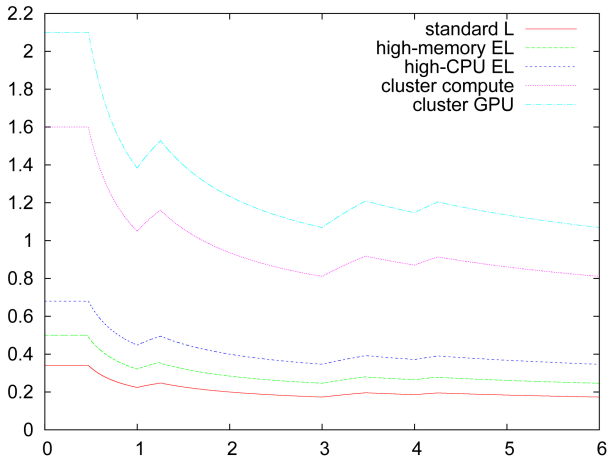

Fig. 1. February 2011 minimal average prices per hour, in US dollars, as a function of the number of years required by the calculation.

computation, none of which is actually completed: all we need is an estimate of the cost of completion, possibly depending on the desired completion time.

Thus assume that using a certain fixed EC2-instance, a full-scale security assessment is expected to take y years for a y that will be at least many thousands. The full computation can be completed using that instance by purchasing about $y/3$ copies of it using reserved pricing with a triennial term, and to use each copy for the full triennial period. Whether or not this (i.e., doing this full computation over the triennial period) is also optimal for that instance depends on issues beyond our control (such as changed pricing and inflation) and is commented on below. Using the same instance, the entire calculation can also be completed in one year at ≈ 1.3 times higher cost (cf. Inequality (2)), or at double the cost in an arbitrarily short period of time (assuming enough copies of the instance are available in EC2, cf. Inequality (1)).

With all instances behaving so similarly over time for the different price bands, the best instance for a particular calculation is selected by taking the one with the lowest hourly rate per ECU as long as it provides adequate memory and network bandwidth per core. From Table 1 and the description of our needs, it follows that the high-CPU EL instance will be most useful to us: using EC2 for a computation that is estimated to require y ECU years for some large y will cost about $\$\,\frac{0.68yY}{2\cdot 20} \approx 150y$ (cf. Inequality (1)) if we can afford to wait three years until completion. Completing it in a year will cost about $\$\,195y$ (cf. Inequality (2)), and doing it as fast as possible will cost $\$\,300y$ (cf. Inequality (1)).

In a similar fashion we derive costs $\$\,\frac{1.60yY}{2\cdot 33.5} \approx 210y$, $\$\,272y$ and $\$\,420y$ for a y ECU year calculation done in three years, one year, and "ASAP", respectively, using EC2 cluster-compute instances, and costs $\$\,\frac{0.50yY}{2\cdot 6.5} \approx 340y$, $\$\,440y$ and $\$\,675y$ for high-memory EL instances. Thus, cluster compute and high-memory EL instances are approximately 40% and 120% more expensive than high-CPU EL instances.

Accommodating future developments. Prices and pricing models will change over time, and so may security assessment strategies and their interaction with advances in processor design and manufacture. In particular, one could imagine that if a party decided to use the Amazon cloud for key recovery or collision search then the increase in demand would induce Amazon to increase the instance costs. However, we assume that the effect of such supply and demand on the pricing is relatively constant over a long period of time. Thus, we assume the non-spot prices are a relatively accurate reflection of the actual economic cost to Amazon (bar a marginal profit) of providing the service.

The cost estimates produced by our approach are valid at the moment they are calculated (in the way set forth above), but cannot take any future developments into account. However, this problem can be mitigated by adopting an open source model for the software components and using (as far as is sensible) platform agnostic programming techniques; for example, this permits the software to maintain alignment with the

latest algorithmic and processor developments, and to add or remove primitives as and when appropriate (cf. [36]). Almost all our test software used has been made available on a web-site which will be updated, as years pass by, with the latest costs:

`http://www.cs.bris.ac.uk/~nigel/Cloud-Keys/`

EC2 versus Total Cost of Ownership (TCO). The approach set forth above associates a monetary cost to key strength, but does so at negligible actual cost; this is useful for many purposes. However, no key recovery nor collision is completed. The question remains, if one desires to complete a computation, whether doing so on EC2 is less expensive than acquiring a similar platform and operating it oneself.

TCO includes many costs that are hard to estimate. Nevertheless, the following may be useful. At moderate volume, a dual node server with two processors, each with twelve 1.9 GHz cores, and 32 GB of memory per node can be purchased for approximately $ 8000. This implies that at that fixed cost approximately $2 \cdot 2 \cdot 12 \cdot 1.9 = 91.2$ ECUs with $1\frac{1}{3}$ GB of memory per core can be purchased. At $\$ \frac{20}{91.2} \cdot 8000 \approx 1750$ per 20 ECUs this compares favourably to the fixed triennial payment $\tau = 2800$ for the 20 ECUs of EC2-instance high-CPU EL. Power consumption of the above server is estimated to be bounded by 600 Watts. Doubling this to account for cooling and so on, we arrive at approximately 265 Watts for 20 ECUs, thus about a quarter kWh. At a residential rate of $ 0.25 per kWh we find that running our own 20 ECUs for three years costs us $\$ 1750 + 3Y \frac{265}{1000} \cdot 0.25 \approx 3500$, as opposed to $\$ 2800 + 3Y \cdot 0.24 \approx 9100$ for EC2's high-CPU EL.

Although in low-volume such a server could be supported without additional infrastructure, in high-volume they require a data center and maintenance personnel; this also implies lower electricity rates and quantum discount for acquisition however. Given expected cost and lifespan of infrastructures and salary costs, it is not unreasonable to estimate that TCO is at most one third of the cost of EC2. We conclude that full computations are still best conducted on one's own equipment.

3 Results

In this section we detail the application of our approach to five different cryptographic primitives: the block ciphers DES and AES, the cryptographic hash function SHA-2, and the public-key cryptosystems RSA and ECC. The first is chosen for historic reasons, whilst the others are the primitives of choice in many current applications.

For each of the five primitives, the fastest methods to recover the symmetric-key (DES and AES), to find a collision (SHA-2), or to derive the private-key (RSA and ECC) proceed very similarly, though realised using entirely different algorithms. In each algorithm, a huge number of identical computations are performed on different data; they can be carried out simultaneously on almost any number (and type) of client cores. With one exception (the NFS matrix step, as alluded to above), each client operates independently of all others, as long as there are central servers tasked with distributing inputs to clients and collecting their outputs. Furthermore, for each client the speed at which inputs are processed (and outputs produced, if relevant) is constant over time: a relatively short calculation per type of client along with a sufficiently accurate estimate of the number of inputs to be processed (or outputs to be produced, if relevant) suffices to be able to give a good indication of the total computational effort required.

The client-server approach has been common in a cryptographic context since the late 1980s, originally implemented using a variety of relatively crude application-specific, pre-cloud approaches [9, 22] (that continue to the present day [5, 18]), and later based on more general web-based services such as [8] that support collaborative compute projects (such as [27]). Thus, for each primitive under consideration, the problem of managing the servers is well understood. Additionally, the computational effort expended by said servers is dwarfed by the total computation required of the clients. As a result, in this section we concentrate on a series of experiments using the client software only: for each primitive, we execute the associated client software for a short yet representative period of time on the most appropriate on-demand EC2 instance, use the results to extrapolate the total key retrieval cost using the corresponding reserved EC2 instance, and relate the result to a discussion of prior work.

We implemented a software component to perform each partial computation on EC2, focusing on the use of platform agnostic (i.e., processor non-specific) programming techniques via ANSI-C. In particular, we used processor-specific operations (e.g., to cope with carries efficiently) only in situations where an alternative in C was at least possible; in such cases we abstracted the operations into easily replaceable macros or used more portable approaches such as compiler intrinsics where possible. As motivated above, the goal of this was portability and hence repeatability; clearly one could produce incremental improvements by harnessing processor-specific knowledge (e.g., of instruction scheduling or register allocation), but equally clearly these are likely to produce improvement by only a (small) constant factor and may defeat said goal.

3.1 DES

We first examine DES (which is considered broken with current technology) to provide a base line EC2 cost against which the cost of other key retrieval efforts can be measured.

Prior work. As one of the oldest public cryptographic primitives, DES has had a considerable amount of literature devoted to its cryptanalysis over the years. Despite early misgivings and suspicions about its design and the development of several new cryptanalytic techniques, the most efficient way to recover a single DES key is still exhaustive search. This approach was most famously realised by *Deep Crack* developed by the EFF [14]. Designed and built in 1998 at a cost of $ 200,000, this device could find a single DES key in 22 hours. Various designs, often based on FPGAs, have been presented since specifically for DES key search. The most famous of these is COPACOBANA [15], which at a cost of $ 10,000 can perform single DES key search in 6.4 days on average. One can also extrapolate from suitable high-throughput DES implementations. For example [33] presents an FPGA design, which on a Spartan FPGA could perform an exhaustive single DES key search in 9.5 years. Using this design, in [39][p. 19] it is concluded that a DES key search device which can find one key every month can be produced for a cost of $ 750. Alternatively, using a time-memory trade-off [32], one can do a one-time precomputation at cost comparable to exhaustive key search, after which individual keys can be found at much lower cost: according to [31] a DES key can be found in half an hour on a $ 12 FPGA, after a precomputation that takes a week on a $ 12,000 device.

DES key search. In software the most efficient way to implement DES key search is to use the bit-sliced implementation method of Biham [6]. In our experiments we used software developed by Matthew Kwan[6] for the RSA symmetric-key challenge eventually solved by the DESCHALL project in 1997. Our choice was motivated by the goal of using platform agnostic software implementations of the best known algorithms.

Given a message/ciphertext pair the program searches through all the keys trying to find the matching key. On average one expects to try 2^{55} keys (i.e., one half of the key space) until a match is found. However in the bit-slice implementation on a 64-bit machine one evaluates DES on the message for 64 keys in parallel. In addition there are techniques, developed by Rocke Verser for the DESCHALL project, which allow one to perform an early abort if one knows a given set of 64 keys will not result in the target ciphertext. For comparison using processor extensions, we implemented the same algorithm on 128 keys in parallel using the SSE instructions on the x86 architecture.

DES key search on EC2. Using EC2 we found that y, the expected number of years to complete a DES calculation on a single ECU, was $y = 97$ using a vanilla 64-bit C implementation, and $y = 51$ using the 128-bit SSE implementation. Combined with our prior formulae of $ 150y, $ 195y and $ 300y for high-CPU EL instances, we estimate the cost of using EC2 to recover a single DES key as follows:

Algorithm	ECU Years	Estimated Key Retrieval Cost		
		3 Years	1 Year	ASAP
DES (Vanilla C)	97	$14,550	$18,915	$29,100
DES (SSE Version)	51	$7,650	$9,945	$15,300

In comparing these to earlier figures for special-purpose hardware, one needs to bear in mind that once a special-purpose hardware device has been designed and built, the additional cost of finding subsequent keys

[6] Available from `http://www.darkside.com.au/bitslice/`.

after the first one is essentially negligible (bar the maintenance and power costs). Thus, unless time-memory trade-off key search is used, the cost-per-key of specialised hardware is lower than using EC2. We repeat that our thesis is that dedicated hardware gives a point estimate, whereas our experiments are repeatable. Thus as long as our costs are scaled by an appropriate factor to take into account the possibility of improving specialised hardware, our estimates (when repeated annually) can form a more robust method of determining the cost of finding a key.

We end with noting that whilst few will admit to using DES in any application, the use of three-key triple DES (or 3DES) is widespread, especially in the financial sector. Because the above cost underestimates the cost of 2^{55} full DES encryptions (due to the early abort technique), multiplying it by 2^{112} lower bounds the cost for a full three-key 3DES key search (where the factor of 3 incurred by the three distinct DES calls can be omitted by properly ordering the search).

3.2 AES

Prior work. Since its adoption around ten years ago, AES has become the symmetric cipher of choice in many new applications. It comes in three variants, AES-128, AES-192, and AES-256, with 128-, 192-, and 256-bit keys, respectively. The new cipher turned out to have some unexpected properties in relation to software side-channels [29], which in turn triggered the development of AES-specific instruction set extensions [16]. Its strongest variant, AES-256, was shown to have vulnerabilities not shared with the others [7]. These developments notwithstanding, the only known approach to recover an AES key is by exhaustive search. With an AES-128 key space of 2^{128} this is well out of reach and therefore it is not surprising that there seems little work on AES specific hardware to realise a key search. One can of course extrapolate from efficient designs for AES implementation. For example using the FPGA design in [40] which on a single Spartan FPGA can perform the above exhaustive key search in $4.6 \cdot 10^{23}$ years, the authors of [39] estimate that a device can be built for $\$\, 2.8 \cdot 10^{24}$ which will find an AES-128 key in one month.

AES key search on EC2. In software one can produce bit-slice versions of AES using the extended instruction sets available on many new processors [24]. However, in keeping with our principle of simple code, which can be run on multiple versions of today's computers as well as future computers, we decided to use a traditional AES implementation in our experiments. We found the estimated number of years y on a single ECU to finish a single AES computation was $y \approx 10^{24}$. Using high-CPU EL instances our costs become (rounding to the nearest order of magnitude),

Algorithm	ECU Years	Estimated Key Retrieval Cost 3 Years / 1 Year / ASAP
AES-128	10^{24}	$\approx \$10^{26}$

Again our comments for DES concerning the cost of specialised hardware versus our own estimates apply for the case of AES, although in the present case the estimates are more closely aligned. However, in the case of AES the new Westmere 32nm Intel core has special AES instructions [16]. It may be instructive to perform our analysis on the EC2 service, once such cores are available on this service[7]. Whilst a 3-to-10 fold performance improvement for AES encryption using Westmere has been reported; our own experiments on our local machines only show a two fold increase in performance for key search.

3.3 SHA-2

Prior work. The term SHA-2 denotes a family of four hash functions; SHA-224, SHA-256, SHA-384 and SHA-512. We shall be concentrating on SHA-256 and SHA-512; the SHA-224 algorithm only being introduced to make an algorithm compatible with 112-bit block ciphers and SHA-384 being just a truncated version of SHA-512. The three variants SHA-256, SHA-384 and SHA-512 were standardised by NIST in 2001, with

[7] As of Feb 2011 none of the 64-bit instances we ran on the EC2 service had the Westmere 32nm Intel core on them.

SHA-224 being added in 2004, as part of FIPS PUB 180-2 [25]. The SHA-2 family of algorithms is of the same algorithmic lineage as MD4, MD5 and SHA-1.

Cryptographic hash functions need to satisfy a number of security properties; for example preimage-resistance, collision resistance, etc. The property which appears easiest to violate for earlier designs, and which generically is the least costly to circumvent, is that of collision resistance. Despite the work on cryptanalysis of the related hash functions MD4, MD5 and SHA-1 [42–45, 41], the best known methods to find collisions for the SHA-2 family still are the generic ones. Being of the Merkle-Damgård family each of the SHA-2 algorithms consists of a compression function, which maps b-bit inputs to h-bit outputs, and a chaining method. The chaining method is needed to allow the hashing of messages of more than b bits in length. The input block size $b = 512$ for SHA-256 but $b = 1024$ for SHA-384 and SHA-512. The output block size h is given by the name of the algorithm, i.e., SHA-h.

SHA-2 collision search. The best generic algorithm for collision search is the parallel "distinguished points" method of van Oorschot and Wiener [28]. In finding collisions this method can be tailored in various ways; for example one could try to obtain two meaningful messages which produce the same hash collision. In our implementation we settle for the simplest, and least costly, of all possible collision searches; namely to find a collision between two random messages.

The collision search proceeds as follows. Each client generates a random element $x_0 \in \{0,1\}^h$ and then computes the iterates $x_i = H(x_{i-1})$ where H is the hash function. When an iterate x_d meets a given condition, say the last 32-bits are zero, we call the iterate a distinguished point. The tuple (x_d, x_0, d) is returned to a central server and the client now generates a new value $x_0 \in \{0,1\}^h$ and repeats the process. Once the server finds two tuples (x_d, x_0, d) and $(y_{d'}, y_0, d')$ with $x_d = y_{d'}$ a collision in the hash function can be obtained by repeating the two walks from x_0 and y_0.

By the birthday paradox we will find a collision in roughly $\sqrt{\pi \cdot 2^{h-1}}$ applications of H, with n clients providing an n-fold speed up. If $1/p$ of the elements of $\{0,1\}^h$ are defined to be distinguished then the memory requirement of the server becomes $O(\sqrt{\pi \cdot 2^{h-1}}/p)$.

SHA-2 collision search on EC2. We implemented the client side of the above distinguished points algorithm for SHA-256 and SHA-512. On a single ECU we found that the expected number of years needed to obtain a collision was given by the "ECU Years" values in the following table, resulting in the associated collision finding costs, where again we round to the nearest order of magnitude and use high-CPU EL instances. Note, that SHA-256 collision search matches the cost of AES-128 key retrieval, as one hopes would happen for a well designed hash function of output twice the key size of a given well designed block cipher.

Algorithm	ECU Years	Estimated Collision Search Cost 3 Years / 1 Year / ASAP
SHA-256	10^{24}	$\approx \$10^{26}$
SHA-512	10^{63}	$\approx \$10^{63}$

3.4 RSA

NFS background. As the oldest public key algorithm, RSA (and hence integer factorisation) has had a considerable amount of research applied to it over the years. The current best published algorithm for factoring integers is Coppersmith's variant [12] of the Number Field Sieve method (NFS) [21]. Based on loose heuristic arguments and asymptotically for $n \to \infty$, its expected run time to factor n is

$$L(n) = \exp((1.902 + o(1))(\log n)^{1/3}(\log\log n)^{2/3}),$$

where the logarithms are natural. For the basic version, i.e., not including Coppersmith's modifications, the constant "1.902" is replaced by "1.923". To better understand and appreciate our approach to get EC2-cost estimates for NFS, we need to know the main steps of NFS. We restrict ourselves to the basic version.

Polynomial selection. Depending on the RSA modulus n to be factored, select polynomials that determine the number fields to be used.

Sieving step. Find elements of the number fields that can be used to derive equations modulo n. Each equation corresponds to a sparse k-dimensional zero-one vector, for some $k \approx \sqrt{L(n)}$, such that each subset of vectors that sums to an all-even vector gives a 50% chance to factor n. Continue until at least $k + t$ equations have been found for a small constant $t > 0$ (hundreds, at most).

Matrix step. Find at least t independent subsets as above.

Square root. Try to factor n by processing the subsets (with probability of success $\geq 1 - (\frac{1}{2})^t$).

Because $L(n)$ number field elements in the sieving step have to be considered so as to find $k + t \approx \sqrt{L(n)}$ equations, the run time is attained by the sieving and matrix steps with memory requirements of both steps, and central storage requirements, behaving as $\sqrt{L(n)}$. The first step requires as little or as much time as one desires – see below. The run time of the final step behaves as $\sqrt{L(n)}$ with small memory needs.

The set of number field elements to be sieved can be parcelled out among any number of independent processors. Though each would require the same amount $\sqrt{L(n)}$ of memory, this sieving memory can be optimally shared by any number of threads; smaller memories can be catered for as well at small efficiency loss. Although all clients combined report a substantial amount of data to the server(s), the volume per client is low. The resulting data transfer expenses are thus not taken into account in our analysis below. The matrix step can be split up in a small number (dozens, at most) of simultaneous and independent calculations. Each of those demands fast inter-processor communication and quite a bit more memory than the sieving step (though the amounts are the same when expressed in terms of the above L-function).

It turns out that more sieving (which is easy, as sieving is done on independent processors) leads to a smaller k (which is advantageous, as it makes the matrix step less cumbersome). It has been repeatedly observed that this effect diminishes, but the trade-off has not been analysed yet.

Unlike DES, AES, or ECC key retrieval or SHA-2 collision finding methods, NFS is a multi-stage method which makes it difficult to estimate its run time. As mentioned, the trade-off between sieving and matrix efforts is as yet unclear and compounded by the different platforms (with different EC2 costs) required for the two calculations. The overall effort is also heavily influenced by the properties of the set of polynomials that one manages to find in the first step. For so-called *special* composites finding the best polynomials is easy: in this case the special number field sieve applies (and the "1.902" or "1.923" above is replaced by "1.526"). For generic composites such as RSA moduli, the situation is not so clear. The polynomials can trivially be selected so that the (heuristic) theoretical NFS run time estimate is met. As it is fairly well understood how to predict a good upper bound for the sieving and matrix efforts given a set of polynomials, the overall NFS-effort can easily be upper bounded. In practice, however, this upper bound is too pessimistic and easily off by an order of magnitude. It turns out that one can quickly recognise if one set of polynomials is "better" than some other set, which makes it possible (combined with smart searching strategies that have been developed) to efficiently conduct a search for a "good" set of polynomials. This invariably leads to substantial savings in the subsequent steps, but it is not yet well understood how much effort needs to be invested in the search to achieve lowest overall run time.

The upshot is that one cannot expect that for any relevant n a shortened polynomial selection step will result in polynomials with properties representative for those one would find after a more extensive search. For the purposes of the present paper we address this issue by simply skipping polynomial selection, and by restricting the experiments reported below to a fixed set of moduli for which good or reasonable polynomials are known – and to offer others the possibility to improve on those choices. The fixed set that we consider consists of the k-bit RSA moduli RSA-k for $k = 512, 768, 896, 1024, 2048$ as originally published on the now obsolete RSA Challenge list [34]. That we consider only these moduli does not affect general applicability of our cost estimates, if we make two assumptions: we assume that for RSA moduli of similar size NFS requires a similar effort, and that cost estimates for modulus sizes between 512 and 2560 bits other than those above follow by judicious application of the L-function. Examples are given below. We find it hazardous to attach significance to the results of extrapolation beyond 2560.

Prior work. Various special purpose hardware designs have been proposed for factoring most notably TWINKLE [37], TWIRL [38] and SHARK [13]. SHARK is speculated to do the NFS sieving step for RSA-

1024 in one year at a total cost of one billion dollars. With the same run time estimate but only ten million dollars to build and twenty million to develop, plus the same costs for the matrix step, TWIRL would be more than two orders of magnitude less expensive. Not everyone agrees, however that TWIRL can be built and will perform as proposed.

In 1999 NFS was used to factor RSA-512 [10] using software running on commodity hardware. Using much improved software and a better set of polynomials the total effort required for this factorisation would now be about 3 months on a single 2.2GHz Opteron core. In 2009, this same software version of NFS (again running on regular servers) was used to factor RSA-768 [17]. The total effort of this last factorisation was less than 1700 years on a single 2.2GHz Opteron core with 2 GB of memory: about 40 years for polynomial selection, 1500 years for sieving, 155 years for the matrix, and on the order of hours for the final square root step. The matrix step was done on eight disjoint clusters, with its computationally least intensive but most memory demanding central stage done on a single cluster and requiring up to a TB of memory for a relatively brief period of time. According to [17], however, half the amount of sieving would have sufficed. Combined with the rough estimate that this would have doubled the matrix effort and based on our first assumption above, 1100 years on a 2.2GHz core will thus be our estimate for the factoring effort of *any* 768-bit RSA modulus. Note that the ratio $\frac{1100}{0.25} = 4400$ of the efforts for RSA-768 and RSA-512 is of the same order of magnitude as $\frac{L(2^{768})}{L(2^{512})} \approx 6150$ (twice omitting the "$o(1)$"), thus not providing strong evidence against our second assumption above.

Factoring on EC2. Based on the sieving and matrix programs used in [17] we developed two simplified pieces of software that perform the most relevant sieving and matrix calculations without outputting any other results than the time required for the calculations. Sieving parameters (such as polynomials defining the number fields, as described above) are provided for the fixed set of moduli RSA-512, RSA-768, RSA-896, RSA-1024 and RSA-2048. For the smallest two numbers they are identical to the parameters used to derive the timings reported above, for both steps resulting in realistic experiments with threading and memory requirements that can be met by EC2. For RSA-896 our parameters are expected to be reasonable, but can be improved, and RSA-896 is small enough to allow meaningful EC2 sieving experiments. For the largest two numbers our parameter choices allow considerable improvement (at possibly substantial computational effort spent on polynomial selection), but we do not expect that it is possible to find parameters for RSA-1024 or RSA-2048 that allow realistic sieving experiments on the current EC2: for RSA-1024 the best we may hope for would be a sieving experiment requiring several hours using on the order of 100 GB of memory, increasing to several years using petabytes of memory for RSA-2048.

The simplified matrix program uses parameters (such as the k value and the average number of non-zero entries per vector) corresponding to those for the sieving. It produces timing and cost estimates only for the first and third stage of the three stages of the matrix step. The central stage is omitted. For RSA-512 and RSA-768 this results in realistic experiments that can be executed using an EC2 cluster instance (possibly with the exception of storage of the full RSA-768 matrix). For the other three moduli the estimated sizes are beyond the capacity of EC2. It is even the case that at this point it is unclear to us how the central stage of the RSA-2048 matrix step should be performed at all: with the approach used for RSA-768 the cost of the central stage would by far dominate the overall cost, whereas for the other three moduli the central stage is known or expected to be negligible compared to the other two.

The table below lists the most suitable ECU instance for each program and the five moduli under consideration so far, and the resulting timings and EC2 cost estimates for RSA-512, RSA-768, and RSA-896. The figures in italics (RSA-896 matrix step, both steps for RSA-1024 and RSA-2048) are just crude L-based extrapolations[8].

The rough cost estimate for factoring a 1024-bit RSA modulus in one year is of the same order of magnitude as the SHARK cost, without incurring the SHARK development cost and while including the cost of the matrix step.

[8] We note the huge discrepancy between EC2-factoring cost and the monetary awards that used to be offered for the factorizations of these moduli [34].

	Sieving Step					Matrix Step				
Modulus	Instance	ECU Years	Estimated Sieving Cost 3 Years	1 Year	ASAP	Instance	ECU Years	Estimated Matrix Cost 3 Years	1 Year	ASAP
RSA-512	high-CPU	0.36	N/A	N/A	$108	high-CPU EL	0.1	N/A	N/A	$30
RSA-768	EL	1650	$250,000	$320,000	$500,000		680	$145,000	$185,000	$285,000
RSA-896	high-mem	$1.5 \cdot 10^5$	$\$5 \cdot 10^7$	$\$7 \cdot 10^7$	$\$10^8$	cluster	$3 \cdot 10^4$	$\$6 \cdot 10^6$	$\$8 \cdot 10^6$	$\$1.3 \cdot 10^7$
RSA-1024	EL	$2 \cdot 10^6$	$\$7 \cdot 10^8$	$\$9 \cdot 10^8$	$\$1.3 \cdot 10^9$	compute	$8 \cdot 10^5$	$\$1.7 \cdot 10^8$	$\$2 \cdot 10^8$	$\$3.4 \cdot 10^8$
RSA-2048		$2 \cdot 10^{15}$		$\$5 \cdot 10^{17}$			$8 \cdot 10^{14}$		$\$2 \cdot 10^{17}$	

3.5 ECC

Prior Work. The security of ECC (Elliptic Curve Cryptography) relies on the hardness of the Elliptic Curve Discrete Logarithm Problem (EC-DLP). In 1997 Certicom issued a series of challenges of different security levels [11]. Each security level is defined by the number of bits in the group order of the elliptic curve. The Certicom challenges were over binary fields and large prime fields, and ranged from 79-bit to 359-bit curves. The curves are named with the following convention. ECCp-n refers to a curve over a large prime field with a group order of n bits, ECC2-n refers to a similar curve over a binary field, and ECC2K-n refers to a curve with additional structure (a so-called Koblitz curve) with group order of n bits over a binary field.

The smaller "exercise" challenges were solved quickly: in December 1997 and February 1998 ECCp-79 and ECCp-89 were solved using 52 and 716 machine days on a set of 500 MHz DEC Alpha workstations, followed in September 1999 by ECCp-97 in an estimated 6412 machine days on various platforms from different contributors. The first of the main challenges were solved in November 2002 (ECCp-109) and in April 2004 (ECC2-109). Since then no challenges have been solved, despite existing efforts to do so.

ECC key search. The method for solving EC-DLP is based on Pollard's rho method [30]. Similar to SHA-2 collision search, it searches for a collision between two distinguished points. We first define a deterministic "random" walk on the group elements; each client starts such a walk, and then when they reach a distinguished point, the group element and some additional information is sent back to a central server. Once the servers received two identical distinguished points one can solve the EC-DLP using the additional information. We refer to [28] for details.

ECC key search on EC2. Because most deployed ECC systems, including the recommended NIST curves, are over prime fields, we focus on elliptic curves defined over a field of prime order p. We took two sample sets: the Certicom challenges [11] which are defined over fields where p is a random prime, and the curves over prime fields defined in the NIST/SECG standards [26, 35], where p is a so-called generalised Mersenne prime, thereby allowing more efficient field arithmetic. Of the latter we took the random curves over fields of cryptographically interesting sizes listed in the table below.

All of the curves were analysed with a program which used Montgomery arithmetic for its base field arithmetic. The NIST/SECG curves were also analysed using a program which used specialised arithmetic, saving essentially a factor of two. The following table summarises the costs of key retrieval using high-CPU EL EC2 instances in February 2011, rounded to the nearest order of magnitude for the larger p. We present the costs for the small curves for comparison with the effort spent in the initial analysis over a decade ago.

Curve Name (Certicom)	ECU Years	Estimated Key Retrieval Cost 3 Years	1 Year	ASAP
ECCp-79	3.5 days	N/A	N/A	$2.85
ECCp-89	104 days	N/A	N/A	$85
ECCp-97	5	$750	$975	$1,500
ECCp-109	300	$45,000	$58,500	$90,000
ECCp-131	10^6		$\approx \$10^8$	
ECCp-163	10^{10}		$\approx \$10^{12}$	
ECCp-191	10^{15}		$\approx \$10^{17}$	
ECCp-239	10^{22}		$\approx \$10^{24}$	
ECCp-359	10^{40}		$\approx \$10^{42}$	

Curve Name (NIST/SECG)	ECU Years	Estimated Key Retrieval Cost 3 Years / 1 Year / ASAP
secp192-r1	10^{15}	$\approx \$10^{17}$
secp224-r1	10^{20}	$\approx \$10^{22}$
secp256-r1	10^{25}	$\approx \$10^{27}$
secp384-r1	10^{44}	$\approx \$10^{46}$

(Note that general orders of magnitude correlate with what we expect in terms of costs related to AES, SHA-2, etc.)

4 Extrapolate

Comparing the current pricing model to EC2's 2008 flat rate of $ 0.10 per hour per ECU, we find that prices have dropped by a factor of three (short term) to six (triennial). This shaves off about two bits of the security of block ciphers over a period of about three years, following closely what one would expect based on Moore's law. The most interesting contribution of this paper is that our approach allows anyone to measure and observe at what rate key erosion continues in the future. A trend that may appear after doing so for a number of years could lead to a variety of useful insights – not just concerning cryptographic key size selection but also with respect to the sanity of cloud computing pricing models.

In future versions of this paper these issues will be elaborated upon in this section. Right now the required data are, unavoidably, still lacking.

5 Can one do better?

If by better one means can one reduce the overall costs of breaking each cipher or key size, then the answer is yes. This is for a number of reasons: Firstly one could find a different utility computing service which is cheaper; however we selected Amazon EC2 so as to be able to repeat the experiment each year on roughly the same platform. Any price differences which Amazon introduce due to falling commodity prices, or increased power prices, are then automatically fed into our estimates on a year basis. Since it is unlikely that Amazon will cease to exist in the near future we can with confidence assume that the EC2 service will exist in a year's time.

Secondly, we could improve our code by fine tuning the algorithms and adopting more efficient implementation techniques. We have deliberately tried not to do this. We want the code to be executed next year, and the year after, on the platforms which EC2 provides, therefore highly specialised performance improvements have not been considered. General optimisation of the algorithm can always be performed and to enable this have made the source code available on a public web site, `http://www.cs.bris.ac.uk/~nigel/Cloud-Keys/`. However, we have ruled out aggressive optimisations as they would only provide a constant improvement in performance and if costs to break a key are of the order of 10^{20} dollars then reducing this to 10^{18} dollars is unlikely to be that significant in the real world.

Finally, improvements can come from algorithmic breakthroughs. Although for all the algorithms we have discussed algorithmic breakthroughs have been somewhat lacking in the last few years, we intend to incorporate them in our code if they occur.

Acknowledgements. The work of the first two authors was supported by the Swiss National Science Foundation under grant numbers 200021-119776, 206021-128727, and 200020-132160. The work in this paper was partially funded by the eCrypt-2 Network of Excellence funded by the European Union. The fourth author's research was also partially supported by a Royal Society Wolfson Merit Award.

References

1. Amazon Elastic Compute Cloud *Limited Beta*, July 2007, http://web.archive.org/web/20070705164650rn_2/www.amazon.com/b?ie=UTF8&node=201590011
2. Amazon Elastic Compute Cloud *Beta*, May 2008, http://web.archive.org/web/20080501182549rn_2/www.amazon.com/EC2-AWS-Service-Pricing/b?ie=UTF8&node=201590011.
3. Amazon Elastic Compute Cloud (Amazon EC2), http://aws.amazon.com/ec2/.
4. F. Bahr, M. Boehm, J. Franke and T. Kleinjung, *Subject: RSA200*. Announcement, 9 May 2005.
5. D.V. Bailey, L. Batina, D.J. Bernstein, P. Birkner, J.W. Bos, H.-C. Chen, C.-M. Cheng, G. van Damme, G. de Meulenaer, L.J.D. Perez, J. Fan, T. Güneysu, F. Gurkaynak, T. Kleinjung, T. Lange, N. Mentens, R. Niederhagen, C. Paar, F. Regazzoni, P. Schwabe, L. Uhsadel, A. Van Herrewege and B.-Y. Yang, *Breaking ECC2K-130*. Cryptology ePrint Archive, Report 2009/541, http://eprint.iacr.org/2009/541, 2009.
6. E. Biham. A fast new DES implementation in software. *Fast Software Encryption – FSE 1997*, Springer-Verlag, LNCS 1297, 260–272, 1997.
7. A. Biryukov, D. Khovratovich and I. Nikolić. Distinguisher and related-key attack on the full AES-256. *Advances in Cryptology – Crypto 2009*, Springer-Verlag, LNCS 5677, 231–249, 2009.
8. The BOINC project, http://boinc.berkeley.edu/.
9. T.R. Caron and R.D. Silverman. Parallel implementation of the quadratic sieve. *J. Supercomputing*, **1**, 273–290, 1988.
10. S. Cavallar, B. Dodson, A. K. Lenstra, P. Leyland, P. L. Montgomery, B. Murphy, H. te Riele, P. Zimmermann, et al. Factoring a 512-bit RSA modulus. *Advances in Cryptology – Eurocrypt 2000*, Springer-Verlag, LNCS 1807, 1–18, 2000.
11. Certicom Inc. *The Certicom ECC Challenge*. http://www.certicom.com/index.php/the-certicom-ecc-challenge.
12. D. Coppersmith. Modifications to the number field sieve. *J. of Cryptology*, **6**, 169–180, 1993.
13. J. Franke, T. Kleinjung, C. Paar, J. Pelzl, C. Pripalta and C. Stahlke. SHARK: A realizable special hardware sieving device for factoring 1024-bit integers. *Cryptographic Hardware and Embedded Systems – CHES 2005*, Springer LNCS 3659, 119–130, 2005.
14. J. Gilmore (Ed.). *Cracking DES: Secrets of Encryption Research, Wiretap Politics & Chip Design*. Electronic Frontier Foundation, O'Reilly & Associates, 1998.
15. T. Güneysu, T. Kasper, M. Novotný, C. Paar, and A. Rupp. Cryptanalysis with COPACOBANA. *IEEE Transactions on Computers*, **57**, 1498–1513, 2008.
16. S. Gueron. Intel's new AES instructions for enhanced performance and security. *Fast Software Encryption – FSE 2009* Springer LNCS 5665, 51–66, 2009.
17. I. Kleinjung, K. Aoki, J. Franke, A.K. Lenstra, E. Thomé, J.W. Bos, P. Gaudry, A. Kruppa, P.L. Montgomery, D.A. Osvik, H. te Riele, A. Timofeev and P. Zimmermann. Factorization of a 768-bit RSA modulus. *Advances in Cryptology – Crypto 2010*, Springer LNCS 6223, 333–350, 2010.
18. I. Kleinjung, J.W. Bos, A.K. Lenstra, D.A. Osvik, K. Aoki, S. Contini, J. Franke, E. Thomé, P. Jermini, M. Thiémard, P. Leyland, P.L. Montgomery, A. Timofeev and H. Stockinger. A heterogeneous computing environment to solve the 768-bit RSA challenge. *J. Cluster Computing*. To appear.
19. A.K. Lenstra. Unbelievable security; matching AES security using public key systems. *Advances in Cryptology – Asiacrypt 2001*, Springer-Verlag LNCS 2248, 67–86, 2001.
20. A.K. Lenstra. Key Lengths. Chapter 114 of *The Handbook of Information Security*, Wiley 2005.
21. A.K. Lenstra and H.W. Lenstra, Jr. (eds.). *The development of the number field sieve*. Lecture Notes in Math. 1554, Springer-Verlag, 1993.
22. A.K. Lenstra and M.S. Manasse. Factoring by electronic mail. *Advances in Cryptology – Eurocrypt'89*, Springer-Verlag LNCS 434, 355–371, 1989.
23. A.K. Lenstra and E.R. Verheul, Selecting Cryptographic Key Sizes. *J. of Cryptology*, **14**, 255–293, 2001.
24. M. Matsui and J. Nakakima. On the power of bitslice implementation on Intel Core2 processor. *Cryptography Hardware and Embedded Systems – CHES 2007*, Springer-Verlag LNCS 4727, 121–134, 2007.
25. NIST. *Secure Hash Signature Standard (SHS) – FIPS PUB 180-2*. Available at http://csrc.nist.gov/publications/fips/fips180-2/fips180-2.pdf
26. NIST. *Digital Signature Standard (DSS) – FIPS PUB 186-2*. Available at http://csrc.nist.gov/publications/fips/fips186-2/fips186-2-change1.pdf
27. NFS@home,http://escatter11.fullerton.edu/nfs.
28. P.C. van Oorschot and M.J. Wiener. Parallel collision search with cryptanalytic applications. *J. of Cryptology*, **12**, 1–28, 1999.

29. D.A. Osvik, A. Shamir and E. Tromer. Efficient Cache Attacks on AES, and Countermeasures. *J. of Cryptology*, **23**, 37–71, 2010.
30. J. Pollard. Monte Carlo methods for index computation mod p, *Math. Comp.*, **32**, 918–924, 1978.
31. J.-J Quisquater and F. Standaert. Exhaustive key search of the DES: Updates and refinements. SHARCS 2005 (2005).
32. J.-J Quisquater and F. Standaert. Time-memory tradeoffs. *Encyclopedia of Cryptography and Security*, Springer-Verlag, 614–616, 2005.
33. G. Rouvroy, F.-X. Standaert, J.-J. Quisquarter and J.-D. Legat. Design strategies and modified descriptions to optimize cipher FPGA implementations: Fact and compact results for DES and Triple-DES. *ACM/SIGDA - Symposium on FPGAs*, 247–247, 2003.
34. The RSA challenge numbers, formerly on `http://www.rsa.com/rsalabs/node.asp?id=2093`, now on for instance `http://en.wikipedia.org/wiki/RSA_numbers`.
35. SECG. *Standards for Efficient Cryptography Group*. SEC2: Recommended Elliptic Curve Domain Parameters version 1.0, `http://www.secg.org`.
36. `http://csrc.nist.gov/groups/ST/hash/sha-3/`.
37. A. Shamir. Factoring large numbers with the TWINKLE device. Manuscript, 2000.
38. A. Shamir and E. Tromer. Factoring large numbers with the TWIRL device. *Advances in Cryptology – Crypto 2003*, Springer-Verlag LNCS 2729, 1–26, 2003.
39. N.P. Smart (Ed.). *ECRYPT II: Yearly report on algorithms and keysizes (2009-2010)*. Available from `http://www.ecrypt.eu.org/documents/D.SPA.13.pdf`.
40. F.-X. Standaert, G. Rouvroy, J.-J. Quisquarter and J.-D. Legat. Efficient implementation of Rijndael encryption in reconfigurable hardware: Improvements and design tradeoffs. *Cryptography Hardware and Embedded Systems – CHES 2003*, Springer-Verlag LNCS 2779, 334–350, 2003.
41. M. Stevens, A. Sotirov, J. Appelbaum, A. Lenstra, D. Molnar, D.A. Osvik and B. de Weger. Short chosen-prefix collisions for MD5 and the creation of a rogue CA certificate. *Advances in Cryptology – Crypto 2009*, Springer-Verlag LNCS 5677, 55–69, 2009.
42. X. Wang, D. Feng, X. Lai and H. Yu. *Collisions for Hash Functions MD4, MD5, HAVAL-128 and RIPEMD*. Cryptology ePrint Archive, Report 2004/199, `http://eprint.iacr.org/2004/199`, 2009.
43. X. Wang, A. Yao and F. Yao, *New Collision Search for SHA-1*. Crypto 2005 Rump session, `www.iacr.org/conferences/crypto2005/r/2.pdf`.
44. X. Wang, Y.L. Yin and H. Yu. Finding Collisions in the Full SHA-1. *Advances in Cryptology – Crypto 2005*, Springer-Verlag LNCS 3621, 17–36, 2005.
45. X. Wang and H. Yu. How to Break MD5 and Other Hash Functions. *Advances in Cryptology – Eurocrypt 2005*, Springer-Verlag LNCS 3494, 19–35, 2005.