Understanding brute force

Daniel J. Bernstein *

Department of Mathematics, Statistics, and Computer Science (M/C 249)
The University of Illinois at Chicago
Chicago, IL 60607–7045
djb@cr.yp.to

1 Introduction

There is a widespread myth that parallelizing a computation cannot improve its price-performance ratio.

The reality is that a parallel computer is often several orders of magnitude faster than a comparably priced serial computer. Consider multiplying two n-bit numbers, for example, or sorting n elements of $\{1, 2, \ldots, n^2\}$. A properly designed 2-dimensional parallel computer of size $n^{1+o(1)}$ can do both jobs in time $n^{1/2+o(1)}$. A serial computer for either problem is much, much, much slower and can't be much smaller: it needs $n^{1+o(1)}$ serial accesses to $n^{1+o(1)}$ bits of memory.

A related myth is that analyzing the time of a computation on a huge serial computer is equivalent to analyzing the price and performance of parallel versions of the same computation. The reality is that parallelization has different effects on different algorithms. When computations that take serial time $n^{1+o(1)}$ are put on a parallel computer of size $n^{1+o(1)}$, some of them end up taking time $n^{o(1)}$; some of them end up taking time $n^{1+o(1)}$; some of them end up taking time $n^{1/2+o(1)}$; etc.

These myths cause three problems in cryptography:

- Cryptographers often wildly overestimate the real-world security of their cryptographic systems—specifically, the cost of carrying out the best attack known—because they are restricting attention to serial attacks.
- Cryptographers often assert that a system has been "broken" by a marginally improved serial attack—even though the serial attack is slower and more expensive than a standard parallel attack.
- Cryptographers often make incorrect choices among systems: they switch to a system that is stronger against serial attacks but is weaker against the best attacks—i.e., against parallel attacks.

---

* The author was supported by the National Science Foundation under grant CCR–9983950, and by the Alfred P. Sloan Foundation. Date of this document: 2005.04.25. Permanent ID of this document: 73e92f5b71793b498288efe81fe55dee. This is a preliminary version meant to announce ideas; it will be replaced by a final version meant to record the ideas for posterity. There may be big changes before the final version. Future readers should not be forced to look at preliminary versions, unless they want to check historical credits; if you cite a preliminary version, please repeat all ideas that you are using from it, so that the reader can skip it.

I first encountered these errors in the context of integer factorization. This paper discusses the same errors in the simpler context of brute-force key search.

Sections 2 and 5 of this paper describe two parallel brute-force key-search machines:

- The "standard parallel machine" in Section 2 is a straightforward parallel implementation of a well-known brute-force algorithm, specifically Oechslin's "rainbow-tables" algorithm in [5].
- The "variant parallel machine" in Section 5 is a straightforward parallel implementation of another well-known brute-force algorithm, specifically Rivest's "distinguished-points" algorithm.

Wiener in [6, Section 6] analyzed the amazing speed of a distinguished-points computation on a 3-dimensional parallel machine. Unfortunately, the capabilities of these machines are still far less widely appreciated than they should be. Section 3 discusses some vastly inferior serial machines in the recent literature.

Section 4 discusses the question of how we can protect against the parallel brute-force key-search machines. This question leads to several additional areas of confusion in the literature.

2 The standard parallel brute-force key-search machine

This section describes the standard parallel brute-force key-search machine. A competent attacker uses the standard parallel machine—and *not* the inferior serial machines described in Section 3—when he cannot find any cipher-specific weaknesses.

The problem

The attacker is trying to find a 16-byte AES key k, given the 16 bytes $H(k) = \mathrm{AES}_k(8675309)$. There's nothing special about the number 8675309, or about AES: this is a brute-force attack that applies to a huge variety of ciphers.

The attacker is actually trying to simultaneously solve the same problem for many independent keys $k_1, k_2, \ldots$. He's given $H(k_1), H(k_2), \ldots$; he'd like to find $k_1, k_2, \ldots$. Let's say he's facing 2^{10} keys overall.

The standard key-search circuit

The attacker builds a very small key-search circuit. The key-search circuit has three inputs: a 12-byte string β, a 4-byte integer n, and a 16-byte string s. The key-search circuit has one output: a 16-byte string $Z(\beta, n, s)$ defined recursively by $Z(\beta, 0, s) = s$ and $Z(\beta, n+1, s) = Z(\beta, n, H(s \oplus (\beta, n)))$.

The key-search circuit is slightly larger than an AES circuit. It takes slightly more time than n AES computations.

This key-search circuit is attached to a comparable-size memory circuit that buffers some inputs and outputs. Let's say the memory circuit is big enough to hold 2^4 inputs and 2^4 corresponding outputs.

The standard key-search machine

The attacker now builds a machine with 2^{32} key-search circuits in a $2^{16} \times 2^{16}$ mesh. Each key-search circuit has its own comparable-size memory circuit. Each key-search circuit is also connected to its immediate neighbors (north, south, east, west) in the mesh. As we'll see later, this network doesn't have to be terribly fast.

Note that quite a few key-search circuits will fit onto a single low-cost chip. This machine is expensive but clearly could be built: it has about 2^{42} bytes of memory, comparable to a 64×64 array of PCs.

The attacker feeds 2^{36} inputs to his 2^{32} key-search circuits; recall that each circuit can buffer 2^4 inputs. Specifically, the attacker selects a 12-byte string β and $2^{36} - 2^{33}$ random 16-byte strings $r_1, r_2, \ldots$; generates one input $(\beta, 2^{23}, r_j)$ for each random r_j; and generates 2^{23} inputs for each target $H(k_i)$, namely $(\beta, 0, H(k_i))$, $(\beta, 1, H(k_i))$, $(\beta, 2, H(k_i))$, $\ldots$, $(\beta, 2^{23} - 1, H(k_i))$.

All 2^{32} key-search circuits now run in parallel, producing $Z(\beta, 2^{23}, r_j)$ for each random r_j and $Z(\beta, 0, H(k_i)), \ldots, Z(\beta, 2^{23}-1, H(k_i))$ for each target $H(k_i)$. This takes slightly more time than $2^{23} \cdot 2^4 = 2^{27}$ AES computations.

The attacker then applies (for example) Schimmler's sorting algorithm to sort the 2^{36} Z values. This algorithm takes just 2^{21} adjacent compare-exchange steps—not a bottleneck compared to 2^{27} AES computations.

If the sorting encounters a collision between two $Z(\beta, 2^{23}, r_j)$'s, it throws one of those r_j's away. If the sorting encounters a collision between $Z(\beta, n, H(k_i))$ and $Z(\beta, 2^{23}, r_j)$, the machine pauses to redo $2^{23} - n$ iterations of the computation of $Z(\beta, 2^{23}, r_j)$. If the intermediate value $(\beta, n+1, s)$ satisfies $H(s \oplus (\beta, n)) = H(k_i)$ then the machine prints $s \oplus (\beta, n)$ as a guess for k_i.

Heuristic analysis

What's the chance that a particular key, say k_1, is found by this machine?

Here is a crude estimate. There are 2^{36} values $Z(\beta, 2^{23}, r_j)$, each of which involved 2^{23} intermediate values $(\beta, n+1, s)$ and 2^{23} inputs $s \oplus (\beta, n)$ to H, for a total of 2^{59} inputs to H. If any of those pseudorandom inputs bumped into k_1 then the machine will find k_1. Specifically, if an intermediate value $(\beta, n+1, s)$ in the computation of $Z(\beta, 2^{23}, r_j)$ satisfies $s \oplus (\beta, n) = k_1$, then $Z(\beta, 2^{23}, r_j) = Z(\beta, n, H(s \oplus (\beta, n))) = Z(\beta, n, H(k_1))$; the sorting will discover this collision, will locate the same intermediate value $(\beta, n+1, s)$, and will print $s \oplus (\beta, n) = k_1$ as its guess for k_1. This occurs with probability $2^{59}/2^{128} = 2^{-69}$.

This estimate is slightly overoptimistic, for three reasons: first, there are not quite as many r_j's; second, there might be collisions among the inputs to H; third, there might be a collision between two $Z(\beta, 2^{23}, r_j)$'s, eliminating the r_j relevant to k_1. But the first effect is small and the other effects are conjecturally negligible. The machine finds k_1 with probability conjecturally close to 2^{-69}.

Of course, the machine simultaneously has a chance of finding k_2, and a chance of finding k_3, and so on. Its chance of finding *at least one* of the 2^{10} target keys is, conjecturally, close to 2^{-59}.

The attacker can increase the chance of success by running the machine repeatedly with new choices of β. For example, after running the machine 2^5 times, the attacker has chance close to 2^{-64} of finding k_1, and chance close to 2^{-54} of finding at least one key. These 2^5 runs take only slightly more time than 2^{32} AES computations. If the machine does find a key—or a collision that doesn't produce a key—then the machine has to take extra time; but this is a rare event, so it can be ignored in evaluating the machine's performance.

Let me summarize. This machine has, conjecturally,

- chance close to 2^{-64} of finding k_1 after the time for 2^{32} AES computations;
- chance close to 2^{-54} of finding at least one of the 2^{10} target keys after the time for 2^{32} AES computations;
- chance close to 2^{-32} of finding k_1 after the time for 2^{64} AES computations;
- chance close to 2^{-22} of finding at least one of the 2^{10} target keys after the time for 2^{64} AES computations;
- chance close to 1 of finding at least one key after the time for 2^{86} AES computations; and
- chance close to 1 of finding most of the 2^{10} target keys after the time for 2^{96} AES computations.

The machine has reasonable size: 2^{32} AES circuits, plus a comparable amount of memory.

Asymptotics

The same machine design can be scaled up to p parallel key-search circuits, for a wide range of values of p. The size-p machine has a good chance of discovering a target b-bit key in the time for $2^b/p$ cipher evaluations.

Even better, the machine can be simultaneously applied to q keys for any q up to roughly $\sqrt{p}$. The machine then has a good chance of discovering *most* of the keys in the time for $2^b/p$ cipher evaluations.

The $\sqrt{p}$ limit arises as follows. Each key is fed to $2^4p/8q$ key-search circuits. Each circuit runs for $2^4p/8q$ iterations. The subsequent sorting of 2^4p numbers takes $8\sqrt{2^4p}$ adjacent compare-exchange steps, which become a bottleneck as q grows past $\sqrt{p}$.

See Section 5 for a variant that efficiently handles larger q.

The success probability of the machine against each key scales linearly with time. For example, in the time for $2^b/pq$ cipher evaluations, the attacker has a good chance of discovering *at least one* key. In the time for $2^{b-20}/pq$ cipher evaluations, the attacker has roughly a 2^{-20} chance of discovering at least one key. This pattern continues down to a very small amount of time.

For $q = 1$, a simpler machine does the same job: distribute $H(k_1)$ to many circuits, each of which searches sequentially through a range of possibilities for k_1. Computations for one key can be merged to some extent with computations for the next key, saving time.

3 The serial alternative

A very small *serial* computer—a single key-search circuit—can compute a b-bit key k from $H(k)$ in at most 2^b evaluations of H; e.g., at most 2^{128} AES evaluations for a 128-bit AES key.

If that's too much time, what does the attacker do? The obvious answer is to build the standard parallel brute-force key-search machine described in Section 2. This is a time-processor tradeoff, trading price linearly for performance.

A much worse answer is to use a time-memory tradeoff: a *serial* computer that computes k from $H(k)$ in fewer than 2^b evaluations of H, after a massive precomputation. This time-memory tradeoff trades price for performance, but not linearly; it might set speed records for serial computers, but it is painfully slow compared to a properly designed parallel computer.

Consider, for example, the same attack as in Section 2, but using memory on a serial computer, rather than processors on a massively parallel computer. We'll see that the serial computer is much, much, much slower and not much smaller.

The attacker selects a 12-byte string β and $2^{36} - 2^{33}$ random 16-byte strings $r_1, r_2, \ldots$. He computes, serially, $Z(\beta, 2^{23}, r_j)$ for each random r_j, and stores the results in an associative array. Then, for each target $H(k_i)$ in turn, the attacker computes $Z(\beta, 0, H(k_i)), \ldots, Z(\beta, 2^{23} - 1, H(k_i))$, and looks up each result in the associative array. Conjecturally he has probability close to 2^{-69} of finding k_i, exactly as in Section 2.

This serial machine is billions of times slower than the parallel machine. The serial machine performs $2^{23}(2^{23} - 1)/2$ AES evaluations for each target k_i, totalling about 2^{55} AES evaluations; and, even worse, 2^{23} AES evaluations for each random r_j, totalling about 2^{59} AES evaluations. For comparison, the parallel machine finishes in the time for just 2^{27} AES evaluations.

The serial machine is not billions of times less expensive than the parallel machine. It isn't even ten times less expensive. It's about half the size of the parallel machine: it doesn't have the 2^{32} AES circuits, but it does have the same $\approx 2^{42}$ bytes of memory. Perhaps I'm overestimating the cost of memory compared to an AES circuit; if so, simply expand the amount of memory in both machines to balance the memory-circuit cost with the AES-circuit cost, and the conclusion will be the same.

To summarize: The time-memory tradeoff produces a ludicrously unbalanced machine with tons of memory waiting for one serial CPU. The attacker would have to be completely insane to use this serial machine.

Asymptotics

The disadvantage of a time-memory tradeoff, compared to a time-processor tradeoff, grows linearly with the size of the machine. For example, a serial key-search machine with about 2^{74} bytes of memory is about 2^{63} times slower than a comparably priced parallel key-search machine, and about 2^{33} times slower than a parallel key-search machine costing 2^{30} times less.

Sometimes I see one cryptanalyst arguing that a system has been "broken" by a time-memory tradeoff on a serial computer with 2^{90} bytes of memory, and another cryptanalyst arguing that building 2^{90} bytes of memory is awfully difficult so the system has not been "broken." This is a pointless argument about an incompetent machine design. The standard parallel brute-force key-search machine is billions of times less expensive than that serial computer *and* billions of times faster. Anyone who thinks that time-memory tradeoffs are worrisome should be utterly terrified by the vastly superior price and performance of a properly designed parallel machine.

Fancier serial attacks

A recent paper advertises a serial computer with 2^{380} bits of fast memory that, using a complicated algorithm, takes only 2^{534} cycles to identify a 544-bit key. This attack is portrayed as being successful because it is (slightly) "faster than exhaustive search."

Let's compare this serial computer to the standard *parallel* brute-force key-search machine with, say, 2^{200} key-search circuits. The parallel machine is nearly 2^{200} times faster than this serial computer, and vastly less expensive.

Why did the author of this paper characterize this serial attack as successful cryptanalysis? It's simply not true that the attack is "faster than exhaustive search"—unless you assume that the attacker is forcing himself to use a serial computer, i.e., that the attacker is an idiot. Perhaps a parallelization of the complicated algorithm could beat the parallel brute-force key-search machine, but I doubt it: at first glance, parallelization will improve the price-performance ratio of the complicated algorithm by a factor of only about 2^{180}.

This is certainly not an isolated mistake. Most of the "breaks" that I see in the literature are slightly faster than *serial* brute-force search but are much slower than a much less expensive *parallel* brute-force search. Many people seem to think that the number of cipher rounds "broken" by a differential attack, for example, is the number of rounds for which the differential attack is marginally faster than a *serial* brute-force search—ignoring the question of whether the differential attack is faster than a *parallel* brute-force search.

This mistake should not be tolerated. A cryptanalytic machine is a failure if it's slower than the standard parallel brute-force key-search machine at the same price. New attacks must be compared to the best previous attacks, not merely the best previous *serial* attacks.

4 Defending against the standard attack

Consider again the standard parallel machine in Section 2, with 2^{32} AES circuits. Recall that, after 2^{64} AES computations, this machine has chance close to 2^{-32} of finding a target key k_1, and chance close to 2^{-22} of finding at least one of the 2^{10} target keys.

Is this an acceptable level of security? Many people don't think so. This section analyzes two different ways to modify cryptographic systems to make the attacker's job more difficult.

Input-space separation

The standard parallel machine attacks a large batch of target keys at about the same cost as attacking a single key.

Often the attacker's benefit is proportional to the number of target keys successfully found. Perhaps the attacker is trying to steal the computational power of as many target computers as possible, and each extra key lets him steal power from an extra computer. The attacker's cost-benefit ratio is then divided by the number of target keys.

This amortization relies on the attacker being given $H(k_1), H(k_2), \ldots$ for the same function H: e.g., $\mathrm{AES}_{k_1}(8675309), \mathrm{AES}_{k_2}(8675309), \ldots$. If there's no overlap between the inputs to AES_{k_1} and the inputs to AES_{k_2} then the attacker can't simultaneously attack k_1 and k_2.

"Aha!" one might say. "We should design our cryptographic protocols so that different keys are applied to disjoint input sets!"

Example: De Canniére, Lano, and Preneel in [2, Section 5] suggest designing stream ciphers so that nonces are as long as keys, and then choosing nonces pseudorandomly. This suggestion fails to achieve the goal—a typical nonce will still be used for many keys if the number of nonces is below the number of keys *times the number of messages per key*—but one can make nonces even longer to prevent repetition.

Unfortunately, [2] focuses entirely on the costs and benefits for the attacker, and neglects to consider the costs and benefits for the cryptographic users:

- Many stream ciphers—counter-mode AES, for example—have small input sizes and would have to be radically reworked, presumably losing speed, to handle long nonces. The authors of [2] assert that "the state is already at least twice the key size"; this is true for some stream ciphers but false for counter mode.
- Even when a stream cipher easily accepts a long nonce, generating a long pseudorandom nonce is much more expensive than generating a standard sequential nonce. Long pseudorandom nonces cost CPU cycles to compute—the pseudorandom number generator is another attack target, so it needs to be at least as strong as the main cipher—and they cost bandwidth to transmit.
- As for benefits: Input-space separation doesn't make *my* key more difficult to find. It stops the attacker from finding other people's keys as part of the same computation, and perhaps this difference will deter the attacker, but perhaps it won't.

To summarize, input-space separation has limited benefits and quite noticeable costs. See below for a different approach that has much larger benefits and much smaller costs.

Larger keys

A more obvious way to make the attacker's job much more difficult is to *use a larger key*. Why fool around with 128-bit keys when we can simply use 256-bit keys?

The benefits to users are clear, and far outweigh the benefits of input-space separation. Brute-force key-search attacks suddenly become 2^{128} times slower—i.e., completely impractical for the foreseeable future.

"But 256-bit AES takes 14 rounds, while 128-bit AES takes only 10!" some people will argue. "In general, 256-bit ciphers have more rounds than 128-bit ciphers. That's a quite serious cost in speed; we need to consider other ways to use the same CPU cycles."

But this argument confuses two different changes. We can switch to 256-bit keys, effectively eliminating brute-force attacks, *without* increasing the number of rounds. The only cost is the cost of generating and storing larger keys, which is normally much smaller than the cost of generating and transmitting large random nonces for every message. The benefit is much larger.

"But 256-bit AES deliberately uses 14 rounds to keep us secure against non-brute-force attacks!" some people will argue. "There's an 8-round attack taking time only 2^{204}, for example. Okay, okay, that's on a machine with 2^{104} bits of memory, but maybe some additional ideas will produce a 10-round attack more efficient than a 256-bit parallel brute-force key-search attack."

But this argument confuses security level with key size. I never said that this change would produce a 256-bit security level. I said that it would make the attacker's job much more difficult—producing larger benefits than input-space separation at lower cost. If the resulting security level is, say, 192 bits, then the mission has been accomplished.

"But you're not allowed to use keys larger than the target security level!" some people will argue. "It's, um, against the law! The standard definition of security varies with your key size!"

But that's a silly definition of security. Perhaps the most efficient way to achieve a 192-bit security level is with a system having a 256-bit key. If the user wants a 192-bit security level, then the user should select that system. There's no justification for demanding a reduced key size.

(Helix, introduced by Ferguson, Whiting, Schneier, Kelsey, Lucks, and Kohno in [3], is an example of a fast stream cipher that uses a key size above its target security level. I don't know whether this is the most efficient approach, but I certainly would not want it to be excluded from consideration.)

I'm not saying that increasing the number of rounds is a bad idea. On the contrary: extra rounds have, historically, been quite effective at stopping non-brute-force attacks. But this has nothing to do with the question of whether randomness should be added to nonces or to keys.

5 The variant parallel brute-force key-search attack

This section describes a variant of the standard parallel brute-force key-search machine. This variant is a little slower but has the advantage that it can handle many more keys simultaneously.

The problem

The problem is the same as in Section 2: the attacker is trying to find 16-byte AES keys $k_1, k_2, \ldots$, given $H(k_1), H(k_2), \ldots$. Let's again assume that the attacker is facing 2^{10} keys overall.

The variant key-search circuit

The attacker builds a very small key-search circuit. The key-search circuit has three inputs: a 23-bit string α, a 16-byte string β, and a 16-byte string s. The key-search circuit has one output: a 16-byte string $D(\alpha, \beta, s)$ defined recursively as $\beta \oplus s$ if $\beta \oplus s$ begins with α, and as $D(\alpha, \beta, H(\beta \oplus s))$ otherwise.

This key-search circuit is attached to a comparable-size memory circuit that buffers 2^4 inputs and 2^4 outputs, as in Section 2.

How long does the key-search circuit take to produce its 2^4 outputs? The 16-byte strings $\beta \oplus s$ bounce around pseudorandomly, and one out of every 2^{23} strings begins with α, so the number of iterations per output is typically on the scale of 2^{23}. Maybe less; maybe more; occasionally the circuit falls into a loop and never produces an output; but the key-search circuit typically computes outputs for most of its inputs within 2^{27} iterations.

The variant key-search machine

The attacker now builds a machine with 2^{32} key-search circuits in a $2^{16} \times 2^{16}$ mesh. Each key-search circuit has its own comparable-size memory circuit, and is connected to its immediate neighbors, as in Section 2.

The attacker feeds 2^{36} inputs to his 2^{32} key-search circuits. Specifically, the attacker selects a 23-bit string α, a 16-byte string β, and $2^{36} - 2^{10}$ random 16-byte strings $r_1, r_2, \ldots$; generates one input (α, β, r_j) for each random r_j; and generates one input $(\alpha, \beta, H(k_i))$ for each target $H(k_i)$.

All 2^{32} key-search circuits now run in parallel for 2^{27} iterations, producing $D(\alpha, \beta, r_j)$ for (conjecturally) most j's and $D(\alpha, \beta, H(k_i))$ for (conjecturally) most i's. This takes slightly more time than 2^{27} AES computations.

The attacker then sorts the D values. If the sorting encounters a collision between two $D(\alpha, \beta, r_j)$'s, it throws one of those r_j's away. If it encounters a collision between $D(\alpha, \beta, H(k_i))$ and $D(\alpha, \beta, r_j)$, the machine pauses to redo the $D(\alpha, \beta, r_j)$ computation. If any intermediate value (α, β, s) satisfies $H(\beta \oplus s) = H(k_i)$ then the machine prints $\beta \oplus s$ as a guess for k_i.

Heuristic analysis

What's the chance that a particular key, say k_1, is found by this machine?

The crude estimate is that there are 2^{36} values $D(\alpha, \beta, r_j)$, each involving 2^{23} intermediate values (α, β, s) and 2^{23} inputs $\beta \oplus s$ to H, for a total of 2^{59} inputs to H. If any of those inputs bumped into k_1 then the machine will find k_1. Specifically, if an intermediate value (α, β, s) in the computation of $D(\alpha, \beta, r_j)$ satisfies $\beta \oplus s = k_1$, then $D(\alpha, \beta, r_j) = D(\alpha, \beta, H(\beta \oplus s)) = D(\alpha, \beta, H(k_1))$; the sorting will discover this collision, will locate the same intermediate value (α, β, s), and will print $\beta \oplus s = k_1$ as its guess for k_1. This occurs with probability $2^{59}/2^{128} = 2^{-69}$.

This estimate is slightly overoptimistic, for all the reasons in Section 2 and more: for example, some r_j's fail to produce values $D(\alpha, \beta, r_j)$. These failures are a disadvantage of the variant machine compared to the standard machine. But one can reasonably conjecture that a random choice of α, β has probability larger than 2^{-71} of finding k_1, not much worse than the standard machine.

As in Section 2, the machine simultaneously has a chance of finding other keys; and the attacker can increase the chance of success by running the machine repeatedly with new choices of α, β. The variant machine also has an advantage over the standard machine: it uses far fewer key-search-circuit inputs per target key, so it can handle far more target keys simultaneously.

References

1. Dan Boneh (editor), *Advances in cryptology: CRYPTO 2003, 23rd annual international cryptology conference, Santa Barbara, California, USA, August 17–21, 2003, proceedings*, Lecture Notes in Computer Science, 2729, Springer, Berlin, 2003. ISBN 3–540–40674–3. MR 2005d:94151.
2. Christophe De Cannière, Joseph Lano, Bart Preneel, *Comments on the rediscovery of time memory data tradeoffs* (2005). URL: `http://www.ecrypt.eu.org/stream/`.
3. Niels Ferguson, Doug Whiting, Bruce Schneier, John Kelsey, Stefan Lucks, Tadayoshi Kohno, *Helix: fast encryption and authentication in a single cryptographic primitive*, in [4] (2003), 330–346. URL: `http://www.macfergus.com/helix/`.
4. Thomas Johansson (editor), *Fast software encryption: 10th international workshop, FSE 2003, Lund, Sweden, February 24–26, 2003, revised papers*, Lecture Notes in Computer Science, 2887, Springer-Verlag, Berlin, 2003. ISBN 3–540–20449–0.
5. Philippe Oechslin, *Making a faster cryptanalytic time-memory trade-off*, in [1] (2003), 617–630.
6. Michael J. Wiener, *The full cost of cryptanalytic attacks*, Journal of Cryptology **17** (2004), 105–124. ISSN 0933–2790. URL: `http://www3.sympatico.ca/wienerfamily/Michael/`.