Transparent Harddisk Encryption

Thomas Pornin

Département d'Informatique, École Normale Supérieure,
45 rue d'Ulm, 75005 Paris, France
thomas.pornin@ens.fr

Abstract. This paper introduces a new block cipher, and discusses its security. Its design is optimized for high-bandwidth applications that do not have high requirements on key-schedule latency. This paper also discusses several security issues about such an application: harddisk encryption.

Keywords: bitslice, encryption, harddisk, mobile computing

1 Introduction

Today's secret key cryptosystems are designed to be versatile enough to fit most usages in a wide variety of environments. The recently chosen new american standard of symetric encryption, the AES [1], is a perfect illustration of that fact: appart from its seemingly good security, it was chosen because it could run reasonably fast on a modern workstation, a low-end personal digital assistant, a smartcard or a specific ASIC. This speed and easiness of implementation are requirements for what the AES was designed to be: a standard; interoperability issues imply that all applications must use the same algorithm, so it must be good everywhere.

However, there are some applications where requirements are different: one of them is on-the-fly harddisk encryption. Such encryption is needed to prevent divulgation of important data if a harddisk is stolen, or scanned during an inactivity period (some sort of lunch-time passive attack). This is especially important for mobile systems, such as portable computers. Another class of attacks that could be worth to counter, is active attacks: an attacker modifies data on the disk. Even if the modification is essentially random, such tampering should be at least detected.

Let us detail what is needed, and what is not:

- We need a very fast cipher; security is not a goal in itself, but a necessary evil used to protect other jobs; and since modern operating systems implement multitasking, only a marginal proportion of the cpu power should be used to perform encryption.
- We do not care about key-schedule latency: the key-schedule is performed only once per session, at boot time, and the cost can be further reduced, so that it should doable in a user-compatible time (the user will not want to wait several minutes every morning).

- We encrypt data-blocks of size multiple of 512 bytes: this is the standard size of an harddisk sector; all reads and writes from and to the disk are performed with this granularity at least (on some systems, it might be higher).
- We need to handle random accesses to the disk, with low overhead; we cannot afford, for instance, extra physical reads on the disk.
- We run on a modern computer, with many registers[1].

Classical block ciphers are ruled out for speed reasons; as a rule of thumb, the maximum allowed cpu overhead should be 15%, on a 1 GHz cpu, with a disk running at 20 Mbytes per second. This means that a bandwidth of at least 120 Mbytes per second (when full cpu is used) is needed. Algorithms such as the AES [1], Blowfish [2] or CS-Cipher [3], although considered as fast, will be limited to about 50 Mbytes per second.

Stream ciphers are also out of the question, due to random access; stream ciphers have a state, that needs to be maintained, in order to encipher and decipher. The initial state depends on the key, and its construction usually requires some time. For instance, although the bandwidth obtained with RC4 is high, the key schedule is rather slow with regards to the production of 512 bytes of stream. Besides, the ciphertext is often too malleable: if the attacker guesses the plaintext, he can easily change that plaintext to whatever he wants.

So we need some sort of very fast block cipher; we present such a cipher in this paper. We will first recall the so-called bitslice programming technique, as described by Eli Biham [4], then describe the algorithm itself, and discuss implementation and security issues. A final section will explicit some general problems related to harddisk encryption, and show how our cipher helps in solving them.

2 Bitslice

Bitslicing is an implementation trick, classical among electronicians, but never really published, and therefore rediscovered several times. Eli Biham was the first cryptographer to document it in [4]; the method basically boils down to an alternate representation of data that allows software implementations to work like hardware ones, with similar optimizations. Bitslice code is also called *orthogonal code*, to refer to this alternate representation.

2.1 Abilities of Modern Processors

Modern processors are more and more of the RISC trend; this means that they have many, wide registers, and are able to perform bitwise logic operations between these registers at high speed. They are however relatively bad at handling byte-formatted data, such as ASCII text.

[1] This extends to the PC, although the Intel instruction set does include only a limited number of addressable registers; see section 2.1.

An emblematic processor is the Alpha [5]: it has 32 64-bit registers, all of them being equivalent; there is no specialized register[2]. All calculating instructions take two registers as operands, and a third one as destination. This design is a good example of what processors will look like in the future; Intel chosed a similar design for its new, market-leading processor, the Itanium [6], which should replace the long-lived Pentium family. The Alpha and the Itanium are native 64-bit processors.

Actually, Pentium processors are already quite RISC: they sure still handle the old 8080-compatible CISC code, with few registers and many complex instructions; but most of those complex instructions are there for backward-compatibility only, and are slow, so the compilers do not use them. Besides, there are many internal registers, and the processor renames, aliases and duplicates the registers visible to the programmer. Memory accesses to the internal cache memory are also made very fast, so we can consider those processors as being on the RISC side. The Pentium is a 32-bit processor, but already owns some 64-bit registers, in the MMX unit.

2.2 Orthogonalization of Data

The natural reflex of the cryptosystem programmer, when a 32-bit data must be used, is to store it into a register. This approach has the following drawbacks:

- When the registers are wider than the data, some of the computing power of the processor is lost.
- Bit permutations cost much; those operations are current in cryptosystems since they help in creating a correct avalanche effect. But those permutations are a mere data routing, and do not perform any real calculation.

The orthogonal representation is the following: spread the data among many registers, one bit per register. You then calculate the algorithm as a circuit, with logic gates that map cleanly to the native bitwise operations of the processor. Since those operations are bitwise, they are performed on all bits of the registers at the same time; if you have n-bit registers, you perform n instances of the algorithm simultaneously. This is heavy parallelism, quite suited to situations where you have much data to encrypt, in ECB mode.

There remains the problem of getting input data to the appropriate storage ordering; this is equivalent to the transposition of a matrix. The figure 1 illustrates this transposition. See [7] for a $O(n \log n)$ method of transposition of a $n \times n$ matrix, when n-bitwise logical operations and shiftings are atomic.

2.3 Applicability

With bitslice, bit permutations are "free": the code just has to use the right register. This is solved at compilation time and does not induce runtime cost.

[2] There is actually one: the register 31 contains always 0; but since this value does not change, it can be safely "duplicated" inside the processor and therefore does not constitute a bottleneck.

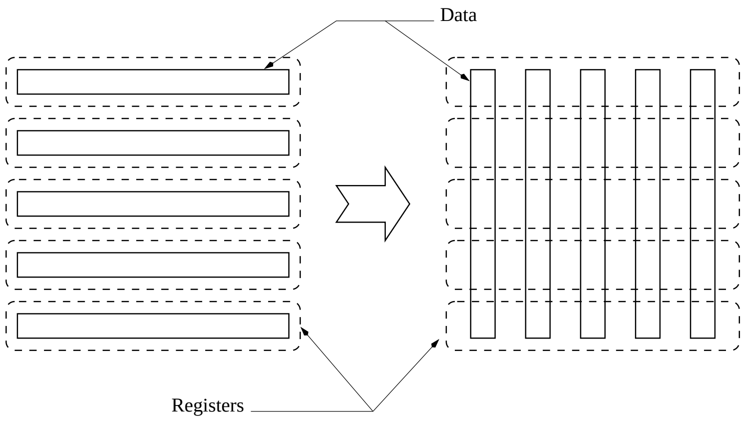

Fig. 1. othogonalization of data

Moreover, the ALU (Arithmetic and Logic Unit) is used at its full potential, since the whole width of registers is used. However, some operations become much more complex: table lookups must be replaced by equivalent circuitry, which means BDD (Binary Decision Diagrams); additions require manual carry propagation; multiplications are definitely out of the question.

Moreover, the algorithm must be representable as a circuit; this is anyway a desirable characteristic for block ciphers, since data-dependant branches lead to timing attacks [8].

To sum up, some cryptosystems are well-optimized for bitslice, others are not. DES can be implemented very efficiently this way; it was done for the DESchall [9] (a software-based DES cracking challenge by exhaustive search of the key space). Serpent [10], candidate to the AES, was also designed to be implemented using these technics. We present in this paper a new algorithm, called FBC (as "Fast Bitslice Cipher"), which is optimized for speed under a bitslice implementation.

3 The FBC Algorithm

3.1 General Structure

The FBC algorithm is a r-round Feistel cipher; it works on w-bit values (w is even). The confusion function is simple: each output bit is the bitwise combination of two different input bits. Four combinations are used: AND, OR, NAND and NOR. Which combinations are used on which bits, is key-dependant and round-dependant. The figure 2 illustrates this setup. Due to practical implementation issues, w must not exceed 512.

The three main ideas are:

– use a simple, fast round function with many rounds (for instance, $r = 64$);

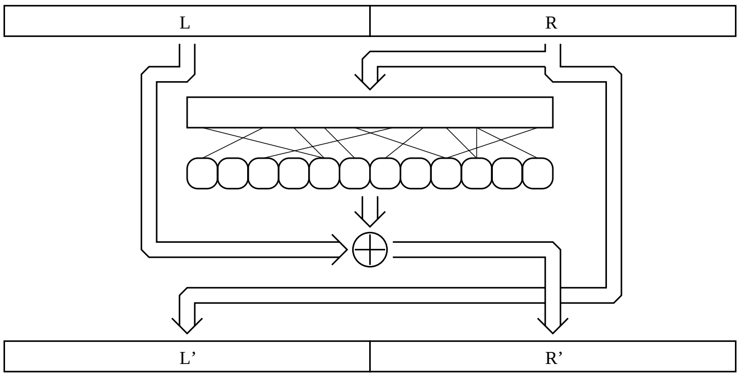

Fig. 2. one round of FBC

- derive each round subkey from the key using a cryptographically strong pseudo-random generator (so that attacks on subkeys cannot be proven to exist, neither can they compromise other subkeys);
- make code generation part of the key schedule; this is slow but allows for many key-dependant features.

In a more formal way: for each binary value, we count bits from left to right, leftmost bit is numbered 1. For the round i, the input is split into two equal parts of size $w/2$: the left part L_i and the right part R_i. There exist $w/2$ functions $\tau_i^j (1 \leq j \leq w/2)$ and two permutations ϕ_i and ψ_i of $w/2$ elements such that:

$$\forall j, \tau_i^j = \text{AND, OR, NAND or NOR}$$

$$\forall j, \phi_i(j) \neq \psi_i(j)$$

The result T_i of the confusion function is such that the bit j of T_i $(1 \leq j \leq w/2)$ is equal to $\tau_i^j(\phi_i(j), \psi_i(j))$. The output of the round is the concatenation of L'_i and R'_i, in that order, where:

$$\begin{aligned} L'_i &= R_i \\ R'_i &= L_i \oplus T_i \end{aligned}$$

After the last round, the left and right part of the result are swapped; this is made so that the decryption algorithm is exactly the same than the encryption algorithm, but the definition of the ϕ_i, ψ_i and τ_i^j functions.

To complete this scheme, we must specify how those functions are chosen from the master key. We use the master key as a seed for a pseudo-random generator, that uses a cryptographically strong hash function.

3.2 The Pseudo-random Generator

The secret key is a k-bit value, where k ranges from 0 to 352. As usual, if k is lower than 80, the scheme is to be considered as vulnerable to exhaustive key search attacks. The AES defines key sizes of 128, 192 and 256 bits; FBC allows these, and other sizes as well, so any useful security level can be achieved with FBC.

We use the hash function SHA-1 as defined in [11]. This specification defines the application of SHA-1 on an arbitrary bit stream, and includes a padding method to extend the bit stream to a size multiple of 512. We do not use that padding, so "SHA" is to be considered in this paper as "application of the SHA-1 core function to an unpadded block of 512 bits".

The key K is extended to the 352-bit value K' by appending zeroes to its right. S is a 160-bit variable which will contain the "state" of the generator. The algorithm is the following:

- 1. $S \leftarrow 0$
- 2. $S \leftarrow \mathrm{SHA}(K'||S)$
- 3. The 20 bytes of S are emitted (leftmost first)
- 4. Return in 2

($||$ denotes concatenation).

Therefore the generator emits bytes. These bytes will be used to choose random numbers between 0 and $w/2 - 1$, which is exactly why w must be at most 512; otherwise, the definition of the key schedule should be adapted.

We will have to choose integers ranging from 0 to some limit n, where n is a posivite number strictly smaller than $w/2$. We calculate m the greatest positive multiple of $n + 1$ that is smaller or equal to 256; for instance, if $n = 6$, we have $m = 252$.

To choose a random number from 0 to n, we get one byte b from the random generator; if this byte is greater or equal to m, we get another byte, until we have a value strictly smaller than m. It is easy to see that the average number of invocations of the random generator is at most 2, so this process is not especially slow. The random number is defined to be the euclidian rest of the division of b by $n + 1$. This process ensures that all integers between 0 and n have an equal probability to appear.

3.3 Choice of a Random Permutation

We must choose random permutations of $w/2$ elements; we will use the following algorithm:

- 1. Fill an array p of $w/2$ elements with the numbers from 1 to $w/2$ in ascending order ($\forall i, p[i] = i$); this array represents the identity permutation.
- 2. For i ranging from 2 to $w/2$
- 3. Choose a random integer a_i between 0 and $i - 1$
- 4. If $a_i + 1 \neq i$, swap the contents of $p[a_i + 1]$ and $p[i]$ (this is equivalent to the composition of p with the transposition $(i\ (a_i + 1))$)
- 5. End for

The chosen permutation is represented by the contents of p at the end of the execution of the algorithm ($p[5] = 8$ means that the permutation sends the fifth element of its input to the eigth emplacement of its output). This algorithm ensures that all permutations of $w/2$ elements have an identical probability to be chosen [12].

3.4 Choice of the Elements of Each Round

To choose the elements constituting the round i (the two permutations ϕ_i and ψ_i, and the $w/2$ boolean functions τ_i^j), we proceed this way:

- 1. Choose randomly ϕ_i.
- 2. Choose randomly ψ_i.
- 3. If there exists at least one j between 1 and $w/2$ such that $\phi_i(j) = \psi_i(j)$, go back to 2.
- 4. For each j from 1 to $w/2$, get one random byte; the euclidian rest of the division of that byte by 4 is a value between 0 and 3. The function τ_i^j will be a AND, OR, NAND and NOR for values of, respectively, 0, 1, 2 and 3.

The elements of each round are chosen from the first round to the last. On the average, for each ϕ_i, we will have to try $e \approx 2.7$ permutations ψ_i before finding one matching the criterion of point 3 (finding a matching ϕ_i is equivalent to finding $\psi_i \circ \phi_i^{-1}$, permutation with no fixed point; see [13] for the proportion of such permutations among all permutations of $w/2$ elements).

4 Implementation of FBC

4.1 Software Implementation

FBC is designed to be implemented in software, using bitslicing techniques. Such code is rather difficult to write, but we developped some sort of automatic tool to produce bitsliced C code from an *ad hoc* description of the algorithm, which can be generated from the key schedule algorithm. That tool is not very well developped but is available for free download and use (see [14]). The C code generated for a fully deployed 64-round FBC is a huge function with about 5000 local variables and 5000 statements; the C compiler fails utterly on such input, so the code must be sliced into small groups of four rounds or so, easier to understand by the compiler.

We ignore the cost of othogonalization of data before encrypting and after encryption; actually, not performing such orthogonalization is equivalent to performing a known, fixed permutation on input and output data blocks (the 512-bit blocks if we perform 64 parallel encryptions with a 64-bit block size). Such a permutation has no security implication, so we can add that permutation, which actually voids the cost of orthogonalization.

Encryption bandwidth achieved for the moment on an Alpha 21164 processor running at 500 MHz is about 32 Mbytes/s using FBC with 64-bit words and

64 rounds ($w = 64, r = 64$); the code is not yet fully optimized and we are still working on it. This speed is half the speed requested (we wanted 120 Mbytes/s on a 1 GHz processor) but is explanable by the relatively old design of the 21164 (that processor was first announced in August 1994, which is a very remote epoch in the rapidly moving chip industry). The 21164 can issue up to four instructions in each cycle, but only two load/store instructions and two logical instructions. Moreover, a load instruction cannot occur during the second next cycle of a store instruction. This restriction, bound to the aging design of the 21164, is responsible for the seemingly bad performance of FBC on that processor; the newer 21264 does not have that problem.

The number of logical τ functions to calculate for one encryption is $\frac{wr}{2}$. If n is the size of each register, the bitslice code will calculate n parallel instances of the algorithm, thus encrypting wn bits. The number of function evaluations needed for each data bit is then $\frac{wr}{2wn} = \frac{r}{2n}$. Since the wanted rate is about one bit per clock cycle, we must execute $\frac{r}{2n}$ τ functions per cycle (it is worth noticing that this value does not depend upon the block size used).

Bitslicing code uses many registers, much more than the really available registers in the processor; therefore, those are to be considered as a cache on the stack, where the values are stored. So data management still comes up as the most constricting issue. Each τ function will require two input operands and one output operand; since each input bit is used twiced in two different τ functions, the number of memory management operations needed can be reduced to one load and one store for each function. Due to the restrictions on such operations, an average of 3 cycles are needed per function. With the unavoidable additional cost of data transfering (this is administrative task outside the core of the cipher), this explains the “low” rate achieved on the 21164 (that rate is equal to the best rates achieved by ciphers such as the AES on the same machine).

However, the current market-leading Alpha processor is the 21264, which has much lower restrictions on memory accesses; from its specifications, it should achieve the correct performance (one cycle per bit enciphered), whereas classical cryptosystems, which use more complex structures of the processor, will not benefit as much of the generation shift (speed measurements [15] from the AES competition show that the fastest candidates would run at 2 clock cycles per bit enciphered on a 21264-equivalent processor). Optimization of the code on the 21264 architecture is still undergoing work.

To the very least, “64 rounds” is a conservative number, in a security point of view. That number could be lowered, and the speed of FBC would raise correspondingly.

4.2 Hardware Implementation

FBC is well-suited for FPGA (“Field Programmable Gate Arrays”) implementations. FPGAs are programmable chips, which can host any circuit, and can be redesigned in little time (less than a second) with no loss.

For a FPGA implementation, the key schedule algorithm would produce a circuit design, to be loaded into the FPGA. Bitslicing (which is parallelization in space) becomes pipelining (parallelization in time), so a fully deployed FBC should run at one block per cycle; since each round is very simple (only one logic gate layer per round), a very fast clocking rate could be achieved. Since late Xilinx FPGA [16] chips may run at rates over 150 MHz, an instance of FBC with 64-bit words on such a chip would encipher over 10 Gbits of data per second; this is the fastest data rate achieved by the best production optic fibers.

If speed is at stake, specific hardware usable, and key schedule time unimportant, then FBC is the way to go.

5 Security of FBC

Security of FBC is based upon the following paradigms:

- many rounds,
- unpredictable random subkeys,
- key-dependant permutations and non-linear functions.

5.1 Many Rounds

It has been said for a long time that "take whatever round function you want, it will be secure if you put up enough rounds". This assertion used to be a joke, but it actually makes much sense.

Modern cryptanalytic attacks, such as differential and linear cryptanalysis, tend to have a complexity exponential in the number of rounds; especially, if the probabilistic advantage of the attacker is 1/2 on one round, then 64 rounds will lower that advantage to 2^{-64}, a quite appropriate number for a FBC operating on 64-bit blocks.

5.2 Unpredictable Random Subkeys

Most cryptanalytic attacks use the fact that information on the key used in one round somehow shows up in a predictable way in some other rounds. Thus FBC produces all key-dependant round material with a cryptographically strong pseudo-random generator, seeded by the master key. If any information, learned or guessed, on the subkey of some rounds can be applied to another round by the attacker, then this would contradict the strength of the generator. Besides, even if all key-dependant material are guessed, thus giving some strong knowledge about the output of the generator, it would still be computationnaly infeasible to guess the master key; thus, a successful attack on a FBC-encrypted link with a simple daily key-updating policy would be limited to one day of decryption.

5.3 Key-Dependant Permutations and Non-linear Functions

In FBC, the permutations are made key-dependant as an attempt to make the avalanche effect unanalyzable by the attacker. One consequence is that permutations cannot be guaranteed to be "strong". We did some sample measures of the avalanche effect; here is the average number of data bit potentially modified by one bit after several rounds, for $w = 64$:

rounds	**bits potentially modified**
1	2.00
2	4.91
3	11.05
4	22.81
5	39.54
6	54.76
7	62.34
8	63.93
9	63.99
10	64.00

This means that the total avalanche effect is obtained in 10 rounds, to compare with the suggested value of 64 rounds.

The τ functions are dependant on the key, and chosen among the four functions AND, OR, NAND and NOR, which are the four symetric non-linear boolean functions. Given random inputs and random functions in this set, the output is well-balanced and statistically not correlated to the input.

5.4 Security Sum Up

The FBC design looks quite secure, with the rule of thumb $r = w$, which means "as many rounds as bits in the block size". This is a conservative estimation, based upon the assumption that if the block size if 64 bits, then the scheme should be secure against attacks using 2^{64} adaptively chosen plaintexts, which is far from being applicable in real life. Typically, if the enciphered text is a harddisk, the maximum amount of ciphertext provided is about 2^{32} blocks or so. Yet, security margins should not be too much fiddled with.

6 Harddisk Encryption

The problem of harddisk encryption is complex, and depends upon the type of attack considered. We will consider passive and active attacks, and detail the threat model and several corresponding solutions.

6.1 Passive Attacks

The model is the following: a computer stores confidential data on its harddisk; the computer or its disk might be stolen while it is not powered, therefore the data on the disk must be stored encrypted only. The attacker is supposed to be able to guess most of the encrypted plaintext, and must not learn anything about the remaining plaintexts.

Several products already address, or try to address this security issue. Some work on a file basis, masking the real names and internal contents of the files; this maps cleanly to network filesystems protocols such as NFS or Samba, which use file-oriented semantics. However, this leaks information, mainly the number of files created, their sizes and modification dates. Therefore the real security provided by those systems is only marginal.

Other products build up real enciphered blocks of data, upon which standard filesystems are applied. Those products use classical block ciphers and induce a performance hit which leads users into reserving encryption for really important data only. This may help the attacker know what computers hold his target inside a large company; good security can therefore be achieved only if all harddisks are completely enciphered, which is possible only if the performance hit is very small.

FBC is designed to encipher data in ECB mode, so that the parallelism given by the bitslice representation of data can be used. ECB mode has the following problem: input blocks are not randomized. Real life data is often very redundant, and equal blocks will be enciphered the same, and the attacker will be able to detect them. The countermeasure is to "add" a counter: each block, before encryption, is combined with its block number prior to encryption, with an addition or a bitwise XOR. The cost of such modification is neglectable with regards to the cost of encryption itself (on an Alpha, it will cost one cycle for 64 bits of data).

One FBC issue is that the key schedule implies the generation of code, a rather slow process (it can take up to several minutes) and which uses much code (a C compiler is not a small application, usually). This can be addressed the following way: the result of the key schedule, that is, the code that encrypts and decrypts, is stored encrypted on the disk, using some other block cipher, the AES for instance. The decryption is done only once at boot time, so there is no real performance issue here. The key used to decipher the FBC code needs not be the same key as the FBC one.

6.2 Active Modifying Attacks

We consider here the following model: the computer is stolen while being unpowered, its contents are modified, and the computer is put back in place before the theft is noticed. A random and destructive modification cannot be prevented, but we want to be sure that it would not go unnoticed. No existing product actually addresses this issue.

The classical solution is to store a MAC, which is easily built up with a hash function: the entire encrypted disk content, appended to some secret key, is processed through the function, and the result is written to some non-encrypted area of the disk (one such area must exist, to store the base decrypting software, that asks for the user key). The major drawbacks of this approach are:

- The speed of the process is limited to the speed of the disk, so it can take an impressive amount of time (one hour on today's typical disk); this has to

be performed at boot time, and no work can take place during this check. This is not acceptable from the user's point of view.
- More critical, the MAC must be calculated again at shutdown time. This is even more impossible to force on the user, especially since some shutdowns are due to OS crash or low battery.
- Even if the disk is logically almost empty, its whole content must be processed. An alternative is to build the MAC only on the blocks that contains allocated data, but this still uses a gruesome amount of time (today's basic operating system installation uses several hundred megabytes of disk space, not including oversized applications).

Here is one solution addressing these problems: each block is double-encrypted in the following manner: if the block number N contains P, and E is the encryption function, then its encrypted counterpart is $C = E(E(P) \oplus N)$. For each file, the exclusive or of all its constituting blocks is stored inside a file-specific structure that is also encrypted (on some systems[3], such a structure exists and is called an *inode*). This encrypted XOR is the MAC.

Using this scheme, the verification of files can be made asynchronously, as a background task; only a locking procedure must be used so that an individual file may not be used prior to its verification. More important, the MAC of each file is maintained during normal operation; this means that a modification of a file requires the reading of either the overwritten data (so that its contribution to the file MAC can be taken away) or the remaining data (that is the recomputation of the MAC). This is not an important cost, because, most of the time, modifications of files are either appending data to the end of the file, or emptying the file and rewriting it from scratch. Anyway, when modifying a file, the size of the extra reads is limited to half the size of the file.

The XOR with the block number between the two encryptions ensures that the data blocks are not swappable by the attacker; this operation is isolated from the plaintext *and* from the ciphertext by the two encryptions. The assumption is that the block cipher is a random permutation, therefore any modification to the ciphertext leads to a random, uncontrollable modification of the plaintext.

The main drawback of this scheme is the double encryption, which halves performance. Therefore the use of a very fast cipher is critical for such a design.

7 Conclusion

We presented a new cipher, FBC, designed to achieve high encryption speed on a modern workstation, adapted to on-the-fly harddisk encryption. We presented some arguments with regards to its security, and discussed some implementation issues. We also discussed some issues related to the problem of harddisk encryption and presented a generic scheme to ensure data integrity at low cost.

We believe that such work will be more and more used in the future, as mobile computing is generalizing at a fast pace. As a side note, OpenBSD [17] (a Unix-like system specialized in security) already includes an encryption mechanism for

[3] Actually, all Unix-like systems, including MacOS and Windows NT/2000.

its swap space. An open question (which does not apply to swap space, since the contents of swap space are not used across reboots) is the possibility of building a secure integrity verification scheme, that does not imply a complex shutdown procedure, neither double encryption, nor too many extra reads when a file is modified.

References

1. The Advanced Encryption Standard, `http://www.nist.gov/aes/`
2. Bruce Schneier, *Description of a New Variable-Length Key, 64-Bit Block Cipher (Blowfish)*, Lecture Notes in Computer Science, Advances in Cryptology, proceedings of Fast Software Encryption 93, Springer-Verlag, 1994, pp. 191–204.
3. Jacques Stern and Serge Vaudenay, *CS-Cipher*, Lecture Notes in Computer Science, Advances in Cryptology, proceedings of Fast Software Encryption 98, Springer-Verlag, 1998, pp. 189–205.
4. Eli Biham, *A Fast New DES Implementation in Software*, Lecture Notes in Computer Science, Advances in Cryptology, proceedings of Fast Software Encryption 97, Springer-Verlag, 1997, pp. 260–272.
5. The Alpha OEM web site, `http://www.digital.com/semiconductor/alpha/alpha.htm`
6. The IA-64 web site, `http://www.intel.com/eBusiness/products/ia64/index.htm`
7. Donald E. Knuth, *The Art of Computer Programming*, Volume 3, Addison-Wesley, 1973, pp. 7 and 573.
8. Paul C. Kocher, *Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems*, Lecture Notes in Computer Science, Advances in Cryptology, proceedings of CRYPTO'96, Springer-Verlag, 1996, pp. 104–113.
9. The DES Challenge web site, `http://www.distributed.net/des/`
10. Ross Anderson, Eli Biham and Lars Knudsen, *Serpent: A New Block Cipher Proposal*, Lecture Notes in Computer Science, Advances in Cryptology, proceedings of Fast Software Encryption 98, Springer-Verlag, 1998, pp. 222–238.
11. The SHA-1 specification, FIPS 180-1, `http://www.itl.nist.gov/fipspubs/fip180-1.htm`
12. Donald E. Knuth, *The Art of Computer Programming*, Volume 2, Addison-Wesley, (2^{nd} ed.), 1997, p. 139.
13. Donald E. Knuth, *The Art of Computer Programming*, Volume 1, Addison-Wesley, 1969, pp. 177–179.
14. Thomas Pornin, *Automatic Software Optimization of Block Ciphers using Bitslicing Techniques*, unpublished, `http://www.di.ens.fr/~pornin/bitslice.ps`
15. AES: best encryption timings for the submissions, `http://www.di.ens.fr/~granboul/recherche/AES/timings.html`
16. The Xilinx web site, `http://www.xilinx.com/`
17. The OpenBSD web site, `http://www.openbsd.org/`