# The CryptoGraphic Disk Driver

Roland C. Dowdeswell
*elric@imrryr.org*
The NetBSD Project

John Ioannidis
*ji@research.att.com*
AT&T Labs – Research

## Abstract

We present the design and implementation of CGD, the CryptoGraphic Disk driver. CGD is a pseudo-device driver that sits below the buffer cache, and provides an encrypted view of an underlying raw partition. It was designed with high performance and ease of use in mind. CGD is aimed at laptops, other single-user computers or removable storage, where protection from other concurrent users is not essential, but protection against loss or theft is important.

## 1 Introduction

The number of laptop users is increasing continuously, and with it the amount of sensitive data that resides outside traditional data protection mechanisms—physical (security guards) or electronic (firewalls). There have been several well-publicized cases of top executives having their laptops lost or stolen, with highly sensitive corporate data in them [3]. In fact, any potentially removable storage suffers from such threats, as the case of the missing disk drives from a government research lab around 2000 [1] demonstrates. Some of the cases involved the violation of doctor–patient privilege [10] which may expose doctors or hospitals to liability for disclosing confidential patient data. When computers are retired, the hard drives may contain sensitive data, complicating their disposal. Cryptography provides the ultimate protection in such cases, but while there exist many techniques and systems that employ cryptography to protect files, many of them are cumbersome to use, require expertise beyond that of the common user, or (unfortunately, in the case of many commercial systems) offer less security than advertised. We review some of these approaches in Section 4.

There are many existing solutions to this problem, such as CFS[4], TCFS[5], OpenBSD's vnd(4) encryption, FreeBSD's GEOM encryption, Linux's loopback encryption, Cryptfs[17] and NCryptfs[16]. CFS and TCFS operate over the file system layer and for that reason are well suited to encrypting files stored over a distributed file system such as NFS. CFS is implemented as a userland NFS server and gains much in the way of portability, but unfortunately gains all of the drawbacks of NFS in the process. The semantics of NFS even on the local machine make file locking less reliable, make caching of data blocks less effective and have a serious overall impact on performance. CFS also does not protect the filesystem metadata as it encrypts each file separately and does not encrypt file system meta-data. It may be possible in many scenarios to get much of the information that is desired simply by examining the directory layout. TCFS also uses NFS as its transport, but operates as an NFS client rather than a server. TCFS has many more features than CFS, but also shares the drawbacks of using NFS.

To fulfill the needs for secure file storage in a laptop environment, we built the CryptoGraphic Disk (CGD). CGD is a device driver that looks just like an ordinary disk drive, and encrypts an entire real disk partition. It is thus suited for single-user environments (mostly laptops) where protection from other users on the same machine is not essential, but protection against physical loss is paramount, as is high performance. CGD was written for and is available in NetBSD, but can be readily ported to any other BSD-derivative with very few changes.

This is a similar approach to that taken by OpenBSD's vnd(4) driver, FreeBSD's GEOM driver and Linux's loopback mounts. Each of these has a few drawbacks. OpenBSD's vnd(4) driver and FreeBSD's GEOM driver do not present a modular framework for defining encryption methods which we discuss in Section 2. We discuss some of the drawbacks of a vnd approach in Section 2.1. In Section 2.2, we discuss how OpenBSD's vnd(4) driver, FreeBSD's GEOM driver and Linux's loopback mounts do not have flexible key generation mechanisms, do not allow for n-factor authentication and do not adequately protect from dictionary attacks against the pass phrase.

## 2 Architecture and Implementation

In setting out to create a secure local storage facility for Unix workstations, the main design considerations were flexibility, modularity, performance and robustness. The flexibility consideration implied the virtual disk approach. Using the disk-driver interface as the chosen abstraction gives its potential users the freedom to use any kind of file system they want on top of it, be it FFS, swap space, or even space for a database application that writes to the raw partition.

It is important that the design be modular since breakthroughs in cryptanalysis are unpredictable and frequent. Being tied to a single cipher, a single method for choosing IVs or a single method for generating keys would force users to upgrade critical operating system components upon the discovery of weaknesses or vulnerabilities. Modularity also allows for use to be tailored to a specific threat model. Allowing multiple ciphers enables the user to perform cost-benefit analysis based on an evaluation of their risk. Allowing multiple key generation methods allows CGD to be used under different threat models. For example, unattended booting might be a requirement and in this case it would make sense to retrieve the keys from a GSS–API key server. N–factor authentication might be required and CGD provides for this.

The performance consideration led to the decision to place the cryptographic disk functionality below the buffer cache; that is, create a virtual disk driver that directly accesses the raw disk beneath it. This allows all kernel-level (or even user-level) software that makes assumptions about disk layout to work (whether these assumptions are justified is beyond the scope of this paper).

The robustness consideration led to the decision to place the most complex code in user land in the configuration utility. The user land configuration utility performs all key management, key retrieval, *etc.* This leaves the kernel code the easier task of encrypting blocks which makes the kernel code straightforward to audit.

The main disadvantage of the pseudo–disk approach is that there is no per-user keying or any other per-user cryptographic isolation. For better or worse, the security model of Unix is left unchanged. This, of course, is not a problem in the intended user-base for CGD, namely, laptop users or owners of removable storage in national nuclear research laboratories. In any case, if the operating system is not trusted to provide adequate protection, potential intruders can read keys or buffer blocks off */dev/kmem* anyway, so this is not as big a problem as it first appears.

The CGD code base consists of two parts: a kernel driver and user-level configuration software. We examine these two parts next.

## 2.1 The Kernel Driver

Once we decided to create a disk driver, we still had a number of design decisions to make. There are several places in the kernel where we could have implemented such functionality. One quick solution could have been to modify the *vnd(4)* driver. *Vnd* is a device driver that turns a Unix file into a block device. This approach is used in the OpenBSD. There are several reasons why this turns out not to be a good idea. Firstly, in *vnd* requests have to traverse the filesystem code twice. This increases the number of places where deadlocks can occur. In fact, we have run into such problems when trying to run a system entirely off *vnd* disks on top of *msdosfs* files. Secondly, because we are laying out an FFS (or some other) filesystem on top of a file which has already been laid out in an FFS filesystem, the upper filesystem will make block allocations based on erroneous assumptions of block locality; these destroy many of the advantages of the FFS layout algorithms. Experience with *vnd* actually indicates that this degradation is not that pronounced, but that may be an artifact of the actual workload. Lastly, adding crypto to *vnd* as a way of providing an encrypted disk runs into ease-of-use problems. To do so, a user would have to create a filesystem on the disk, allocate a single maximal-sized file, configure it with vnd and partition the resulting pseudo-disk. Attaching and mounting at runtime also add their share of complexity. Complexity for the user is unacceptable unless it delivers value; it is preferable to spend extra effort once if it results in easier use. Moreover, a user will immediately recognize that, for example, */dev/cgd0a* is an encrypted device, rather than have to remember whether they configured the */dev/vnd0a* with or without encryption. This last reason is why we did not release a null encryption transform.

CGD is a pseudo disk, in the same vein as the concatenated disk driver (*ccd(4)*) or RAIDframe (*raid(4)*). *Ccd* takes multiple partitions and presents them as a single disk by either concatenating them or interleaving their blocks (RAID 0). RAIDframe uses the same techniques to provide RAID 0, 1, 4, and 5.

A disk device presents both a block interface and a character interface. CGD does its interesting work in the *strategy()* call in the block interface and the *ioctl()* call in the character interface. The rest of the interface simply presents a normal disk/pseudo-disk in the same way as *ccd(4)*, *raid(4)* or *vnd(4)*. For further information about the block and character devices please refer to "The Design and Implementation of the 4.4BSD Operating System"[11]. For the purposes of this project, we implemented a generic set of functions that could be shared by all of the pseudo–disks to simplify the code.

The *ioctl()* function responds to all the normal disk *ioctl(2)s* and also presents two more: *CGDIOCSET* and *CGDIOCCLR*. The first *ioctl* will cause CGD to attach to an underlying disk device or partition and configure the required parameters (such as the key, the encryption algorithm, the key length, the IV method, *etc.*). The second *ioctl* will detach the CGD from the underlying disk and free the parameters.

CGD's *strategy()* is responsible for scheduling an I/O request. If the CGD is not configured then this call will result in an error. We are passed a *buf* structure which contains all the relevant information about the request we are about to perform. As with all *strategy()* routines, we perform basic bounds checking and index into the disklabel to calculate the real offset of the operation.

If we are reading then we allocate a new *buf* structure and populate it, setting its buffer to be the buffer that we have been passed.

If we are writing then we also allocate a new *buf* structure and populate it, but this time we also allocate memory for the transfer. We do this because the contents of the buffer cache must be stored in plain text and so if we encrypt in place then we would be forced to decrypt when the operation completes. This would double the CPU usage for writing. It is also not clear that this would not cause problems for processes which have *mmap(2)*ed the buffers on which we are working. OpenBSD's *vnd(4)* driver encrypts the data in place and decrypts it when the write completes.

We then pass the newly created *buf* structure to the underlying disk driver. When it completes its work, it will call back into CGD which will check for errors, decrypt the data in the read case and register its completion.

We do not modify the block size of the underlying device because that would violate the atomicity of single writes and the file system code relies on said atomicity to ensure that data can be recovered after a crash. It would also needlessly complicate the driver making it more difficult to audit. We also do not provide integrity checking since this would require the storage of hashes and would break the atomicity assumptions.

We define a modular framework for adding cryptographic algorithms. It is not enough to rely on an underlying framework such as the OpenBSD Crypto Framework because CGD is making additional decisions beyond simply choosing a cipher and a mode. Even with a simple CBC mode the IV needs to be chosen.

In Cipher Block Chaining (CBC) mode, we encrypt each disk block using a block cipher; a different IV is used for each block. We support three ciphers in the initial implementation: AES[6], 3DES[12, 15] and Blowfish[14]. We only support one IV generation method: the block number encrypted under the same key as the data; we do provide the ability in the code to define more if the need arises.

Each block is encrypted separately from any other block. To ward against structural analysis, we use a different IV for each block. In the initial implementation we use as an IV the block number encrypted under the same key as the one used for data. This provides the guarantee that each block has a different IV since the block cipher is a bijection. IVs for successive blocks thus come from plaintexts that differ by very few bits; this may lead to some rather obscure attacks, and we plan to add additional IV generation methods in the next release of CGD.

We decided against trying to use complicated schemes of generating multiple keys for different parts of the disk, as this would increase complexity (which rarely makes for better security) without any identifiable benefit. Moreover, we do not permute the layout of blocks on the disk because it would adversely affect performance, again without any identifiable security benefits.

## 2.2 The Userland Program

Configuration is performed from userland using the *cgdconfig(8)* utility. This utility performs all necessary functions including managing the configuration files, generating or fetching keys and configuring the device.

We define two types of configuration file: the main configuration file */etc/cgd/cgd.conf* and per-CGD "parameters files." The main configuration file lists all of the CGD devices that should be automatically configured and the mapping between the CGD and the real disk. The parameters files define per-CGD parameters such as the encryption algorithm, the key generation methods and the IV generation method.

The userland configuration utility needs to derive a key for the encryption. We support an extensible framework for defining mechanisms to derive the key. The parameters file contains a variable number of key generation stanzas. The derived key is the direct sum of the evaluation of all of the stanzas. This provides for n–factor authentication.

Currently we support four different key generation methods, namely *pkcs5_pbkdf2*, *gssapi_keyserver*, *randomkey*, and *storedkey*.

The *pkcs5_pbkdf2* method is an implementation of PKCS#5 PBKDF2[2]. If used, it will prompt the user for a passphrase which will then be used to derive the key. PKCS#5 PBKDF2 was chosen because it is well understood and can generate keys of different lengths safely. The use of PKCS#5 PBKDF2 addresses perhaps the most common weakness of otherwise well designed crypto–systems, namely dictionary attacks against the passphrase. Since the method uses chained HMACs and includes an iteration count, it is possible to configure enough iterations to make a dictionary attack arbitrarily prohibitive at a fixed configuration–time cost. The inclusion of a salt from the parameters file precludes an off line attack.

Since it is unlikely that the entropy contained in the pass phrases that users can remember will increase with Moore's Law, it is essential that the default iteration count is chosen with care. The *pkcs5_pbkdf2* algorithm is run infrequently, so it is acceptable that it takes a reasonably long time to derive a key from the pass phrase. We decided that one second was the appropriate amount of time in light of these considerations. When generating a parameters file, *cgdconfig(8)* will calibrate to the current hardware and choose an iteration count that will cause the *pkcs5_pbkdf2* algorithm to take one second to derive the key.

This should provide a reasonable level of protection for the foreseeable future. If we assume that the user's pass phrase contains $n$ bits of entropy against a dictionary attack, then it will take $2^n$ seconds to crack. If Moore's Law is that computer speed will double approximately every 18 months, then in $m$ years it will take $2^{n-2m/3}$ seconds to crack. For $n = 40$, we get that today it should take $2^{40}$ seconds $= 38865$ years. Although the work can be spread over many machines, this is still quite a formidable amount of time. When $m = 10$, however, it will take $2^{40-15}$ seconds $= 1.06$ years which is still a considerable amount of time, but within the capacity of an attacker with sufficient resources.

By contrast, many of the other cryptographic disks use a simple cryptographic hash (such as FreeBSD's GEOM driver) or no transform at all (such as OpenBSD's vnd(4) driver). This provides no protection against dictionary attacks.

The *gssapi_keyserver* method fetches the key from a keyserver using GSS–API to authenticate and protect the key. This method allows for unattended reboots.

The *randomkey* method reads the appropriate number of bits from */dev/random* and uses that as a key. This method is intended to be used in situations where persistence across reboots is not necessary. Examples would be a swap partition or perhaps a cache that contains sensitive data.

The *storedkey* method has the key in its stanza in the parameters file. It is used for two purposes. First if the parameters file is stored on separate media, such as a USB Mass Storage device, it can be part of an n-factor authentication scheme. Second, we use it to generate new parameters files that have different passphrases but still derive the same key. We shall discuss the latter point later in this section.

In the cases where the user must enter a passphrase, we need a mechanism to detect if the passphrase is entered incorrectly. If verification fails, then *cgdconfig(8)* will print a warning and prompt for the passphrase again. We define methods that use the information that is already present on the disk in its normal usage, such as a disklabel or a filesystem. By using information which is already present we provide no information to an attacker that they could not already intuit from other unencrypted information on the system. We considered storing a hash of the generated key, but this would allow the attacker to perform a dictionary attack against the passphrase even if only in possession of the parameters file. The verification methods that we have defined require that the attacker have both the parameters file and the encrypted disk before a dictionary attack against the passphrase can commence. The main importance of this is the n–factor authentication where the parameters file is stored on a removable storage device.

We define a framework for the verification methods and currently support three mechanisms: *none*, *disklabel* and *ffs*. The first performs no checking. The *disklabel* and *ffs* methods scan for a disklabel or FFS filesystem after configuration, respectively.

The userland utility also provides the ability to generate additional parameters files which generate the same key. To do this, it fully evaluates the original parameters file including asking for passphrases and retrieving the keys from a keyserver producing key $K_1$. It then generates a new parameters file (which may use different key generation methods) and evaluates it producing key $K_2$. It then appends a key generation stanza of method *storedkey* where the stored key is $K_1 \oplus K_2$. The new parameters file will generate the same key:

$$K_2 \oplus (K_1 \oplus K_2) = K_1$$

The ability to generate new parameters files allows the user to change their passphrase without rekeying the disk. It also allows multiple administrators to access the disk with different passphrases. An administrator could also configure one parameters file with *pkcs5_pbkdf2* and another with *gssapi_keyserver* to allow for either a fallback when the keyserver is down, or to allow quicker booting of a laptop in a trusted environment.

The standard NetBSD start up scripts will recognize the existence of the main configuration file and automatically run *cgdconfig(8)* at various points to configure the CGDs. It is necessary to split CGD configuration up in this manner because some of the key generation methods may require network access, such as *gssapi_keyserver*, whereas some of the disks may need to be configured before the network is brought up. We solve this with the aforementioned tags in the main configuration file.

Tightly integrating CGD support into the default NetBSD start up scripts provides system administrators with the ability to use CGD with a minimum of effort.

## 3 Evaluation

In our current implementation, the kernel code is comprised of the files *cgd.c*, *cgd_crypto.c*, *cgd_crypto.h*, *cgdvar.h*, *dksubr.c* and *dkvar.h* located in the directory *src/sys/dev/*. The files *dksubr.c* and *dkvar.c* were written during the course of this project, but are not strictly a part of CGD. They are common functionality which CGD shares with *ccd* and *raid*. Excluding these files, CGD's kernel code is 1348 lines including comments. Of this, the code which interfaces with the ciphers is 91 lines for Blowfish, 103 lines for 3DES and 102 lines for AES. This provides a good metric for how difficult it would be to interface to a new cipher.

The userland program is comprised of all the files in the directory *src/sbin/cgdconfig/*. *Cgdconfig* is 2763 lines of code including comments.

In the following, we analyse the performance of the cryptographic disk. Since the performance varies widely depending on the hardware in question, we analyse three systems: a DEC Personal Workstation 500a (alpha); a Pentium 4-based PC; and an IBM ThinkPad 600E (Pentium II).

Since CGD is designed as a pseudo-disk device, we concentrate our analysis on the raw disk to avoid the secondary effects of the file system code. Our initial speculation was that CGD would increase the latency of a disk transaction by a factor depending on the size of the transfer while using much CPU. The results on the Alpha bear this theory out, while the results on the i386 raise a few questions. We measured disk throughput by reading and writing 100 MBytes from the raw disk devices using different block sizes. The test is a single thread with each operation waiting for the previous operation to complete.

The ciphers had the same relative performance on the different platforms we tested. The only cipher which consistently had a significant detrimental impact on performance was 3DES. The other ciphers performed substantially better.

## 3.1 DEC Personal Workstation 500a

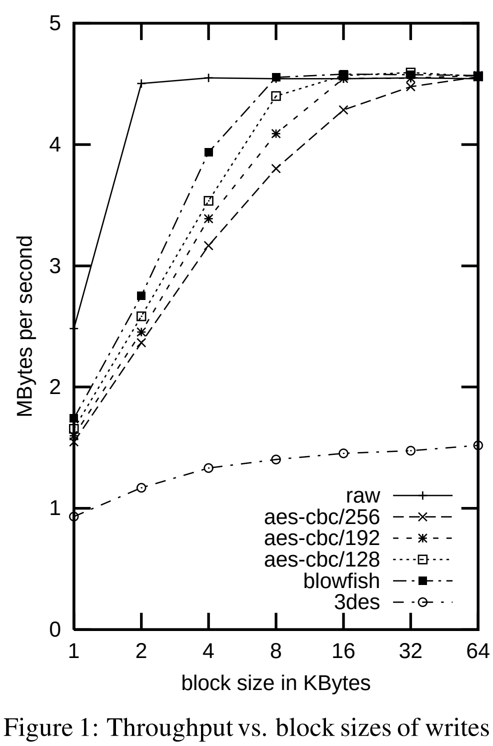


Figure 1: Throughput vs. block sizes of writes to the raw disk devices on PWS500a

These measurements were performed on a DEC Personal Workstation 500a (PWS 500a) running NetBSD 1.6I. The PWS 500a has a 500 MHz 21164a Alpha processor. The disk on which the tests were run is a 2GByte Ultra-Wide SCSI disk, a Seagate model ST32155W.

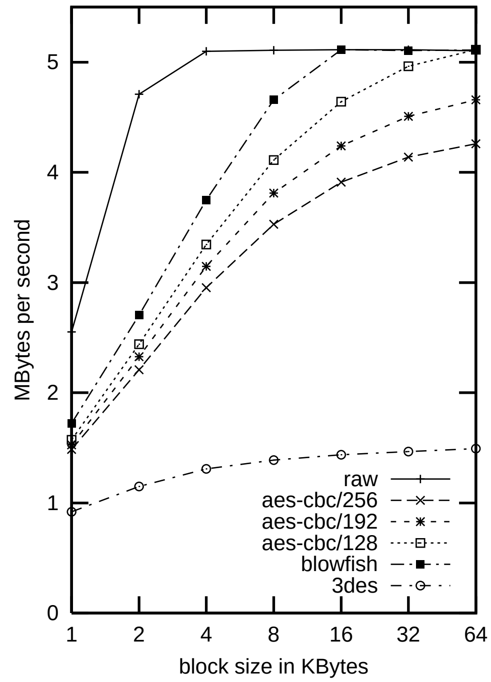


Figure 2: Throughput vs. block sizes of reads from the raw disk devices on PWS500a

We note in Figures 1 and 2 that at small block sizes performance is degraded quite substantially, whereas as the block size increases performance returns to a quite reasonable level. In the write case AES and Blowfish both quickly began to reach full disk throughput as the block size was increased. The read case proved to be almost as quick, although at the larger key sizes AES could not quite keep up. 3DES performance was poor, as we expected.

NetBSD on the Alpha does not have assembly optimized versions of any of the ciphers used, unlike the i386 architecture.

## 3.2 Pentium 4-based PC

These measurements were performed on a 1.7 GHz Pentium 4 system running NetBSD 1.6Q. The pseudo-disks were constructed on a 19 GByte Ultra100 IDE disk, a WDC WD200BB-00CXA0.

We note from Figures 3 and 4 that results are substantially more complex than they were on the Alpha.

For writing, CGD keeps up at the low block sizes and performance falls off and then starts catching back up. With a 64 KByte block size, Blowfish attains $85\%$ of the raw disk throughput and 128-bit AES attains $73\%$.

The read performance of the raw disk experiences a large jump between the block sizes of 16KB and 32KB,

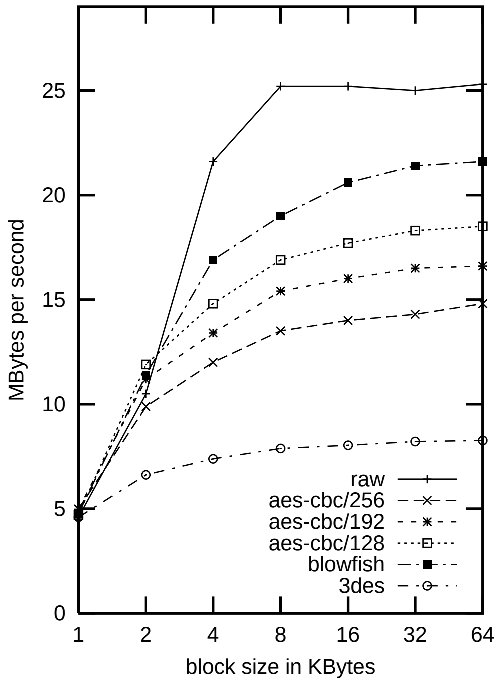


Figure 3: Throughput vs. block sizes of writes to the raw disk devices on P4 class machine

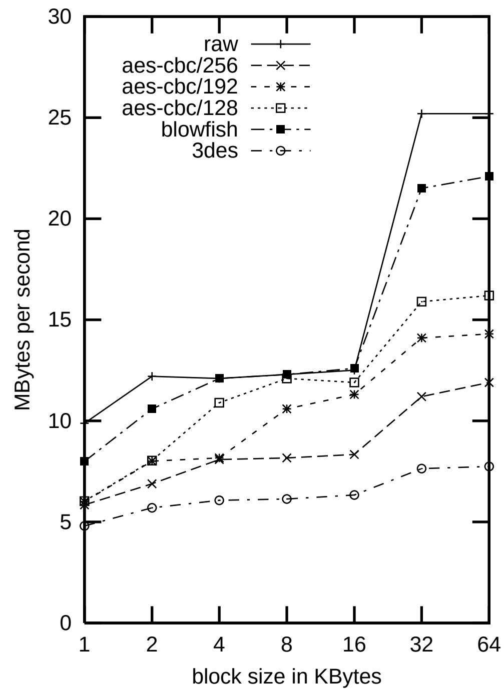


Figure 4: Throughput vs. block sizes of reads from the raw disk devices on P4 class machine

but CGD still maintains the level of performance that one would expect. Blowfish almost keeps up with the raw disk for all of the block sizes and at 64 KByte attains $88\%$ of the raw disk throughput. 128-bit AES attains $64\%$ of the raw disk throughput.

### 3.3 IBM ThinkPad 600E

These measurements were performed on a ThinkPad 600E running NetBSD 1.6Q. The ThinkPad has a Mobile Pentium II processor running at 400 MHz. The pseudo-disks were constructed on a 10 GByte Ultra33 IDE disk, a TravelStar.

The results are much simpler in this case as we note from Figures 5 and 6.

The read and write cases are quite similar. For Blowfish at 64 KByte block sizes, writing achieves $84\%$ of the raw disk throughput and reading achieves $85\%$. 128-bit AES at 64 KByte block sizes achieves $64\%$ and $58\%$ for writing and reading, respectively.

## 4 Related Work

The Cryptographic File System (CFS)[4] is perhaps the most widely used cryptographic file system. The abstraction it presents is encrypted file contents as well as encrypted file names. It runs as a user-level NFS daemon listening to NFS requests on the loopback interface (so that only requests from the same machine will be honored). This user-level daemon, *cfsd*, provides virtual directories under */crypt*. An encrypted directory hierarchy to its corresponding cleartext hierarchy, visible under */crypt/*username using the *cattach* utility. Keying is done on a per-user basis; the encrypted hierarchy is initialized with a key, given as a passphrase, which has to be passed to *cattach* when the latter is invoked to attach the encrypted directory. While, of course, only *root* can run the initial *mount* command that provides the */crypt* hierarchy, *cfsd* can be run as an unprivileged server, in which case only that user's files are available. If it is run as *root*, it enforces unix-domain credentials (userid and groupid), so that only processes running under the same userid can access the CFS directories. CFS is extremely useful when a user's home directory lives on an NFS server, and the user wants to take advantage of system-wide file availability and backup services, but still wants more privacy than the NFS primitives allow.

The Microsoft Windows 2000 and Windows XP NTFS file system supports a similar notion of encrypted files and directories. Any subdirectory (or just a leaf file) can be declared encrypted, but only the file contents (and not the file- or directory name) are encrypted. However, the feature is better integrated in the filesystem layout, in that the encrypted files do not have to reside under a special top-level directory.

*Vnd* under OpenBSD supports encryption; each block of the underlying file is just encrypted using a symmetric cipher, as we have already explained in Section 2. There are numerous other examples of loopback disk drivers, such as *ccd(4)* and *vinum*[9], any of which could

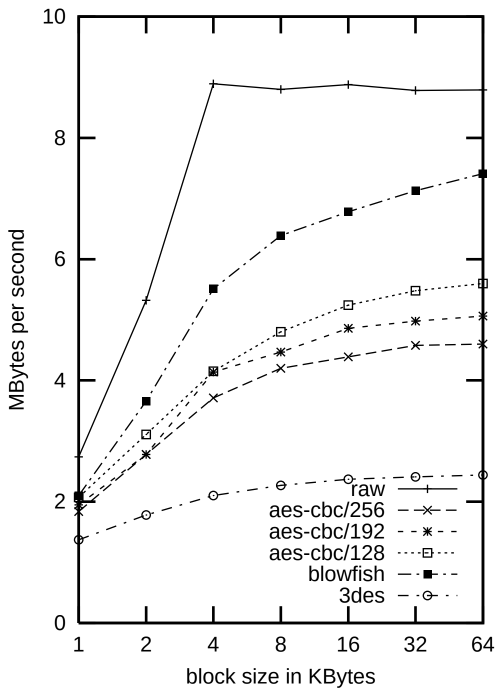


Figure 5: Throughput vs. block sizes of writes to the raw disk devices on ThinkPad 600E

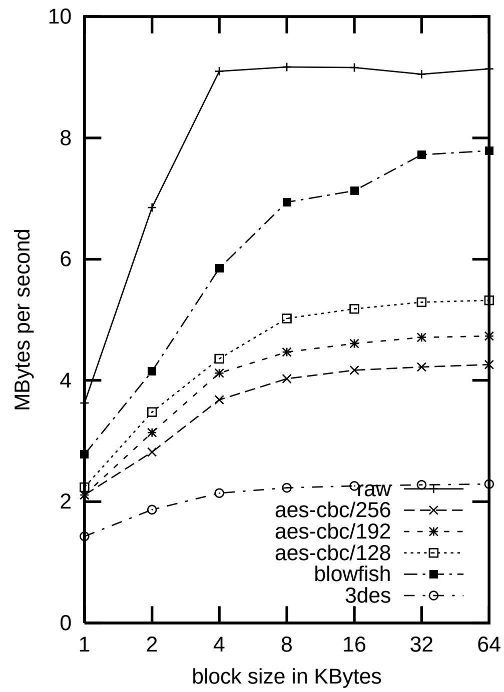


Figure 6: Throughput vs. block sizes of reads from the raw disk devices on ThinkPad 600E

potentially be modified to support encryption, in addition to their primary functionality. Linux loopback devices support encryption either to a file or to a disk partition. BestCrypt[8] is a commercially available loopback encryption device. FreeBSD's GEOM layer supports encrypting a partition. None of these use PKCS#5 PBKDF2 or any other iterated salted key generation method and so do not offer adequate protection from dictionary attacks against the pass phrase.

Another driver that closely resembles CGD is the encrypting swap device[13] on OpenBSD. In order to protect against keys or other sensitive information lingering in the swap space long after their corresponding processes have terminated, the swap device itself is encrypted. Their keys are chosen randomly, and change frequently, thereby invalidating old data.

Half-way between the loopback drivers and the NFS approaches lie encrypting file systems implemented as a vnode layer. Stackable vnodes were introduced by Rosenthal [7], and were used to implement among other things two encrypting file systems Cryptfs [17, 18] and NCryptfs [16]. This approach offers substantially higher performance than the NFS approaches, but is less portable.

## 5 Conclusions and Future Work

Many computers fall into the category of "trusted administrator, low physical security" and for these computers an encrypted disk provides adequate protection against typical threats. We have designed and implemented an encrypted disk, taking care to avoid some of the most common errors that have been made in other systems by ensuring that it is easy to configure and the cryptography is designed according to best practices.

We have demonstrated that the performance of our cryptographic disk is acceptable, although the exact performance is quite dependent on the hardware configuration. Blowfish is quite fast, suffering only about a $15\%$ performance hit on the platforms we tested and AES offers performance that is still acceptable.

CGD is in use today, both in the laptop of one of the authors and in the laptops of many NetBSD users. One user even reported that his use of CGD has already kept the contents of his laptop from the prying eyes of one of his client's employees who decided to use a Linux boot floppy to examine his laptop while he was taking his lunch break.

Because CGD defines extensible frameworks for many aspects of its configuration, there is much room for future growth. Currently CGD supports three ciphers and one IV generation method. We may add additional ciphers if there is demand. We plan on adding additional IV generation methods.

There is much room for future work on CGD. Additional key generation methods could be defined. CGD could be modified to use the *crypto(9)* API which was not available in NetBSD when CGD was written.

The software is freely available as part of the NetBSD operating system. The NetBSD source tree can be accessed online via *http://www.NetBSD.org/*.

## Acknowledgments

We wish to thank the anonymous USENIX reviewers, our shepherd Robert Watson and Erez Zadok for reviewing the paper and providing helpful comments.

We wish to thank Lee Nussbaum for offering valuable comments during the development of CGD. We also thank Love Hörnquest Åstrand, Thor L. Simon and Jason R. Thorpe for reviewing the code. We would also like to thank Chris G. Demetriou for his tolerance.

## References

[1] Amy Paulson. Senate hearing examines loss of nuclear secrets at Los Alamos lab. *http://www.cnn.com/2000/ALLPOLITICS/stories/06/14/losalamos.hearing/*, June 2000.

[2] B. Kaliski. PKCS #5: Password-Based Cryptography Specification Version 2.0. RFC 2898, *http://www.ietf.org/rfc/rfc2898.txt*, September 2000.

[3] Betsy Schiffman. Stolen Qualcomm Laptop Contains Sensitive Data. *http://www.forbes.com/2000/09/19/mu5.html*, September 2000.

[4] M. Blaze. A Cryptographic File System for Unix. In *Proc. of the 1st ACM Conference on Computer and Communications Security*, November 1993.

[5] Giuseppe Cattaneo, Luigi Catuogno, Aniello Del Sorbo, and Pino Persiano. The Design and Implementation of a Transparent Cryptographic File System for UNIX. In *Proceedings of the Annual USENIX Technical Conference*, June 2001.

[6] Joaen Daemen and Vincent Rijmen. AES Proposal: Rijndael. *http://www.esat.kuleuven.ac.be/~rijmen/rijndael/*, June 1998.

[7] David S.H. Rosenthal. Evolving the Vnode Interface. In *Proceedings of the Annual USENIX Technical Conference*, June 1999.

[8] Jetico, Inc. BestCrypt software home page. *http://www.jetico.com/*, 2002.

[9] Greg Lehey. The Vinum Volume Manager. In *Proceedings of the Annual USENIX Technical Conference, FREENIX Track*, June 1999.

[10] John Leyden. For sale: memory stick plus cancer patient records. *http://www.theregister.co.uk/content/55/29752.html*, March 2003.

[11] Marshall Kirk McKusick, Keith Bostic, Michael J. Karels, John S. Quaterman. *The Design and Implementation of the 4.4BSD Operating System*, pages 196–204. Addison-Wesley Publishing Company, 1996.

[12] National Bureau of Standards. Data Encryption Standard, January 1977. FIPS-46.

[13] Niels Provos. Encrypting Virtual Memory. In *Proceedings of the 10th Usenix Security Symposium*, August 2000.

[14] Bruce Schneier. Description of a new variable–length key, 64-bit block cipher (blowfish). In *Fast Software Encryption, Cambridge Security Workshop Proceedings*. Springer-Verlag, December 1993.

[15] Bruce Schnier. *Applied Cryptography*, pages 265–301. John Wiley and Sons, second edition, October 1995.

[16] C. P. Wright, M. Martino, and E. Zadok. NCryptfs: A Secure and Convenient Cryptographic File System. In *Proceedings of the Annual USENIX Technical Conference*, June 2003.

[17] Erez Zadok, Ion Badulescu, and Alex Shender. Cryptfs: A stackable vnode level encryption file system. Technical Report CUCS-021-98, Computer Science Department, Columbia University, June 1998.

[18] Erez Zadok, Ion Badulescu, and Alex Shender. Extending File Systems Using Stackable Templates. In *Proceedings of the Annual USENIX Technical Conference*, June 1999.