LAPPEENRANTA UNIVERSITY OF TECHNOLOGY

DEPARTMENT OF INFORMATION TECHNOLOGY

LABORATORY OF COMMUNICATIONS ENGINEERING

Transparent and secure remote network storage system using an untrusted server

The subject has been approved by the council of the Department of Information Technology on 15.11.2006.

Examiners: D.Sc. Jouni Ikonen and D.Sc. Pekka Jäppinen

Instructor: D.Sc. Pekka Jäppinen.

Lappeenranta 30.11.2006

Mika Boström

Myllypellonkatu 2

53850 Lappeenranta

Mika.Bostrom@iki.fi

Tiivistelmä

Tekijä: Boström, Mika

Nimi: **Transparent and secure remote network storage system using an untrusted server**

Osasto: Tietotekniikan osasto

Vuosi: 2006

Paikka: Lappeenranta

Diplomityö. Lappeenrannan teknillinen korkeakoulu. 70 sivua ja 9 kuvaa.

Tarkastajat: TkT Jouni Ikonen, TkT Pekka Jäppinen

Ohjaaja: TkT Pekka Jäppinen

Hakusanat: security, encryption, iSCSI, off-site, remote, backup, untrusted, insecure, network block-device

Työssä kehitettin läpinäkyvä Internet Small Computer Systems Interface-verkkolevyä (iSCSI) käyttävä varmistusjärjestelmä. Verkkolevyn sisältö suojattiin asiakaspään salauskerroksella (dm-crypt). Järjestely mahdollisti sen, että verkkolevylle tallennetut varmuuskopiot pysyivät luottamuksellisina, vaikka levypalvelinta tarjoava taho oli joko epäluotettava tai suorastaan vihamielinen. Järjestelmän hyötykäyttöä varten kehitettiin helppokäyttöinen prototyyppisovellus.

Järjestelmän riskit ja haavoittuvuudet käytiin läpi ja analysoitiin. Järjestelmälle tehtiin myös karkea kryptoanalyysi sen teknisten ominaisuuksien pohjalta. Suorituskykymittaukset tehtiin sekä salatulle että salaamattomalle iSCSI-liikenteelle. Näistä todettiin, että salauksen vaikutus suorituskykyyn oli häviävän pieni jopa 100 megabittiä sekunnissa siirtävillä verkkonopeuksilla. Lisäksi pohdittiin teknologian muita sovelluskohteita ja tulevia tutkimusalueita.

Abstract

Author: Boström, Mika

Subject: **Transparent and secure remote network storage system using an untrusted server**

Department: Information technology

Year: 2006

Place: Lappeenranta

Master's thesis. Lappeenranta University of Technology. 70 pages and 9 figures.

Examiners: D.Sc. (tech) Jouni Ikonen, D.Sc (tech) Pekka Jäppinen

Instructor: D.Sc. (tech) Pekka Jäppinen

Keywords: security, encryption, iSCSI, off-site, remote, backup, untrusted, insecure, network block-device

A transparent and remote backup system was built on top of Internet Small Computer Systems Interface (iSCSI), and secured by using a local cryptographic layer (dm-crypt) on top of the network block-device. The construction allowed to store backups safely on untrusted or even outright hostile servers. An easy-to-use prototype implementation was written and tested to work.

Risks and potential weaknesses in the system were assessed and evaluated. Also, a rudimentary, non-mathematical cryptanalysis was performed based on the technical aspects of the setup. Performance of the construction was measured with and without encryption, and it was found that the performance impact from encryption was negligible even with the network running at 100Mbit/s. Other possible uses and further research avenues for the technology were also considered.

Foreword

This work was done at the Lappeenranta University of Technology's Laboratory of Communications Engineering, for the university's Department of Information Technology.

This was a personal project, inspired by a friend in need. That I had an opportunity to do something apparently novel was a definite bonus, and due to personal interest, not once during the project did it feel like work. I have been lucky.

I want to offer my sincere thanks to the parties who made writing this thesis possible:

- Mr. Ari Flinkman, who first postulated the problem which my thesis proposes to solve;
- The Department of Information Technology, who provided me with the resources I needed; and
- My wife Henna, who kept on nudging me before and during the project

Contents

Acronyms and abbreviations

AES	Advanced Encryption Standard
ATA	Advanced Technology Attachment
CBC	Cipher-Block Chaining
CDB	[SCSI] Command Descriptor Block
cgd	Cryptographic device driver
CIFS	Common Internet File System
CRC	Cyclic Redundancy Check
dm	Device Mapper
dm-crypt	Device Mapper with cryptographic support
ESSIV	Encrypted Salt-Sector IV
FIPS	Federal Information Processing Standards
gbde	GEOM[etry] Based Disk Encryption
GPG	Gnu Privacy Guard
iSCSI	Internet Small Computer Systems Interface
IV	Initialisation Vector
IP	Internet Protocol
LBA	Logical Block Addressing
LVM	Logical Volume Management
MITM	Man-in-the-middle [attack]
OS	Operating System
PDA	Personal Digital Assistant
PDU	Protocol Data Unit
PTD	Personal Trusted Device

SCSI	Small Computer Systems Interface
SHA1	Secure Hash Algorithm 1
SHA-256	Secure Hash Algorithm, 256-bit digest variant
SSH	Secure Shell
SFTP	SSH File Transfer Protocol
RAID	Redundant Array of Inexpensive Disks
TCP	Transmission Control Protocol
TCP/IP	Common abbreviation for TCP, explicitly meaning that TCP runs on top of IP
UI	User Interface
UUID	Universally Unique Identifier
VPN	Virtual Private Network

1 Introduction

Backups are an integral part of any organisation's day-to-day life. To be of any use, the backups must meet a list of three separate criteria:

1. They must be up-to-date and correct;
2. They must be available and possible to use with reasonable amount of work; and
3. At least one copy must be stored off-site

The first two requirements are just common sense. The third one is, first and foremost, a contingency measure.

In case of a fire, flood or other natural calamity, on-site backups are almost surely destroyed along with the premises and equipment. Proper insurance can and will cover for the physical damages, but very little can be done about lost information. As such, proper off-site backups allow to continue the operation of a business after only a relatively short delay. That is, the time it takes to set up new equipment in temporary premises. However, there is one aspect that needs to be considered. Backups, by their very nature, contain all, or at least most of a company's sensitive and business-critical data in one package. Hence they must be secured to a level that exceeds the security of regular business premises. That way the backups will be a harder target, and therefore not the attacker's first choice.

For off-site storage, backup-management introduces some new issues. The only reasonable way to maintain current backups is to automate the process. This works very well when the backups are stored locally. However, the off-site backups need to be transferred to remote location. A physical media must be transferred by hand. As this involves people, it is by default error prone. It is also very cumbersome, and makes recovery rather more convoluted a process than is necessary.

A natural instinct is, of course, to transfer the backups over a network connection. This allows to standardise the backup process, but requires to somehow secure the backups against unauthorised access, not only during the transfer, but also in storage. Straightforward and logical solution is to encrypt the backups before transferring them. That way the backup storage can be under anyone's control, and the only way to recover the data

within is by having access to the encryption key used. This, however, adds some overhead to backup-restoration process as the backups now require an additional decryption step before being usable.

Transferring backups over the network underlines two crucial aspects. Large backups require plenty of bandwidth, and storing the backups over a longer period of time can easily take up more storage space than the backed up data consumes by itself. This holds true even when the backups are compressed. To mitigate the storage space problem, backups are usually taken in combination of two different ways. First, there are periodic backup runs that take complete snapshots of the storage area contents. These are called *full backups.* Secondly, there are more frequent (usually daily) backup runs that compare the current storage area contents to a stored snapshot and then create a backup that contains only the changes since the snapshot. These are called *incremental backups*. A good backup practise combines both.

The question of how to make the off-site approach as easy to use – and as reliable as possible – has prompted several answers over the years. However, none of them address the issue of providing a solution regardless of the Operating System used, and few try to solve the issue of transparency. In this thesis I propose a solution that is usable on any modern Operating System, works automatically, provides very high level of transparency for backup restoration, and finally – does this all securely, even taking into account that the service provider may be untrusted.

My solution, in all simplicity, is to use a network block device with Internet Small Systems Computer Interface (iSCSI) and combine that with partition-level encryption. The latter is normally only used for local hard drive partitions to secure sensitive data from thieves and intruders.

This construction has an interesting property. By assuming that the server is untrusted or even outright hostile, the burden of trust is never imposed upon the iSCSI operator. The only requirement is that the service provider is reliable, and has financial interest to keep the service operational. The separation of trust and reliability is a crucial issue, especially in our world of increasing security problems, converging services and merging companies. We can not know nor anticipate when any one of our services is acquired by a competing party – but regardless of the owner, we still need to use the service.

2 Related work

The idea of doing backups over a network is certainly not new. It is also commonly expected that the server the backups are stored on is hostile, insecure or even compromised. As such, the basic problem of doing remote backups in secure manner has been solved several times over. The existing solutions can be split into two distinct categories; those that apply specialised software solutions, [1, 2, 3, 4, 5] and those that combine several generally available software packages to achieve similar results [6, 7, 8, 9].

Using networked storage is likewise a very old idea. The almost ubiquitous Network File System, NFS [10], was introduced by Sun Microsystems as early as 1984 and formally specified in 1989 [11]. In an equally archaic fashion, the basis for Windows's shared file servers, Server Message Block, SMB [12], was formally specified in 1988. The current version of SMB is known as CIFS [13].

2.1 Specialised client

These solutions are all based on the same idea: the company rents space on their servers. To access and use this space, a separate client is provided. The client then manages the details of encryption and decryption so that the backup data is encrypted before it leaves the local network. In order to ensure the security of the backups, the encryption key or keys are kept tightly with the client software and never sent to the server. Even if the service was compromised, no-one but the holder of the client's key could read the contents.

Client holds the logic for dealing with the stored backups and restoring them when necessary. Since the connection, backup and restoration logic is in one place only, backup creation and restoration are easily done in relation to the current local state. This is a definite advantage.

On the downside, using a dedicated client creates a single point of failure and any errors during server or client software can render the service inaccessible, inoperable, or both. The clients are primarily written for one OS only. Exceptions to this rule are rare, but they do exist [5]. For heterogeneous networks this is a serious issue. Even when the clients are available for multiple OS's, they are invariably proprietary and closed source. In addition to imposing an artificial barrier for unsupported architectures or OS's, the practice makes

it very difficult to validate the client's and the entire backup system's security.

Apart from technical limitations, there are certain external factors which need to be considered. The company offering the service may, due to any number of reasons, discontinue their service or even their entire operation. Migrating the backups to a new service provider and their system requires to assess and choose between new, and hence different solutions. Due to these factors, solutions based on specialised clients may be considered somewhat disquieting.

2.2 Combinations of existing, unrelated software

These solutions are not so much designed as they are engineered to employ existing software solutions for backing up, transferring or securing data [7, 6, 8, 9].

The idea and sequence of operation is always the same:

1. Backups are created as local archives or modifications, with tools specifically written for this purpose [14, 15]
2. Created files are encrypted with a standard tool, most often GPG [16], with such options that only holders of certain public keys can decrypt the package
3. The encrypted backup archive is transferred over an SSH [17] or SFTP [18] connection to the remote location for storage

The steps are done in order with custom scripts or by software especially written for this purpose, such as Duplicity [9]. The apparent simplicity of the setup is, unfortunately, misleading. Creating and encrypting the backups is a straightforward process, which is trivially automatable. But to support unattended SSH or SFTP transfers, a rather complex setup [19] is required. SSH authentication is done with a dedicated public key, which must be separate from the user's public key. Modern SSH implementations allow to restrict connections authenticated with certain public keys, and limit the commands or operations that are allowed on the server's side. In order to automatically transfer the backup, the private key must be unencrypted, that is, it must not have any passphrase to protect it. The separate key pair is therefore required for automation, as well as ensuring that access to the unprotected key does not yield access to the remote server for other than intended operations.

It is apparent that the ”secure backup” part is comparatively easy to accomplish. The ”remote” part is where the trouble and complexity introduce themselves. Complexity is fertile breeding-ground for human errors and, being quite difficult, the arrangement may require professional help during set up. Because the transfer part is independent from other components, network issues can trigger unexpected problems. These can vary from prolonged transfer times to potentially lost backups. Covering all the corner cases adds even further complexity and as such I consider these systems fragile.

Additionally, all of these solutions depend on using SSH accessible servers for rather unorthodox purposes. Servers with SSH access are generally multi-user systems, with each user having only a very limited amount of storage space, or *quota*, available. Such systems are not in general designed for individual users to store potentially hundreds of megabytes of their critical data. Although very elegant feats of engineering, these backup solutions basically extend existing setups beyond their intended use, in what can only be described as ad-hoc manner. Relying on such constructions for long term may be ill-advised.

My final argument against using these kinds of backup solutions is the imbued complexity and difficulty when restoring these backups in case of an emergency. Having both the transfer and encryption mechanism as separate components makes the already tedious process of backup restoration even more stressing.

2.3 Network storage

A cleaner solution for storing off-site backups would be to use networked storage systems. Why not just take and secure the backups, and then copy them to a path that physically resides on the remote, off-site server? Even though this adds up to offloading the transfer to yet another subsystem, at least it would be one that has been designed for transparently moving files around. The answer is quite simple: commonly available and publicly accessible network storage services are not available.

Some globally accessible storage services are (as of September 2006) starting to emerge. America On-Line’s XDrive [20] is the first and set to launch during September 2006; two competing large-scale services will be Google’s GDrive [21] and Microsoft’s LiveDrive [22]. All three are highlighted in [23]. The full details, including use scenarios, of these services are at the time of writing still unclear. Nonetheless it appears that they are all

designed for general storage, not for transparency and certainly not for privacy.

Network storage can be divided into two distinct categories. There are network filesystems, which expose their filesystem contents and directory structure over a network connection. There are also network block devices, which expose underlying storage hardware directly. In both of these, a major advantage is that the networked storage appears as a local resource. For network filesystems, the resource is just a separate directory hierarchy. For network block devices, the resource is a locally visible storage device.

As described above, network block devices work on a different premise from network file systems. The idea is to export just raw, unfettered storage space which is usable as any other storage media. Network block devices appear as local storage and are logically identical to directly attached hard drives. As such, they do not impose any arbitrary, filesystem-specific limitations. On the other hand, they do not offer any filesystem-specific services either. It is up to the user of the service to create the filesystem of their choice on the device.

The available network block device technologies at the moment are Fibre Channel (FC) [24], Internet Fibre Channel Storage Networking (iFCP) [25], Fibre Channel over TCP/IP (FCIP) [26] and Internet Small Computer Systems Interface (iSCSI) [27]. Fibre Channel is commonly tied to a specific kind of hardware, which is rather expensive. iSCSI on the other hand is available as both hardware and software, so it is possible to offer iSCSI storage with either dedicated hardware or a custom-built software solution. There is also a software-only implementation, Network Block Device (nbd) [28], but it seems to have been superseded by iSCSI. I have intentionally omitted the latest solution, ATA over Ethernet [29], as it works only in local networks, and as such competes against oldest Fibre Channel solutions. It does not work over IP networks.

2.4 Partition-level encryption

Partition-level encryption is used to protect entire drives or partitions, not just individual files. Effectively this means that data on the partition is readable only with a correct encryption setup.

On a Linux system, one can use dm-crypt [30] from the kernel itself, or any of the other available drive encryption solutions [31]. On a FreeBSD system the proposed solution

is gbde [32], and on a NetBSD system, cgd [33]. On an OpenBSD system, a virtual filesystem driver, vnd [34, 35], is required on top of normal partition access. In contrast, Apple OS X and Microsoft Windows do not natively support partition-level encryption, so they require third-party software for this purpose [31, 36].

3 Technology and used solution for encrypted backup storage

My proposed solution is to extend the idea of remote network storage. Instead of using an exported filesystem, I use a network block device with Internet Small Computer Systems Interface, or iSCSI for short. This way the storage appears as a local device, not just a locally visible resource. Hence, we can employ any security measures available for securing local storage or media.

By using iSCSI, we gain a definite advantage due to several factors.

1. The system becomes Operating System agnostic: any OS that can use iSCSI, can also employ whatever partition-level encryption mechanism available for them.
2. The choice of filesystem becomes almost a non-issue. Whatever the filesystem is, it resides on top of the encryption layer, and as such everything written to the partition is scrambled. This includes the filesystem metadata. However, there is one caveat from a security standpoint: use of a journaling filesystem exposes the setup to easier traffic analysis and allows known-plaintext based cryptanalytic attacks. (See Sections 5.1 and 5.2.)
3. In comparison with existing non-commercial solutions, the server is simpler to build and secure [9, 7, 6, 8]. There is no need to offer SSH accounts, or even have a public SSH service running at all. An iSCSI server is, for all intents and purposes, a networked appliance, and as such designed to provide a single, well-defined service. This in turn means that users' access control is greatly simplified.
4. The storage server is not limited to expensive Fibre Channel hardware. Instead, we can utilise any iSCSI target hardware, or even exploit common off-the-shelf hardware running modern Linux system and acting as an iSCSI target.

To better understand how iSCSI and dm-crypt work, the following sections describe all the elements individually, and finally in conjunction with one another.

3.1 Local hard drive

This is by far the most common use of a hard drive. A local, physically attached drive is used to store data. All read and write operations go directly[1] from the OS to the drive. Data is stored and retrieved in as straight-forward manner as possible. Figure 1 illustrates the use of a physically attached drive.

Figure 1: Normal read-write operations

Directly attached storage quite naturally has unmatched performance. Reading-speed is limited only by the speeds of the drive itself or the bus it is connected to, whichever is less. Although the filesystem adds a small delay on top of raw disk access, the difference should be negligible.

[1] An oversimplification, which deliberately omits such details as the filesystem, cache and storage bus

3.2 Local hard drive with partition-level encryption

To encrypt whole partitions, one needs to add a cryptographic translation layer just before disk operations. Figure 2 illustrates how such a method is used on a modern Linux system. All reads and writes go through a cryptographic layer, and as such all the data stored on the hard drive is encrypted. When data is read from an encrypted partition, the cryptographic layer intervenes and applies decryption operations on the data, before passing the result to the OS. In an equal fashion, when data is written to the encrypted partition, the cryptographic layer applies encryption operations on the data, before passing it on to the actual storage media, which finally stores the data. For clarity, dm-crypt is a Linux-specific method. Other OS's have semantically identical, but differently named approaches, as shown in Section 2.4.

Some filesystems allow to select parts of their directory hierarchy as automatically encrypted storage area. Any files and subdirectories stored in such areas are automatically and individually encrypted. Depending on the implementation, the actual directory structure may still be visible, even if the files' names and contents are encrypted. Such constructions are known as *natively encrypted filesystems*. Compared to these filesystems, partition-level encryption works on a lower level. The content of the entire partition is secured with encryption, as opposed to just the contents of the filesystem. This effectively means that there is absolutely nothing readable on the partition. Even filesystem metadata, such as listing of files, their sizes and modification dates are all protected by the encryption.

The presence of partition-level encryption can only be guessed at, and even then only by assuming that the illegible binary garbage is stored there on purpose. To access the protected partition, the translation layer must be told what encryption parameters are used. The required parameters are the algorithm, the key and the used encryption mode. The layer does not validate the parameters, it just blindly uses them. Hence, with incorrect parameters the partition appears to contain only binary garbage, albeit different garbage than what is actually stored on the disk.

Performance-wise, at least for the day, this kind of setup is obviously limited by the speed of cryptographic operations. No matter what the encryption operations are, or how fast they are performed, they always impose a slight delay. And at least for now, hard drive speeds are much greater than encryption speeds. As such, the bottle-neck is the CPU and its processing power. This may change at some point, as future processors are likely to

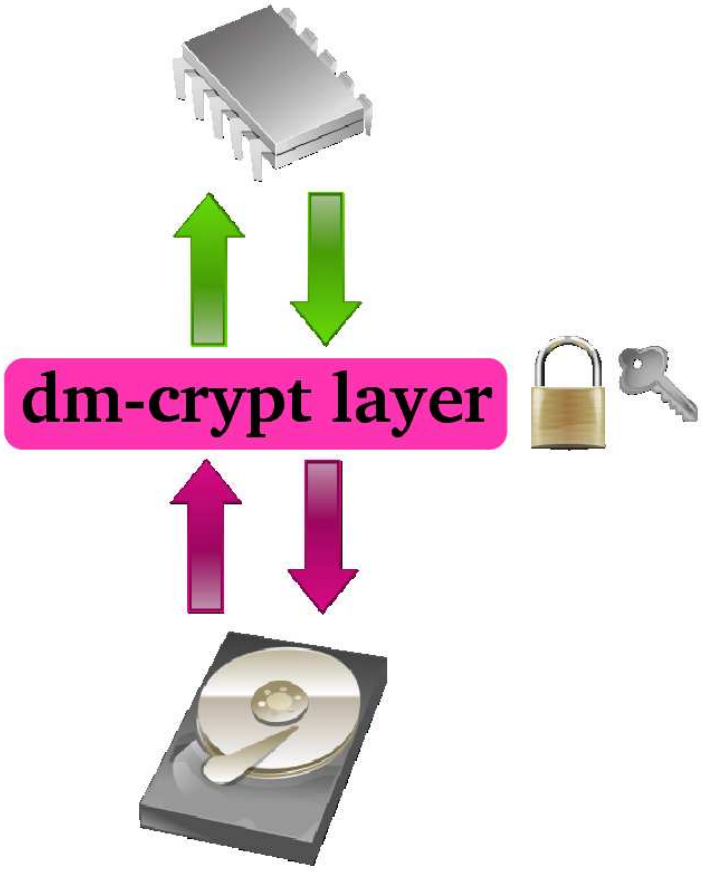

Figure 2: Read and write operations through dm-crypt translation layer. Purple arrows indicate encrypted content.

include special hardware accelerators for most common encryption algorithms. The first steps in this trend were taken by VIA with their PadLock technology [37], which implements dedicated hardware acceleration for AES [38] encryption algorithm inside their processors. AES stands for Advanced Encryption Standard, and is the current standard encryption algorithm for securing sensitive data. A more detailed description of AES is in Section 3.5.

As already pointed out, dm-crypt is a Linux-specific method to achieve partition-level encryption. Yet its mode of operation is straightforward enough to work as a logical example for all such setups. The following paragraphs describe how dm-crypt is used, and what practical steps are required when setting up the cryptographic layer.

How dm-crypt works

Dm-crypt is a cryptographic translation layer operating at block-device level. It functions as a regular block device and manipulates the payload of read and write commands. In practice this means that all the data written to the device is encrypted, while the write commands themselves are not touched. The same holds true for reading. If there is a way to monitor read and write operations, then locations and approximate sizes of files are known. This approach was used in traffic analysis and is further explained in Section 5.1.

Dm-crypt is used through a *mapping*. Dm-crypt mapping is a construction which encloses both the underlying physical device and the cryptographic translation layer, but to a user appears as a regular, unmodified device. When this device is accessed through mapping, the dm-crypt layer applies a cryptographic transformation to the command payload, and then passes the command to the real block-device. For a user or process this operation is completely transparent.

An example of how to create this mapping on Linux system illustrates the logical steps of the operation. The mapping itself is created with the command `cryptsetup`. In addition to the operation mode, real block-device and target name, the command accepts options that specify the cryptographic parameters used for the translation layer. A created mapping appears as a new block-device `/dev/mapper/name`.

To create a filesystem on the encrypted partition, the standard `mkfs` command is used on the mapping. Likewise, mounting an encrypted partition is no different from mounting a regular partition. In the following example a dm-crypt mapping with the name ”example” is created for hard drive partition /dev/hda10, an encrypted filesystem is created through the mapping, and finally the filesystem is mounted:

```
cryptsetup create example /dev/hda10
mkfs -t ext2 /dev/mapper/example
mount /dev/mapper/example /mnt/secret
```

3.3 Remote storage used with iSCSI

iSCSI is a network block device that runs on top of TCP/IP. A device offering storage space with iSCSI is known as an *iSCSI target*. A client that connects to a target is known as an *iSCSI initiator*. Protocol naming is not an accident, as any iSCSI resource is perceived

as a regular SCSI device.

Since a network block device appears as a local device, and storage partitions do not visibly differ from local drives, the disk operations with iSCSI, as illustrated in Figure 3, follow a straightforward logic. The arrows on the upper left-hand section signify Operating System's read and write operations. Similar arrows on the lower right-hand section signify physical read and write operations, which are performed on the actual storage media. When a process wants to read data, the iSCSI transport layer emits a command and delivers it over the network to the iSCSI target. The storage system reads data from the physical media, as specified by the command, and sends it back, in pieces if needed, through the transport layer. When the data packets reach the initiator, iSCSI autonomously merges their contents, reordering the packets as necessary, and delivers the data to the requesting process. For a process writing to the iSCSI drive, the transport layer splits the data into packets and transmits them to the target. As the packets arrive, the iSCSI target extracts the data and writes it to the physical media.

For both a user and an application, this is completely transparent. Even in the case of a temporary network failure the data is not lost. All filesystem operations go through an OS-maintained cache; if and when a device does not immediately respond, the operation is queued and eventually delivered when possible. The end-to-end transport involves several subsystems, including (but not limited to) SCSI, networking and TCP/IP stack, each with their individual timeout periods. As long as no timeouts trigger, the system will automatically recover from link failures and continue the interrupted operation as if nothing had happened.

SCSI protocol has been designed to work in modern, practically uncontrollable and insecure networks. As such, before an initiator may use a target's resources, it must log in. The login procedure uses CHAP authentication [39, 40] with a strong recommendation to use 128-bit secrets. CHAP stands for Challenge Handshake Authentication Protocol, and it is a remarkably simple but effective way to authenticate a remote party against a shared secret. When CHAP is employed, the party who wants to authenticate its peer generates a random value and sends it to the other end. The peer calculates a cryptographic hash from the combination of the random value and the pre-shared secret, and sends this hash back. The authenticator compares this received value with a locally calculated hash, and if the two values match, the other end is successfully authenticated. If the peers share two separate secrets, they can authenticate each other. This method is known as *mutual authentication* and it allows both parties to ensure that their peer is not an impostor. No matter which method of CHAP authentication is used, the shared secret (or secrets, if mu-

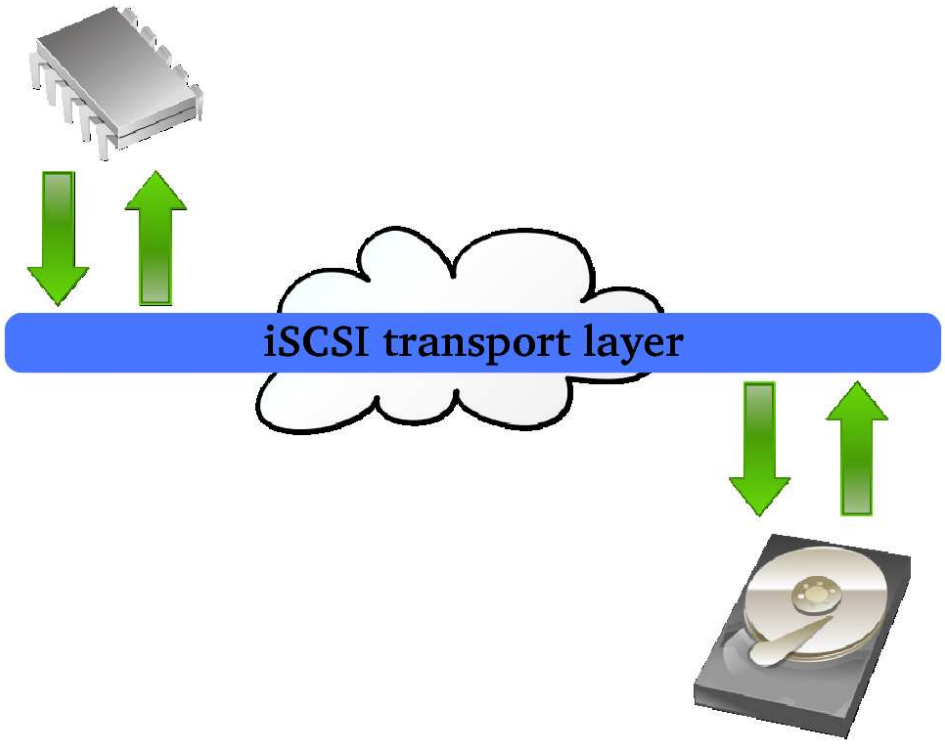

Figure 3: Read-write operations over iSCSI transport

tual authentication is desired) must have been transferred separately – by hand or postal service.

Interestingly enough, the iSCSI protocol does not require or even provide encryption. This is to address a common use-case, where an iSCSI storage is used inside a local, or otherwise secure network, as a simple storage facility. Insecure connections are meant to be protected with IPsec [27, 41].

An iSCSI target is a logically identical replacement to a directly connected SCSI device. As such, it must support multiple concurrent, mixed read and write operations. This requires asynchronous operation – which by itself is not surprising, but it adds an interesting twist. In order to operate with several initiators at the same time, even the login procedure must be asynchronous. As such, the protocol specification requires that iSCSI login parameters may arrive in multiple, distinct packets.

An iSCSI login procedure is used to form a TCP connection, negotiate session parameters and optionally authenticate either one or both of the parties. Although optional, authentication is strongly recommended. During login the payload contains key-value pairs, using the form ”key=value”, in plain text format. Below is an excerpt from an initiator’s actual

login message:

```
TargetPortalGroupTag=1
AuthMethod=CHAP
CHAP_A=5
CHAP_N=bostik
CHAP_R=0xd9d8be8ed12f3044fcfec9f20f00a2fe
CHAP_I=51
CHAP_C=0x33f537aba09d7036b781ccbb73e7e448
HeaderDigest=CRC32C,None
DataDigest=CRC32C,None
DefaultTime2Wait=0
```

There are notably more keys and values, but presenting them all here would be unproductive. The iSCSI target responds in kind, indicating in iSCSI header whether the login succeeded or not. In case of a successful login, the response also has a set of key-value pairs which tell the initiator the final session parameters.

An iSCSI protocol packet is illustrated in Figure 4. The packet has a combined TCP/IP header at the beginning, with iSCSI header embedded at the start of the TCP payload. The iSCSI header may be followed by an optional CRC (Cyclic Redundancy Check) field, the presence of which is negotiated during login. The rest of the payload area is then used up by an optional iSCSI payload. The payload may also be followed by an optional CRC field. If the iSCSI payload exists and is too large to fit in a single TCP packet, subsequent packets will contain only the continued iSCSI payload. An iSCSI header is therefore only present in the first TCP/IP's Protocol Data Unit (PDU) of any iSCSI operation. For the same reason, if a payload CRC is used, the field is present only in the last TCP packet, immediately after the iSCSI payload.

Figure 4: Packet fields in iSCSI protocol

The fields of the packets are described in Table 1. Both the iSCSI header and payload may have individual CRC32 checksums appended at the end of the protocol fields. The use and possible presence of CRC32 checksums is negotiated during login, and is therefore consistent throughout the iSCSI session. If employed, these checksums are used to detect accidental transmission errors.

Protocol field	Field length
IP header	20 bytes
TCP header + options	32 bytes
iSCSI PDU (in TCP payload)	
iSCSI header CRC (optional)	48 bytes +4 bytes
iSCSI payload CRC (optional)	rest of the packet +4 bytes

Table 1: Fields of an iSCSI packet, and their respective lengths

As already explained, an exported iSCSI resource appears as a regular SCSI disk drive. Before it can be used for storage, the resource requires the same operations as any physical drive; namely partitioning and filesystem creation.

iSCSI operation The flow of iSCSI command and data packets depends on the direction of the transmission. The presence of each individual field is best described in a matrix form, as Table 2 shows. The columns indicate iSCSI protocol fields, and rows specify each common packet type. In some cases iSCSI header may be followed by an Additional Header Segment (AHS). If such a field is present, then it invariably contains a SCSI Command Descriptor Block, or CDB. This special field denotes raw SCSI commands, and contains the instructions for accessing the SCSI device. The structure of this field is only meaningful for traffic analysis, and is used extensively in Section 5.1. For the purpose of understanding the iSCSI protocol's operation, it is sufficient to know that CDB contains the parameters for on-disk data accesses. The protocol itself is quite simple, and to better understand Table 2, is described in ample detail on the following paragraphs.

Before delving in to iSCSI protocol specifics, it may help to explain the two data transfer packet types, Data In and Data Out. The packets are named so that they are logical from the initiator's point of view. Data In is a packet that contains incoming data, and is therefore a response to an issued read command. Likewise, Data Out is a packet that contains the data which initiator wants to write on the iSCSI storage.

An iSCSI read consists of a single command packet, and has no payload. The command contains, in its CDB, the logical block number (LBA address) from which to start reading, and the amount of sectors to read, beginning from the given address. In response to a read command, the iSCSI target returns the data in Data In packets. The first TCP packet's payload starts with the iSCSI header, and the rest of the packet is used up by iSCSI payload – the data read from the device. If the read returns more data than fits in the

single TCP packet, subsequent TCP packets contain only the continued iSCSI payload. The presence of iSCSI header in the first packet is denoted by an asterisk in the table.

An iSCSI write works in a different fashion. Each write consists of two separate actions: a write command, and separate Data Out packets. The command contains, in its CDB, the LBA address from where to start writing, and the number of sectors to write. This command packet is followed by one or more Data Out packets. The first of these contains the iSCSI header, along with the data to be written on the iSCSI device. Subsequent packets, if such are required, contain only the continued payload. Again, the presence of an iSCSI header in the first packet is denoted by an asterisk.

	iSCSI header	**iSCSI payload**
Read	X	
Write	X	
Data In	X*	X
Data Out	X*	X

Table 2: iSCSI protocol field matrix. Asterisk denotes that the field is only present in the first TCP packet of an iSCSI PDU.

Reads and writes to an iSCSI drive all go through the transport layer, with operation type specified in iSCSI header and actual data in the payload. This allows for a highly efficient operation. The mere protocol part takes up very little space, leaving most of the available bandwidth for actual payload.

With a plain, unsecured iSCSI connection over a 100 megabit network, copying 2.2 gigabytes of data took an average of about 220 seconds. This translates to roughly 10 megabytes per second. (See Appendix 2 for data.)

Logically it follows that without IPsec, the performance of an iSCSI connection is bound by the speed of the network.

3.4 Storage used with iSCSI and partition-level encryption

We know that it is possible to apply partition-level encryption to any block-device. To encrypt a partition on an iSCSI device is just taking the encryption of a local hard-drive one logical step further.

Figure 5 illustrates this mode of operation. Partition-level encryption is placed on top of

an iSCSI transport, so that all the data is encrypted before it is ever transmitted over the network. To get a complete picture of the entire construction, Figure 6 illustrates messages and data flowing in it. The dotted line is a visual separator, dividing the chart in two sections, with a write sequence above the line and a read sequence below it. In addition, each of the circled numbers in the figure denotes a separate operation. In the sequence description below, a solid line denotes the separator, and the individual operations are:

1. A process writes data to a path that resides on an encrypted iSCSI partition. The write command encapsulates the location of the data on the partition, the length of the data, and naturally the data itself. From hereon the position and length are referred to as attributes.
2. Dm-crypt applies cryptographic transformation to the data, while leaving the attributes untouched.
3. On the iSCSI layer, the initiator receives input from dm-crypt, and uses the attributes of the write command to send an iSCSI write command to the iSCSI target. This command carries no payload; instead the following packets contain all the encrypted data, split into pieces that fit inside individual TCP payloads.
4. iSCSI target, upon receiving the packets from previous step, combines the payloads of data packets and uses the command attributes to write the data to the physical storage device.

---

5. When a process wants to read data from an encrypted iSCSI partition, it issues a command through the dm-crypt layer. The command contains only the parameters of where to start reading the data, and how much to read.
6. Since there is no storage payload, dm-crypt passes the command and attributes to iSCSI layer unchanged.
7. iSCSI initiator uses the command attributes and sends an iSCSI read command to the target. There is no payload.
8. iSCSI target reads the encrypted data from physical media.
9. The encrypted data is split into pieces and sent to the iSCSI initiator.
10. Upon receiving the data packets, iSCSI initiator reassembles the complete data from the pieces and passes the data to dm-crypt layer.

11. Dm-crypt applies cryptographic transformation to decrypt the data and finally passes the now unencrypted data to the requesting process.

The beauty of this setup is that for the process that is reading and writing the data, the very existence of dm-crypt and iSCSI layers is transparent, and only the latencies introduced by iSCSI layer are noticeable.

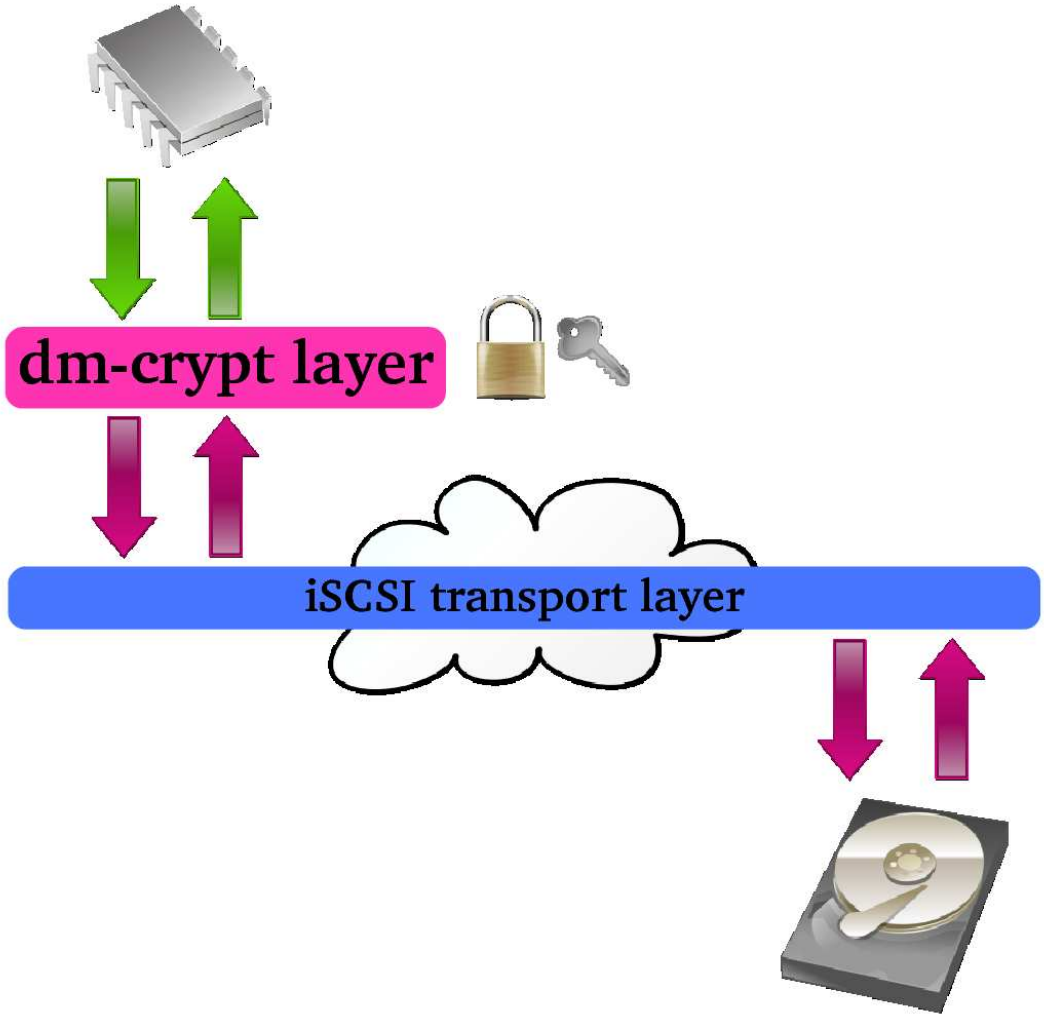

Figure 5: Read-write operations through dm-crypt translation layer combined with iSCSI transport. Purple arrows indicate encrypted content.

Individual iSCSI packets in this construction are illustrated in Figure 7. When compared to Figure 4, it is clear that the packet structure is exactly the same. The standard TCP/IP header is followed by an unencrypted iSCSI header, and only the actual payload – the data that is stored on iSCSI device – is encrypted. Most importantly, the CRC fields are calculated and added by iSCSI layer, and therefore the payload CRC is calculated from the encrypted content, but is itself again unencrypted.

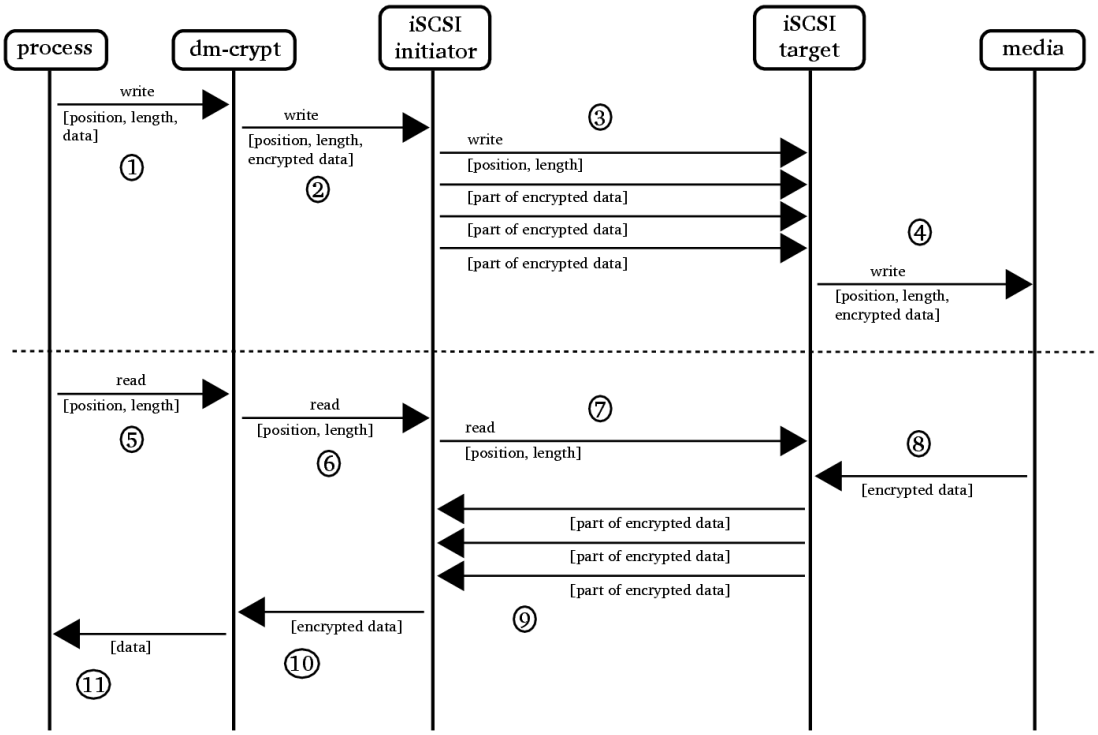

Figure 6: Data flow and sequence chart for dm-crypt on top of iSCSI

Figure 7: Packet fields in iSCSI protocol, with partition-level encryption applied to payload. Purple indicates encrypted content.

The iSCSI target hosting the physical storage can only see that the initiator stores illegible binary garbage on the allotted storage area. It is this property which allows us to use a networked storage area for secure backup, even if the server itself is considered insecure.

Performance of this setup is bound by both the CPU processing power and network speed, with naturally the lesser of the two dictating the results. Current hardware and commonly available network connectivity practically ensures that at least for now the bottle-neck will be the network. Upstream speeds that even reach, not to mention exceed 10Mbit/s are very rare outside operator or university networks.

3.5 Selected cryptographic primitives

For the use of the cryptographic translation layer, the following cryptographic primitives were chosen:

- AES for encryption algorithm,
- CBC for encryption mode,
- SHA-256 for key generation, and
- ESSIV for Initialisation Vector generation mode

AES, Advanced Encryption Standard Advanced Encryption Standard is the current FIPS (Federal Information Processing Standards) standard for securing sensitive documents [38]. The encryption algorithm itself is Rijndael [42], which supports variable key sizes. For AES purposes, the key sizes have been fixed at 128, 192 and 256 bits. The algorithm is considered secure against currently known cryptanalytic attacks, and as such is now commonly employed world-wide, in countless implementations outside governmental uses. Without a working cryptanalytic attack, the best way to recover an unknown key is an exhaustive search over the entire key space.

AES (or Rijndael) is a block-cipher with 128-bit block size. For the uninitiated, this means that data is encrypted in individual 128-bit blocks.

As far as key sizes are concerned, even the lowly 128 bits offer extensive protection against exhaustive searches. To illustrate: there are $365 \cdot 24 \cdot 3600 = 3.15 \cdot 10^7$ seconds in a year. 128 bits allow for $2^{128} \approx 10^{38}$possibilities. If we assume that a tremendously powerful super-computer can try 100 billion (10^{11}) keys in a second, we get:

$$\frac{10^{38}}{10^7 \cdot 10^{11}} = 10^{38-(7+11)} = 10^{20} \text{ years}$$

As a quick comparison, the universe is generally considered somewhat over 10 billion (10^{10}) years old. Adding a zero or three to the key search speed will not change the fact that this kind of attack is infeasible. Hence, the choice of key size is, for practical purposes, irrelevant.

For those interested in implementation details, using SHA-256 for key and IV generation automatically causes dm-crypt to use a 256-bit encryption key. This allows for $2^{256} \approx 10^{77}$possibilities and thus different encryption keys.

CBC, Cipher Block Chaining An encryption mode dictates how individually encrypted blocks affect to, or depend on, one another.

The CBC [43] mode of operation is very simple. Figure 8 illustrates CBC mode in encryption, while Figure 9 illustrates decryption. As can be seen, for encryption each block is first XOR'ed with the previous encrypted block, and only then fed to the encryption algorithm. For decryption, each block is first decrypted and then the result is XOR'ed with the previous encrypted block. To solve the problem with first block, IV is used in place of the missing previous block.

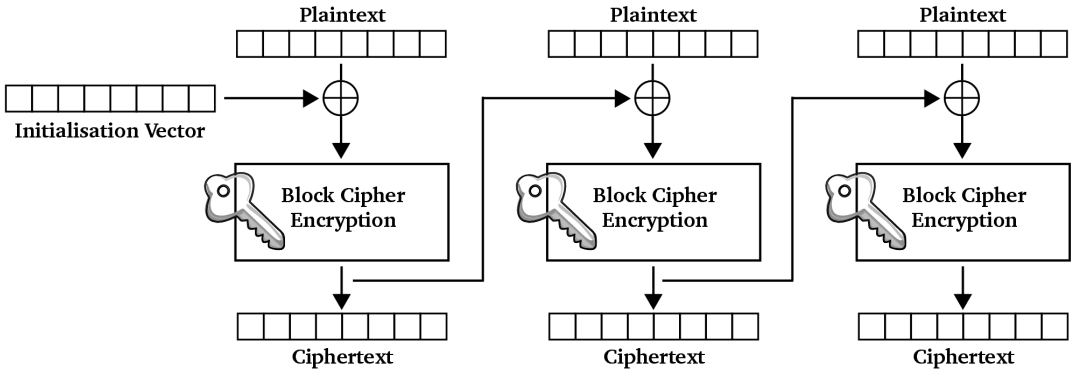

Figure 8: CBC encryption mode

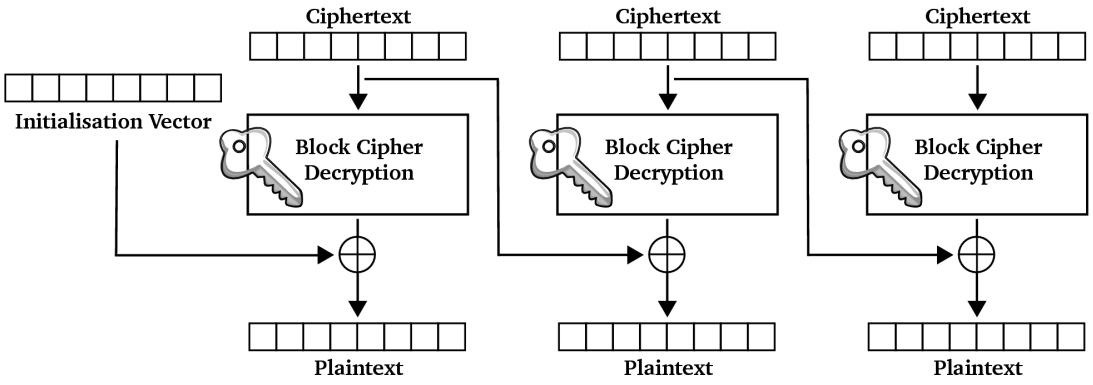

Figure 9: CBC decryption mode

SHA-256, Secure Hash Algorithm (256-bit variant) SHA are a family of digest (or hash) algorithms, defined in FIPS standard 180-2 [44]. SHA-256 is a member of this family, and like its name implies, produces 256-bit digests.

To find two different inputs that produce the same SHA-256 hash, one would have to try, on average, 2^{128} inputs. Such an occurrence is known as a *collision*. Hence, the computational security of SHA-256 is also 2^{128}, as it should require an attacker that many attempts to find a collision. But how can a secure hash of 256 bits provide only 128 bits of security? The reason for this numerical discrepancy is a mathematical concept known as the *birthday paradox*[2] [45].

ESSIV, Encrypted Salt-Sector IV ESSIV [46, pages 7 and 98] is possibly the only unfamiliar term for the general, security-minded audience. It is a special method, designed by Fruhwirth, for generating individual and secure IV's, especially for CBC mode. Although Schneier states in [43] that the IV should not require any security, that statement only applies in the context of the encryption mode alone.

Saarinen discovered that for partition-level encryption, where each sector is encrypted individually, CBC mode with known IV's allows for a watermark attack [47]. The vulnerability makes it possible for an attacker to prove with extremely high probability that certain file is stored in the encrypted partition. Not only does this attack provide a method of identifying encrypted volumes, but by compromising privacy and confidentiality it also allows to locate the watermarked files. As such it makes it possible to mount further cryptanalytic attacks (namely, known-plaintext and chosen-plaintext) against the encrypted partition.

ESSIV is a method to counter this attack. When ESSIV is employed, the IV of a sector is a function of both the sector number and the encryption key. For an outsider this renders the IV unpredictable, and the watermark attack powerless.

3.6 Secure and fast data destruction from solid-state media

The presented solution, as well as the prototype implementation rely on storing both the connection parameters and the encryption key on a USB thumb drive. Therefore, the

[2]The birthday paradox states that it is far more likely to find two identical values in any randomly distributed set of data than it is to find a match for a given value in the same set.

thumb drive is effectively a security token. This in turn means that a removable storage device contains critical and sensitive data. If, at any point, the thumb drive is replaced, the old one must be wiped clean of all the data stored in it. As modern USB thumb drives are basically solid-state memories based on NAND (Boolean not-and operator) flash memory technology [48], this may not be as easy as one would think.

Destroying data from a storage media permanently and at will is a well-understood concept in computer and information security. For magnetic media, such as computer hard drives, overwriting the drive several times with random and pseudo-random data is a commonly known and used technique [49].

In the case of solid-state media the things get somewhat more interesting. A flash storage cell can withstand only a limited amount of rewrites before it burns out. In order to avoid wearing the memory chips out prematurely, all flash-memory based storage device controllers automatically employ rather sophisticated wear-leveling algorithms [48]. In practice this means that the physical location of data changes between writes, even if the logical location stays the same. Furthermore, some algorithms move static data around to allow all storage cells to be used in equal fashion. Effectively this means that deleting and even overwriting files on a USB thumb drive does not actually destroy the data.

In an even nastier turn of events, solid-state memories have spare storage cells. The actual amount of storage space is therefore greater than the visible size of the device, and there is no way to exert control over how those spare cells are used or rotated. To securely wipe data from a solid-state storage device would, in fact, require to overwrite the entire storage space so many times that all cell rotations have been through at least once.[3]

Considering that the sizes of thumb drives currently range from couple of hundred megabytes to several gigabytes, securely and permanently deleting data from them is actually impractical. The fastest write speeds are no more than few megabytes per second. These two aspects compound to a single fact: completely wiping data from flash memory would take an inconsiderable amount of time. A practical method to securely delete data from solid-state storage would therefore be quite welcome.

Oddly enough, the idea of fast and provably secure data destruction sparked the imagination of some of my friends and acquaintances. After some brainstorming, the following single-shot procedure was suggested. A bulky, fireproof safe is installed in close vicinity

[3]There is a lack of public research regarding application of forensic recovery techniques on solid-state memory, so we do not know whether a single overwrite is actually enough.

of the devices using USB thumb drives. Inside this safe are stored a small box of thermite[4] and a sparkler. To destroy the data on a USB thumb drive, one would simply insert the device in the box of thermite, ignite the sparkler and close the door. Thermite contains all the oxygen it needs for burning, so the safe can (and should!) be sealed tight. Fire hazard, safety and other aspects would of course need a separate study and plenty of prototyping. Due to the highly flamboyant nature of such experiments, I am willing to hazard a guess there should be no shortage of volunteers.

[4]Mixture of finely ground aluminium and black iron oxide (Fe_3O_4). Once ignited, thermite burns with an extremely hot flame, and the temperature easily exceeds 2000°C.

4 Implementation of the secure backup storage

The first and foremost idea behind the work has been that a user or a business entity could do off-site backups securely and transparently. Understanding the technology and the solution details should not be required. By far the easiest way to do this would be to encase both the local file storage and the off-site backup facility logic in a non-intimidating and relatively cheap device. In order to achieve full transparency with iSCSI and partition-level encryption, deployment needs to be as straight-forward as possible. To this end, I can foresee a simple file server appliance that:

- connects to local network;
- has no monitor, just a simple liquid crystal display (LCD) for status and device's IP address;
- comes with plenty of RAID-1 hard disk space and allows to easily replace broken drives;
- exposes the disk space via Samba and NFS; and
- has a simple browser-based control interface.

The iSCSI connection parameters are provided on a USB thumb drive, and ideally the service provider would deliver this device with the service agreement, making the service easy enough for anyone to use. Also, the locally generated encryption key would be stored on the same thumb drive, rendering the process as straightforward as possible.

After initial deployment, a small office user needs to change nothing but the Samba and NFS access controls, and possibly the backup and check frequency. Restoring backups must require nothing more than a selection of date. Only requirement for restoring is that none of the current files are overwritten – in practice, the backup is extracted to a separate path, which must be visible over Samba and NFS.

With the work done so far, designing and building such an appliance product should take no longer than about two months. There are only two technical limitations I can envision, and they both stem from backup restoration. As the backup expands to complete copy of the backup state at given date, there needs to be an abundance of space available. Also, since the backup may be rather large and the network connection relatively slow, the

restoration must be done asynchronously. Especially the latter is, more than anything, an UI issue.

Going through a complete productisation is the only way I can see the technology being used as a truly transparent solution. However, in order to make such appliances actually usable, there first must be a service that offers iSCSI storage space over the Internet. Selling appliances like these in conjunction with the service might provide both a marketing advantage and an extra revenue source.

An interesting property with the implementation is that it should be possible to scale to relatively large installations. The critical variables are available storage space and bandwidth. Cryptographic performance needs to be only as fast as the network speed, requiring only a fast enough processor.

In the following sections I describe my prototype implementation. This not only proves that the concept is usable and functional, but also that it really is possible to create a highly secure but still transparent off-site backup system.

4.1 Building an iSCSI target

To test and use dm-crypt on top of iSCSI, we need an iSCSI target. Easiest way to get one is to build it. For that we need the following:

- A computer running Linux, kernel version 2.6.14 or later
- A current version of iSCSI Enterprise Target software (IET) [50]

In my setup I used a Debian unstable machine, custom-built kernel 2.6.16.20 and IET version 0.4.13.

IET allows to use any block device or file for export. As pointed out in Section 3.3, a rather neat side effect of this is that regardless of the physical iSCSI storage media, it appears as a SCSI device. The exported storage area can be anything, including a RAID or even LVM device.

In my test setup, I exported a 3-gigabyte, unformatted IDE hard drive partition. On the iSCSI target, it was a local partition `/dev/hdc10`; For the iSCSI initiator, the device

appeared as `/dev/sda`. The device names are included here for a reason: with a Linux system, the disk bus and connection type are reflected in their device names. IDE drives are shown as `/dev/hdX`, and the partitions on them are therefore `/dev/hdXN`, where X is a letter enumerating the drive and N is a number identifying the partition. On the other hand, SCSI drives, as well as other drives that are used through the SCSI layer, are shown as `/dev/sdX`, and the partitions on the drives are therefore shown as `/dev/sdXN`. Naming convention for X and N is the same.

All aspects of an IET service are defined in configuration file, `/etc/ietd.conf` or at run-time with the tool `ietdadm`. The configuration file will be covered only to the extent required by this setup, and the administration tool will not be covered any further.

IET configuration file

There are very few aspects of IET configuration file that need to be covered. iSCSI targets support *service discovery*, which means that iSCSI initiators can connect to the target and get a list of available storage area resources. Whether this operation requires authentication, is configurable.

For an initiator to gain access to a resource, they usually need to authenticate. **Note:** this authentication may be, and in case of IET is, completely separate from discovery authentication.

In the case of IET, each iSCSI storage area is defined with a unique block. This block has the following elements:

Target This directive begins an IET target configuration block. It is followed by the target's iSCSI Qualified Name, as defined in [51]. Any lines following this directive are configuration options for the particular target.

IncomingUser This directive is followed by a user's login name and their respective CHAP secret for authentication. There may be several such directives for a single block, which means that any of the users may log in to gain access to the target.

OutgoingUser This directive is followed by iSCSI target's effective user name and their respective CHAP secret. The name can be anything, and is not limited to the name of the user IET service is being run as.

Lun This directive is followed by a SCSI Logical Unit Number, path to exported resource and the resource's type. Logical Unit Numbers are an artifact of SCSI architecture, and were used to identify devices connected to the same SCSI bus. In my implementation using a value of 0 was always correct. Resource type is specific to IET, and allows to specify whether the resource is real storage space (”Type=fileio”) or just a traffic sink (”Type=nullio”).

iSCSI options There can be any number of iSCSI target options, some of which are honoured and some which have not yet been implemented in IET version 0.4.13.

The essential lines for a resource are Target and Lun, and with those one can define a minimal block configuration. Authentication is optional, and iSCSI options have actually quite sane defaults. If authentication is used, it is essential to use only printable characters for CHAP secret.

To generate CHAP secrets that fulfill the aforementioned 128-bit requirement, one can use for example this simple command:

```
head -c 16 /dev/urandom | mimencode
```

The command reads 16 bytes (128 bits) of *computationally* random data and then base64-encodes it to create a printable string. Base64-encoding preserves entropy, so the printable string is just as difficult to guess as the original 16-byte binary blob.

4.2 Building an iSCSI initiator

iSCSI initiator is used to connect to an iSCSI target, and the core of this thesis depends on having a solid iSCSI initiator implementation. I needed something that works with IET and runs on a Linux system. As such, the choice of open-iscsi [52] was really not an issue. To use open-iscsi, the Linux kernel must be built with CRC32C and Cryptographic API for MD5. (CRC32C is a specially optimised version of CRC32.) The iSCSI modules from the kernel are not really needed, because open-iscsi provides more up-to-date replacements in any case.

To build the package, the only requirement is that the sources for the currently running kernel are available. For my tests, I used release open-iscsi-1.0-r485. In addition

to kernel modules, open-iscsi provides two userspace programs for using the iSCSI facility, `iscsid` and `iscsiadm`. The daemon, `iscsid`, is the actual iSCSI initiator; `iscsiadm` is a tool for issuing commands to the daemon and configuring the initiator's parameters.

To use open-iscsi, there needs to be one additional file in place: `/etc/initiatorname.iscsi`. This file tells `iscsid` which iSCSI name it should use for itself. The file has two directives – a mandatory `InitiatorName`, and an optional `InitiatorAlias`. The contents of the file on my system were:

```
InitiatorName=iqn.2006-06.invalid.net.demo.plop:01.500ab8edf540
InitiatorAlias=Plop
```

4.3 iSCSI target preparation

Before use, the iSCSI target must be prepared. The newly exposed block-device is treated just like any other storage device. In other words, it needs partitioning, and the partition requires a filesystem on it. To make iSCSI resource usable for remote backup, the following sequence of operations need to be taken first.

4.3.1 Initial iSCSI login

For the initial login, we need the following information:

- iSCSI target's IP address
- Our assigned login name and associated CHAP secret
- Target's identifier name and associated CHAP secret[5]
- The node identifier of the iSCSI storage area allocated to us[6]

[5]CHAP secrets for the target and the initiator must be different. The iSCSI protocol specification explicitly states that if the two CHAP secrets are identical, it is a severe violation and is therefore checked against during login stage. A failed check is treated as authentication failure. [27]

[6]This requirement has been abandoned during further development. Future versions of open-iscsi will use target's address and node name directly.

Presumably the iSCSI provider has prepared all of these for our use. For my setup, I stored the parameters in a simple configuration file residing on a USB thumb drive and accessed that directly. This approach was used to keep user intervention and manual steps to a minimum.

The initial login is not as straightforward as it might be. Configuration for open-iscsi's targets are kept in persistent databases, and without such configuration no connections are possible. The very first operation is to issue an iSCSI discovery request to the target. The command for this is:

```
iscsiadm -m discovery -t sendtargets -p <target's IP address>
```

In response the target will tell us names and identifiers of accessible iSCSI resources. The response output looks something like this:

```
[5e6c31] 157.24.24.206:3260,1 \
iqn.2006-06.invalid.net.demo.target:storage.c47d1ec75077
[5c03d6] 157.24.24.206:3260,1 \
iqn.2006-06.invalid.net.demo.target:storage.5a611d98c719
```

The first field between brackets is an iSCSI node identifier, and it is needed for all subsequent iSCSI operations. The following three fields are: target's IP address and port, separated from each other by a colon (":"), and the number after a comma is the target's Portal Group Tag. This tag is a certain kind of a connection identifier. iSCSI supports multiple sessions over a single connection, and may act as a single point of entry to a group of targets. The groups must be identified and somehow separated from one another. Portal Group Tag is used to identify which group any given target belongs to.

The final field is the storage area's iSCSI name. The received values, with the exception of Portal Group Tag, are stored in open-iscsi's persistent database. The reason for not storing the tag is that an iSCSI target may be reconfigured between sessions, and as such the Portal Group Tag can change. Since the tag is only required for the duration of a session, re-discovering the value on each connection is justified.

Open-iscsi's default values for any discovered iSCSI target assume that no authentication is required. When a target is discovered, the configuration is written to persistent database. Although `iscsid` re-reads its configuration file for each new session, in practice I noticed that at least authentication data is not updated. Hence, before logging in to

the target, the authentication parameters need to be explicitly set with `iscsiadm`. To change and store authentication parameters for first of the target nodes above, the following commands are issued:

```
(1) iscsiadm –m node –r 5e6c31 –o update \
–n node.session.auth.authmethod –v CHAP
(2) iscsiadm –m node –r 5e6c31 –o update \
–n node.session.auth.username_in –v <target's user name>
(3) iscsiadm –m node –r 5e6c31 –o update \
–n node.session.auth.password_in –v <target's CHAP secret>
(4) iscsiadm –m node –r 5e6c31 –o update \
–n node.session.auth.username –v <our user name>
(5) iscsiadm –m node –r 5e6c31 –o update \
–n node.session.auth.password –v <our CHAP secret>
```

The commands do the following:

1. Authentication method is set to CHAP. As mentioned, the default setting would be not to try authentication at all.
2. The iSCSI target's username for the connection is set to the value given after -v
3. The iSCSI target's CHAP secret for the connection is set to the value given after -v
    - Commands 2 and 3 both have "_in" at the end of their configuration options. These are used to signify that the options refer to values that are "coming in" from the iSCSI target
4. The iSCSI initiator's username for the connection is set to the value given after -v
5. The iSCSI initiator's CHAP secret for the connection is set to the value given after -v

Once given, the settings will remain in the persistent database. They need updating only when one or more of the settings change.

The final step is to log in to the iSCSI node. The command for this is:

```
iscsiadm –m node –r 5e6c31 --login
```

Now that the user has logged in with proper credentials, there is a new unpartitioned device visible in `/proc/partitions`.

4.3.2 Partitioning the target

A newly accessed iSCSI resource is just like any untouched storage device. In order for us to be able to store anything on it, we need to create a partition table, so that the OS sees the device as a usable storage.

The standard tool for partitioning devices in a Linux system is `cfdisk`. We create a single, primary partition with all available space and save the new partition table.

Partitioning is done without dm-crypt layer for two reasons:

1. Partition-table needs to be visible. When an iSCSI initiator connects to the target, the OS reads the partition table directly from the new device.
2. Backup purposes. It should be possible to read the contents of the partition and store that as a snapshot image. The contents will not be usable without correct encryption parameters, so storing the raw disk image amounts to storing an encrypted backup.

4.3.3 Refreshing partition table view

This step should not be required, but I chose to include it to eliminate any possible corner cases that might otherwise arise.

A freshly partitioned device may not always show up in the OS with its updated partition table. For this reason, and this reason alone, we need to log out from the iSCSI target, and then log in again. For the OS this appears the same as removing and re-attaching a device. Upon finding a new device, the OS reads the partition table.

We now have a guaranteed, fresh view of the iSCSI target's partition table.

4.3.4 Cryptographic mapping

In step 4.3.2 we created a single partition on the iSCSI storage device. To encrypt all of the contents of this partition, we now create a cryptographic mapping for that partition. All subsequent disk operations will be done through this mapping layer.

The command to create a cryptographic mapping is `cryptsetup`. For the mapping, I chose sane and safe values: AES encryption algorithm, CBC encryption mode, and ESSIV as IV generation mode. These were described in Section 3.5.

Before creating the cryptographic mapping, we need to decide, or agree on, the encryption key, and for proper security, the key must be randomly generated. Since we are going to store the encryption key on a small, removable USB thumb drive, it is only prudent to create the encryption key directly on that. In the implementation, I use the path `/mnt/usbkey` for the USB device's mount point. To generate the encryption key, I used the command:

```
head -c 32 /dev/random > \
/mnt/usbkey/iscsi-partition-encryption-key
```

This command reads 32 bytes (256 bits) of random data from the OS's entropy pool and stores the bits in a file on the USB thumb drive. Now we can at last create the cryptographic mapping. By using the stored key from a thumb drive we never need to remember (or even personally know) the encryption key – and as such can not mistype or forget it either.

We assume that the iSCSI partition shows as `/dev/sdb1`. Then the command to create the cryptographic translation layer and corresponding mapping is:

```
cryptsetup -c aes-cbc-essiv:sha256 -h sha256 -d
/mnt/usbkey/iscsi-partition-encryption-key create target
/dev/sdb1
```

When created, the mapping is given a name. In the above command the name is "target", and as a result, the created device mapping will be `/dev/mapper/target`. The additional options are used to select encryption key and cryptographic primitives as outlined in Section 3.5.

4.3.5 Partition randomisation

Even though we have a mapping for the partition, there is one notable problem. The partition is completely empty, and as such would allow both the iSCSI provider and a possible attacker to see how much of the disk space is used.

Therefore, the very first operation before doing anything else is to randomise the (non-existent) contents of the entire partition. We do this by writing the whole partition full of unknown data through the cryptographic mapping layer. This way the partition is prefilled with binary garbage, and should anyone gain access to the iSCSI target area, they can not even deduce which parts of the storage area have data and which are empty.

For randomisation, the following command can be used:

```
dd if=/dev/urandom of=/dev/mapper/name bs=8192
```

Without the last parameter, the data would be written one sector at a time. In practise this would mean that the writes are done 512 bytes at a time, and consequently each sector would require a separate iSCSI command. By using a block size of 8 kilobytes, we can cut down the number of iSCSI commands – and hence additional command round trips – by a factor of 16.

Although this will take a rather long time, the performance of using /dev/urandom is not an issue. The randomisation is done only once, when the remote storage is first prepared. Anyhow, in most cases the bottleneck will still be the network connection: no matter how quickly the randomisation data was produced, it could not be used any faster than the network upstream speed allows.

4.3.6 Filesystem creation

Now that the partition is completely randomised, it is finally feasible to create a filesystem on it. As with all disk operations, the filesystem is created through the cryptographic mapping. In choosing the filesystem, there is one additional factor to be considered. In this setup an active iSCSI connection is subject to both traffic analysis and cryptanalysis. Therefore we want the traffic to contain as little known or identifiable details as possible. We also want the best possible performance, because in majority of the cases the

bottleneck will be the network connection.

To this end, I chose to employ an unjournaled filesystem, namely ext2. See also Section 5.1 for further details. Command used was:

```
mkfs -t ext2 -m 0 /dev/mapper/target
```

In the command, there is somewhat unusual option used. The `-m` switch is used to specify how large a percentage of the created filesystem should be reserved for the root user alone. In my implementation, backups are taken and accessed as a non-root user. Therefore it would be counterproductive and actually questionable not to use all available space.

The choice of ext2 was made on the premises that it is well understood, unjournaled, and most of all, readily available and accessible from practically any modern OS. Finding an optimal filesystem format for a secure and remote storage goes beyond the scope of the thesis. It would, however, make an interesting research project.

There may be at times other SCSI devices present in the system. Because of this we need a way to positively identify the iSCSI partition from regular SCSI partitions. Any automated backup scripts will work with fixed paths, and therefore the mount points must be used persistently. Partitions do not have any meaningful identifiers themselves, but filesystems do. When created, modern filesystems are assigned a UUID, which stands for Universally Unique Identifier [53]. This is a 128-bit, partially unpredictable bit-sequence having certain generation rules that make it virtually impossible for any two properly generated UUID values to collide. As such, we can extract and store the UUID of the newly generated filesystem as our identifier. Each time we wish to mount the iSCSI partition, we need to first find the correct SCSI partition by its UUID. Only this way can we be certain that the filesystem is the correct one.

Incidentally, new Linux systems have a piece of software, called `udev`, that uses the same method to create device nodes dynamically. In fact, it creates several symbolic links that allow to access filesystems by many of their identifiers. To persistently map removable storage, UUID-based identification is often used. For all the filesystems `udev` discovers, it creates, among other things, a link `/dev/disk/by-uuid/<UUID of filesystem>`, which points to the actual device. However, `udev` does not know about dm-crypt's device mappings and as such does not scan the available mappings in

`/dev/mapper`.[7] Therefore we can not rely on `udev`'s help when mounting the encrypted filesystem. For this reason, and this reason alone, we need to initially store and later on manually extract and check the UUID of our encrypted filesystem before mounting.

4.4 iSCSI target usage

Using an iSCSI target for secure backup follows the same logic as preparation, only there are not as many steps. The iSCSI target is ready, so we only need to log in, create the cryptographic mapping, and then mount the filesystem.

The command for logging in is the same as in 4.3.1. The operation should succeed without problems, as the parameters have not changed. A successful iSCSI login causes a new device and its partition table to appear in the system. Next step is to create a dm-crypt mapping for the iSCSI partition, and for that we can reuse the command from 4.3.4.

Although said before, there is one fact that bears mentioning again. Creating the dm-crypt translation and mapping does not, and can not, verify the content of the parameters. This really means that providing incorrect encryption key would not be noticed at this point.

With both the iSCSI connection and dm-crypt mapping in place, we can now mount the filesystem on encrypted partition. The command does not have any special options, and is just simply:

```
mount /dev/mapper/target /mnt/target
```

This is the first and only step where it is possible to know whether the cryptographic translation layer was configured with correct parameters. With the right parameters, the remote filesystem will be mounted silently. In case of a parameter mismatch, only a mount error indicates that the cryptographic layer produces unexpected results.

[7] Mappings created in LUKS [46, pages 101–110] format are scanned, but LUKS imposes certain restrictions that were not acceptable in the scope of the thesis.

Helper scripts to rescue

To make the development and testing somewhat easier, and to prototype unassisted use I wrote two simple wrapper scripts. These allow to attach and detach the remote, encrypted filesystem at will. The following scripts, along with the entire prototype implementation, are included with the bound copy of this thesis on a CD-ROM, as Appendix 3.

The first script, `attach.py` logs in to the iSCSI target, creates the cryptographic mapping and mounts the remote filesystem after checking that its UUID matches the one that was stored in Section 4.3.6. The second, `detach.py` reverses the process.

So, to attach the secure backup drive, one only needs to plug in the USB thumb drive and issue the command

```
python attach.py
```

and likewise to remove the backup drive from the system, one uses the command

```
python detach.py
```

before removing the thumb drive. The less there are manual steps involved, the better.

4.5 Making backups

The remote iSCSI device is now attached, prepared and mounted over dm-crypt. As such the remote storage is finally usable for secure backups.

In all its simplicity, doing a daily backup requires only a single command, executed in the `src/` directory:

```
python run_backup.py
```

This is another helper script I wrote to demonstrate the automation capabilities of the system. The associated python classes under the hood will take care of everything. On first invocation, a full backup is created. On subsequent invocations, full backup is taken if the previous one is at least a week old. Otherwise, only an incremental backup is created.

To maintain this level of simplicity, certain limitations are imposed. First and foremost, the backups are taken from files and directories residing inside a single directory. In practice this path would be the root directory of a common data repository, so the limitation is quite acceptable. Second, the locations of stored backups are fixed. This allows to browse and restore backups with ease, while also removing the need for excessive manual steps.

The most important factor is that it is quite simple to run the backup creation script periodically, unattended. There are some limitations, though. The script tries to ensure that the directory hierarchy on the iSCSI storage is sane and fitted with proper permissions to do backups as a non-root user. To this end, the script will invoke certain commands as root (through `sudo` [54]) if needed. Consequently, the backup user must be able call those commands through `sudo`, and moreover, without needing to provide a password. Apart from a working Python installation and standard modern *nix tools, the only additional tool is `rdiff` [14].

For purposes of integrity checking, each created backup file is hashed using SHA1 algorithm [55], and the resulting cryptographic checksum is stored locally. Running periodic integrity checks on the stored backups ensures not only that the files have not been altered, but also makes a certain class of attacks much more difficult, if not outright futile. (See Section 5.3.1 for details.) My prototype includes a very straightforward script, which compares all the backup files stored on the iSCSI partition against their local hashes. A smarter implementation should compare the backups from the previous two or three weeks, as well as some randomly selected older files.

Backup operation

The fixed paths on local side are `/var/local/backup` as the backup area's top directory, `/var/local/backup/files` as the root of the data repository and `/var/local/backup/sha1` as the hash directory. On the remote side, there are only two directories used. Assuming that the remote, encrypted filesystem is mounted on `/mnt/target`, the directories are `/mnt/target/full` for weekly backups, and `/mnt/target/delta` for incremental daily backups.

Whenever a weekly backup is required, the contents of `/var/local/backup/files` are packed inside a tarball, which is then fed to `rdiff` for signature generation. This signature file is stored in the top directory `/var/local/backup`. The tarball is then

further compressed with gzip. The SHA1 hash of this compressed file is stored in `/var/local/backup/sha1`, and finally the compressed tarball is moved over to the iSCSI backup drive, in the aforementioned `full` directory.

For daily backups the procedure is a bit different. The contents of `/var/local/backup/files` are once again packed inside a tarball, but this particular tarball is not stored anywhere. Instead it is piped directly to `rdiff`, which then generates an incremental backup by comparing the weekly signature file and the incoming tarball. This incremental backup contains only the modifications made to the files since the creation of the signature. This modification file, or *delta*, is then stored, compressed with gzip and hashed, before finally moving it over to the iSCSI backup drive. The deltas are stored, unsurprisingly, in the remote `delta` directory.

Although it would be more straightforward to directly compress the tarball before generating the signature, this would actually result in considerably larger deltas. Assume a small data repository has some tens of megabytes of files. For two consecutive days, the delta for uncompressed tarballs with few changes is usually some hundreds of kilobytes. In comparison, delta against the **same** data sets with compressed tarballs can be several megabytes. The difference (and delta size) only increases as the amount of data grows. Hence, with deltas against uncompressed tarballs, we can use the remote storage space much more efficiently; transfer times are shorter, and we can store backups for longer periods of time.

4.6 Restoring backups

Backups are useless if they can not be restored when a need arises.

For any given date, there are at most two files to retrieve from the backup device. For dates with full backup, the complete snapshot is used directly. For any of the intervening dates, the most recent full backup and the delta from actual date are used together. To automate the process, I used a simple interface: simply specifying the date from which the backups are required, the relevant files are transparently fetched and extracted to a fixed path. The location for this path is right next to the repository's root directory.

In other words, when the command

```
python restore_backup.py
```

is issued, the backups from the chosen date are automatically extracted to `/var/local/backup/extracted`. From there it is easy to pick the files to restore. The extraction directory should be considered temporary, as it is removed and recreated each time a restore operation is run.

4.7 Performance

To measure the performance of iSCSI and dm-crypt's impact on transfer speeds and execution times, I ran several tests. The resulting data is in Appendix 2. The computers used were somewhat old desktop PC's. The iSCSI target was a 1.1GHz AMD Athlon, and the iSCSI initiator was a 1.8GHz Intel Pentium 4.

My conclusion from the results is that with any modern PC, the impact of dm-crypt is measurable but negligible. With plain iSCSI, I measured transfer rates of 10.0 - 10.3 MB/s. When doing the same transfers through dm-crypt's cryptographic translation layer, the slowest transfer rates fell to 9.8 MB/s. From the two tables, 7 and 8, it is obvious that as long as we have powerful enough CPU available, the bigger drop in performance is actually caused by enabling both iSCSI header and data CRC checks. However, even with all computationally intensive operations enabled I could still nearly saturate a 100Mbit network link.

Without dm-crypt, the CPU load varied between 25% and 30%. With dm-crypt enabled, the load peaked at 89%, and varied between 65% and 75%. From these results I am willing to assert that with modern PC's that sport more than twice the processing power compared to the relatively old 1.8GHz CPU I used, the major bottleneck will be the speed of the network connection, not the encryption. The introduction of IPsec in the equation was not implemented, and consequently the impact and overhead have not been measured.

5 Security and risk assessment

Proposing any kind of cryptographic setup absolutely requires a proper analysis. I have done everything I can to identify the risk factors, partly due to the apparent novelty of the solution, but mainly because there are parts of the communication that travel unencrypted. Assessing their both individual and combined effects on the security is crucial.

5.1 Traffic analysis

Traffic analysis was done with tcpdump and Wireshark[8] [56]; the latter was originally known as Ethereal, but the project was forced to change their name. Tcpdump was used to capture the traffic, and Wireshark was later used to display the captured packets and their contents in a more intuitive way. The choice of using two programs for a task where one should have been enough was due to my personal, innate distrust for running complex and monolithic graphical applications with elevated privileges.

Analysing iSCSI traffic is quite easy. The protocol is divided into logical commands that are simple to follow. Each iSCSI command has a 32-bit Initiator Task Tag, which is used to identify the data associated with the command. This allows to have several concurrent operations on a single iSCSI resource. Moreover, it makes it relatively simple to find request-response command pairs from a traffic dump. The task is further simplified by the fact that iSCSI uses Logical Block Addressing (LBA) [57] for positioning reads and writes.

Read operations are very straightforward: the command contains LBA block number and length of data to read, starting from the beginning of the block. iSCSI write commands are notably different. Each iSCSI write actually consists of two steps, both of which are sent in separate command packets.

1. Write command with LBA block number, which tells the iSCSI device where to position the ”drive head” within the storage area
2. The data to be written, starting from the beginning of the LBA block sent in the previous step

[8] Ethereal was renamed to Wireshark on 2006-06-07. At the same time, the project website moved to www.wireshark.org.

This makes it trivial to identify and classify the traffic flowing between an iSCSI initiator and target. As I needed to identify the write commands and payloads, I copied an easily recognisable text file over the iSCSI connection. The file had 2048 A-characters, followed by as many B's, C's and D's. With such a payload, finding the correct commands was a trivial task. More importantly, it allowed to identify basic traffic patterns for each type of filesystem. This information was later used when analysing traffic with encrypted payloads.

The task was somewhat complicated by Wireshark not dissecting iSCSI traffic properly. For example, the known data with ext2 [58] filesystem could only be recognised by looking at individual TCP packets and the iSCSI PDU's inside them. The reassembled iSCSI Data Out packets were shown as having the correct length, but their data section was only marked as 444 bytes long. In addition to this, the traffic dissector constantly claimed that the SCSI exchange was unknown and refused to decode it. As a result, Wireshark displayed *wrong* data contents for the larger iSCSI transmissions.

Also, when measuring transfer lengths, there is an interesting property: Command Descriptor Block (CDB) contains raw SCSI commands. Due to this, the transfer lengths, as shown by Wireshark, are actually *disk sectors*, not bytes. From previous experiments and device properties we know that sector size was 512 bytes.

I performed traffic analysis of three separate filesystems: ext2 [58], ext3 [59] and XFS [60]. These were selected because I needed one unjournaled and two different journaled filesystems for comparison. Ext2 was chosen for its simplicity and lack of journaling. Ext3 was chosen as the first journaled filesystem due to its similarity and common structure with ext2. XFS was the third choice, on the basis that it is a completely different kind of filesystem, and has been originally designed for high-availability systems. With these three filesystems, I could identify common traffic patterns, as well as individual filesystem-specific characteristics.

Ext2 filesystem

I started by identifying the write commands and sequences. This was easy due to the fact that writing data on an ext2 filesystem follows a rather simple logic. For any file there are two distinct write operations:

1. An ext2 *inode* is reserved and written to allocation table
2. Actual file contents are written to the location of the inode

Traffic detection was only slightly obscured by Wireshark's oddities. In fact, the best way to identify the two write operations was to do the identification backwards. I identified the actual data transfer first, noted down the Initiator Task Tag for the iSCSI command, and then assumed that since these tags grow sequentially, the previous tag was used for inode reservation.

After the previous assumption, it became apparent that inode reservation would have been easy to spot in any case. The ext2 allocation table resides at the start of the filesystem, so inode reservations were always directed to very low LBA numbers. With no other activity, the next Initiator Task Tag number was used for writing the file. This operation also had significantly higher LBA number, so it was obviously outside the allocation table area.

With all the previous properties combined, it was simple to extract actual traffic patterns. Comparing plaintext and encrypted traffic dumps side by side, one can easily identify inode reservations and file writes by looking at the iSCSI commands and CDB's alone. More to the point, the payloads are in fact irrelevant. In practice this means that identifying ext2 filesystem operations from iSCSI traffic is surprisingly easy.

Ext3 filesystem

The filesystem was created with default settings. This implies ordered journaling mode. For ext3, ordered means that only metadata is journaled, and as such files are written on the filesystem as normal, but they will become visible only after the entire write operation has succeeded. In effect, when a file is written to an ordered journal ext3 filesystem, the file contents are written as normal, but the metadata will be delayed; when the actual operation has been completed, the metadata is updated in both the journal and the filesystem, and finally the metadata is marked in the journal as done. This ensures constant filesystem consistency. In case of a crash between two writes, the journal can be replayed backwards, effectively removing the last written file. This keeps damage to the filesystem at minimum and allows to recover both rapidly and effectively from failures.

This means that there will be several write operations for each file, which in turn need to be identified from iSCSI traffic. A sensible approach is to follow write operations, and

note their sizes and locations. Since ext3 is ”just” a journaled version of ext2, all non-journal operations are effectively the same. The only difference is that due to the presence of journal, the locations of files will start a little further into the filesystem, and hence the LBA numbers of their respective writes will be distinctively large.

So, for each file there are four separate write operations:

1. File contents are written on the filesystem
2. The inode metadata is written to the journal
3. The actual metadata is written to the filesystem
4. The journal is updated to reflect the current situation

All of these operations are easy to identify. With journal location and size being constant, the operations are even easier to spot. In fact, once the traffic pattern is known, file contents become irrelevant for ext3 as well. As such, identifying ext3 filesystem operations from iSCSI traffic takes very little effort.

XFS filesystem

XFS was originally created for Silicon Graphics's IRIX systems, and as such it has completely different kind of journaling semantics [60]. The journal in XFS is known as the log space, which is somewhat confusing when reading through the specifications. Use of the term ”log” also implies that the journal only cares about filesystem metadata, and the actual file contents are not journaled. The journal thus contains the information that file, of size X and attributes Y, will be written to location Z.

Each filesystem has a distinctive superblock, by which it can be identified and also automatically mounted with correct filesystem parameters. In XFS's case, the superblock is at the very beginning of the partition, at LBA block 1. Also, there is one very particular and easily detectable property in XFS's journal operations, as XFS updates the superblock contents after each write to the journal. For each file write operation there are two single-sector writes to the superblock. After identifying this very distinctive traffic pattern, it takes no effort at all to spot XFS operations from iSCSI traffic.

Sequence of any write operation on an XFS filesystem consists of the following five visible steps:

1. Journal is updated with information about forthcoming operation
2. The XFS superblock is modified
3. The file is written on the drive
4. Journal is updated and the operation marked as done
5. The XFS superblock is modified again

There appears to be a somewhat ironic trend regarding filesystem features and identifiability. One would think the more feature-rich filesystems sported more complex patterns. However, contrary to my expectations, as the complexity of the filesystems increased, their traffic patterns became easier to identify. Especially XFS's tendency to operate on the filesystem superblock, which even resides at a very peculiar location, clearly and unmistakably identifies the filesystem from iSCSI traffic headers.

5.2 Rudimentary cryptanalysis

I do not claim to be a cryptographer, and I certainly do not have the expertise to analyse and quantify the security of every individual aspect of the setup. For what it's worth, I can not even hope to be able to think of all the possible combinations. I can, however, point out and evaluate all of the obvious and some of the less obvious aspects.

First and foremost are the cryptographic primitives. AES is considered secure: there are no known attacks that would work against the full 14 rounds of 256-bit AES encryption [42, 61]. In cryptographic algorithms, a round is a set of transformations. This set of transformations is repeated a specified number of times, and the result after the final round is the output of the encryption algorithm. The best attack so far works against 9 rounds [61], and even then the complexity is 2^{224}, which is certainly less than 2^{256} but nonetheless impractical.

The hashing algorithm, SHA-256 is also considered secure. There is a known attack [62] against the basic variant SHA-1 that reduces the attack complexity from 2^{80} to 2^{69} but so

far none against the newer family of SHA-256, SHA-384 or SHA-512. Improvements on the attack against SHA-1 are probable and as such only a matter of time.[9]

The used encryption mode, CBC, would be vulnerable to a watermark attack if used alone, as has been already pointed out. However, doing the IV generation in ESSIV mode hardens the construction and renders the watermark attack powerless. As far as implementation security goes, there are no other known cryptanalytic attacks against dm-crypt. From all the previous facts it is possible to infer that the cryptographic elements, together with their implementation, are secure.

However, the security of the entire system depends on many more factors. Because only the iSCSI payloads are encrypted, and as the details of the system are assumed public knowledge, there are several classes of information leak vulnerabilities. The unencrypted iSCSI commands expose several characteristics, such as approximate file sizes, file locations, and indirectly, the underlying filesystem.

The encrypted data is written in sector-size blocks, so for every file there will be N 512-byte sectors. From this we can easily calculate that the file size is within$(N-1)\cdot 512+1$ and $N \cdot 512$ bytes. The location of the first byte of the file is always known. Whether this information is of any use, is unclear for me but certainly depends on the sophistication of the attacks.

The filesystem can be derived from the traffic pattern, as shown in Section 5.1. I expected this from the very beginning, and chose a non-journaled filesystem. This shrinks the information leak to a somewhat smaller factor, but getting the exact figures would need assistance from an experienced cryptanalyst.

The compressed files themselves also provide a possible attack vector, as the files are all compressed with `gzip`. The tool adds a 16-bit identifier at the beginning of every compressed file; therefore, every single one of the backup files starts with the bytes `0x1f8b`. This identifier allows to identify when a successful decryption has occurred, in addition to providing 16 bits of known data. This particular identifier is also known as *gzip magic header*. This non-obvious term is commonly used in *nix systems to describe a sequence of bytes a program or tool inserts at the beginning of its output. Also standardised data and storage formats have their own magic headers. The practice allows to classify and

[9] Incidentally, during the review process of this thesis an improved attack on SHA-1 was revealed. During the Rump Session of Crypto 2006 [63], De Cannière and Rechberger displayed a collision on SHA-1 reduced to 64 rounds [64, 65]. Full SHA-1 consists of 80 rounds.

identify data files by only reading their first bytes.

As it happens, the backup files themselves provide at least one additional avenue of attack. Incremental backups, or deltas are taken against a fixed snapshot no older than one week. Even though the files are compressed, the deltas are taken against uncompressed files. The changes from day to day are usually not that large, so consecutive deltas against the same snapshot will cover the same changes. This on the other hand means that some parts of the compressed deltas will be identical. In practice this means that there will be sectors that contain the same data, encrypted with the same key. The single difference is that due to use of ESSIV, these encrypted sectors differ only by their initialisation vectors. Size differences between two consecutive days will also provide a rough estimate at how much there have been changes between the two backup runs.

These characteristics may incite one to speculate whether the system is then at all secure. To answer that in short: yes it is. A longer answer requires some explaining. As far as I understand, the most dangerous property is that there are multiple sectors that decrypt to same plaintext. However, the IV generation mode, ESSIV, derives its output partly from the encryption key – and as such the security of each of those sectors is almost as strong as the encryption key itself. After all, the employed encryption algorithm has been designed to resist all currently known cryptanalytic attacks. Therefore the relatively scarce details that are available to an attacker should not really help. The security margin of the system is so high that even if it was reduced by a factor of 2^{64}, the overall work required would still be computationally infeasible.

The final vulnerability related to dm-crypt, as pointed out by Saarinen in his watermark attack paper [47], makes it possible to copy any sector of an encrypted partition somewhere else, and later on revert the same sector to its original contents by using the copy. The sector will still decrypt to its plaintext content, but if the area has been overwritten in the meantime, the file that spans that sector will be corrupted. This vulnerability is greatly mitigated by doing the periodical integrity checks. Any modifications will be eventually noticed when the files are checked against their locally stored hashes.

5.3 Risk analysis

It is good to remind the reader at this point that we assume that the server is untrusted or even outspokenly hostile. This means that one has to also posit that the contents of

the iSCSI storage area are known and subject to inspection at any time, by antagonists of all kinds. From a security point of view, this *major* premise means that the commands and payloads of the setup's iSCSI traffic are considered known; after all, the server operator could capture the traffic just before it reaches the iSCSI target. And finally; since we assume hostility, we also maintain that the server operator can and will use outside specialists.

Due to these conjectures, and because of the service context, we have to accept the paradox that employing IPsec would add very little in terms of security. No matter what kind of protection is used, the traffic is always considered unprotected. Nevertheless, IPsec *can* provide us with one benefit – it would add extremely robust integrity checks, and as such protect against accidental corruption far better than any of the iSCSI's own, optional, CRC checks. It would also make traffic hijacking practically impossible, and render a class of subtle denial-of-service attacks impotent. (See Section 5.3.1)

As such, we encounter another paradox: using IPsec in *null encryption* mode would actually add to overall security. It is important to stress that in basically any other conceivable case, this would be an utter disaster from a security point of view. Null encryption is a very specific term used with secure traffic protocols; it means that the protocol is used as normal, but the payload is explicitly not encrypted. In this particular setup, the critical payloads are already encrypted, so there would be no sense to re-encrypt them at IPsec level. However, even in null encryption mode, the combined traffic and CPU overhead of IPsec would be excusable, as it should in practice eliminate undetected transmission errors, and hence inhibit backup corruption.

No practical cryptographic protocol or setup is completely secure. There are always attacks that compromise at least one of the cornerstones of any implementation, these being security, integrity, privacy and accessibility. Crude denial-of-service attacks that target only accessibility will not be discussed. Such attacks can only prevent the use of a service and as such can be dealt with patience and good connectivity plans. The rest of the possible attacks are discussed below.

Key compromise (cornerstones targeted: security, integrity and privacy)

There are three different keys used in the setup. There are the two CHAP secrets of the iSCSI parties, and there is the encryption key for the partition.

Assume the worst-case scenario: the server administrator uses the same secret for all, or even some of the iSCSI targets. In this case an attacker would only need to rent iSCSI storage area for herself. The service provider would then give the attacker her login credentials, from which she can lift the target's CHAP secret. If the administrator uses a pool of secrets for this purpose, purchasing access to several iSCSI target areas will eventually yield all the secrets used. Because we do not know how the administrator *really* operates, we must expect the worst, and as such consider the target's CHAP secret compromised.

If the server administrator is dangerously detached from reality, the same attack could be applied to the initiator's CHAP secret as well. However, in this context I assume that each iSCSI node is assigned a unique CHAP secret, and that they are all randomly and independently generated. One should be able to expect at least that level of sanity from any serious service provider. Therefore the best methods to gain access to an initiator's CHAP secret are physical entrance and social engineering. The latter can be especially effective – few minutes of preparative detective work and good speaking skills may fool the service provider or their support staff into giving the shared secret with a single phone call. There are implementation techniques available that will allow to detect modifications and as such CHAP compromises. Even so, the detection only works if the attacker modifies data. Consequently we must consider the initiator's CHAP secret insecure, but not entirely compromised.

The encryption key for the iSCSI partition is never sent over the network. It is only known by the initiator, and furthermore, stored as a raw binary blob on a USB thumb drive. To actually expose the encryption key requires someone to knowingly dump or copy the keyfile contents. Therefore the only conceivable way of gaining access to the partition's encryption key is honest, physical burglary into the initiator's premises. In such a case, all bets regarding security are off anyhow. Accordingly, we consider the partition's encryption key secure.

Compromise of CHAP secrets would allow a crude, destructive denial-of-service attack against the iSCSI target node. If the attacker can log in to the iSCSI node, she can simply overwrite the entire allocated space. This attack is useful only for malice or revenge, as it prevents anyone from using the stored data. The security of the backups is not compromised, even if their use has suddenly been prevented. Note that since we assume the attacker may be the service provider itself, we can expect the contents of the node to be copied somewhere else for cryptanalysis. Hence the attacker does not necessarily need to bother to copy the node contents.

Compromise of the partition's encryption key on the other hand would be disastrous. Since one can assume that access to the iSCSI target is available to a determined and resourceful attacker, this key is the only thing standing between the backups and a full-blown compromise. An attacker with the correct encryption key could, at her leisure, use earlier measures to get both CHAP secrets and access all backup data stored in the iSCSI node.

Verdict: vulnerable if secrets are compromised.

Physical compromise (cornerstones targeted: security, integrity, privacy and accessibility)

Very few, if any, security measures work against physical access. Hard drives can be removed, or even whole computers may be carried away.

In the case of this particular application, all the relevant data resides entirely on the USB thumb drive. An attacker would only need to locate the computer acting as the iSCSI initiator and pocket the thumb drive.

There is one, highly unlikely but nonetheless interesting case where this capability can be used to protect from a compromise. If the remote backups contain such highly sensitive data, that mere exposure would be utterly catastrophic, revoking access to them altogether may be desired. By removing the thumb drive and unpowering the iSCSI initiator, the only means of accessing the backups are on the small thumb drive. This device can be physically destroyed with a large hammer or a good crusher. Physical thumb drive destruction is required due to flash memory's use of wear-leveling techniques, which prevent the data from simply being overwritten directly [48]. Effectively this amounts to destroying the data, but is possible only if the imminent physical compromise is known shortly in advance. However, in environments that may ever require such extreme measures, storing encryption keys on removable tokens would be misguided in the first place.

Interestingly enough, there does not seem to have been any public research into forensic data recovery from partly destroyed or damaged solid-state memory. If such research exists, the author would very much like to know. (See also Section 3.6.)

Verdict: vulnerable; only a single, unlikely corner-case can be guarded against.

Man-in-the-middle and impersonation attacks (cornerstones targeted: security, integrity and privacy)

In a man-in-the-middle (MITM) attack, the attacker somehow places herself on the connection path, and acts as a proxy between the two parties. To succeed, this attack would require the attacker to know both parties' CHAP secrets. The secrets are issued by the iSCSI service provider and transmitted physically, either in print or stored on a USB thumb drive – they are never sent over the network. As long as the iSCSI login requires mutual authentication (both parties identify themselves to the other end) and the CHAP secrets are computationally unpredictable, MITM attacks are not possible.

Verdict: not vulnerable, unless both CHAP secrets are compromised.

Sniffing (cornerstones targeted: security and privacy)

Sniffing is, simply put, nothing fancier than reading network traffic as it goes by. The attacker needs network-level access to at least one device on the path between communicating parties.

The attacker reads and copies the traffic flowing by, trying to catch data that would be useful for more advanced attacks.

Due to the encryption of iSCSI payloads, this attack is not particularly useful. The attacker sees the iSCSI traffic, can easily find out where the reads and writes occur and therefore can also see the sizes of backups. The location of data on the iSCSI device is interesting and potentially valuable information, but basically only makes traffic analysis easier, by pointing to the attacker where on the drive any particular encrypted section is located. Without additional methods this attack is hence reduced to a regular, albeit somewhat simplified traffic analysis.

It is worth noting that even with IPsec a passive attacker sniffing the sessions could find out the sizes of write operations. This can be done by sniffing several active sessions, run-

ning statistical analysis on transmission frequencies and correlating this data with measured packet sizes.

Verdict: vulnerable, usefulness greatly reduced due to encryption.

One extraordinary type of attack deserves special examination, because of its capability to silently wreck both integrity and accessibility – data corruption as a delayed denial-of-service. The attack targets only two of the four cornerstones, but does so in a very sinister manner. Without specific countermeasures, the attack would not be noticed until the backups are needed, at which point it would be too late.

5.3.1 Delayed denial-of-service attack via traffic hijacking

The reason this type of attack works at all is that even with the encryption layer in place, only the iSCSI payloads are protected. As already seen in Figure 7, the iSCSI header and checksums are unencrypted, and the payload checksum is calculated directly from the iSCSI payload. There are two possible ways to hijack an iSCSI connection. The access required is the same as for sniffing: if the attacker can read raw network traffic, she can certainly create and inject such traffic as well.

1. The attacker hijacks live iSCSI session between the two parties with a modified Mitnick attack
    - In a Mitnick attack [66], an attacker hijacks a live TCP session and makes the client think the connection was aborted. A modified attack would hijack the route so that both the client and server send their packets through the attacker's station.
2. The attacker executes a network-level proxy attack

In fact, both attacks can be effectively reduced to the second case: the attacker only needs to control a single node on the traffic path. The attack works like this: attacker monitors the traffic passing through her station, and selectively alters some iSCSI commands and/or encrypted payloads.

iSCSI write commands contain both the starting LBA block **and** the length of data to write. By changing the location where the writes begin, the attacker can force a single write operation to corrupt the contents of all the LBA blocks between the starting point and the last block. If the attacker allows the write commands that are updating filesystem metadata to go through untouched, the data currently being written is corrupted as well. A nasty side effect of this kind of attack is that it also corrupts the filesystem itself. As a result, the initiator will very quickly realise something is wrong, and can take immediate action to remedy the situation.

A subtler way to execute the attack is to modify only the encrypted payloads, and most of all, do so very selectively. That way the filesystem metadata is always up to date and valid; only the contents of the files themselves are slightly wrong. When the target of the attack needs to resort to backups, he will find that some of them are inexplicably broken and unusable.

iSCSI's CRC checksums are designed to detect accidental traffic corruption. They do not, and can not, protect against this kind of attack; the attacker can always replace the checksum with a new one that matches the data she has just generated. Without IPsec-provided integrity protection, additional measures are required to defend the integrity of backups.

The workaround is to store cryptographic hashes of the backups locally on initiator's end, and periodically (preferably often) check that the files on iSCSI device match against the local hashes. Guarding the attack against this kind of safety measure is possible, but not really useful. It would require the attacker to store copies of the unaltered data blocks – and effectively turn her into a redundant backup service. It would be far easier to modify the iSCSI target node's contents directly and redirect reads of changed blocks to unaltered copies. That kind of attack requires, yet again, iSCSI service provider's intervention or assistance, and as such would be likely to occur in their premises altogether. As if that wasn't enough, to be successful as a denial-of-service attack, the attacker would have to know exactly when the victim is actually reaching for backups and not just checking their integrity.

Therefore, attacking the safeguard measures would not be worth the effort.

Verdict: vulnerable, workaround exists.

5.3.2 Delayed denial-of-service attack via local data corruption

This attack is fundamentally the same as the previous one, with only one major difference. Instead of an outside attacker, the hostile party is the iSCSI service provider herself.

Especially noteworthy is the fact that the use of IPsec would make absolutely no difference for this threat scenario. The iSCSI storage is, by definition, under the service provider's control. As such, they can modify the data at will, and at any time. Effectively this means that local, intentional data corruption is an off-line attack and the only way to protect against such a threat is to use the same integrity check safeguard as described in Section 5.3.1, previous to this one.

Verdict: vulnerable, workaround exists.

6 Further work

The scope of this thesis, and the accompanying implementation should be considered only a starting point. The idea of combining network block devices and partition-level encryption is somewhat novel, but only in the sense that the combination has not, apparently, been evaluated so far. However, the concept calls for further avenues of development and research.

Nonetheless, even the current prototype allows to develop new kinds of applications that take advantage of the possibilities. Apart from the backup system that I have so far outlined and proposed, there is at least one obvious and everyday application that could become immediately useful: an omnipresent and secure personal data storage.

A potentially advantageous use of easily accessible iSCSI storages would be to provide networked storage areas for mobile clients. The idea of storing frequently needed information on a network service is certainly not novel. In fact, one such a service, T-Mobile Sidekick, gained notoriety when one of their celebrity users' password got stolen and her collection of contact information consecutively was plastered around the dark corners of the Internet [67].

Such events underline both the advantages and dangers of on-line storages. Having frequently used data available practically everywhere is certainly compelling, but a single password warding access to all of the data is not enough to protect it from outsiders.

With the advent of Linux-based PDA's and even smart phones, it would be possible to implement secure, omnipresent storage. As a protocol, iSCSI is lightweight enough, and partition-level encryption is natively supported by the OS via dm-crypt. The really interesting part is that such a bundle would turn the PDA partially into a Personal Trusted Device, or PTD for short [68]. Conversely, if included in a PTD, such a bundle would expand the capabilities of an already powerful concept.

As far as further research is needed, the topic circles around iSCSI, both as a protocol and as the storage service. The idea of securing iSCSI transports with IPsec is a no-brainer and is even recommended by the iSCSI specification [27] itself.

However, the specific implementation outlined in this thesis adds some rather unexpected factors. Since the basic assumption is that the iSCSI service provider is either insecure,

hostile, or both, re-encrypting the traffic on IPsec level makes little sense. As mentioned in Section 5.3, the biggest advantage in this scenario would stem from IPsec's use of cryptographically strong integrity protection, and due to the extraordinary setup, could safely operate in null encryption mode. This would still effectively block the delayed-denial-of-service hijack attack examined in Section 5.3.1. It would also allow us to drop iSCSI's own CRC checks, since IPsec's hash-based integrity protection far exceeds the level of protection offered by any CRC checksums.

Usually IPsec is associated with VPN technology and general traffic encryption. However, since the actual data payloads are already separately encrypted, applying another layer of encryption would only serve to protect the iSCSI commands. And since the iSCSI service provider can still inspect the plain traffic after it has been processed by IPsec, full-blown IPsec would add little to no security.

Also, re-encrypting the entire payload would add IPsec's full overhead. This was not implemented, so the exact impact remains unknown. It could be a subject for a separate study. Setups that require both maximum security and top-notch performance may even consider using a dedicated IPsec gateway device.

Another obvious research target would be a simple iSCSI target allocation and management tool. In order to provide large-scale (or even small-scale) iSCSI service, the service providers need proper tools to manage their iSCSI target allocations – and especially their users. For use patterns described in this thesis, there should be no complex needs. Also, as far as tools go, less is more and simpler is better.

Publicly available iSCSI services could, and likely will drive development of further applications. To cater for the future uses, more complex management back-ends will eventually emerge. In the meantime, some simple but clean tools are needed to manage iSCSI user accounts and their respective storage areas. One final aspect is that extended use of iSCSI calls for performance evaluation of both different target and initiator implementations on multiple Operating Systems as well as on varying hardware platforms. It can be expected that in networks that exceed the speed of 1Gbit/s, iSCSI performance has implementation-specific upper bounds. Whatever these bounds are, they will have adverse effects on deployment plans: individual iSCSI targets can each serve a multitude of initiators and compound traffic will eventually reach the upper performance limit of any single target.

7 Conclusions

As expected, using a a network block device (iSCSI) in combination with partition-level encryption works without glitches. The real surprise was that the construction was even easier to set up and use than I ever surmised. The setup, once in effect, is completely transparent to both processes and human users. By selecting proper encryption parameters, the system is also reasonably secure against all practical threats – not to mention that the security margin should be high enough to deter even most of the improbable and impractical attacks as well.

The major outcome is that it is both possible *and practical* to build a transparent backup system that is simple to use, and yet secure even in the presence of untrusted or outright hostile service provider. This very scenario, which is the one my thesis has attempted to solve from the start, has some unexpected properties. If the service provider is assumed untrusted, using IPsec to secure the iSCSI traffic becomes pointless. The foremost threat is directed, not toward the data in transit, but to the physical storage. IPsec could only protect the transmission. However, a true paradox in the setup is that due to its unorthodox nature, employing IPsec without encryption (usually a huge security hole) would in fact increase the overall security of the system.

The choice of the filesystem that is used on the remote, encrypted partition is far from irrelevant. Counter-intuitively, the filesystem should be as simple and scarce on features as possible. Using a more complex filesystem yields potential attackers additional information, although whether that information is actually useful remains an open issue. A more concrete effect is that complex filesystems issue multiple metadata updates, each of them requiring separate iSCSI commands. By its very nature, iSCSI instills network-related latency. Although not really an issue with this particular implementation, additional filesystem-specific commands introduce further latency if working with lots of files.

Last of all, when the proposed backup system is up and running, it is unimportant who controls the iSCSI service – or what their interests toward the stored data are. The worst may have already happened and the service provider has been acquired by our direct competitor. However, the backups remain secure. An inconvenience that can materialise is that we may be required to migrate to another iSCSI service provider. The backup system continues to function with a simple change of configuration – and most importantly, the day-to-day business operations are not interrupted.

Acknowledgements

The following additional resources were used in creation of this thesis:

Open Clip Art Library

Location: `http://www.openclipart.org/`

Author	Works used
Benji Park	Green and purple arrows Placed in Public Domain
Andy Fitzsimon	Hard drive Placed in Public Domain
Havok_redhand.pl	Microchip Placed in Public Domain (Image is possibly automatically generated by a script, original author is unknown)
A.J. Ashton	Lock Placed in Public Domain
Francesco Rollandin	Key Placed in Public Domain
Nicu Buculei	Network cloud Placed in Public Domain

BibTeX databases of IETF's RFC documents and drafts

Location: `http://www.tm.uka.de/~bless/rfc.bib`,
`http://www.cs.columbia.edu/~hgs/bib/i-d.bib`

References

[1] US Internet Corp. Secure remote data backup & disaster recovery service, 2005. URL http://www.zstorage.com/solutions/deployment_process.htm. Visited on 2006-06-14.

[2] LCC Byte Fortress Technologies. Datavault pro overview, 2006. URL http://www.bytefortress.net/datavault_pro.html. Visited on 2006-06-14.

[3] LLC Data Protection Services. About our solution / how it works, 2006. URL http://www.dataprotection.com/secure-online-backup/how-secure-online-backup-works/. Visited on 2006-06-14.

[4] Techfusion. Offsite backup, 2006. URL http://www.techfusion.com/remotebackup.htm. Visited on 2006-06-14.

[5] Vembu Technologies Pvt. Ltd. A flexible data backup software that works with your existing hardware, 2006. URL http://www.vembu.com/. Visited on 2006-06-14.

[6] Kellen. Unattended, encrypted, incremental network backups: Part 1, August 2005. URL http://www.debian-administration.org/articles/209. Visited on 2006-06-14.

[7] Carlos Justiniano. Automate backups on linux. URL http://www-128.ibm.com/developerworks/linux/library/l-backup/?ca=dgr-lnxw41Backup. Visited on 2006-06-14.

[8] Brice Burgess. Making secure remote backups with rsync, November 2004. URL http://servers.linux.com/servers/04/11/04/0346256.shtml?tid=119&tid=47. Visited on 2006-06-14.

[9] Ben Escoto. Encrypted bandwidth-efficient backup using the rsync algorithm, August 2005. URL http://duplicity.nongnu.org/. Visited on 2006-06-14.

[10] Sun Microsystems. NFS: Network File System Protocol specification. RFC 1094 (Informational), March 1989. URL http://www.ietf.org/rfc/rfc1094.txt.

[11] Sun Microsystems. Company information: Sun history, 2006. URL http://www.sun.com/aboutsun/coinfo/history.html. Visited on 2006-06-14.

[12] Microsoft Corp. and Intel Corp. Microsoft networks/opennet file sharing protocol, intel part number 138446, document version 2.0. Technical report, Microsoft, Intel, 1988.

[13] Paul Leahy and Dan Perry. Cifs: A common internet file system. *Microsoft Internet Developer*, November 1996.

[14] Ben Escoto. A remote incremental backup of all your files, January 2006. URL `http://www.nongnu.org/rdiff-backup/`. Visited on 2006-06-15.

[15] Jeff Bailey, Paul Eggert, Sergey Poznyakoff, John Gilmore, and Thomas Bushnell. Gnu tar: an archiver tool, December 2004. URL `http://directory.fsf.org/tar.html`. Visited on 2006-06-15.

[16] Werver Koch, Matthew Skala, Michael Roth, Niklas Hernaeus, and Rémi Guyomarch. The gnu privacy guard, April 2006. URL `http://www.gnupg.org/`. Visited on 2006-06-15.

[17] T. Ylonen and C. Lonvick. The Secure Shell (SSH) Protocol Architecture. RFC 4251 (Proposed Standard), January 2006. URL `http://www.ietf.org/rfc/rfc4251.txt`.

[18] J. Galbraith, T. Ylonen, and S. Lehtinen. SSH file transfer protocol. Internet Draft draft-ietf-secsh-filexfer-05, Internet Engineering Task Force, February 2004. URL `http://www.ietf.org/internet-drafts/draft-ietf-secsh-filexfer-05.txt`. Work in progress.

[19] John Ouellette. Paranoid penguin: managing ssh for scripts and cron jobs. *Linux J.*, 2005(137):13, 2005. ISSN 1075-3583.

[20] America On-Line. XDrive - The Internet Hard Drive, 2006. URL `http://www.xdrive.com/`. Visited on 2006-08-29.

[21] Philipp Lenssen. Google Gdrive Surfaces, July 2006. URL `http://blog.outer-court.com/archive/2006-07-10-n48.html`. Visited on 2006-08-29.

[22] Mary Jo Foley. Microsoft Readies Storage Service to Rival Google's 'Gdrive', April 2006. URL `http://www.microsoft-watch.com/article2/0,2180,1951172,00.asp`. Visited on 2006-08-29.

[23] Priya George. Microsoft Set To Compete with Google, AOL, Yahoo in Online Storage Space, August 2006. URL `http://www.sda-india.com/sda/`

features/psecom,id,546,nodeid,1,_language,India.html. Visited on 2006-08-29.

[24] R. Weber, M. Rajagopal, F. Travostino, M. O'Donnell, C. Monia, and M. Merhar. Fibre Channel (FC) Frame Encapsulation. RFC 3643 (Proposed Standard), December 2003. URL http://www.ietf.org/rfc/rfc3643.txt.

[25] M. Rajagopal, E. Rodriguez, and R. Weber. Fibre Channel Over TCP/IP (FCIP). RFC 3821 (Proposed Standard), July 2004. URL http://www.ietf.org/rfc/rfc3821.txt.

[26] C. Monia, R. Mullendore, F. Travostino, W. Jeong, and M. Edwards. iFCP - A Protocol for Internet Fibre Channel Storage Networking. RFC 4172 (Proposed Standard), September 2005. URL http://www.ietf.org/rfc/rfc4172.txt.

[27] J. Satran, K. Meth, C. Sapuntzakis, M. Chadalapaka, and E. Zeidner. Internet Small Computer Systems Interface (iSCSI). RFC 3720 (Proposed Standard), April 2004. URL http://www.ietf.org/rfc/rfc3720.txt. Updated by RFC 3980.

[28] Pavel Machek and Wouter Verhelst. Network Block Device, 1999. URL http://nbd.sourceforge.net/. Visited on 2006-06-16.

[29] S. Hopkins, B. Coyle, and Inc. The Brantley Coile Company. AoE (ATA over ethernet), August 2004. URL http://www.coraid.com/documents/AoEr8.txt. Visited on 2006-08-11.

[30] Christophe Saout. dm-crypt: a device-mapper crypto target, 2005. URL http://www.saout.de/misc/dm-crypt/. Visited on 2006-06-14, badly out of date.

[31] TrueCrypt Foundation. Free Open-Source On-The-Fly Encryption, 2006. URL http://www.truecrypt.org. Visited on 2006-06-20.

[32] Lucky Green. Encrypting disk partitions, 2006. URL http://www.freebsd.org/doc/en_US.ISO8859-1/books/handbook/disks-encrypting.html. Visited on 2006-06-14.

[33] Roland C. Dowdeswell and John Ioannidis. The cryptographic disk driver. In *USENIX 2003 Annual Technical Conference, FREENIX Track*. USENIX Association, 2003.

[34] OpenBSD. *OpenBSD Programmer's Manual, vnd - Vnode Disk Driver*, December 1995. URL http://www.openbsd.org/cgi-bin/man.cgi?query=vnd&sektion=4. Visited on 2006-06-14.

[35] OpenBSD. *Openbsd programmer's manual, vnconfig - configure vnode disks for file swapping or pseudo file systems*, June 1993. URL http://www.openbsd.org/cgi-bin/man.cgi?query=vnconfig&sektion=8. Visited on 2006-06-14.

[36] SubRosaSoft.com Inc., 2006. URL http://www.subrosasoft.com/OSXSoftware/index.php?main_page=product_info&products_id=33. Visited on 2006-06-20.

[37] Michal Ludvig. Via padlock-wicked fast encryption. *Linux J.*, 2005(133):4, 2005. ISSN 1075-3583.

[38] FIPS. Announcing the Advanced Encryption Standard (AES). Technical Report 197, FIPS, November 2001.

[39] W. Simpson. PPP Challenge Handshake Authentication Protocol (CHAP). RFC 1994 (Draft Standard), August 1996. URL http://www.ietf.org/rfc/rfc1994.txt. Updated by RFC 2484.

[40] G. Zorn. Microsoft PPP CHAP Extensions, Version 2. RFC 2759 (Informational), January 2000. URL http://www.ietf.org/rfc/rfc2759.txt.

[41] B. Aboba, J. Tseng, J. Walker, V. Rangan, and F. Travostino. Securing Block Storage Protocols over IP. RFC 3723 (Proposed Standard), April 2004. URL http://www.ietf.org/rfc/rfc3723.txt.

[42] Joan Daemen and Vincent Rijmen. *The design of Rijndael: AES — the Advanced Encryption Standard*. Springer-Verlag, 2002.

[43] Bruce Schneier. *Applied Cryptography: Protocols, Algorithms, and Source Code in C*, pages 193–194. John Wiley & Sons, Inc., New York, NY, USA, 2nd edition, 1996. ISBN 0-471-12845-7.

[44] FIPS. Announcing the Secure Hash Standard. Technical Report 180-2, FIPS, August 2002.

[45] Bruce Schneier. *Applied Cryptography: Protocols, Algorithms, and Source Code in C*, pages 165–166. John Wiley & Sons, Inc., New York, NY, USA, 2nd edition, 1996. ISBN 0-471-12845-7.

[46] Clemens Fruhwirth. New Methods in Hard Disc Encryption. Technical report, Vienna University of Technology, June 2005. URL http://clemens.endorphin.org/nmihde/nmihde-A4-os.pdf. Visited on 2006-07-17.

[47] Markku-Juhani Olavi Saarinen. Encrypted watermarks and linux laptop security. In Chae Hoon Lim and Moti Yung, editors, *Information Security Applications*, volume 3325 of *Lecture Notes in Computer Science*, pages 27–38. Springer Berlin / Heidelberg, 2005. ISBN 978-3-540-24015-0. doi: 10.1007/b103174.

[48] Eran Gal and Sivan Toledo. Algorithms and data structures for flash memories. *ACM Comput. Surv.*, 37(2):138–163, 2005. ISSN 0360-0300. doi: http://doi.acm.org/10.1145/1089733.1089735.

[49] *Secure Deletion of Data from Magnetic and Solid-State Memory*, July 1996. USENIX. URL http://www.cs.auckland.ac.nz/~pgut001/pubs/secure_del.html.

[50] FUJITA Tomonori. iSCSI Enterprise Target, 2006. URL http://iscsitarget.sourceforge.net. Visited on 2006-06-16.

[51] M. Bakke, J. Hafner, J. Hufferd, K. Voruganti, and M. Krueger. Internet Small Computer Systems Interface (iSCSI) Naming and Discovery. RFC 3721 (Informational), April 2004. URL http://www.ietf.org/rfc/rfc3721.txt.

[52] Alex Aizman and Dmitry Yusupov. iSCSI initiator implementation for Linux, 2006. URL http://open-iscsi.org. Visited on 2006-06-22.

[53] P. Leach, M. Mealling, and R. Salz. A Universally Unique IDentifier (UUID) URN Namespace. RFC 4122 (Proposed Standard), July 2005. URL http://www.ietf.org/rfc/rfc4122.txt.

[54] Todd Miller, Chris Jepeway, Aaron Spangler, Jeff Nieusma, and Dave Hieb. Sudo - superuser do. URL http://www.sudo.ws/sudo/sudo.html. Visited on 2006-08-18.

[55] D. Eastlake 3rd and P. Jones. US Secure Hash Algorithm 1 (SHA1). RFC 3174 (Informational), September 2001. URL http://www.ietf.org/rfc/rfc3174.txt.

[56] Felix Fuentes and Dulal C. Kar. Ethereal vs. tcpdump: a comparative study on packet sniffing tools for educational purpose. *J. Comput. Small Coll.*, 20(4):169–176, 2005.

[57] DEW Associates Corporation. Logical Block Addressing, 2002. URL http://www.dewassoc.com/kbase/hard_drives/lba.htm. Visited on 2006-08-28.

[58] Rémy Card, Theodore Ts'o, and Stephen Tweedie. Design and Implementation of the Second Extended Filesystem. In *Proceedings of the First Dutch International Symposium on Linux*, September 1994. ISBN 90-367-0385-9. URL http://web.mit.edu/tytso/www/linux/ext2intro.html. Visited on 2006-08-30.

[59] Juri Haberland. Linux ext3 FAQ, October 2004. URL http://batleth.sapienti-sat.org/projects/FAQs/ext3-faq.html. Visited on 2006-08-30.

[60] Adam Sweeney. *xFS Transaction Mechanism*. Silicon Graphics Inc., October 1993. URL http://oss.sgi.com/projects/xfs/design_docs/xfsdocs93_pdf/trans.pdf.

[61] Yee Wei Law, Jeroen Doumen, and Pieter Hartel. Survey and benchmark of block ciphers for wireless sensor networks. *ACM Trans. Sen. Netw.*, 2(1):65–93, 2006. ISSN 1550-4859. doi: http://doi.acm.org/10.1145/1138127.1138130.

[62] Arjen Lenstra. Progress in hashing cryptanalysis, September 2004. URL http://cm.bell-labs.com/who/akl/hash.pdf. Visited on 2006-08-18.

[63] Christophe De Cannière and Christian Rechberger. SHA-1 collisions: Partial meaningful at no extra cost?, August 2006. URL http://www.iacr.org/conferences/crypto2006/rumpsched.html. Visited on 2006-08-28.

[64] IAIK Krypto Group. Collision for 64-step SHA-1, August 2006. URL http://www.iaik.tugraz.at/research/krypto/collision/index.php. Visited on 2006-08-28.

[65] Heise Security. SHA-1 hash function under pressure, August 2006. URL http://www.heise-security.co.uk/news/77244. Visited on 2006-08-28.

[66] J. P. McDermott. Attack net penetration testing. In *NSPW '00: Proceedings of the 2000 workshop on New security paradigms*, pages 15–21, New York, NY, USA, 2000. ACM Press. ISBN 1-58113-260-3. doi: http://doi.acm.org/10.1145/366173.366183.

[67] Lucy Sherriff. Paris Hilton's Sidekick hacked, February 2005. URL http://www.theregister.co.uk/2005/02/21/paris_hacked/. Visited on 2006-07-25.

[68] Jari Porras, Pekka Jäppinen, Petri Hiirsalmi, Arto Hämäläinen, Sami Saalasti, Raine Koponen, and Satu Keski-Jaskari. Personal Trusted Device in Personal Communications. In *Proceedings of 1st International Symposium on Wireless Communication Systems*, Mauritius, September 2004.

Appendix 1: Glossary

Birthday paradox Name given to a statistical phenomenon, that finding two identical values in any given set is is much more likely than finding a pair for any predetermined value from the same set.

Block device A hardware device which is accessed a fixed-size block at a time. For example, a hard drive, CD-ROM or a removable storage media.

Ciphertext Encrypted data (or a block thereof.)

Delta Difference between two similar sets. Comparison of one set and a delta allows to construct the second set.

Entropy Measurable uncertainty or statistical noise. In other words, randomness.

Hash A cryptographic checksum. A fixed-length sequence of bits that identifies any given block of data; calculating a hash of any input is easy, but reversing the process is considered computationally infeasible, if not impossible. Sometimes also referred to as a digital fingerprint.

Heterogeneous network A network of computers which consists of multiple different Operating Systems and their versions.

Initialisation Vector Initial value used for encryption modes. Needed to scramble the first encryption block, in order to prevent certain cryptanalytic attacks.

Inode Location of a file, and a container for its properties on a *nix filesystem.

Logical Block Addressing Address mode for all present-day hard drives, where the drive is conceptually divided into fixed-size, sequential blocks. Reading or writing requires only the knowledge of the number of the block where the data begins.

Message digest See hash.

Null encryption In protocols using cryptography; omitting the encryption and transmitting the payloads in clear. Usually used only for testing correctness of protocol implementation.

Off-line attack An attack that does not require an active session or any traffic to flow between the parties involved. Attacks are done with, and targeted against the collected data.

Partition-level encryption Setup where all file system operations are done through a cryptographic translation layer. Anything written to the drive partition is encrypted on the fly, and anything read from the partition is similarly decrypted. In effect, even the filesystem system metadata is encrypted. Without the correct encryption setup (algorithm, key and encryption mode) the partition appears to contain nothing but illegible garbage. Sometimes also called sector-level encryption.

Plaintext Unencrypted or decrypted data (or a block thereof.)

Portal Group Tag iSCSI supports multiple sessions over the same connection. There may be several such connections, each dealing with different storage areas; separating these connections from one another is therefore needed. Portal Group Tag is the identifier for this separation.

RAID-1 Storage solution where a drive is transparently duplicated to at least one other drive. Visible size is that of a single drive. Malfunction of one drive does not prevent the RAID array from operating, but until the defective drive is replaced, redundancy factor is decreased by the number of defective drives. Also known as mirror RAID.

Appendix 2: Performance measurements

Device speeds measured with hdparm

Run	Unencrypted drive (MB/s)	Encrypted partition (MB/s)
1	45.68	11.61
2	45.64	18.21
3	46.40	12.31
4	45.95	17.14
5	46.84	11.22
6	47.52	19.47
7	47.98	11.65
8	41.18	19.41
9	46.92	18.16
10	46.22	15.29
Average	**46.03**	**15.45**

Table 3: Buffered read speeds for a local hard drive with and without dm-crypt

Run	Unencrypted drive (MB/s)	Encrypted partition (MB/s)
1	9.26	8.33
2	9.30	8.34
3	9.32	7.69
4	9.32	7.62
5	9.33	7.57
6	9.33	7.60
7	9.35	8.52
8	8.81	7.58
9	9.36	8.51
10	9.29	9.77
Average	**9.27**	**8.15**

Table 4: Buffered read speeds for iSCSI device with both HeaderDigest and DataDigest enabled

Run	Unencrypted drive (MB/s)	Encrypted partition (MB/s)
1	9.32	8.15
2	9.36	7.59
3	9.40	8.28
4	9.36	7.98
5	9.37	8.33
6	9.33	7.98
7	9.37	8.35
8	9.37	8.26
9	9.39	7.61
10	9.36	7.58
Average	**9.36**	**8.01**

Table 5: Buffered read speeds for iSCSI device with both HeaderDigest and DataDigest disabled

Device speeds measured with real files

Run	Time (seconds)
1	11.7
2	8.7
3	9.2
4	11.7
5	9.2
6	11.6
7	11.5
8	11.7
9	11.8
10	9.4
Average	**10.65**

Table 6: Copying 250MB of data to an encrypted local partition. Average write speed in this case was 23.47MB/s.

The data in following test runs consisted of 323 files and 22 directories.

Run	None	Header	Data	Both
1	217.0	221.5	221.6	221.6
2	223.2	218.7	219.9	224.5
3	221.3	218.7	223.9	224.7
4	217.5	221.7	222.1	221.0
5	218.4	217.3	~~225.6~~	219.9
6	220.1	~~226.6~~	222.2	~~219.0~~
7	~~216.8~~	218.0	223.0	224.2
8	~~223.5~~	217.6	220.3	222.6
9	221.9	217.8	224.0	~~228.3~~
10	218.5	~~217.0~~	~~219.8~~	220.3
Average	**219.7**	**218.9**	**222.1**	**222.4**
Speed (MB/s)	**10.22**	**10.26**	**10.11**	**10.09**

Table 7: Copying 2245MB of data to an iSCSI target, with all combinations of iSCSI digest options. Times are in seconds. Average is calculated by omitting the smallest and largest value from each set. The omitted values have been struck out.

Run	None	Header	Data	Both
1	216.0	219.4	230.1	~~237.7~~
2	~~216.0~~	217.9	238.1	224.0
3	~~224.9~~	221.1	221.7	230.9
4	216.7	216.5	~~218.5~~	231.9
5	219.4	~~216.0~~	~~238.5~~	223.9
6	217.5	218.5	231.4	~~223.3~~
7	218.6	~~224.6~~	228.6	232.7
8	219.0	218.0	225.6	230.5
9	216.4	219.0	222.7	231.7
10	221.4	218.9	223.6	229.8
Average	**218.1**	**218.7**	**227.7**	**229.4**
Speed (MB/s)	**10.29**	**10.27**	**9.86**	**9.77**

Table 8: Copying 2245MB of data to an iSCSI target, with encryption layer placed on top of iSCSI. All combinations of iSCSI digest options are again tested. Times are in seconds. Average is calculated by omitting the smallest and largest value from each set. The omitted values have been struck out.

Appendix 3: CD-ROM and its contents

The CD-ROM media distributed with the bound thesis contains the following directory hierarchy:

```
.
+- Makefile
+- dippa-bostik.lyx
+- dippa-bostik.pdf
+- template_dippa.lyx
+- bib/             (bibliography files)
+- data/            (traffic dumps)
+- images/          (image files)
  +- construction/ (illustration files)
+- src/             (prototype implementation)
  +- old/           (early test scripts)
```