`cryptar`: Secure, Untrusting, Differencing Backup

Jeff Abrahamson
Department of Computer Science
Drexel University, Object Recognition and Applied Algorithms Lab
Philadelphia, PA
`jeffa@cs.drexel.com`; `http://www.cs.drexel.edu/~jeffa/`

Adam J. O'Donnell*
Department of Electrical and Computer Engineering
Drexel University, Computer Communications Laboratory
Philadelphia, PA
`adam@ece.drexel.com`; `http://www.ece.drexel.edu/CCL/odonnell/`

Abstract

We present an algorithm and its implementation to perform secure incremental backups to untrusted servers. The algorithm avoids unencrypted data leaving the local host, but is nonetheless able to compute a set of differences between the encrypted data on the remote server and the changed local files.

The `cryptar` algorithm is derived from the rsync algorithm. It is conceptually similar to running rsync with the remote image encrypted.

1 Introduction

Traditional backup techniques focus on writing a set of files (or a subset in the case of incremental backups) to tape or other removable media that may then be kept in a safe place. Incremental backups typically write the complete content of each changed file to removable media.

Inexpensive local storage and storage-intensive non-text content (audio, video, images) have made this practice more expensive: a typical user has much more data than a few years ago. A small business or individual may easily have such a large quantity of data on a $100 hard drive that tape storage becomes more expensive than the original hard drive.

*This work was supported in part by the United States National Science Foundation Graduate Research Fellowship.

Moreover, proper off-line storage practices require a second, remote site, something few people do in practice.

For these reasons we have developed a new remote backup tool called `cryptar`, which allows a remote computer and its hard drive to be used for secure full or incremental backup. The key innovation in `cryptar`, however, lies in keeping all data in the remote store encrypted, while still being able to compute efficient deltas so as to conserve network resources.

We do this by maintaining a very small amount of local meta-data, about one hundred bytes per file, while storing nearly all other information remotely. We will describe the precise details in section 3.2 ff.

1.1 Why `cryptar` is Interesting

The principal historical problems related to remote on-line storage have been bandwidth and security. Most people who consider backup at all, though, now have low-latency broadband connections that make maintaining an off-site backup feasible.

This leaves us, however, with concern for the safety of data stored remotely. Unless one completely trusts the remote machine, a serious concern remains that the remotely stored data may be perused by a curious (or evil) person with access to the remote machine.

`cryptar` solves the data integrity and privacy problems by encrypting the remote content before transmission while storing a small signature locally. Indeed, cryptar stores each local file remotely in encrypted blocks, then keeps a list of hash signatures of those blocks to permit comparing blocks without needing direct access to them.

cryptar is thus able to compute the necessary updates without extensive access to the remote and encrypted data. This permits efficient low-bandwidth updates without allowing unencrypted remote data.

1.2 Trust

If we trust the remote site completely, we can use rsync snapshotting [Rubel] or CVS. We would like to imagine a world, however, in which we are not obliged to find secure machines for our offsite storage needs. Already we can purchase inexpensive disk space from many ISP's. Except for the questions of trust and efficiency of data transfer, we could use this space for our backups.

The principal problem with maintaining backup data on insecure servers, however, is the risk that the information is read by unauthorized parties or even altered without our knowledge. We can solve the first of these problems by encrypting the data, and the second by signing the data we store. But encrypted backups typically do not admit efficient delta computations, since similar files are generally dissimilar once encrypted.

The problem we solve with cryptar assumes that secure and authenticated data transport is easy (e.g., IPsec or ssh), and that computational power is cheaper than bandwidth. (If bandwidth is cheaper, we can simply transfer full encrypted files without concern for efficient deltas.)

In cryptar we encrypt blocks before transfer to the remote archive while keeping block signatures locally.

Because the remotely stored blocks are encrypted locally before transfer, only encrypted data leaves the local host. A remote intruder would find in the archive only encrypted data, useless up to the strength of the encryption scheme.

In addition, the local store maintains a set of SHA-1 hashes for the remote data. Any data retrieved is therefore compared against the local hashes to authenticate it. A more detailed discussion of the strength of this assurance resides in section 3.7.

1.3 How cryptar is Different

cryptar's principal innovation is the combination of keeping remote data encrypted and yet being able to compute differences efficiently.

Rsync, for example, which is discussed in section 3.1, computes differences efficiently, but must have access to unencrypted remote data. A number of programs encrypt remote data but are incapable of computing an efficient delta.

Indeed, the poor man's solution to encrypted remote backup has been to encrypt data to tape or to a remote machine by simply piping the output of the unix tar program through an encryption program (cf. section 2).

We will discuss related work to further understand how cryptar is different from other cryptographic and archive software and algorithms. After a brief discussion of trust and who we do and don't trust, we discuss the actual algorithm in depth. Finally, we'll discuss security, the cryptar protocol, and make some short notes on future directions.

2 Related Work

A venerable solution under unix-like operating systems is `tar | gpg | dd` (with various options to make this make sense)[1]. This effectively encrypts the remote data before it leaves the local machine, but provides no ability to compute deltas or to update the remote store without recomputing the entire remote image. Using find or even tar, incremental backups are possible, but only in the sense that an entire file that has changed since the last backup can be re-encoded. We are concerned here that the files we recover are what we originally stored: this scheme provides no such guarantee.

It's interesting to note that the following ad hoc remote archive scheme fails: For each file foo, encrypt foo to foo.crypt, rsync foo.crypt to the remote server, then delete foo.crypt locally. Remote files are always encrypted, and one could easily add signing to this scheme. Unfortunately for this scheme, encryption preserves exceedingly little structure, and so rsync cannot compute a delta smaller than the file itself. Rsync is thus

[1]gpg is GNU Privacy Guard, an OpenPGP compliant crypto program. The UNIX program dd copies files.

forced to copy the entire file, which makes this scheme too inefficient for our needs.

The rsync program allows for incremental backup, albeit via a somewhat cumbersome interface. RIBS, the rsync incremental backup system (`http://rustyparts.com/scripts.php`), provides a nicer interface around incremental backup with rsync, but the remote files remain unencrypted. No certificate guarantees that a recovered file is correct. The rsync-backup program (`http://www.stearns.org/rsync-backup/`) and Scylla Charybdis (`http://www.scylla-charybdis.com/`) suffer the same limitations.

Duplicity (`http://www.nongnu.org/duplicity/`) stores GnuPG-encrypted tar archives on one or more remote machines. It uses GnuPG to sign the archives in order to know that what it gets back is what it sent. On the other hand, it maintains a great deal of data in its (local) incremental storage space, and the size of the backup can far exceed the size of the original files if the files change often.

The Distributed Internet Backup System, DIBS (`http://www.csua.berkeley.edu/~emin/source_code/dibs/`) maintains GnuPG signed and encrypted archives on one or more remote servers. It uses Reed-Solomon codes to distribute data over multiple servers. When a file changes, however, the entire file must be retransmitted.

A number of P2P systems offer superficially similar services. Chord/CFS (`http://www.pdos.lcs.mit.edu/chord/`) and Mnet (`http://mnet.sourceforge.net/`), for example, both provide encrypted network storage. These P2P systems, however, do not compute efficient differences.

Plan 9's Venti block store [Quinlan] uses a SHA-1 indexed block-based differencing scheme to store files. The goal of Venti (enterprise file storage) is quite different from rsync or `cryptar`, but the block-based differencing scheme is similar.

The Self-Certifying File System (`http://www.fs.net/`), SFS, operates much like NFS, but encrypts its traffic. The user need not trust the network, but must trust the remote host with his unencrypted data, since the encryption only covers transport, not storage.

3 Algorithm

It is useful to begin with a brief explanation of the rsync algorithm. We will then describe the `cryptar` algorithm in overview, define some terms useful for a more detailed analysis of `cryptar`, then describe the specific steps involved in archive and restore operations. We will conclude our discussion of the algorithm with discussions of the signature algorithms, our confidence in those signatures, and the cryptographic framework for remote block protection.

3.1 The Rsync Algorithm (Overview)

Rsync is a client-server binary differencing algorithm designed for contexts where bandwidth is more expensive than computing time. The goal is to transform a remote file such that it is identical to a local file. It uses checksums to determine where changes have occurred.

Rsync depends on an easily computed 32-bit checksum, called the weak checksum, and a stronger but more expensive cryptographic hash called the strong checksum.

Given the two files, the client requests that the server compute both the weak and the strong checksums of each B-byte non-overlapping block of the remote file R. That is, it computes the checksums of the byte range $(0, B-1)$, of $(B, 2B-1)$, and so forth. The window size B is a constant of the algorithm, although Tridgell [Tridgell2] discusses theoretical and practical optimal ranges for this constant.

The server sends this signature (the set of weak and strong checksums) to the client. The client computes *every* weak checksum of the local file. That is, it computes weak checksums for intervals $(0, B-1)$, $(1, B)$, $(2, B+1)$,$(3, B+2)$, etc., see figure 1. The client then uses a hash of these weak checksums to find candidate matches to the server side checksums.

The client computes a block's strong checksum only to confirm a weak checksum hit. A weak checksum miss is clear proof that the blocks are not identical.

Knowing which blocks are identical between the local and the remote files, the client can instruct the server to move blocks as needed, transmitting to the server only that data that is not already represented in the remote file according to this block view. The process is symmetric for file recovery.

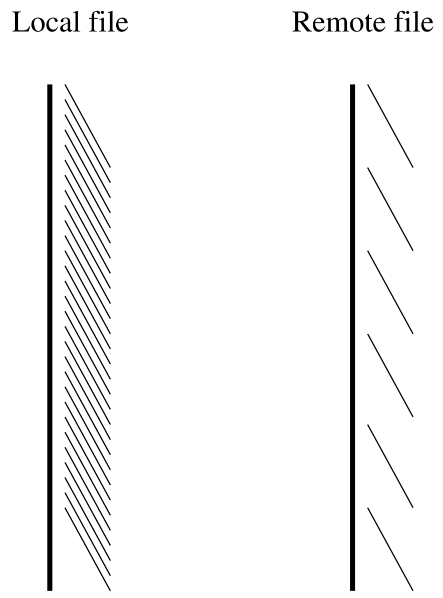

Figure 1: In rsync the remote file (here the thick line on the right) is covered by disjoint blocks (diagonal lines) while the local file (the thick line on the left) is covered by blocks at all offsets. In `cryptar` the same holds, but the remote file's covering was stored in the database at the time of archive, and so is recovered from the database rather than computed from an actual remote file.

3.2 The `cryptar` Algorithm (Overview)

`cryptar` uses a split database to simulate a remote server with access to remote files. The remote `cryptar` process acts merely as a block server.

As in rsync, a local `cryptar` process establishes communication with a remote `cryptar` process whose only difference is the method of invocation: the remote process runs in server mode. The details of communicating are at the user's discretion, but might typically involve an ssh connection.

The essential idea behind the `cryptar` algorithm is to store encrypted data remotely, but only a small amount of meta-data locally needed for interacting with the remote. We thus store in the local database a signature as well as a small amount of bookkeeping information.

Of course, while the purpose of rsync is compressed copy, the purpose of `cryptar` is secure archive. The remote file is actually virtual: it exists as an image in a split database, the remote database being encrypted.

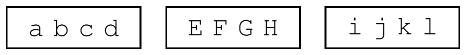

Figure 2: A full minimal covering

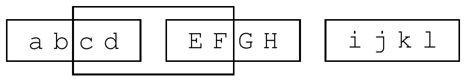

Figure 3: A full non-minimal covering

Said another way, the local database stores enough information to simulate the response of an rsync server if we were running rsync. This local meta-data is what allows the remote data to be kept encrypted.

The client may thus retrieve the virtual remote file's signature from the database, compute the delta, and store the delta in the database. By storing almost all the information encrypted remotely, the local database can remain small enough to avoid burdening the local system's storage, but still hold enough information to permit us to avoid querying the remote database unless a file has in fact changed.

3.3 Definitions

We call the local set of files which we want to archive the *fileset*. The data that we store remotely we call the *remote (archive) database*.

A *covering* of a file is a set of possibly overlapping pieces of the file that, taken together, completely specify the file's contents. A *partial covering* is a set of possibly overlapping pieces of the file that fail to specify the entire file (i.e., that leave "holes" in the file). We will only concern ourselves with coverings using fixed size blocks for efficiency reasons.

To illustrate coverings, consider a 12 byte file (string) "abcdEFGHijkl". An example of a minimal covering (Figure 2) would be bytes 0–3 ("abcd"), bytes 4–7 ("EGFH"), and bytes 8–11 ("ijkl"). Figure 3 shows a non-minimal covering of the same file.

If we insert two bytes after byte 2 ("abcxydEFGHijkl"), the last two blocks above only form a partial covering (figure 4): "EFGH" and "ijkl" cover part of the file, but "abcxyd" is uncovered. We can add two more blocks, "abcx" and "ydEF" to form a full covering, as in figure 5.

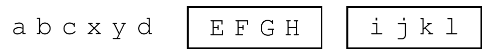

Figure 4: A partial covering

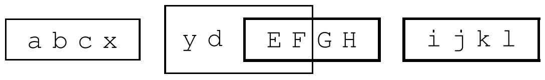

Figure 5: Two partial coverings whose union is a full covering

In section 3.5 we will describe a scheme for computing and storing weak checksums and strong checksums for each file. We call the set of weak and strong checksums of the blocks in the covering the *block list*. The statistical summary of the file (its length, a SHA-1 hash of the file, and its modification date) we call the *summary signature*. The summary signature and the block list together we will sometimes call the *signature* of that file at a given time. The *local database* stores the summary signature; the block list we store (compressed and encrypted) remotely. Table 2 shows an example of a signature.

Local DB	Remote DB
Filenames	File blocks
Summary signataure	Block list
(Keys)	

Table 1: Summary of what is stored where. Note that the remote database has no knowledge of the structure of its blocks, it merely stores key-block pairs. As noted in section 3.6, the key table is currently omitted, but the implementation foresees its possible use.

3.4 Walk-through

3.4.1 Archive

Given a file in the fileset, either it is already represented in the local database or it is not.

If it is not represented, this is the first time we have seen this file. We compute the file's signature, store the summary signature locally, and store the file itself, in compressed and encrypted blocks, and the block list remotely.

If the file is represented in the local database, we can compute, by comparing the stored and actual modification date and SHA-1 hash, whether it has been modified

Summary Signature:

Length:	351824 bytes
File SHA-1:	6854 7369 6920 2073 6F6E 2074 2061 4853 2D41 2E31
Block list SHA-1:	63C6 16BF 8C4C 0D1D AE4B 1A1A BA95 D082 A558 A276
Modification date:	Sun Nov 17 21:46:29 EST 2002

Block List:

block id	offset	weak sum	SHA-1
23867	0	759E AF5E	6979 2969 C569 92FE 6572 FE17 2D9F EFC6 FBFF BCBF
23868	2048	1B76 1BB8	27C1 0BC1 A7C1 EF22 7F0D 3B04 7B82 BF82 E882 82FC
23869	4096	B565 07BC	63BB 9617 7D89 F290 F761 BE96 3D8F 57DA 13EC FB8F
⋮			
about N/L blocks: in this example L=2048			

Table 2: Example signature of a file. In practice, some additional bookkeeping information is kept in the signature as well.

since the last archive pass. If it has not been modified, we are done. If it has, we compare it to its last stored signature, compute a partial covering using blocks in the archive (computed based on the signature), and finally compute a second partial covering that fills the holes left by the first covering. We update the summary signature in the local database and send the new partial covering and the new block list, both compressed and encrypted, to the remote database.

3.4.2 Retrieval

To recover a file, we look up its summary signature in the local database and fetch the block list from the remote. We then determine the block identifiers[2] to request from the remote store, request them, and then reassemble the file locally.

Reassembly, of course, means that we decrypt and uncompress the blocks we have received. During reassembly we check the hashes from the signature against the retrieved blocks to verify that the blocks we have recov-

[2]The block identifiers are assigned sequentially by `cryptar` and serve as primary keys for blocks.

ered are, in fact, those we sent.

If we are recovering a file that already exists in some version locally, `cryptar` may use the local copy for recovery of those blocks unchanged since the last archive operation, and so avoid transferring the entire file.

It's worth noting that we can determine the authenticity of the retrieved blocks by means of the SHA-1 hash that we have stored locally for each block. In order for an attacker to give us forged data, they would have to know not only the SHA-1 hash that a block must match, but also give us blocks of known length that match our given SHA-1 hash. The reconstructed file must also match a pre-computed SHA-1 hash. But SHA-1 [FIPS-180-1] is a 160 bit cryptographic hash designed specifically to make this hard.

3.5 Signatures

As in rsync, we compute an inexpensive but weak rolling checksum of every block in our file (cf. figure 1). That is, given an N-byte file $F = \langle b_1, b_2, \ldots, b_N \rangle$, we compute a total of $N - L$ weak checksums, where L is the block size: csum$\langle b_1, \ldots, b_L \rangle$, csum$\langle b_2, \ldots, b_{L+1} \rangle$, and so forth.

We create a hash table of weak checksums of every such L-byte long block of the file. At this point rsync would ask its server to compute checksums of a minimal covering of the remote file. The block list that we stored at the time of the previous backup contains the checksums for a covering of the file. This allows us to compute the delta without access to the remote file.

Note that the set of weak checksums computed on the covering is a subset of the full set of weak checksums that we compute above.

The weak checksum we use is the rsync weak checksum based on Adler32 [Deutsch] (but in this form due to Andrew Tridgell, who in turn credits Paul Mackerras [Tridgell2]):

$$r_1(k) = \left(\sum_{i=0}^{L-1} a_{i+k} \right) \bmod M$$

$$r_2(k) = \left(\sum_{i=0}^{L-1} (L-i) a_{i+k} \right) \bmod M$$

$$r(k) = r_1(k) + M r_2(k)$$

Here $k \in \{0, \ldots, N - L\}$ is the offset in the N-byte long file of the beginning of the window on which we are computing a checksum, and L is the size (length) of the window.

We use $M = 2^{16}$ for efficiency of computation. Note in particular that the sums overlap, and so computing a block's checksum $r(k)$ once we know the checksum for the block offset one byte earlier, $r(k-1)$, requires only a few arithmetic operations.

For each block in the remote file, whose checksum we know from the block list, we look up the weak checksum in the hash table to determine where it might fit in a partial covering of the local file. If we don't find a hit, we may safely assume the block is no longer represented in the local file. If we find a hit, we compute the strong checksum to verify. As noted previously, we use 160 bit SHA-1 for the strong checksum. If the strong checksum also matches, we conclude the remote block corresponds to the offset in the local file as indicated by the block we matched.

Tridgell used MD4 in rsync, but [Dobberton1, Dobberton2] points out attacks on MD4 such that a stronger hash is advisable when cryptographic strength is a goal. In rsync, the goal of the strong checksum is simple collision avoidance. In our case, though, we require our strong checksum to be able to certify the authenticity of a block as well as avoiding collisions.

After we have exhausted the remote blocks, we scan for holes in the covering and cover any such holes with new blocks. We send the new block list and the new blocks (all compressed and encrypted) to the remote and write a new summary signature in the local database. Every signature in the local database represents a full covering of the file.

A user more interested in conserving remote disk space than in an historical record of his files may delete those remote blocks and block lists which are no longer used and remove the old summary signature from the local database.

3.6 Cryptography

`cryptar` utilizes a standard block cipher for encrypting the data sent to the remote database. Currently the application uses the AES [FIPS-197] cipher, but the specific cipher is sufficiently abstracted in the implementation that another cipher could be substituted if it were desired, say due to the discovery of a vulnerability.

Since the `cryptar` blocks are significantly larger than the size of our cipher's block, we use the cipher in counter mode [Schneier]:

$$\begin{aligned} C_{-1} &= C_{-1} + 1 \pmod{2^W} \\ C_i &= P_i \oplus E_k(C_{-1}) \end{aligned}$$

The initial value of C_{-1} is discussed below.

For reasons of security we can not use the cipher in electronic codebook (ECB) mode. If a single character is changed in a file and the resultant block is updated, the only difference between the initial `cryptar` block and the final `cryptar` block would be a single contiguous block of binary data whose length corresponds to the size of the block cipher used. This could form the basis of a differential cryptanalysis based upon a known ciphertext blocks and is therefore undesirable here.

Using counter mode requires us to use an initialization vector (IV) for each execution of our cipher. We restrict each cipher instantiation to a single `cryptar` block, so each block is written to the data store with an individual, randomly generated IV. This value is stored as the first ciphertext block, C_{-1}.

Counter mode alone provides weak security against someone changing a single cipher block. Assurance that the cipherblock has not been changed, however, is provided by the locally stored cryptographic (SHA-1) hash.

The storage of the IV in the clear on the data store does not decrease the security of our crypto scheme: the IV is cryptographically no different than another ciphertext block. Indeed, an additional benefit to rotating IVs on a per-block basis is that we are able to reuse a single key across many `cryptar` blocks without damaging the integrity of our key due to overuse.

It should be mentioned that while we are currently using only one key across the entire ciphertext, it is possible to rotate the key upon any schedule determined by the user, and this functionality has been implemented in the application. The use of a single key does not degrade security: the use of randomized initialization vectors for each block helps decrease the possibility of cryptanalyzing the key from the large ciphertext store.

We note, finally, that using a new random IV for each `cryptar` block that we encrypt prevents us from using a differencing scheme on the ciphertext blocks for cryptanalysis.

3.7 Hash Confidence

Because `cryptar` stores SHA-1 hashes for each block stored remotely, it has a very good assurance that the blocks it retrieves are the unmodified blocks that it originally transmitted [FIPS-180-1].

By way of illustrative example, consider a 10 MB file stored in 10,240 blocks of size 1K each. From a strictly probabilistic viewpoint, since SHA-1 is a 160 bit hash, the probability that no bad block passes the hash is

$$\begin{aligned} \left(1 - 2^{-160}\right)^{10240} &\approx 1 - 10240\left(2^{-160}\right) \\ &\approx 1 - 2^{13}2^{-160} \\ &= 1 - 2^{-147} \end{aligned}$$

More expansively, suppose we transfer 1000 such files per second without stop for 100 years. Then the number of blocks transferred would be

$$(1000)(10240)(60 \cdot 60 \cdot 24 \cdot 365.25 \cdot 100) \approx 2^{55}$$

and the probability that no bad block will pass the hash at some point during that century of work is

$$\begin{aligned} \left(1 - 2^{-160}\right)^{\left(2^{55}\right)} &\approx 1 - 2^{55}2^{-160} \\ &= 1 - 2^{-105} \end{aligned}$$

Thus, the probability of a bad block being mistakenly accepted for a good block during a 100 years of such transfers is 2^{-105}.

If a bad block does pass the hash, it would still have to meet the file level hash, also a 160 bit SHA-1. Clearly the a priori probability of failure is extremely low.

It appears, moreover, that it is extremely difficult to create blocks that meet a given SHA-1 hash, let alone a block of a fixed length that must then meet a second (whole-file) SHA-1 hash.

Note that the guarantee we provide is that if we get data back, we will know that it is our data. We have no assurance that our data will not be deleted due to malice, disaster, or administrative error.

4 Security

The ready availability of ssh relieves us of any requirement to provide a secure protocol for communication with the archive. We simply have `cryptar` talk to the remote using the user's choice of secure transport agent.

We compress and then encrypt data before sending it to the archive. We allow for padding the encrypted data with random data to hide compression efficiency. We currently use AES[FIPS-197] to encrypt data.

We distinguish between two types of attackers: a network attacker and a remote host attacker.

The network attacker can see traffic, but, presumably, make no sense of it, as the channel is encrypted by the prudent user. Some information is still available to the network attacker, however, such as the time of day at which backups are made and how much data is transferred.

As a fanciful example, if the amount of data tends to increment in units of 60 MB and the `cryptar` user is known to be fond of ripping CD's, the attacker might guess that the data represents mp3's encoded at 128 kbps.

A remote host attacker, on the other hand, could also glean some information from noting the order in which blocks have been sent to the archive. Although we could randomly generate remote block identifiers to hide block order, a remote attacker could still see the order in which the blocks arrive.

To impede such an attacker, we could transmit items randomly from our queue of items to send, even operating on more than one file at once, in an effort to hide information about which blocks come from which files. The protection thus gained, however, would not be significant compared to the protection afforded by AES itself.

In any case, if I know that you modified file `foo` this morning, and you do a backup this afternoon, and I am a sufficiently privileged attacker, I could at least learn that some subset of a small set of blocks corresponds to the delta of file `foo`.

In addition, because block lists are stored remotely, a remote host attacker could conclude that any session in which significant data moves from the local to the remote is an archive operation, and so those blocks that travel in the reverse direction are block lists.

It is at this point that we depend upon the strength of our encryption to protect our data. We use different random initialization vectors for each block, and we allow, moreover, for different encryption keys for different blocks. If multiple keys are utilized, an attacker with access only to the remote store and who successfully cryptanalyzes a single key would still not necessarily be able to gain access to a complete file, since a single file may require multiple blocks for a covering.

As previously discussed, the SHA-1 hash provides a very good assurance that the block we retrieve is, in fact, the block we sent[FIPS-180-1].

Finally, it should be mentioned that `cryptar` does not attempt to be robust against timing analysis.

5 Protocol and Usage

`cryptar` uses a simple binary wire protocol. We thought of using a popular protocol like XML or HTTP, but these protocols require encoding binary data as text, which introduces too much overhead. Since the bulk of the data transferred by `cryptar` is binary (compressed blocks), it made sense for the protocol to allow binary data without further encoding.

Each `cryptar` exchange consists of a one byte command, a version number (not shown below), and a payload. The commands are `Hello`, `Bye`, `PutBlock`, `PutBlockAck`, `GetBlock`, and `GetBlockAck`. The only version of the protocol for now is version zero.

(Hello, Message)

(Bye, Reason)

(PutBlock, id , data length , data)

(PutBlockAck, id)

(GetBlock, id)

(GetBlockAck, id , data length , data)

The interface to `cryptar` is similar to that of `tar`.

6 Performance

A frequent objection to rsync-like protocols is that a single changed byte invalidates an entire block. Indeed, an adversary (were our expected local environment adversarial) could change one byte every $L-1$ bytes, where L is the block size, and thus invalidate every block. While it is important to note that the worst case performance of our algorithm deteriorates to full file copy, real world usage does not typically involve such regular and sparse changes.

7 Applications

While cryptar clearly provides secure backup, several other interesting applications arise.

For example, cryptar can provide a secure form of pervasive version control. Traditional version control tools, such as CVS, are designed for collaboration rather than security. Using them to manage, say, a user's home directory is cumbersome, invasive to work flow, and insecure. Files must be explicitly added and deleted from the VC system, for example. The user has no automatic way, moreover, of being warned that a file has changed without his being the agent of that change. cryptar, on the other hand, automatically handles these details.

Unlike most other version control systems, however, cryptar does not provide collaboration support. A few version control systems provide collaboration support as well as some degree of security, but remain intrusive to normal file operations [Shapiro2002, Shapiro2003].

cryptar also solves an important problem related to memory "suspend and resume" work, which lets users freeze and save their current environment on one computer and move it to another machine. [Kozuch] Without machine migration, this is the familiar case of putting a laptop to sleep and waking it up again. A user may not wish to use or move a laptop, however.

Internet suspend and resume allows the user to put his current environment to sleep and resume on another machine. The memory state image, however, must first move to the new machine. The first time this happens, of course, the image must be fully copied. The next suspend and restore, however, need only copy the part of memory that has changed.

Since the data that must be moved from one machine to another is the entire memory image of the computer; this presents a very attractive target for a system attacker and must be correspondingly well protected. Standard cryptographic techniques would force us to move a newly encrypted copy of the entire memory core from system to system, whereas cryptar would allow us to move only the changed memory pages.

8 Future Directions

The current cryptar implementation does not adequately address failure modes. If the network connection dies during backup, for example, remote blocks involved in the currently archiving file may be orphaned. This is a matter of bookkeeping that should be addressed. While important, its effect is limited to inefficiency: orphaned blocks merely waste space.

The protocol should be expanded to provide for statistically verifying the remote archive, as well as for verifying that a given file of the user's choice can be restored without necessarily restoring it.

If the user loses his entire local database, the current implementation of cryptar provides no mechanism for restoring anything or even notifying the remote host that the remote database is no longer needed. The latter is trivially addressed by the addition of a purge command to the protocol. The former is initially addressable through ordinary backup techniques. For example, while it is often impractical to back up a user's entire file set to removable media, the local database should fit on a CD-R.

This said, cryptar could recursively back up its own local database. Indeed, the entire local database could be encrypted and sent as a single block to the remote store. cryptar could then print a short bit of hex data representing the remote host, remote store identifier, and the block number, SHA-1, and encryption key to retrieve the local database. The user could, in the worst case, enter this information by hand to bootstrap a complete restore in the case of a total loss.

The protocol includes some bookkeeping information in order to match requests and acknowledgments. A clever attacker could, without breaking encryption, glean some information about which blocks go with which files, even though they would not know the names or contents of those files. Together with knowledge of the user's files, however, this could allow an attacker to focus his efforts more efficiently.

The protocol should be changed to avoid these revelations. The client would need to do more bookkeeping to handle request–acknowledgment matching.

cryptar does not currently address the question of server reliability, only of data privacy. Using error correcting codes, one could distribute the data over several remote servers so that any k of them could fail without our losing data.

9 History and Status

cryptar is new software that borrows many ideas and some code from rsync. It is written in C, utilizes the GLIB and LIBTOMCRYPT [StDenis] libraries, and is available under the GPL. The initial implementation was built under Debian GNU/linux. While the software will no doubt grow to meet its users' extended needs, the core functionality described in this paper is implemented.

10 Conclusion

cryptar implements a scheme for securely storing and efficiently updating data on untrusted remote servers. As such, it provides an efficient and safe means to manage remote backups.

11 Availability

cryptar is available at http://www.cs.drexel.edu/~jeffa/cryptar/.

References

[Deutsch] P. Deutsch and J.-L. Gailly, RFC 1950: ZLIB Compressed Data Format Specification version 3.3, http://www.ietf.org/rfc/rfc1950.txt (May 1996).

[Dobberton1] Hans Dobbertin, Cryptanalysis of MD5 Compress, (May 1996). http://citeseer.nj.nec.com/dobbertin96cryptanalysis.html

[Dobberton2] Hans Dobbertin. The First Two Rounds of MD4 are Not One-Way. In Fast Software Encryption '98, pp. 284-292. Springer-Verlag, (1998). http://citeseer.nj.nec.com/dobbertin97first.html

[Rubel] Mike Rubel, Easy Automated Snapshot-Style Backups with Linux and Rsync, http://www.mikerubel.org/computers/rsync_snapshots/ (2002).

[Tridgell1] Andrew Tridgell and Paul Mackerras , *The rsync algorithm* http://samba.anu.edu.au/rsync/tech_report/tech_report.html, (November 1998).

[Tridgell2] Andrew Tridgell, *Efficient Algorithms for Sorting and Synchronization*, PhD Thesis, The Australian National University (April 2000).

[FIPS-180-1] Secure Hash Standard, Federal Information Processing Standards Publication 180-1 http://www.itl.nist.gov/fipspubs/fip180-1.htm, April 1995.

[FIPS-197] Advanced Encryption Standard, Federal Information Processing Standards Publication 197, November 2001.

[Kozuch] Kozuch, M., Satyanarayanan, M., Bressoud, T. and Ke, Y., "Efficient State Transfer for Internet Suspend/Resume," IRP-TR-02-03, May. 1, 2002

[Quinlan] Sean Quinlan and Sean Dorward, *Venti: a New Approach to Archival Storage*, FAST 2002 Conference on File and Storage Technologies, (January 2002).

[Schneier] Bruce Schneier, *Applied Cryptography, 2nd Edition*, Wiley and Sons, Inc., 1996.

[Shapiro2002] Jonathan S. Shapiro and John Vanderburgh, "CPCMS: A Configuration Management System Based on Cryptographic Names," Proc. 2002 USENIX Annual Technical Conference, FreeNIX Track, Monterey, CA, 2002

[Shapiro2003] Jonathan S. Shapiro, John Vanderburgh, and Jack Lloyd "OpenCM: Early Experiences and Lessons Learned," Proc. 2003 USENIX Annual Technical Conference, FreeNIX Track.

[StDenis] Tom St. Denis, LibTomCrypt, http://libtomcrypt.org/.