From bsdtar to tarsnap Building an online backup service

> Colin Percival Tarsnap Backup Inc. cperciva@tarsnap.com

September 28, 2013

- A computer scientist.
 - Publications in computational mathematics, numerical analysis, data compression, and cryptography.
- A FreeBSD developer (since 2004).
 - FreeBSD Security Officer from August 2005 to May 2012.
 - Maintainer of the FreeBSD/EC2 platform.
- Author of some small UNIX utilities: bsdiff, freebsd-update, portsnap, scrypt, spiped, and kivaloo.
- "The Tarsnap guy".
 - This is my day job.

What is Tarsnap?

- Tarsnap is "online backups for the truly paranoid".
 - Data is encrypted and signed with keys held at the client side.
 - You have the source code audit it, please!
- Tarsnap is online backup for UNIX-like operating systems, with a tar-line command line:

tarsnap -cf home-2013-09-28 --exclude *.core /home

- Tarsnap is snapshotted deduplicated storage.
- Tarsnap is a pay-as-you-go service: 300 picodollars / byte-month of storage. 300 picodollars / byte of bandwidth.

- 1. Start with bsdtar...
 - ... and libarchive, which does most of the work.
- 2. Deduplicate archive data.
 - 2.1 Split data into blocks.
 - 2.2 Reference-count the blocks.
- 3. Add cryptography.
 - 3.1 Encrypt all the blocks.
 - 3.2 Sign the archive.
 - 3.3 Sign all the blocks.
- 4. Upload the new data...
 - ... while the archive is being created.
- 5. Have a server which puts the data somewhere safe...
 - ... and can find it for you again when you need it!

1. Start with bsdtar

- In February 2004, Tim Kientzle imported libarchive into the FreeBSD src tree.
 - Two months later, he added bsdtar.
- libarchive:gnutar :: llvm:gcc
 - libarchive is clean, well-organized, designed as a reusable library, and BSD licensed.
- bsdtar is a front-end to libarchive 5% of the size, and mostly concerned with parsing command-line options.
- Until October 2008, all libarchive development happened in the FreeBSD src tree.

- Set archive format: archive_write_set_format_pax
- Read and write headers: archive_write_header, archive_read_next_header
- Read and write file data: archive_write_data, archive_read_data
- Skip file data: archive_read_data_skip
- Create a new archive: archive_write_open(...open_callback, write_callback, close_callback)
- Read an archive: archive_read_open2(..., open_callback, read_callback, skip_callback, close_callback)

2.1. Split data into blocks

 For deduplication in a filesystem, splitting data into fixed-size blocks works well:

Theq uick brow nfox jump sove rthe lazy dog. Theq uick brow nfox jump sove rthe EDIT dog.

- Even if writes aren't aligned it isn't too bad: Theq uick brow nfox jump sEDI Tthe lazy dog.
- With archives you need to handle files being added or deleted in the middle however:

```
Theq uick foxj umps over thel azyd og.
```

 Solution: Use variable-sized blocks constructed by splitting at content-dependent points:

```
Th quickbrow nf oxjumpsov erth elazy dog.
```

Th quickf oxjumpsov erth elazy dog.

• Simple method: Split before byte x_n if

$$H(x_{n-k}, x_{n-k+1}, \dots, x_{n-2}, x_{n-1}) = 0$$

for a convenient hash function H and a fixed k.

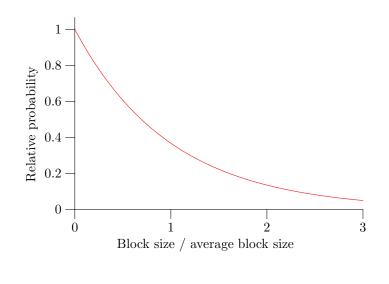
• If you use the "rolling" hash function

$$H(x_0\ldots x_{k-1})=\sum_i x_i\alpha^i \bmod p$$

the H() values can be computed in two modular multiplications and two modular additions per byte processed.

• This yields an equal probability of splitting at each position and an exponential distribution of block sizes.

Exponential block size distribution



• Improved method: Split before byte x_n if

$$H(x_{n-k}, x_{n-k+1}, \dots, x_{n-2}, x_{n-1}) = 0$$

for a convenient hash function H and any value of k.

• Trick: Use the same function

$$H(x_0\ldots x_{k-1})=\sum_i x_i\alpha^i \bmod p,$$

compute the values $H(x_0 \dots x_{n-1})$, and store them in a hash table.

• Split if we find values such that

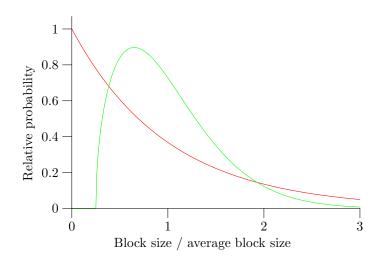
$$H(x_0\ldots x_{m-1})=H(x_0\ldots x_{n-1}).$$

Data-dependent splitting

$$H(x_0 \dots x_{m-1}) = H(x_0 \dots x_{n-1})$$
$$\sum_{i=0\dots m-1} x_i \alpha^i = \sum_{i=0\dots n-1} x_i \alpha^i$$
$$\sum_{i=n\dots m-1} x_i \alpha^i = 0$$
$$\alpha^n \sum_{i=n\dots m-1} x_i \alpha^{i-n} = 0$$
$$\sum_{i=n\dots m-1} x_i \alpha^{i-n} = 0$$
$$H(x_n \dots x_{m-1}) = 0$$

)

Improved block size distribution



2.2 Reference-count the blocks

- The splitting process converts an archive into a sequence of blocks plus an index.
 - Blocks are identified by their HMAC-SHA256 values.
 - The index is identified by the HMAC-SHA256 of the archive name.
- The client keeps a "cache directory" locally which contains block HMACs and reference counts.
 - Required for creating or deleting archives, but not for reads.
 - Updated atomically when an archive creation or deletion is committed.
 - Can be regenerated by a "fsck" operation which reads the block indexes for each archive.
- The server has no idea how many archives use a particular block or when a new archive is re-using an existing block.

- File data, tar headers, the block index, archive metadata should all be kept confidential.
 - Everything making up an archive (including its name) is part of an encrypted block with an ID generated via HMAC-SHA256.
- Don't necessarily want someone who can *write* archives to be able to *read* them.
 - Blocks are encrypted with an ephemeral 256-bit AES key, which is itself encrypted using a 2048-bit RSA key.
- Don't want anyone to be able to tamper with archives (even blindly).
 - The block index contains HMACs of blocks, the archive metadata contains the hash of the index, and the metadata is signed with a 2048-bit RSA key.

- Encrypting blocks and signing archives is sufficient for *cryptographic* security, but not sufficient for *real-world* security.
 - Block data is compressed using the "deflate" algorithm.
 - The HMACs used to verify blocks cover the *uncompressed* data.
 - If a block is tampered with, we wouldn't know until after decompression.
 - zlib has a long history of vulnerabilities!
- To protect against an attacker who has a zlib exploit and can tamper with our backups, we append a "physical" HMAC to the end of each block.

4. Networking

• Tarsnap uses a custom request-response protocol:

- Read block, write block, delete block.
- List blocks.
- Start a transaction and cancel any ongoing transaction.
- Commit transaction if it has not been committed or cancelled.
- Transactions are necessary to handle client crashes without being left with "orphaned" data.
- All operations are idempotent running them twice has the same effect as running them once.
 - This makes it safe for the client to retry any failed requests.
 - This allows the server to drop connections arbitrarily and rely on the client handling the failure.

- Requests and responses are signed with "read", "write", or "delete" HMAC keys.
 - Using separate keys makes it possible to have a server which can upload data but cannot tamper with existing backups.
 - These are the only client keys possessed by the server.
- Everything is wrapped inside a cryptographic transport layer which is similar to SSL using an RSA_DH certificate, except without all the certificate authority nonsense.
 - The server public key is distributed with the tarsnap source code.

- As tarsnap archives are generated, they are streamed to the server.
- Obvious strategy: Have one thread generating the archive and another thread doing networking.
 - Anyone who's done serious work with concurrent systems knows that they are actively malicious. Robert Watson
- Instead, I decided to use non-blocking networking.
 - Using a framework like libevent would probably have been a good idea, but instead I wrote my own.
- Sprinkled throughout the tar code I have "go do non-blocking networking stuff" calls.
- If too many requests are pending, we block until the networking catches up with the tar code.

5. Server

- Tarsnap stores data in Amazon S3 by synthesizing a log-structured filesystem.
 - Amazon advertises that S3 is designed to provide 99.999999999% durability.
 - Incoming requests are logged to S3 before responses are sent back to the tarsnap client.
- The Tarsnap server code runs in Amazon EC2 and keeps a local cache of metadata.
 - (machine X, block Y) \rightarrow (S3 object A, offset B, length C)
 - S3 is the ultimate "source of truth".
- The Tarsnap server runs a continuous "cleaning" process which reads data from S3 and writes it back minus any deleted blocks.
 - This is why Tarsnap uses EC2: S3 ↔ EC2 bandwidth is free, and cleaning uses a lot of bandwidth.

Availability

- Tarsnap source code (not open source licensed): http://www.tarsnap.com
- Open source "spin-off" work:
 - scrypt key derivation function: http://www.tarsnap.com/scrypt.html
 - spiped secure pipe daemon: http://www.tarsnap.com/spiped.html
 - kivaloo data store: http://www.tarsnap.com/kivaloo.html
 - FreeBSD/EC2:

http://www.daemonology.net/freebsd-on-ec2/

Questions?