Attack and Defense Toolkit against Encrypted Deduplication

Introduction

Encrypted deduplication seamlessly combines encryption and deduplication to simultaneously achieve both data security and storage efficiency. State-of-the-art encrypted deduplication systems mostly adopt a deterministic encryption approach that encrypts each plaintext chunk with a key derived from the content of the chunk itself, so that identical plaintext chunks are always encrypted into identical ciphertext chunks for deduplication. However, such deterministic encryption inherently reveals the underlying frequency distribution of the original plaintext chunks. This allows an adversary to launch frequency analysis against the resulting ciphertext chunks, and ultimately infer the content of the original plaintext chunks.

We study how frequency analysis practically affects information leakage in encrypted deduplication storage, from both attack and defense perspectives. We propose a new inference attack that exploits chunk locality to increase the coverage of inferred chunks. We conduct trace-driven evaluation on a real-world dataset, and show that the new inference attack can infer a significant fraction of plaintext chunks under backup workloads. To protect against frequency analysis, we present two defense schemes, namely MinHash encryption and scrambling, which aim to disturb the frequency rank or break the chunk locality of ciphertext workloads. Our trace-driven evaluations show that our combined MinHash encryption and scrambling scheme effectively mitigates the inference attack, while incurring limited storage and performance overhead.

The toolkit includes the attack and defense implementations against the FSL dataset, as well as a deduplication storage prototype based on an existing realistic deduplication system of data domain file system (DDFS).

Publication

- Jingwei Li, Chuan Qin, Patrick P. C. Lee, Xiaosong Zhang. Information Leakage in Encrypted Deduplication via Frequency Analysis. In Proc. of IEEE/IFIP DSN, 2017. Special thanks to Chufeng Tan for his help in preparing source code.

Preparation

The toolkit is running under Linux (e.g., Ubuntu 14.04) with a C++ compiler (e.g., g++). To run the programs, you need to install/compile the following dependencies.

- Libssl API: run the command `sudo apt-get install libssl-dev` for installation.
- Snappy compression library: run the command `sudo apt-get install libsnappy-dev` for installation.
- Google Leveldb: a version of 1.20 is provided in `util/`.
- fs-hasher: a version
  of 0.9.4 is provided in `util/`.

All components of our toolkit depend on `fs-hasher` and `leveldb`. Before configuring each component (e.g., the attacks, the defenses, and the prototype), need to copy `util/fs-hasher/` and `util/leveldb/` into the corresponding directory (e.g., `attack/basic/`, `attack/locality/`, `defense/minhash/`, `defense/scrambling/`, `defense/combined/`, and `prototype/`) and compile them, respectively.

Attacks

We provide the basic and the locality-based attacks against encrypted deduplication.

Basic Attack

The basic attack builds on classical frequency analysis. Follow the following steps to simulate the basic attack.

Step 1, configure basic attack: modify variables in `attack/basic/basic_script.sh` to adapt expected settings:

- `fsl` specifies the path of the FSL trace.
- `users` specifies which users are collectively considered in backups.
- `date_of_aux` specifies the backup of which date is considered as auxiliary information.
- `date_of_latest` specifies the latest backup of which date is the target for inference.

Step 2, run basic attack: type the following commands to compile and run the basic attack.

```
$ cd attack/basic/ && make
$ ./basic_script.sh
```

Locality-based Attack

The locality-based attack exploits chunk locality to improve attack severity. To simulate the locality-based attack, follow the steps below.

Step 1, configure locality-based attack: In addition to the common variables (e.g., `fsl`, `users`, `date_of_aux` and `date_of_latest`), the locality-based attack builds on four specific parameters that are defined in `attack/locality/locality_script.sh`:

- `u` specifies the number of most frequent chunk pairs to be returned by frequency analysis in initializing the inferred set.
- `v` specifies the number of most frequent chunk pairs to be returned by frequency analysis in each iteration.
- `w` specifies the maximum number of ciphertext-plaintext chunk pairs that can be held by the inferred set.
- `leakage_rate` specifies the ratio of the number of ciphertext-plaintext chunk pairs known by the adversary to the total number of ciphertext chunks in the latest backup.

Step 2, run locality-based attack: type the following commands to compile and run the locality-based attack.

```
$ cd attack/locality/ && make
$ ./locality_script.sh
```

Defenses

We provide the MinHash encryption, the scrambling, and the combination of both to defend against frequency analysis.

MinHash Encryption

MinHash encryption derives an encryption key based on the minimum fingerprint over a set (called segment) of adjacent chunks, such that some identical plaintext chunks can be encrypted into multiple distinct ciphertext chunks thereby disturbing frequency rank. To simulate the MinHash encryption, follow the steps below.

Step 1, configure MinHash encryption: the MinHash encryption builds on two parameters that are defined in `defense/minhash/k_minhash.cc`:

- Segment size: the MinHash implementation uses variable-size segmentation and identifies segment boundary based on chunk fingerprints. By default, we set the average segment size, maximum segment size and minimum segment size at 1MB, 2MB and 512KB, respectively. It is feasible to change segment sizes by modifying macro variables `SEG_SIZE`, `SEG_MIN` and `SEG_MAX`; note that when changing `SEG_SIZE`, it is needed to adjust the code in line 112 of `defense/minhash/k_minhash.cc` to adapt the global divisor, for example if the average segment size is 512KB and 2MB, the line of code should be changed as follows.

```
if (sq_size + size > SEG_MAX || (sq_size >= SEG_MIN && (hash[5] << 3) >> 3 == 0x1f))    // corre
spond to an average segment size of 512KB
if (sq_size + size > SEG_MAX || (sq_size >= SEG_MIN && (hash[5] << 1) >> 1 == 0x7f))    // corre
spond to an average segment size of 2MB
```

- K: our implementation supports k-MinHash that derives an encryption key from a random k-minimum fingerprint of a segment. By default, we use MinHash and set `K_MINHASH` by 1.

Step 2, configure locality-based attack: it is identical to the Step 1 of the guideline of locality-based attack, except the attack parameters (e.g., `u`, `v` and `w`) locate in `defense/minhash/minhash.sh`.

Step 3, run MinHash encryption to defend against locality-based attack: type the following commands to compile and run.

```
$ cd defense/minhash/ && make
$ ./minhash.sh
```

Scrambling

Scrambling disturbs the processing sequence of chunks, so as to prevent an adversary from correctly identifying the neighbors of each chunk in the locality-based attack. To simulate the scrambling scheme, follow the steps below.

Step 1, configure scrambling scheme: Like MinHash encryption, scrambling works on a segment basis, and builds on three parameters of `SEG_SIZE`, `SEG_MIN` and `SEG_MAX` to define variable-size segmentation. You can follow the Step 1 of the guideline of MinHash encryption to configure these parameters that are defined in `defense/scrambling/scrambling.cc`.

Step 2, configure locality-based attack: it is identical to the Step 1 of the guideline of locality-based attack, except the attack parameters (e.g., `u`, `v` and `w`) locate in `defense/scrambling/scrambling.sh`.

Step 3, run scrambling to defend against locality-based attack: type the following commands to compile and run.

```
$ cd defense/scrambling/ && make
$ ./scrambling.sh
```

Combined

We also introduce a combined scheme that first scrambles the orders of chunks in a segment basis, and then encrypts each chunk via MinHash encryption. The guideline of the attack is identical with that of MinHash encryption.

Deduplication Prototype

We design and implement a deduplication-based storage prototype based on DDFS. The key design is to store unique chunks in logical order and further exploit chunk locality to accelerate deduplication. Instead of storing actual data, our prototype works on metadata level. You can follow the following steps to evaluate the metadata access overhead of either message-locked encryption (MLE) or the combined scheme based on our prototype.

Step 1, configure prototype: the prototype builds on two types of parameters, all of which are defined in `prototype/ddfs.cc`:

- The cache-related parameter is the size `LRU_SIZE` of fingerprint cache. Note that we describe `LRU_SIZE` by the maximum number of fingerprints that the cache can hold.

- The Bloom filter-related parameters include the maximum number `BLOOM_lenth` of entries in the Bloom filter array, and the false positive rate `ERROR` of Bloom filter.

Step 2, configure encryption scheme: we provide the deduplication simulation from the ciphertext chunks by either the MLE or the combined scheme. You can configure the parameters of the combined scheme by modifying `SEG_MIN`, `SEG_MAX`, `SEG_SIZE`, and `K_MINHASH` in `prototype/combined.cc` (see the Step 1 of the guideline of MinHash encryption).

Step 3, run storage simulation: type the following commands to compile and run the simulation.

```
$ cd prototype/ && make
$ ./combined.sh  // run the combined scheme
$ ./mle.sh  // run the MLE scheme
```

Outputs

Attack/Defense

The output format of attack/defense is shown as follows.

```
==========================Attack/Defense==========================
Auxiliary information: YYYY-MM-DD;      Target backup: YYYY-MM-DD
[Parameters: (u, v, w) = ...]
Total number of unique ciphertext chunks: X
[Leakage rate: ...]
Correct inferences: Y
Inference rate: ...

Successfully inferred following chunks:
......
```

`X` is the number of unique ciphertext chunks in the encryption of the target backup, while `Y` is the number of (unique) chunks that can be successfully inferred by the attacks. The inference rate is computed by `Y/X`, that is slightly affected by the sorting algorithm in frequency analysis. The reason is different sorting algorithms may break tied chunks (that have the same frequency counts) in different ways and lead to (slightly) different results. The `parameters` and `leakage rate` are only available in the simulation of locality-based attack and its defense. We output the fingerprints of inferred plaintext chunks in both attack and defense simulation.

Deduplication Simulation

We output the metadata access overhead in storage simulation in the following format:

```
fslhomes-userX-YYYY-MM-DD
Index access: A
Update access: B
Loading access: C
```

The information elaborates the metadata access overhead of storing user `X`'s backup on the date of `YYYY-MM-DD`. The metadata access overhead includes index access, update access and loading access, all of which are evaluated in the unit of times.