Forgetful Data Structures for Text

Moiri Gamboni, Advisor: Azza Abouzied

Project Summary

Abstract

While data loss is considered harmful in computer systems, we argue that forgetfulness is a tool which can be used to achieve data privacy, plausible deniability, and the right to be forgotten. We extend the forgit, a forgetful data structure which can store numerical and image data[1], to accomodate text data. Some forgetfulness properties satisfied by numerical and image data cannot be acheived in text due to the lack of significance ordering. We relax those required properties to implement a text forgit support gracedful and passive data loss, at some cost to storage and pre-processing. Some experiments showing the practicality of a text forgit are run, and several implementation methods are investigated. Finally, we show how future research could improve the usefulness of a text forgit.

1 Introduction

1.1 Specific Aims

Data loss is viewed as an error of computer systems often resulting from failures in low-level software systems or hardware. From this perspective, these failures should be prevented, mitigated, and if possible recovered from. However, a distinction should be made between the unintentional data loss just described and predictable data loss. Advancements in hardware and software have made the former less common and more easily recovered from, however, systems that take advantage of controlled data loss already exist. The applications are varied: some examples include caching, soft-state, and lossy compression. These all harness data loss to achieve desirable objectives such as increased efficiency and flexibility.

Furthermore, there have been several ephemeral applications designed with the explicit and overt goal of forgetting data. This is motivated primarily by privacy concerns of both users who refuse that their data be stored indefinitely [21], and of society in general as exemplified by the right to be forgotten in the European Union. [10, 11] For example, Snapchat, Wickr, and Silent Circle can be used to send data that self-destructs to the benefit of their users. [18, 23, 17, 16]

Finally, the storage of data carries costs. While hardware and maintenance costs are relatively cheap, the cost of storing an ever-increasing amount of data indefinitely overweighs the potential financial gain it could generate. Data then needs to be deleted and important information could be irreversibly lost. Compression of data is only a short-term solution to a growing problem. Furthermore, costs other than hardware costs should not be overlooked. For example, internal company data kept indefinitely increases the risk of high litigation costs, and prevents plausible deniability in the case of warrants, subpoenas, government surveillance, etc.

For all these reasons, we believe that a forgetful data store would be beneficial for individuals, societies, and companies. We define forgetfulness as having four properties: passive loss, graceful degradation, no hi-low storage, and negligible pre-processing overhead. Forgetful data stores keep more recent data at a higher fidelity than stale data by forgetting the least significant data. These

data stores could fulfill the needs above by providing a framework for gracefully and predictably losing data.

1.2 Background

1.2.1 Related Work

Data Privacy Most mechanisms that forget data for privacy purposes do so by periodic, active deletion [9]. Research has mainly focused on providing guarantees for the deletion or expiration of data [7, 12, 20]. Encryption can also be used but requires secure storage of the decryption key. More recent efforts to ensure privacy focus on securing mobile data and wiping data left after communications. [19, 5]

Compression Lossy compression only stores humanly perceptible differences, which leads to a greater level of compression compared to lossless compression schemes. Lossy compression is only possible if significance ordering of the data is possible. Fourier transforms of time-series data can achieve this [3], and standard lossy image compression algorithms such as JPG also store the data with a significance ordering correlated to human perception (see Image Forgit).

Approximate Computation Approximate computation focuses on the performance benefits of lossy computation rather than space reduction by tolerating errors in the data. Unfortunately, only certain domains (media processing, artificial intelligence, data encoding, etc.) have been able to significantly benefit from approximate computation [13, 22]. Recently, database query approximation systems have allowed users to sacrifice accuracy for performance while still giving statistical guarantees on the data returned. [?, 6, 8, 14, 15]

Forgetful Data Stores Forgetful data stores can employ or improve on the methods above. Firstly, forgetful data stores lose data in a way which better fits human privacy requirements, since data is gradually lost as it gets older. Forgetful data stores are compatible with existing encryption mechanisms and can employ data compression schemes outputting data ordered by significance. Additionally, they allow multiple levels of fidelity to be stored simultaneously. Approximate computation might also benefit from forgetful data stores. Abouzied and Chen describe how a forgetful data store, which they call the forgit, can be implemented for numerical data and images. [1]

1.2.2 Forgits

Properties of Forgetfulness Abouzied and Chen attempt to define forgetfulness so as to best fit human intuitions about memory. Firstly, human memories are not stored at perfect fidelity. As time goes on, older memories gradually and passively degrade. Additionally, it is reasonable to assume that only a single copy of each memory is stored rather than multiple copies which are chosen depending on how old the memory is. This leads to the four following properties which a forgetful data store must satisfy:

1. Passive fidelity loss: A forgetful data store does not actively scan its data to drop fragments. Loss occurs as a side-effect of pushing more data into a finite data store.

2. Graceful degradation: Forgetful data stores can store data in varying degrees of fidelity, and forget the least significant fragments first. As passive loss occurs, old data is eventually completely deleted.

3. No hi-lo storage: A forgetful data store does not inflate the stored representation of the data to support the other properties. means that storing data in a forgit is not significantly more computationally expensive than a conventional data store.

4. Negligible pre-processing overhead: The overhead of pushing data into the store should be negligible compared to conventional data stores.

Forgits support infinite appends and retrievals of data decomposable into F segments s 1 , s 2 , ..., s F such that segment s i1 is less significant than segment s i . The store consists of F circular buffers b 1 , b 2 , ..., b F such that 1 size(b i1) size(b i) and segment s i is appended to buffer b i . [1]

Numerical Forgits The binary representation of numbers already preserves significance ordering. Thus, no pre-processing is required to determine which segments should be forgotten. Given F fidelity levels, the binary representation of the number is split into F equal-sized segments. This method can be easily adapted to floating point representations. [1]

Image Forgits Commonly used compressed image formats such as JPEG also preserve significance ordering. An image stored in the JPEG format is represented by a discrete cosine transform (DCT) matrix [2]. This preserves the significance ordering as human eyesight is more sensitive to low frequency coefficients than high frequency coefficients. The natural significance ordering of coefficients in the DCT matrix is a zigzag through the matrix from the top-left to the bottom-right coefficient, where the bottom-right is the least significant.

2 Goals and Potential Impact

Describe why the proposed research is important. Provide a statement of what you think the impact of your research will be. Be as specific as possible.

3 Methodology

3.1 Overview

As opposed to numbers and images, text has no significance ordering. Therefore, not all previously defined properties can be satisfied. The properties of passive fidelity loss and graceful degradation are considered essential to the concept of a forgit. Therefore, absence of hi-lo storage and/or negligible pre-processing overhead are candidates for properties which can be abandoned in text forgits. We propose a specific implementation of a text forgit which requires pre-processing and incurs some hi-lo storage costs. The following steps give a broad description of the process involved in instantiating the forgit and subsequently using to retrieve partially forgotten texts.

1. In addition to the storage mechanism described for numerical or image data, the text forgit is backed by a trie data structure. This trie is hashed by n-grams computed from texts which are progressively added as users generate them. The number of occurrences of each n-gram is stored in the corresponding node in the trie. We also investigate the possibility of storing a hash of the text in the forgit which is never forgotten to allow for perfect reconstruction of the text (see 4. below).

2. When a text is added to the forgit, its n-grams and their occurrences are added to the trie. Due to the behaviour of the forgit, some percentage of previously stored texts is forgotten in the process.

3. When a text needs to be retrieved, the forgit generates a tree depth-first, where each node is a 1-gram. A path from the root to a leaf corresponds to a possible reconstructed text. This tree is computed from the partially forgotten source text and the information stored in the trie. We also investigate the possibility of using a bloom filter of the tokens of the source text to speed up the tree generation.

4. It is guaranteed that only one path in the tree corresponds to the source text. However, depending on the length of the source text and the percentage forgotten, it might be intractable to generate the entire tree. When the hash of the source text is available, we obtain a perfect reconstruction of the text in the retrieval step (as the chance of hash collisions is negligible). If a hash is not used, we instead get a list of possible texts.

3.2 Dataset

The Enron dataset wass used for all experiments [4]. This dataset contains approximately half a million emails from about 150 people. These company emails were made public by Federal Energy Regulatory Commission during its investigation into Enron. This provides an opportunity to test the forgit on real-life data which is especially suited to a forgetful data store.

3.3 Pre-processing

3.3.1 Email Parsing

The emails were parsed by the email package in the Python Standard Library. In all email threads, the body of each email (the most recent email, all previous replies to the original email, and the original email) was extracted and other data such as sender, recipients, and date were discarded. Some emails failed to parse due to encoding errors: of the 517402 total emails, 517311 emails were successfully parsed.

3.3.2 Tokenization

Unique emails were then tokenized by splitting on contiguous whitespace. Splitting on contiguous whitespace prevents the occurrence of many n-grams which are not relevant to the textual information contained in the email, for example, new lines or mistyped double spaces. However, because tokenization needs to be reversible to perfectly reconstruct emails, the md5 hash of the emails was computed by joining the tokens with spaces. A bloom filter with an error rate of 0.1 was computed by adding all unique tokens using the pybloom_live package. Trigrams were then computed and stored in a trie, which was generated using the marisa-trie package providing Python bindings for a C++ implementation of MARISA tries. This package provides a static and memory-efficient trie data structure. For the Enron dataset, INSERTNUMBERHERE trigrams were found, resulting in a trie that takes INSERTNUMBERHER when stored on disk. The forgetting mechanism is not currently implemented, thus, each email was saved without any data degradation. The shelve package in the Python Standard Library was used to store a persistent dictionary-like object on disk of the tokenized emails and their bloom filters hashed by the md5 hash. The shelf has a total size of INSERTNUMBERHERE. When serializing the data by directly writing the bytes bloom filter and a string representation of the tokenized email, we find that the total size of bloom filter

is INSERTNUMBERHERE and INSERTNUMBERHERE for the emails, with an average size of INSERTNUMBERHERE for a single bloom filter, and INSERTNUMBERHERE for an email. The md5 hash is 128-bits by definition, which results in a total of INSERTNUMBERHERE.

simple tokenization: unique parsed emails: 248962 unique n-grams: 41488203 split tokenization: unique parsed emails: 243460 unique n-grams: 36274347 moses tokenization: unique parsed emails: 243386 unique n-grams: 29152950

3.4 Email Forgetting

Direct data deletion was used to simulate the data degradation that occurs in a fully implemented forgit. A pre-specified percentage of tokens were randomly dropped. The position of the forgotten tokens needs to be kept for the trie to be used during recall. Therefore, the forgotten tokens were replaced by a null value rather than completely deleted from the email.

3.5 Email Recall

Given the trie data structure, a source email obtained by partially forgetting the target email (a list of tokens or null values), the bloom filter containing the tokens of the target email (if available), and the n-gram length, the following algorithm returns possible reconstructions of the target email.

4 References

Algorithm 1 GENERATE_EMAIL algorithm

procedure GENERATE_EMAIL(tokens, trie, bloom_filter, ngram_length)
▷ *tree_levels* allows us to retrieve nodes based on their depth
tree_levels ← a list containing *length*(*tokens*) empty lists
level ← *ngram_length*
▷ *node_indices* is used to store state for backtracking
node_indices ← a list of 0's of size *length*(*tokens*)
▷ We treat the first *ngram_length* tokens separately
to_search ← first *ngram_length tokens*
$possible\_ngrams \leftarrow FIND\_NGRAMS(to\_search, trie, bloom\_filter, ngram\_length)$
for *ngram* in *possible_ngrams* **do**
for *token* in *ngram* **do**
if *token* is the first token in *ngram* **then**
node ← a tree node with value *token*
else
node ← a tree node with value *token* and parent equal to the previous *node*
end if
tree_levels ← insert *node* in the list corresponding to *node*'s position in *tokens*
end for
end for
▷ Main loop
while $node\_indices[0] < length(graph\_levels[0])$ **do**
▷ If we have looped through all parents, increase the grand-parent index and go back one level
if $node\_indices[level-1] == length(graph\_levels[level-1])$ **then**
$node\_indices[level-2] \leftarrow node\_indices[level-2] + 1$
$level \leftarrow level - 1$
continue
end if
token ← *tokens*[*level*]
$parent\_node \leftarrow graph_levels[level-1][node\_indices[level-1]]$
▷ Sometimes we have no parents as a result of a previous $FIND\_NGRAMS$ call or filtering, so we skip the node
if $length(parent\_node.parents) > 0$ **then**
$prefix \leftarrow$ n-gram of length $ngram\_length - 1$ from *parent_node* and its ancestors
if *token* is `None` **then**
$possible\_ngrams \leftarrow FIND\_NGRAMS(prefix + token, trie, bloom\_filter, ngram\_length)$
for *ngram* in $possible_ngrams$ **do**
end for
end if
end if
end while
end procedure