Foreword

This chapter is based on lecture notes from coding theory courses taught by Venkatesan Guruswami at University at Washington and CMU; by Atri Rudra at University at Buffalo, SUNY and by Madhu Sudan at MIT.

This version is dated **March 30, 2012**. For the latest version, please go to

http://www.cse.buffalo.edu/ atri/courses/coding-theory/book/

The material in this chapter is supported in part by the National Science Foundation under CAREER grant CCF-0844796. Any opinions, findings and conclusions or recomendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation (NSF).

Chapter 13

Cutting Data Down to Size: Hashing

In this chapter, we will study hashing, which is a method to compute a small digest of data that can be used as a surrogate to later perform quick checks on the data. We begin with brief descriptions of three practical applications where hashing is useful. We then formally state the definition of hash functions that are needed in these applications (the so called “universal" hash functions). Next, we will show how in some sense good hashing functions and good codes are equivalent. Finally, we will see how hashing can solve a problem motivated by outsourced storage in the “cloud."

13.1 Why Should You Care About Hashing?

Hashing is one of the most widely used objects in computer science. In this section, we outline three practical applications that heavily use hashing. While describing the applications, we will also highlight the properties of hash functions that these applications need.

Before we delve into the applications, let us first formally define a hash function.

Definition 13.1.1 (Hash Function)**.** *Given a domain $\mathbb{D}$ and a range Σ, (typically, with $|\Sigma| < |\mathbb{D}|$), a hash function is a map*

$$h : \mathbb{D} \to \Sigma.$$

Of course, the definition above is too general and we will later specify properties that will make the definition more interesting.

Integrity Checks on Routers. Routers on the Internet process a lot of packets in a very small amount of time. Among other tasks, router has to perform an “integrity check" on the packet to make sure that the packet it is processing is not corrupted. Since the packet has well defined fields, the router could check if all the field have valid entries. However, it is possible that one of the valid entry could be turned into another valid entry. However, the packet as a whole could still be invalid.

If you have progressed so far in the book, you will recognize that the above is the error detection problem and we know how to do error detection (see e.g., Proposition 2.3.3). However, the algorithms that we have seen in this book are too slow to implement in routers. Hence, Internet protocols use a hash function on a domain $\mathbb{D}$ that encodes all the information that needs to go into a packet. Thus, given an $\mathbf{x} \in \mathbb{D}$, the packet is the pair $(\mathbf{x}, h(\mathbf{x}))$. The sender sends the packet $(\mathbf{x}, h(\mathbf{x}))$ and the receiver gets $(\mathbf{x}', y)$. In order to check if any errors occurred during transmission, the receiver checks if $h(\mathbf{x}') = y$. If the check fails, the receiver asks for a re-transmission otherwise it assumes there were no errors during transmission. There are two requirements from the hash function: (i) It should be super efficient to compute $h(\mathbf{x})$ given $\mathbf{x}$ and (ii) h should avoid “collisions," i.e. if $\mathbf{x} \neq \mathbf{x}'$, then $h(\mathbf{x}) \neq h(\mathbf{x}')$.[1]

Integrity Checks in Cloud Storage. Say, you (as a client) have data $\mathbf{x} \in \mathbb{D}$ that you want to outsource $\mathbf{x}$ to a cloud storage provider. Of course once you “ship" off $\mathbf{x}$ to the cloud, you do not want to store it locally. However, you do not quite trust the cloud either. If you do not audit the cloud storage server in any way, then nothing stops the storage provider from throwing away $\mathbf{x}$ and send you some other data $\mathbf{x}'$ when you ask for $\mathbf{x}$. The problem of designing an auditing protocol that can verify whether the server has the data $\mathbf{x}$ is called the *data possession* problem.

We consider two scenarios. In the first scenario, you access the data pretty frequently during “normal" operation. In such cases, here is a simple check you can perform. When you ship off $\mathbf{x}$ to the cloud, compute $z = h(\mathbf{x})$ and store it. Later when you access $\mathbf{x}$ and the storage provider send you $\mathbf{x}'$, you compute $h(\mathbf{x}')$ and check if it is the same as the stored $h(\mathbf{x})$. This is exactly the same solution as the one for packet verification mentioned above.

Now consider the scenario, where the cloud is used as an archival storage. In such a case, one needs an “auditing" process to ensure that the server is indeed storing $\mathbf{x}$ (or is storing some massaged version from which it can compute $\mathbf{x}$– e.g. the storage provider can compress $\mathbf{x}$). One can always ask the storage provider to send back $\mathbf{x}$ and then use the scheme above. However, if $\mathbf{x}$ is meant to be archived in the cloud, it would be better to resolve the following question:

Question 13.1.1. *Is there an auditing protocol with small client-server communication*[a]*, which if the server passes then the client should be able to certain (with some high confidence) that the server is indeed storing* $\mathbf{x}$*?*

[a] In particular, we rule out solutions where the server sends $\mathbf{x}$ to the client.

We will see later how this problem can be solved using hashing.

Fast Table Lookup. One of the most common operations on databases is the following. Assume there is a table with entries from $\mathbb{D}$. One would like to decide on a data structure to store

[1] Note that in the above example, one could have $\mathbf{x} \neq \mathbf{x}'$ and $h(\mathbf{x}) \neq h(\mathbf{x}')$ but it is still possible that $y = h(\mathbf{x}')$ and hence the corrupted packet $(\mathbf{x}', y)$ would pass the check above. Our understanding is that such occurrences are rare.

the table so that later on given an element $\mathbf{x} \in \mathbb{D}$, one would quickly like to decide whether $\mathbf{x}$ is in the table or now.

Let us formalize the problem a bit: assume that the table needs to store N values $a_1, \dots, a_N \in \mathbb{D}$. Then later given $\mathbf{x} \in \mathbb{D}$ one needs to decide if $\mathbf{x} = a_i$ for some i. Here is one simple solution: sort the n elements in an array T and given $\mathbf{x} \in \mathbb{D}$ use binary search to check if $\mathbf{x}$ is in T or not. This solution uses $\Theta(N)$ amounts of storage and searching for $\mathbf{x}$ takes $\Theta(\log N)$ time. Further, the pre-processing time (i.e. time taken to build the array T) is $\Theta(N \log N)$. The space usage of this scheme is of course optimal but one would like the lookup to be faster: ideally we should be able to perform the search in $O(1)$ time. Also it would be nice to get the pre-processing time closer to the optimal $O(N)$. Further, this scheme is very bad for dynamic data: inserting an item to and deleting an item from T takes $\Theta(N)$ time in the worst-case.

Now consider the following solution: build a boolean array B with one entry for each $z \in \mathbb{D}$ and set $B[a_i] = 1$ for every $i \in [N]$ (and every other entry is 0).[2] Then searching for $\mathbf{x}$ is easy: just lookup $B[\mathbf{x}]$ and check if $B[\mathbf{x}] \neq 0$. Further, this data structure can easily handle addition and deletion of elements (by incrementing and decrementing the corresponding entry of B respectively). However, the amount of storage and pre-processing time are both $\Theta(|\mathbb{D}|)$, which can be much much bigger than the optimal $O(N)$. This is definitely true for tables stored in real life databases. This leads to the following question:

Question 13.1.2. *Is there a data structure that supports searching, insertion and deletion in $O(1)$ time but only needs $O(N)$ space and $O(N)$ pre-processing time?*

We will see later how to solve this problem with hashing.

13.2 Avoiding Hash Collisions

One of the properties that we needed in the applications outlined in the previous section was that the hash function $h : \mathbb{D} \to \Sigma$ should avoid collisions. That is, given $\mathbf{x} \neq y \in \mathbb{D}$, we want $h(\mathbf{x}) \neq h(y)$. However, since we have assumed that $|\Sigma| < |\mathbb{D}|$, this is clearly impossible. A simple counting argument shows that there will exist an $\mathbf{x} \neq y \in \mathbb{D}$ such that $h(\mathbf{x}) = h(y)$. There are two ways to overcome this hurdle.

The first is to define a cryptographic collision resistant hash function h, i.e. even though there exists collisions for the hash function h, it is computationally hard for an adversary to compute $\mathbf{x} \neq y$ such that $h(\mathbf{x}) = h(y)$.[3] This approach is out of the scope of this book and hence, we will not pursue this solution.

[2] If one wants to handle duplicates, one could store the number of occurrences of y in $B[y]$.

[3] This is a very informal definition. Typically, an adversary is modeled as a randomized polynomial time algorithm and there are different variants on whether the adversary is given $h(\mathbf{x})$ or $\mathbf{x}$ (or both). Also there are variants where one assumes a distribution on $\mathbf{x}$. Finally, there are no unconditionally collision resistant hash function but there exists provably collision resistant hash function under standard cryptographic assumptions: e.g. factoring is hard.

The second workaround is to define a family of hash functions and then argue that the probability of collision is small for a hash function chosen randomly from the family. More formally, we define a hash family:

Definition 13.2.1 (Hash Family)**.** *Given $\mathbb{D}, \Sigma$ and an integer $m \geq 1$, a hash family $\mathscr{H}$ is a set $\{h_1, \ldots, h_m\}$ such that for each $i \in [m]$,*

$$h_i : \mathbb{D} \to \Sigma.$$

Next we define the notion of (almost) universal hash function (family).

Definition 13.2.2 (Almost Universal Hash Family)**.** *A hash family $\mathscr{H} = \{h_1, \ldots, h_m\}$ defined over the domain $\mathbb{D}$ and range Σ is said to be ε-almost universal hash function (family) for some* $0 < \varepsilon \leq 1$ *if for every* $\mathbf{x} \neq y \in \mathbb{D}$,

$$\Pr_i \left[h_i(\mathbf{x}) = h_i(y) \right] \leq \varepsilon,$$

where in the above i is chosen uniformly at random from $[m]$.

We will show in the next section that ε-almost universal hash functions are equivalent to codes with (large enough) distance. In the rest of the section, we outline how these hash families provides satisfactory solutions to the problems considered in the previous section.

Integrity Checks. For the integrity check problem, one pick random $i \in [m]$ and chooses $h_i \in \mathscr{H}$, where $\mathscr{H}$ is an ε-almost universal hash function. Thus, for any $\mathbf{x} \neq y$, we're guaranteed with probability at least $1 - \varepsilon$ (over the random choice of i) that $h_i(\mathbf{x}) \neq h_i(y)$. Thus, this gives a randomized solution to the integrity checking problem in routers and cloud storage (where we consider the first scenario in which the cloud is asked to return the original data in its entirety).

It is not clear whether such hash functions can present a protocol that answers Question 13.1.1. There is a very natural protocol to consider though. When the client ships off data $\mathbf{x}$ to the cloud, it picks a random hash function $h_i \in \mathscr{H}$, where again $\mathscr{H}$ is an ε-universal hash function, and computes $h_i(\mathbf{x})$. Then it stores $h_i(\mathbf{x})$ and ships off $\mathbf{x}$ to the cloud. Later on, when the client wants to audit, it asks the cloud to send $h_i(\mathbf{x})$ back to it. Then if the cloud returns with z, the client checks if $z = h_i(\mathbf{x})$. If so, it assumes that the storage provider is indeed storing $\mathbf{x}$ and otherwise it concludes that the cloud is not storing $\mathbf{x}$.

Note that it is crucial that the hash function be chosen randomly: if the client picks a deterministic hash function h, then the cloud can store $h(\mathbf{x})$ and throw away $\mathbf{x}$ because it knows that the client is only going to ask for $h(\mathbf{x})$. Intuitively, the above protocol might work since the random index $i \in [m]$ is not known to the cloud till the client asks for $h_i(\mathbf{x})$, it seems "unlikely" that the cloud can compute $h_i(\mathbf{x})$ without storing $\mathbf{x}$. We will see later how the coding view of almost universal hash functions can make this intuition rigorous.

Fast Table Lookup. We now return to Question 13.1.2. The basic idea is simple: we will modify the earlier solution that maintained an entry for each element in the domain $\mathbb{D}$. The new solution will be to keep an entry for all possible hash values (instead of all entries in $\mathbb{D}$).

More formally, let $\mathscr{H} = \{h_1, \dots, h_m\}$ be an ε-almost hash family with domain $\mathbb{D}$ and range Σ. Next we build an array of link list with one entry in the array for each value $v \in \Sigma$. We pick a random hash function $h_i \in \mathscr{H}$. Then for each a_j ($j \in [N]$) we add it to the link list corresponding to $h_i(a_j)$. Now to determine whether $\mathbf{x} = a_j$ for some j, we scan the link list corresponding to $h_i(\mathbf{x})$ and check if $\mathbf{x}$ is in the list or not. Before we analyze the space and time complexity of this data structure, we point out that insertion and deletion are fairly easy. For inserting an element $\mathbf{x}$, we compute $h_i(\mathbf{x})$ and add $\mathbf{x}$ to the link list corresponding to $h_i(\mathbf{x})$. For deletion, we first perform the search algorithm and then remove $\mathbf{x}$ from the list corresponding to $h_i(\mathbf{x})$, if it is present. It is easy to check that the algorithms are correct.

Next we analyze the space complexity. Note that for a table with N elements, we will use up $O(N)$ space in the linked lists and the array is of size $O(|\Sigma|)$. That is, the total space usage is $O(N + |\Sigma|)$. Thus, if we can pick $|\Sigma| = O(N)$, then we would match the optimal $O(N)$ bound for space.

Now we analyze the time complexity of the various operations. We first note that insertion is $O(1)$ time (assuming computing the hash value takes $O(1)$ time). Note that this also implies that the pre-processing time is $O(N + |\Sigma|)$, which matches the optimal $O(N)$ bound for $|\Sigma| \le O(N)$. Second, for deletion, the time taken after performing the search algorithm is $O(1)$, assuming the lists as stored as doubly linked lists. (Recall that deleting an item from a doubly linked list if one has a pointer to the entry can be done in $O(1)$ time.)

Finally, we consider the search algorithm. Again assuming that computing a hash value takes $O(1)$ time, the time complexity of the algorithm to search for $\mathbf{x}$ is dominated by size of the list corresponding to $h_i(\mathbf{x})$. In other words, the time complexity is determined by the number of a_j that collide with $\mathbf{x}$, i.e., $h_i(\mathbf{x}) = h_i(a_j)$. We bound this size by the following simple observation.

Claim 13.2.3. *Let $\mathscr{H} = \{h_1, \dots, h_m\}$ with domain $\mathbb{D}$ and range Σ be an ε-almost universal hash family. Then the following is true for any (distinct) $\mathbf{x}, a_1, a_2, \dots, a_N \in \mathbb{D}$:*

$$\mathbb{E}_i \left[|\{a_j | h_i(\mathbf{x}) = h_i(a_j)\}| \right] \le \varepsilon \cdot N,$$

where the expectation is taken over a uniformly random choice of $i \in [m]$.

Proof. Fix a $j \in [N]$. Then by definition of an ε-almost universal hash family, we have that

$$\Pr_i[h_i(\mathbf{x}) = h_i(a_j)] \le \varepsilon.$$

Note that we want to bound $\mathbb{E}\left[\sum_{j=1}^{N} \mathbb{1}_{h_i(a_j)=h_i(\mathbf{x})}\right]$. The probability bound above along with the linearity of expectation (Proposition 3.1.4) and Lemma 3.1.3 completes the proof. □

The above discussion then implies the following:

Proposition 13.2.4. *Given an $O\left(\frac{1}{N}\right)$-almost universal hash family with domain $\mathbb{D}$ and range Σ such that $|\Sigma| = O(N)$, there exists a randomized data structure that given N elements $a_1, \dots, a_N \in \mathbb{D}$, supports searching, insertion and deletion in expected $O(1)$ time while using $O(N)$ space in the worst-case.*

Thus, Proposition 13.2.4 answers Question 13.1.2 in the affirmative if we can answer the following question in the affirmative:

Question 13.2.1. *Given a domain $\mathbb{D}$ and an integer $N \geq 1$, does there exist an $O\left(\frac{1}{N}\right)$-almost universal hash function with domain $\mathbb{D}$ and a range Σ such that $|\Sigma| = O(N)$?*

We will answer the question above (spoiler alert!) in the affirmative in the next section.

13.3 Almost Universal Hash Function Families and Codes

In this section, we will present a very strong connection between almost universal hash families and codes with good distance: in fact, we will show that they are in fact *equivalent.*

We first begin by noting that any hash family has a very natural code associated with it and that every code has a very natural hash family associated with it.

Definition 13.3.1. *Given a hash family $\mathscr{H} = \{h_1, \dots, h_n\}$ where for each $i \in [n]$, $h_i : \mathbb{D} \to \Sigma$, consider the following associated code*

$$C_{\mathscr{H}} : \mathbb{D} \to \Sigma^n,$$

where for any $\mathbf{x} \in \mathbb{D}$, we have

$$C_{\mathscr{H}}(\mathbf{x}) = (h_1(\mathbf{x}), h_2(\mathbf{x}), \dots, h_n(\mathbf{x}))\,.$$

The connection also goes the other way. That is, given an $(n,k)_\Sigma$ code C, we call the associated hash family $\mathscr{H}_C = \{h_1, \dots, h_n)$, where for every $i \in [n]$,

$$h_i : \Sigma^k \to \Sigma$$

such that for every $\mathbf{x} \in \Sigma^k$ and $i \in [n]$,

$$h_i(\mathbf{x}) = C(\mathbf{x})_i.$$

Next we show that an ε-almost universal hash family is equivalent to a code with good distance.

Proposition 13.3.2. *Let $\mathscr{H} = \{h_1, \dots, h_n\}$ be an ε-almost universal hash function, then the code $C_{\mathscr{H}}$ has distance at least $(1-\varepsilon)n$. On the other hand if C is an $(n, k, \delta n)$-code, then $\mathscr{H}_C$ is a $(1-\delta)$-almost universal hash function.*

Proof. We will only prove the first claim. The proof of the second claim is essentially identical and is left as an exercise.

Let $\mathscr{H} = \{h_1, \dots, h_n\}$ be an ε-almost universal hash function. Now fix arbitrary $\mathbf{x} \neq \mathbf{y} \in \mathbb{D}$. Then by definition of $C_{\mathscr{H}}$, we have

$$\{i | h_i(\mathbf{x}) = h_i(\mathbf{y})\} = \{i | C_{\mathscr{H}}(\mathbf{x})_i = C_{\mathscr{H}}(\mathbf{y})_i\}.$$

This implies that

$$\Pr_i\left[h_i(\mathbf{x}) = h_i(\mathbf{y})\right] = \frac{|\{i|h_i(\mathbf{x}) = h_i(\mathbf{y})\}|}{n} = \frac{n - \Delta(C_{\mathcal{H}}(\mathbf{x}), C_{\mathcal{H}}(\mathbf{y}))}{n} = 1 - \frac{\Delta(C_{\mathcal{H}}(\mathbf{x}), C_{\mathcal{H}}(\mathbf{y}))}{n},$$

where the second equality follows from the definition of the Hamming distance. By the definition of ε-almost universal hash family the above probability is upper bounded by ε, which implies that

$$\Delta(C_{\mathcal{H}}(\mathbf{x}), C_{\mathcal{H}}(\mathbf{y})) \geq n(1-\varepsilon).$$

Since the choice of $\mathbf{x}$ and $\mathbf{y}$ was arbitrary, this implies that $C_{\mathcal{H}}$ has distance at least $n(1-\varepsilon)$ as desired. □

13.3.1 The Polynomial Hash Function

We now consider the hash family corresponding to a Reed-Solomon code. In particular, let C be a $[q, k, q-k+1]_q$ Reed-Solomon code. By Proposition 13.3.2, the hash family $\mathcal{H}_C$ is an $\frac{k-1}{q}$-almost universal hash family– this hash family in the literature is called the polynomial hash. Thus, if we pick q to be the smallest power of 2 larger than N and pick $k = O(1)$, then this leads to an $O(1/N)$-universal hash family that satisfies all the required properties in Question 13.2.1.

Note that the above implies that $|\mathbb{D}| = N^{O(1)}$. One might wonder if we can get an $O(1/N)$-almost universal hash family with the domain size being $N^{\omega(1)}$. We leave the resolution of this question as an exercise.

13.4 Data Possession Problem

In this section, we return to Question 13.1.1. Next we formalize the protocol for the data possession problem that we outlined in Section 13.2. Algorithm 7 presents the pre-processing step.

Algorithm 7 Pre-Processing for Data Possession Verification

```
INPUT: Data $\mathbf{x} \in \mathbb{D}$, hash family $\mathcal{H} = \{h_1, \ldots, h_m\}$ over domain $\mathbb{D}$

1: Client computes an index $i$ for $\mathbf{x}$.
2: Client picks a random $j \in [m]$.
3: Client computes $z \leftarrow h_j(\mathbf{x})$ and stores $(i, j, z)$.
4: Client sends $\mathbf{x}$ to the server.
```

Algorithm 8 formally states the verification protocol. Note that if the server has stored $\mathbf{x}$ (or is able to re-compute $\mathbf{x}$ from what it had stored), then it can pass the protocol by returning $a \leftarrow h_j(\mathbf{x})$. Thus, for the remainder of the section, we will consider the case when the server tries to cheat. We will show that if the server is able to pass the protocol in Algorithm 8 with high enough probability, then the server indeed has stored $\mathbf{x}$.

Before we formally prove the result, we state our assumptions on what the server can and cannot do. We assume that the server follows the following general protocol. First, when the

Algorithm 8 Verification for Data Possession Verification

```
INPUT: Index $i$ of data $\mathbf{x} \in \mathbb{D}$
OUTPUT: 1 if Server has $\mathbf{x}$ and 0 otherwise

Client sends a challenge $(i, j)$ to the server.
Client receives an answer $a$.
IF $a = z$ THEN
    RETURN 1
ELSE
    RETURN 0
```

server receives $\mathbf{x}$, it does performs some computation (that terminates) on $\mathbf{x}$ to produce $\mathbf{y}$ and then it stores $\mathbf{y}$. (E.g., the server could store $\mathbf{y} = \mathbf{x}$ or $\mathbf{y}$ could be a compressed version of $\mathbf{x}$.) Then when it receives the challenge (i, j) for $\mathbf{x}$, it uses another algorithm $\mathscr{A}$ and returns the answers $a \leftarrow \mathscr{A}(\mathbf{y}, j)$. We assume that $\mathscr{A}$ always terminates on any input.[4] Note that the server is allowed to use arbitrary (but finite) amount of time to compute its answer. Next, we will prove that under these assumptions, the server cannot cheat with too high a probability.

Theorem 13.4.1. *Assume that the hash family $\mathscr{H}$ is an ε-almost universal hash family. Then if the server passes the protocol in Algorithm 8 with probability $> \frac{1}{2} + \frac{\varepsilon}{2}$, then the server has enough information to recreate* $\mathbf{x}$.

Proof. To prove the claim, we present an algorithm that can compute $\mathbf{x}$ from $\mathbf{y}$. (Note that we do not need this algorithm to be efficient: it just needs to terminate with $\mathbf{x}$.) In particular, consider Algorithm 9.

Algorithm 9 Decompression Algorithm

```
INPUT: $\mathscr{A}, \mathbf{y}$
OUTPUT: $\mathbf{x}'$

$\mathbf{z} \leftarrow \left(\mathscr{A}(\mathbf{y}, j)\right)_{j \in [m]}$.
Run the MLD algorithm (Algorithm 1) for $C_{\mathscr{H}}$ on $\mathbf{z}$ and let $C_{\mathscr{H}}(\mathbf{x}')$ be its output.
RETURN $\mathbf{x}'$
```

To complete the proof, we will show that $\mathbf{x}' = \mathbf{x}$. Towards this end we claim that $\Delta(\mathbf{z}, C_{\mathscr{H}}(\mathbf{x})) < \frac{m}{2} \cdot (1 - \varepsilon)$. Assuming this is true, we complete the proof. Note that Proposition 13.3.2 implies that $C_{\mathscr{H}}$ has distance at least $m(1 - \varepsilon)$. Thus, Proposition 1.4.3 (in particular, its proof) implies that Algorithm 1 will return $C_{\mathscr{H}}(\mathbf{x})$ and thus, $\mathbf{x}' = \mathbf{x}$, as desired.

Finally, we argue that $\Delta(\mathbf{z}, C_{\mathscr{H}}(\mathbf{x})) < m(1 - \varepsilon)/2$. To see this note that if the server passes the protocol in Algorithm 8 (i.e. the client outputs 1), then it has to be the case that $z_j \overset{\text{def}}{=} \mathscr{A}(\mathbf{y}, j) =$

[4] We have stated the algorithm to be independent of $\mathbf{y}$ and j but that need not be the case. However later in the section, we will need the assumption that $\mathscr{A}$ is independent of $\mathbf{y}$ and j, so we will keep it that way.

$h_j(\mathbf{x})$. Recall that by definition of $C_{\mathcal{H}}$, $h_j(\mathbf{x}) = C_{\mathcal{H}}(\mathbf{x})_j$ and that the server passes the protocol with probability $> 1/2+\varepsilon/2$. Since j is chosen uniformly from $[m]$, this implies that for $> m(1/2+\varepsilon/2)$ positions j, $z_j = C_{\mathcal{H}}(\mathbf{x})_j$, which then implies the claim. □

13.4.1 Driving Down the Cheating Probability

One of the unsatisfying aspects of Theorem 13.4.1 is that the probability of catching a "cheating" server is strictly less than 50%.[5] It is of course desirable to drive this up as close to 100% as possible. One way to obtain this would be to "repeat" the protocol: i.e. the client can choose multiple random hashes and store the corresponding values (in Algorithm 7) and (in Algorithm 8) asks the server to send back all the hash values and accepts if and only if all the returned answers match with the stored hash valiues. This however, comes at a cost: the client has to store more hash values (and also the communication cost betwen the client and the server goes up accordingly.)

Next we argue using list decoding that the protocol in Algorithm 7 and 8 as is gives a more powerful guarantee than the one in Theorem 13.4.1. To see why list decoding might buy us something, let us look back at Algorithm 9. In particular, consider Step 2: since we run MLD, we can only guarantee unique decoding up to half the (relative) distance of $C_{\mathcal{H}}$. This in turn leads to the bound of $1/2+\varepsilon/2$ in Theorem 13.4.1. We have seen that list decoding allows us to beyond half the distance number of errors. So maybe, running a list decoding algorithm instead of MLD in Step 2 of Algorithms 9 would help us get better results. There are two potential issues that we'll need to tackle:

- We will need a general bound that shows that list decoding is possible close to 100% for any $C_{\mathcal{H}}$ for an ε-almost universal hash family; and
- Even if the above is possible, what will we do when a list decoding algorithm returns a *list* of possible data?

We will get around the first concern by using the Johnson bound 7.3.1. To get around the second issue we will indirectly use "side information" (like we mentioned in Section 7.2). For the latter, we will need the notion of Kolmogorov complexity, which captures the amount of information content in any given string. For our purposes, the following informal description is enough:

Definition 13.4.2. *Given a string* $\mathbf{x}$*, its Kolmogorov complexity, denoted by* $\mathcal{K}(\mathbf{x})$ *is the minimum of* $|\mathbf{y}| + |\mathcal{D}|$*, where* $\mathcal{D}$ *is a decompression algorithm that on input* $\mathbf{y}$ *outputs* $\mathbf{x}$ *(where* $|\mathbf{x}|$ *and* $|\mathcal{D}|$ *are the length of* $\mathbf{x}$ *and (a description of)* $\mathcal{D}$ *in bits).*

Informally, $\mathcal{K}(\mathbf{x})$ is the amount of information that can be obtained algorithmically from $\mathbf{x}$. Kolmogorov complexity is a fascinating topic that it outside the scope of this book. Here we will only need to use the definition of $\mathcal{K}(\mathbf{x})$. We are now ready to prove the following list decoding counterpart of Theorem 13.4.1:

[5] Note that Theorem 13.4.1 states that a server that cannot recreate $\mathbf{x}$ can pass the test with probablity at most $1/2+\varepsilon/2$. In other words, the probability that such a server is caught is at most $1/2-\varepsilon/2 < 1/2$.

Theorem 13.4.3. *Assume that the hash family $\mathscr{H}$ is an ε-almost universal hash family. Then if the server passes the protocol in Algorithm 8 with probability $> \sqrt{\varepsilon}$, then the amount of information server has stored for* $\mathbf{x}$ *is at least* $\mathscr{K}(\mathbf{x}) - O(\log|\mathbf{x}|)$.

We note that the above is not a strict generalization of Theorem 13.4.1, as even though probability of catching a cheating server has gone up our guarantee is weaker. Unlike Theorem 13.4.1, where we can guarantee that the server can re-create $\mathbf{x}$, here we can only guarantee “storage enforceability"– i.e. we can only force a server to store close to $\mathscr{K}(\mathbf{x})$ amounts of memory.

Proof of Theorem 13.4.3. Here is the main idea of the proof. We first assume for the sake of contradiction that $|\mathbf{y}| < \mathscr{K}(\mathbf{x}) - O(\log(|\mathbf{x}|))$. Then using we construct a decompression algorithm $\mathscr{D}$ that on given input $\mathbf{y}$ and $O(\log(|\mathbf{x}|))$ extra information (or “advice"), outputs $\mathbf{x}$. Then we will show this overall contradicts the definition of $\mathscr{K}(\mathbf{x})$ (as this gives an overall smaller description of $\mathbf{x}$).

Before we state the decompression algorithm, we recall some facts. First note that $C_{\mathscr{H}}$ by Proposition 13.3.2 is a q-ary code (with $|\Sigma| = q$) with distance $m(1-\varepsilon)$. Further, by the Johnson bound (Theorem 7.3.1), $C_{\mathscr{H}}$ is a $(1-\sqrt{\varepsilon}, L)$-list decodable, where

$$L \le qm^2. \tag{13.1}$$

Next, in Algorithm 10, we present a decompression algorithm that can compute $\mathbf{x}$ from $\mathbf{y}$ and an advice string $a \in [L]$. (As before, we do not need this algorithm to be efficient: it just needs to terminate with $\mathbf{x}$.)

Algorithm 10 Decompression Algorithm Using List Decoding

```
INPUT: $\mathscr{A}, \mathbf{y}, a$
OUTPUT: $\mathbf{x}$

z ← $(\mathscr{A}(\mathbf{y}, j))_{j \in [m]}$.
$\mathscr{L} \leftarrow \emptyset$.
FOR $\mathbf{x}' \in \mathbb{D}$ DO
    IF $\Delta(C_{\mathscr{H}}(\mathbf{x}'), \mathbf{z}) \le (1-\sqrt{\varepsilon})m$ THEN
        Add $\mathbf{x}'$ to $\mathscr{L}$
RETURN The $a$th element in $\mathscr{L}$
```

To complete the proof, we claim that there exists a choice of $a \in [L]$ such that Algorithm 10 outputs $\mathbf{x}$. Note that this implies that $(\mathbf{y}, a)$ along with Algorithm 10 gives a complete description of $\mathbf{x}$. Now note that Algorithm 10 can be described in $O(1)$ bits. This implies that the size of this description is $|\mathbf{y}| + \log L + O(1)$, which by Definition 13.4.2 has to be at least $\mathscr{K}(\mathbf{x})$. This implies that

$$|\mathbf{y}| \ge \mathscr{K}(\mathbf{x}) - |a| - O(1) = \mathscr{K}(\mathbf{x}) - \log L - O(1) \ge \mathscr{K}(\mathbf{x}) - O(\log|\mathbf{x}|),$$

where the last inequality follows from (13.1).

Next, we argue the existence of an appropriate $a \in [L]$. Towards this end we claim that $\Delta(\mathbf{z}, C_{\mathcal{H}}(\mathbf{x})) < m(1-\sqrt{\varepsilon})$. Note that this implies that $\mathbf{x} \in \mathcal{L}$. Since $|\mathcal{L}| \le L$, then we can just assign a to be the index of $\mathbf{x}$ in $\mathcal{L}$. Finally, we argue that $\Delta(\mathbf{z}, C_{\mathcal{H}}(\mathbf{x})) < m(1-\sqrt{\varepsilon})$. To see this note that if the server passes the protocol in Algorithm 8 (i.e. the client outputs 1), then it has to be the case that $z_j \overset{\text{def}}{=} \mathscr{A}(\mathbf{y}, j) = h_j(\mathbf{x})$. Recall that by definition of $C_{\mathcal{H}}$, $h_j(\mathbf{x}) = C_{\mathcal{H}}(\mathbf{x})_j$ and that the server passes the protocol with probability $> \sqrt{\varepsilon}$. Since j is chosen uniformly from $[m]$, this implies that for $> m\sqrt{\varepsilon}$ positions j, $z_j = C_{\mathcal{H}}(\mathbf{x})_j$, which then implies the claim. □

13.5 Bibliographic Notes

Universal hash functions were defined in the seminal paper of Carter and Wegman [3]. Almost universal hash function family was defined by Stinson [48].

Kolmogorov complexity was defined by Kolmogorov [32]. For a thorough treatment see the textbook by Li and Vitányi [34].