The Binomial Ladder Frequency Filter and its Applications to Shared Secrets

Stuart Schechter[†]
Unaffiliated
stuart@post.harvard.edu

Cormac Herley
Microsoft Research, Redmond, WA, USA
cormac@microsoft.com

Abstract—**We introduce the binomial ladder filter: a data structure used to examine a stream of values to identify those that occur frequently, while keeping the identity of infrequent values private. Password-protected services may use the filter when users choose passwords, to isolate the common (weak) passwords and forbid their use—without storing or revealing infrequent (likely-strong) passwords. Similarly, when users login, they may filter the stream of passwords from failed login attempts to isolate frequent incorrect passwords, which result when attackers guess a password they believe to be common against many user accounts. The filter would ignore passwords from users' failed logins, which exhibit errors that are broadly distributed and infrequent (even if collectively more frequent). Among the features of the binomial ladder filter are that it protects the privacy both of the frequency of values in a stream and the identity of the values themselves; its size does not grow with the number of elements in the stream or the number of possible values any element can represent; it does not need to be replaced or re-initialized with age; and with small modifications it can be implemented efficiently in highly-distributed systems with minimal coordination.**

I. Introduction

A frequency filter is an online data structure that is used to examine a stream of elements (values), identifying frequently-occurring values while filtering out rare elements, as illustrated in Figure 1. Systems that rely on passwords, PINs, or other shared secrets can apply frequency filters to detect users' 'popular' choices and prevent more users from relying on secrets that are already too frequent among the user base [15]. Password-based authentication systems can also use frequency filters to examine the passwords submitted in failed login attempts, identify passwords being frequently-guessed by attackers, and use this information to more quickly and accurately block hosts engaged in guessing [2].

We introduce a privacy-preserving frequency filter that not only limits what the output reveals about individual contributions to the frequency of elements in the stream, but that also protects the very identity of infrequent elements—even in the event that the internal data structures of the filter are captured by an adversary. Systems can use this filter to track users frequently-chosen passwords *without* maintaining a list of infrequently-observed passwords. They can identify passwords that frequently appear in incorrect login attempts without keeping records of incorrect passwords that resulted from users' unique (or relative rare) mistakes.

[†]Work performed while the author was at Microsoft.

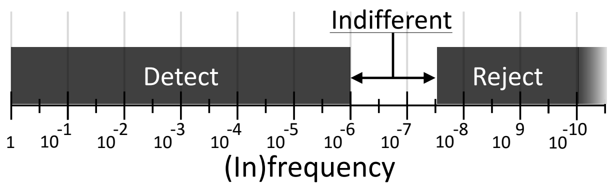
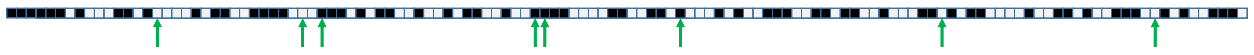


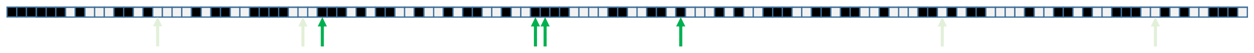
Fig. 1: A frequency filter is used to examine a stream of values to identify frequently-occurring values while filtering out infrequent values. In other words, the filter should detect values that arrive at a rate that exceeds the detection threshold: a frequency of one in one million, or 10^{-6}, in this illustration. The filter should reject values that arrive at a rate below the rejection threshold: a frequency of one in fifty million, or $2 \cdot 10^{-8}$, in this illustration. The filter may detect or reject values that arrive at rates between the two thresholds—a segment of the frequency range that we call the *indifference* region.

This *binomial ladder filter* is an approximate data structure in the spirit of Bloom filters [5]. Applications configure the filter to detect very-frequent elements that occur at a rate above a specified detection threshold and to reject those elements that occur at a rate below a specified rejection threshold. In between these two frequencies lies an indifference region, which contains elements that neither occur very frequently nor very rarely, and which the filter is free to treat either as frequent or rare.

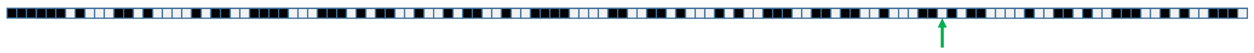
We describe the design of the binomial ladder filter in Section II and describe its modes of operation and behavior in Section III. We explain the mathematics underlying its probabilistic behavior, and our algorithmic approach to precisely calculate detection probabilities, in Section IV. We present a model for understanding the filter's privacy guarantees in Section V. We describe how to configure the filter for different applications in Section VI. We explain how the filter can be modified to scale to distributed systems, and do so with minimal coordination overhead, in Section VII. We provide case studies of how the filter can be used in password policies and to prevent guessing attacks in Section VIII. We compare the filter with a number of existing tools for identifying frequent values and protecting privacy in Section IX.



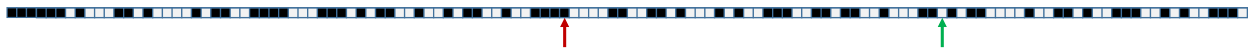
(a) We begin the `height` operation by associating the element with $\mathbf{H} = 8$ bits of the array, which we will refer to as the rungs of the element's ladder. To do so we invoke the filter's **H** hash functions to index into **H** bits of the array. The green arrows above represent one element's rungs.



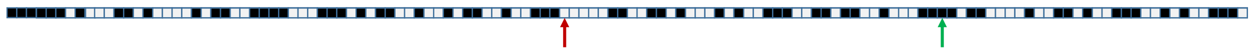
(b) The `height` operation returns the number of rungs *below* the element: it counts the number of one bits associated with the element. We use four green arrows to show which four rungs of the eight in Subfigure (a) that should be counted (the one bits).



(c) We begin the `step` operation by identifying the rungs (bits) associated with the element, as shown in step (a) above. We then select one of the rungs above the element (a zero bit) at random with uniform probability, shown above by the remaining arrow. (If there are no rungs above, we select a zero bit at random from the entire filter array.)



(d) From all the one bits in the entire array that are *not* among this element's rungs, we select one at random (with uniform probability). In the example, we identify this bit with a red arrow.



(e) We swap the values of the two selected bits, setting the element's rung to one and clearing the bit that is not associated with the element. The swap increases the element's height on its ladder while preserving the invariant that the number of zero and one bits in the array remains constant.

Fig. 2: An example of the `height` operation to identify the height of an element on the binomial ladder (a,b) and the `step` operation (c-e) to raise the height of the element on its ladder by one rung. The filter in this example has $\mathbf{N} = 128$ bits and $\mathbf{H} = 8$ hash functions that associate each element to a subset of those bits (rungs).

II. Design

Like a Bloom filter [5], a binomial ladder filter pairs an array of **N** bits with a family of **H** hash functions used to index into the array. The hash functions *associate* elements (values) with **H** different[1] bits in the array, which we call *rungs*. Together, these **H** rungs make up an element's *binomial ladder*. Continuing with the ladder analogy, we say that an element has climbed those rungs that have value one (it is *above* them on the ladder) and has yet to climb those rungs with value zero (it is *below* them). The height h of an element on its ladder is the number of these rungs below it: the number of associated bits with value one.

A. Initialization

Upon construction, we initialize a random subset of half of the bits of the binomial ladder filter to be one and the remaining half to be zero. The equal ratio of zero bits to one bits stays constant during the lifetime of the filter (in Section VII we relax this constraint slightly to accommodate distributing the filter over multiple hosts).

B. Operations

The binomial ladder filter supports two fundamental operations, `height` and `step`, as illustrated in detail in Figure 2.

`Height`(element): The `height` operation returns the height of an element: the number of bits associated with the element (they element's **H** *rungs*) that are set to one (placing them below the element on its ladder).

[1]In the event that a hash function yields a position already indexed by an earlier (lower-indexed) hash function, re-hashing can ensure that the set of rungs represent distinct positions in the filter array.

`Step`(element): The `step` operation increases the element's height by one rung, while maintaining the invariant that the number of zero and one bits must be equal. To do so it first identifies the bits associated with the element (the element's rungs), identifying the rungs above the element (the zero bits) and those below the element (the one bits). It then selects one of the rungs that are zero bits at random (with uniform probability) with the intent of flipping its value to one, which would push the rung below the element and in so doing raise the element's height up by that one rung. If there are no rungs above the element (the element is already at the top of its ladder), `step` picks a zero element at random from the set of zero elements in the entire array.

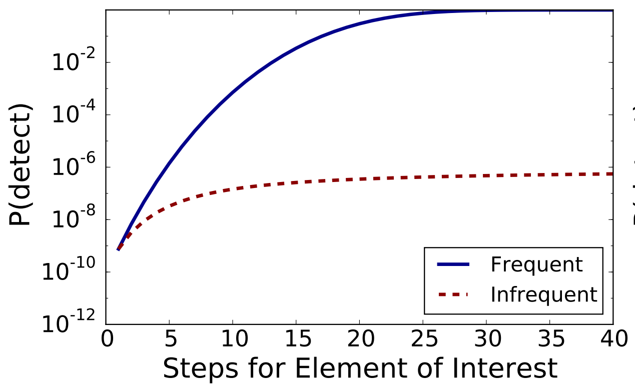
To ensure that the number of zero and one bits in the filter's array stays constant, `step` must select a one bit from the array to clear to zero. It selects this one bit uniformly and at random from the set of one bits in the entire array, excluding those associated with the element that is taking the step.

The `step` operation then swaps the values of the zero bit to be set and the one bit to be cleared, preserving the invariant that the number of zero and one bits remains constant. Moving the rung from above the element (its bit had value zero) down below (setting the bit to one) raises the element one rung higher on its ladder.

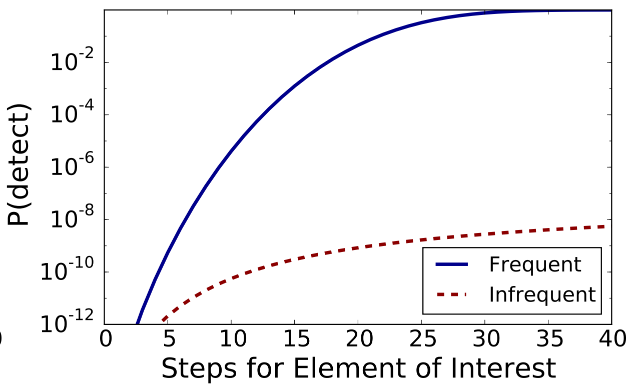
The `step` operation returns the height of the element *before* the operation took place.

III. Application Usage Modes

We analyze the binomial ladder filter for two important usage modes. In both modes, applications calls `step` for each element in a stream regardless of the value of the element.



(a) Perpetual mode (**T**=44)



(b) Sticky mode (**T**=48)

Fig. 3: The probability a binomial ladder filter ($\mathbf{H} = 48$, $\mathbf{N} = 2^{29}$) will correctly detect frequent elements that arrive at the detection frequency (one in a million in this example) and falsely detect infrequent elements that arrive at the rejection frequency (one in fifty million). After a ramp-up period, the behavior closely approximates the ideal behavior shown in Figure 1. (Note that since infrequent elements occur at a rate that is fifty times lower than frequent elements, it takes $50\times$ longer for an infrequent element to take the same number of `steps`, with $50\times$ more `steps` taken for other elements.) Subfigure (a) shows a filter operating in perpetual mode that registers a detection when the element is at ladder height threshold $\mathbf{T} = 44$ prior to a step. Subfigure (b) shows a filter operating in sticky mode that will permanently classify an element as frequent if it is at the top of its ladder ($\mathbf{T} = 48$) when `step` is called. The lower threshold used for perpetual mode results in faster detection, but a higher rate of false positives in the short-run. The slow continual increase in the false-positive rate of sticky mode will, if run long enough, eventually outpace the false-positive rate for perpetual mode, which flattens out over time.

A. *Perpetual mode*

In *perpetual* mode, an element is treated as frequent if its height at the start of the `step` operation exceeds a height threshold for detection **T**. Since even the most-frequently observed elements may sometimes drop a rung on their ladders due to bits cleared at random by `steps` for other elements, applications should choose a threshold **T** one or more rungs below the top of its ladder **H**. Frequently-observed elements that become infrequent will eventually drop below the height threshold and again be treated as infrequent.

In Figure 3a we illustrate the detection rates for a binomial ladder filter used in perpetual mode, using analysis techniques presented in Section IV.

B. *Sticky mode*

In *sticky* mode, the application uses the binomial ladder filter only to identify when an element *first* exceeds the detection threshold, relying on a separate data structure to maintain the set of detected elements. The application will add an element to the detection set when `step` is called on an element that is already at the height threshold on its ladder—typically at the top. Depending on the data structure the application uses to track elements that have been identified by the filter, it may come to include elements that are no longer frequent and its size may be unbounded.

In Figure 3b we show the detection rates for a binomial ladder filter used in sticky mode, also using the analysis techniques below.

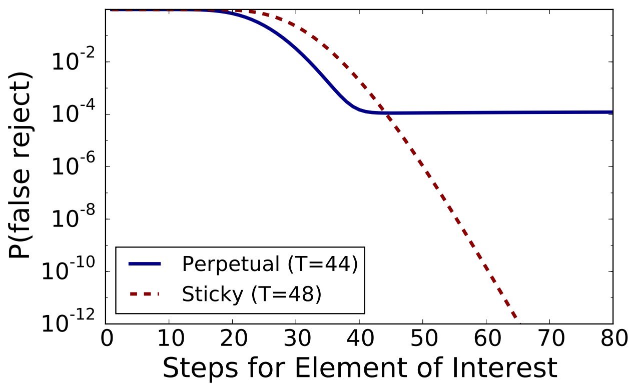


Fig. 4: The probability that a binomial ladder filter ($\mathbf{H} = 48$, $\mathbf{N} = 2^{29}$) will incorrectly reject elements that occur at the detection frequency threshold: one in one million in this example. We cannot detect the element until it occurs a sufficient number of times in the stream to be deemed frequent, and so some initial number of occurrences will always be false rejections. In perpetual mode, the chance of a false rejection levels out over time because there is always the possibility a cluster of drops will occur between two arrivals of the element. In sticky mode, once an element is detected it will always be treated as detected, so the chance of false rejection approaches zero.

IV. Computing Detection Rates

To understand how to compute the probability that a binomial ladder filter will detect a value that occurs with a given frequency, we begin by recalling that the only operation that changes the state of the filter is `step`. The height of an element of interest may change when it is passed to the `step` operation, or when another element passed to the step operation flips a bit associated with the element of interest (one of the element of interest's rungs).

Examining the potential effects of the `step` operation case by case, an element of interest, $\mathbf{e}$, may rise in height when another element, takes a `step` causing one of the rungs above $\mathbf{e}$ (a zero bit) to be moved below it (flipping it to a one bit). The probability $r(h)$ that $\mathbf{e}$ will *r*ise when another element is passed to `step` is equal to the number of rungs of $\mathbf{e}$ that are zero bits, $\mathbf{H}-h$, divided by the total number of zero bits in the entire array, $\mathbf{N}/2$, from which the bit to be set is selected.

$$r(h)=\frac{\mathbf{H}-h}{\mathbf{N}/2}=\frac{2\mathbf{H}-2h}{\mathbf{N}}$$

Similarly, $\mathbf{e}$ will drop in height when another element takes a `step` and the bit cleared to zero is one of $\mathbf{e}$'s rungs. The probability $d(h)$ that $\mathbf{e}$ will *d*rop when another element is passed to `step` is equal to the number of rungs of $\mathbf{e}$ that are one bits, $\mathbf{H}$, divided by the total number of one bits in the entire array, $\mathbf{N}/2$, from which the bit to be cleared is selected.

$$d(h)=\frac{h}{\mathbf{N}/2}=\frac{2h}{\mathbf{N}}$$

We calculate detection and false-positive rates by iteratively calculating the probability that an element is at a particular height after each step taken for *any* element in the stream. In each iteration i, we track the probability that an element has taken s steps and is currently at height h. Whereas the iteration counter i is incremented on on the arrival (step taken) for any element in the stream, only steps for the element we are interested in increment s. We define $P[i,s,h]$ to be the probability that an element of interest is at height h after taking exactly s steps (for the element of interest) by iteration i (the i_{th} call to `step` for the i_{th} element in the stream).

In the base state before the first iteration ($i=0$) no steps have been taken, and so we initialize to zero the probabilities of all states that represent more than zero steps (when $s>0$). The probability that an element for which no steps have been taken is at a height h is the probability that h of its rungs were chosen from the set of $\mathbf{N}/2$ bits initially set to one and the other $\mathbf{H}-h$ rungs from those initially set to zero.

$$P[0,s,h] \quad = \quad \begin{cases} 0, & \text{if } s>0 \\ \frac{\binom{\mathbf{N}/2}{h}\binom{\mathbf{N}/2}{\mathbf{H}-h}}{\binom{\mathbf{N}}{\mathbf{H}}}, & \text{if } s=0 \end{cases}$$

The above equation accounts for the fact that the probability is that a rung is initially assigned be a zero or one bit is *not* independent, but slightly dependent, on the other rungs; if the first rung of an element is a zero bit, the ratio of remaining bits is **N**/2-1 one bits to **N**/2 zero bits, and so the probability that the next bit will also be one is slightly less than one half.

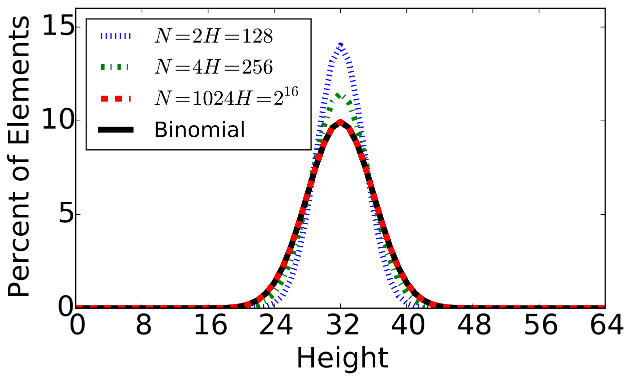


Fig. 5: The initial height of elements that have not been observed approaches the binomial distribution as $\frac{\mathbf{N}}{\mathbf{H}}\rightarrow\infty$. This figure shows that, for when $\mathbf{N}$ is a small multiple of $\mathbf{H}$, elements that have not been observed will skew more closely to the middle of their ladders than the binomial distribution. The lines for $\mathbf{N}=2^{16}$ and the binomial distribution, listed separately in the legend, are on top of each other.

As such, elements skew slightly more toward the center of the ladder than they would if rungs were truly independent.

For most configurations, $\mathbf{N}\gg\mathbf{H}$, and so we can approximate the probability that each rung is one or zero as exactly one half—as if they were independent. Using this approximation, the probability of being at each height is given by the binomial distribution: the chance that h of the $\mathbf{H}$ rungs (bits) associated with the element were set to 1 by chance with each rung having a 50% chance of being either value. Figure 5 illustrates the skew from the binomial distribution when $\mathbf{N}$ is not much larger than $\mathbf{H}$, and how the two distributions are indistinguishable when $\mathbf{N}\gg\mathbf{H}$.

$$P[0,0,h]\approx\binom{\mathbf{H}}{h}\cdot\frac{1}{2^{\mathbf{H}}}$$

(As an aside, if when associating elements with $\mathbf{H}$ indexes into the bit array, we had allowed these elements to overlap such that one bit could represent two or more rungs, we would have arrived at a binomial distribution for even small values of $\mathbf{N}$. However, elements might rise or fall by more than one rung in a single step.)

Given the initial state of a binomial ladder filter, we walk through the inductive process of calculating the probability of being at each state – having taken s steps and being at a height h – at each subsequent iteration i. In stepping through this calculation, we will often use the probabilities events did not happen, and so define the complement probabilities $\overline{f}=1-f$ for a step being taken for an element other than the element of interest, $\overline{r}(h)=1-r(h)$ that a step taken by another element will not cause the element of interest to rise, and $\overline{d}(h)=1-d(h)$ that a step taken by another element will not cause the element of interest to drop.

The probabilities at iteration i are derived from those at iteration $(i-1)$:

$$P[i,s,h] = \begin{array}{l} \overline{f} \cdot P[i-1,s,h] \cdot r(h) \cdot d(h) \\ +\overline{f} \cdot P[i-1,s,h] \cdot \overline{r}(h) \cdot \overline{d}(h) \\ +\overline{f} \cdot P[i-1,s,h-1] \cdot r(h-1) \cdot \overline{d}(h-1) \\ +\overline{f} \cdot P[i-1,s,h+1] \cdot \overline{r}(h+1) \cdot d(h+1) \\ +f \cdot P[i-1,s-1,h-1] \end{array}$$

The first four lines are the probability of reaching this state due to a step taken for another element, which occurs with probability of $1-f$. Hence, each of these four terms begins with $\overline{f}$.

The first two of these lines are the probability of arriving at this state from a prior iteration at the same height. In other words, the probability that the height did not change. The first line represents the likely case that a step for another element neither caused this element to rise or fall. The second line represents the very rare possibility that a step for another element caused both a rise (a rung being set to one) and a fall (another rung being set to zero), leaving the height unchanged.

The third and fourth lines of the equation represent changes in height due to a step for another element. The third line represents a rise from an element at height $h-1$, and so the probability is the product of rising from that height, $r(h-1)$, not dropping from that height, $\overline{d}(h-1)$, and the probability of being at that height given the same number of steps have been taken: $P[i-1,s,h-1]$. This line is replaced with 0 when calculating height 0 as there is no position on the ladder at height -1 to rise up from. The fourth line represents a drop from height $h+1$ and is the product of not rising, and falling, and being at height $h+1$. The fourth line is replaced with 0 when calculating height $\mathbf{H}$ as there is no position on the ladder at height $\mathbf{H}+1$ to drop down from.

The fifth and final line represents the probability of reaching this state due to a step for this element, which occurs with probability f and is multiplied by the probability of having been at the state one step below (and thus at a height one below).

Perpetual Mode

When calculating the iterative probability of an element being at the top of its ladder in perpetual mode, we must also add a sixth line – shown added below – since a step from a previous iteration that is at the top of the ladder will keep the element at the top of its ladder.

$$P[i,s,\mathbf{H}] = \cdots + f \cdot P[i-1,s-1,\mathbf{H}]$$

Up to this point in the analysis we have used the frequency f to represent the probability that an element will occur at any point in the stream. Our stochastic simulation yields a broad distribution of actual frequencies, both higher and lower, that is only centered at the expected frequency f. Yet the goal of our filter is to detect and reject elements based on the rate at which they occur, not at the rate they were expected to occur. Hence, to calculate detection and rejection rates when an element takes its i_{th} step, we examine the portion of the probability distribution where the number of steps taken is equal to $s-1$.

We calculate the sum of the probabilities of being at step $s-1$ after $sf^{-1}-1$ iterations, such that a step at the next iteration would result in an observed frequency of exactly f. We calculate the detection rate as the proportional probability of being at or above the detection height threshold $\mathbf{T}$:

$$P\,(\text{detect}) = \frac{\sum_{h=\mathbf{T}}^{\mathbf{H}} P[sf^{-1}-1,s-1,h]}{\sum_{h=0}^{\mathbf{H}} P[sf^{-1}-1,s-1,h]}.$$

As time goes on (and the number of steps increases) there is a (very slowly) increasing chance that the overall frequency of an element is low, but the recent frequency is high, resulting in a false positive.

Sticky Mode

When calculating the iterative probability of an element being at the top of its ladder in sticky mode, we do not add the sixth line for height $\mathbf{H}$ but instead a new height to the top of the ladder to represent the stuck-at state. The only way to reach this state is to have been in the state during a previous iteration, or to take step for an element that is at the top of the ladder ($\mathbf{H}$) but not yet at the stuck-at state above the top.

$$P[i,s,\mathbf{H}+1] = \begin{array}{r} P[i-1,s,\mathbf{H}+1] \\ +f \cdot P[i-1,s-1,\mathbf{H}] \end{array}$$

To calculate the detection rate for sticky mode, we sum the probabilities of being at the top of the ladder or already being in the stuck-at state above it.

$$P\,(\text{detect}) = \frac{\sum_{h=H}^{\mathbf{H}+1} P[sf^{-1}-1,s-1,h]}{\sum_{h=0}^{\mathbf{H}+1} P[sf^{-1}-1,s-1,h]}$$

V. Privacy and Rare Elements

The binomial ladder filter's ability to protect the identity of rare elements is what differentiates it from other data structures. It is designed to limit the information revealed to an attacker who captures the state of the binomial ladder filter and wants to determine if an element $\mathbf{e}$ that occurs with frequency below the rejection threshold has, nonetheless, appeared in the stream.

The null hypothesis is that the element $\mathbf{e}$ has not appeared in the stream and that all of the one bits associated with $\mathbf{e}$ were set at random either during initialization or due to `steps` taken for other elements. By measuring the height of $\mathbf{e}$'s binomial ladder, an attacker can calculate a likelihood ratio LR: the ratio of the probability that the value has been observed with some tiny frequency or in some tiny number of events to the probability that this height of the ladder was reached by chance.

The binomial ladder achieves privacy because it shares bits between elements. A one stored in a bit associated with an element $\mathbf{e}$ due to a `step` taken for $\mathbf{e}$ is indistinguishable from a one set during initialization of the filter or a `step` taken for another element. More generally, the filter does not reveal when x bits were set at random and ϵ bits were set by `step` operations and when $x + \epsilon$ bits were set at random. The only difference between these two scenarios is the likelihood with which they occur under the null hypothesis.

Consider, for example, that we are concerned about attackers testing to see whether elements observed five times or fewer are stored in the binomial ladder filter in our running example (ladder height $\mathbf{H} = 48$, $\mathbf{N} = 2^{29}$). If the starting height of an element is $\frac{\mathbf{H}}{2} = 24$ rungs, it would be at 29 rungs after five observations. The probability of being at or above a given height by chance (calculated from the binomial CDF) decreases from 0.557 at height 24 to 0.096 at height 29: a proportional increase in the likelihood ratio for rejecting the null hypothesis of 5.76. Stated another way, one of every 5.76 of the candidate elements that were previously as attractive to attackers as this element are still more attractive to attackers after these five steps were taken.

More generally, the multiplicative increase to the likelihood ratio LH due to ϵ steps for an element of interest $\mathbf{e}$ starting at height h is

$$\Delta LH = \frac{\sum_{i=h}^{\mathbf{H}} \binom{\mathbf{H}}{i}}{\sum_{i=h+\epsilon}^{\mathbf{H}} \binom{\mathbf{H}}{i}}.$$

Not all elements will start at height $\frac{\mathbf{H}}{2}$. An element observed only a few times may have the misfortune of starting high up on its ladder by pure chance. This will result in a larger increase in the likelihood ratio as each actual observation of the element occurs. For our example with $\mathbf{H} = 48$, there is just under a two in a million chance that an element will start at height 40 (i.e. $P[0, 0, 40] \approx 2 \cdot 10^{-6}$). The first actual observation of such a value will increase the likelihood ratio for rejecting the null hypothesis (i.e. that we reached height 41 by chance) by a factor of 5.3. Still, if we looked only at the roughly two in a million values that have *at least* 40 elements set to one by chance, one out of every $25,181$ of them will have 45 bits set by chance, as if they too had been observed another five times.

Increasing the ladder height $\mathbf{H}$ can reduce the amount of statistical information revealed by each step, but at the cost of increasing computation, space requirements, and the number of observations required to detect keys.

VI. Configuration

We derive initial configurations for the binomial ladder filter by setting as our goal that elements with frequencies at, or even close to, the detection frequency should rise to the top of their ladders. Similarly, less frequent elements at the rejection threshold hover closer to the midpoint of their ladders (the expected height for elements that do not occur and for which $f = 0$).

We define the equilibrium height of a value with a given frequency, $\tilde{h}$, to be the height at which the probability of dropping a rung down its ladder at each iteration is equal to the probability of moving up its ladder (or, in the event that an element is at the top of its ladder, the probability that it would move up if only there were more rungs to climb).

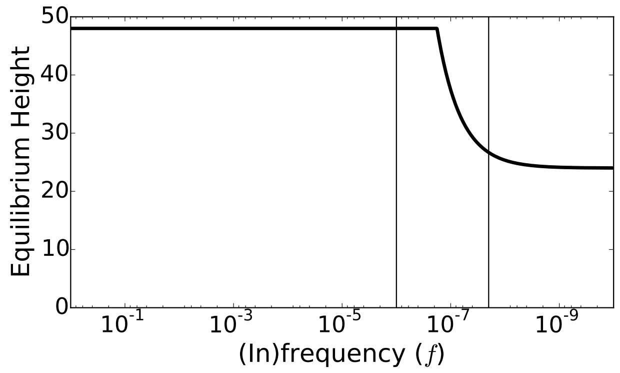


Fig. 6: The equilibrium height function, $\tilde{h}(f)$, for an element that occurs with frequency f on a binomial ladder filter with $\mathbf{N} = 2^{29}$ bits and ladder height $\mathbf{H} = 48$. We illustrate an indifference region bounded by a detection threshold of one in a million (the vertical line to the left) and a rejection threshold of one in fifty million (the vertical line to the right).

For an element of interest that occurs with frequency f, the probability of climbing is equal to the probability of a step taken for the element of interest (f) plus the probability that another element is observed $(\overline{f})$ that causes the element of interest to rise.

$$f + \overline{f} \cdot r(\tilde{h}) \cdot \overline{d}(\tilde{h})$$

Falling occurs with a probability that a step is taken for another element $(\overline{f})$ which causes the element of interest to drop.

$$\overline{f} \cdot \overline{r}(h) \cdot d(h)$$

Below the equilibrium height, the upward forces exceed downward forces, causing the element to (eventually) rise. Above the equilibrium height the downward pressures exceed the upward forces, causing the element to (eventually) fall. Thus, homeostatic forces are constantly pushing elements toward their equilibrium height. The equilibrium height can be derived by solving for height at which the probability of climbing equal the probability of falling.

$$f + \overline{f} r(\tilde{h}) \overline{d}(\tilde{h}) = \overline{f} \overline{r}(\tilde{h}) d(\tilde{h})$$

We begin by simplifying the above equation to isolate the input, f, on the right side.

$$\overline{r}(\tilde{h}) \cdot d(\tilde{h}) - r(\tilde{h}) \cdot \overline{d}(\tilde{h}) = \frac{f}{1 - f}$$

After multiplying and cancelling terms, we expand $d(\tilde{h})$ and $r(\tilde{h})$.

$$\begin{aligned} d(\tilde{h}) - r(\tilde{h}) &= \frac{f}{1-f} \\ \frac{2\tilde{h}}{\mathbf{N}} - \frac{2\mathbf{H} - 2\tilde{h}}{\mathbf{N}} &= \frac{f}{1-f} \\ \tilde{h} &= \frac{1}{2}\mathbf{H} + \frac{f}{1-f}\frac{\mathbf{N}}{4} \end{aligned}$$

Rewriting the equilibrium height as a function of f, and recognizing that the `step` operation cannot cause an element to climb above the top of its ladder, we obtain the equilibrium height equation:

$$\tilde{h}(f) = \min\left(\frac{1}{2}\mathbf{H} + \frac{f}{1-f}\frac{\mathbf{N}}{4},\ \ \mathbf{H}\right). \qquad (1)$$

We graph the equilibrium height as a function of frequency for a sample binomial ladder filter in Figure 6. Elements that never occur, with $f = 0$, have an equilibrium height half way up their ladders at $\tilde{h}(0) = \mathbf{H}/2$, as half of the bits associated with the element are expected to be one by chance. The equilibrium height grows sharply within the region of indifference, thereby making the height a powerful signal with which to identify elements with frequency above the detection threshold to distinguish them from elements below the rejection threshold.

A simple approach to configuring a binomial ladder filter is to aim to have the equilibrium height reach the top of the ladder half way between the rejection frequency f_r and the detection frequency f_d on a log scale. We call this this midpoint frequency, f_m.

$$f_m = e^{\frac{\ln(f_r)+\ln(f_d)}{2}}$$

Seting the equilibrium height in Equation 1 to $\mathbf{H}$ and the frequency to f_m yields:

$$\mathbf{H} = \frac{1}{2}\mathbf{H} + \frac{f_m}{1-f_m}\frac{\mathbf{N}}{4}.$$

We can then solve for the size of the filter in bits, $\mathbf{N}$, as a function of the ladder height $\mathbf{H}$.

$$\mathbf{N} = 2 \cdot \mathbf{H} \cdot \frac{1-f_m}{f_m}.$$

To determine the best ladder height, $\mathbf{H}$, simply increase values until the rate of false detections and rejections fall into an acceptable range. For example, Figure 7 shows the reduction in error rates that result from moving from a ladder of height $\mathbf{H} = 48$ with threshold $\mathbf{T} = 44$ (solid lines) to a ladder of height $\mathbf{H} = 64$ with threshold $\mathbf{T} = 56$. The reduced long-term error rates come at a cost of a longer ramp-up time to detect frequent values. Increasing the ladder height also linearly increases the number of memory accesses and hash calculations required.

To avoid expensive division operations when indexing into the bit array, some applications may choose to configure $\mathbf{N}$ to

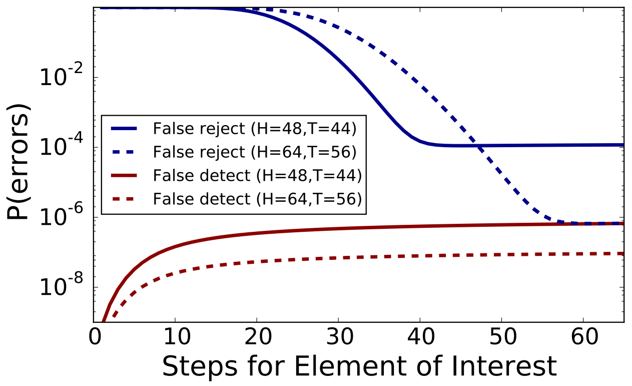


Fig. 7: We can reduce the probability of errors by increasing the height of the binomial ladder, as illustrated in perpetual mode moving from a ladder of height $\mathbf{H} = 48$ with threshold $\mathbf{T} = 44$ (solid lines) to a ladder of height $\mathbf{H} = 64$ with threshold $\mathbf{T} = 56$ (dashed lines). False rejections (in blue) are initially higher with a larger ladder to climb, but decrease as a taller ladder can support a greater distance between the detection threshold $\mathbf{T}$ and the top of the ladder. False detections (in red) are consistently lower. In this example, the $\mathbf{H} = 48, \mathbf{T} = 44$ ladder converges to have keep false detections and false rejections below one in a million.

the closest power of two. The setting of $\mathbf{H} = 48$, $\mathbf{N} = 2^{29}$ bits (64MB) used throughout the paper is the result of applying this process with a detection frequency of one in a million and a rejection frequency of one in fifty million, with a midpoint frequency of approximately one in seven million.

VII. Adjustments for Use in Distributed Systems

We can increase the throughput and reliability of the binomial ladder filter by implementing it as a high-performance distributed system.

An approximate data structure, which can only gives probabilistic guarantees of the number of observations required to detect a frequent value, should offer implementation advantages over data structures that give precise, deterministic detection guarantees. For example, an approximate data structure should be able to tolerate some amount of data loss and be able to operate correctly even when its underlying storage medium fails to provide atomic or consistent transactions. Yet the specification in Section II has a very precise requirement: the invariant that the number of zero bits and one bits be exactly equal. We will first show how to relax the invariant via small changes to the `step` operation. We will then describe how to distribute the data structure to minimize the amount of communication and coordination required.

A. *Probabilistic bit-ratio enforcement*

Enforcing the equal-ratio invariant for a filter array divided over multiple hosts would require that hosts coordinate to ensure that a bit is never set without also ensuring that a corresponding bit (which may have been distributed to another host) is set to one, requiring the execution of a distributed

Bit-Ratio	Average			Worst-Case		
Enforcement	Reads	Writes	Total	Reads	Writes	Total
Equal	2	2	4	$2 + \mathbf{N}/2$	2	$4 + \mathbf{N}/2$
Probabilistic	0	4†	4†	0	4	4

TABLE I: A comparison of the number of read and write operations to perform a `step` operation under the original enforcement of an equal bit-ratio from Section II and under the probabilistically-enforced bit-ratio in Section VII-A. We exclude the **H** bit reads needed to determine which rungs are below the element (the one bits) from those above (the zero bits)—it is the same for both approaches. The † in the probabilistic row indicates that we chose a conservative value: while four writes will be required for elements the top of their ladders, other elements require only three writes.

agreement protocol. Further, if data were lost, the system would need to count the exact number of ones in the surviving data in order to know how many bits should be set in the replacement data.

To make the binomial ladder filter more amenable to use in distributed systems, we relax the equal-ratio invariant to instead ensure that approximately half of all bits are one. We initialize the filter with random bits, setting each bit to one with probability one half. We then modify the `step` operation to use homeostasis to drive an equilibrium state in which half the bits are set to one.

Whereas the `step` operation previously identified a single random one bit to clear to zero, we instead pick two random bits to clear with no regard to their current value. Similarly, when taking a step for an element that is already at the top of its ladder, we choose two random bits of the array to set to one, whereas before we had searched to identify a single random zero bit. For an element that is not at the top of its ladder and has rungs above it, we preserve the `step` operation's existing behavior of setting the bit for one of those rungs to one.

At equilibrium, when half of all values are zero and half are one, writing a value to a randomly-selected bit position has probability .5 of changing the value of that bit. Hence, we expect to change one bit on average by assigning a value to two randomly-selected bits of unknown value. If we allow for data to be lost or for failures to occur between the operation of setting bits and clearing bits, the ratio of zeros and ones may become biased. The modifications made to `step` work to restore equilibrium. If the filter array contain more ones than zeros, the `step` operation will be biased to clear more bits than it sets until equilibrium is restored: the expected number of bits cleared will exceed one and the number of bits set when an element is at the top of its ladder will fall below one. If the filter array contains more zeros than ones, the `step` operation will be biased to set more bits than it clears until until equilibrium is restored.

Impact on performance: Adopting probabilistic bit-ratio enforcement should not hurt performance, even in non-distributed implementations of the filter.

As illustrated in Table I, the `step` operation with probabilistic ratio enforcement requires no more memory operations than the original. Both require **H** bit reads to identify which rungs are above and below the element, and so these reads are not shown in the figure. The revised `step` requires up to four writes: two bits are cleared and either one or two bits are set: one is set if the element is below the top of its ladder and two are set otherwise. While the original `step` required only two writes (one set and one clear), it required an expected two reads to identify a bit to clear. Since the revised `step` operation has a fixed number of memory operations, it can be relied upon to meet tighter latency bounds.

On systems where writing is much more expensive than reading, implementations of the revised `step` function will perform two extra reads to ensure that it writes only the values that change. On average, half of values have changed, and so the expected number of write operations is two, which is the same as for the original `step`.

B. Distributing the array

We divide the filter array into a fixed number of shards, M, of equal size ($\mathbf{N}/\mathrm{M}$ bits) which can be distributed to different servers. For example, a binomial ladder filter with 2^{29} bits (64MB) might be divided into 1024 shards of 2^{19} bits (64KB).

If we were to continue to map elements to bits in the entire array, the bits for an element might be stored in as many as **H** shards on **H** different servers. Both the `height` and `step` operations would only be able to return after the slowest of as many as **H** servers responded to the requests for the state of each rung. This naive approach to sharding would also be inefficient, as a separate communication request might be required for every one of the **H** rungs queried.

Instead, we will restrict each element's rungs to a single shard. We first hash the element to associate it with a single shard. We then use the **H** hash functions to associate the element with bits within the shard associated with the element. We can now compute the height of an element by identifying the shard associated with it, identifying the server responsible for storing that shard, and making a single request to that server (instead of as many as **H** requests). This design choice requires us to make the number of shards constant, so it is important to provision enough shards to allow them to be nearly evenly divided even if the system grows very large.

Co-locating an element's rungs within a single shard ensures that the `step` function will require at most five cross-server requests. The first request is issued by the client taking a `step` for an element. After identifying the shard associated with the element, it contacts the server that hosts that shard. The server that hosts the element's shard will examine the element's rungs locally, and can set a rung to one if the element is not at the top of its ladder. The server will then identify two bits in the entire array (not limited to its shard) to clear. The server will send (up to) two requests to the servers hosting the shards that contain the bits to be cleared. In the event that the element was already at the top of its ladder, and no rung was set to one, the server will also identify two bits in the entire array to set and send set requests to the responsible servers.

For all four of the potential requests to change bits on other servers, the server that hosts the element's shard does not need to know the values of the bits before they were set. Nor does that server need any guarantees about consistency if another sever is setting these values. Since the result of the `step` operation does not depend on the results of these set and clear operations, the server that host's the element's shard can issue these requests in the background and return a result to the client immediately. In the event that a small fraction of these background requests fail, biasing the ratio of ones to zeros, the homeostatic design works to restore equilibrium.

C. Handling loss

In the event that the server hosting a shard fails, the shard's data will be lost and the server that takes over responsibility for hosting that shard will need to replace the lost data. The simplest means of recovery is to replace the shard with a random array of ones and zeros, though this will cause all elements that map within the shard to reset to an average height of $\mathbf{H}/2$. To provide more comprehensive recovery, a shard's host may periodically write the shard to off-host back-up, such as a database or to the host next in line to be responsible for the shard. The period used should be equal to the acceptable amount of time (`step` events in the stream) that the application will tolerate the loss of. For example, if the application will tolerate a loss of only one second of `steps` for a lost shard, then all shards should be backed up every second. Backing up all of the non-replicated shards of a 64MB filter requires a write throughput of 64MB/second, or 0.5 Gbps, which is tiny for most highly-distributed systems.

D. Preventing hot spots

One way to attack a distributed binomial ladder filter would be to overload a server by directing a large fraction of the `step` operations to a single shard. To prevent attackers from identifying elements that would map to the same shard, the hash functions used to map elements to shards should be salted with a secret: some value not accessible to attackers. If a binomial ladder filter is to be used by clients controlled by parties that cannot be trusted to prevent hot spots (clients that may be controlled by an attacker), all operations should be mapped to a load-balanced set of trusted clients.

Attackers may still attempt to overload individual servers by issuing a large number of requests for a single element. To prevent element-specific hot-spots, clients should maintain a cache of recent results. The cache for `step` should include recent elements that have reached the top of their ladders. When `step` is called for one of these elements, the client can skip the height lookup and immediately issue two set and two clear operations to random bits not associated with the element.

If the ladder supports calls to `height` separate from the implicit `height` operation within `step`, the server may also want to cache results of the `height` operation. The server that hosts an element may indicate when client requesting a `height` should cache a value and should invalidate cache entries when heights exceed a detection threshold.

VIII. Applications to Passwords

The binomial ladder filter has two complimentary applications for protecting services that use password-based authentication from guessing attacks.

First, the filter can be applied to identify passwords frequently chosen by users and prevent these passwords from being used too frequently. In other words, it can enforce a policy that no password should be used by more than a certain number, or fraction, of a service's users. This policy addresses attacks that attempt to identify the most-commonly-used passwords on the system and to try them against a broad swath of user accounts.

Second, the filter can be applied to track the passwords submitted in authentication attempts that fail. The failed login attempts of attackers engaged in guessing common passwords are much more likely to contain a frequent-incorrect password than the failed login attempts of legitimate users, and so services can use such filteres to identify hosts engaged in these attacks with greater speed and accuracy.

The service may use this list of frequently-guessed passwords to accelerate its ability to detect hosts engaged in online-guessing attacks. The service may also decide to force users whose passwords are among those frequently-guessed by attackers to choose new passwords, potentially requiring an additional authentication factor to perform the password reset.

A. Frequently-chosen passwords

Preventing passwords from exceeding a maximum-frequency limits the probability that an attacker will succeed when guessing the most-frequently chosen passwords. If a service imposes a maximum-frequency limit $\hat{f}$, even an attacker who knows the set of passwords most-frequently employed by users must expect to issue an average of $1/\hat{f}$ guesses to compromise an account using one of these passwords.

A binomial ladder filter enables a service to identify frequently-chosen passwords while bounding the information it stores about rarely-chosen passwords—information attackers could learn if they managed to gain access to the filter.

The greater the number of accounts a service provider hosts, the easier it is to identify frequently-chosen passwords and forbid more than $\hat{f}$ users from using any given password. For example, a service with 100 million users can use a binomial ladder filter of height $\mathbf{H} = 20$, an array large enough to make clears exceedingly infrequent (e.g., $\mathbf{N} = 2^{33}$ bits or 1GB). Such a service can expect to prevent most passwords from becoming more frequent than one in 10 million and prevent *all* passwords from becoming more frequent than one in 5 million. However, a site with only five million users will only observe one password before it becomes common with one in every five million users. The smaller the site, the more difficult the trade-off between the frequency limit $\hat{f}$, false positives,

Application	**N**	**H**	**T**	steps per observation
Prevent any password from being used by more than ˜10 users in population of 100m	2^{33}	20	20	1
Prevent any password from being used by more than ˜3 users in population of 5m	2^{33}	16	16	3
For each IP, track the number of login failures using frequently-guessed passwords	2^{29}	48	44	1
Block all logins and force password reset if account's password is among frequent guesses	2^{29}	64	58	1

TABLE II: Suggested configurations for common applications.

and the amount of information revealed by using the `step` function to record each observed password.

In addition to increasing the frequency limit, $\hat{f}$, and decreasing the ladder height to speed detection, we can increase the number of steps taken each time a password is observed. This causes the filter to record more statistical evidence that each element occurred, reducing false positives (but weakening privacy due to the extra steps' change in the likelihood ratio).

A false positive, indicating a user's chosen password it too common when it is in fact unique, will force the user to choose another password unnecessarily. Even without false positives, data from past breaches suggest that well over one in a hundred users choose a common password on their first try. A false positive rate as high as one in ten thousand will not increase the number of users forced to choose another password by even a percent.

For example, consider a deployment scenario of a service with five million users that wants to prevent passwords from being used more than five users: $\hat{f} = 1/1,000,000$. Assume the service uses a binomial ladder filter height $\mathbf{H} = 16$, a sufficiently large filter array, and issues three steps each time a password is chosen. With this filter, most passwords would be forbidden after being employed by three users, a small number would be employed by four users, and a much smaller number by five users. While its possible that a tiny number of passwords might be used by as many as six users, an attacker would be unable to identify which of the passwords at the top of their ladders were used by twice as many users as others. A user who chose a unique password would be prevented from using it with probability 2^{-16}, and so we would expect 76 users in our population of 5 million users (0.0015%) would be forced to choose a new password after having chosen something unique on their first attempt.

Web-based services that already have a large user base with their passwords stored in a hashed format can take advantage of a frequently-chosen password filter by marking each account with a bit to indicate whether the account's password has been submitted to the filter. When a user successfully logs in with the correct password, and the flag indicates that the user's password has not yet been submitted to the filter, the service adds the user's password to the filter by calling `step` on the submitted password (which is available in plaintext during login). It then sets the bit indicating the the user's password has been sent to the filter.

B. Frequently-guessed passwords

Password-based services have a number of reasons to identify and track frequently-occurring *incorrect* passwords: passwords submitted in login attempts that either contain an invalid account name or that fail to match the assigned password for an existing account. The service may infer that a host (IP address) that issues a failed login attempt with a frequently-guessed password is more likely to be engaged in guessing than a host that fails to login using a rarely-guessed password. The service may prevent users from choosing passwords that are frequently guessed by attackers, or may force users to choose new passwords if their current password is one that attackers have guessed frequently.

The binomial ladder filter enables a service to identify frequently-occurring incorrect passwords while bounding the information it stores about the set of rarely-occurring ones. The set of rarely-occurring passwords that occur at least once may include legitimate users' actual passwords, which may be recorded when legitimate users enter incorrect usernames. It may also include typos of users' passwords.

An individual's password may appear frequent if she makes the same mistake repeatedly or if a client operating on her behalf repeatedly attempts to login with the wrong password. To prevent a single user's error to be counted repeatedly, the account identifier (user name or email) and password should be hashed together to create a salted hash that remains consistent even if the account identifier is invalid. A password should only be added to the filter if this salted hash of the password has not been seen before (or, at least, not recently).

A binomial ladder filter used to track frequently incorrect passwords might use an array of $\mathbf{N} = 2^{29}$ bits, ladder height of $\mathbf{H} = 48$, detection height threshold $\mathbf{T} = 44$, and take one `step` per observation. This will ensure that, with high probability, incorrect passwords that occur with frequency one in a million will be detected after 30 occurrences. Passwords less frequent than one in fifty million will rarely result in false positives.

C. Combining approaches

If a site prevents passwords from being used by more than one out of every million users ($\hat{f} = 1/1,000,000$), the best-case for an attacker is if all users choose from a set of one million passwords and so all passwords occur with probability $\hat{f} = 1/1,000,000$. This is equivalent to dividing up the 6-digit PIN space among all users. In such a case, an attacker has a 30 in one million chance of guessing an account that uses a password before the password is identified as frequently-guessed and users' who have chosen that password are locked out. Thus, an attacker could expect to compromise 30 out of every million accounts before all users are locked out. In reality, passwords distributions typically resemble a zipf distribution, and so only a small fraction of users who chose passwords predictable enough to be guessed online by attackers will be affected. With each round, a significant number of the users forced to choose a new password will have chosen one that the attacker will not guess, and so with each round the number of users forced to reset their password will be a small fraction of the number in the previous round.

IX. Related Work

The binomial ladder filter shares many of the same goals as much of the research in 'differential privacy', a framework introduced by Dwork [10]. Differential privacy is typically used to address the problem of reporting the frequency of events in data while bounding what is revealed by any individual event (or subset) in the data—so as not to allow data reported in aggregate to be used to make conclusions about individuals whose data were analyzed. For example, if examining patient records to count the fraction of one condition who have another condition, differential privacy could be used to add sufficient randomness such that the counters do not reveal information that would allow those with access to the data to draw conclusions about individuals. Within this framework Dwork *et al.* also introduced the concept of pan privacy to account for privacy in the event that the data structures being used to count events are compromised (once or repeatedly) while recording statistics from the stream of events [11]. Chan *et al.* [7] extend the concept for counters that report outcomes as each element in a stream arrives. Where work in differential privacy focuses releasing and storing the *frequency* of a set of pre-specified values in a stream, in our setting the potential values themselves (possible passwords) are, and should remain, unknown unless they prove sufficiently common.

Recent work by Blocki *et al.* [4] used differential privacy to introduce enough randomness to protect the privacy of users and release password-frequencies from a historical data set of 70 million Yahoo! users [6]. The researchers analyzed hashes generated from the password and a secret key that they did not have access to, to prevent attempts to reverse the hashes and find the original passwords if the data set were compromised. Thus, even though attackers surely knew many of the top 10-passwords (e.g., likely `123456` and `password`), the researchers can not identify or reveal them.

Another similar line of work focuses on optimizing the space required to detecting the most frequent elements, or 'heavy hitters', in a stream. Solutions include Cormode and Muthukrishnan's count-min sketch [8], a probabilistic data structure similar to the Bloom filter, and probabilistic algorithms based on more conventional array structures such as proposed by Metwally, Agrawal, and Abbadi [13]. More recent work includes that of Berinde *et al.*, who provide an excellent survey of solutions from this literature [3].

While these space-efficient data structures designed to detect frequent elements were not designed for privacy, security researchers have proposed employing space-efficient data structures to prevent weak passwords dating back as far as 1992, when Spafford introduced a weak-password prevention system that stored dictionaries of forbidden passwords using bloom filters [12].

In a 2009 study of security questions that revealed that many of these shared secrets had common answers, Schechter *et al.* proposed banning common ones [14]. As early as December of that year, it was observed that Twitter blocked a list of known-common passwords [1] and Microsoft followed in 2011 [9]. In 2010, Schechter, Herley, and Mitzenmacher proposed using Cormode and Muthukrishnan's count-min sketch to identify passwords as soon as they became common so they could be blocked immediately. They proposed introducing noise into the count-min sketch to introduce false positives and reduce the value of the sketch to attackers if it were compromised. However, they did not specify an algorithm for introducing such noise or provide proofs of what attackers could learn by analyzing the data structure itself. In contrast, simple analysis using the binomial distribution allows one to analyze the change in likelihood ratio that results when an element is recorded in a binomial ladder filter via the `step` operation.

X. Conclusion

We introduced the binomial ladder filter for detecting frequently-occurring elements in a stream while protecting the privacy of rarely-occurring elements. It is particularly valuable for applications that handle secrets because secrecy and frequency are inversely correlated: a good secret is known to at most a handful of individuals and will not be witnessed frequently, whereas a value that is shared by many or commonly guessed by others is not very secret. The binomial ladder filter allows applications to identify values that are too common to be anything more than nominally secret while minimizing the risk to rare and truly secret values. For example, it can be used to discover passwords commonly chosen by users or commonly guessed by attackers without putting users with unique or nearly-unique passwords at unnecessary risk.

References

[1] Olaf Alders. Twitter and avoiding weak passwords. http://blog.wundercounter.com/2009/12/twitter-and-avoiding-weak-passwords.html, December 8, 2009.

[2] Anonymized. Using guessed passwords to thwart online password guessing. In *Submission*, May 2016.

[3] Radu Berinde, Piotr Indyk, Graham Cormode, and Martin J. Strauss. Space-optimal heavy hitters with strong error bounds. *ACM Transactions on Database Systems*, 35:1–28, 2010.

[4] Jeremiah Blocki, Anupam Datta, and Jospeh Bonneau. Differentially private password frequency lists: Or, how to release statistics from 70 million passwords (on purpose). In *Proceedings of Network and Distributed Systems Security (NDSS)*, February 21–24 2016.

[5] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, July 1970.

[6] Jospeh Bonneau. The science of guessing: Analyzing an anonymized corpus of 70 million passwords. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, pages 538–552, May 2012.

[7] T.-H. Hubert Chan, Elaine Shi, and Dawn Song. Private and continual release of statistics. *ACM Trans. Inf. Syst. Secur.*, 14(3):26:1–26:24, November 2011.

[8] Graham Cormode and S Muthukrishnan. An improved data stream summary: the count-min sketch and its applications. *Journal of Algorithms*, 55(1):58–75, 2005.

[9] Dick Craddock. Inside windows live: Hey! my friends account was hacked! Blog post archived at http://bit.ly/1Z5MBTV, July 14, 2011.

[10] Cynthia Dwork. Differential privacy. In *Proceedings of the 33rd International Conference on Automata, Languages and Programming - Volume Part II*, ICALP'06, pages 1–12, Berlin, Heidelberg, 2006. Springer-Verlag.

[11] Cynthia Dwork, Moni Naor, Toniann Pitassi, and Guy N. Rothblum. Differential privacy under continual observation. In *Proceedings of the Forty-second ACM Symposium on Theory of Computing*, STOC '10, pages 715–724, New York, NY, USA, 2010. ACM.

[12] Eugene H. Spafford. OPUS: Preventing weak password choices. *Computers & Security*, 11(3):273–278, 1992.

[13] Ahmed Metwally, Divyakant Agrawal, and Amr El Abbadi. Efficient computation of frequent and top-k elements in data streams. In *Proceedings of the International Conference on Database Theory (ICDT)*, pages 398–412. Springer LNCS Volume 3363, 2005.

[14] Stuart Schechter, A. J. Bernheim Brush, and Serge Egelman. It's no secret: Measuring the security and reliability of authentication via 'secret' questions. In *Proceedings of the 2009 IEEE Symposium on Security and Privacy*, Berkeley, CA, USA, May 2009. IEEE Computer Society.

[15] Stuart Schechter, Cormac Herley, and Michael Mitzenmacher. Popularity is everything: A new approach to protecting passwords from statistical-guessing attacks. In *The 5th USENIX Workshop on Hot Topics in Security (HotSec '10)*. USENIX, August 2010.