Certifying Certainty and Uncertainty in Approximate Membership Query Structures

Kiran Gopinathan[1] and Ilya Sergey[2,1]

[1] School of Computing, National University of Singapore, Singapore
[2] Yale-NUS College, Singapore
{kirang,ilya}@comp.nus.edu.sg

Abstract. Approximate Membership Query structures (AMQs) rely on randomisation for time- and space-efficiency, while introducing a possibility of false positive and false negative answers. Correctness proofs of such structures involve subtle reasoning about bounds on probabilities of getting certain outcomes. Because of these subtleties, a number of unsound arguments in such proofs have been made over the years.

In this work, we address the challenge of building rigorous and reusable computer-assisted proofs about probabilistic specifications of AMQs. We describe the framework for systematic decomposition of AMQs and their properties into a series of interfaces and reusable components. We implement our framework as a library in the Coq proof assistant and showcase it by encoding in it a number of non-trivial AMQs, such as Bloom filters, counting filters, quotient filters and blocked constructions, and mechanising the proofs of their probabilistic specifications.

We demonstrate how AMQs encoded in our framework guarantee the absence of false negatives *by construction*. We also show how the proofs about probabilities of false positives for complex AMQs can be obtained by means of *verified reduction* to the implementations of their simpler counterparts. Finally, we provide a library of domain-specific theorems and tactics that allow a high degree of automation in probabilistic proofs.

1 Introduction

Approximate Membership Query structures (AMQs) are probabilistic data structures that compactly implement (multi-)sets via hashing. They are a popular alternative to traditional collections in algorithms whose utility is not affected by some fraction of wrong answers to membership queries. Typical examples of such data structures are Bloom filters [6], quotient filters [5,37], and count-min sketches [12]. In particular, versions of Bloom filters find many applications in security and privacy [16,18,35], static program analysis [36], databases [17], web search [22], suggestion systems [44], and blockchain protocols [19,42].

Hashing-based AMQs achieve efficiency by means of losing precision when answering queries about membership of certain elements. Luckily, most of the applications listed above can tolerate *some* loss of precision. For instance, a static points-to analysis may consider two memory locations as aliases even if they are not (a *false positive*), still remaining sound. However, it would be unsound for such an analysis to claim that two locations do not alias in the case they do (a *false negative*). Even if it increases the number of false positives, a randomised data structure can be used to answer aliasing queries in a sound way—as long as it does not have false negatives [36]. But *how much* precision would be lost if, *e.g.*, a Bloom filter with certain parameters is chosen to answer these queries? Another example, in which quantitative properties of false positives are critical, is the security of Bitcoin's Nakamoto consensus [34] that depends on the counts of block production per unit time [19].

In the light of the described above applications, of particular interest are two kinds of properties specifying the behaviour of AMQs:

– *No-False-Negatives* properties, stating that a set-membership query for an element x always returns `true` if x is, in fact, in the set represented by the AMQ.
– Properties quantifying the rate of *False Positives* by providing a probabilistic bound on getting a wrong "yes"-answer to a membership query, given certain parameters of the data structure and the past history of its usage.

Given the importance of such claims for practical applications, it is desirable to have machine-checked formal proofs of their validity. And, since many of the existing AMQs share a common design structure, one may expect that a large portion of those validity proofs can be reused across different implementations.

Computer-assisted reasoning about the absence of *false negatives* in a particular AMQ (Bloom filter) has been addressed to some extent in the past [7]. However, to the best of our knowledge, mechanised proofs of probabilistic bounds on the *rates of false positives* did not extend to such structures. Furthermore, to the best of our knowledge, no other existing AMQs have been formally verified to date, and no attempts were made towards characterising the commonalities in their implementations in order to allow efficient proof reuse.

In this work, we aim to advance the state of the art in machine-checked proofs of probabilistic theorems about false positives in randomised hash-based data structures. As recent history demonstrates, when done in a "paper-and-pencil" way, such proofs may contain subtle mistakes [8, 10] due to misinterpreted assumptions about relations between certain kinds of events. These mistakes are not surprising, as the proofs often need to perform a number complicated manipulations with expressions that capture probabilities of certain events. Our goal is to factor out these reasoning patterns into a standalone library of *reusable* program- and specification-level definitions and theorems, implemented in a proof assistant enabling computer-aided verification of a variety of AMQs.

Our contributions. The key novel observation we make in this work is the decomposition of the common AMQ implementations into the following components: (a) a hashing strategy and (b) a state component that operates over hash outcomes, together capturing most AMQs that provide fixed constant-time insertion and query operations. Any AMQ that is implemented as an instance of those components enjoys the *no-false-negatives* property *by construction.* Furthermore, such a decomposition streamlines the proofs of structure-specific bounds on false positive rates, while allowing for proof reuse for complex AMQ implementations, which are built on top of simpler AMQs [39]. Powered by those insights, this work makes the following technical contributions:

– A Coq-based mechanised framework Ceramist, specialised for reasoning about AMQs.[3] Implemented as a Coq library, it provides a systematic decomposition of AMQs and their properties in terms of Coq modules and uses these interfaces to to derive certain properties "for free", as well as supporting proof-by-reduction arguments between classes of similar AMQs.
– A library of non-trivial theorems for expressing closed-form probabilities on false positive rates in AMQs. In particular, we provide the first mechanised proof of the closed form for Stirling numbers of the second kind [25, Chapter 6].
– A collection of proven facts and tactics for effective construction of proofs of probabilistic properties. Our approach adopts the style of Ssreflect reasoning [21,30], and expresses its core lemmas in terms of rewrites and evaluation.
– A number of case study AMQs mechanised via Ceramist: ordinary [6] and counting [45] Bloom filters, quotient filters [5,37], and Blocked AMQs [39].

For ordinary Bloom filters, we provide the first mechanised proof that the probability of a false positive in a Bloom filter can be written as a closed form expression in terms of the input parameters; a bound that has often been mischaracterised in the past due to oversight of subtle dependencies between the components of the structure [6,33]. For Counting Bloom filters, we provide the first mechanised proofs of several of their properties: that they have no false negatives, its false positive rate, that an element can be removed without affecting queries for other elements, and the fact that Counting Bloom filters preserve the number of inserted elements irrespective of the randomness of the hash outputs. For quotient filters, we provide a mechanised proof of the false positive rate and of the absence of false negatives. Finally, alongside the standard Blocked Bloom filter [39], we derive two novel AMQ data structures: *Counting Blocked Bloom filters* and *Blocked Quotient filters*, and prove corresponding no-false-negatives and false positive rates for all of them. Our case studies illustrate that Ceramist can be repurposed to verify hash-based AMQ structures, including entirely new ones that have not been described in the literature, but rather have been obtained by composing existing AMQs via the "blocked" construction.

Our mechanised development [24] is entirely *axiom-free*, and is compatible with Coq 8.11.0 [11] and MathComp 1.10 [30]. It relies on the infotheo library [2] for encoding discrete probabilities.

Paper outline. We start by providing the intuition on Bloom filters, our main motivating example, in Sec. 2. We proceed by explaining the encoding of their semantics, auxiliary hash-based structures, and key properties in Coq in Sec. 3. Sec. 4 generalises that encoding to a general AMQ interface, and provides an overview of Ceramist, its embedding into Coq, showcasing it by another example instance—Counting Bloom filters. Sec. 5 describes the specific techniques that help to structure our mechanised proofs. In Sec. 6, we report on the evaluation of Ceramist on various case studies, explaining in detail our compositional treatment of blocked AMQs and their properties. Sec. 7 provides a discussion on the state of the art in reasoning about probabilistic data structures.

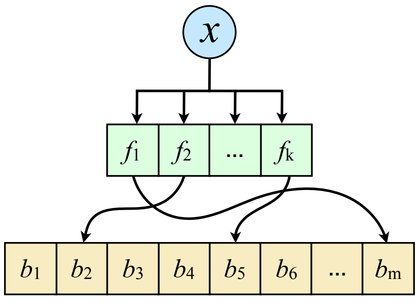
[3] Ceramist stands for **Cer**tified **A**pproximate **M**embersh**i**p **St**ructures.

2 Motivating Example

Ceramist is a library specialised for reasoning about AMQ data structures in which the underlying randomness arises from the interaction of one or more hashing operations. To motivate this development, we thus consider applying it to the classical example of such an algorithm—a Bloom filter [6].

2.1 The Basics of Bloom Filters

Bloom filters are probabilistic data structures that provide compact encodings of mathematical sets, trading increased space efficiency for a weaker membership test [6]. Specifically, when testing membership for a value *not* in the Bloom filter, there is a possibility that the query may be answered as positive. Thus a property of direct practical importance is the exact probability of this event, and how it is influenced by the other parameters of the implementation.



A Bloom filter *bf* is implemented as a binary vector of m bits (all initially zeros), paired with a sequence of k hash functions $f_1, \ldots, f_k$, collectively mapping each input value to a vector of k indices from $\{1 \ldots m\}$, the indices determine the bits set to true in the m-bit array Assuming an ideal selection of hash functions, we can treat the output of $f_1, \ldots, f_k$ on new values as a uniformly-drawn random vector. To insert a value x into the Bloom filter, we can treat each element of the "hash vector" produced from $f_1, \ldots, f_k$ as an index into *bf* and set the corresponding bits to ones. Similarly, to test membership for an element x, we can check that all k bits specified by the hash-vector are raised.

2.2 Properties of Bloom Filters

Given this model, there are two obvious properties of practical importance: that of false positives and of false negatives.

False Negatives. It turns out that these definitions are sufficient to guarantee the lack of false-negatives with complete certainty, *i.e.*, irrespective of the random outcome of the hash functions. This follows from the fact that once a bit is raised, there are no permitted operations that will unset it.

Theorem 1 (No False Negatives). *If* $x \in bf$*, then* $\Pr[x \in_? bf] = 1$*, where* $x \in_? bf$ *stands for the approximate membership test, while the relation* $x \in bf$ *means that* x *has been previously inserted into* bf*.*

False Positives. This property is more complex as the occurrence of a false positive is entirely dependent on the particular outcomes of the hash functions $f_1, \ldots, f_k$ and one needs to consider situations in which the hash functions happen to map some values to *overlapping* sets of indices. That is, after inserting a series of values xs, subsequent queries for $y \notin xs$ might incorrectly return true.

This leads to subtle dependencies that can invalidate the analysis, and have lead to a number of incorrect probabilistic bounds on the event, including in the analysis by Bloom in his original paper [6]. Specifically, Bloom first considered the probability that inserting l distinct items into the Bloom filter will set a particular bit b_i. From the independence of the hash functions, he was able to show that the probability of this event has a simple closed-form representation:

Lemma 1 (Probability of a single bit being set). *If the only values previously inserted into bf are $x_1, \ldots, x_l$, then the probability of a particular single bit at the position i being set is* $\Pr\left[i^{\text{th}} \text{ bit in } bf \text{ is set}\right] = 1 - \left(1 - \frac{1}{m}\right)^{kl}$.

Bloom then claimed that the probability of a false positive was simply the probability of a single bit being set, raised to the power of k, reasoning that a false positive for an element $y \notin bf$ only occurs when all the k bits corresponding to the hash outputs are set.

Unfortunately, as was later pointed out by Bose *et al.* [8], as the bits specified by $f_1(x), \ldots, f_{k-1}(x)$ may overlap, we cannot guarantee the independence that is required for any simple relation between the probabilities. Bose *et al.* rectified the analysis by instead interpreting the bits within a Bloom filter as maintaining a set $\text{bits}(bf) \subseteq \mathbb{N}_{[0,\ldots,m-1]}$, corresponding to the indices of raised bits. With this interpretation, an element y only tests positive if the random set of indices produced by the hash functions on y is such that $\text{inds}(y) \subseteq \text{bits}(bf)$. Therefore, the chance of a positive result for $y \notin bf$ resolves to the chance that the random set of indices from hashing y is a subset of the union of $\text{inds}(x)$ for each $x \in bf$. The probability of this reduced event is described by the following theorem:

Theorem 2 (Probability of False Positives). *If the only values inserted into bf are $x_1, \ldots, x_l$, then for any $y \notin bf$,* $\Pr\left[y \in_? bf\right] = \frac{1}{m^{k(l+1)}} \sum_{i=1}^{m} i^k i! \binom{m}{i} \left\{ {kl \atop i} \right\}$, *where* $\left\{ {s \atop t} \right\}$ *stands for the* Stirling number of the second kind, *capturing the number of surjections from a set of size s to a set of size t.*

The key step in capturing these program properties is in treating the outcomes of hashes as *random variables* and then propagating this randomness to the results of the other operations. A formal treatment of program outcomes requires a suitable semantics, representing programs as distributions of such random variables. In moving to mechanised proofs, we must first fully characterise this semantics, formally defining a notion of a probabilistic computation in Coq.

3 Encoding AMQs in Coq

To introduce our encoding of AMQs and their probabilistic behaviours in Coq, we continue with our running example, transitioning from mathematical notation to Gallina, Coq's language. The rest of this section will introduce each of the key components of this encoding through the lens of Bloom filters.

3.1 Probability Monad

Our formalisation represents probabilistic computations using an embedding following the style of the FCF library [38]. We do not use FCF directly, due to its primary focus on cryptographic proofs, wherein it provides little support for proving probabilistic bounds directly, instead prioritising a reduction-based approach of expressing arbitrary computations as compositions of known distributions.

Following the adopted FCF notation, a term of type `Comp` A represents a probabilistic computation returning a value of type A, and is constructed using the standard monadic operators, with an additional primitive `rand` n that allows sampling from a uniform distribution over the range $\mathbb{Z}_n$:

$$\begin{aligned}
\texttt{ret} &: A \to \texttt{Comp}\ A \\
\texttt{bind} &: \texttt{Comp}\ A \to (A \to \texttt{Comp}\ B) \to \texttt{Comp}\ B \\
\texttt{rand} &: (n : \mathbb{N}) \to \texttt{Comp}\ (\mathbb{Z}_n)
\end{aligned}$$

We implement a Haskell-style `do`-notation over this monad to allow descriptions of probabilistic computations within Gallina. For example, the following code is used to implement the query operation for the Bloom filter:

```
hash_res <-$ hash_vec_int x hashes; (* hash x using the hash functions *)
let (new_hashes, hash_vec) := hash_res in
(* check if all the corresponding bits are set *)
let qres := bf_query_int hash_vec bf in
(* return the query result and the new hashes *)
    ret (new_hashes, qres).
```

In the above listing, we pass the queried value `x` along with the hash functions `hashes` to a probabilistic hashing operation `hash_vec_int` to hash `x` over each function in `hashes`. The result of this random operation is then bound to `hash_res` and split into its constituent components—a sequence of hash outputs `hash_vec` and an updated copy `new_hashes` of the hash functions, now incorporating the mapping for `x`. Then, having mapped our input into a sequence of indices, we can query the Bloom filter for membership using a corresponding deterministic operation `bf_query_int` to check that all the bits specified by `hash_vec` are set. Finally, we complete the computation by returning the query outcome `qres` and the updated hash functions `new_hashes` using the `ret` operation to lift our result to a probabilistic outcome.

Using the code snippet above, we can define the query operation `bf_query` as a function that maps a Bloom filter, a value to query, and a collection of hash functions to a probabilistic computation returning the query result and an updated set of hash functions. However, because our computation type does not impose any particular semantics, this result only encodes the *syntax* of the probabilistic query and has no actual meaning without a separate interpretation.

Thus, given a Gallina term of type `Comp` A, we must first evaluate it into a distribution over possible results to state properties on the probabilities of its outcomes. We interpret our monadic encoding in terms of Ramsey's probability monad [41], which decomposes a complex distribution into composition of primitive ones bound together via conditional distributions. To capture this interpretation within Coq, we then use the encoding of this monad from the infotheo library [1,2], and provide a function `eval_dist` : `Comp` $A \to$ `dist` A that evaluates computations into distributions by recursively mapping them to the probability monad. Here, `dist A` represents infotheo's encoding of distributions over a finite support `A`, defined as being composed of a measure function `pmf` : $A \to \mathbb{R}^{+}$, and a proof that the sum of the measure over the support A produces 1.

This mapping from computations to distributions must be done to a program e (involving, *e.g.*, Bloom filter) before stating its probability bound. Therefore, we hide this evaluation process behind a notation that allows stating probabilistic properties in a form closer to their mathematical counterparts:

$$\Pr[e = v] \triangleq (\texttt{eval\_dist}\ e)\ v$$
$$\Pr[e] \triangleq (\texttt{eval\_dist}\ e)\ \mathsf{true}$$

Above, v is an arbitrary element in the support of the distribution induced by e. Finally, we introduce a binding operator $\rhd$ to allow concise representation of dependent distributions: $e \rhd f \triangleq \texttt{bind}\ e\ f$.

3.2 Representing Properties of Bloom Filters

We define the state of a Bloom filter (`BF`) in Coq as a binary vector of a fixed length m, using Ssreflect's `m.-tuple` data type:

```
Record BF := mkBF { bloomfilter_state: m.-tuple bool }.
Definition bf_new : BF := (* construct a BF with all bits cleared *).
Definition bf_get_int i : BF → bool := (* retrieve BF's ith bit *).
```

We define the deterministic components of the Bloom filter implementation as pure functions taking an instance of `BF` and a series of indices assumed to be obtained from earlier calls to the associated hash functions:

$$\texttt{bf\_add\_int} : \texttt{BF} \to \texttt{seq}\ \mathbb{Z}_m \to \texttt{BF}$$
$$\texttt{bf\_query\_int} : \texttt{BF} \to \texttt{seq}\ \mathbb{Z}_m \to \texttt{bool}$$

That is, `bf_add_int` takes the Bloom filter state and a sequence of indices to insert and returns a new state with the requested bits also set. Conversely, `bf_query_int` returns `true` *iff* all the queried indices are set. These pure operations are then called within a probabilistic wrapper that handles hashing the input and the book-keeping associated with hashing to provide the standard interface for AMQs:

$$\texttt{bf\_add} : B \to (\texttt{HashVec}\ B * \texttt{BF}) \to \texttt{Comp}\ (\texttt{HashVec}\ B * \texttt{BF})$$
$$\texttt{bf\_query} : B \to (\texttt{HashVec}\ B * \texttt{BF}) \to \texttt{Comp}\ (\texttt{HashVec}\ B * \texttt{bool})$$

The component `HashVec` B (to be defined in Sec. 3.3), parameterised over an input type B, keeps track of *known results* of the involved hash functions and is provided as an external parameter to the function rather than being a part of the data structure to reflect typical uses of AMQs, wherein the hash operation is pre-determined and shared by *all* instances.

With these definitions and notation, we can now state the main theorems of interest about Bloom filters directly within Coq:[4]

Theorem 3 (No False Negatives). *For any Bloom filter state bf, a vector of hash functions hs, after having inserted an element x into bf, followed by a series xs of other inserted elements, the result of query $x \in_? bf$ is always* `true`*. That is, in terms of probabilities:* $\Pr\left[\texttt{bf\_add}\ x\ (hs, bf) \rhd \texttt{bf\_addm}\ xs \rhd \texttt{bf\_query}\ x\right] = 1$.

Lemma 2 (Probability of Flipping a Single Bit). *For a vector of hash functions hs of length k, after inserting a series of l distinct values xs, all unseen in hs, into an empty Bloom filter bf, represented by a vector of m bits, the probability of its any index i being set is* $\Pr\left[\texttt{bf\_addm}\ xs\ (hs, \texttt{bf\_new}) \rhd \texttt{bf\_get}\ i\right] = 1 - \left(1 - \frac{1}{m}\right)^{kl}$*. Here,* `bf_get` *is a simple embedding of the pure function* `bf_get_int` *into a probabilistic computation.*

Theorem 4 (Probability of a False Positive). *After having inserted a series of l distinct values xs, all unseen in hs, into an empty Bloom filter bf, for any unseen $y \notin xs$, the probability of a subsequent query $y \in_? bf$ for y returning true is given as* $\Pr\left[\texttt{bf\_addm}\ xs\ (hs, \texttt{bf\_new}) \rhd \texttt{bf\_query}\ y\right] = \frac{1}{m^{k(l+1)}} \sum_{i=1}^{m} i^k i! \binom{m}{i} \left\{ {kl \atop i} \right\}$.

The proof of this theorem required us to provide *the first axiom-free mechanised proof* for the closed form for Stirling numbers of the second kind [25].

In the definitions above, we used the output of the hashing operation as the bound between the deterministic and probabilistic components of the Bloom filter. For instance, in our earlier description of the Bloom filter query operation in Sec. 3.1, we were able to implement the entire operation with the only probabilistic operation being the call `hash_vec_int x hashes`. In general, structuring AMQ operations as manipulations with hash outputs via *pure* deterministic functions allows us to decompose reasoning about the data structure into a series of specialised properties about its deterministic primitives and a separate set of reusable properties on its hash operations.

3.3 Reasoning about Hash Operations

We encode hash operations within our development using a random oracle-based implementation. In particular, in order to keep track of *seen* hashes learnt by hashing previously observed values, we represent a *state* of a hash function from elements of type `B` to a range $\mathbb{Z}_m$ using a finite map to ensure that previously hashed values produce the same hash output:

```
Definition HashState B := FixedMap B 'I_m.
```

[4] `bf_addm` is a trivial generalisation of the insertion to multiple elements.

The state is paired with a hash function generating uniformly random outputs for unseen values, and otherwise returns the value as from its prior invocations:

```
Definition hash value state : Comp (HashState B * B) :=
  match find value state with
  | Some(output) ⇒ ret (state, output)
  | None ⇒ rnd <-$ rand m;
           new_state <- put value rnd state;
           ret (new_state, rnd)
  end.
```

A *hash vector* is a generalisation of this structure to represent a vector of states of k independent hash functions:

```
Definition HashVec B := k.-tuple HashState B.
```

The corresponding hash operation over the hash vector, `hash_vec_int`, is then defined as a function taking a value and the current hash vector and then returning a pair of the updated hash vector and associated random vector, internally calling out to `hash` to compute individual hash outputs.

This random oracle-based implementation allows us to formulate several helper theorems for simplifying probabilistic computations using hashes by considering whether the hashed values *have been seen before or not*. For example, if we knew that a value x had not been seen before, we would know that the possibility of obtaining any particular choice of a vector of indices would be equivalent to obtaining the same vector by a draw from a corresponding uniform distribution. We can formalise this intuition in the form of the following theorem:

Theorem 5 (Uniform Hash Output). *For any two hash vectors hs, hs' of length k, a value x that has not been hashed before, and an output vector $\imath s$ of length m obtained by hashing x via hs, if the state of hs' has the same mappings as hs and also maps x to $\imath s$, the probability of obtaining the pair $(hs', \imath s)$ is uniform:* $\Pr\left[\texttt{hash\_vec\_int}\ x\ hs = (hs', \imath s)\right] = \left(\frac{1}{m}\right)^k$

Similarly, there are also often cases where we are hashing a value that we *have already seen*. In these cases, if we know the exact indices a value hashes to, we can prove a certainty on the value of the outcome:

Theorem 6 (Hash Consistency). *For any hash vector hs, a value x, if hs maps x to outputs $\imath s$, then hashing x again will certainly produce $\imath s$ and not change hs, that is,* $\Pr\left[\texttt{hash\_vec\_int}\ x\ hs = (hs, \imath s)\right] = 1$.

By combining these types of probabilistic properties about hashes with the earlier Bloom filter operations, we are able to prove the prior theorems about Bloom filters by reasoning primarily about the core logical interactions of the *deterministic components* of the data structure. This decomposition is not just applicable to the case of Bloom filters, but can be extended into a general framework for obtaining modular proofs of AMQs, as we will show in the next section.

4 Ceramist at Large

Zooming out from the previous discussion of Bloom filters, we now present Ceramist in its full generality, describing the high-level design in terms of the various interfaces it requires to instantiate to obtain verified AMQ implementations.

The core of our framework revolves around the decomposition of an AMQ data structure into separate interfaces for hashing (AMQHash) and state (AMQ), generalising the specific decomposition used for Bloom filters (hash vectors and bit vectors respectively). More specifically, the AMQHash interface captures the probabilistic properties of the hashing operation, while the AMQ interface captures the deterministic interactions of the state with the hash outcomes.

4.1 AMQHash Interface

The AMQHash interface generalises the behaviours of hash vectors (Sec. 3.3) to provide a generic description of the hashing operation used in AMQs.

The interface first abstracts over the specific types used in the prior hashing operations (such as, *e.g.*, `HashVec B`) by treating them as opaque parameters: using a parameter `AMQHashState` to represent the state of the hash operation; types `Key` and `Value` encoding the hash inputs and outputs respectively, and finally, a deterministic operation `AMQHash_add_internal : AMQHashState → Key → Value → AMQHashState` to encode the interaction of the state with the outputs and inputs. For example, in the case of a single hash, the state parameter `AMQHashState` would be `HashState B`, while for a hash vector this would instead be `HashVec B`.

To use this hash state in probabilistic computations, the interface assumes a separate probabilistic operation that will take the hash state and randomly generate an output (*e.g.*, `hash` for single hashes and `hash_vec_int` for hash vectors):

```
Parameter AMQHash_hash: Key → AMQHashState → Comp (AMQHash * Value).
```

Then, to abstractly capture the kinds of reasoning about the outcomes of hash operations done with Bloom filters in Sec. 3.3, the interface assumes a few predicates on the hash state to provide information about its contents:

```
Parameter AMQHash_hashstate_contains: AMQHashState → Key → Value → bool.
Parameter AMQHash_hashstate_unseen: AMQHashState → Key → bool.
```

These components are then combined together to produce more abstract formulations of the previous Theorems 5 and 6 on hash operations.

Property 1 (Generalised Uniform Hash Output) *There exists a probability p_{hash}, such that for any two AMQ hash states hs, hs', a value x that is unseen, and an output ιs obtained by hashing x via hs, if the state of hs' has the same mappings as hs and also maps x to ιs, the probability of obtaining the pair $(hs', \iota s)$ is given by:* $\Pr\left[\texttt{AMQHash\_hash}\ x\ hs = (hs', \iota s)\right] = p_{hash}$.

Property 2 (Generalised Hash Consistency) *For any AMQ hash state hs, a value x, if hs maps x to an output ιs, then hashing x again will certainly produce ιs and not change hs:* $\Pr\left[\texttt{AMQhash\_hash}\ x\ hs = (hs, \iota s)\right] = 1$

Proofs of these corresponding properties must also be provided to instantiate the AMQHash interface. Conversely, components operating over this interface can assume their existence, and use them to abstractly perform the same kinds of simplifications as done with Bloom filters, resolving many probabilistic proofs to dealing with deterministic properties on the AMQ states.

4.2 The AMQ Interface

Building on top of an abstract AMQHash component, the AMQ interface then provides a unified view of the state of an AMQ and how it deterministically interacts with the output type `Value` of a particular hashing operation.

As before, the interface begins by abstracting the specific types and operations of the previous analysis of Bloom filters, first introducing a type `AMQState` to capture the state of the AMQ, and then assuming deterministic implementations of the typical *add* and *query* operations of an AMQ:

```
Parameter AMQ_add_internal: AMQState → Value → AMQState.
Parameter AMQ_query_internal: AMQState → Value → bool.
```

In the case of Bloom filters, these would be instantiated with the `BF`, `bf_add_int` and `bf_query_int` operations respectively (*cf.* Sec. 3.2), thereby setting the associated hashing operation to the hash vector (Sec. 3.3).

As we move on to reason about the behaviours of these operations, the interface diverges slightly from that of the Bloom filter by conditioning the behaviours on the assumption that the state has sufficient capacity:

```
Parameter AMQ_available_capacity: AMQState → nat → bool.
```

While the Bloom filter has no real deterministic notion of a capacity, this cannot be said of all AMQs in general, such as the Counting Bloom filter or Quotient filter, as we will discuss later.

With these definitions in hand, the behaviours of the AMQ operations are characterised using a series of associated assumptions:

Property 3 (AMQ insertion validity) *For a state s with sufficient capacity, inserting any hash output ιs into s via* `AMQ_add_internal` *will produce a new state s' for which any subsequent queries for ιs via* `AMQ_query_internal` *will return* `true`*.*

Property 4 (AMQ query preservation) *For any AMQ state s with sufficient remaining capacity, if queries for a particular hash output ιs in s via* `AMQ_query_internal` *happen to return* `true`*, then inserting any further outputs $\iota s'$ into s will return a state for which queries for ιs will* still *return* `true`*.*

Even though these assumptions seemingly place strict restrictions on the permitted operations, we found that these properties are satisfied by most common AMQ structures. One potential reason for this might be because they are in fact *sufficient* to ensure the No-False-Negatives property standard of most AMQs:

Theorem 7 (Generalised No False Negatives). *For any AMQ state s, a corresponding hash state hs, after having inserted an element x into s, followed by a series xs of other inserted elements, the result of query for x is always* `true`*. That is,* $\Pr\left[\texttt{AMQ\_add}\ x\ (hs, s)\ \rhd\ \texttt{AMQ\_addm}\ xs\ \rhd\ \texttt{AMQ\_query}\ x\right] = 1$.

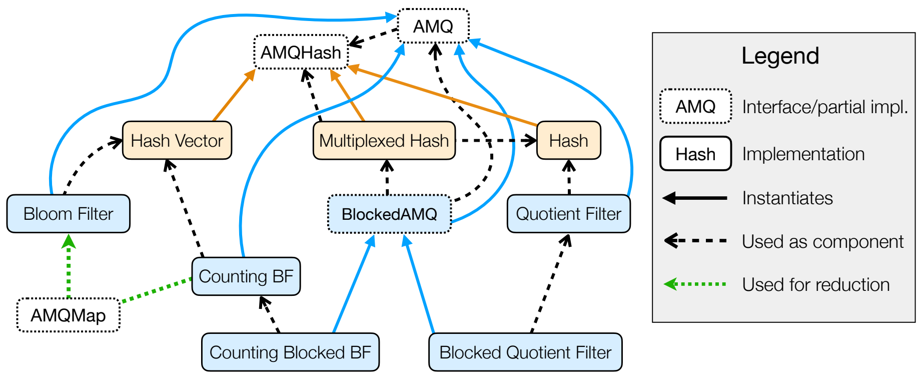


Fig. 1: Overview of Ceramist and the dependencies between its components.

Here, `AMQ_add`, `AMQ_addm`, and `AMQ_query` are generalisations of the probabilistic wrappers of Bloom filters (*cf.* Sec. 3.1) for doing the bookkeeping associated with hashing and delegating to the internal deterministic operations.

The generalised Theorem 7 illustrates one of the key facilities of our framework, wherein by simply providing components satisfying the AMQHash and AMQ interfaces, it is possible to obtain proofs of certain standard probabilistic properties or simplifications *for free*.

The diagram in Fig. 1 provides a high-level overview of the interfaces of Ceramist, their specific instances, and dependencies between them, demonstrating Ceramist's take on compositional reasoning and proof reuse. For instance Bloom filter implementation instantiates the AMQ interface implementation and uses, as a component, hash vectors, which themselves instantiate AMQHash used by AMQ. Bloom filter itself is also used as a proof reduction target by Counting Bloom filter. We will elaborate on this and the other noteworthy dependencies between interfaces and instances of Ceramist in the following sections.

4.3 Counting Bloom Filters through Ceramist

To provide a concrete demonstration of the use of the AMQ interface, we now switch over to a new running example—Counting Bloom filters [45]. A Counting Bloom filter is a variant of the Bloom filter in which individual bits are replaced with counters, thereby allowing the removal of elements. The implementation of the structure closely follows the Bloom filter, generalising the logic from bits to counters: insertion increments the counters specified by the hash outputs, while queries treat counters as set if greater than 0. In the remainder of this section, we will show how to encode and verify the Counting Bloom filter for the standard AMQ properties. We have also proven two novel domain-specific properties of Counting Bloom filters, which, due to space limits, we outline in Appendix A.

First, as the Counting Bloom filter uses the same hashing strategy as the Bloom filter, the hash interface can be instantiated with the Hash Vector structure used for the Bloom filter, entirely reusing the earlier proofs on hash vectors.

Next, in order to instantiate the AMQ interface, the state parameter can be defined as a vector of bounded integers, all initially set to 0:

```
Record CF := mkCF { countingbloomfilter_state: m.-tuple ℤp }.
Definition cf_new : CF := (* a new CF with all counters set to 0 *).
```

As mentioned before, the *add* operation increments counters rather than setting bits, and the *query* operation treats counters greater than 0 as raised.

$$\texttt{cf\_add\_int} : \texttt{CF} \to \texttt{seq}\ \mathbb{Z}_m \to \texttt{CF}$$
$$\texttt{cf\_query\_int} : \texttt{CF} \to \texttt{seq}\ \mathbb{Z}_m \to \texttt{bool}$$

To prevent integer overflows, the counters in the Counting Bloom filter are bounded to some range $\mathbb{Z}_p$, so the overall data structure too has a maximum capacity. It would not be possible to insert any values if doing such would raise any of the counters above their maximum. To account for this, the capacity parameter of the AMQ interface is instantiated with a simple predicate `cf_available_capacity` that verifies that the structure can support l further inserts by ensuring that each counter has at least $k * l$ spaces free (where k is the number of hash functions used by the data structure).

The add operation can be shown to be monotone on the value of any counter when there is sufficient capacity (Property 3). The remaining properties of the operations also trivially follow, thereby completing the instantiation, and allowing the automatic derivation of the No-False-Negatives result via Theorem 7.

4.4 Proofs about False Positive Probabilities by Reduction

As the observable behaviour of Counting Bloom filter almost exactly matches that of the Bloom filter, it seems reasonable that the same probabilistic bounds should also apply to the data structure. To facilitate these proof arguments, we provide the AMQMap interface that allows the derivation of probabilistic bounds by reducing one AMQ data structure to another.

The AMQMap interface is parameterised by two AMQ data structures, AMQ A and B, using the same hashing operation. It is assumed that corresponding bounds on False Positive rates have already been proven for AMQ B, while have not for AMQ A. The interface first assumes the existence of some mapping from the state of AMQ A to AMQ B, which satisfies a number of properties:

```
Parameter AMQ_state_map: A.AMQState → B.AMQState.
```

In the case of our Counting Bloom filter example, this mapping would convert the Counting Bloom filter state to a bit vector by mapping each counter to a raised bit if its value is greater than 0. To provide the of the false positive rate boundary, the AMQMap interface then requires the behaviour of this mapping to satisfy a number of additional assumptions:

Property 5 (AMQ Mapping Add Commutativity) *Adding a hash output to the AMQ B obtained by applying the mapping to an instance of AMQ A produces the same result as first adding a hash output to AMQ A and then applying the mapping to the result.*

Property 6 (AMQ Mapping Query Preservation) *Applying B's query operation to the result of mapping an instance of AMQ A produces the same result as applying A's query operation directly.*

In the case of reducing Counting Bloom filters (A) to Bloom filters (B), both results follow from the fact that after incrementing the some counters, all of them will have values greater than 0 and thus be mapped to raised bits.

Having instantiated the AMQMap interface with the corresponding function and proofs about it, it is now possible to derive the false positive rate of Bloom filters for Counting Bloom filters for free through the following generalised lemma:

Theorem 8 (AMQ False Positive Reduction). *For any two AMQs A, B, related by the* AMQMap *interface, if the false positive rate for B after inserting l items is given by the function f on l, then the false positive rate for A is also given by f on l. That is, in terms of probabilities:*

$$\Pr\left[\texttt{B.AMQ\_addm}\ xs\ (hs, \texttt{B.AMQ\_new}) \rhd \texttt{B.AMQ\_query}\ y\right] = f(\texttt{length}\ xs) \implies$$
$$\Pr\left[\texttt{A.AMQ\_addm}\ xs\ (hs, \texttt{A.AMQ\_new}) \rhd \texttt{A.AMQ\_query}\ y\right] = f(\texttt{length}\ xs).$$

5 Proof Automation for Probabilistic Sums

We have, until now, avoided discussing details of how facts about the probabilistic computations can be composed, and thereby also the specifics of how our proofs are structured. As it turns out, most of this process resolves to reasoning about summations over real values as encoded by Ssreflect's bigop library. Our development also relies on the tactic library by Martin-Dorel and Soloviev [31].

In this section, we outline some of the most essential proof principles facilitating the proofs-by-rewriting about probabilistic sums. While most of the provided rewriting primitives are standalone general equality facts, some of our proof techniques are better understood as combining a series of rewritings into a more general rewriting pattern. To delineate these two cases, will use the terminology **Pattern** to refer to a general pattern our library supports by means of a dedicated Coq tactic, while **Lemma** will refer to standalone proven equalities.

5.1 The Normal Form for Composed Probabilistic Computations

When stating properties on outcomes of a probabilistic computation (*cf.* Sec. 3.1), the computation must first be recursively evaluated into a distribution, where the intermediate results are combined using the probabilistic `bind` operator. Therefore, when decomposing a probabilistic property into smaller subproofs, we must rely on its semantics that is defined for discrete distributions as follows:

$$\texttt{bind\_dist}\ (P : \texttt{dist}\ A)\ (f : A \to \texttt{dist}\ B) \triangleq \sum_{a:\ A} \sum_{b:\ B} P\ a\ \times\ (f\ a)\ b$$

Expanding this definition, one can represent any statement on the outcome of a probabilistic computation in a *normal form* composed of only nested summations over a product of the probabilities of each intermediate computational step. This paramount transformation is captured as the following pattern:

Pattern 1 (Bind normalisation)

$$\Pr[(c_1 \rhd \ldots \rhd c_m) = v] = \sum_{v_1} \cdots \sum_{v_{m-1}} \Pr[c_1 = v_1] \times \cdots \times \Pr[c_m\ v_{m-1} = v]$$

Here, by $c_i\ v_{i-1} = v_i$, we denote the event in which the result of evaluating the command $c_i\ v_{i-1}$ is v_i, where v_{i-1} is the result of evaluating the previous command in the chain. This transformation then allows us to resolve the proof of a given probabilistic property into proving simpler statements on its substeps. For instance, consider the implementation of Bloom filter's query operation from Section 3.1. When proving properties of the result of a particular query (as in Theorem 3), we use this rule to decompose the program into its component parts, namely as being the product of a hash invocation $\Pr[\texttt{hash\_vec\_int}\ x\ hs]$ and the deterministic query operation `bf_query_int`. This allows dealing with the hash operation and the deterministic component *separately* by applying subsequent rewritings to each factor on the right-hand side of the above equality.

5.2 Probabilistic Summation Patterns

Having resolved a property into our normal form via a tactic implementing Pattern 1, the subsequent reductions rely on the following patterns and lemmas.

Sequential composition. When reasoning about the properties of composite programs, it is common for some subprogram e to return a probabilistic result that is then used as the arguments for a probabilistic function f. This composition is encapsulated by the operation $e \rhd f$, as used by Theorems 3, 2, and 4. The corresponding programs, once converted to the normal form, are characterised by having factors within its internal product that simply evaluate the probability of the final statement `ret` v' to produce a particular value v_k:

$$\sum_{v_1} \cdots \sum_{v_{m-1}} \underbrace{\Pr[c_1 = v_1] \times \cdots \Pr\left[\texttt{ret}\ v' = v_k\right]}_{e} \underbrace{\cdots \times \Pr[c_m\ v_{m-1} = v]}_{f}$$

Since the return operation is defined as a delta distribution with a peak at the return value v', we can simplify the statement by removing the summation over v_k, and replacing all occurrences of v_k with v', via the following pattern:

Pattern 2 (Probability of a Sequential Composition)

$$\sum_{v_1} \cdots \sum_{v_{m-1}} \Pr\left[\texttt{ret}\ v' = v_1\right] \cdots \times \Pr[c_m\ v_{m-1} = v]] =$$
$$\sum_{v_2} \cdots \sum_{v_{m-1}} \Pr\left[[v'/v_1](c_2\ v_1) = v_2\right] \times \cdots \times \Pr\left[[v'/v_1]c_m\ v_{m-1} = v\right]$$

Notice that, without loss of generality, Pattern 2 assumes that the v'-containing factor is in the head. Our tactic implicitly rewrites the statement to this form.

Plausible statement sequencing. One common issue with the normal form, is that, as each statement is evaluated over the entirety of its support, some of the dependencies between statements are obscured. That is, the outputs of one statement may in fact be constrained to *some subset* of the complete support.

To recover these dependencies, we provide the following theorem, that allows reducing computations under the assumption that their inputs are plausible:

Lemma 3 (Plausible Sequencing). *For any computation sequence $c_1 \rhd c_2$, if it is possible to reduce the computation $c_2\ x$ to a simpler form $c_3\ x$ when x is amongst plausible outcomes of c_1, (*i.e., $\Pr[c_1 = x] \neq 0$ *holds) then it is possible to rewrite c_2 to c_3 without changing the resulting distribution:*

$$\sum_x \sum_y \Pr[c_1 = x] \times \Pr[c_2\ x = y] = \sum_x \sum_y \Pr[c_1 = x] \times \Pr[c_3\ x = y]$$

Plausible outcomes. As was demonstrated in the previous paragraph, it is sometimes possible to gain knowledge that a particular value v is a plausible outcome for a composite probabilistic computation $c_1 \rhd \ldots \rhd c_m$:

$$\sum_{v_1} \cdots \sum_{v_{m-1}} \Pr[c_1 = v_1] \times \cdots \times \Pr[c_m\ v_{m-1} = v] \neq 0$$

This fact in itself is not particularly helpful as it does not immediately provide any usable constraints on the value v. However, we can now turn this inequality into a conjunction of inequalities for individual probabilities, thus getting more information about the intermediate steps of the computation:

Pattern 3 *If* $\sum_{v_1} \cdots \sum_{v_{m-1}} \Pr[c_1 = v_1] \times \cdots \times \Pr[c_m\ v_{m-1} = v] \neq 0$, *then there exist* $v_1, \ldots, v_{m-1}$ *such that* $\Pr[c_1 = v_1] \neq 0 \wedge \cdots \wedge \Pr[c_m = v] \neq 0$.

This transformation is possible due to the fact that probabilities are always non-negative, thus if a summation is positive, there must exist at least one element in the summation that is also positive.

Summary of the development. By composing these components together, we obtain a comprehensive toolbox for effectively reasoning about probabilistic computations. We find that our summation patterns end up encapsulating most of the book-keeping associated with our encoding of probabilistic computations, which, combined with the AMQ/AMQHash decomposition from Sec. 4, allows for a fairly straightforward approach for verifying properties of AMQs.

5.3 A Simple Proof of Generalised No False Negatives Theorem

To showcase the fluid interaction of our proof principles in action, let us consider the proof of the generalised No-False-Negatives Theorem 7, stating the following:

$$\Pr\left[\underbrace{\texttt{AMQ\_add}\ x\ (hs, s)}_{(a),(b)} \rhd \underbrace{\texttt{AMQ\_addm}\ xs}_{(c)} \rhd \underbrace{\texttt{AMQ\_query}\ x}_{(d),(e)}\right] = 1 \qquad (1)$$

As with most of our probabilistic proofs, we begin by applying normalisation Pattern 1 to reduce the computation into our normal form:

$$\sum_{\iota s_0, hs_0} \sum_{s_0} \sum_{s_1, hs_1} \sum_{\iota s_2, hs_2} \begin{pmatrix} (a)\ \Pr[\texttt{AMQHash\_hash}\ x\ hs = (\iota s_0, hs_0)] & \times \\ (b)\ \Pr[\texttt{ret}\ (\texttt{AMQ\_add\_internal}\ s\ \iota s_0) = s_0] & \times \\ (c)\ \Pr[\texttt{AMQ\_addm}\ xs\ (s_0, hs_0) = (s_1, hs_1)] & \times \\ (d)\ \Pr[\texttt{AMQHash\_hash}\ x\ hs_1 = (\iota s_2, hs_2)] & \times \\ (e)\ \Pr[\texttt{ret}\ (\texttt{AMQ\_query\_internal}\ s_1\ \iota s_2)] & \end{pmatrix}$$

We label the factors to be rewritten as (a)–(e) for the convenience of the presentation, indicating the correspondence to the components of the statement (1). From here, as all values are assumed to be unseen, we can use Property 1 in conjunction with the sequencing Pattern 2 to reduce factors (a) and (b) as follows:

$$\sum_{\iota s_0} \sum_{s_1, hs_1} \sum_{\iota s_2, hs_2} \begin{pmatrix} (a)\ p_{\text{hash}} & \times \\ (c)\ \Pr\left[\texttt{AMQ\_addm}\ xs\ ((s \leftarrow_{\text{add}} \iota s_0), (hs \leftarrow_{\text{hash}} (x : \iota s_0))) = (s_1, hs_1)\right] & \times \\ (d)\ \Pr\left[\texttt{AMQHash\_hash}\ x\ hs_1 = (\iota s_2, hs_2)\right] & \times \\ (e)\ \Pr\left[\texttt{AMQ\_query\_internal}\ s_1\ \iota s_2\right] & \end{pmatrix}$$

Here, p_{hash} is the probability from the statement of Property 1. We also introduce the notations $s \leftarrow_{\text{add}} \iota s_0$ and $hs \leftarrow_{\text{hash}} (x : \iota s_0)$ to denote the deterministic operations `AMQ_add_internal` and `AMQHash_add_internal` respectively. Then, using Pattern 3 for decomposing plausible outcomes, it is possible to separately show that any plausible hs_1 from `AMQ_addm` must map x to ιs_0, as hash operations preserve mappings. Combining this fact with Lemma 3 (plausible sequencing) and Hash Consistency (Property 2), we can derive that the execution of `AMQHash_hash` on x in (d) must return ιs_0, simplifying the summation even further:

$$\sum_{\iota s_0} \sum_{s_1, hs_1} \begin{pmatrix} (a)\ p_{\text{hash}} & \times \\ (c)\ \Pr\left[\texttt{AMQ\_addm}\ xs\ ((s \leftarrow_{\text{add}} \iota s_0), (hs \leftarrow_{\text{hash}} (x : \iota s_0))) = (s_1, hs_1)\right] & \times \\ (e)\ \Pr\left[\texttt{AMQ\_query\_internal}\ s_1\ \iota s_0\right] & \end{pmatrix}$$

Finally, as s_1 is a plausible outcome from `AMQ_addm` called on $s \leftarrow_{\text{add}} \iota s_0$, we can then show, using Property 4 (query preservation), that querying for ιs_0 on s_1 must succeed. Therefore, the entire summation reduces to the summation of distributions over their support, which can be trivially shown to be 1.

6 Overview of the Development and More Case Studies

The Ceramist mechanised framework is implmented as library in Coq proof assistant [24]. It consists of three main sub-parts, each handling a different aspect of constructing and reasoning about AMQs: (i) a library of *bounded-length data structures*, enhancing MathComp's [30] support for reasoning about finite sequences with varying lengths; (ii) a library of *probabilistic computations*, extending the infotheo probability theory library [2] with definitions of deeply embedded probabilistic computations and a collection of tactics and lemmas on summations described in Sec. 5; and (iii) the *AMQ interfaces and instances* representing the core of our framework described in Sec. 4.

Section	Size (LOC) Specifications	Proofs
Bounded containers	286	1051
Notation (§3.1)	77	0
Summations (§5)	742	2122
Hash operations (§4.1)	201	568
AMQ framework (§4.2)	594	695
Bloom filter (§3.2)	322	1088
Counting BF (§4.4, §A)	312	674
Quotient filter (§6.1)	197	633
Blocked AMQ (§6.2)	269	522

Alongside these core components, we also include four specific case studies to provide concrete examples of how the library can be used for practical verification. Our first two case studies are the mechanisation of the Bloom filter [6] and the Counting Bloom filter [45], as discussed earlier. In proving the false-positive rate for Bloom filters, we follow the proof by Bose *et al.* [8], also providing the first mechanised proof of the closed expression for Stirling numbers of the second kind. Our third case study provides mechanised verification of the quotient filter [5]. Our final case study is a mechanisation of the Blocked AMQ—a family of AMQs with a common aggregation strategy. We instantiate this abstract structure with each of the prior AMQs, obtaining, among others, a mechanisation of Blocked Bloom filters [39]. The sizes of each library component, along with the references to the sections that describe them, are given in the table above.

Of particular note, in effect due to the extensive proof reuse supported by Ceramist, the proof size for each of our case-studies *progressively decreases*, with around a 50% reduction in the size from our initial proofs of Bloom filters to the final case-studies of different Blocked AMQs instances.

6.1 Quotient Filter

A quotient filter [5] is a type of AMQ data structure optimised to be more cache-friendly than other typical AMQs. In contrast to the relatively simple internal vector-based states of the Bloom filters, a quotient filter works by internally maintaining a hash table to track its elements.

The internal operations of a quotient filter build upon a fundamental notion of *quotienting*, whereby a single p-bit hash outcome is split into two by treating the upper q-bits (the quotient) and the lower r-bits (the remainder) separately. Whenever an element is inserted or queried, the item is first hashed over a single hash function and then the output quotiented. The operations of the quotient filter then work by using the q-bit quotient to specify a bucket of the hash table, and the r-bit remainder as a proxy for the element, such that a query for an element will succeed if its remainder can be found in the corresponding bucket.

A false positive can occur if the outputs of the hash function happen to exactly collide for two particular values (collisions in just the quotient or remainder are not sufficient to produce an incorrect result). Therefore, it is then possible to reduce the event of a false positive in a quotient filter to the event that at least one in several draws from a uniform distribution produces a particular value. We encode quotient filters by instantiating the AMQHash interface from Sec. 4.1 with a *single* hash function, rather than a vector of hash functions, which is used by the Bloom filter variants (Sec. 2). The size of the output of this hashing operation is defined to be $2^q * 2^r$, and a corresponding quotienting operation is defined by taking the quotient and remainder from dividing the hash output by 2^q. With this encoding, we are able to provide a mechanised proof of the false positive rate for the quotient filter implemented using p-bit hash as being:

Theorem 9 (Quotient filter False Positive Rate). *For a hash-function hs, after inserting a series of l unseen distinct values xs into an empty quotient filter qf, for any unseen $y \notin xs$, the probability of a query $y \in_? qf$ for y returning* true *is given by:* $\Pr\left[\texttt{qf\_addm}\ xs\ (hs, \texttt{qf\_new}) \rhd \texttt{qf\_query}\ y\right] = 1 - \left(1 - \frac{1}{2^p}\right)^l$.

6.2 Blocked AMQ

Blocked Bloom filters [39] are a cache-efficient variant of Bloom filters where a single instance of the structure is composed of a vector of m independent Bloom filters, using an additional "meta"-hash operation to distribute values between the elements. When querying for a particular element, the meta-hash operation would first be consulted to select a particular instance to delegate the query to.

While prior research has only focused on applying this blocking design to Bloom filters, we found that this strategy is in fact generic over the choice of AMQ, allowing us to formalise an abstract Blocked AMQ structure, and later instantiate it for particular choices of "basic" AMQs. As such, this data structure highlights the scalability of Ceramist *wrt.* composition of programs and proofs.

Our encoding of Blocked AMQs within Ceramist is done via means of two higher-order modules as in Fig. 1: (i) a *multiplexed-hash* component, parameterised over an arbitrary hashing operation, and (ii) a *blocked-state* component, parameterised over some instantiation of the AMQ interface. The multiplexed hash captures the relation between the meta-hash and the hashing operations of the basic AMQ, randomly multiplexing hashes to particular hashing operations of the sub-components. We construct a multiplexed-hash as a composition of the hashing operation H used by the AMQ in each of the m blocks, and a meta-hash function to distribute queries between the m blocks. The state of this structure is defined as pairing of m states of the hashing operation H, one for each of the m blocks of the AMQ, with the state of the meta-hash function. As such, hashing a value v with this operation produces a *pair* of type $(\mathbb{Z}_m, \texttt{Value})$, where the first element is obtained by hashing v over the meta-hash to select a particular block, and the second element is produced by hashing v again over the hash operation H for this selected block. With this custom hashing operation, the state component of the Blocked AMQ is defined as sequence of m states of the AMQ, one for each block. The insertion and query operations work on the output of the multiplexed hash by using the first element to select a particular element of the sequence, and then use the second element as the value to be inserted into or queried on this selected state.

Having instantiated the data structure as described above, we proved the following abstract result about the false positive rate for blocked AMQs:

Theorem 10 (Blocked AMQ False Positive Rate). *For any AMQ A with a false positive rate after inserting l elements estimated as $f(l)$, for a multiplexed hash-function hs, after having inserted l distinct values xs, all unseen in hs, into an empty Blocked AMQ filter bf composed of m instances of A, for any unseen $y \notin xs$, the probability of a subsequent query $y \in_? bf$ for y returning* `true` *is given by:* $\Pr\left[\texttt{BA\_addm}\ xs\ (hs, \texttt{BA\_new}) \rhd \texttt{BA\_query}\ y\right] = \sum_{i=0}^{l} \binom{l}{i} (\frac{1}{m})^i (1 - \frac{1}{m})^{l-i} f(i)$.

We instantiated this interface with each of the previously defined AMQ structures, obtaining the Blocked Bloom filters, Counting Blocked Bloom filters and Blocked Quotient filter along with proofs of similar properties for them, for free.

7 Discussion and Related Work

Proofs about AMQs. While there has been a wealth of prior research into approximate membership query structures and their probabilistic bounds, the prevalence of paper-and-pencil proofs has meant that errors in analysis have gone unnoticed and propagated throughout the literature.

The most notable example is in Bloom's original paper [6], wherein dependencies between setting bits lead to an incorrect formulation of the bound (equation (17)), which has since been repeated in several papers [9, 14, 15, 32] and even textbooks [33]. While this error was later identified by Bose *et al.* [8], their own analysis was also marred by an error in their definition of Stirling numbers of the second kind, resulting in yet another incorrect bound, corrected two years later by Christensen *et al.* [10], who avoided the error by eliding Stirling numbers altogether, and deriving the bound directly. Furthermore, despite these corrections, many subsequent papers [13, 27–29, 39, 40, 45] still use Bloom's original incorrect bounds. For example, in Putze *et al.* [39]'s analysis of a Blocked Bloom filter, they derive an incorrect bound on the false positive rate by assuming that the false positive of the constituent Bloom filters are given by Bloom's bound.

Mechanically Verified Probabilistic Algorithms. Past research has also focused on the verification of probabilistic algorithms, and our work builds on the results and ideas from several of these developments.

The ALEA library also tackles the task of proving properties of probabilistic algorithms [3]. In contrast to our choice of a deep embedding for encoding probabilistic computations, ALEA uses a shallow embedding through a Giry monad [20], representing probabilistic programs as measures over their outcomes. As ALEA axiomatises a custom type to represent the subset of reals between 0 and 1 for capturing probabilities, they must independently prove any properties on reals required for their theorems, considerably increasing the proof effort.

The Foundational Cryptography Framework (FCF) [38] was developed for proving the security properties of cryptographic programs and provides an encoding for probabilistic algorithms. Rather than developing specific tooling for solving probabilistic obligations as we do, their library prioritises a proof strategy of proving the probabilistic properties of computations by reducing them to standard "difficult" programs with known distributions. While this strategy closely follows the typical structure of cryptographic proofs, their simple encoding increases the complexity of directly proving probabilistic properties.

Tassarotti *et al.*'s Polaris [46] library is a Coq framework for reasoning about probabilistic concurrent algorithms. Polaris uses the same reduction strategy for probabilistic specifications as the FCF library, inheriting some of the same issues with proving standalone bounds.

Hölzl considered mechanised verification of probabilistic programs in Isabelle/HOL [26]. While Hölzl uses a similar composition of probability and computation monads to encode and evaluate probabilistic programs, his construction defines the semantics of programs as infinite Markov chains, represented as a co-inductive stream of probabilistic outputs. This design makes the encoding unsuitable for capturing terminating programs, yet it is the only encoding we are aware of that enables probabilistic proofs about non-terminating programs.

Our previous effort on mechanising the probabilistic properties of blockchains also considered the encoding of probabilistic computations in Coq [23]. While that work also relied on infotheo's probability monad, it primarily considered the mechanisation of a restricted form of probabilistic properties (those with complete certainty), and did not deliver reusable tooling for this task.

While the Ceramist development is the first, to the best of our knowledge, that provides a mechanised proof of the probabilistic properties of Bloom filters, prior research has considered their deterministic properties. Blot *et al.* [7] provided a mechanised proof of the absence of false negatives for their implementation of a Bloom filter as part of their work on a library for using abstract sets to reason about the bit-manipulations in low-level programs.

Proofs of differential privacy. A popular motivation for reasoning about probabilistic computations is for the purposes of demonstrating differential privacy.

Barthe *et al.*'s CertiPriv framework [4] extends ALEA to support reasoning using a Probabilistic Relational Hoare logic, and uses this fragment to prove probabilistic non-interference arguments. However, CertiPriv focuses on proving relational probabilistic properties of coupled computations rather than explicit numerical bounds as we do. More recently, Barthe *et al.* [43] have developed a mechanisation that supports a more general coupling between distributions. In the future, we plan to employ Ceramist for extending the verification of AMQs to infer the induced probabilistic bounds on differential privacy guarantees [16].

8 Conclusion

The key properties of Approximate Membership Query structures are inherently probabilistic. Formalisations of those properties are frequently stated incorrectly, due to the complexity of the underlying proofs. We have demonstrated the feasibility of conducting such proofs in a machine-assisted framework. The main ingredients of our approach are a principled decomposition of structure definitions and proof automation for manipulating probabilistic sums. Together, they enable scalable and reusable mechanised proofs about a wide range of AMQs.

Acknowledgements. We thank Georges Gonthier, Karl Palmskog, George Pîrlea, Prateek Saxena, and Anton Trunov for their comments on the prelimiary versions of the paper. We thank the CPP'20 referees (especially Reviewer D) for pointing out that the formulation of the closed form for Stirling numbers of the second kind, which we adopted as an axiom from the work by Bose *et al.* [8] who used it in the proof of Theorem 4, implied False. This discovery has forced us to prove the closed form statement in Coq from the first principles, thus getting rid of the corresponding axiom and eliminating all potentially erroneous assumptions. Finally, we are grateful to the CAV'20 reviewers for their feedback.

Ilya Sergey's work has been supported by the grant of Singapore NRF National Satellite of Excellence in Trustworthy Software Systems (NSoE-TSS) and by Crystal Centre at NUS School of Computing.

References

1. Reynald Affeldt and Manabu Hagiwara. Formalization of Shannon's Theorems in SSReflect-Coq. In *ITP*, volume 7406 of *LNCS*, pages 233–249. Springer, 2012.
2. Reynald Affeldt, Manabu Hagiwara, and Jonas Sénizergues. Formalization of Shannon's Theorems. *J. Autom. Reasoning*, 53(1):63–103, 2014.
3. Philippe Audebaud and Christine Paulin-Mohring. Proofs of randomized algorithms in Coq. *Science of Computer Programming*, 74(8):568–589, 2009.
4. Gilles Barthe, Boris Köpf, Federico Olmedo, and Santiago Zanella Béguelin. Probabilistic relational reasoning for differential privacy. In *POPL*, pages 97–110. ACM, 2012.
5. Michael A. Bender, Martin Farach-Colton, Rob Johnson, Russell Kraner, Bradley C. Kuszmaul, Dzejla Medjedovic, Pablo Montes, Pradeep Shetty, Richard P. Spillane, and Erez Zadok. Don't thrash: How to cache your hash on flash. *PVLDB*, 5(11):1627–1637, 2012.
6. Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, July 1970.
7. Arthur Blot, Pierre-Évariste Dagand, and Julia Lawall. From Sets to Bits in Coq. In *FLOPS*, volume 9613 of *LNCS*, pages 12–28. Springer, 2016.
8. Prosenjit Bose, Hua Guo, Evangelos Kranakis, Anil Maheshwari, Pat Morin, Jason Morrison, Michiel Smid, and Yihui Tang. On the false-positive rate of Bloom filters. *Information Processing Letters*, 108(4):210–213, 2008.
9. Andrei Z. Broder and Michael Mitzenmacher. Survey: Network Applications of Bloom Filters: A Survey. *Internet Mathematics*, 1(4):485–509, 2003.
10. Ken Christensen, Allen Roginsky, and Miguel Jimeno. A new analysis of the false positive rate of a Bloom filter. *Information Processing Letters*, 110(21):944–949, 2010.
11. Coq Development Team. *The Coq Proof Assistant Reference Manual - Version 8.10*, Jan 2020. http://coq.inria.fr/.
12. Graham Cormode and S. Muthukrishnan. An improved data stream summary: the count-min sketch and its applications. *J. Algorithms*, 55(1):58–75, 2005.
13. Biplob Debnath, Sudipta Sengupta, Jin Li, David J Lilja, and David HC Du. BloomFlash: Bloom filter on flash-based storage. In *2011 31st International Conference on Distributed Computing Systems*, pages 635–644. IEEE, 2011.
14. Sarang Dharmapurikar, Praveen Krishnamurthy, Todd S. Sproull, and John W. Lockwood. Deep Packet Inspection using Parallel Bloom Filters. *IEEE Micro*, 24(1):52–61, 2004.
15. Sarang Dharmapurikar, Praveen Krishnamurthy, and David E. Taylor. Longest prefix matching using Bloom filters. *IEEE/ACM Trans. Netw.*, 14(2):397–409, 2006.
16. Úlfar Erlingsson, Vasyl Pihur, and Aleksandra Korolova. RAPPOR: Randomized Aggregatable Privacy-Preserving Ordinal Response. In *CCS*, pages 1054–1067. ACM, 2014.
17. Apache Software Foundation. Apache cassandra documentation: Bloom filters, 2016. http://cassandra.apache.org/doc/4.0/operating/bloom_filters.html.
18. Thomas Gerbet, Amrit Kumar, and Cédric Lauradoux. The Power of Evil Choices in Bloom Filters. In *DSN*, pages 101–112. IEEE Computer Society, 2015.
19. Arthur Gervais, Srdjan Capkun, Ghassan O. Karame, and Damian Gruber. On the privacy provisions of Bloom filters in lightweight bitcoin clients. In *ACSAC*, pages 326–335. ACM, 2014.

20. Michele Giry. A categorical approach to probability theory. In *Categorical aspects of topology and analysis*, pages 68–85. Springer, 1982.
21. Georges Gonthier, Assia Mahboubi, and Enrico Tassi. A Small Scale Reflection Extension for the Coq system. Technical Report 6455, Microsoft Research – Inria Joint Centre, 2009.
22. Bob Goodwin, Michael Hopcroft, Dan Luu, Alex Clemmer, Mihaela Curmei, Sameh Elnikety, and Yuxiong He. BitFunnel: Revisiting Signatures for Search. In *SIGIR*, pages 605–614. ACM, 2017.
23. Kiran Gopinathan and Ilya Sergey. Towards mechanising probabilistic properties of a blockchain. In *CoqPL 2019: The Fifth International Workshop on Coq for Programming Languages*, 2019.
24. Kiran Gopinathan and Ilya Sergey. Ceramist: Verified Hash-based Approximate Membership Structures, 2020. CAV 2020 Artefact DOI: 10.5281/zenodo.3749474, sources available at https://github.com/certichain/ceramist.
25. Ronald L. Graham, Donald E. Knuth, and Oren Patashnik. *Concrete Mathematics: A Foundation for Computer Science, 2nd Ed.* Addison-Wesley, 1994.
26. Johannes Hölzl. Markov Processes in Isabelle/HOL. In *CPP*, pages 100–111. ACM, 2017.
27. Chi Jing. Application and Research on Weighted Bloom Filter and Bloom Filter in Web Cache. In *2009 Second Pacific-Asia Conference on Web Mining and Web-based Application*, pages 187–191, 2009.
28. Yun-Zhao Li. Memory Efficient Parallel Bloom Filters for String Matching. In *2009 International Conference on Networks Security, Wireless Communications and Trusted Computing*, volume 1, pages 485–488, 2009.
29. Hyesook Lim, Jungwon Lee, and Changhoon Yim. Complement Bloom Filter for Identifying True Positiveness of a Bloom Filter. *IEEE Communications Letters*, 19(11):1905–1908, 2015.
30. Assia Mahboubi and Enrico Tassi. *Mathematical Components*. 2017. Available at https://math-comp.github.io/mcb.
31. Érik Martin-Dorel and Sergei Soloviev. A formal study of boolean games with random formulas as payoff functions. In *TYPES 2016*, volume 97 of *LIPIcs*, pages 14:1–14:22. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2018.
32. Michael Mitzenmacher. Compressed Bloom filters. *IEEE/ACM Transactions on Networking*, 10(5):604–612, 2002.
33. Michael Mitzenmacher and Eli Upfal. *Probability and Computing: Randomized Algorithms and Probabilistic Analysis*. Cambridge University Press, 2017. ISBN 978-1-107-15488-9, Second Edition.
34. Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. http://bitcoin.org/bitcoin.pdf, 2008.
35. Moni Naor and Eylon Yogev. Bloom Filters in Adversarial Environments. *ACM Trans. Algorithms*, 15(3):35:1–35:30, 2019.
36. Rupesh Nasre, Kaushik Rajan, Ramaswamy Govindarajan, and Uday P. Khedker. Scalable Context-Sensitive Points-to Analysis Using Multi-dimensional Bloom Filters. In *APLAS*, volume 5904 of *LNCS*, pages 47–62. Springer, 2009.
37. Anna Pagh, Rasmus Pagh, and S. Srinivasa Rao. An optimal Bloom filter replacement. In *SODA*, pages 823–829. SIAM, 2005.
38. Adam Petcher and Greg Morrisett. The Foundational Cryptography Framework. In *POST*, volume 9036 of *LNCS*, pages 53–72. Springer, 2015.
39. Felix Putze, Peter Sanders, and Johannes Singler. Cache-, hash-, and space-efficient bloom filters. *ACM Journal of Experimental Algorithmics*, 14, 2009.

40. Yan Qiao, Tao Li, and Shigang Chen. One memory access Bloom filters and their generalization. In *INFOCOM*, pages 1745–1753. IEEE, 2011.
41. Norman Ramsey and Avi Pfeffer. Stochastic lambda calculus and monads of probability distributions. In *POPL*, pages 154–165. ACM, 2002.
42. Nate Rush. ETH Goes Bloom: Filling up Ethereum's Bloom Filters, 2018. https://medium.com/@naterush1997/eth-goes-bloom-filling-up-ethereums-bloom-filters-68d4ce237009.
43. Pierre-Yves Strub, Tetsuya Sato, Justin Hsu, Thomas Espitau, and Gilles Barthe. Relational $\star$-liftings for differential privacy. *Logical Methods in Computer Science*, 15(4), 2019.
44. Jamie Talbot. What are Bloom filters?, 2015. https://blog.medium.com/what-are-bloom-filters-1ec2a50c68ff.
45. Sasu Tarkoma, Christian Esteve Rothenberg, and Eemil Lagerspetz. Theory and Practice of Bloom Filters for Distributed Systems. *IEEE Communications Surveys and Tutorials*, 14(1):131–155, 2012.
46. Joseph Tassarotti and Robert Harper. A separation logic for concurrent randomized programs. *PACMPL*, 3(POPL):64:1–64:30, 2019.

A Domain-Specific Properties of Counting Bloom Filters

While the No-False-Negatives and false positive rate properties are practically important aspects of an AMQ, in the case of a Counting Bloom filter, there are a few other probabilistic behaviours of the structure that are of importance. One such property is the ability to remove some elements from a Counting Bloom filter without affecting queries for other ones, by decrementing the counters corresponding to the removed element.

To demonstrate the flexibility of our framework, we also provide a mechanised proof of the validity of this removal operation, which, to the best of our knowledge, has not been previously formalised:

Theorem 11 (Counting Bloom filter removal). *For any Counting Bloom filter cf with sufficient capacity and associated hashes hs, removing a previously inserted value x' will not change the query for any other previously inserted value x, that is:* $\Pr\left[\texttt{cf\_add}\ x'\ (hs, cf) \rhd \texttt{cf\_add}\ x \rhd \texttt{cf\_remove}\ x' \rhd \texttt{cf\_query}\ x\right] = 1$.

The operation `cf_remove` from the theorem statement deletes a value from the Counting Bloom filter by decrementing the associated counters, and is provided as a custom operation externally to the other Ceramist components, as removal operations are not a typical operation in AMQ interfaces.

Our development also provides a proof of another specialised property of the structure—that inserting any value will increase the total sum of the counters by a fixed amount. This property characterises how the modified state of the Counting Bloom filter allows tracking more detailed information, than just element membership, in terms of the exact number of insertions.

Theorem 12 (Certainty of Counter Increments). *For any counting Bloom filter cf, a value y that was not previously inserted into cf, if the sum of the values of all counters d_i in cf is l, then after inserting y, the sum of the counters will certainly increment by k, that is:* $\Pr\left[\sum_{d_i \in cf} d_i = l + k\right] = 1$.