algorithms

IMPACT FACTOR 2.1

CITESCORE 4.5

Review

Bloom Filters at Fifty: From Probabilistic Foundations to Modern Engineering and Applications

Paul A. Gagniuc, Ionel-Bujorel Păvăloiu and Maria-Iuliana Dascălu

MDPI

https://doi.org/10.3390/a18120767

algorithms

Review

Bloom Filters at Fifty: From Probabilistic Foundations to Modern Engineering and Applications

Paul A. Gagniuc , Ionel-Bujorel Păvăloiu * and Maria-Iuliana Dascălu

Faculty of Engineering in Foreign Languages, National University of Science and Technology Politehnica Bucharest, RO-060042 Bucharest, Romania; paul_gagniuc@acad.ro (P.A.G.)
* Correspondence: bujor.pavaloiu@upb.ro

Abstract

The *Bloom* filter remains one of the most influential constructs in probabilistic computation, a structure that achieves a mathematically elegant balance between accuracy, space efficiency, and computational speed. Since the original formulation of Dr. Burton H. Bloom in 1970, its design principles have expanded into a family of approximate membership query (AMQ) structures that now underpin a wide spectrum of modern computational systems. This review synthesizes the theoretical, algorithmic, and applied dimensions of *Bloom* filters, tracing their evolution from classical bit-vector models to contemporary learned and cryptographically reinforced variants. It further underscores their relevance in artificial intelligence and blockchain environments, where they act as relevance filters. Core developments, which include counting, scalable, stable, and spectral filters, are outlined alongside information-theoretic bounds that formalize their optimality. The analysis extends to adversarial environments, where cryptographic hashing and privacy-oriented adaptations enhance resilience under active attack, and to data-intensive domains such as network systems, databases, cybersecurity, and bioinformatics. Through the integration of historical insight and contemporary advances in learning, security, and system design, the *Bloom* filter emerges not merely as a data structure but as a unified paradigm for computation under uncertainty. The results presented in this review support practical advances in network traffic control, cybersecurity analysis, distributed storage systems, and large-scale data platforms that depend on compact and fast probabilistic structures.

Keywords: Bloom filter; approximate membership query (AMQ); probabilistic data structures; learned Bloom filter; cuckoo filter; XOR filter; cybersecurity; antivirus

check for updates

Academic Editor: Eugene Levner

Received: 9 October 2025
Revised: 1 December 2025
Accepted: 3 December 2025
Published: 4 December 2025

Citation: Gagniuc, P.A.; Păvăloiu, I.-B.; Dascălu, M.-I. Bloom Filters at Fifty: From Probabilistic Foundations to Modern Engineering and Applications. *Algorithms* **2025**, *18*, 767. https://doi.org/10.3390/a18120767

1. Introduction

Approximate membership queries (AMQs) represent a foundational class of probabilistic data structures that balance space efficiency with computational speed in exchange for a tolerable rate of false positives. The core problem addressed by AMQs is to determine whether an element is a member of a set when the entire set cannot be stored explicitly due to memory constraints. Such data structures are often encountered in modern computing systems, and they support large-scale indexing, network routing, web caching, security filtering, and bioinformatics databases, where deterministic methods would be prohibitively expensive in terms of memory or processing overhead. The conceptual roots of AMQs can be traced to the work of Burton H. Bloom, who in 1970 introduced the *Bloom filter* as a compact probabilistic data structure for set membership testing [1]. The main formulation of Dr. Bloom offered an elegant trade-off between space and accuracy through the use of multiple hash functions to map elements into a shared bit array. In this structure, each inserted element sets a small number of bits, and membership queries simply check whether all bits that correspond to the element are set. The simplicity of this design and its tunable false positive rate made the *Bloom filter* one of the most influential data structures in computer science. Since its introduction, the *Bloom filter* has inspired a wide range of algorithmic innovations and has become a standard component in diverse applications, from web search engines and database indexing to cybersecurity systems.

The modern research landscape surrounding *Bloom filters* has evolved significantly over the past five decades. A seminal survey by Broder and Mitzenmacher [2] systematically explored their use in network systems, highlighting how probabilistic hashing enables scalable data transmission and filtering mechanisms in distributed environments. Further refinements such as compressed *Bloom filters* [3], scalable Bloom filters [4], and counting Bloom filters [5] extended the utility of the original concept, which has addressed challenges related to dynamic data, deletions, and the efficient transmission of compressed bit representations. More recently, the field has witnessed a paradigm shift with the advent of learned Bloom filters, which integrate statistical models with classical probabilistic hashing to adaptively improve space utilization and query efficiency [6]. In addition to their theoretical elegance, *Bloom filters* are central to numerous engineering applications, including those in antivirus engines and cybersecurity frameworks, where probabilistic prefilters enable the rapid elimination of non-matching signatures before deeper inspection (Appendix A.1) [7]. The current review starts from foundational concepts and moves toward modern adaptations. Namely, Section 2 describes the theoretical foundations and information bounds. Section 3 defines the engineering principles behind practical use. Section 4 surveys major variants, and Section 5 details Bloomier filters and optimal AMQ forms. Section 6 reviews modern successors such as Cuckoo, quotient, XOR, and Vacuum filters. Section 7 presents the learned-filter paradigm. Section 8 outlines applications, Section 9 examines security and privacy aspects, and Section 10 states open challenges and future directions.

2. Theoretical Foundations of Bloom Filters

The *Bloom filter* is fundamentally a probabilistic structure for efficient set membership testing under constrained memory conditions. It operates on the principle of a representation that encodes a potentially vast set of elements within a compact bit vector of fixed length m, which utilizes k independent hash functions $h_1, h_2, \ldots, h_k$, to distribute elements uniformly across this space. Each inserted element sets a subset of bits, and membership queries verify whether all corresponding positions remain set. The data structure thus guarantees no false negatives, though it allows a small, quantifiable probability of false positives. Formally, let n denote the number of elements inserted into the filter (See Appendix A.2). For a single bit in the array, the probability that it remains zero after all insertions is [1]:

$$\left(1 - \frac{1}{m}\right)^{kn}$$

Consequently, the probability that it is set to one is [1]:

$$1 - \left(1 - \frac{1}{m}\right)^{kn}$$

A membership query yields a false positive when all k bits that correspond to the queried element happen to be set by other insertions. Under the independence assumption, the probability of this event is given by [1]:

$$P = \left(1 - e^{-\frac{kn}{m}}\right)^k$$

The optimal number of hash functions that minimizes p can be derived by differentiation, yielding [1]:

$$k = \frac{m}{n} \ln 2$$

At this optimum, the false positive probability simplifies to:

$$p_{min} \approx (0.6185)^{\frac{m}{n}}$$

These relationships, originally formulated by Bloom [1], define the essential trade-offs governing all approximate-membership-query (AMQ) structures. The computational characteristics of the *Bloom filter* are appealing: both insertions and membership queries execute in $O(k)$ time, independent of the total number of elements n, while space consumption remains $O(m)$. This constant-time behavior, combined with the logarithmic relationship between space and false-positive rate, makes *Bloom filters* highly suitable for applications that demand scalable probabilistic indexing. However, the performance depends critically on the quality and independence of the hash functions. Kirsch and Mitzenmacher later demonstrated that full independence is unnecessary: through the application of a double-hashing technique, the i-th hash can be generated as:

$$g_i(x) = (h_1(x) + i \times h_2(x)) \bmod m$$

which allows for equivalent theoretical performance with only two base hashes [8]. This result drastically reduces computational overhead while it preserves the same asymptotic false-positive behavior. From an information-theoretic perspective, *Bloom filters* operate close to the lower bound of space efficiency for probabilistic set representations. Pagh et al. established that any AMQ structure that distinguishes between members and non-members with a false-positive rate p requires at least:

$$\frac{m}{n} \geq \frac{\log_2\left(\frac{1}{p}\right)}{\ln 2}$$

bits per stored element [9]. The *Bloom filter* achieves this bound within a small constant factor, which confirms its near-optimality in the entropy-space domain. The theoretical framework behind this inequality aligns *Bloom filters* with the limits of Shannon information theory, as they encode the uncertainty inherent in probabilistic membership decisions. The conceptual elegance of the *Bloom filter* enables clear pedagogical implementation, as demonstrated by Gagniuc in [7] (Appendix A). The implementation defines a bit array of length n and a list of hash functions used for insertion and membership testing (Supplementary Material S1). Each element is hashed k times, and the corresponding bits in the array are set to 1. Membership testing involves the verification of these bits, which return *True* if all are set and *False* otherwise. While simple, the code effectively demonstrates the probabilistic essence of *Bloom* filtering: compact storage, constant-time operations, and tunable accuracy. Nevertheless, this minimal model exposes certain limitations. The use of built-in *hash*() function from Python 3.14.1 which varies between runtime sessions, introduces nondeterminism that can affect reproducibility. Furthermore, deletions are unsupported, as clearing bits could inadvertently remove information from overlapping elements, a constraint that motivated subsequent developments such as *Counting Bloom Filters* [5], which maintain per-bit counters instead of binary flags. Similarly, the pedagogical version lacks bit-packing optimizations, which makes it less memory-efficient than practical implementations that exploit byte-level compression or hardware vectorization [3,4].

3. Engineering the Classic Bloom Filter

While the mathematical properties of the *Bloom filter* are well understood, its real-world performance depends critically on engineering choices related to hashing, memory layout, and data-access locality. As datasets and architectures have scaled, researchers have refined the design of the *Bloom filter* to minimize computational and bandwidth overheads while they preserve its probabilistic guarantees. These optimizations have transformed it from a theoretical construct into a highly efficient primitive in modern distributed and multicore systems. As seen above, at the core of the *Bloom filter* lies hashing. In the canonical model, each element is hashed k times using independent functions, but the repeated computation of many hashes is costly in high-throughput environments such as routers, caches, or antivirus engines. Kirsch and Mitzenmacher [8] demonstrated that full independence is unnecessary: all k hash values can be derived from two base functions,

$$g_i(x) = (h_1(x) + i \times h_2(x)) \bmod m$$

which maintains the expected false-positive probability with a significant reduction of computation [8]. This "double-hashing" method underlies most efficient Bloom filter implementations today. The choice of hash functions further affects both speed and reliability. While early designs used cryptographic hashes such as SHA-1 or MD5, these are unnecessarily slow for non-adversarial settings. High-performance implementations now favor non-cryptographic families such as *MurmurHash*, *CityHash*, or *FarmHash*, which combine excellent statistical dispersion with low CPU cost and cache-friendly behavior [10–12]. Hardware trends also influence *Bloom filter* design. Because each query accesses k pseudorandom bits across a large array, traditional implementations suffer from poor cache locality. In order to address this, Putze, Sanders, and Singler proposed blocked *Bloom filters*, which partition the bit vector into cache-line-sized blocks [13]. Elements are mapped to one block via an auxiliary hash function, which ensures that all bit accesses occur within the same cache line, and this yields lower memory latency and higher throughput [13]. On modern processors, vectorized instructions (SIMD) further accelerate bit-level operations by allowing parallel insertions and lookups within registers [13]. In distributed and bandwidth-sensitive systems, the bit array of the *Bloom filter* is often transmitted between nodes or stored compactly. Mitzenmacher introduced compressed Bloom filters, in which the bit vector is entropy-compressed after construction to save transmission or storage space [3]. Compression inevitably increases the false-positive rate slightly, as decompression distorts uniformity, but the trade-off is favorable when communication cost dominates. Nonetheless, practical deployments confirm the relevance of these principles. Real systems depend on compact membership tests to regulate throughput, reduce memory demand, and sustain rapid decisions under strict latency limits; this turns the theoretical guarantees of the filter into measurable operational value. Also, since classical *Bloom filters* do not support dynamic updates, unbounded growth, or hostile inputs, the variants introduced in the next section extend the original model to address these demands.

4. Functional Extensions and Variants

The original *Bloom filter*, while theoretically elegant, is constrained by its static structure and binary semantics, which limit its ability to handle deletions, variable-sized datasets, or dynamic updates. Over the past two decades, a rich ecosystem of functional variants has emerged, and each variant extends the probabilistic model to new computational contexts (Table 1) [14–17]. These extensions maintain the foundational principle of approximate membership queries and introduce mechanisms for count support, adaptivity, frequency estimation, and sensitivity to recency (i.e., sensitivity to recent insertions) [18–21]. Thus, they demonstrate the versatility of the original concept made by Dr. Bloom, when embedded within complex data systems in continuous evolution (Figure 1).

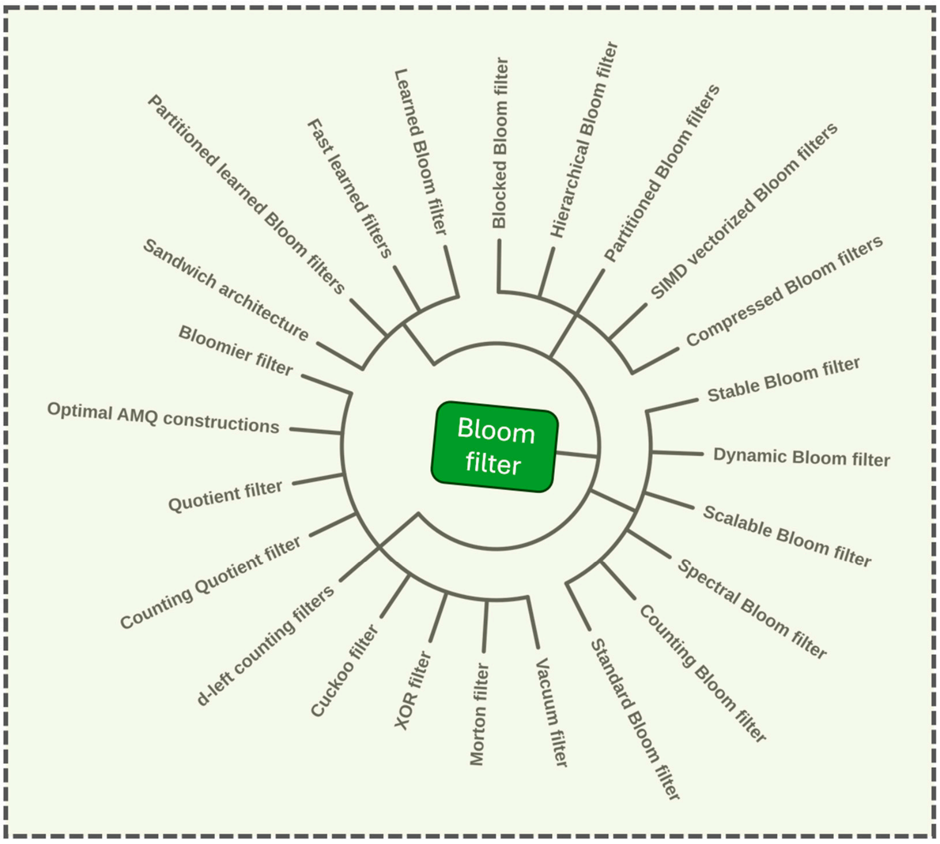

Figure 1. Conceptual taxonomy of Bloom filters and related approximate-membership query structures. The central element represents the classical Bloom filter from 1970. The branches group later developments into four main families. Core Bloom variants extend the classical form with support for deletion, multiplicity estimation, dynamic growth, and temporal decay. Engineering variants refine layout, cache use, compression, vectorized execution, and hierarchical partitioning to raise performance on modern hardware. Modern AMQs include successor structures such as Cuckoo filters, quotient-based filters, XOR filters, Morton filters, and Vacuum filters. These structures aim for near optimal space use and fast query cost. Learned and hybrid filters use statistical models together with probabilistic methods, which results in adaptive forms such as learned Bloom filters, fast learned filters, partitioned learned filters, and the sandwich architecture. The diagram displays the Bloom-filter lineage and the relations among its main descendants and replacements.

Table 1. Bloom filter versions and functional roles. A compact overview of major Bloom filter variants, their recommended use-cases, key strengths that define their operational value, and the primary limits that may affect suitability for specific systems. Each entry provides a direct comparison to assist selection of an appropriate filter design under practical constraints.

Variant	**Primary Use-Case**	**Strength**	**Limitation**
Standard Bloom Filter [1]	Static sets, minimal space	Very low memory cost	No deletion support
Counting Bloom Filter [5]	Dynamic sets with removal	Full deletion support	Higher memory cost due to counters

Table 1. *Cont.*

Variant	Primary Use-Case	Strength	Limitation
Spectral Bloom Filter [14]	Frequency estimation in data streams	Multiplicity estimates	Counter noise under high load
Scalable Bloom Filter [4]	Unknown or growing sets	Stable false-positive rate under expansion	Multiple internal layers
Partitioned Bloom Filter [16]	Cache-sensitive systems	Uniform and predictable memory access	Rigid segment structure
Blocked Bloom Filter [13]	High-performance routing and caching	Excellent cache locality	Requires careful block-size tuning
Cuckoo Filter [20]	Dynamic sets with low-latency removal	Constant-time deletion and good space usage	Relocation overhead under heavy occupancy
Quotient Filter [21]	Static sets with compact storage	Efficient space use and fast lookups	More complex structure and cluster handling

4.1. Counting Bloom Filters

In order to overcome the inability of classical *Bloom filters* to support deletions, Bonomi et al. introduced the *Counting Bloom Filter* (CBF), in which each bit in the array is replaced by a small integer counter [1,5]. Insertions increment the associated counters, while deletions decrement them; thus, the preservation of probabilistic guarantees remains valid as long as counter overflows are avoided. The CBF thus transforms the static binary vector into a dynamic multiset structure, and this transformation enables element removal without filter reconstruction. However, this functionality introduces new trade-offs: counter storage increases space consumption by a multiplicative factor proportional to the counter width, and saturation effects may yield residual false positives or negatives. Despite this cost, the CBF remains indispensable in network intrusion detection, distributed cache invalidation, and antivirus signature management, where entries must be dynamically revoked or refreshed [7].

4.2. Spectral Bloom Filters

Expanding upon the CBF, Cohen and Matias proposed the *Spectral Bloom Filter* (SBF), designed to estimate element multiplicities in data streams rather than mere presence or absence [14]. The interpretation of the counters as frequency estimators allows the SBF to act as a compact sketch structure that approximates histograms with sublinear space complexity. The expected estimation error depends on both counter width and load factor but remains bounded under uniform hashing assumptions. In contrast to traditional sketches such as *Count-Min* or *Count sketches*, the SBF preserves the probabilistic semantics of the *Bloom filter* and extends it to frequency-aware domains. It has thus found applications in flow-size measurement, keyword frequency analysis, and adaptive cache-replacement policies [14].

4.3. Scalable and Dynamic Bloom Filters

A fundamental limitation of the canonical Bloom filter is the requirement to predefine its size based on an expected number of insertions n. When n exceeds this estimate, the false-positive rate rises exponentially. Almeida et al. addressed this through the introduction of the *Scalable Bloom Filter* (SBF), which maintains a sequence of subordinate filters with geometrically reduced false-positive probabilities [4]. Upon saturation of one filter, a new instance is appended, that secures a bounded false-positive probability regardless of data growth. This hierarchical expansion preserves asymptotic space efficiency (i.e., space usage proportional to the number of stored elements) and allows the filter to adapt automatically to dataset scaling. A related concept, the *Dynamic Bloom Filter* (DBF), partitions the universe into segments managed by independent filters, that supports element deletion and efficient space reclamation (i.e., space reuse) [15]. These scalable variants have become standard in large-scale distributed storage systems and stream-processing frameworks, where input size is a priori unknown and memory elasticity is essential.

4.4. Stable Bloom Filters

For data streams in a perpetual evolution, where older elements lose relevance, Fan et al. developed the *Stable Bloom Filter* (StBF), a variant that maintains a bounded memory footprint by probabilistically decaying stored information over time [16]. Instead of indefinite growth, the StBF continuously overwrites a small random subset of bits at each insertion, effectively "forgetting" older items while it preserves approximate membership for recent ones. This design makes the StBF particularly suited for applications that require temporal adaptivity, such as anomaly detection, sliding-window analytics, and real-time intrusion monitoring. Its steady-state false-positive rate remains constant even under unbounded input, which makes it an efficient model for high-throughput streaming systems that operate under strict memory constraints [16].

4.5. Partitioned and Hierarchical Bloom Filters

Beyond functional extensions, architectural variants have been devised to optimize load balancing, memory locality, and query parallelism. *Partitioned Bloom* filters divide the bit vector into k disjoint segments, each associated with one hash function, which eliminates bit overlap and reducing inter-hash correlation [17]. This approach simplifies analysis of false-positive rates and improves cache behavior on parallel architectures. *Hierarchical Bloom* filters extend this concept by stacking multiple filters in tree-like arrangements, which enable distributed query delegation and multi-level caching [18]. At each level, filters encode progressively finer subsets of the data space, which leads to a minimization of network traffic and latency in distributed search environments. These hierarchical models underpin large-scale systems such as web caches, DNS resolvers, and distributed blockchain indexing services, where query routing efficiency and memory heterogeneity are critical [17,18].

Therefore, the technical differences between these variants translate directly into system-level effects (Table 1). Counting filters allow controlled mutation of dynamic sets, partitioned layouts provide stable cache access paths in high-rate pipelines, and scalable filters control false-positive rates in unpredictable workloads. These features illustrate why the structural design of each variant holds practical importance and why it influences efficiency across data-intensive platforms.

5. Beyond Membership: Bloomier Filters and Optimal Replacements

Bloomier filters extend the Bloom filter idea from membership tests to key-value association [19]. Their role is to store a static map of keys to small values with very low space cost, while they accept a small false-retrieval probability [19]. Instead of placing bits in a vector, a Bloomier filter uses a table of values over a finite field so that each key reconstructs its associated value through a small set of hash-based probes [19]. This shift from membership to association motivates the need for optimal AMQ structures, which aim to reach the theoretical lower limits for space while they preserve constant-time access [19]. These concepts form the basis of the more technical results that follow. The conceptual evolution of Bloom filters naturally led to a family of structures that generalize beyond binary membership testing. Among these, the Bloomier filter introduced by Chazelle, Kilian, Rubinfeld, and Tal in 2004 represents a major theoretical milestone [19]. Whereas the classical *Bloom filter* encodes the approximate membership of a set, the Bloomier filter enables static associative lookup, which maps elements to arbitrary values while it preserves space efficiency and can tolerate a small probability of false retrievals. It thus bridges the gap between probabilistic set membership and approximate dictionary data structures. The Bloomier filter constructs a hash-based linear system over a finite field, which stores value assignments in an array indexed by multiple hash functions. In the query phase, the filter recombines the entries that map to the query element through XOR operations to recover the stored value or a null indicator if the element is absent. The structure that results provides deterministic lookups for inserted keys and maintains a small false-value rate for absent keys, which achieves information-theoretic near-optimality for static data sets. This innovation established a conceptual boundary between membership-based AMQs and associative probabilistic tables, that inspired numerous successors such as d-left counting (i.e., hash-partitioned counting filters that divide the table into d disjoint subtables, and each stores the minimal counter among hashed locations to reduce collisions and memory overhead), XOR-based (i.e., minimal perfect-hash filters that store short fingerprints combined via exclusive-or operations to enable constant-time lookups with minimal space), and quotient-based filters (i.e., hash quotienting schemes that store remainders compactly in contiguous clusters to preserve locality and support efficient insertions and deletions). As previously stated, the notion of optimal approximate membership queries (AMQs) was formalized by Pagh, Pagh, and Rao in 2005, who derived lower bounds on the minimal number of bits required per element to achieve a given false-positive probability p [9]. They proved that any AMQ data structure distinguishing between members and non-members must use at least

$$\frac{\log_2\left(\frac{1}{p}\right)}{\ln 2} \tag{1}$$

bits per stored key, up to lower-order terms [9]. This bound was later interpreted as a tight information-theoretic limit that the *Bloom* filter approaches within a small constant factor. However, Pagh et al. also proposed constructions that asymptotically (i.e., in the limit as input size increases) meet this bound while they allow deletions and improved query efficiency; this establishes a rigorous optimality framework for probabilistic set representations [9]. The interplay between *Bloomier* and optimal AMQ models underscores a fundamental distinction: whereas *Bloom* filters embody probabilistic existence checks, *Bloomier* and optimal AMQs address the encoding of functional associations and structural limits of randomness in hash-based data summarization. Collectively, these contributions define the conceptual horizon of the *Bloom* filter lineage, from approximate presence detection to minimal-space associative storage. Nonetheless, the developments reviewed so far remain rooted in the bit-vector origins of the *Bloom filter*. The next section moves beyond that lineage to modern successors such as Cuckoo, quotient, XOR, and Vacuum filters, which aim for better support for dynamic updates, lower memory cost, and greater resilience to hostile inputs.

6. Modern Successors and Competing Data Structures

The resurgence of interest in approximate data structures over the last decade has produced a new generation of *Bloom* filter successors that extend or supersede its capabilities. These designs leverage advances in hashing, fingerprinting, and cache-optimized memory layouts to approach or surpass the information-theoretic limits established for classical AMQs, often with superior update performance and lower empirical false-positive rates. Among these, the *Cuckoo* filter introduced by Fan, Andersen, Kaminsky, and Mitzenmacher in 2014 constitutes one of the most influential modern successors [20]. Derived from the Cuckoo hashing paradigm, it replaces bit arrays with compact fingerprint tables. Each element is hashed into two potential buckets, and its short fingerprint is stored in one of them. Lookup and deletion operations are constant-time, as fingerprints can be directly matched or evicted using *Cuckoo*-style relocations. The *Cuckoo* filter supports dynamic insertions and deletions with predictable time complexity, and often outperforms traditional *Bloom* filters in both memory efficiency and operational speed at moderate false-positive rates ($p \approx 10^{-3}$). Moreover, its use of small fingerprints (typically 8–16 bits) yields higher cache locality and lower memory overhead, which makes it the structure of choice in high-performance networking and database systems.

Closely related to the *Cuckoo* filter are the *Quotient Filter* (QF) and its derivative, the *Counting Quotient Filter* (CQF), which encode keys as quotients and remainders of hash values. Proposed by Bender et al. and later refined by Pandey et al., these filters exploit the contiguous layout of quotient tables to improve memory access locality [20,21]. The quotient of each element identifies a slot, while its remainder is stored within a small cluster of nearby entries, which preserves insertion order and enables compact serialization. The CQF further maintains counters for multiplicities, that merges the functionality of counting and scalable *Bloom* filters [21]. These designs are highly efficient on solid-state storage and flash memory, where sequential access dominates performance constraints. The most recent wave of probabilistic filters, which includes *XOR filters*, *Morton filters*, and *Vacuum filters*, approaches theoretical optimality while it retains practical simplicity. The *XOR filter*, proposed by Graf and Lemire in 2020, constructs a minimal perfect hash representation of fingerprints such that each query requires only three memory accesses [22]. It achieves both compactness and deterministic query cost, storing roughly 1.23 $\log_2(1/p)$ bits per element, close to the Pagh-Rao lower bound. The *Morton filter*, introduced by Putze and coauthors, extends the quotient principle by an incorporation of time-decay semantics and hierarchical encoding, and it combines ideas from scalable and stable *Bloom* filters [23]. The *Vacuum filter*, a more recent refinement, applies entropy-aligned reallocation of fingerprints to eliminate wasted space in sparsely populated regions, and is able to achieve superior load factors and faster insertions than *Cuckoo* and *XOR filters* under high occupancy [24].

Across these structures, comparative analysis reveals distinct trade-offs between memory efficiency, false-positive probability, and update performance. *Bloom* filters and their counting or scalable variants minimize conceptual simplicity and code footprint, which excel in streaming and distributed environments. *Cuckoo* and *quotient* filters optimize for dynamic mutability and cache efficiency, while *XOR* and *Vacuum* filters approach the information-theoretic optimum for static sets. Insertion and query latency scale approximately linearly with the number of memory probes, typically between two and four, whereas the false-positive probability depends exponentially on fingerprint length. Empirical studies consistently show that *Cuckoo* and *XOR* filters outperform classical *Bloom* filters at equal memory budgets for $p \approx 10^{-3}$, whereas *Bloom* and scalable variants remain preferable in systems that place emphasis on incremental construction, low-latency inserts, and hardware simplicity.

7. The Learned Bloom Filter Paradigm

The rise of learned *Bloom* filters represents a conceptual departure from purely combinatorial hashing toward the integration of statistical learning into approximate membership query (AMQ) structures, where a membership query denotes a test that determines whether an element is likely to belong to a predefined set. This paradigm, introduced by Kraska, Beutel, Chi, Dean, Polyzotis, and others in 2018, reinterprets the *Bloom* filter as a probabilistic classifier trained to distinguish between members and non-members of a set [25]. Rather than an exclusive reliance on random hash functions, a learned *Bloom* filter uses a predictive model $f_\theta(x)$ that outputs a membership probability. A threshold τ is chosen such that if $f_\theta(x) \geq \tau$, the element is classified as present; otherwise, a fallback classical

Bloom filter stores the false negatives that remain. In this formulation, the model acts as a differentiable function that approximates the characteristic function of the primary set, which replaces part of the hash-induced randomness with learned structure. The hybrid data structure that results achieves a lower expected false-positive rate for skewed or structured data distributions, particularly when the input domain exhibits correlations that the model can exploit. The *sandwich architecture*, introduced in the same study and later formalized by Mitzenmacher [6], encapsulates the model between two probabilistic layers, namely a preliminary filter that removes obviously irrelevant elements and a secondary *Bloom* filter that corrects model errors. The first layer (i.e., the learned component) reduces the effective query space through latent structure in the data, while the second ensures correctness guarantees equivalent to classical AMQs. This design allows learned *Bloom* filters to maintain bounded false-negative probability and attain substantial space savings compared to traditional filters of equivalent accuracy. The sandwich model thus unifies statistical prediction and probabilistic hashing into a cohesive computational framework, a framework that merges data-driven priors with information-theoretic constraints.

Subsequent work has expanded this approach to address key limitations in scalability and inference latency (i.e., the computational delay that appears in model evaluation for each membership query). Sato, Yamamoto, and Onizuka proposed *Partitioned Learned Bloom Filters*, in which the dataset is segmented into disjoint regions, each modeled by an independent subnetwork trained to capture localized feature distributions [26]. This partition strategy reduces overfit and improves query throughput by parallel inference across small, specialized models. The same study introduced *Fast Learned Filters*, that employ lightweight linear or tree-based learners to approximate the membership function at microsecond-level latency, which reduces the computational overhead that previously constrained deployment. Together, these refinements render learned filters viable for dynamic and high-frequency workloads, which include online caching and real-time query routing (i.e., the process that directs a lookup request toward the node or data partition most likely to contain the target item).

Also, despite their theoretical promise, the practical utility of *learned Bloom* filters is strongly dataset-dependent. When the feature space exhibits high regularity, such as in URL blacklists, genomic signatures, or structured keyspaces, learned filters can dramatically reduce false positives and use less memory than classical designs. However, when data distributions are uniform, non-stationary, or adversarially manipulated, the learned component may fail to generalize, which leads to unstable false-positive and false-negative rates. Empirical studies show that beyond a critical entropy threshold, the predictive advantage of learning vanishes, and traditional *Bloom* or *Cuckoo* filters outperform due to their statistical robustness. Consequently, the learned paradigm should be viewed not as a replacement but as a conditional extension of the *Bloom* filter model, effective only under structural predictability and bounded data drift.

Open challenges remain in the case of generalization, robustness, and theoretical consistency. Data drift, namely the gradual evolution of key distributions, necessitates a continuous retrain cycle, which may offset the memory and latency gains obtained in the inference phase (i.e., the phase that computes the model output for each membership query). Adversarial robustness is an equally critical issue: learned *Bloom* filters are susceptible to targeted perturbations that exploit model gradients (that is, the variation of the data-driven model output under small adversarial perturbations of the input space) to induce false negatives, a vulnerability absent in hash-based methods. Furthermore, generalization bounds for learned AMQs are still under active investigation; formal results on sample complexity and stability under covariate shift (i.e., a mismatch between the distribution of training data and the distribution of real queries) remain scarce.

8. Applications Across Domains

The persistent influence of *Bloom* filters and their variants extends across a broad range of computational systems, from large-scale network infrastructures to cybersecurity engines and distributed storage architectures. Their ubiquity arises from the balance between probabilistic accuracy and memory efficiency, properties that make them indispensable as front-line prefilters in high-throughput data environments.

In network systems and caching, *Bloom* filters underpin core mechanisms for routing, web proxy validation, and peer-to-peer indexing. Routers employ them to represent forwarding tables compactly, which enables the rapid exclusion of non-existent destinations and reduction of control-plane overhead [2,16]. Web proxies and content-distribution networks integrate *Bloom* filters for cache admission and invalidation, allowing for billions of URLs to be tested against compact probabilistic sets [2]. In peer-to-peer (P2P) networks, they provide decentralized membership verification and query propagation suppression, and this reduces bandwidth consumption and accelerates distributed lookups.

Within database and distributed storage systems, *Bloom filters* serve as lightweight indexing structures for on-disk tables and key-value stores such as *Bigtable*, *Cassandra*, and *LevelDB* [2,16]. Each implements a *Bloom filter* per SSTable (i.e., *Sorted String Table*, a persistent, immutable file format that stores key-value pairs in sorted order for efficient lookups and sequential reads) or data block, which allows the system to avoid unnecessary disk reads for absent keys. This design reduces query latency and I/O amplification in LSM-tree architectures (i.e., *Log-Structured Merge*-tree storage systems that organize data across multiple immutable levels, and use sequential writes and periodic merges to optimize disk I/O and update performance), often yield orders-of-magnitude improvements in read performance under mixed workloads. Scalable and partitioned *Bloom* filters further extend these benefits to distributed databases and sharded storage layers, and this preserves consistent query efficiency even under dynamic resizing [4,15].

In cybersecurity and antivirus engineering, *Bloom filters* function as critical prefiltration mechanisms that enable rapid, probabilistic elimination of benign signatures before deep inspection. As demonstrated in [7], antivirus engines employ *Bloom* filters to store and test malware fingerprints, domain blacklists, and binary pattern fragments at scale. These filters drastically reduce the computational burden of pattern-matching algorithms by excluding non-matching items early in the pipeline. Beyond antivirus engines, similar principles govern intrusion-detection systems, DNS reputation services, and email spam filters, where probabilistic filtering ensures throughput under adversarial input loads.

Moreover, in bioinformatics and computational biology, *Bloom* filters have become foundational tools that address the growth of genomic data. They enable memory-efficient representation of large k-mer sets in genome assembly, sequence alignment, and metagenomic classification pipelines, where exhaustive hash tables would be computationally prohibitive. Modern DNA assemblers and taxonomic profilers employ distributed or counting *Bloom* filters to store billions of short sequence fragments, that allow for rapid membership tests for nucleotide patterns as part of the read correction and de *Bruijn* graph traversal, for example through canonicalized k-mer indexing to reduce redundancy. This probabilistic indexing paradigm underlies high-throughput tools such as *ABySS*, *Bifrost*, and *MetaCache*, which achieve scalability by trading minimal false positives for orders-of-magnitude reductions in memory footprint and I/O cost.

9. Security, Privacy, and Adversarial Robustness

While *Bloom* filters were originally conceived as neutral probabilistic data structures, their widespread deployment in networked and security-critical systems has exposed a complex landscape of adversarial vulnerabilities. The non-cryptographic nature of classical Bloom filters, coupled with deterministic hash behavior, renders them susceptible to targeted attacks that exploit hash collisions, side-channel leakage, and predictable bit patterns. Naor and Yogev formally analyzed this threat model, and proved that adaptive adversaries can, through a bounded number of queries, reconstruct significant portions of the internal state of the filter or induce specific false positives [27]. Such attacks exploit the asymmetry between insertion and query operations, and convert membership responses into an oracle for probabilistic state inference.

Collision bias also constitutes a practical vector of exploitation. Many systems employ deterministic, non-cryptographic hash functions such as *Murmur* or *CityHash* for speed; adversaries can craft colliding inputs that target identical bit positions, which inflate false-positive rates or force the filter to its capacity limit [27]. Related work in secure multiparty computation shows that side-channel observations, which include timing and cache traces, that can leak partial information about bit occupancy, a risk much amplified in shared-hardware environments [28]. Countermeasures focus on the introduction of cryptographic entropy into the hashing process. Keyed and salted hashing randomizes bit mappings across deployments, which eliminates cross-instance inference [29]. Cryptographic primitives such as HMAC-SHA-256 or BLAKE2, though computationally more expensive, provide preimage resistance and prevent adaptive collision generation [30]. Hybrid approaches combine double hashing with per-query salts (i.e., random, single-use values, often called nonces, incorporated into each hashing operation to decorrelate identical queries and prevent adaptive inference attacks), which decouple observable access patterns from stored values while it preserves theoretical false-positive bounds [31].

In parallel, privacy has emerged as a critical axis of development. Privacy-aware *Bloom* filters encode elements in encrypted or masked form, which allow membership evaluation under constrained trust. Laud and Willemson formalized encrypted *Bloom* filter protocols based on homomorphic evaluation of bitwise conjunctions, which ensures that queries reveal no structural information [31]. Dong et al. and Faber et al. extended this approach to large-scale set intersection and federated matching, which enable secure comparison of datasets across untrusted domains [29,30]. More recent systems integrate differential-privacy mechanisms that inject calibrated randomness into bit updates, balancing inference protection with statistical utility [32]. These developments converge in modern cybersecurity contexts. Probabilistic prefilters embedded in intrusion-detection pipelines or forensic provenance systems rely on cryptographically hardened *Bloom* filters to record and verify event traces without any leak of sensitive identifiers [32]. The same primitives underpin malware-signature databases with privacy guarantees, where encrypted *Bloom* filters facilitate collaborative detection across vendors while they are able to maintain proprietary secrecy. Collectively, such extensions demonstrate that *Bloom* filters can serve as both probabilistic and cryptographically secure data structures when fortified with entropy and encryption.

10. Open Problems and Future Directions

Despite half a century of research, *Bloom* filters continue to expose fundamental open problems that intersect probability theory, machine learning, and systems design. One major frontier is the formulation of provably robust learned filters, which ties together the statistical generalization guarantees of machine learning with the probabilistic bounds of AMQs. Hybrid architectures that exist today, lack formal error stability under adversarial drift; thus, a precise PAC-style generalization (i.e., low error on unseen data from the same distribution) or concentration bounds (i.e., limits on deviation of random variables from their expected values) for learned AMQs remains unsolved [25–27].

Another persistent challenge involves nonstationary data distributions. In dynamic environments, such as web crawling, DNS analytics, or blockchain indexing, the key distribution evolves, which invalidates static false-positive estimates. Adaptive AMQs that recalibrate parameters without any form of global rebuild or retrain are necessary to maintain efficiency over time. The emergence of new research in quantum and homomorphic computation hints at entirely new realizations. Quantum AMQs based on amplitude encoding or superposition query models could, in theory, achieve sublogarithmic false-positive amplification, while homomorphic variants may allow fully encrypted membership evaluation without disclosure of intermediate bit states [31]. Yet these remain theoretical, constrained by noise growth and circuit depth.

The hardware frontier presents equally significant opportunities. Energy-aware and FPGA/GPU-optimized *Bloom* filters exploit deep parallelism to achieve microsecond-scale lookups at minimal power cost [33,34]. Network interface card-level implementations integrate filters directly into packet-processing pipelines, and thus reach line-rate filter throughput for terabit-scale routing tables. However, no unified analytic model yet captures the energy-accuracy-latency trade-off across architectures.

10.1. Contemporary Context and Cross-Domain Relevance

Although the introductory section outlines the classical trajectory of *Bloom filter* research, its impact on emergent computational domains warrants further emphasis. Recent studies underscore the relevance of the *Bloom filter* variants in blockchain infrastructures, where probabilistic indexing supports scalable ledger synchronization and decentralized verification [35]. Parallel developments in social-data analytics demonstrate similar benefits, with machine learning enhanced *Bloom filters* applied to large-scale behavioral datasets and high-dimensional feature streams [36]. In cybersecurity research, probabilistic filters remain fundamental tools for forensic estimation, threat quantification, and risk-aware detection pipelines, especially in legal and financial contexts that require efficient uncertainty models [37]. Complementary work in hierarchical cyber-risk evaluation places Bloom type structures as prefilters for adversarial-event stratification and multi-layer threat assessment [38]. Collectively, these recent publications highlight a sustained evolution of *Bloom filters* from classical probabilistic constructs toward modern AI-aligned, security-aware, and distributed computational ecosystems.

10.2. Practical Interpretation of the Theoretical Framework

Beyond the unresolved theoretical challenges noted above, the broader relevance of Bloom filters can be clarified for readers who do not work with their formal derivations. The core parameters of the framework, false-positive probability, the entropy bound m/n and the optimal value of k, shape real systems in direct and measurable ways. In cybersecurity, these constraints define the precision of prefilters that support intrusion-analysis components before deep neural modules engage, as shown in recent intrusion-detection architectures that fuse probabilistic elements with metaheuristic and neural optimizers [39]. In digital-learning platforms, Bloom type structures enable compact trace summaries that assist early-activity predictors based on decision-tree models, thus with a lower computational weight of user-behavior analysis [40]. AI-driven financial-service systems rely on the same theoretical guarantees to regulate the response times of advisory or recommendation modules that must screen large sets of transactional keys with minimal memory cost [41]. These examples clarify how the mathematical structure of the *Bloom filter* shapes performance across heterogeneous operational domains and aligns the theoretical constraints of Section 2 with their practical consequences in real systems.

Moreover, learned Bloom filters extend the classical structure with a predictive component that seeks to reduce false-positive rates, yet their use introduces trade-offs that depend on the stability of input distributions and the presence of adversarial or unstructured data. A learned score may drift under domain shift, which forces the backup filter to absorb a larger volume of elements and thus raises memory cost or false-positive pressure. Real systems highlight this effect: intrusion-analysis components face unstable feature patterns, and content-moderation pipelines in social-data platforms must handle hostile inputs that attempt to bypass statistical models. Additional application contexts also confirm the limits and advantages of learned filters in large-scale systems. Within blockchain infrastructures, probabilistic filters assist ledger summaries and transaction-precheck stages, with learned components used to refine address-group classifications [42]. In AI-driven financial environments, similar structures support anomaly detection and risk triage in high-volume transaction streams where predictive models operate under strict latency and accuracy constraints [43]. These examples clarify the operational boundaries of learned Bloom filters and demonstrate their evolution across decentralized platforms, cybersecurity pipelines, and AI-oriented financial systems.

10.3. Security-Critical Risks and Future Directions

The vulnerabilities in adversarial and security-critical environments can be clarified through documented assaults that target weaknesses in hash independence, key exposure, and predictable bit layouts. Prior analyses reveal that controlled adversarial inputs can force bit saturation or exploit weak hash families in order to raise false-positive levels in classical filters [5,27,44,45]. Additional work in privacy protocols shows how hostile users can abuse Bloom filter encodings to reveal private set elements or infer sensitive structure under partial leak models [28,46]. Further examples from cryptanalysis expose flaws in basic filter deployments in record linkage and identity-matching contexts where hash collisions or structural bias enable reconstruction attacks [47]. Cryptographic enhancements reduce some of these risks, yet they face practical barriers such as secure key distribution, cost of strong hash primitives, and limits in hardware support for keyed evaluations [48]. A path for future work includes robust adversarial models, scalable cryptographic primitives that fit to AMQ structures, and hybrid probabilistic-cryptographic designs supported by deployment frameworks that define operational constraints in forensic platforms, privacy protocols, and intrusion-analysis systems [7,49].

11. Conclusions

Across five decades of development, the *Bloom* filter has transcended its original role as a compact hashing structure to become a foundational concept in probabilistic computation. Its extensions, which span from counting and scalable filters to *Cuckoo*, *XOR*, and learned architectures, demonstrate how probabilistic data summarization continues to balance theoretical optimality with engineering pragmatism. The long-term success of this structure arises from a universal trade-off between entropy, memory, and error tolerance, an equilibrium that remains central to modern computation. In security and privacy-sensitive domains, cryptographically hardened *Bloom* filters reconcile efficiency with confidentiality, while in hardware and large-scale systems, FPGA and GPU accelerations bring near real-time membership testing to terabit throughput. The learned filter paradigm redefines this legacy by the integration of statistical generalization within classical combinatorics, which extends probabilistic data structures into the age of intelligent indexing. The *Bloom* filter lineage thus continues to exemplify a deeper principle: that uncertainty, when quantified and engineered, becomes a resource for computational efficiency. Future versions of *Bloom filters* will influence new architectures that need compact structures for fast trust validation and high-rate decision pipelines. In distributed record systems, compact verification layers will enable lighter consensus paths and faster cross-chain proof checks. In cyber defense platforms, next-generation filters will guide rapid threat triage when massive event streams exceed the limits of conventional indexing. Hardware-aware variants will support microsecond-scale lookups in autonomous systems that must evaluate large rule sets under strict latency limits. Also, hybrid probabilistic and cryptographic forms will most likely allow secure checks in forensic workflows and privacy protocols. These directions, reveal a lineage that will continue to shape large-scale data systems and confirm the Bloom filter as a core instrument for future computational innovation. The consolidated insights offered here lay a foundation for progress in data-intensive environments that require compact membership structures, such as distributed indexing layers, privacy-aware verification paths, intrusion analysis systems, and financial platforms that process high-rate event streams. Future research can define tighter limits for AMQ structures, test filter variants under adversarial conditions, and evaluate hardware support in FPGA, NIC, and GPU contexts. Further work can examine the role of learned filters under domain shift and extend cryptographic forms that enable secure membership checks in privacy-aware systems. In our view, these directions outline the next steps for the Bloom filter development.

Supplementary Materials: The following supporting information can be downloaded from here: https://www.mdpi.com/article/10.3390/a18120767/s1.

Author Contributions: Conceptualization, P.A.G., I.-B.P. and M.-I.D.; methodology, P.A.G., I.-B.P. and M.-I.D.; software, P.A.G., I.-B.P. and M.-I.D.; validation, P.A.G., I.-B.P. and M.-I.D.; formal analysis, P.A.G., I.-B.P. and M.-I.D.; investigation, P.A.G., I.-B.P. and M.-I.D.; resources, P.A.G., I.-B.P. and M.-I.D.; data curation, P.A.G., I.-B.P. and M.-I.D.; writing—original draft preparation, P.A.G., I.-B.P. and M.-I.D.; writing—review and editing, P.A.G., I.-B.P. and M.-I.D.; visualization, P.A.G., I.-B.P. and M.-I.D.; supervision, P.A.G., I.-B.P. and M.-I.D.; project administration, P.A.G., I.-B.P. and M.-I.D.; funding acquisition, not applicable. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Conflicts of Interest: The authors declare no conflict of interest.

Abbreviations

The following abbreviations are used in this manuscript:

BF	Bloom Filter
CBF	Counting Bloom Filter
DNS	Domain Name System
AMQ	Approximate Membership Query
SBF	Scalable Bloom Filter
StBF	Stable Bloom Filter
DBF	Dynamic Bloom Filter
QF	Quotient Filter
CQF	Counting Quotient Filter
XORF	XOR Filter
GPU	Graphics Processing Unit
FPGA	Field-Programmable Gate Array
NIC	Network Interface Card

Appendix A

Appendix A.1 Supplementary Implementation Example

Supplementary Material S1. A minimal *Bloom* filter implementation (Python). A minimal educational example of the *Bloom* filter is provided as a supplementary resource, which illustrates the core insertion and membership-testing logic using Python is 3.14.1. The code demonstrates how multiple hash functions operate over a shared bit array to achieve probabilistic membership verification. The main repository can be found here (https://github.com/gagniuc/the-bloom-filter) (accessed on 1 November 2025).

Appendix A.2 Supplementary Glossary

Supplementary Material S2. A concise glossary of key terms related to *Bloom filter* theory is provided as an auxiliary resource. This glossary defines the principal concepts that appear in the main text, such as approximate membership queries, bit vectors, false-positive probability, entropy bounds, optimal hash counts, load factors, fallback filters, fingerprints, and prefix filters. Its purpose is to support readers who require brief clarification of specialized terminology without an alteration of the technical structure of the formal derivations in Section 2.

References

1. Bloom, B.H. Space/Time Trade-offs in Hash Coding with Allowable Errors. *Commun. ACM* **1970**, *13*, 422–426. [CrossRef]
2. Broder, A.; Mitzenmacher, M. Network Applications of Bloom Filters: A Survey. *Internet Math.* **2004**, *1*, 485–509. [CrossRef]
3. Mitzenmacher, M. Compressed Bloom Filters. *IEEE/ACM Trans. Netw.* **2002**, *10*, 604–612. [CrossRef]
4. Almeida, P.S.; Baquero, C.; Preguiça, N.; Hutchison, D. Scalable Bloom Filters. *Inf. Process. Lett.* **2007**, *101*, 255–261. [CrossRef]
5. Bonomi, F.; Mitzenmacher, M.; Panigrahy, R.; Singh, S.; Varghese, G. An Improved Construction for Counting Bloom Filters. In Proceedings of the European Symposium on Algorithms (ESA), Zurich, Switzerland, 11–13 September 2006; pp. 684–695.
6. Mitzenmacher, M. A Model for Learned Bloom Filters and Related Structures. *arXiv* **2018**, arXiv:1802.00884. [CrossRef]
7. Gagniuc, P.A. *Antivirus Engines: From Methods to Innovations, Design, and Applications*; Elsevier Syngress: Cambridge, MA, USA, 2024.
8. Kirsch, A.; Mitzenmacher, M. Less Hashing, Same Performance: Building a Better Bloom Filter. *Random Struct. Algorithms* **2008**, *33*, 187–218. [CrossRef]
9. Pagh, A.; Pagh, R.; Rao, S.S. An Optimal Bloom Filter Replacement. In Proceedings of the ACM-SIAM Symposium on Discrete Algorithms (SODA), Vancouver, British Columbia, 23 January 2005; pp. 823–829.
10. Dietzfelbinger, M. Universal Hashing and k-Wise Independent Random Variables via Integer Arithmetic without Primes. In Proceedings of the Annual Symposium on Theoretical Aspects of Computer Science, Grenoble, France, 22–24 February 1996; Lecture Notes in Computer Science (STACS). Springer: Berlin/Heidelberg, Germany, 1996; Volume 1046, pp. 569–580.
11. Mitzenmacher, M.; Vadhan, S. Why Simple Hash Functions Work: Exploiting the Entropy in a Data Stream. In Proceedings of the ACM–SIAM Symposium on Discrete Algorithms (SODA), San Fracisco, CA, USA, 20–22 January 2008; pp. 746–755.
12. Thorup, M. High Speed Hashing for Integers and Strings. *arXiv* **2015**, arXiv:1504.06804.
13. Putze, F.; Sanders, P.; Singler, J. Cache-, Hash-, and Space-Efficient Bloom Filters. *J. Exp. Algorithmics* **2009**, *14*, 1–18. [CrossRef]
14. Cohen, S.; Matias, Y. Spectral Bloom Filters. In Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD), San Diego, CA, USA, 10–12 June 2003; pp. 241–252.
15. Luo, G. Dynamic Bloom Filters. In Proceedings of the IEEE International Conference on Computer and Information Technology (CIT), Xiamen, China, 11–14 October 2009; pp. 121–125.
16. Fan, L.; Cao, P.; Almeida, J.; Broder, A.Z. Summary Cache: A Scalable Wide-Area Web Cache Sharing Protocol. *IEEE/ACM Trans. Netw.* **2000**, *8*, 281–293. [CrossRef]
17. Lee, K.S.; Byun, H.; Kim, S.W. Partitioned Bloom Filters. In Proceedings of the IEEE International Conference Communications (ICC), Ottawa, ON, Canada, 10–15 June 2012; pp. 1–6.
18. Donnet, B.; Baynat, B.; Friedman, T. Retouched Bloom Filters: Allowing Networked Applications to Trade Off False Positives Against False Negatives. In Proceedings of the ACM CoNEXT Conference, Lisboa, Portugal, 4–7 December 2006; pp. 1–12.
19. Chazelle, B.; Kilian, J.; Rubinfeld, R.; Tal, A. The Bloomier Filter: An Efficient Data Structure for Static Support Lookup Tables. In Proceedings of the ACM–SIAM Symposium on Discrete Algorithms (SODA), New Orleans, LA, USA, 11–14 January 2004; pp. 30–39.

20. Fan, B.; Andersen, D.G.; Kaminsky, M.; Mitzenmacher, M. Cuckoo Filter: Practically Better Than Bloom. In Proceedings of the ACM International Conference on Emerging Networking Experiments and Technologies (CoNEXT), Sydney, Australia, 2–5 December 2014; pp. 75–88.
21. Bender, M.A.; Farach-Colton, M.; Johnson, R.; Kraner, R.; Kuszmaul, B.C.; Medjedovic, D.; Montes, P.; Shetty, P.; Spillane, R.P.; Zadok, E. Don't Thrash: How to Cache Your Hash on Flash. In Proceedings of the USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage), Boston, MA, USA, 13–14 June 2012; pp. 1–6.
22. Graf, T.; Lemire, D. XOR Filters: Faster and Smaller Than Bloom and Cuckoo Filters. *J. Exp. Algorithmics* **2020**, *25*, 1–16. [CrossRef]
23. Alex, D. Breslow and Nuwan S. Jayasena. Morton Filters: Faster, Space-Efficient Cuckoo Filters via Biasing, Compression, and Decoupled Logical Sparsity. *Proc. VLDB Endow.* **2018**, *11*, 1041–1055.
24. Li, M.; Zheng, J.; Wang, K. Vacuum Filters: More Space-Efficient and Faster Replacement for Bloom and Cuckoo Filters. In Proceedings of the IEEE International Conference Data Engineering (ICDE), Anaheim, CA, USA, 3–7 April 2023; pp. 1611–1623.
25. Kraska, T.; Alizadeh, M.; Beutel, A.; Chi, E.H.; Dean, A.; Polyzotis, N. The Case for Learned Index Structures. In Proceedings of the ACM SIGMOD International Conference Management of Data (SIGMOD), Houston, TX, USA, 10–15 June 2018; pp. 489–504.
26. Sato, S.; Yamamoto, T.; Onizuka, M. Partitioned and Fast Learned Bloom Filters. In Proceedings of the ACM SIGMOD International Conference Management of Data (SIGMOD), Seattle, WA, USA, 18–23 June 2023; pp. 1650–1663.
27. Naor, M.; Yogev, E. Bloom Filters in Adversarial Environments. In Proceedings of the International Conference Theory of Cryptography (TCC), Warsaw, Poland, 23–25 March 2015; pp. 565–584.
28. Pinkas, B.; Schneider, T.; Zohner, M. Faster Private Set Intersection Based on OT Extension. In Proceedings of the USENIX Security Symposium, San Diego, CA, USA, 20–22 August 2014; pp. 797–812.
29. Dong, C.; Chen, L.; Wen, Z. When Private Set Intersection Meets Big Data: An Efficient and Scalable Protocol. In Proceedings of the ACM Conference Computer and Communications Security (CCS), Berlin, Germany, 4–8 November 2013; pp. 789–800.
30. Faber, S.; Jarecki, S.; Krawczyk, H.; Nguyen, Q.; Rosulek, M.; Steiner, M. Privacy-Preserving Data Sharing and Bloom Filters. In Proceedings of the ACM Conference Computer and Communications Security (CCS), Denver, CO, USA, 12–16 October 2015; pp. 131–142.
31. Niedermeyer, F.; Steinmetzer, S.; Kroll, M.; Schnell, R. Cryptanalysis of Basic Bloom Filters Used for Priva-cy-Preserving Record Linkage. *J. Priv. Confidentiality* **2014**, *6*, 59–79.
32. Zawoad, S.; Hasan, R. Providing Proofs of Past Data Possession in Cloud Forensics. In Proceedings of the 2013 IEEE 5th International Conference on Cloud Computing Technology and Science (CloudCom 2013), Bristol, UK, 2–5 December 2013; pp. 25–30.
33. Sateesan, A.; Vliegen, J.; van der Leest, V.; Cmar, R. Hardware-oriented optimization of Bloom filter algorithms and architectures for ultra-high-speed lookups in network applications. *Microprocess. Microsyst.* **2022**, *94*, 104619. [CrossRef]
34. Xiong, S.; Yao, Y.; Berry, M.; Qi, H.; Cao, Q. Frequent traffic flow identification through probabilistic Bloom filter and its GPU-based acceleration. *J. Netw. Comput. Appl.* **2017**, *87*, 60–72. [CrossRef]
35. Durachman, Y.; Wahab, A.; Rahman, A. Blockchain and the Evolution of Decentralized Finance: Navigating Growth and Vulnerabilities. *J. Curr. Res. Blockchain* **2024**, *1*, 166–177. [CrossRef]
36. Kumar, V.P.; Priya, S.; Batumalay, M. Examining the Association Between Social Media Use and Self-Reported Social Energy Depletion: A Machine Learning Approach. *J. Digit. Soc.* **2025**, *1*, 216–229.
37. Alamsyah, R.; Wahyuni, S. Quantifying the Financial Impact of Cyber Incidents: A Machine Learning Approach to Inform Legal Standards and Risk Management. *J. Cyber Law* **2025**, *1*, 264–281.
38. Mai, R. Using Information Technology to Quantitatively Evaluate and Prevent Cybersecurity Threats in a Hierarchical Manner. *Int. J. Appl. Inf. Manag.* **2023**, *3*, 1–10. [CrossRef]
39. Hamad, N.A.; Jasim, O.N. Firefly Algorithm-Optimized Deep Learning Model for Cyber Intrusion Detection in Wireless Sensor Networks Using SMOTE-Tomek. *J. Appl. Data Sci.* **2025**, *6*, 2127–2143. [CrossRef]
40. Lenus, L.; Hananto, A.R. Predicting User Engagement in E-Learning Platforms Using Decision Tree Classification: Analysis of Early Activity and Device Interaction Patterns. *Artif. Intell. Learn.* **2025**, *1*, 174–194.
41. El Emary, I.M.M.; Sanyour, R. Examination of User Satisfaction and Continuous Usage Intention in Digital Financial Advisory Platforms: An Integrated-Model Perspective. *J. Digit. Mark. Digit. Curr.* **2025**, *2*, 114–134. [CrossRef]
42. Gramlich, V.; Guggenberger, T.; Principato, M.; Schellinger, B.; Urbach, N. A multivocal literature review of decentralized finance: Current knowledge and future research avenues. *Electron. Mark.* **2023**, *33*, 11. [CrossRef]
43. Srinivasan, B.; Wahyuningsih, T. Navigating Financial Transactions in the Metaverse: Risk Analysis, Anomaly Detection, and Regulatory Implications. *Int. J. Res. Metaverse* **2024**, *1*, 59–76. [CrossRef]
44. Schnell, R.; Bachteler, T.; Reiher, J. Privacy-preserving record linkage using Bloom filters. *BMC Med. Inform. Decis. Mak.* **2009**, *9*, 41. [CrossRef]
45. Li, W.; Huang, K.; Zhang, D.; Qin, Z. Accurate Counting Bloom Filters for Large-Scale Data Processing. *Math. Probl. Eng.* **2013**, *2013*, 516298. [CrossRef]

46. Liu, B.; Zhang, X.; Shi, R.; Zhang, M.; Zhang, G. SEPSI: A Secure and Efficient Privacy-Preserving Set Intersection with Identity Authentication in IoT. *Mathematics* **2022**, *10*, 2120. [CrossRef]
47. Vatsalan, D.; Christen, P.; Verykios, V.S. A Taxonomy of Privacy-Preserving Record Linkage Techniques. *Inf. Syst.* **2013**, *38*, 946–969. [CrossRef]
48. Pinkas, B.; Schneider, T.; Tkachenko, O.; Yanai, A. Efficient Circuit-Based PSI with Linear Communication. In Proceedings of the Advances in Cryptology—EUROCRYPT, Darmstadt, Germany, 19–23 May 2019; pp. 122–153.
49. Simion, E.; Pătrașcu, A. Applied Cryptography and Practical Scenarios for Cyber Security Defense. *UPB Sci. Bull. Ser. C* **2013**, *75*, 131–142.