This article has been accepted for publication in IEEE Transactions on Dependable and Secure Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2023.3298967

On the Privacy of Multi-Versioned Approximate Membership Check Filters

Pedro Reviriego, Alfonso Sánchez-Macian, Peter C. Dillinger and Stefan Walzer

Abstract—Approximate membership filters are increasingly used in many computing and networking applications and new filter designs are being continuously presented to improve one or more performance metrics. Therefore, understanding their security and privacy is an important issue. Previous works have considered attackers that only have access to an individual filter in isolation. For applications that generate many related filters, such as a filter for a deny list that evolves over time, that analysis is insufficient. This paper considers an attacker with access to several versions of a filter that share most of the same input elements. We find that for typical implementations of Bloom, cuckoo, and quotient filters, the attacker gains little or no advantage with access to multiple versions of a filter. However, typical xor filters do reveal more information about their input elements by querying multiple versions of a filter, and we propose techniques to enhance the privacy of xor filters and others.

Index Terms—Privacy, Approximate membership checking, Bloom filters, Cuckoo filters, Quotient filters, Xor filters.

1 Introduction

Privacy is a key requirement in applications and systems that process or store sensitive data [1]. For instance, there are strong regulatory requirements to process personal data [2]. Probabilistic data structures such as sketches [3] or approximate membership check filters [4] can play a role in preserving privacy by storing summaries rather than original data. Indeed, Bloom filters have been proposed to obfuscate data while enabling approximate checking in several applications [5],[6],[7],[8],[9].

However, the fact that the original data is not preserved in the sketches nor filters does not guarantee that privacy is preserved, as shown in several works [10],[11],[12]. For approximate membership check filters, the privacy of Bloom filters was studied in [12],[13] showing that when the universe is small, an attacker that has access to the filter contents and hash functions can infer the probability that an element was in the *original set* of the filter—the set of original inputs inserted and not later removed. More recently, the distribution of the positives in the universe has also been used to extract information on the original set of the filter [14]. Similarly, attacks on Bloom filters and possible countermeasures when used for privacy preserving record linkage have also been widely studied [15]. Therefore, in some settings the privacy is not preserved although it is true that the assumptions made on the attacker and universe are quite strong.

New filter types and improvements are introduced frequently [4], to satisfy broad interest in their performance and capabilities. For example, many enhancements to the original Bloom filter [16] have been proposed over the years to reduce the false positive probability [17],[18]. Similarly, alternative filters like the cuckoo [19], quotient [20],[21] and xor [22] filters have been introduced. Recently, adaptivity has also been proposed to eliminate false positives once they occur so that new queries for the same element would not produce further false positives [23]. Where there is an interest in new filters, an interest in their privacy follows.

Security analysis must consider ways attackers might gain an advantage not predicted in existing models, and build new models to understand them. For filters, previous studies only considered that the attacker has access to a single instance of the filter. However in many applications, the filter changes over time as elements are inserted or removed, and it might be possible to learn more about the original set from multiple versions of a filter. For example, let us consider a filter that stores a list of malicious URLs. It is likely that the list is updated periodically to add new URLs and thus so would be the filter. Let us consider daily updates and an attacker that has access to say the filters that correspond to the last 15 days. Can a persistent attacker get any additional information about the presence of a given element in the filters? To the best of our knowledge, this attack model relevant to some real applications has not been studied.

In this paper, we explore the privacy of the elements stored in various approximate membership check filters when the attacker has access to several instances of the filter that store a similar set of elements with some additions and removals. Like a typical client, the attacker has only black-box access to the filters to perform queries. In this scenario, the attacker chooses an arbitrary element, and we want to determine what the attacker can infer about the element being in the original set inserted into the filters. The main contributions of the paper are:

1) To propose a model for attacking and analyzing the

P. Reviriego is with Universidad Politécnica de Madrid, Avda Complutense 30, 28040 Madrid, Spain. Email: pedro.reviriego@upm.es.
A. Sánchez-Macian is with Universidad Carlos III de Madrid, Avda de la Universidad 30, Leganés, Madrid, Spain. Email: alfonsan@it.uc3m.es.
P. C. Dillinger is with Meta Platforms, Inc., Seattle, Washington, USA. Email: peterd@meta.com.
S. Walzer is with University of Cologne, Cologne, Germany. Email: walzer@cs.uni-koeln.de.

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/

This article has been accepted for publication in IEEE Transactions on Dependable and Secure Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2023.3298967

privacy of multi-versioned approximate membership check filters.
2) To analyze the privacy of several filters under that model: Bloom, quotient, cuckoo and xor filters.
3) To show that under some typical assumptions, Bloom, quotient and cuckoo filters do not have any significant privacy loss under multi-versioned attack compared to a single instance attacker.
4) To show that xor probing filters are much more vulnerable to multi-versioned attack.
5) To propose techniques to make xor filters more robust against persistent attackers.

A naive view of randomized algorithms suggests that more randomness is better. Like many other papers [24],[25],[26],[27], we show how careful use of *less* randomness, *less* independent hash information, can give a better overall solution for a set of applications. Here, using and exposing more hash information than necessary is bad for privacy.

The rest of the paper is organized as follows: section 2 briefly describes the approximate membership check filters of interest: Bloom, cuckoo, quotient and xor filters. In section 3 we present the persistent attacker model and analyze the privacy of various filters under this model. This section also discusses some techniques to mitigate the privacy issues. The attack is evaluated in section 4 by simulation. The implications of the attack for several cases studies are briefly analyzed in section 5. The paper ends with the conclusions as well as some directions for future work, in section 6.

2 PRELIMINARIES

This section briefly describes the filters that will be considered and introduces the notation that will be used in the rest of the paper.

2.1 Bloom filters

The Bloom filter is probably the best known approximate membership check filter [16]. Bloom uses hash functions to map each element x to k positions in an array of m bits, initially all zeros. As shown in Figure 1, inserting an element into the filter sets the bits at those k positions to one. Querying an element checks the k positions and returns positive if all are set to one. The filter does not suffer false negatives, but false positives can occur when the k positions have been set by other elements inserted. The expected false positive probability depends on the number of elements inserted in the filter, the size of the bit array (m) and the number of positions associated with each element (k) [4]. Elements can be inserted dynamically as needed but not removed as setting bits back to zero can create false negatives. The counting Bloom filter is a variant that supports removals by replacing each bit with a small counter [4],[28]. This is not a memory-efficient representation, but it can easily be projected into a standard Bloom filter for broad publication in a compact form.

2.2 Cuckoo filters

Different from Bloom filters, cuckoo filters store a fingerprint of f bits per element on a table of buckets [19].

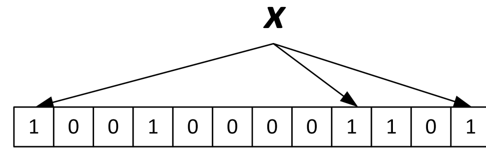

Fig. 1. Illustration of a Bloom filter with $k = 3$ functions. The lookup for x returns a positive as the three positions it maps to are set to one.

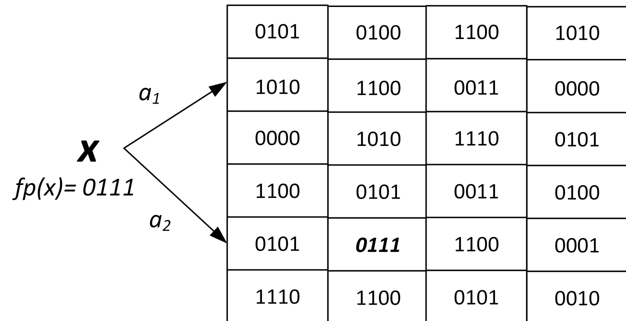

Fig. 2. Illustration of a cuckoo filter. The lookup for x returns a positive as the fingerprint stored in the second cell of bucket a_2 matches $fp(x)$.

Typically each bucket has four cells or slots that can store one fingerprint each. A given element is mapped to two buckets a_1, a_2 and its fingerprint can be stored in any of the cells of those two buckets. The values of a_1, a_2 are computed using two hash functions h_1, h_2 and the fingerprint fp as follows: $a_1 = h_1(x), a_2 = h_1(x)$ bitwise-xor $h_2(fp(x))$. Space efficiency requires hashing independence between h_1 and fp, because the fingerprints would not be as effective at avoiding false positives if they could be predicted by location in the table.

To insert an element, its fingerprint is stored in any empty cell among the two assigned buckets. If both buckets are full, then cuckoo hashing is applied: one of the fingerprints already stored there is moved to its alternative bucket to make room for the newly inserted element. This can make insertions complex as a sequence of such movements may be required. However queries are much simpler as only buckets a_1, a_2 have to be read to check if the fingerprint of the queried element is stored in any of the allowed cells. On a match, a positive is returned and a negative is returned otherwise. An advantage of cuckoo filters is support for removals (under certain assumptions), simply by locating the fingerprint and erasing it. An example cuckoo filter is shown in Figure 2.

2.3 Quotient filters

Quotient filters [20],[21] are formed by an array of cells composed of a fingerprint and three logical metadata bits as shown in Figure 3. For each element, a hash function $h(x)$ is computed and the value is divided in two parts $h(x) = (p(x), fp(x))$, the quotient and the reminder. The quotient, $p(x)$ is used to determine the cell that the element maps to that is denoted as its canonical cell and the remainder $fp(x)$ is used as the fingerprint. As there

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/

This article has been accepted for publication in IEEE Transactions on Dependable and Secure Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2023.3298967

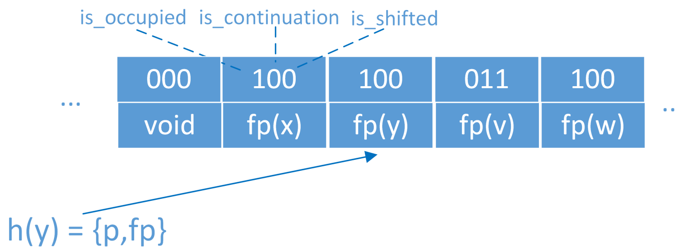

Fig. 3. Illustration of a quotient filter. The lookup for y returns a positive as the fingerprint stored in the third cell matches $fp(y)$.

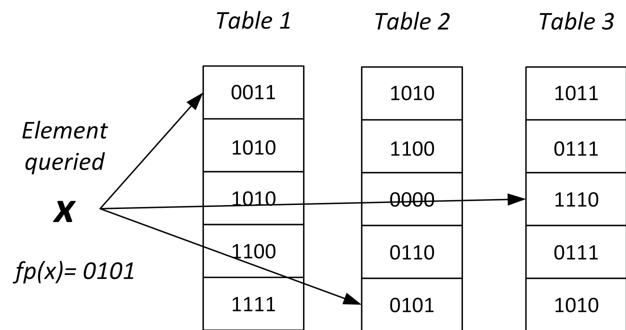

Fig. 4. Illustration of an xor filter. The lookup for x returns a negative as the xor of the values stored on the positions given by h_1, h_2, h_3 does not match $fp(x)$.

can be collisions, metadata is added to each position on the filter to enable locating $fp(x)$ when it is not stored in its canonical position $p(x)$. The metadata bits are used to signal (a) whether there are elements that map to this cell ($is_occupied$), (b) whether the element stored in this cell is in its canonical position ($is_shifted$) and finally (c) whether this element has the same canonical cell as the previous one ($is_continuation$). Optimizations can reduce actual metadata bits to two per cell [29].

To query for an element x, its canonical cell is accessed and the metadata bits are used to locate the cells that store the fingerprints associated with that cell. Then all of those are compared with $fp(x)$ and if there is a match a positive is returned, otherwise the result is a negative. Insertion is done by checking if the canonical cell is empty and if so placing the fingerprint there updating the metadata bits. If it is not empty, the element is inserted to the right after the last element that maps to that canonical cell, this may displace other elements to the right and the metadata for the relevant cells is updated. Similarly, removals are supported by just locating $fp(x)$ and removing it from the filter after possibly rearranging a few elements to keep the filter in a consistent state. As we will see in the following, quotient filters have a similar behaviour to that of cuckoo filters in the scenarios considered in this paper.

2.4 Xor filters

Another approach to design approximate membership check filters is to construct a data structure so that an element's fingerprint matches the bitwise-xor of several positions in the table [30],[31],[22],[32]. One example of this design approach is the xor filter using three tables and mapping elements to one position in each table using hash functions h_1, h_2, h_3. Each position on the tables stores an r-bit value. A query for element x returns a positive when the bitwise-xor of the contents of $h_1(x)$,$h_2(x)$,$h_3(x)$ matches the r-bit fingerprint $fp(x)$ of element x, shown in Figure 4.

Xor filters are considered static because they do not support dynamic insertions nor removals. In more detail, constructing an xor filter involves two distinct phases that each iterate over the entire original set S, so constructing a filter for a modified original set involves re-processing the entire filter, even if the first phase work is saved and largely reused. Despite this limitation, xor filters have a smaller memory footprint in many cases, especially with compression [22] or advanced techniques [26]. Likewise, since a filter is created for a given number of elements, the table size is chosen to be just large enough to support S.

In order to understand the construction of an xor filter, the two phases of this process are shown in Figure 5 and Figure 6 using a single shared table instead of three separate tables to make the example simpler.

The first part of the process maps one position of the xor table to each element. Figure 5 shows an example of this procedure. Three structures are used:

- A table that for each position holds the number of elements with a hash mapping to that position (and also a way of identifying the elements). This table is called Occupancy Table in Figure 5.
- An auxiliary stack that keeps the positions with just one active element assigned.
- The final map that stores the element-position relationship.

The algorithm works as follows:

1) In the first pass, all the hashes of all the elements from S are calculated and the occupancy table is populated with that information. For instance, the first hash of x and y (shown on the top of the figure) are mapped to the first position, so it holds 2 elements. See INITIAL step from Figure 5.
2) Once this is completed, all the positions with just one element are pushed into the auxiliary stack in order. In the figure, position 2 is pushed, then 3 and then 5 as shown in the bottom part of the INITIAL step.
3) Then, positions start to be popped from the stack and assigned to elements from the set S. In the example, the position in the top of the stack is 5. The only element that maps to this position is x, so the pair is included in the map. Then, x is no longer an active element of the algorithm and its positions are removed from the occupancy table as shown in STEP 1.
4) After removing x, three positions from the table reduced their occupancy and some of them may hold just one element. This is the case of position 1 in STEP 1 of the figure, and positions 4 and 6 in STEP 2. These

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/

This article has been accepted for publication in IEEE Transactions on Dependable and Secure Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2023.3298967

positions are pushed into the stack in the corresponding step.

5) The process continues until the auxiliary stack is empty. If all elements were mapped to positions, then the process completed successfully, otherwise, a different set of hash functions should be used and the process has to start from scratch.

At the end of this first part of the construction process, a map is obtained with a position for each element in S. This map will be used in the second part, but traversing it in reverse order (i.e. processing first the last mapping that was found).

An example of this second part of the process is described in Figure 6. The data table corresponds to the internal content of the xor filter. It could be initialized to zero or to random values as in the figure. The algorithm executes the following steps:

1) Take a pair from the map generated in the previous process. Pairs are taken in reverse order, so position 6 for element z will be used first.
2) Calculate the value to be inserted into that position as the xor of the fingerprint of the element ($f(z)$ in this case) and the value of the other positions assigned by the hash functions for the element. In the example, $data(6)$ has to be filled to generate a positive for z. The hash functions map z to positions 2, 4 and 6. Thus, the content of $data(6)$ will be the xor function of the values of $data(2)$, $data(4)$ and $f(z)$.
3) Repeat the process until the map is empty.

At the end of this execution, the xor filter content is filled with the appropriate information and the construction process ends.

Notice that adding an element, even if the size of the table is not increased, can completely change the content of the filter as it may modify the mapping from the first part of the process. This would change the occupancy table thus generating different values in the second part.

2.5 Notation

Before proceeding to describe the attacks in the next section, the main notations used in the rest of the paper are summarized in Table 1.

3 Multi-Versioned Filter Privacy

In this section we consider an attacker trying to determine whether an arbitrary element x is in any of the original sets for a series of related filters, $F_1 \ldots F_t$, or informally, multiple versions of the same filter. The first thing that the attacker can do is to query for x in each of the filters. On a negative, x cannot be in the original set for that filter. Because of false positives, however, a positive query is typically little evidence that x is in the original set for that filter [1]. Therefore, before understanding the relationship between filters, the attacker does not have strong evidence for x in any original set from these individual filter queries.

1. A sufficiently small universe or cryptographically strong hash function and false positive rate could make a false positive unlikely or infeasible.

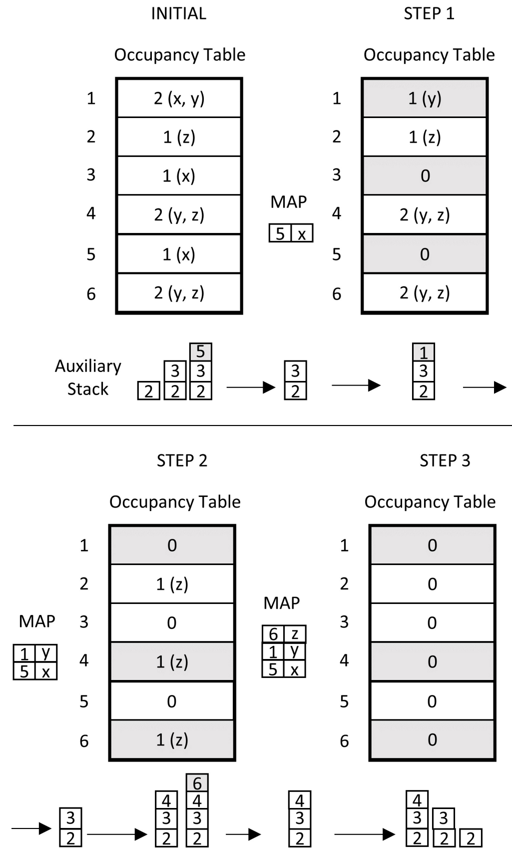

Fig. 5. Process of mapping elements to positions when building an xor filter.

We are most interested in the case in which the original sets of the F_i filters change very little from filter to filter, as this gives the attacker the best chance to draw inferences, and has applications in accept/deny lists [33],[34], packet processing [35], and databases [32]. In particular, elements shared by the original sets of all the F_i filters, call it set $\hat{S}$ of size $\hat{s} \neq 0$, have the best chance of being inferred by the attacker.

The attacker might be able to learn more by looking at filter contents rather than just the result of queries. This has been analyzed for Bloom filters [12],[13], with successful inference in some cases like a small universe of elements. To contain the scope of this paper, we assume the attacker only has access to the results of queries, not to the underlying filters (future work). However we do assume the attacker can infer or guess the kind of filter and some configuration and implementation details. A real-world example would be an online service running open source software, where there are filters on the server side and clients can infer filter query results based on the server's responses or other behaviors.

3.1 Bounds and hashing independence

The hash information available to the filter data structures can put useful bounds on the information leaked to an at-

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/

This article has been accepted for publication in IEEE Transactions on Dependable and Secure Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2023.3298967

TABLE 1
Summary of main notations

Symbol	Meaning
F_i	i^{th} instance of the filter
t	number of instances of the filter
U	universe of elements
u	number of elements in the universe
S	the original set of a filter (elements inserted)
s	size of the original set
δ	number of elements added/removed from one filter instance to the next
o	number of elements with reserved space for future insertions
b	number of bits of hash information used by each filter insertion
k	number of hash functions in a Bloom filter
r	fingerprint bit length

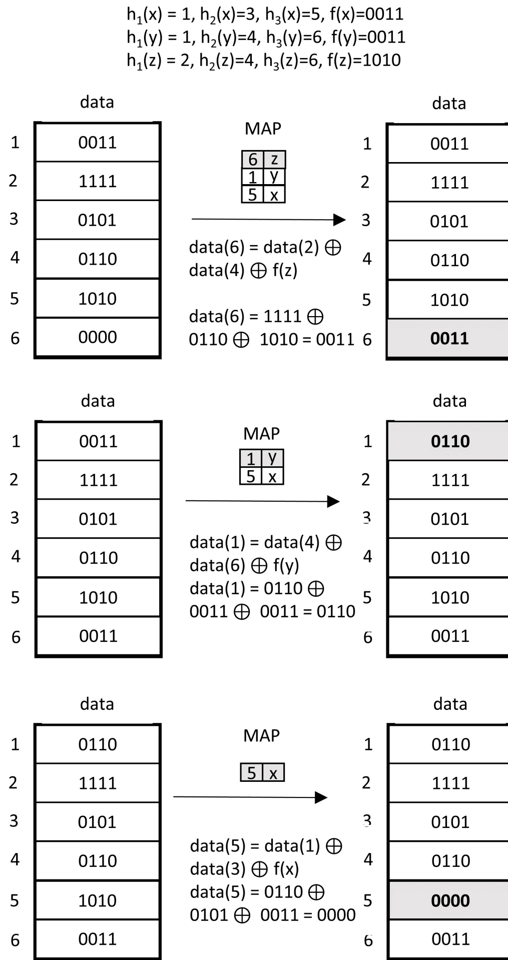

Fig. 6. Process of assigning values to positions when building a xor filter.

tacker. Let b be the number of bits of hash information used by each filter insertion in our F_i filters. If each filter uses hash functions that are independent of the other filters, then there is the potential to leak tb bits of data about an element that is in the original sets of all the filters. Specifically, for the null hypothesis that x is in none of the original sets of the filters, the probability of randomly seeing all positive queries for $x \notin \hat{S}$ could be as low as 2^{-tb}, depending on the filter implementation. Because of the compounding exponential, this will typically be overwhelming evidence for the attacker to conclude that x is in some original sets (reject the null hypothesis). In summary, independent hashing between filters is notable for leaking information about the original set, but is not the most interesting case for further consideration.

More typically, different versions of a filter will use the same base hash function(s) and seed(s), such as for simple dynamic updates to a filter. In this case, the probability of randomly seeing all positive queries for $x \notin \hat{S}$ cannot be lower than 2^{-b}, which may or may not be enough evidence for the attacker to conclude that x is in the original set. This suggests that filters using more hash information are more susceptible to leaking information to attackers, and that will play a role in analyzing various kinds of filters and their variants.

The above bounds are based on the worst case of filters containing just one element in their original set, and these bounds can be improved. (The empty filter case is excluded because there's no element to leak information about.) We are really looking for a minimum or *best case* false positive rate for a filter using b bits of hash information to insert each element, because without access to the filter representation, the false positive probabilities are the only tool for our attacker to make inferences from query results. The probability that the hash information for $x \notin \hat{S}$ collides with one of the s elements in the original set is well known:

$$1 - (1 - 2^{-b})^s \approx 1 - e^{-s/2^b} \approx s/2^b \tag{1}$$

(The joint probably for t filters with independent hashing is the same raised to the t power.) Intuitively, it might seem that s would not matter for privacy in a filter that preserves all b hash bits about each element, but that intuition comes from exact representations of elements. Probabilistic information is diluted as more entries are accumulated. In the extreme, a completely full filter with 100% false positive rate has no usable information to distinguish elements in the original set from other random elements. Conversely, this observation is a warning against sharing or exposing under-populated filters, as they could reveal much more about the small number of elements in the original set than the same filter configuration would with a much larger original set.

3.2 Filter sizing and hashing

A common pattern for building hashed data structures gives us a different bound between completely independent

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/

This article has been accepted for publication in IEEE Transactions on Dependable and Secure Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2023.3298967

hashing and completely the same hashing. For efficient memory utilization, it is common to use non-power-of-two sizes for filter data structures, which means reducing a word-sized hash value, say w bits, down to some smaller range, say $[0, m)$. For a relatively uniform mapping, typically $w \geq 5 + \log m$. The traditional approach uses the modulo (remainder) operator, but an approach based on wide multiplication is much faster and equally effective under standard assumptions [36]. The problem with both of these approaches is that a small change to the size of the data structure has widespread impact on which values collide with each other in that reduced range. An example with xor filters is shown in Figure 7. Because hash collisions play a major role in filter false positives, filters with the same base hash function(s) and seed(s) but different mapped sizes are somewhat independent within the bound of the pre-reduced hash size. The probability of randomly seeing all positive queries for x in t filters of different sizes, all roughly m, but all using the same w-bit hash function and seed, cannot be lower than $(1-(1-1/m)^s)^t$ from the somewhat independent filters, nor lower than $1-(1-2^{-w})^s$ from the pre-reduced hash. (Except where noted, we assume the F_i filters are the same size, and for generality allow b to be a non-integer, as in $b = \log m$.)

3.3 Equivalence class-based filters

Many kinds of filters including cuckoo, quotient, and $k = 1$ Bloom are essentially different ways of encoding the same kind of filtering information. Hash function(s) divide the universe of elements into equivalence classes, and the filter exactly represents the minimum set of equivalence classes that covers the original set. Equivalence classes correspond to hash values, often a composite of the hash for table location and the fingerprint hash, but could be just one of the two in some filter designs.

While quotient filters derive from an exact representation for sets [37], it is not as obvious that cuckoo filters are strictly based on equivalence classes. In the standard cuckoo filter design, there are two composite hash values for each equivalence class; each gives the same set of two buckets and same fingerprint, but returns the buckets in alternate orders. The order does not matter for false positives (scope of this paper) but is typically used for preferred insertion location, so could be partially exposed in the filter representation (future work).

In these kinds of filters, a queried element x is a false positive if it has the same equivalence class as some element in the original set. This is ideal from a privacy perspective because all of the hash information used by the filter goes to minimizing the false positive rate. Conversely, a chosen false positive rate minimizes the hash information used by the filter and, thus, opportunities for further leakage. Specifically, our prior bound for the best case false positive rate, $1-(1-2^{-b})^s$, *is* the expected false positive rate of an equivalence class-based filter.

These filters are also optimal for the multi-versioned filter case, in terms of minimizing information leakage through changes in the false positive set. For what is optimal, consider two filters with identical configuration, including a very low false positive rate, less than $1/s$, but with disjoint original sets. A false positive in one of the sets would be expected not to be a false positive in the other: otherwise would imply false positives are not uniformly randomized. Consequently, if we remove or replace δ/s elements from one filter to the next, the best we can hope for (in the very low false positive rate regime, generalization omitted) is for δ/s portion of false positives in the first filter to no longer be false positives in the second, and this is what we see with equivalence class-based filters. Although a strategy of not removing elements at all could work around this limitation, it would likely be unsustainable for maintaining a target false positive rate.

3.4 Bloom filters

False positives in Bloom filters with $k > 1$ are not based on equivalence classes, so are more complex. Also, standard Bloom filters are known to use more hash information than is strictly required for their false positive rate [24], which suggests hypotheses that (a) they are more susceptible to information leakage than filters based on equivalence classes, and (b) techniques to improve Bloom hashing efficiency could also improve privacy.

Note that the attacker should generally assume that removals from a Bloom filter are possible, either because the filter is a counting filter or comes from one, or simply because the writer of the filter can rebuild from a modified original set.

We can understand the somewhat degraded privacy of Bloom filters (vs. equivalence class-based) by their steep false positive rate curve vs. s, especially for larger k. This is illustrated in Figure 8. The first consequence of this is that under-populated Bloom filters have extremely low false positive rates. As seen in the figure, Bloom filters with k independent hash functions compound the under-population problem warned against earlier, simply by their extremely low false positive rates.

The second consequence of the steep false positive rate curve applies to multi-versioned filters. The steep curve implies bigger changes to the false positive set by adding or removing a small number of elements in the original set. It is well known that in a Bloom filter tuned to optimize false positive rate and memory efficiency, approximately half the bits are set to one. In this case, when we remove an element from the original set, we expect $k/2$ bits to revert back to zero, out of $sk/(2\ln 2) \approx 0.72sk$ bits set to one. With that, the probability that a bit previously set to one reverts back to zero is $(\ln 2)/s$, so the probability of a false positive being removed is approximately $(k \ln 2)/s \approx 0.69k/s$. Compare this to typical $1/s$ for equivalence class-based filters. For larger k, this is a notable increase in information leakage associated with multi-versioned filters, but within constant factors.

3.5 Xor filters

The case of xor filters is different from cuckoo, quotient and Bloom filters as xor filters have to be reconstructed even when a single element is added or removed. The first consequence of this is that without specific consideration, implementations are more likely to hit the leakage issues associated with independent hashing and different table

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/

This article has been accepted for publication in IEEE Transactions on Dependable and Secure Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2023.3298967

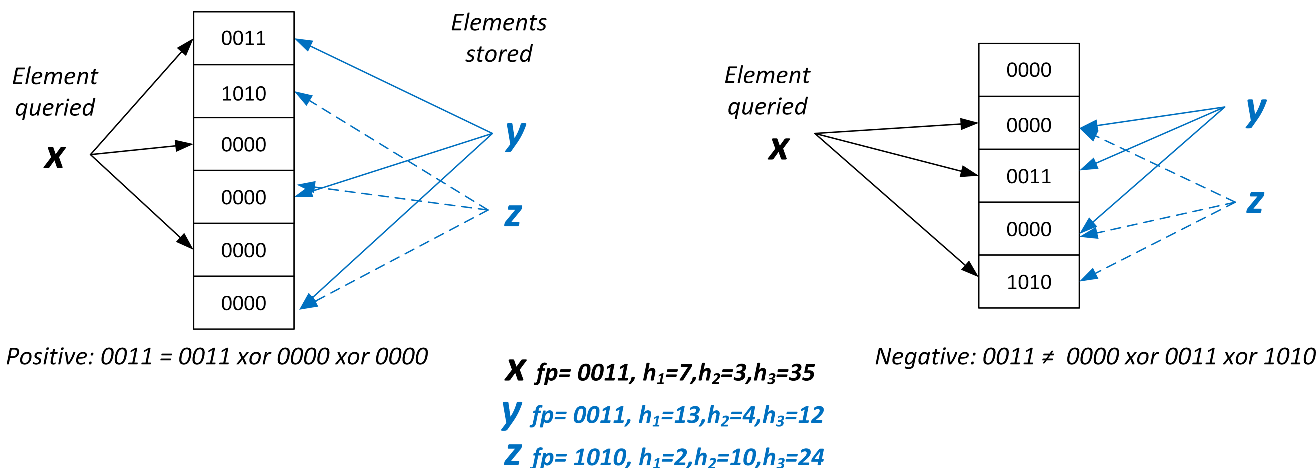

Fig. 7. Two xor filters of different sizes showing the mapping of a queried element x and two stored elements y, z. The positions are computed by taking the modulo of the hash values over the number of buckets.

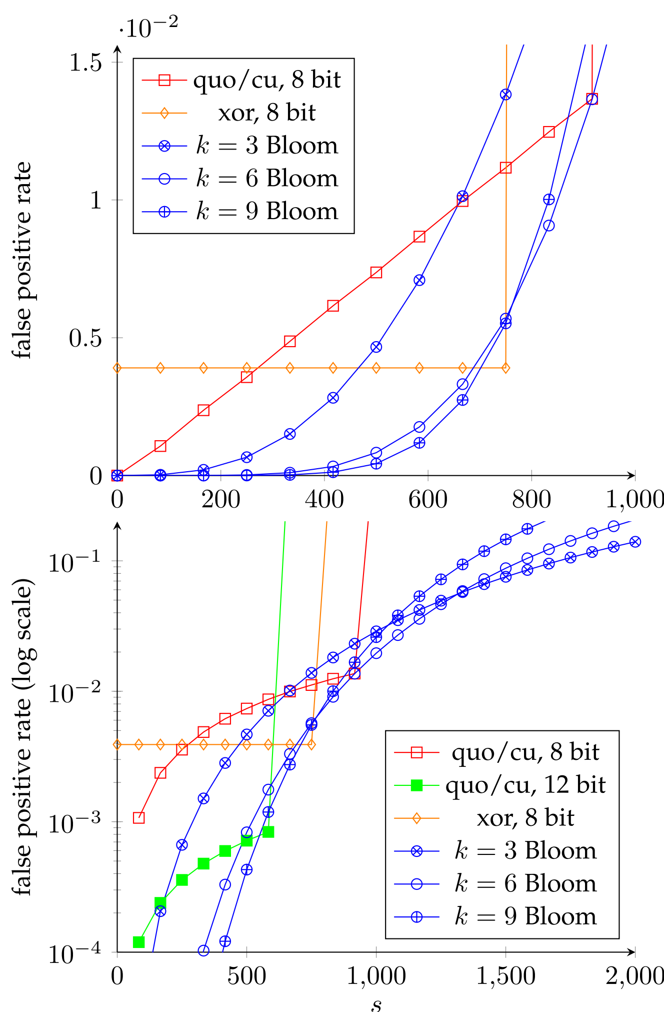

Fig. 8. Predicted false positive rates for 1KB filters of various designs and configurations, versus number of elements in the original set. Some structures fill up so are not usable beyond some s. The lower graph is mostly the same data using log scale. "quo/cu, n bit" refers to optimized quotient and cuckoo filters that lose two of those n bits per cell to either explicit or implicit metadata.

sizes, as in Figure 7. A minor contributor to this issue is that xor filter construction can fail with some probability, and that is resolved by modifying the hashing.

A related interesting aspect of xor filters is a false positive rate that is independent from the number of elements inserted (s). Although we can generally choose a false positive rate for filters, for most kinds of filters the resulting configuration only has that false positive rate when you add the intended number of elements. Thus for most kinds of filter, the false positive rate is a curve or sloped line vs. s, as in Figure 8. In a standard xor filter, the false positive rate is fixed by the configuration regardless of the number of elements inserted. A good effect is that this prevents the problem of a much lower-than-intended false positive rate associated with under-populated filters. This is almost like a cuckoo filter whose unused cells are filled with random "junk" entries, and only someone with access to the original set knows which entries are junk and which are real.

Our most interesting privacy question about xor filters is how vulnerable they are to leaks with multi-versioning when keeping hashing and table sizes constant. At first glance, xor filters use three independent hash values with $\Theta(s)$ range each, or using $3\log(s) + r + \pm O(1)$ bits of hash data, asymptotically the same as a $k = 3$ Bloom filter. Compared to $\log(s)+r+\pm O(1)$ for equivalence-class based filters, this indicates the potential for substantial information leakage from multi-versioned xor filters.

On the other hand, an aspect of standard xor filters puts a useful bound on the information leakage, at least under the working assumptions. Let a *residual false positive* refer to an element that maps to three positions unused by any elements in the original set (typical probability of $(e^{-3/1.23})^3$ or $2/3000$) and expects the xor to match all zero bits (probability of 2^{-r}), so overall typical probability of $2^{1-r}/3000$. Such false positives can be predicted without considering the element inter-dependencies in generating a full xor filter, so importantly, these false positives are much more stable under small changes to the original set, comparable to a $k = 3$ Bloom filter. Experimental validation is in Section 4. However, it is unusual for a true positive query to map to

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/

This article has been accepted for publication in IEEE Transactions on Dependable and Secure Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2023.3298967

three positions of all zero bits, even assuming the xor is all zero bits, so when the attacker has access to the filter representation, residual false positives likely do not provide the same privacy backstop for multi-versioned filters.

3.6 Privacy Protection Schemes

In this section we propose some techniques that could make it harder for an attacker to infer whether an element is in the original sets of related filters. Future work should consider more protection schemes, especially those that would address potential leakage through the filter representation.

As discussed above, minimizing the independent hash information used through a series of filters is crucial to bounding the information leakage. Many protection schemes are related to this, and work along with the core approaches already discussed, including keeping the same core hash function and seed(s) across filter versions.

3.6.1 Filter variants using less hash information

Some variants of filters not based on equivalence classes use less hash information than their standard designs. For example, Bloom filters based on double hashing [24], [38] use roughly $2\log(sk)$ bits of hash information instead of $k\log(sk) \pm O(1)$.

A promising variant of xor filter uses a technique called spatial coupling to improve table utilization (memory efficiency) and access locality [26]. Interestingly, this is accomplished using less hash information than a standard xor filter, roughly $\frac{7}{3}\log(s) + r$ bits instead of $3\log(s) + r \pm O(1)$ bits, because in a typical coupling design, all but the first probing location can only be offset by $\Theta(s^{2/3})$ from the previous.

3.6.2 Sharding

Sharding a filter means building it from sub-filters, where a hash function and/or some metadata determine which sub-filter is used for each element of the universe. Sharding is a top-down approximation or hybridization of equivalence class-based filtering, with some of the same privacy benefits. In the extreme, a sub-filter could be a single bit indicating whether anything was added or not, and a $k = 1$ Bloom filter is simply a sharding of these simplest sub-filters (recall that $k = 1$ Bloom filters are based on equivalence classes).

For a more interesting example, a cache-local Bloom filter [39] is a sharded structure using a constant c bits of memory per shard, typically $c = 512$. The hash information used is $k\log(c) + \log(sk/c) \pm O(1)$ which for large s scales more like an equivalence class-based filter ($\log(s)+r\pm O(1)$) than a standard Bloom filter ($k\log(sk) \pm O(1)$).

Sharding can also be used with xor filters, and with another likely benefit to privacy. In addition to reduced hash information use, sharding contains the scope of small changes to filters, which could be useful for containing the potential cascading effects of small changes to the original set of an xor filter. For example, adding or removing δ elements from the original set of a filter with n independent shards can modify at most (in expectation) δ/n portion of the false positives.

3.6.3 Oversizing

As discussed before, a potential protection scheme would be to oversize the filter to reduce the chances or frequency of needing to change the filter size, which exposes more base hash information through a different reduced range. There is an obvious trade-off with memory and/or I/O efficiency when using over-sizing to protect privacy, though this might be mitigated with compressed Bloom filters [40] or compressed xor filters [22].

A potential major issue with oversizing is the under-population false positive rate problem from above, which can affect most filters except xor. In other words, either over-sizing or undersizing (leading to resizing) could compromise filter privacy, which complicates representing original sets that vary in size. To limit the scope of this paper, we focus on cases in which the filter versions have original sets of roughly the same size.

4 Evaluation

The first step in the evaluation was to review the original xor filter Java code [22] for hash function usage and independence. Indeed, a random value is used to seed the hash functions, making them different each time a filter of the same size with exactly the same elements is constructed. As described before, this is an ideal setup for an attacker because full hash independence maximizes information leaked through queries (for chosen false positive rates). This was tested by 1) checking that the filter contents are completely different for different runs when inserting exactly the same set of elements and 2) checking that independence holds by simulation [2].

Now, for filters that use the same hash function in their construction, to evaluate the proposed scheme, first s elements are selected randomly and an xor filter is created with them. In the second step, another set of n elements that are not in the filter are selected randomly. Then, the n elements are queried on the filter and only the false positives are kept. Let us denote as n_1 the number of those false positives. Then, the process starts again but with an additional δ random elements in the original set, and only querying the n_1 prior false positives to determine n_2. This sequence is repeated t times so that at the end we have n_t elements that are false positives on all the filters. Then the ratio $\frac{n_t}{n}$ is compared with $\frac{1}{2^{r\cdot t}}$ to make sure that the two values are similar. If that is the case it validates the conjecture and thus the ability of an attacker to eliminate false positives up to the desired probability.

In this first experiment, we fix $s = 64K$ [3], $n = 128M$, $r = 6$, $\delta = 1, 2, 3$ and $t = 5$ and elements are taken from the universe of 64 bit elements. The average false positive probability after each iteration for 1000 runs is shown in Figure 9, and compared to the theoretical estimate given by $\frac{1}{2^{r\cdot t}}$. Observe that when $\delta = 1$ or 2, the number of false positives is reduced with each iteration, but it is still above the theoretical estimate for independent filters. Because the implementation uses three equal-size tables per xor filter,

2. The code used in our experiments is available at https://github.com/amacian/PrivacyXOR
3. In this section, K refers to 2^{10} and M to 2^{20}.

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/

This article has been accepted for publication in IEEE Transactions on Dependable and Secure Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2023.3298967

targeting 1.23 total slots per element, the table size may not increase when the increment is below three. Indeed, the number of false positives in common is smaller for $\delta = 2$ than for $\delta = 1$ as the change in size is more likely to occur. Finally, for $\delta = 3$, it can be observed that the simulated results match the theoretical estimates based on the conjecture of the filters being independent. This shows that a typical xor filter implementation is susceptible to substantial information leakage by adding just a few elements to the original set.

The same experiment was done for $s = 1M$ to see if the filter size has any impact. The results are shown in Figure 10. It can be seen that when $\delta = 1, 2$ the results are slightly different due to the different rounding values for the table size, however for $\delta = 3$ they are similar and again confirm the independence of the filter instances.

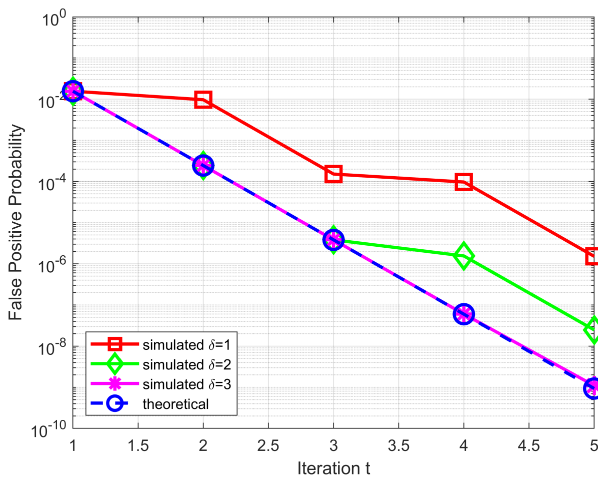

Fig. 9. False positive probability after each iteration for an xor filter with $s = 64K, n = 128M, r = 6, t = 5$ over 1000 runs when $\delta = 1, 2, 3$.

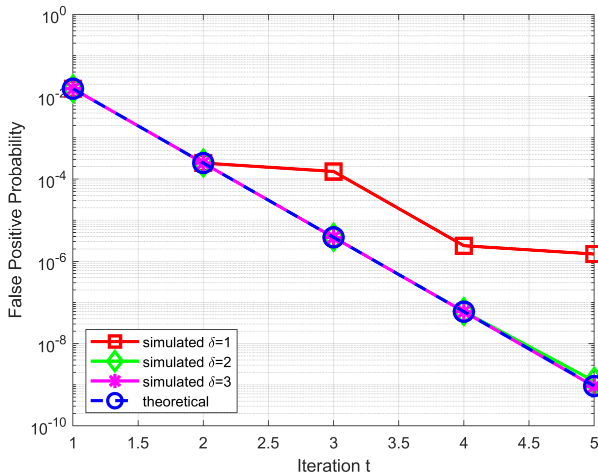

Fig. 10. False positive probability after each iteration for an xor filter with $s = 1M, n = 128M, r = 6, t = 5$ over 1000 runs when $\delta = 1, 2, 3$.

The second experiment uses different larger values for δ to see if they have any effect on the results. In more detail, the values for δ are selected randomly in the range of 4 to 512 for each run and the values of n_3 are logged. The values obtained are shown in Figure 11. It can be observed that the value of δ does not influence the results. Therefore, the independence conjecture seems to be valid regardless of the value of δ as long at it is three or larger. As in the first experiment, the simulations were repeated with $s = 1M$ and the results are shown in Figure 12. It can be seen that they are similar and confirm that the filter size has no effect on the filter instance independence.

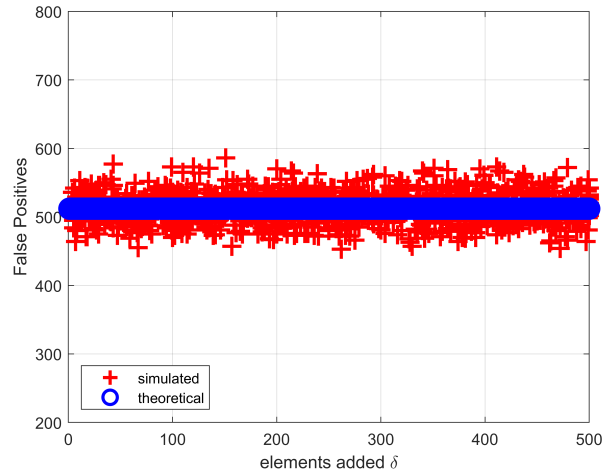

Fig. 11. Number of false positives n_3 after the third iteration for an xor filter with $s = 64K, n = 128M, r = 6$ and random values of δ elements added.

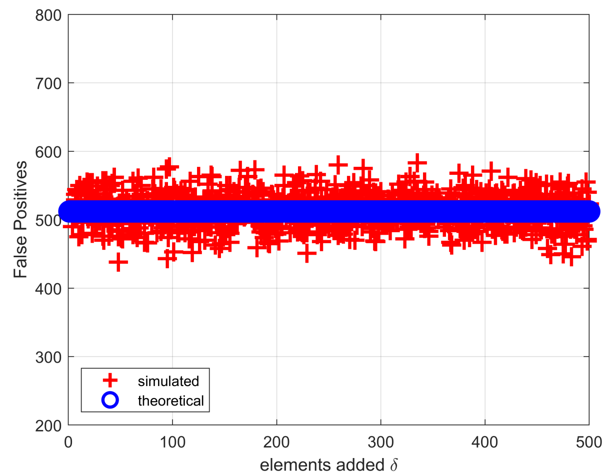

Fig. 12. Number of false positives n_3 after the third iteration for an xor filter with $s = 1M, n = 128M, r = 6$ and random values of δ elements added.

As changing the filter size can compromise the privacy of the elements stored in the filter, one of the potential solutions could be to use initially a larger filter so that additional insertions can be supported without changing the filter size. For example, the size can be dimension for $s + \delta$ elements so that after the insertion of the initial s elements

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/

This article has been accepted for publication in IEEE Transactions on Dependable and Secure Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2023.3298967

another δ elements can be inserted without modifying the filter size. Therefore, we finally consider the case where the filter size remains the same and a fraction of the elements are randomly selected among those stored in the filter and replaced by new elements also randomly chosen. As in previous experiments we set $s = 64K$, $n = 128M$, $r = 6$, $t = 5$ and the fraction of replaced elements to be approximately 0.01%, 0.1%, 1% and 10%. The results are summarized in Figure 13 showing the false positive probability over 1000 runs. The theoretical estimates for independent filters ($\frac{1}{2^{r \cdot t}}$) and for residual false positives ($2^{1-r}/3000$) are also shown in Figure 13. Observe that querying multiple instances does reduce false positives but the reduction deviates from that of independent filters and saturates around the residual false positives as expected. After reaching that value, querying additional instances has limited benefit as false positives are reduced only when one of the δ newly-inserted elements maps to one of the remaining false positives. Therefore, the reduction is larger for larger values of δ as can be seen in Figure 13. This means that in this case, the effectiveness of the attack is reduced but still many false positives can be excluded by querying several filter instances. The same experiment was repeated with $s = 1M$ and the results are shown in Figure 14. It can be observed that again the results are similar to those of the initial simulations with $s = 64K$.

Going back to compare Figure 13 with Figure 9, it is clear that changing the underlying filter size and thus range of the reduced hash does not have the backstop of residual false positives, as these also change when the size changes.

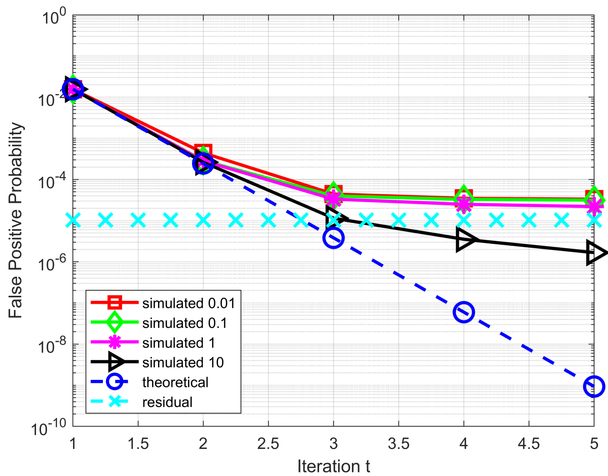

Fig. 13. False positive probability after each iteration for an xor filter with $s = 64K, n = 128M, r = 6, t = 5$ over 1000 runs when replacing a percentage of the element stored in the filter.

5 Case Studies

The ability of a persistent attacker to infer if an arbitrary element has been inserted in an xor filter depends on several factors: the false positive probability of the xor filter (which depends on the number of fingerprint bits r), the number of instances that the attacker can access (t), the total independent hash bits used from the hash functions (b) and whether the xor filter uses oversizing to protect against the

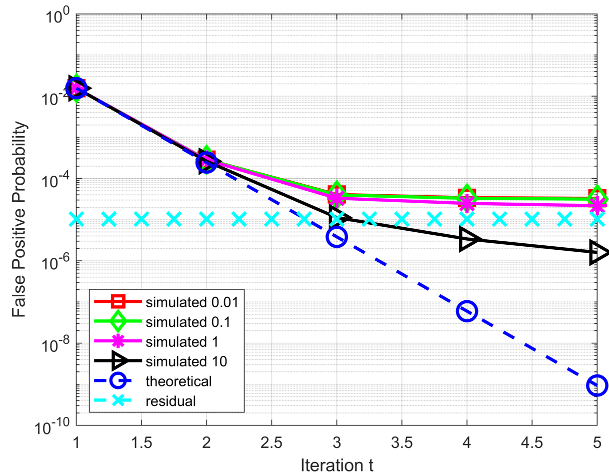

Fig. 14. False positive probability after each iteration for an xor filter with $s = 1M, n = 128M, r = 6, t = 5$ over 1000 runs when replacing a percentage of the element stored in the filter.

attack. In more detail, the false positive probability that an attacker can get when querying t filter instances is approximately $\frac{1}{2^{r \cdot t}}$ when no oversizing is used and $2^{1-r}/3000$ when sufficient oversizing is used so that dynamic insertions and removals can be supported with the same filter size.

Considering a universe U that has u elements and an attacker that wants to infer whether each element in the universe has been inserted in the filter, the expected number of false positives would be approximately:

$$FPs_{novs} = \frac{u}{2^{r \cdot t}} + \frac{u}{2^b} \quad (2)$$

without oversizing. The right term comes from our hashing bound.

For oversizing the number would be limited by the residual false positives:

$$FPs_{ovs} = \frac{u}{2^{r-1} \cdot 3000} \quad (3)$$

In the first case, t can be adjusted to achieve a value that is almost negligible, within the hashing limits. Instead in the second, once the limit is quickly reached, increasing t has little effect. In comparison, a space-optimized cuckoo filter of consistent size will have false positives roughly $u/2^{r-2}$ regardless of t.

To better understand the number of false positives in practical settings, we can consider a few case studies:

- Filtering of IPv4 addresses: the universe is formed by the $2^{32} \approx 4.3 \cdot 10^9$ possible IPv4 addresses.
- Filtering of Ethernet MAC addresses: the universe is formed by the $2^{48} \approx 2.8 \cdot 10^{14}$ possible MAC addresses.
- Filtering of compromised passwords: we assume that the passwords are represented by 160 bit SHA-1 hashes[4] ($\approx 1.5 \cdot 10^{48}$).
- Filtering of names: the names are encoded in text and possibly, the number of names is very large.

4. A list containing the SHA-1 of millions of passwords is available at https://haveibeenpwned.com/Passwords

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/

This article has been accepted for publication in IEEE Transactions on Dependable and Secure Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2023.3298967

However, the number of names that correspond to a person can be bounded to be the number of persons. Considering as an example the names of Spanish citizens, the universe of names would be approximately 50 million.

The expected number of false positives for each case study when $r = 8$ and for different values of t are summarized in Table 2. It can be observed that when oversizing is used, there will be false positives in all four case studies and thus the presence of a given element x in the filter cannot be fully asserted. Instead, when no oversizing is used, by selecting the right value for t, the number of false positives can be made practically zero[5] so that the presence of an element can be asserted with almost certainty. This occurs for all the case studies except the passwords when $t = 8$, so after checking eight instances. Instead for the passwords, $t = 24$ instances are needed. However, even this number is low and could be easily obtained by an attacker over time. Note that when the universe has a manageable size so that the attack can be done for all elements in the universe, the attacker can actually infer all the elements stored in the filter. Therefore, to protect privacy, xor filter implementations should use some oversizing to ensure that an attacker does not have access to filter instances that are fully independent.

Let us now assume that the filter is using oversizing. Table 3 shows the predicted number of false positives when using fingerprints of different sizes. It can be seen that for the Names case study, values below one are obtained in some cases. Therefore, even when oversizing is used, an attacker may be able to infer with some confidence the exact stable original set of the filter if r is large and the universe has a moderate size. As shown in the table, the privacy improves with an equivalence class-based filter such as a cuckoo filter.

As for the cost of oversizing, it would largely depend on the variability of the size of the stored set. For example, in a accept or deny list of IPv4 addresses we would expect a small variability from one day to the next so probably allocating a 10% oversize would ensure that most instances have the same size. The same reasoning applies to accept or deny lists of person names or list of compromised passwords. Therefore, in many applications oversizing would reduce the attacker's ability to infer whether an element is stored in the filter while incurring in a small memory overhead.

Finally, it should also be noted that in some applications the number of elements stored in the filter may grow considerably over time and thus periodic reconstructions would be needed for all filter types. Although this may help in reducing Denial of Service attacks produced by an adversary taking advantage of false positives [41] (as these change after reconstruction), it also creates a potential vulnerability as it would mean that several independent instances of the filter will be used. Therefore, alternative techniques to protect privacy will be needed.

5. Although the standard implementation uses $b = 64$ with some hash reuse tricks, this argument assumes a sufficient number of independent hash bits is used, also shown in Table 2.

6 Conclusion and Future Work

This paper has examined the privacy of approximate membership check filters with a focus on xor filters and multi-versioned attacks, unlike previous works. A series of related filters is common in many applications in which the set of elements for filtering changes slowly over time. Our analysis shows that cuckoo and quotient filters have some ideal properties for minimizing information leakage across versions of a filter, and Bloom filters are close to that ideal. For xor filters, however, access to several instances can often reduce the probability of false positives dramatically to a point where an attacker can infer the presence of an element with almost no error. This analysis has been validated by simulation under different configurations. Therefore, xor filters should be used with caution in applications for which privacy is an issue.

To mitigate some xor privacy issues, we demonstrate using oversizing in the filter construction so that additional elements can be accommodated without changing the filter size. This reduces change in the false positive set from filter to filter and, thus, also reduces the advantage of an attacker seeing many related filters. Despite the improvements, however, xor filters are substantially worse for privacy across many related filters than more traditional alternatives.

The only known natural privacy advantage of an xor filter is a consistent false positive rate even when the filter is under-populated. Especially in Bloom filters but also cuckoo and quotient, an attacker with access to a sparsely populated filter can have extra confidence in the identity of the small number of elements used to populate the filter. At least without access to the underlying filter (only query results), the xor filter is not vulnerable in the same way.

Future work is possible in several areas. New attack scenarios could include filters growing or shrinking in size more substantially, or access to the filter contents. Best practices and more protection schemes should be explored for the various scenarios and kinds of filters. More filter designs such as ribbon filters [32] should also be studied.

7 Acknowledgements

This work was partially supported by the ACHILLES project PID2019-104207RB-I00 and the ENTRUDIT project TED2021-130118B-I00 funded by the Spanish Agencia Estatal de Investigacion (AEI).

References

[1] M. Langheinrich, "The golden age of privacy?" *IEEE Pervasive Computing*, vol. 17, no. 4, pp. 4–8, 2018.
[2] N. Kshetri and J. Voas, "Thoughts on general data protection regulation and online human surveillance," *Computer*, vol. 53, no. 1, pp. 86–90, 2020.
[3] G. Cormode, "Data sketching," *Commun. ACM*, vol. 60, no. 9, p. 48–55, Aug. 2017. [Online]. Available: https://doi.org/10.1145/3080008
[4] L. Luo, D. Guo, R. T. B. Ma, O. Rottenstreich, and X. Luo, "Optimizing Bloom filter: Challenges, solutions, and comparisons," *IEEE Communications Surveys and Tutorials*, vol. 21, no. 2, pp. 1912–1949, 2019.

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/

This article has been accepted for publication in IEEE Transactions on Dependable and Secure Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2023.3298967

TABLE 2
Predicted number of universe elements that are false positive in all filters, $r = 8$ xor filter

	No size change	Each filter distinct size (no oversizing)					
Case Study	$\mathbf{t \to \infty}$	$\mathbf{t = 1}$	$\mathbf{t = 2}$	$\mathbf{t = 4}$	$\mathbf{t = 8}$	$\mathbf{t = 16}$	$\mathbf{t = 24}$
IPv4	11142	$16.7 \cdot 10^6$	65536	1	$2.3 \cdot 10^{-10}$	$1.3 \cdot 10^{-29}$	$6.8 \cdot 10^{-49}$
MAC	$7.3 \cdot 10^8$	$1.1 \cdot 10^{12}$	$4.3 \cdot 10^9$	65536	$1.5 \cdot 10^{-5}$	$8.3 \cdot 10^{-25}$	$4.5 \cdot 10^{-44}$
Passwords	$3.8 \cdot 10^{42}$	$5.7 \cdot 10^{45}$	$2.2 \cdot 10^{43}$	$3.4 \cdot 10^{38}$	$7.9 \cdot 10^{28}$	$4.3 \cdot 10^9$	$2.3 \cdot 10^{-10}$
Names	129.7	195310	762.9	0.01	$2.7 \cdot 10^{-12}$	$1.5 \cdot 10^{-31}$	$8.0 \cdot 10^{-51}$
Assumed hash bits			$b > 16$	$b > 32$	$b > 64$	$b > 128$	$b > 192$

TABLE 3
Predicted number of false positives in all filters, many ($t \to \infty$) filters with no size change (successful oversizing), xor and space-optimized cuckoo

Case study	Filter	$\mathbf{r = 8}$	$\mathbf{r = 12}$	$\mathbf{r = 16}$	$\mathbf{r = 20}$
IPv4	xor	11185	699	43.7	2.73
	cuckoo	$6.7 \cdot 10^7$	$4.2 \cdot 10^6$	$2.6 \cdot 10^5$	16384
MAC	xor	$7.3 \cdot 10^8$	$4.6 \cdot 10^7$	$2.9 \cdot 10^6$	$1.8 \cdot 10^5$
	cuckoo	$4.4 \cdot 10^{12}$	$2.8 \cdot 10^{11}$	$1.7 \cdot 10^{10}$	$1.1 \cdot 10^9$
Passwords	xor	$3.8 \cdot 10^{42}$	$2.4 \cdot 10^{41}$	$1.5 \cdot 10^{40}$	$9.3 \cdot 10^{38}$
	cuckoo	$2.3 \cdot 10^{46}$	$1.4 \cdot 10^{45}$	$8.9 \cdot 10^{43}$	$5.6 \cdot 10^{42}$
Names	xor	130	8.1	0.51	0.03
	cuckoo	$7.8 \cdot 10^5$	48828	3052	191

[5] R. van Rijswijk-Deij, G. Rijnders, M. Bomhoff, and L. Allodi, "Privacy-conscious threat intelligence using dnsbloom," in *IFIP/IEEE International Symposium on Integrated Network Management, IM 2019, Washington, DC, USA, April 09-11, 2019*, J. Betser, C. J. Fung, A. Clemm, J. François, and S. Ata, Eds. IFIP, 2019, pp. 98–106. [Online]. Available: http://dl.ifip.org/db/conf/im/im2019/189282.pdf

[6] W. Xue, D. Vatsalan, W. Hu, and A. Seneviratne, "Sequence data matching and beyond: New privacy-preserving primitives based on bloom filters," *IEEE Transactions on Information Forensics and Security*, vol. 15, pp. 2973–2987, 2020.

[7] R. Schnell, T. Bachteler, and J. Reiher, "Privacy-preserving record linkage using bloom filters," *BMC Medical Informatics and Decision Making*, vol. 9, no. 1, pp. 1–11, 2009.

[8] S. M. Bellovin and W. R. Cheswick, "Privacy-enhanced searches using encrypted Bloom filters," *IACR Cryptol. ePrint Arch.*, p. 22, 2004.

[9] S. Shomaji, P. Ghosh, F. Ganji, D. Woodard, and D. Forte, "An analysis of enrollment and query attacks on hierarchical bloom filter-based biometric systems," *IEEE Transactions on Information Forensics and Security*, vol. 16, pp. 5294–5309, 2021.

[10] D. Desfontaines, A. Lochbihler, and D. Basin, "Cardinality estimators do not preserve privacy," *Proceedings on Privacy Enhancing Technologies*, vol. 2019, no. 2, pp. 26–46, 2019. [Online]. Available: https://doi.org/10.2478/popets-2019-0018

[11] A. Gervais, S. Capkun, G. O. Karame, and D. Gruber, "On the privacy provisions of Bloom filters in lightweight bitcoin clients," in *Annual Computer Security Applications Conference*, 2014.

[12] G. Bianchi, L. Bracciale, and P. Loreti, ""better than nothing" privacy with Bloom filters: To what extent?" in *International Conference on Privacy in Statistical Databases*, 2012.

[13] P. Reviriego, A. Sánchez-Macian, S. Walzer, E. Merino-Gómez, S. Liu, and F. Lombardi, "On the privacy of counting bloom filters," *IEEE Transactions on Dependable and Secure Computing*, vol. 20, no. 2, pp. 1488–1499, 2023.

[14] P. Reviriego, A. Sánchez-Macián, E. Merino-Gómez, O. Rottenstreich, S. Liu, and F. Lombardi, "Attacking the Privacy of Approximate Membership Check Filters by Positive Concentration," *IEEE Transactions on Computers*, vol. 72, no. 5, pp. 1409–1419, 2023.

[15] M. Franke, Z. Sehili, F. Rohde, and E. Rahm, "Evaluation of hardening techniques for privacy-preserving record linkage." in *EDBT*, 2021, pp. 289–300.

[16] B. Bloom, "Space/time tradeoffs in hash coding with allowable errors," *Communications of the ACM*, vol. 13, no. 7, 1970.

[17] O. Rottenstreich, Y. Kanizo, and I. Keslassy, "The variable-increment counting Bloom filter," *IEEE/ACM Trans. Netw.*, vol. 22, no. 4, pp. 1092–1105, 2014.

[18] S. Pontarelli, P. Reviriego, and J. A. Maestro, "Improving counting Bloom filter performance with fingerprints," *Information Processing Letters*, vol. 116, no. 4, pp. 304–309, 2016.

[19] B. Fan, D. G. Andersen, M. Kaminsky, and M. D. Mitzenmacher, "Cuckoo filter: Practically better than Bloom," in *ACM CoNEXT*, 2014.

[20] A. Pagh, R. Pagh, and S. S. Rao, "An optimal bloom filter replacement," in *Proceedings of the Sixteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, ser. SODA '05. USA: Society for Industrial and Applied Mathematics, 2005, p. 823–829.

[21] M. A. Bender, M. Farach-Colton, R. Johnson, R. Kraner, B. C. Kuszmaul, D. Medjedovic, P. Montes, P. Shetty, R. P. Spillane, and E. Zadok, "Don't thrash: How to cache your hash on flash," *Proc. VLDB Endow.*, vol. 5, no. 11, p. 1627–1637, jul 2012. [Online]. Available: https://doi.org/10.14778/2350229.2350275

[22] T. M. Graf and D. Lemire, "Xor filters: Faster and smaller than bloom and cuckoo filters," *ACM J. Exp. Algorithmics*, vol. 25, Mar. 2020. [Online]. Available: https://doi.org/10.1145/3376122

[23] M. Mitzenmacher, S. Pontarelli, and P. Reviriego, "Adaptive cuckoo filters," *ACM J. Exp. Algorithmics*, vol. 25, pp. 1–20, 2020.

[24] P. C. Dillinger and P. Manolios, "Fast and accurate bitstate verification for SPIN," in *Model Checking Software, 11th International SPIN Workshop, Barcelona, Spain, April 1-3, 2004, Proceedings*, ser. Lecture Notes in Computer Science, S. Graf and L. Mounier, Eds., vol. 2989. Springer, 2004, pp. 57–75. [Online]. Available: https://doi.org/10.1007/978-3-540-24732-6_5

[25] M. Dietzfelbinger and S. Walzer, "Efficient gauss elimination for near-quadratic matrices with one short random block per row, with applications," in *27th Annual European Symposium on Algorithms, ESA 2019, September 9-11, 2019, Munich/Garching, Germany*, ser. LIPIcs, M. A. Bender, O. Svensson, and G. Herman, Eds., vol. 144. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019, pp. 39:1–39:18. [Online]. Available: https://doi.org/10.4230/LIPIcs.ESA.2019.39

[26] S. Walzer, "Peeling close to the orientability threshold: Spatial coupling in hashing-based data structures," *CoRR*, vol. abs/2001.10500, 2020. [Online]. Available: https://arxiv.org/abs/2001.10500

[27] P. C. Dillinger, M. Farach-Colton, G. Tagliavini, and S. Walzer, "Optimal uncoordinated unique ids," in *Proceedings of the 42nd ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, PODS 2023, Seattle, WA, USA, June 18-23, 2023*, F. Geerts, H. Q. Ngo, and S. Sintos, Eds. ACM, 2023, pp. 221–230. [Online]. Available: https://doi.org/10.1145/3584372.3588674

[28] L. Fan, P. Cao, J. Almeida, and A. Z. Broder, "Summary Cache: A Scalable Wide-Area Web Cache Sharing Protocol," *IEEE/ACM Trans. Netw.*, vol. 8, no. 3, p. 281–293, jun 2000. [Online]. Available: https://doi.org/10.1109/90.851975

[29] P. C. Dillinger and P. P. Manolios, "Fast, all-purpose state storage," in *Model Checking Software*, C. S. Păsăreanu, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 12–31.

[30] B. Chazelle, J. Kilian, R. Rubinfeld, and A. Tal, "The bloomier filter: an efficient data structure for static support lookup tables," in

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/

This article has been accepted for publication in IEEE Transactions on Dependable and Secure Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2023.3298967

Proceedings of the fifteenth annual ACM-SIAM symposium on Discrete algorithms. Citeseer, 2004, pp. 30–39.

[31] M. Genuzio, G. Ottaviano, and S. Vigna, "Fast scalable construction of (minimal perfect hash) functions," in *Experimental Algorithms - 15th International Symposium, SEA 2016, St. Petersburg, Russia, June 5-8, 2016, Proceedings*, ser. Lecture Notes in Computer Science, A. V. Goldberg and A. S. Kulikov, Eds., vol. 9685. Springer, 2016, pp. 339–352. [Online]. Available: https://doi.org/10.1007/978-3-319-38851-9_23

[32] P. C. Dillinger, L. Hübschle-Schneider, P. Sanders, and S. Walzer, "Fast succinct retrieval and approximate membership using ribbon," in *20th International Symposium on Experimental Algorithms, SEA 2022, July 25-27, 2022, Heidelberg, Germany*, ser. LIPIcs, C. Schulz and B. Uçar, Eds., vol. 233. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022, pp. 4:1–4:20. [Online]. Available: https://doi.org/10.4230/LIPIcs.SEA.2022.4

[33] Y.-H. Feng, N.-F. Huang, and C.-H. Chen, "An efficient caching mechanism for network-based URL filtering by multi-level counting bloom filters," in *2011 IEEE International Conference on Communications (ICC)*. IEEE, jun 2011. [Online]. Available: https://doi.org/10.1109%2Ficc.2011.5963090

[34] F. Soldo, A. Markopoulou, and K. Argyraki, "Optimal filtering of source address prefixes: Models and algorithms," 05 2009, pp. 2446 – 2454.

[35] P. Reviriego, J. Martínez, D. Larrabeiti, and S. Pontarelli, "Cuckoo filters and bloom filters: Comparison and application to packet classification," *IEEE Transactions on Network and Service Management*, vol. 17, no. 4, pp. 2690–2701, 2020.

[36] D. Lemire, "Fast random integer generation in an interval," *ACM Trans. Model. Comput. Simul.*, vol. 29, no. 1, jan 2019. [Online]. Available: https://doi.org/10.1145/3230636

[37] J. G. Cleary, "Compact hash tables using bidirectional linear probing," *IEEE Trans. Computers*, vol. 33, no. 9, pp. 828–834, 1984. [Online]. Available: https://doi.org/10.1109/TC.1984.1676499

[38] P. C. Dillinger and P. Manolios, "Bloom filters in probabilistic verification," in *Formal Methods in Computer-Aided Design, 5th International Conference, FMCAD 2004, Austin, Texas, USA, November 15-17, 2004, Proceedings*, ser. Lecture Notes in Computer Science, A. J. Hu and A. K. Martin, Eds., vol. 3312. Springer, 2004, pp. 367–381. [Online]. Available: https://doi.org/10.1007/978-3-540-30494-4_26

[39] F. Putze, P. Sanders, and J. Singler, "Cache-, hash-, and space-efficient bloom filters," *ACM J. Exp. Algorithmics*, vol. 14, 2009. [Online]. Available: https://doi.org/10.1145/1498698.1594230

[40] M. Mitzenmacher, "Compressed bloom filters," *IEEE/ACM Trans. Netw.*, vol. 10, no. 5, pp. 604–612, 2002. [Online]. Available: https://doi.org/10.1109/TNET.2002.803864

[41] T. Gerbet, A. Kumar, and C. Lauradoux, "The Power of Evil Choices in Bloom Filters," in *2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, 2015, pp. 101–112.

Pedro Reviriego received the M.Sc. and Ph.D. degrees in telecommunications engineering from the Technical University of Madrid, Madrid, Spain, in 1994 and 1997, respectively. From 1997 to 2000, he was an Engineer with Teldat, Madrid, working on router implementation. In 2000, he joined Massana to work on the development of 1000BASE-T transceivers. From 2004 to 2007, he was a Distinguished Member of Technical Staff with the LSI Corporation, working on the development of Ethernet transceivers. From 2007 to 2018 he was with Nebrija University, from 2018 to 2022 with Universidad Carlos III de Madrid. He is currently with Universidad Politécnica de Madrid working on probabilistic data structures, high speed packet processing and machine learning.

Alfonso Sánchez-Macián received the M.Sc. and Ph.D. degrees in telecommunications engineering from the Universidad Politécnica de Madrid, Madrid, Spain, in 2000 and 2007, respectively. He has worked as a Lecturer and a Researcher with several universities, such as the Universidad Politécnica de Madrid; the IT Innovation Centre, University of Southampton, Southampton, U.K.; the Universidad Antonio de Nebrija, Madrid, and the Universidad Carlos III de Madrid, where he currently works. His current research interests include nonfunctional properties of the systems, including security, fault tolerance, and reliability.

Peter C. Dillinger has earned M.Sc. from Georgia Tech (2003) and Ph.D. from Northeastern University (2010) in Computer Science, focusing on software verification and supporting data structures and algorithms. He worked on bug-finding static analysis for Coverity, acquired by Synopsys, and currently works on a reliable and efficient storage engine RocksDB at Meta (Facebook), focusing on data structures and algorithms.

Stefan Walzer studied computer science and mathematics at the Karlsruhe Institute of Technology and earned a doctorate degree in computer science at Technische Universität Ilmenau in 2020. His dissertation dealt with cuckoo hashing, retrieval data structures and xor filters from a mathematical perspective. He continues to be interested in these topics and is currently working at the University of Cologne with funding from the German Research Foundation (DFG).

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/