XOR-Satisfiability Set Membership Filters

Sean A. Weaver[0000−0001−9883−4473], Hannah J. Roberts, and Michael J. Smith

Information Assurance Research Group
U.S. National Security Agency
9800 Savage Rd., Suite 6845, Ft. George G. Meade, MD 20755
saweave@tycho.ncsc.mil

Abstract. Set membership filters are used as a primary test for whether large sets contain given elements. The most common such filter is the Bloom filter [6]. Most pertinent to this article is the recently introduced Satisfiability (SAT) filter [31]. This article proposes the *XOR-Satisfiability filter*, a variant of the SAT filter based on random k-XORSAT. Experimental results show that this new filter can be more than 99% efficient (i.e., achieve the information-theoretic limit) while also having a query speed comparable to the standard Bloom filter, making it practical for use with very large data sets.

1 Introduction

To support timely computation on large sets, and in cases where being certain is not necessary, a quick, probabilistic test of a *set membership filter* is often used. A set membership filter is constructed from a set and queried with elements from the corresponding domain. Being probabilistic, the filter will return either *Maybe* or *No*. That is, the filter can return false positives, but never false negatives. The most well-known set membership filter is the Bloom filter [6]. Though many other set membership filters have been proposed, the most important to this work is the SAT filter [31].

A *SAT filter* is a set membership filter constructed using techniques based on SAT [5]. In [31], the authors describe the process of building a SAT filter as follows. First, each element in a set of interest is translated into a CNF clause (disjunction of literals). Next, every clause is logically conjoined into a CNF formula. Finally, solutions to the resulting formula are found using a SAT solver. These solutions constitute a SAT filter. To query a SAT filter, an element is translated into a clause (using the same method as during filter building) and if the clause is satisfied by all of the stored solutions, the element may be in the original set, otherwise it is definitely not in the original set. Parameters for tailoring certain aspects of the SAT filter such as false positive rate, query speed, and amount of long term storage are described in [31].

This article describes a new, practical variant of the generic SAT filter where clauses are considered to be the XORs of Boolean variables, rather than the traditional inclusive OR (disjunction) of literals. This approach (mentioned as possible future research in [31]) offers many advantages over a disjunction-based

SAT filter such as practically near perfect filter efficiency [30], faster build and query times, and support for metadata storage and retrieval.

In terms of related work, there are other filter constructions that attempt to achieve high efficiency (e.g. via compression) (e.x., see [7–9,16,24]). Most similar to the XORSAT filter construction introduced here are Matrix filters [13,26]. Insofar as XORSAT equations are equivalent to linear equations over $GF(2)$, there are two obvious (and independent) ways to generalize such a linear system: either by considering equations over larger fields like $GF(2^s)$ (Matrix filters), or remaining over $GF(2)$ and working with s right-hand sides (XORSAT filters). In both constructions, the solutions can be used to store probabilistic membership in sets, as well as values corresponding to keys, but the XORSAT filter construction is motivated by some clear computational advantages.

First, Matrix filters require a hash function that yields elements over $GF(2^s)^n$, whereas hash functions for XORSAT filters yield elements over $GF(2)^n$ — an s-fold improvement in the data required. Also, Matrix filters require arithmetic over $GF(2^s)$, whereas XORSAT filters work entirely over $GF(2)$ and as such are more naturally suited to highly-optimized implementations; all computations devolve to simple and fast word operations (like AND and XOR) and bit-parity computations which are typically supported on modern computers. This article also proposes some simple and more practical methods for bucketing and handling sparse variants, which likewise correspond to efficiency and performance improvements.

2 XORSAT Filters

This section briefly describes XORSAT and the XORSAT filter.

2.1 XORSAT

Construction and query of an XORSAT filter rely heavily on properties of random k-XORSAT, a variant of SAT where formulas are expressed as conjunctions of random XOR clauses, i.e. the exclusive OR of Boolean variables.

Definition 1. *An XOR clause is an expression of the form*

$$v_{i_1} \oplus \ldots \oplus v_{i_{k_i}} \equiv b_i,$$

where the symbol $\oplus$ represents XOR, the symbol $\equiv$ represents logical equivalence, each v_i is a Boolean variable and each b_i (right-hand side) is a constant, either 0 *(for False) or* 1 *(for True).*

Definition 2. *A width k XOR clause has exactly k distinct variables.*

Definition 3. *A random k-XORSAT instance is a set of XOR clauses drawn uniformly, independently, and with replacement from the set of all width k XOR clauses* [20].

As with random k-SAT [1], a random k-XORSAT instance is likely to be satisfiable if its clauses-to-variables ratio is less than a certain threshold α_k, and likely to be unsatisfiable if greater than α_k [25]. Experimental results have established approximate values of α_k for small values of k, though it asymptotically approaches 1. Experimental values are given next and are reproduced from [11, 12].

Table 1. Various α_k values for random k-XORSAT

k	2	3	4	5	6	7
α_k	0.5	0.917935	0.976770	0.992438	0.997379	0.999063

Polynomial time algorithms exist for reducing random k-XORSAT instances into reduced row echelon form. For example, Gaussian elimination can solve such instances in $\mathcal{O}(n^3)$ steps and the 'Method of Four Russians' [4, 29] in $\mathcal{O}(\frac{n^3}{\log_2 n})$. Once in this reduced form, collecting random solutions (kernel vectors) is trivial — assign random values to all of the free variables (those in the identity submatrix), and backsolve for the dependent ones.

2.2 XORSAT Filter Construction

This section presents the basic XORSAT filter construction. Later sections provide enhancements which enable such filters to be used in practice. The XORSAT filter is built and queried in a manner very similar to the SAT filter. Provided below are updated algorithms for construction and query, analogous to those in [31], where deeper discussion on how to construct and query SAT filters can be found.

Building an XORSAT Filter Being a variant of the the SAT filter, the XORSAT filter has similar properties. Building an XORSAT filter for a data set $Y \subseteq D$ (where D is a domain) is one-time work. The XORSAT filter is an offline filter, so, once built, it is not able to be updated. To build an XORSAT filter, all elements $y \in Y$ are transformed into width k XOR clauses that, when conjoined, constitute a random k-XORSAT instance. If the instance is unsatisfiable, a filter cannot be constructed for the given data set and parameters. Otherwise, a solution for that instance and acts as a filter for Y.

Algorithm 1 shows how to transform an element $e \in D$ into a width k XOR clause using a set of hash functions. Algorithm 2 shows how to build an XORSAT filter from a given set $Y \subseteq D$.

Querying a SAT Filter Querying an XORSAT filter with an element $x \in D$ is very similar to querying a SAT filter. First, x is transformed into a k width XOR

Algorithm 1 ElementToXorClause($e \in D, n, k, h_0, \ldots, h_{k-1}, h_b$)
e is the element used to generate an XOR clause
n is the number of variables per XORSAT instance
k is the number of variables per XOR clause
$h_0, \ldots, h_{k-1}$ are functions that map elements of D to $[0, n)$
h_b is a function that maps elements of D to $[0, 1]$

```
nonce := 0
repeat
    V := {}
    for i := 0 to k − 1 do
        v := h_i(e, nonce), hash e to generate variable v
        V := V ∪ {v}, add v to the XOR clause
    end for
    nonce := nonce + 1
until all variables of V are distinct
b := h_b(e), hash e to generate the right-hand side
return (V, b)
```

Algorithm 2 BuildXorSatFilter($Y \subseteq D, n, k, h_0, \ldots, h_{k-1}, h_b$)
Y is the set used to build an XORSAT filter
n is the number of variables per XORSAT instance
k is the number of variables per XOR clause
$h_0, \ldots, h_{k-1}$ are functions that map elements of D to $[0, n)$
h_b is a function that maps elements of D to $[0, 1]$

```
X_Y := {}, the empty formula
for each element y ∈ Y do
    (V_y, b_y) := ElementToXorClause(y, n, k, h_0, ..., h_{k−1}, h_b)
    X_Y := X_Y ∪ {(V_y, b_y)}
end for
if the random k-XORSAT instance X_Y is unsatisfiable then
    return failure
else
    Let F_Y be a single solution to X_Y
    return F_Y
end if
```

clause. Then, if the clause is satisfied by the solution generated by Algorithm 2 for a set Y, x is *maybe* in Y, otherwise x is definitely not in Y. Algorithm 3 shows how to query an XORSAT filter.

Algorithm 3 QueryXorSatFilter($F_Y, x \in D, n, k, h_0, \ldots, h_{k-1}, h_b$)
x is the element used to query the XORSAT filter F_Y
n is the number of variables per XORSAT instance
k is the number of variables per XOR clause
$h_0, \ldots, h_{k-1}$ are functions that map elements of D to $[0, n)$
h_b is a function that maps elements of D to $[0, 1]$

```
(V_x, b_x) ≔ ElementToXorClause(x, n, k, h_0, ..., h_{k-1}, h_b)
for each variable v ∈ V_x do
    b_x ≔ b_x ⊕ F_Y(v)
end for
if b_x = 0 then
    return Maybe
end if
return No
```

2.3 False Positive Rate, Query Time, and Storing Multiple Solutions

The false positive rate of an XORSAT filter is the probability that the XOR clause generated by the query is satisfied by the stored solution. This is equal to the probability that a random width k XOR clause is satisfied by a random solution, i.e., $\frac{1}{2}$. As with the SAT filter, the false positive rate can be improved by either storing multiple solutions to multiple XORSAT instances or storing multiple *uncorrelated* solutions to a single XORSAT instance. For SAT filters, this second method is preferred because querying is much faster (only one clause needs to be built, so the hash functions are called fewer times), but the challenge of finding uncorrelated solutions to a single instance has yet to be overcome, though recent work seems promising [3, 14, 17, 21].

Fortunately, moving from SAT to XORSAT also moves past this difficulty. Since the XORSAT solving method used here, reduction to echelon form, is agnostic to the type of the elements in the matrix being reduced, s XORSAT instances can be encoded by treating the variables and right-hand side of each XOR clause as vectors of Booleans[1]. Then, the transformation to reduced row echelon form uses bitwise XOR on vectors (during row reduction) rather than Boolean XOR on single bits. Hence, s XORSAT instances can be solved in par-

[1] The intuition for this idea came from Bryan Jacobs' work on isomorphic k-SAT filters and work by Heule and van Maaren on parallelizing SAT solvers using bitwise operators [19].

allel, and just as fast as solving a single instance [2]. Also, since the XORSAT instances have random right-hand sides, the s solutions, one for each instance, will be uncorrelated.

The solutions are stored in the same manner as the SAT filter, that is, all s solution bits corresponding to a variable are stored together (the transpose of the array of solutions). If the solutions are stored this way, querying the s-wide filter is just as efficient as querying a filter created from a single instance. Moreover, the false positive rate is improved to $\frac{1}{2^s}$ because, during XORSAT filter query, s different right-hand side bits are generated from each element and each have to be satisfied by the corresponding solution.

2.4 Dictionaries

A small modification to the XORSAT filter construction can enable it to produce filters that also store and retrieve metadata d associated with each element y. To insert the tuple (y, d), a key-value pair, into the filter, append a bitwise representation of d, say r bits wide, to the right-hand side of the clause for y. Now every variable is treated as an $s + r$ wide vector of Booleans and the resulting instance is solved using word-level operations [3]. When querying, the first s bits of the right-hand side act as a check (to determine if the element passes the filter) and, if so (and not a false positive) the last r bits will take on the values of the bitwise representation of d.

On a purely practical note, the instances generated during build need not be entirely random k-XORSAT instances. By removing the check for duplicate variables, XOR clauses with less than k variables can be generated because duplicate variables in an XOR clause simply cancel out. In practice, this only slightly decreases efficiency (increases the size of the filter), but moderately decreases query time.

Algorithm 4 shows how to create a bit-packed sequence of XOR clauses, including support for dictionaries. Algorithm 5 shows how to query using the new Algorithm 4. To use these new algorithms to do purely filtering, set r to 0. For a pure dictionary, set s to 0. If both $r > 0$ and $s > 0$, the stored metadata will only be returned when an element passes the filter. If the element is a false positive, the returned metadata will be random.

2.5 Blocked XORSAT Filters

XORSAT filters suffer from the same size problem as SAT filters, namely, it is not practical to build filters for large sets. The reason being that the time it takes a modern solver to find a solution to an instance (with say millions of variables) is often too long for common applications. The natural way to overcome this with SAT filters is to increase the number of variables in the random k-SAT problem,

[2] As long as s is not greater than the native register size of the machine on which the solver is running.

[3] Adding an extra r bits of metadata means that the filter now has r more solutions.

Algorithm 4 ElementToXorClauses($e \in D, d, k, h_0, \ldots, h_{k-1}, h_b$)
e is the element used to generate s XOR clauses
n is the number of variables per XORSAT instance
d is data to be stored, a bit-vector in $[0, 2^r)$
k is the number of variables per XOR clause
$h_0, \ldots, h_{k-1}$ are functions that map elements of D to $[0, n)$
h_b is a function that maps elements of D to bit-vectors in $[0, 2^s)$

1: $V \coloneqq \{\}$
2: **for** $i \coloneqq 0$ **to** $k - 1$ **do**
3: $\quad v \coloneqq h_i(e)$, hash e to generate variable v
4: $\quad V \coloneqq V \cup \{v\}$, add v to the XOR clause
5: **end for**
6: $b \coloneqq h_b(e)$, hash e to generate the right-hand side
7: **return** $(V, b||d)$, append d to the right-hand side

Algorithm 5 QueryXorSatDictionary($F_Y, x \in D, n, k, s, r, h_0, ..., h_{k-1}, h_b$)
x is the element used to query the XORSAT filter F_Y
n is the number of variables per XORSAT instance
k is the number of variables per XOR clause
s is the number of solutions to be found
r is the number of bits of metadata stored with each element
$h_0, \ldots, h_{k-1}$ are functions that map elements of D to $[0, n)$
h_b is a function that maps elements of D to bit-vectors in $[0, 2^s)$

1: $(V_x, b_x = [b_0, \ldots, b_{s+r-1}]) \coloneqq$ ElementToXorClauses$(x, 0, k, h_0, \ldots, h_{k-1}, h_b)$
2: **for** each variable $v \in V_x$ **do**
3: $\quad b_x \coloneqq b_x \oplus F_Y(v)$
4: **end for**
5: **if** $[b_0, \ldots, b_{s-1}] = 0$ **then**
6: $\quad$ **return** (*Maybe*, $[b_s, \ldots, b_{s+r-1}]$)
7: **end if**
8: **return** *No*

decreasing efficiency, but also making the SAT problem easier by backing off of the k-SAT threshold [23]. This technique is not applicable to random k-XORSAT instances, that is, increasing the number of variables does not make significantly easier instances.

XORSAT (and SAT) filter build time can be decreased by first hashing elements into blocks (or buckets) and then building one filter for each block of elements, a process that is trivially parallelizable. This is a tailoring of a Blocked Bloom filter [22, 28] to SAT filters of any constraint variation. The number of blocks can be determined by the desired runtime of the build process; the more blocks the faster the build process. The issue here is that, given a decent random hash function, elements are distributed into blocks according to a Poisson distribution [15], that is, some blocks will likely have a few more elements than others. Hence, to store the solutions for each block, one also needs to store some information about the number of variables in each block so that they can be accessed during query. Depending on the technique used, this is roughly a small number of extra bits per block. Otherwise, the blocks can be forced to a uniform size by setting the number variables for each block to be the maximum number of variables needed to make the largest block satisfiable. Either way, the long-term storage of the filter has slightly increased, slightly decreasing efficiency at the benefit of a (potentially much) shorter build-time. So, here is one trade-off between build time and efficiency that can make SAT filters practical for large datasets. Also, blocking can increase query speed since, depending on block size, the k lookups will be relatively near each other in a computer's memory, giving the processor an opportunity to optimize the lookups. In fact, this is the original motivation of Blocked Bloom Filters; it's simply advantageous that the idea can also be used to drastically decrease the build time of SAT filters.

The next section provides the mathematics needed to choose appropriate parameters for the XORSAT filter construction.

3 Filter Efficiency

As introduced in [30], given a filter with false positive rate p, n bits of memory, and $m = |Y|$, the efficiency of the filter is

$$\mathcal{E} = \frac{-\log_2 p}{n/m} \ .$$

Efficiency is a measure of how well a filter uses the memory available to it. The higher the efficiency, the more information packed into a filter. A filter with a fixed size can only store so much information. Hence, efficiency has an upper-bound, i.e., the *information-theoretic limit*, namely

$$\mathcal{E} = 1 \ .$$

Since "$m/n = 1$ remains a *sharp* threshold for satisfiability of constrained[4] k-XORSAT for every $k \geq 3$" [25], the XORSAT filter construction, like the SAT

[4] A *constrained* model is one where every variable appears in at least two equations.

filter construction, theoretically achieves $\mathcal{E} = 1$. In other words, it is possible to build an XORSAT filter for a given data set and false positive rate that uses as little long-term storage as possible. XORSAT filter efficiency tends to 1 faster than that of SAT filters, and the corresponding satisfiability threshold is much sharper. This means that, since there are diminishing returns as k grows, a small k (five or six) can give near optimal efficiency (see Figure 1), and, unlike the SAT filter, these high efficiencies are able to be achieved in practice.

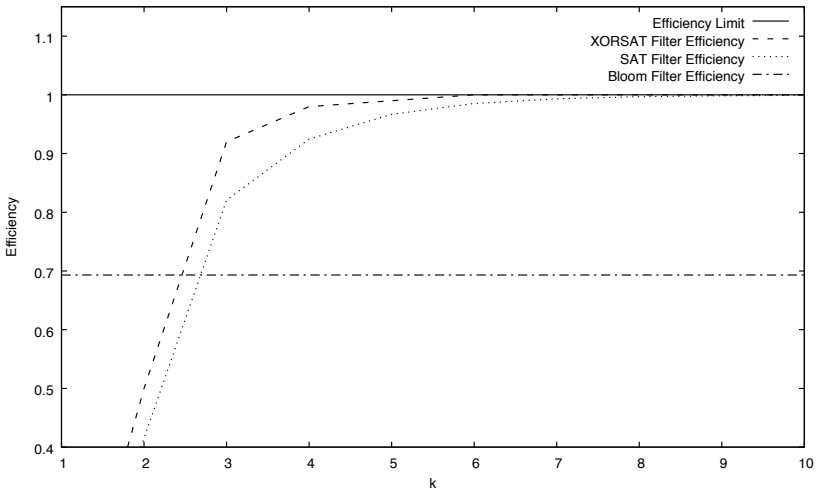

Fig. 1. Theoretically achievable XORSAT filter efficiency for various k.

4 XORSAT Filter Parameters

This section discusses the selection of parameters for XORSAT filters (see Table 2).

Table 2. XORSAT filter parameters

p	the false positive rate of an XORSAT filter
s	the number of XORSAT instances
r	the number of bits of metadata stored with each element
n	the number of variables per XORSAT instance
m	the number of XOR clauses per XORSAT instance
k	the number of variables per XOR clause

As with the SAT filter, a value for k should be selected first. A small k (five or six) is sufficient to achieve near perfect efficiency (see Figure 1). Larger k are undesirable as efficiency will not notably increase and query speed will significantly decrease.

The value of m is the number of elements being stored in the filter. Since the satisfiability threshold $\alpha_k = \frac{m}{n}$ is *sharp* for random k-XORSAT and tends quickly to 1, n should be set equal to, or slightly larger than m. For example, if $k = 3$, m should be roughly 91% of n (see Table 1 for precise calculations). Setting n much larger than m will cause a drop in efficiency without any advantage. This is not true for the SAT filter because random k-SAT problems become harder the closer they are to the satisfiability threshold [23], so, increasing n decreases build time. This is not the case with XORSAT filters and is the main reason they can practically achieve near perfect efficiency. Finally, a value for either s or p should be selected. These parameters determine the false positive rate $p = \frac{1}{2^s}$ and the amount of long-term storage (sn) of the filter.

To give an example set of parameters, an XORSAT filter for $m = 2^{16}$ elements with a false positive rate of $p \approx \frac{1}{2^7}$ needs $s = 7$ solutions to be stored and $n = 2^{16} + \epsilon$. Such an XORSAT filter, with $k = 6$, can be built and will use $sn \approx 460000$ bits of long-term storage, a 30% reduction over an optimal Bloom filter's long-term storage ≈ 660000 bits. See Section 6 for metrics on different size data sets, efficiencies, and query times.

5 Detailed Example

This section presents a detailed example of how to build and query an XORSAT filter, including details on how to use the filter to store and retrieve metadata. For the sole purpose of this example, let the set of interest be $Y =$ [("cat", 0), ("fish", 1), ("dog", 2)]. Here, Y is a list of three tuples where each tuple contains a word and an integer representing the tuple's index in the list.

5.1 Building the Filter

The first step in building an XORSAT filter for Y is to decide on parameters (see Section 4). For this example, let k, the number of variables per XOR clause, be three. Since there are three elements, m will be three. Let the number of variables per XORSAT instance be a number slightly larger than m to ensure the instances are satisfiable, say $n = 4$. Let p, the desired false positive rate, be $\frac{1}{2^3}$. This fixes s, the number of XORSAT instances, to three. Since there are three indices, only two bits of metadata, r, are needed to represent an index.

The next step is to create a list of hashes corresponding to each of the three words. This example will make use of the 32-bit xxHash algorithm [10]. Let the list of hashes be

$$\begin{aligned} H &= [\mathsf{xxHash}(\text{“cat”}), \mathsf{xxHash}(\text{“fish”}), \mathsf{xxHash}(\text{“dog”})] \\ &= [\texttt{0xb85c341a}, \texttt{0x87024bb7}, \texttt{0x3fa6d2df}] \ . \end{aligned}$$

Next, the hashes are used to generate XOR clauses, one per hash. For the purpose of this example a scheme needs to be devised that will transform a hash into an XOR clause. One simple method is to first treat the hash as a bit-vector, then split the vector into parts and let each part represent a new variable in the XOR clause. Here, let the hashes be split into 4-bit parts, as $2^4 > n$ and it will be easy to see the split (represented in hexadecimal). The list of split hashes is

$$\begin{aligned} SH = [&[\texttt{0xb}, \texttt{0x8}, \texttt{0x5}, \texttt{0xc}, \texttt{0x3}, \texttt{0x4}, \texttt{0x1}, \texttt{0xa}], \\ &[\texttt{0x8}, \texttt{0x7}, \texttt{0x0}, \texttt{0x2}, \texttt{0x4}, \texttt{0xb}, \texttt{0xb}, \texttt{0x7}], \\ &[\texttt{0x3}, \texttt{0xf}, \texttt{0xa}, \texttt{0x6}, \texttt{0xd}, \texttt{0x2}, \texttt{0xd}, \texttt{0xf}]] \ . \end{aligned}$$

The next step is to use the split hashes to create XOR clauses. This is done here by treating the groupings of 4 bits (under proper modulus) as variable indices and right-hand side of each clause. The variable indices and right-hand side for each clause would be

$$\begin{aligned} \mathcal{I}_{Y.0} &= [[SH_{00} (\text{mod } n), SH_{01} (\text{mod } n), SH_{02} (\text{mod } n), SH_{03} (\text{mod } 2^s)], \\ &\quad [SH_{10} (\text{mod } n), SH_{11} (\text{mod } n), SH_{12} (\text{mod } n), SH_{13} (\text{mod } 2^s)], \\ &\quad [SH_{20} (\text{mod } n), SH_{21} (\text{mod } n), SH_{22} (\text{mod } n), SH_{23} (\text{mod } 2^s)]] \\ &= [[\texttt{0xb} (\text{mod } 4), \texttt{0x8} (\text{mod } 4), \texttt{0x5} (\text{mod } 4), \texttt{0xc} (\text{mod } 8)], \\ &\quad [\texttt{0x8} (\text{mod } 4), \texttt{0x7} (\text{mod } 4), \texttt{0x0} (\text{mod } 4), \texttt{0x2} (\text{mod } 8)], \\ &\quad [\texttt{0x3} (\text{mod } 4), \texttt{0xf} (\text{mod } 4), \texttt{0xa} (\text{mod } 4), \texttt{0x6} (\text{mod } 8)]] \\ &= [[3, 0, 1, 4], \\ &\quad [0, 3, 0, 2], \\ &\quad [3, 3, 2, 6]] \ . \end{aligned}$$

In practice, these first few steps are the bottleneck in terms of query speed and need to be heavily optimized. The simple scheme presented here is purely for demonstration purposes. A more practical but complex scheme is given in Section 6. As well, this scheme does not guarantee width k XOR clauses are generated because duplicates may arise. However, duplicate variables in XOR clauses simply cancel each other out, so, for the purpose of this example, this simplified scheme is enough to demonstrate the main concepts. Also, for specific applications, duplicate detection and removal may be too computationally expensive to outweigh any benefit gained in efficiency.

The three XORSAT instances are encoded as follows:

$$\begin{aligned} \mathcal{X}_{Y.0} = [&x_3 \oplus x_0 \oplus x_1 \equiv [1, 0, 0], \\ &x_0 \oplus x_3 \oplus x_0 \equiv [0, 1, 0], \\ &x_3 \oplus x_3 \oplus x_2 \equiv [1, 1, 0]] \\ = [&x_0 \oplus x_1 \oplus x_3 \equiv [1, 0, 0], \\ &x_3 \equiv [0, 1, 0], \\ &x_2 \equiv [1, 1, 0]] \ . \end{aligned}$$

Next, append each element's two bits of metadata to the right-hand side of each corresponding XOR clause, creating $s + r = 5$ instances.

$$\begin{aligned}\mathcal{X}_Y = [&x_0 \oplus x_1 \oplus x_3 \equiv [1,0,0] \;||\; [0,0],\\ &x_3 \equiv [0,1,0] \;||\; [0,1],\\ &x_2 \equiv [1,1,0] \;||\; [1,0]]\\ = [&x_0 \oplus x_1 \oplus x_3 \equiv [1,0,0,0,0],\\ &x_3 \equiv [0,1,0,0,1],\\ &x_2 \equiv [1,1,0,1,0]]\ .\end{aligned}$$

The final steps are to solve and store $s + r = 5$ solutions, one for each of the XORSAT instances encoded by $\mathcal{X}_Y$. Though there are many different solutions to these instances, five such solutions are

$$\begin{aligned}S_Y = [&[x_0 = 1, x_1 = 0, x_2 = 1, x_3 = 0],\\ &[x_0 = 0, x_1 = 1, x_2 = 1, x_3 = 1],\\ &[x_0 = 0, x_1 = 0, x_2 = 0, x_3 = 0],\\ &[x_0 = 1, x_1 = 1, x_2 = 1, x_3 = 0],\\ &[x_0 = 1, x_1 = 0, x_2 = 0, x_3 = 1]]\ .\end{aligned}$$

The filter F_Y is the transpose of the solutions S_Y, along with those parameters necessary for proper querying, namely

$$\begin{aligned}F_Y = ([&[1,0,0,1,1],\\ &[0,1,0,1,0],\\ &[1,1,0,1,0],\\ &[0,1,0,0,1]],\\ &n = 3, k = 3, s = 3, r = 2)\ .\end{aligned}$$

The filter F_Y is now complete and the next part of the example will demonstrate how to query it.

5.2 Querying the Filter

The process to query F_Y with an example element $x =$ "horse" follows many of the same steps as building the filter. First, the same hash scheme from above is used to generate an XOR clause for "horse".

$$\begin{aligned}H &= \mathsf{xxHash}(\text{“horse”})\\ &= \texttt{0x3f37a1a7}\ .\end{aligned}$$

Next, the hash is split into groups of 4 bits.

$$SH = [\texttt{0x3}, \texttt{0xf}, \texttt{0x3}, \texttt{0x7}, \texttt{0xa}, \texttt{0x1}, \texttt{0xa}, \texttt{0x7}]\ .$$

Then, three clause indices and a right-hand side are generated from the hash.

$$\begin{aligned}\mathcal{I} &= [SH_0(\text{mod } n), SH_1(\text{mod } n), SH_2(\text{mod } n), SH_3(\text{mod } 2^s)] \\ &= [\texttt{0x3}(\text{mod } 4), \texttt{0xf}(\text{mod } 4), \texttt{0x3}(\text{mod } 4), \texttt{0x7}(\text{mod } 8)] \\ &= [3, 3, 3, 7] \ .\end{aligned}$$

Finally the clause is created from the indices and right-hand side and two bits (all True) are appended to support metadata retrieval.

$$\begin{aligned}C &= x_3 \oplus x_3 \oplus x_3 \equiv [1, 1, 1] \ || \ [1, 1] \\ &= \qquad\qquad x_3 \equiv [1, 1, 1, 1, 1] \ .\end{aligned}$$

In Algorithm 5, the right-hand side metadata bits are all set to False and the terminal equivalence ($\equiv$) is treated as an XOR ($\oplus$). That choice was made purely for presentation of the algorithm. This example demonstrates that either way is acceptable.

Now that the clause C has been built, it can be tested against the filter F_Y. To do so, assign the variables in C their values in the stored solutions of F_Y and evaluate the resulting equation.

$$\begin{aligned}C_{F_Y} &= F_Y(3) \equiv [1, 1, 1, 1, 1] \\ &= [0, 1, 0, 0, 1] \equiv [1, 1, 1, 1, 1] \\ &= [0, 1, 0, 0, 1] \ .\end{aligned}$$

Since the first three bits of C_{F_Y} are not all True, the element does not pass the filter. Hence, the string "horse" is definitively not in Y.

The final part of this example demonstrates a query that passes and returns stored metadata. Specifically, F_Y will be queried with $x =$ "cat". Again, the same hash scheme from above is used to generate an XOR clause for "cat". Since this was already demonstrated in the previous section, the details will not be repeated. Instead, the clause C is simply stated next, including the two True bits appended to support metadata retrieval.

$$C = x_0 \oplus x_1 \oplus x_3 \equiv [1, 0, 0, 1, 1] \ .$$

Evaluating C against F_Y produces

$$\begin{aligned}C_{F_Y} &= F_Y(0) \oplus F_Y(1) \oplus F_Y(3) \equiv [1, 0, 0, 1, 1] \\ &= [1, 0, 0, 1, 1] \oplus [0, 1, 0, 1, 0] \oplus [0, 1, 0, 0, 1] \equiv [1, 0, 0, 1, 1] \\ &= [1, 0, 0, 0, 0] \equiv [1, 0, 0, 1, 1] \\ &= [1, 1, 1, 0, 0] \ .\end{aligned}$$

Since the first three bits of C_{F_Y} are all True, the element passes the filter. Hence, "cat" is in Y with a $\frac{1}{2^3}$ chance of being a false positive. The last two bits of C_{F_Y}, $[0, 0]$, represent the stored metadata, namely, the index 0.

6 Experimental Results

This section serves to demonstrate that it is practical to build efficient XORSAT filters for very large data sets. To do so, a research-grade XORSAT solver and XORSAT filter construction were implemented in the C language. The solver performs the 'Method of the Four Russians' [29].

As a proof of concept, seventeen dictionaries were built consisting of $2^{10}, \ldots,$ and 2^{26} random 16-byte strings. To ensure that random k-XORSAT instances were generated, the strings were transformed into XOR clauses using the 64-bit xxHash hash algorithm [10]. Each string was fed into a single call of xxHash and the output was used to seed a linear feedback shift register (LFSR) with 16-bit elements and primitive polynomial $1 + x^2 + x^3 + x^4 + x^8$. XOR clauses were produced by stepping the LFSR k times. Duplicate removal was not considered as searching for duplicates drastically increases query performance yet only marginally increases efficiency. All of the following results were collected using a late 2013 MacBook Pro with a 2.6-GHz Intel Core i7 and 16 GB of RAM. All times are reported in seconds.

Table 3. Achieved efficiency and seconds taken to build *non-blocked* XORSAT filters with $p = \frac{1}{2^{10}}$. m is the number of elements in the data set being stored and k is the number of variables per XOR clause.

m	$k = 3$	$k = 4$	$k = 5$	$k = 6$
2^{10}	(88%, < 1)	(93%, < 1)	(93%, < 1)	(93%, < 1)
2^{11}	(89%, < 1)	(97%, < 1)	(97%, < 1)	(97%, < 1)
2^{12}	(90%, < 1)	(97%, < 1)	(98%, < 1)	(98%, < 1)
2^{13}	(91%, 1)	(97%, 1)	(98%, 1)	(99%, 1)
2^{14}	(91%, 2)	(97%, 3)	(99%, 4)	(99%, 5)
2^{15}	(89%, 17)	(97%, 21)	(98%, 28)	(98%, 36)

Table 3 presents the achieved efficiency and time taken to build *non-blocked* XORSAT filters, that is, for each filter, $s = 10$ XORSAT instances were generated and one solution was found for each. The instances were solved in parallel using a single call to the XORSAT solver.

Unlike SAT filters, the number of solutions found does not affect either efficiency or runtime so long as s is less than the word-size of the computer (typically 64 bits, a very reasonable assumption in practice given that $s > 64$ would mean building a filter with a false positive rate less than $\frac{1}{2^{64}}$). Efficiency is not affected because the s right-hand sides are all uncorrelated. Runtime is not affected because all s instances are solved in parallel using bit-packing and word-level operations.

The XORSAT filters in Table 3 and Table 4 achieve the desired false positive rate. This was verified experimentally by querying each XORSAT filter with 2^{23}

4-byte elements and using the results to calculate the achieved false positive rate and, for Table 4, query speed as well.

In terms of efficiency, the experimental results match the theoretical results from Table 1. And, if the number of XOR clauses per instance is above 2^{14}, filters can be practically built that are very close to the optimal efficiency possible for each given k.

The results also hint correctly that it is not practical to build *non-blocked* XORSAT filters for very large data sets as runtime will grow and become prohibitive in practice. It is likely that filter build time can be reduced by using a more powerful solver (such as M4RI [2]), but this has not been explored here.

However, build time can be significantly reduced by building *blocked* XORSAT filters. As discussed in Section 2.3, first hashing elements into small blocks and then, in parallel, building one filter for each block can drastically reduce the build time of a large data set without increasing query time and only marginally reducing efficiency. Since build time is one-time work, discovering techniques for reducing build time any further is likely unnecessary.

Results in Table 3 can be used to tune blocked XORSAT filter schemes. For example, setting the block size between 2^{11} and 2^{12} and $k = 5$ will enable fast building of blocked XORSAT filters with $\mathcal{E} \approx 98\%$. Table 4 presents the build time, achieved efficiency, filter size, and query speed for blocked XORSAT and SAT filters using these sample parameters. The table also presents query speed for Bloom filters built and queried using the same data sets, false positive rate, bucketing, and element hashing scheme. Though, Bloom filters can only achieve a maximum efficiency of $\ln 2 = 69\%$, meaning that they use approximately 44% more long-term storage than an XORSAT filter for the same data at the same false positive rate.

Table 4 demonstrates that it is practical to build XORSAT filters very near the information theoretic limit while maintaining a high query speed. Since each block can be build in parallel, linear speedup is achieved and demonstrated in the results. As with SAT filters, XORSAT filter query speed can be increased by decreasing k which may in turn decrease efficiency.

The query speed of the above filter implementations begin to drop after data sets grow above 2^{20}. This is due to size of the filter overwhelming the caching mechanisms of the computer running the experiments. It may be possible to create a cache-aware implementation of XORSAT filters that increases query speed overall and removes some of the query speed variance seen in Table 4, though this has not been explored.

Efficiency also slowly drops as filters increase in size. As discussed in Section 2.3, since blocks may not all hold the same number of elements, it is necessary to store additional information so that the blocks can be accessed during query. This additional information must be stored as part of the filter and, hence, increases the size of the filter, decreasing efficiency.

Table 4. Achieved efficiency, size (in KB), and seconds taken to build *blocked* XORSAT and SAT filters with an expected 3072 elements per block, variables per clause $k = 5$, and desired false positive rate $p = \frac{1}{2^{10}}$. Desired SAT filter efficiency was set to 75% and desired XORSAT filter efficiency was set to 98%. The SAT filter hamming weight metric [31] was set to 48%. Timeout ('-') was set at one hour. Query speed (in millions of queries per second) is also given for XORSAT, SAT, and Bloom filters.

	XORSAT Filter				SAT Filter						
	Build Time				Build Time				Query Speed		
m	1 Core	8 Cores	$\mathcal{E}$	Size	1 Core	8 Cores	$\mathcal{E}$	Size	XORSAT	SAT	Bloom
2^{15}	< 1	< 1	98%	41	336	105	43%	56	18	4	23
2^{16}	1	< 1	98%	81	883	183	43%	111	18	4	23
2^{17}	2	< 1	98%	163	1768	394	43%	222	18	4	23
2^{18}	5	1	98%	326	3441	723	44%	444	18	4	23
2^{19}	8	1	97%	659	-	1724	44%	887	18	4	23
2^{20}	17	2	97%	1321	-	-	-	-	18	-	22
2^{21}	33	4	97%	2646	-	-	-	-	17	-	22
2^{22}	92	12	97%	5298	-	-	-	-	13	-	20
2^{23}	186	26	97%	10601	-	-	-	-	9	-	20
2^{24}	372	52	97%	21204	-	-	-	-	11	-	20
2^{25}	751	104	96%	42416	-	-	-	-	10	-	17
2^{26}	1515	208	96%	84958	-	-	-	-	7	-	12

7 Conclusions and Future Work

The XORSAT filter is the first *practical* SAT filter construction, overcoming many of the previous hurdles presented in [31]. It is a simple offline filter construction for very large data sets that can consistently achieve the efficiency bound in practice while maintaining fast queries. This new filter construction is also parameterized so that it can be easily tailored to support an application needing, for example, fast build time, fast queries, a small memory footprint, and metadata storage and retrieval.

Potential future work includes considering XORSAT filters as part of a secure search scheme [18]. This would involve tailoring the filter construction to make it secure or resistant to various attacks such as inversion and intersection, as well as many others [27, 32].

Moving from disjunctive clauses to XOR clauses provides for a SAT filter with different features (ex. near perfect efficiency, fast build time, metadata support, hints of security). Hence, it is possible that SAT filters built from other constraint types could provide other common sought after filter features such as streaming (online filters), element deletion, or element counting.

References

1. Achlioptas, D.: Random satisfiability. In: Biere et al. [5], pp. 245–270
2. Albrecht, M., Bard, G.: The m4ri library (2018), https://m4ri.sagemath.org
3. Azinović, M., Herr, D., Heim, B., Brown, E., Troyer, M.: Assessment of quantum annealing for the construction of satisfiability filters. SciPost Phys. 2, 013 (2017), https://scipost.org/10.21468/SciPostPhys.2.2.013
4. Bard, G.V.: The Method of Four Russians, pp. 133–158. Springer, Boston, MA (2009), https://doi.org/10.1007/978-0-387-88757-9_9
5. Biere, A., Heule, M., van Maaren, H., Walsh, T. (eds.): Handbook of Satisfiability. Frontiers in Artificial Intelligence and Applications **185**, IOS Press (2009)
6. Bloom, B.H.: Space/time trade-offs in hash coding with allowable errors. Communications of the ACM 13(7), 422–426 (1970)
7. Brodnik, A., Munro, J.I.: Membership in constant time and almost-minimum space. SIAM Journal on Computing 28(5), 1627–1640 (1999)
8. Chazelle, B., Kilian, J., Rubinfeld, R., Tal, A.: The Bloomier filter: an efficient data structure for static support lookup tables. In: Proceedings of the Fifteenth Annual ACM-SIAM Symposium on Discrete Algorithms. pp. 30–39. Society for Industrial and Applied Mathematics (2004)
9. Cohen, S., Matias, Y.: Spectral Bloom filters. In: Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data. pp. 241–252. ACM (2003)
10. Collet, Y.: xxHash: Extremely fast hash algorithm (2017)
11. Daudé, H., Ravelomanana, V.: Random 2-XORSAT at the satisfiability threshold. In: Latin American Symposium on Theoretical Informatics. pp. 12–23. Springer (2008)
12. Dietzfelbinger, M., Goerdt, A., Mitzenmacher, M.D., Montanari, A., Pagh, R., Rink, M.: Tight thresholds for cuckoo hashing via XORSAT. In: International Colloquium on Automata, Languages, and Programming. pp. 213–225. Springer (2010)
13. Dietzfelbinger, M., Pagh, R.: Succinct data structures for retrieval and approximate membership. In: International Colloquium on Automata, Languages, and Programming. pp. 385–396. Springer (2008)
14. Douglass, A., King, A.D., Raymond, J.: Constructing SAT filters with a quantum annealer. In: International Conference on Theory and Applications of Satisfiability Testing. pp. 104–120. Springer (2015)
15. Erdős, P., Renyi, A.: On a classical problem of probability theory (1961)
16. Fan, B., Andersen, D.G., Kaminsky, M., Mitzenmacher, M.D.: Cuckoo filter: Practically better than Bloom. In: Proceedings of the 10th ACM International Conference on emerging Networking Experiments and Technologies. pp. 75–88. ACM (2014)
17. Fang, C., Zhu, Z., Katzgraber, H.G.: NAE-SAT-based probabilistic membership filters. preprint arXiv:1801.06232 (2018)
18. Goh, E.J., et al.: Secure indexes. IACR Cryptology ePrint Archive 2004, 216 (2004)
19. Heule, M.J., van Maaren, H.: Parallel SAT solving using bit-level operations. Journal on Satisfiability, Boolean Modeling and Computation 4, 99–116 (2008)
20. Ibrahimi, M., Kanoria, Y., Kraning, M., Montanari, A.: The set of solutions of random XORSAT formulae. In: Proceedings of the Twenty-third Annual ACM-SIAM Symposium on Discrete Algorithms. pp. 760–779. SIAM (2012)

21. Kader, A.A., Dorojevets, M.: Novel integration of Dimetheus and WalkSAT solvers for k-SAT filter construction. In: Systems, Applications and Technology Conference (LISAT2017). pp. 1–5. IEEE (2017)
22. Krimer, E., Erez, M.: The power of $1+\alpha$ for memory-efficient Bloom filters. Internet Mathematics 7(1), 28–44 (2011)
23. Mitchell, D., Selman, B., Levesque, H.: Hard and easy distributions of SAT problems. In: AAAI92. pp. 459–465 (1992)
24. Mitzenmacher, M.D.: Compressed Bloom filters. IEEE/ACM Transactions on Networking (TON) 10(5), 604–612 (2002)
25. Pittel, B., Sorkin, G.B.: The satisfiability threshold for k-XORSAT. Combinatorics, Probability and Computing 25(02), 236–268 (2016)
26. Porat, E.: An optimal Bloom filter replacement based on matrix solving. In: Frid, A.E., Morozov, A., Rybalchenko, A., Wagner, K.W. (eds.) Computer Science — Theory and Applications, Fourth International Computer Science Symposium in Russia, CSR 2009, Proceedings. Lecture Notes in Computer Science, vol. 5675, pp. 263–273. Springer (2009), https://doi.org/10.1007/978-3-642-03351-3_25
27. Pouliot, D., Wright, C.V.: The shadow nemesis: Inference attacks on efficiently deployable, efficiently searchable encryption. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security. pp. 1341–1352. ACM (2016)
28. Putze, F., Sanders, P., Singler, J.: Cache-, hash-, and space-efficient Bloom filters. Journal of Experimental Algorithmics 14, 4 (2009)
29. V. l., A., Dinits, E., Kronrod, M., I. A., F.: On economical construction of transitive closure of an oriented graph. Doklady Akademii Nauk SSSR 194(3), 487 (1970)
30. Walker, A.: Filters. Master's thesis, Haverford College (2007), http://math.uchicago.edu/~akwalker/filtersFinal.pdf
31. Weaver, S.A., Ray, K.J., Marek, V.W., Mayer, A.J., Walker, A.K.: Satisfiability-based set membership filters. Journal on Satisfiability, Boolean Modeling and Computation 8, 129–148 (2014)
32. Zhang, Y., Katz, J., Papamanthou, C.: All your queries are belong to us: The power of file-injection attacks on searchable encryption. In: USENIX Security Symposium. pp. 707–720 (2016)