中国科学技术大学

读书报告

布隆过滤器及其衍生的新型数据结构

作者姓名：　柳枫

学科专业：　网络空间安全

导师姓名：　薛开平教授

完成时间：　二〇二四年九月二十三日

目　录

符 号 说 明

a	The number of angels per unit area
N	The number of angels per needle point
A	The area of the needle point
σ	The total mass of angels per unit area
m	The mass of one angel
$\sum_{i=1}^{n} a_i$	The sum of a_i

第 1 章　简　　介

1.1　布隆过滤器

在构造网络安全相关协议时，我们通常会使用到许多不同类型的数据结构。其中，布隆过滤器 (Bloom filter)[1] 作为一种经典的概率型数据结构，在 IP 地址过滤、识别恶意邮件、DoS 和 DDos 攻击检测等场景有着广泛的应用[2]。布隆过滤器的作用是快速判断元素是否属于某一集合 (membership query)，它是由 k 个哈希函数构造的哈希表结构，具有空间效率高、判断速度快的特点。布隆过滤器的构造如图 1.1 所示，哈希表的每个位置上存储的是 0/1 比特，每个元素对应的位置由 k 个哈希函数所确定。以包含 n 个元素的集合 $S = \{x_1, x_2, \cdots, x_n\}$ 为例，

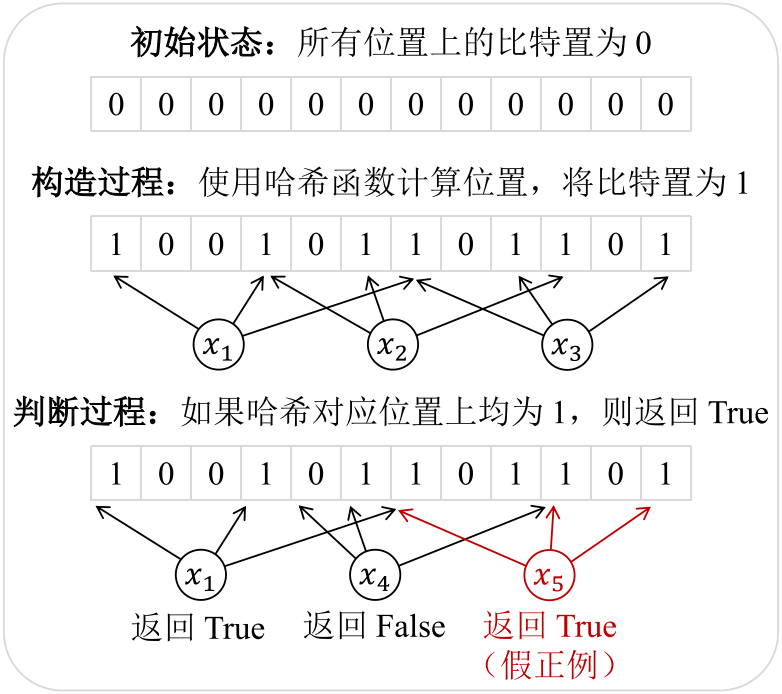

图 1.1　布隆过滤器示例（$k = 3$）

假设构造的布隆过滤器长度为 m，使用的哈希函数为 $\{h_1, h_2, \cdots, h_k\}$，其中每个哈希函数 $h_i : \{0,1\}^* \to [1, m]$ 为任意长度的输入到布隆过滤器上某一位置的映射。首先我们将布隆过滤器 m 个位置上的比特都置为 0，然后插入集合 S 中的每一个元素。在插入元素 x 时，需要使用 k 个哈希函数计算出 k 个位置信息，即 $\{h_1(x), h_2(x), \cdots, h_k(x)\}$。最后将布隆过滤器上这 k 个位置上的比特都置为 1。在判断某个元素是否属于集合 S 时，只需要计算该元素对应的 k 个位置，然后检查这 k 个位置上的比特是否都为 1。只要有一个位置上出现了 0，那么判断结果就是不属于；否则，布隆过滤器认为该元素属于集合 S。对于大小为 n 的集合，其对应的的布隆过滤器需要 $O(nk)$ 的存储开销以及 $O(k)$ 的判断复杂度。

从布隆过滤器的构造和判断过程可以看出，如果一个元素属于集合 S，那么判断结果一定是正确的；但是如果一个元素不属于集合 S （如图 1.1 中的 x_5），布隆过滤器也有可能会认为该元素属于 S（输出结果为 True），此时判断错误的概率也称为假正例率 (false positive rate)。尽管布隆过滤器存在误判的问题，但在实际应用场景中，只要将误判率控制在较小的值，一般认为以一定的误判换取低空间开销和高效判断是值得的。

布隆过滤器的假正例率 ϵ 由布隆过滤器的长度 m，使用的哈希函数个数 k 和集合中的元素个数 n 所决定。根据文献[3]中的推导，理论上，误判率 ϵ 与它们的关系如公式 (1.1)所示：

$$\epsilon = \left[1 - \left(1 - \frac{1}{m}\right)^{nk}\right]^k \approx \left(1 - e^{-\frac{kn}{m}}\right)^k, \tag{1.1}$$

其中，$(1 - 1/m)^{nk}$ 近似成 $e^{-kn/m}$ 的形式。为了尽可能降低误判率 ϵ，那么就需要尽可能降低 $e^{-kn/m}$ 的值，这样一来，k 的最优取值为：

$$k_{opt} = \frac{m}{n} \ln 2 \approx \frac{9m}{13n}. \tag{1.2}$$

此时，误判率大约为 $\epsilon \approx 2^{-k} \approx 0.6185^{m/n}$。通常在实际应用中的误判率要比理论分析上的更高，也有一些工作[4-5]对误判率做了更精确的刻画。布隆过滤器的优势主要体现在以下几个方面：

- **较低的存储开销：**布隆过滤器的大小与元素的大小无关，只与集合中元素的数量有关。比如，当给定 m 与 n 的比值为 5 时，根据公式 (1.2) 可以计算出需要的哈希函数数量为 3 或 4。因为布隆过滤器中每个位置上存储的都是比特，所以整体长度也就是 m 比特。
- **较高的判断效率：**因为只需要检查 k 个位置上的比特是否全为 1，因此检查一个元素的时间复杂度为 $O(k)$。相比于树形结构的查询复杂度 $O(\log(n))$ 或列表结构的查询复杂度 $O(n)$ 都要更低。
- **不会漏判：**尽管布隆过滤器在查询时会存在误判的情况，但是它不会出现漏判（假负例）的情况。也就是只要是布隆过滤器判断元素 x 不属于 S，那么该论断一定是正确的。

但是，布隆过滤器也存在一些局限性：

- **假正例率与过滤器长度：**由于布隆过滤器中存在假正例的情况，而假正例率与过滤器的长度 m 成反比。根据式 1.1，二者之间的关系为 $m \approx \frac{13}{9}nk \approx 1.44n\log_2(1/\epsilon)$。为了保证足够低的假正例率，就需要更大的 m 来避免不同哈希映射导致的冲突。如此一来，过滤器的空间利用率就会降低。
- **动态性：**布隆过滤器本身不支持元素的删除，因为如果是简单将需要删除元素所对应位置的比特置为 0，那么就会影响对其他元素的判断。

- **功能性：**布隆过滤器只能判断元素是否属于某个集合，并不能检索元素对应的某一函数值或映射结果。

1.2　定义及分类

在介绍布隆过滤器相关衍生的数据结构之前，我们需要对这些数据结构进行分类。为此，我们首先要对这些数据结构进行统一的定义。

定义 1.1 (过滤器)　令 $\mathcal{U}$ 表示元素的集合，$\mathcal{H}$ 为哈希函数的集合。过滤器一般包含以下两个算法：

- $\mathsf{Construct}(S, \mathcal{H}) \to F/\perp$：输入集合 $S \subseteq \mathcal{U}$ 和预先给定的哈希函数集合 $\mathcal{H}$，输出构造的过滤器 F（或者以可忽略的概率输出错误指示符 $\perp$）。
- $\mathsf{Evaluate}(x, \mathcal{H}, F) \to$ True/False：输入元素 x，预先给定的哈希函数集合 $\mathcal{H}$，输出结果 True 或者 False。

正确性：对于任意的 $S \subseteq \mathcal{U}$，都有：a). 构建过程中，输出 $\perp$ 的概率是可忽略的；b). 如果 $F \leftarrow \mathsf{Construct}(S, \mathcal{H})$，且 $F \neq \perp$，那么在判断过程中，对于任意的 $x \in S$，$\Pr[\mathsf{Evaluate}(x, \mathcal{H}, F) = \text{True}] = 1$；对于任意 $x' \notin S$，$\Pr[\mathsf{Evaluate}(x, \mathcal{H}, F) = \text{True}]$ 为可忽略的。

从以上定义中可以看出，如果一个元素在原本输入的集合中，那么过滤器在判断过程中一定能返回正确的结果，即过滤器中不存在假负例的情况；反之，如果一个元素不存在于输入的集合中，过滤器会大概率返回正确的结果，即过滤器中会存在一定的假正例率。

在判断过程中，需要使用 $\mathcal{H}$ 中的哈希函数计算出元素在 F 中对应的位置，再对这些位置上记录的结果进行计算，最后根据计算结果与事先定义的 $f(x)$ 进行比较返回 Ture 或者 False。计算过程也被称为探测 (probing)，文献[6]根据探测方式将过滤器分为以下三种类型：

- **AND 型：**在通过哈希函数计算出的位置中，如果所有位置上结果都与 $f(x)$ 相等，那么就输出 True；
- **OR 型：**在通过哈希函数计算出的位置中，如果至少有一个位置上的值与 $f(x)$ 相等，那么就输出 True，否则输出 False。
- **XOR 型：**在通过哈希函数计算出的位置中，如果所有位置上值的异或结果与 $f(x)$ 相等，那么就输出 True，否则输出 False。

从以上分类可以看出，布隆过滤器的判断方式属于 AND 型。OR 型的典型代表是布谷鸟过滤器 (cuckoo filter)[7]，这种构造的特点是支持元素的动态插入和删除。XOR 型的典型代表是异或过滤器 (xor filter)[7]，这类过滤器在结构上非常紧凑，具有较高的空间利用率，但受限于构建方式，这类构造通常不能支持元素的

动态更新。

不同的过滤器中对 $f(x)$ 的定义也略有不同。一般来说分为以下几种情况：

- $f(x)$ 为比特 1，最典型的是布隆过滤器，即要求 x 对应所有位置上的结果都为 1，过滤器才会返回 True。
- $f(x)$ 等于元素 x 本身，典型的是混淆布隆过滤器 (garbled Bloom filter)，即计算的结果为 x，过滤器才会返回 True。
- $f(x)$ 为 x 的指纹 (fingerprint)，一般指的是 x 的哈希值，典型代表为布谷鸟过滤器，即当有一个位置上的值与 x 的指纹相同，过滤器才会返回 True。
- $f(x)$ 为任意函数，典型的是 Bloomier 过滤器，也就是说 $f(x)$ 的形式并不重要，或者说只要满足是 x 一种映射关系即可。

从上述分类可以看出，对 $f(x)$ 的定义可以是简单的比特 1，也可以是关于 x 的任意一种映射关系。从另一个角度来看，键值型数据也可以看作是从键到值的映射，这样一来，键值型数据也可以编码成过滤器的形式。按照这种想法得到的构造称作不经意键值存储 (Oblivious Key-Value Store, OKVS)，与过滤器不同，不经意键值存储返回的不是 True 或者 False，而是与输入键相对应的值。不经意键值存储的定义如下：

定义 1.2 (不经意键值存储)　令 $\mathcal{K}$ 和 $\mathcal{V}$ 分别表示键和值的集合。不经意键值存储包含两个算法：

- $\mathsf{Encode}(\{(x_1, y_1), (x_2, y_2), \cdots, (x_n, y_n)\})$：输入一组键值对 $\{(x_i, y_i)\}_{i\in[n]} \subseteq \mathcal{K}\times\mathcal{V}$，输出编码结果 D 或以可忽略的概率输出错误指示符 $\perp$。
- $\mathsf{Decode}(D, k)$：输入编码结果 D 和键 $k \in \mathcal{K}$，输出值 $v \in \mathcal{V}$。

正确性：对于任意键值对集合 $A \subseteq \mathcal{K}\times\mathcal{V}$，如果 $\mathsf{Encode}(A) = D$ 且 $D \neq\perp$，那么对于任意的 $(k, v) \in A$，都有 $\mathsf{Decode}(D, k) = v$。

不经意性：随机给定两个不同的集合 $\mathcal{K}_0 = \{x_1^0, x_2^0, \cdots, x_n^0\}$ 和 $\mathcal{K}_1 = \{x_1^1, x_2^1, \cdots, x_n^1\}$，再随机构造集合 $\{y_i\}_{i\in[n]} \leftarrow \mathcal{V}$，若 $D \neq\perp$，则分布 $\{D|y_i \leftarrow \mathcal{V}, i \in [n], \mathsf{Encode}(\{(x_1^0, y_1), (x_2^0, y_2), \cdots, (x_n^0, y_n)\})\}$ 与分布 $\{D|y_i \leftarrow \mathcal{V}, i \in [n], \mathsf{Encode}(\{(x_1^1, y_1), (x_2^1, y_2), \cdots, (x_n^1, y_n)\})\}$ 统计不可区分。

对于部分安全需求较高的协议，不经意键值存储还需要满足随机性，即：

随机性：对于任意的集合 $A = \{(x_i, y_i)\}_{i\in[n]} \subseteq \mathcal{K}\times\mathcal{V}$ 和 $x' \notin \{x_1, x_2, \cdots, x_n\}$，如果 $\mathsf{Encode}(A) = D$ 且 $D \neq\perp$，则 $\mathsf{Decode}(D, x')$ 的输出与随机选择的 $v \leftarrow \mathcal{V}$ 统计不可区分。

我们主要考虑这些数据结构在存储和计算两个方面的性能。在存储开销方面，一是要衡量过滤器本身的长度，即 m 的数值，我们一般使用负载因子 (load factor) $\alpha = n/m$ 作为评估指标，即 α 越接近 1，说明空间利用率越高；二是需要衡量均摊空间开销 (amortized space cost)，即平均每个元素上在过滤器中所占用的

存储空间。在计算开销方面，我们主要分析构建/评估（或编码/解码）过程的复杂度。表 1.1对本文中所介绍的过滤器以及不经意键值存储结构进行了总结。我们将在接下来的内容里对这些新型数据结构的构建方法进行详细介绍。

表 1.1　不同过滤器及相关结构总结

名称	探测类型	$f(x)$	负载因子 α
布隆过滤器	AND 型	1	$1.44\log_2(1/\epsilon)$
布谷鸟过滤器	OR 型	x 的指纹值	
异或过滤器	XOR 型	x 的指纹值	
二进制引信过滤器	XOR 型	x 的指纹值	
缎带过滤器	XOR 型	x 的指纹值	
Bloomier 过滤器	XOR 型	$f(x)$	
混淆布隆过滤器	XOR 型	x	
随机带状不经意键值存储	XOR 型	键 x 对应的值 y	

1.3　总结

布隆过滤器是一种近似成员查询过滤器 (approximate membership query filter)，即以一定的误判率回答查询元素是否存在于集合中。得益于其简洁的构造方式以及高效的判断性能，布隆过滤器的应用场景非常广泛。除了前面提到的 IP 地址过滤、识别恶意邮件、DoS 和 DDos 攻击检测等场景之外，布隆过滤器及其衍生的数据结构也常被用于构造隐私保护相关协议[8]。布隆过滤器自提出后，各种变体形式层出不穷，如计数式布隆过滤器 (counting Bloom filter)[9]，压缩布隆过滤器 (compressed Bloom filter)[10]，d-left 计数式布隆过滤器[11]，块布隆过滤器 (blocked Bloom filter)[12]等。关于布隆过滤器的变体不在本文的讨论范围内，我们主要介绍布隆过滤器衍生出的新型数据结构，包括布谷鸟过滤器、异或过滤器、不经意键值存储等。我们将在第 2章对这些衍生数据结构进行详细介绍。最后，我们将在第 3章介绍这些数据结构在隐私保护方面的应用。

第 2 章　新型数据结构

在这一章，我们将重点介绍由布隆过滤器衍生出来的几个特殊数据结构，分别是布谷鸟过滤器，异或型过滤器以及不经意键值存储。

2.1　布谷鸟过滤器

从上一章的分类我们可以知道，布谷鸟过滤器是一种 OR 型的过滤器，且 $f(x)$ 为 x 的指纹信息。布谷鸟过滤器的概念最早是由 Fan 等人[7] 于 2014 年提出的，其构造方式受到了布谷鸟哈希表 (cuckoo hash table)[13] 的启发。在介绍布谷鸟过滤器之前，我们首先介绍布谷鸟哈希表的构造。

2.1.1　布谷鸟哈希表

布谷鸟哈希表可以看作一个由多个桶 (bucket) 组成的数组结构。对于每个元素 x 来说，它在哈希表中对应两个候选位置，分别由两个哈希函数 $h_1(x)$ 和 $h_2(x)$ 决定。在插入元素 x 时，首先检查 x 对应的两个位置上的桶中是否有多余位置。如果两个桶都有多余空间，则直接将 x 放入桶中；如果两个桶都已满，则随机在一个候选位置上踢出一个元素并将 x 放入该位置上。踢出的元素则重新插入到它对应的另一个候选位置上。如图 2.1 所示，假设每个桶中都只能存储一个元素，当插入元素 x 时，首先计算出它的两个候选位置分别是 2 和 7。因为 2 和 7 两个

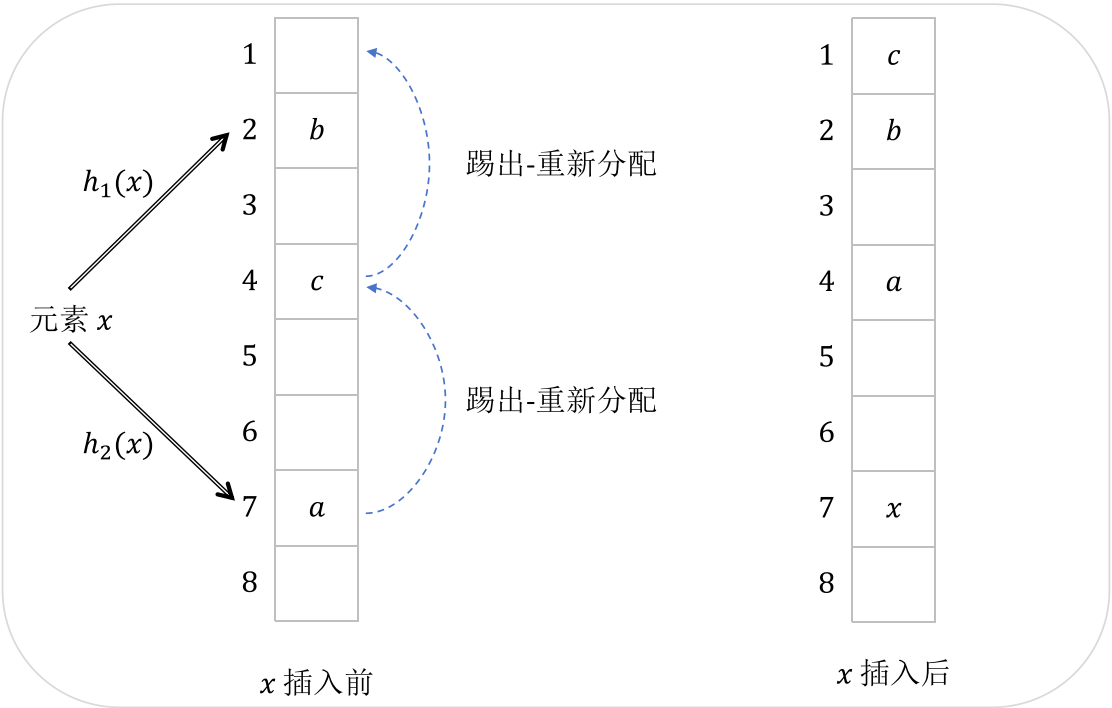

图 2.1　布谷鸟哈希表示例

位置都已满，这里选择踢出位置 7 上的元素 a，并将 x 放入其中。被踢出的 a 则重新插入到它的另一个候选位置，也就是 4 上。由于位置 4 上已有元素 c，则把 c 踢出，将 a 存储在位置 4 中，并将 c 重新分配到它的候选位置 1 上。这种 “踢出-重新分配” 的思路与布谷鸟下蛋时会将蛋放入其他鸟的巢穴，并挤出原本巢中蛋的行为很相似，因此而得名。

2.1.2　布谷鸟过滤器的构造

与布谷鸟哈希表类似，布谷鸟过滤器也是由多个桶组成的数组结构。不同的是，在布谷鸟过滤器中每个桶中存储的并不是元素本身，而是元素的指纹。这就导致在将桶中的元素指纹踢出时，无法根据指纹信息确定它的另一个候选位置。因此布谷鸟过滤器采用了部分密钥布谷鸟哈希 (partial-key cuckoo hashing) 的技巧来解决该问题。也就是将元素的两个候选位置与元素的指纹值建立联系，这样就能在只知道元素的指纹值和其中一个位置信息的情况下计算出另一个位置信息。具体来说，对于元素 x，其对应的两个候选位置计算如下：

$$h_1(x) = \mathsf{hash}(x) \tag{2.1}$$

$$h_2(x) = h_1(x) \oplus \mathsf{hash}(x\text{'s fingerprint}) \tag{2.2}$$

式 2.2 中的异或操作正好满足了上述性质，即 $h_1(x)$ 也可以通过 $h_2(x)$ 和 x 的指纹信息计算得出。另外，在异或操作中采用的是 x 指纹的哈希而不是 x 的指纹本身，这样做是因为如果只用指纹本身的话，两个候选位置之间的距离就受限于指纹的取值范围。比如使用 8 比特长度的指纹，那么两个候选位置之间最多相差 256。而使用指纹的哈希则可以确保两个候选位置可以分布在过滤器中的任意位置，从而降低哈希碰撞的概率并提高存储空间的利用率。

通过上述讨论，我们可以直接给出布谷鸟过滤器的 Construct 算法思路。以输入集合 S 为例，对于每一个元素 $x \in S$，插入 x 的过程描述如下：

- 首先计算 x 的指纹值 f，以及两个候选位置信息 $h_1(x)$ 和 $h_2(x)$。
- 只要 $h_1(x)$ 和 $h_2(x)$ 两个位置上有一个桶有空余，那么直接将 f 插入空余的桶中，否则进入下一步。
- 随机从 $h_1(x)$ 和 $h_2(x)$ 中选取一个位置 i，并从该位置上踢出一个指纹，并将 f 插入。踢出的指纹再计算出它的另一个候选位置，执行插入步骤。

在布谷鸟过滤器的构造过程中，会设置一个最大踢出值，当踢出的指纹超过最大值时将直接返回错误指示符 $\perp$。

布谷鸟过滤器的 Evaluate 算法比较直接，给定输入的元素 x，只需要计算它对应的两个位置 $h_1(x)$ 和 $h_2(x)$，如果在这两个位置上至少有一个桶中含有 x 的指纹，那么返回 True，否则返回 False。

除了在**定义 1.1**中的 Construct 和 Evaluate 两个算法之外，布谷鸟过滤器中还支持删除操作。这也是布谷鸟过滤器相比布隆过滤器的一大优势。删除操作与 Evaluate 过程类似，也比较直接，即对需要删除的元素 x 计算出 $h_1(x)$ 和 $h_2(x)$ 两个位置之后，如果这两个位置上有一个包含 x 的指纹，则直接移除该位置上的指纹信息。注意在删除过程中，如果找到两个位置上都存在 x 的指纹时，只需要移除其中一个位置上的指纹信息即可。这是因为当两个元素具有相同的指纹信息时，这样做就不会影响对另一个元素的存在性判断。当然，只删除一个会带来假正例的问题，但这对于大部分过滤器结构来说都是无法避免的，我们只需要将假正例的概率控制在较小的值即可。

2.1.3　布谷鸟过滤器的性能

我们用 $|f|$ 表示指纹值的比特长度，当给定 $h_1(x)$ 的值时，也就确定了 $h_2(x)$ 有 $2^{|f|}$ 种不同取值。假设布谷鸟过滤器中包含 m 个桶，当 $2^{|f|} < m$ 时，$h_2(x)$ 的取值范围也就是整个过滤器长度的子集。因此，当 $|f|$ 取值越小时，哈希碰撞的几率也会越大，构建过滤器的失败概率也会随之增大。而且当 m 与 $2^{|f|}$ 之间的差距越大时，布谷鸟过滤器的空间利用率也会越低。如何设定合适的参数就显得尤为重要。

如章节 1.2 中所述，我们使用负载因子 α $(0 \leqslant \alpha \leqslant 1)$ 来表示布谷鸟过滤器的空间利用率，它的定义是过滤器中已占用空间大小与过滤器大小的比值。因此当 α 的值越接近 1，那就表示空间利用率越高。在给定指纹长度 $|f|$ 和负载因子 α 的情况下，对于每个元素均摊的空间开销 C 可以表示为：

$$C = \frac{\text{过滤器的存储大小}}{\text{存储的条目数}} = \frac{|f| \cdot \text{总条目数}}{\alpha \cdot \text{总条目数}} = \frac{|f|}{\alpha} \text{ bits.} \tag{2.3}$$

负载因子的大小受到桶大小（用 b 表示）的影响。当 $b = 1$ 时，负载因子仅有 50%，而当 $b = 4$ 或 $b = 8$ 时，负载因子随之增长为 95% 和 98%。而当桶越大时，就越容易出现碰撞（即相同指纹信息）的情况。为了保证相同的假正例概率，就要求指纹长度越长。根据文献[7]中的推导，指纹长度 $|f|$ 与假正例率 ϵ 和桶大小 b 之间的关系如下所示：

$$|f| \geqslant \lceil \log_2(2b/\epsilon) \rceil = \lceil \log_2(1/\epsilon) + \log_2(2b) \rceil \text{ bits.} \tag{2.4}$$

从式 2.3和式 2.4可以得出：

$$C \leqslant \lceil \log_2(1/\epsilon) + \log_2(2b) \rceil / \alpha. \tag{2.5}$$

当 $b = 4$ 时，此时均摊空间开销约为 $(\log_2(1/\epsilon) + 3)/\alpha$，其中 $\alpha \approx 95\%$。而对于布隆过滤器，其均摊空间开销约为 $1.44 \log_2(1/\epsilon)$。因此，相比于布隆过滤器，布谷

鸟过滤器可以实现更优的均摊空间开销。文献[7]中的实验结果表示，当 b 取 4 的时候，布谷鸟过滤器能在假正例率和空间开销之间取得较好的平衡。

2.1.4　布谷鸟过滤器的优化

由于布谷鸟过滤器使用桶作为存储单元，桶的大小也是影响布谷鸟过滤器性能的重要因素之一。在布谷鸟过滤器原文[7]中就提到可以通过对桶中元素进行排序的方式来进一步降低存储开销。以桶大小 $b=4$，指纹大小 $|f|=4$ 比特为例，在无任何优化的情况下，每个桶中需要存储 $4\times 4=16$ 比特。当桶中的指纹进行排序的话，那么就会出现 3876 种可能性①。因此，每个桶中只需要使用 12 比特的索引 ($2^{12}=4096>3876$) 而不是 16 比特，即为每个指纹节省了 1 比特。在这之后，Breslow 和 Jayasena[14]提出了 Morton 过滤器，通过调整指纹在桶中的分布对存储和插入性能进行了优化。Wang 等人[15]提出了 Vacuum 过滤器，通过将整个过滤器划分成多个大小相同的块，而每个块中桶的个数为 2 的指数，并保证每个元素对应的两个位置都处于同一个块中。通过这样的划分，Vaccum 可以避免布谷鸟过滤器在实际使用中由于桶的个数需要设置为 2 的指数而造成的空间浪费。针对布谷鸟过滤器的优化方案还有很多，这里就不一一列举。这些方案的判断过程与布谷鸟过滤器基本一致，都可以归类为 OR 型过滤器的范畴。

2.2　异或型过滤器

Bloomier 过滤器[16-17] 是首个异或型结构的过滤器。与前面介绍的布隆过滤器和布谷鸟过滤器不同，它并不是用来判断元素是否属于某一集合，而是用来返回元素对应函数值的一种概率型数据结构。从这一角度来看，Bloomier 过滤器的定义更接近于不经意的键值存储（即**定义 1.2**）而非过滤器（即**定义 1.1**）。由于 Bloomier 过滤器不关注其存储的函数 $f(x)$ 是如何定义的，它也被看作是其他过滤器的一种一般化形式[18-19]。严格来说，Bloomier 过滤器的定义与不经意的键值存储还是不同，因为在不经意的键值存储中，要求对于任意 $x'\notin S$ 均返回一个随机结果，但 Bloomier 过滤器要求大概率返回 $\perp$。为了叙述上的统一，本文还是将 Bloomier 过滤器归类为过滤器而非不经意的键值存储。

早期的 Bloomier 过滤器[16] 采用的是两个哈希表的构造。对于给定的元素集合 $S=\{x_1,x_2,\cdots,x_n\}$，Bloomier 过滤器为集合中每一个元素 x_i 通过哈希函数计算出一组位置信息 $\{h_1(x_i),h_2(x_i),\cdots,h_k(x_i)\}$，并通过贪心算法确定出与其他元素均不冲突的位置 $\tau(x_i)$。然后将该位置信息的编码通过异或拆分并记录在第一个哈希表 T_1 的各个位置 $\{h_1(x_i),h_2(x_i),\cdots,h_k(x_i)\}$ 上，将 $f(x)$ 的结果存储在

① 这里将空的条目看作 0，根据重复组合公式，即从 2^4 种不同的数中有重复地取出 4 个进行组合，一共有 $C_{16+4-1}^{4}=3876$ 种可能的情况。

第二个哈希表 T_2 的位置 $\tau(x_i)$ 上。判断时，以输入 x_i 为例，Bloomier 过滤器首先计算出对应的位置信息 $\{h_1(x_i), \cdots, x_n\}$，然后通过将 T_1 上这些位置上对应的值进行异或得到 $\tau(x_i)$ 的编码。如果解码后得到的结果在 T_2 长度范围内，则直接返回所在位置的结果，否则返回 $\perp$。因为该构造需要使用两个哈希表进行存储，且构造需要使用贪心算法，无论在存储方面还是在构建过程中都存在较大的开销。后续的工作[17]通过转换成图的形式，将构建复杂度从原本的 $O(n\log(n))$ 降低为 $\log(n)$。但这些工作为了确保返回的是 $f(x)$，需要在增加额外的信息用于判断 $x' \notin S$ 的情况，在构建效率和存储开销上都不能进一步提高。后续的异或型过滤器[6,18,20]继承了 Bloomier 过滤器中异或操作的思路，但它们只考虑做元素是否属于某一集合的判断，并不考虑返回 $f(x)$ 本身。这些过滤器无论在构建效率上还是在存储开销上都相比 Bloomier 过滤器有极大的提升，以下我们将对它们进行逐一介绍。

2.2.1　异或过滤器

首先介绍异或过滤器 (xor filter)[18]。正如前面所说，异或过滤器继承了 Bloomier 过滤器中异或操作的思想，但它返回的并不是 $f(x)$，而是判断 x 是否属于集合 S（返回 True 或者 False），即符合**定义 1.1**中的描述。异或过滤器的 Construct 过程包含以下步骤：

- 首先选择一个长度为 $\approx 1.23n$ 的数组，并将数组划分成三个相等的区域，即每个区域长度为 $\approx 1.23n/3$。
- 使用三个哈希函数计算出每个元素对应三个区域上的位置。
- 在确定所有元素的位置之后，我们开始统计数组中每一个位置上对应的元素个数。如果找到某个位置上只存在一个元素，那么将该元素压入栈中，并将该元素在数组上的信息全部移除。每次移除一个元素，数组中就有可能出现新的只存在一个元素的位置。
- 循环上一步骤，直到所有元素都压入栈中，否则构建失败，返回 $\perp$。
- 最后只需要将元素从栈中逐个取出，计算该元素对应的三个位置，并将元素对应的指纹信息通过异或拆分成三份放入这三个位置。

按照这样的移除方式，对于每个出栈的元素，可以确保至少有一个位置上为空。因此可以通过为该位置计算出特殊的取值，保证每个元素对应所有位置上的异或结果正好是其指纹值。以元素 x_i 为例，假如它的三个位置分别为 $h_1(x_i)$，$h_2(x_i)$ 和 $h_3(x_i)$，需要构建的过滤器为 F，且 $F[h_3(x_i)]$ 上为空，其它两个位置上已有信息，那么可以通过以下方式进行计算：

$$F[h_3(x_i)] = f(x_i) \oplus F[h_1(x_i)] \oplus F[h_2(x_i)]. \tag{2.6}$$

这里 $f(x_i)$ 表示为 x_i 的指纹值。这样计算也就是为了让 x_i 在 F 上对应三个位置上的值异或后的结果为 $f(x_i)$。图 2.2 给出了一个具体的构造示例。这里我们使用不同颜色对不同区域进行区分。三个元素 a，b，c 在每个区域内都对应一个位置。当按序扫描时，首先发现在位置 6 上只有一个元素 a，因此将 a 移除并入栈。由于 a 移除后，位置 8 上只有一个元素 c，同样将 c 移除并入栈。最后将 b 移除并入栈。在所有元素都压入栈之后，开始依次将元素出栈。对于出栈的元素 b，将 $f(b)$ 拆分成 $[b]_1$，$[b]_2$ 和 $[b]_3$ 三个部分，并存储在元素 b 三个位置。再出栈元素 c，由于 c 对应的位置上只有位置 8 为空，此时只需要根据 $f(c)$ 和不为空位置上的数值计算得到 $[c]_1$，并存储到位置 8 上。最后对于元素 a 也是类似的操作。如此一来，我们就能保证每个元素对应的三个位置上结果的异或正好与元素对应的指纹值相等。

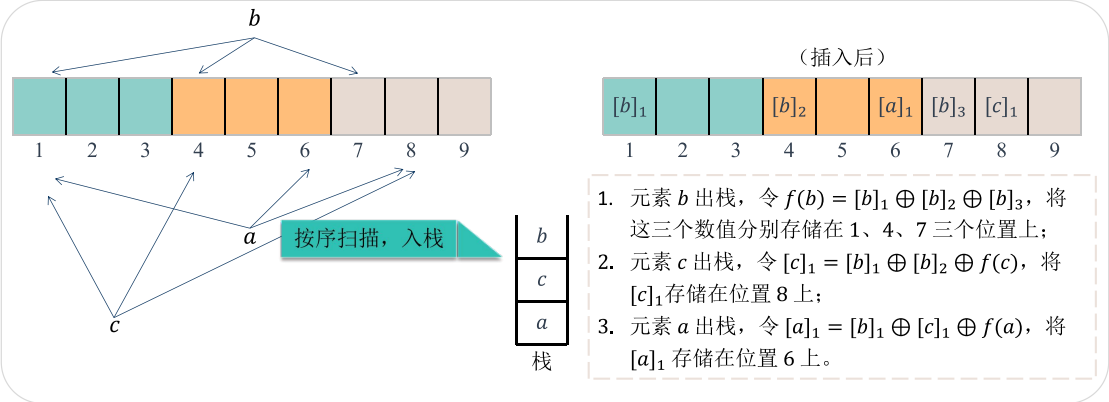

图 2.2　异或过滤器构造过程示例

在理解了异或过滤器的构造之后，它的判断过程也就非常直接。对于任意给定元素 x_i，其 Evaluate 过程如下：

- 首先通过哈希函数计算出它的三个位置信息 $h_1(x_i)$，$h_2(x_i)$ 和 $h_3(x_i)$。
- 将这三个位置上对应的值进行异或，得到 $v = F[h_1(x_i)] \oplus F[h_2(x_i)] \oplus F[h_3(x_i)]$。如果 v 与 $f(x)$ 相等，那么返回 True，否则返回 False。

异或过滤器出现假正例的情况就是出现 $x' \notin S$ 且 Evaluate 结果正好等于 x' 的指纹。也就是说异或过滤器的假正例率与指纹长度有关。假设指纹长度为 $|f(x)|$，那么假正例率 $\epsilon = 1/2^{|f(x)|}$。当集合非常大时，异或过滤器构造成功的概率可以达到 100%。比如当集合大小为 10^7 时，构建成功的概率几乎为 1。对于小集合，文献[18]通过实验发现，当过滤器长度设定为 $1.23n + 32$ 时，成功构建的概率要大于 0.8。

相比布谷鸟过滤器，异或过滤器具有更低的存储开销。为了分析异或过滤器的空间性能，我们还是用 α 表示异或过滤器的负载因子。当过滤器长度设定为 $1.23n + 32$ 时，α 大约为 0.81。在给定指纹长度 $f(x)$ 与负载因子 α 之后，我们可

以得到每个元素均摊的空间开销 C 为

$$C = \frac{\text{过滤器存储大小}}{\text{集合元素的数量}} = \frac{|f(x)| \cdot \text{过滤器条目数}}{\alpha \cdot \text{过滤器条目数}} = \frac{|f(x)|}{\alpha} = \frac{\log_2(1/\epsilon)}{\alpha} \text{ bits.} \quad (2.7)$$

从式 2.7和式 2.5可以看出，当指纹长度和负载因子相同时，异或过滤器的均摊空间要比布谷鸟过滤器更低。在实际中，由于两个过滤器在构造方法上完全不同，它们的负载因子也不能取到相同的结果。比如在布谷鸟过滤器中，负载因子 α 与桶的长度密切相关，$b = 1$ 时，α 仅为 0.5，当 $b = 4$ 时，α 可以达到 0.95。而在异或过滤器中，理想情况下过滤器长度设定为 $1.23n + 32$，即 α 为 0.81。尽管异或过滤器并不能像布谷鸟过滤器那样可以调节 α 的值，但考虑到布谷鸟过滤器是通过扩大桶的容量来增大 α 的值，其均摊存储开销会随着桶的容量增加而增加。就整体均摊存储开销而言，异或过滤器还是要优于布谷鸟过滤器。

相比 Bloomier 过滤器[16]，异或过滤器在构建上更为高效。因为异或过滤器在构造时并不需要使用贪心算法来计算互不冲突的位置，而是直接对元素进行按序扫描，复杂度只与集合元素数量有关。在存储开销方面，异或过滤器由于只需要使用一个哈希表直接记录，相比 Bloomier 过滤器采用的两个哈希表的方式所需存储空间更小。

2.2.2　二进制引信过滤器

尽管异或过滤器在存储开销方面要优于布隆过滤器和布谷鸟过滤器，但它还存在优化的空间。2022 年，异或过滤器的作者 Graf 和 Lemire 在之前的基础上提出了一种新的异或型过滤器，称为二进制引信过滤器 (binary fuse filter)[20]。相比异或过滤器，二进制引信过滤器将存储开销降低了 10% 到 15%。而做到这一点仅仅只需要修改哈希函数的映射方式。

二进制引信过滤器根据使用的哈希函数数量不同又分为 3-wise 二进制引信过滤器和 4-wise 二进制引信过滤器，前者使用 3 个哈希函数，后者使用 4 个哈希函数。这里为了方便介绍以及与异或过滤器进行对比，我们默认哈希函数数量为 3。二进制引信过滤器的构造与异或过滤器类似，其 Construct 过程描述如下：

- 首先选择一个长度为 $\approx 1.125n$ 的数组，并将它划分成若干个区域，每个区域长度为 $2^{\lfloor \log_{3.33} n + 2.25 \rfloor}$。
- 使用三个哈希函数计算出每个元素对应的三个位置。这里要求三个哈希函数映射的三个位置所在区域为连续的三个区域。
- 根据第一个哈希函数映射的位置对元素进行排序，这样一来，第一个元素就应该被映射到前三个区域。按照排序后的顺序，逐个寻找只有一个元素的位置。如果找到，就将该元素压入栈中，并将该元素在数组中的所有信息全部移除。每次移除一个元素，数组中就有可能出现新的只存在一个元

素的位置。

- 循环上一步骤，直到所有元素都压入栈中，否则构建失败，返回 ⊥。
- 最后只需要将元素从栈中逐个取出，计算该元素对应的三个位置，并将元素对应的指纹信息通过异或拆分成三份放入这三个位置。

从构建过程来看，二进制引信过滤器与异或过滤器非常相似。二者主要在映射方式上有所区别。在异或过滤器中，首先会将数组分成三个长度相同的区域，然后使用三个哈希函数将元素映射到这三个区域。而在二进制引信过滤器中，将数组分成的就不是三个区域，而是若干个长度为 $2^{\lfloor \log_{3.33} n+2.25 \rfloor}$ 的区域。在使用哈希函数映射时，要求得到的三个位置对应连续的三个不同区域。仅仅通过调整了哈希函数的映射方式，二进制引信过滤器就能将构造的数组长度从异或过滤器的 $1.23n$ 压缩到 $1.125n$。

二进制引信过滤器的 Evaluate 过程与异或过滤器很类似，也是通过计算三个位置信息，再将数组中这三个位置上的值进行异或，得到结果与所输入的元素指纹做对比，如果相等则返回 True，否则返回 False。

与异或过滤器相同，二进制引信过滤器均摊空间开销 C 也是与假正例率 ϵ 和负载因子 α 有关，即

$$C = \frac{|f(x)|}{\alpha} = \frac{\log_2(1/\epsilon)}{\alpha}. \tag{2.8}$$

但因为二进制引信过滤器采用不同的映射方式，其负载因子 α 要比异或过滤器更大，也就是说当二者指纹函数长度相同时，二进制引信过滤器的均摊开销要比异或过滤器更低。二进制引信过滤器的负载因子取决于使用的哈希函数数量。当使用三个哈希函数时，负载因子理论上大约为 0.88，而当使用四个哈希函数时，负载因子可以提升到 0.93。文献[20]给出了数组长度和每个区域大小的参考公式，如表 2.1所示。

表 2.1　不同哈希函数数量情况下的参数设置

过滤器类型	过滤器长度	每个区域大小
3-wise	$\left\lfloor \left(0.875 + 0.25 \max\left(1, \frac{\log 10^6}{\log n}\right)\right) n \right\rfloor \geqslant \lfloor 1.125n \rfloor$	$2^{\lfloor \log_{3.33} n+2.25 \rfloor} \approx 4.8 \cdot n^{0.58}$
4-wise	$\left\lfloor \left(0.77 + 0.305 \max\left(1, \frac{\log(6 \cdot 10^5)}{\log n}\right)\right) n \right\rfloor \geqslant \lfloor 1.075n \rfloor$	$2^{\lfloor \log_{2.91} n-0.5 \rfloor} \approx 0.7 \cdot n^{0.65}$

2.2.3　缎带过滤器

在二进制引信过滤器提出的几乎同一时间，还有另一个独立工作[6]提出了比异或过滤器更优的异或型过滤器，名为缎带过滤器 (ribbon filter)。与异或过滤器及二进制引信过滤器不同，缎带过滤器通过求解方程组的形式来完成过滤器的构造。这里我们使用 Z 来表示缎带过滤器构造的数组，假设其长度为 m，即此时的负载因子为 $\alpha = n/m$。我们将数组看作 m 长度的向量，假设指纹长度为 r，

那么 Z 就可以看作是 $m \times r$ 的比特矩阵，即 $Z \in \{0,1\}^{m \times r}$。对于每个元素 x_i，假设存在一个哈希函数 h 使得 $h(x)$ 的长度为 m 比特。我们只需要构造这样的 Z，使得对于集合 S 中的每一个元素 x_i 都有 $h(x_i) \cdot Z = f(x_i)$ 成立。这样 Evaluate 也就可以看作计算内积 $h(x_i) \cdot Z$ 并比对结果是否为 $f(x_i)$ 的过程。因此，目前最大的问题就是如何构造这样的 Z，该问题的根本在于哈希函数 h 应该如何设计。

受到文献[21]中所构造的快速高斯消元法的启发，缎带过滤器将哈希函数 h 的计算分为两个部分。首先需要设定一个小于 m 的参数 w，w 也被称为缎带宽度 (ribbon width)。对于给定的元素 x，$h(x)$ 的值取决于一个随机起始位置 $s(x) \in [m - w - 1]$ 和一个长度为 w 比特的随机系数向量 $c(x) \in \{0,1\}^w$。在给定这两个值之后，哈希函数 $h(x)$ 的形式为：

$$h(x) = 0^{s-1} c(x) 0^{m-s-w+1}. \tag{2.9}$$

与文献[6]不同，在缎带过滤器的构造中，需要强制将 $c(x)$ 的第一个比特设为 1[①]。在 Construct 过程中，缎带过滤器需要构造形如 $M \cdot Z = B$ 的方程组，其中 M 为 $m \times m$ 的矩阵，B 为 $m \times r$ 的矩阵。最终求解得到的 Z 就是所构建的过滤器。我们用 $M[i]$ 表示矩阵 M 的第 i 行，在构建之前，缎带过滤器需要初始化全为 0 的 M 和 B 两个矩阵。对于集合 S 中的每一个元素 x，缎带过滤器中 M 和 B 的构造过程如下：

- 根据 x 计算其起始位置 $i \leftarrow s(x)$，哈希函数结果 $c \leftarrow h(x)$，以及指纹函数结果 $b \leftarrow f(x)$。
- 如果 $M[i]$ 为 0^m，则直接将令 $M[i] \leftarrow c$，$B[i] \leftarrow c$，并返回插入成功。否则，计算 $c \leftarrow c \oplus M[i]$，$b \leftarrow b \oplus B[i]$。
- 当 $c = 0^m$ 且 $b = 0^m$ 时，返回重复插入；当 $c = 0^m$ 但 $b \neq 0^m$ 时，返回插入失败；当 $c \neq 0^m$ 时，找到 c 上第一个比特为 1 的位置，记为 i，并返回上一步继续执行。

这里唯一出现插入失败的情况是当 $c = 0^m$ 但 $b \neq 0^m$ 时，换句话说，在这种情况下两个元素产生了哈希碰撞但又具有不同的指纹结果，这样就会导致方程无解，从而导致插入元素失败。当集合 S 中所有元素都成功插入，则开始对方程组 $M \cdot Z = B$ 进行求解，得到的 Z 便是所构建的过滤器。从以上过程也可以发现，这样所构建的矩阵 M 为一个近似于三角矩阵的形式，如图 2.3所示，因此可以直接使用向后替换法来快速对方程求解。

缎带过滤器的 Evaluate 过程就比较直接，对于给定的输入 x，只需要计算 $h(x) \cdot Z$ 的结果是否与 $f(x)$ 相等即可。因为方程中所有元素都是使用二进制形式来表示，实际上这一过程就是将 $h(x)$ 中所有为 1 的位置对应在 Z 上的值进行异

[①]作者表示这样的改变并不会影响解方程的效率，在分析时还是假设 $c(x)$ 为 $\{0,1\}^w$ 上的均匀分布[6]。

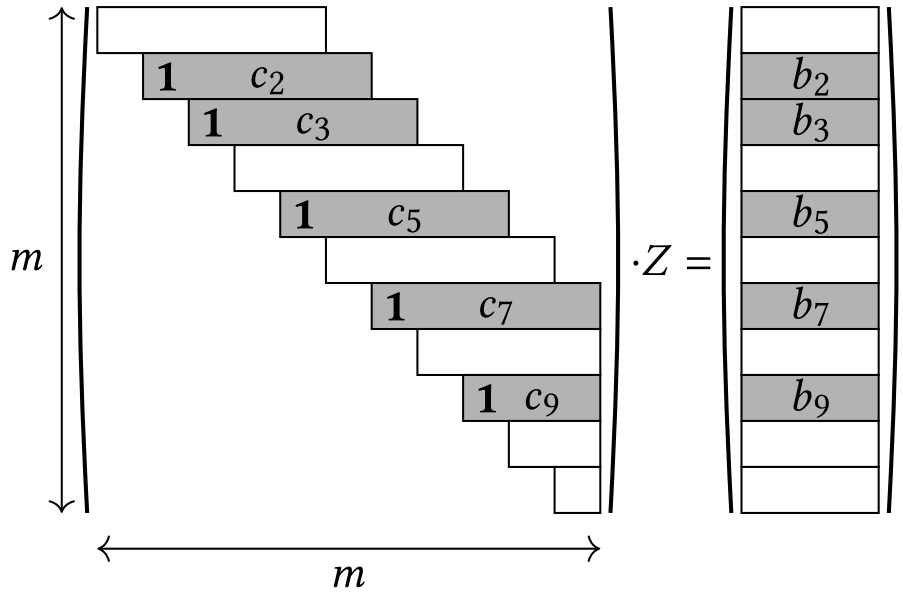

图 2.3　缎带过滤器构造过程中的矩阵示例（来源：文献[6]）

或，如式 2.10所示。而异或过滤器和二进制引信过滤器在 Evaluate 也是类似的方式，只不过它们只需要找到三个（或四个）位置上的值进行异或。但考虑到异或操作本身执行效率非常高，缎带过滤器虽然需要处理更多的异或操作，但实际中这些差距并不明显。

$$h(x) \cdot Z = \bigoplus Z[i], \forall i \in h(x) \text{ and } h(x)[i] = 1. \tag{2.10}$$

缎带过滤器的原文[6]并没有给出理论上的存储开销分析，而是通过一系列实验验证缎带过滤器在存储上的优势。而实验中主要是将异或过滤器和缎带过滤器的两个扩展版本进行对比，这两个扩展版本分别是齐次缎带过滤器 (homogeneous ribbon filter) 和平衡缎带过滤器 (balanced ribbon filter)。从实验结果来看，在相同的假正例率情况下，随着哈希函数中的参数 w 越大，负载因子也随之增大。当 $w = 32$，$\epsilon = 2^{-8}$ 时，齐次缎带过滤器的负载因子与异或过滤器相当。而对于平衡缎带过滤器，当 w 分别取 32 和 64 时，其负载因子可以达到 0.96 甚至 0.99。

2.3　不经意键值存储

不经意键值存储 (Obilivious Key-Value Store，OKVS) 的概念最早是由 Garimella 等人[22]于 2021 年提出的。正如**定义 1.2**中所述，不经意键值存储包含 Encode 和 Decode 两个算法，它主要对键值型数据进行编码。对于编码结果 $\boldsymbol{D}$，当其存储的键值对 $\{k, v\}$ 中 v 为随机的数值，那么 $\boldsymbol{D}$ 会隐藏关于 k 的信息。这种不经意的性质使得不经意键值存储可以应用到隐私集合运算协议中。尽管不经意键值存储的概念提出得比较晚，但相似的数据结构在之前的隐私集合运算协议（如文献[23-24]）也有出现。目前能在存储开销和编解码开销之间取得较

好平衡的方案由 Bienstock 等人[25]于 2023 年提出。在介绍该方案之前，我们首先介绍目前已有的一些构造思路。

2.3.1　构造思路

多项式插值法：为了编码 $\{(k_1, v_1), (k_2, v_2), \cdots, (k_n, v_n)\}$ 这样的键值型数据，最直接的方式是采用多项式插值法。也就是说把 $k_i \mapsto v_i$ 的映射看作坐标 (k_i, v_i)，通过插值构造多项式 p，使得对于所有的 $i \in [1, n]$，$p(k_i) = v_i$ 始终成立。而编码结果就是多项式的系数，系数个数刚好就是插值的坐标数量，因此这种构造方式的负载因子为 1。尽管在存储方面达到最优，但采用多项式插值的方式在编码时需要 $O(n \log^2 n)$ 的复杂度[26]，当编码的键值对数量太大时，编码开销较大。

混淆布隆过滤器 (garbled Bloom filter)[23]：根据表 1.1中的总结，混淆布隆过滤器严格意义上属于异或型过滤器的范畴，因为它存储的是插入元素本身（即 $f(x) = x$），不满足不经意的性质。但如果将它存储的 $f(x)$ 设置为随机值，并将它的输出由 True/False 修改为 $f(x)$，那么它也可以看作是不经意键值存储。混淆布隆过滤器与布隆过滤器类似，都是使用 k 个不同的哈希函数 $h_1, h_2, \cdots, h_t$ 来计算映射位置。对于键值型数据 $\{(x_i, y_i)\}_{i \in [1,n]}$，编码结果记为 $\boldsymbol{D}$，其编码过程如下：

- 使用 k 个哈希函数计算出 x_i 的 k 个位置，如果 $\boldsymbol{D}$ 的 k 个位置上都为空值，则直接将 y_i 拆分成 k 个值的异或，存储在 $\boldsymbol{D}$ 的 k 个位置上。
- 如果 $\boldsymbol{D}$ 的 k 个位置上至少有一个为空，则在不为空的位置共用已有的值，剩下的位置通过异或进行填充，确保这 k 个位置上的异或结果为 y_i。
- 如果 $\boldsymbol{D}$ 的 k 个位置上都已被占用，则返回构造失败。

从编码过程可以看出，混淆布隆过滤器可以看作是将布隆过滤器改造成异或型过滤器。因此混淆布隆过滤器在存储方面性能与布隆过滤器基本一致。根据式 1.2，我们可以推导出在 k 取最优的情况下，$\boldsymbol{D}$ 的长度 $m = 1.44kn$，对应的假正例率为 $1/2^k$。文献[22]指出混淆过滤器编码结果的大小为 $O(kn)$（即负载因子为 $O(1/k)$），这在隐私集合操作相关协议中会造成较大的通信开销。

混淆布谷鸟哈希表 (garbled cuckoo table，GCT)[22,24]：混淆布谷鸟哈希表实际上是 Bloomier 过滤器[17] 的一种变体。在章节 2.2中我们介绍了 Bloomier 过滤器的最初构造方案[16]。在 2008 年，Charles 等人[17]对初始方案进行了改进，将构造过滤器的过程看作是给图的顶点赋值过程。以两个哈希函数 h_1 和 h_2 为例，他们思路是将每个元素 x 对应的 $f(x)$ 看作一条边，$\boldsymbol{D}[h_1(x)]$ 和 $\boldsymbol{D}[h_2(x)]$ 看作边对应的两个顶点。在构造过程中，首先确定整体图的形状，再通过剥离 (peeling) 操作对为顶点赋值。剥离操作依次移除图中度为 1 的节点及它对应的边。如果剥离成功，则按照移除节点的逆序为各节点赋值，使其满足 $f(x) = \boldsymbol{D}[h_1(x)] \oplus \boldsymbol{D}[h_2(x)]$。

如果图中剩余节点，意味图中出现了环，则需要重新选择哈希函数，直到能够剥离成功为止。混淆布谷鸟哈希表主要在此基础上进行了两方面的改进，一是混淆布谷鸟哈希表中使用的哈希函数一开始就会确定，不会因为环的出现而重新选择；二是当出现环的情况，混淆布谷鸟哈希表采用联立方程组的方式，利用高斯消元法对环中节点进行求解。根据使用的哈希函数个数，又分为 2H-GCT[24] 和 3H-GCT[22]两种构造，即分别使用 2 个哈希函数和 3 个哈希函数。根据文献[25]中的实验结果，2H-GCT 的负载因子约为 0.4，而 3H-GCT 在 0.77 到 0.81 之间。

高斯消元法：当我们将编码结构 $\boldsymbol{D}$ 看作长度为 m 的向量时，那么构建 $\boldsymbol{D}$ 的过程就可以看作解线性方程组。以键值数据 $\{(k_i, v_i)\}_{i\in[n]}$ 为例，假设存在一个随机映射 $r_{\mathcal{F}} : \mathcal{K} \to \{0,1\}^m$，可以将所有的键 $k \in \mathcal{K}$ 映射为长度为 m 的向量。那么我们就可以构造 $n \times m$ 的矩阵 $\boldsymbol{M}$：

$$\boldsymbol{M} = \begin{bmatrix} r_{\mathcal{F}}(k_1)^\intercal \\ r_{\mathcal{F}}(k_2)^\intercal \\ \cdots \\ r_{\mathcal{F}}(k_n)^\intercal \end{bmatrix}. \tag{2.11}$$

而对于值 $v = [v_1, v_2, \cdots, v_n]^\intercal$ 则可以看作长度为 n 的向量。此时，只需要对方程 $\boldsymbol{M} \cdot \boldsymbol{D} = v$ 使用高斯消元法求解，得到的 $\boldsymbol{D}$ 就是我们需要的编码结果。文献[22]给出了使用随机矩阵的构造方式。为了确保方程有解，要求 $m = n + O(\log n)$，在实际中的编码过程复杂度为 $O(n^3)$。这种方式编码开销甚至超过了基于多项式插值的方法，计算成本太大。文献[27]通过使用矩阵变换的方式将 $\boldsymbol{M}$ 转换成三角矩阵，从而加速高斯消元的过程。该方案的存储开销与 3H-GCT 基本相同，负载因子大约在 0.78 到 0.81 之间。最近的一篇工作[25]采用随机带状矩阵 (Random Band Matrix) 进一步了降低存储开销，最优情况下，负载因子可以达到 0.97。

表 2.2　基于不同构造的 OKVS 性能对比（编码失败概率：$2^{-\lambda}$）

OKVS 类型	负载因子	编码复杂度	解码复杂度
多项式	1	$O(n\log^2 n)$	$O(n)$
随机矩阵[22]	1	$O(n^3)$	$O(n)$
混淆布隆过滤器[23]	$O(1/\lambda)$	$O(n\lambda)$	$O(n)$
2H-GCT[24]	0.4	$O(n\lambda)$	$O(\lambda)$
3H-GCT[22]	$0.77-0.81$	$O(n\lambda)$	$O(\lambda)$
矩阵三角化算法[27]	$0.78-0.81$	$O(n\lambda)$	$O(\lambda)$
随机带状矩阵[25]	$0.91-0.97$	$O(n\lambda)$	$O(\lambda)$

表 2.2 对现有构造的存储复杂度和编解码复杂度进行了总结。从表中我们可以看出，目前综合性能最优的是基于随机带状矩阵构造的方案[25]，即随机带状不经意键值存储 (Random Band Oblivious Key-Value Store，RB-OKVS)。下面将具体介绍它的构造方式。

2.3.2　随机带状不经意键值存储

随机带状不经意键值存储 (Random Band Oblivious Key-Value Store，RB-OKVS) 的构造是基于高斯消元法。正如前文所述，重点是如何构建矩阵 $\boldsymbol{M}$ 使得方程 $\boldsymbol{M} \cdot \boldsymbol{D} = v$ 能快速求解。构建 $\boldsymbol{M}$ 的核心在于映射函数 $r_{\mathcal{F}}$ 的设计。在 RB-OKVS 中，$r_{\mathcal{F}}$ 的构造方法与前面介绍的缎带过滤器[6]非常类似，都需要提前设置一个小于 m 的参数 w RB-OKVS 的编码需要使用两个哈希函数 h_1 和 h_2，其中 h_1 将任意的 $k \in \mathcal{K}$ 映射到 $\{1, 2, \cdots, m-w\}$ 中的数值，h_2 将任意的 $k \in \mathcal{K}$ 映射为长度为 w 的向量。对于 k_i 来说，其对应的映射结果为：

$$r_{\mathcal{F}}(k_i) = 0^{h_1(k_i)-1} h_2(k_i) 0^{m-w-h_1(k_i)+1}. \tag{2.12}$$

对于输入的键值型数据 $\{(k_i, v_i)\}_{i\in[1,n]}$，RB-OKVS 的 Encode 过程描述如下：

- 使用上述的映射函数 $r_{\mathcal{F}}$ 对每一个 k_i 进行计算，得到结果形成矩阵 $\boldsymbol{M}$，将所有 v_i 组成的向量记作 v。
- 求解方程 $\boldsymbol{M} \cdot \boldsymbol{D} = v$：首先根据 $h_1(k_i)$ 的大小对 $\boldsymbol{M}$ 和 v 重新排序，再直接使用向后替换法进行消元，完成对 $\boldsymbol{D}$ 的求解。

因为哈希函数 h_1 的值决定了矩阵 $\boldsymbol{M}$ 中行向量不为 0 的起始位置，所以只需要根据 h_1 的值进行排序，便可以快速得到近似于三角矩阵的形式，从而可以使用后向替换法快速消元。这种构造形式与缎带过滤器如出一辙，二者不同之处在于缎带过滤器没有使用排序而是在插入时排列。为了保证不经意性，编码结果 $\boldsymbol{D}$ 中的自由变量均为随机值。

RB-OKVS 的 Decode 过程与缎带过滤器类似，对于需要查询的键 x，只需要使用 h_2 映射得到的长度为 w 的向量与编码结果 $\boldsymbol{D}$ 中对应位置的向量做内积运算，即：

$$\sum_{i=1}^{w} h_2(x)[i] \cdot \boldsymbol{D}[h_1(x) + i - 1]. \tag{2.13}$$

在 RB-OKVS 中，通常假设矩阵都为二进制形式。这样无论是在编码过程还是解码过程，因为大部分操作都是异或操作，索引 RB-OKVS 在实际应用中具有非常高的计算效率。编码过程分为排序和解方程两个步骤，分别需要 $O(n)$ 和 $O(nw)$ 的时间。解码过程只需要执行两个长度为 w 向量的内积，复杂度为 $O(w)$。假定编码失败的概率为 $2^{-\lambda}$，文献[28]通过实验给出了对于不同 n 和 α 取值的情况下，λ 与 w 之间的关系。比如当 $n = 2^16$，$\alpha = 0.97$ 时，λ 与 w 的关系为：$\lambda = 0.08241w - 7.023$。相比现有的 3H-GCT[22]和基于矩阵三角化的方案[27]，RB-OKVS 具有更低的存储开销。

2.4　总结

本章介绍了许多由布隆过滤器衍生的数据结构，这些数据结构本质上与布隆过滤器相同，都属于哈希表（或者说数组）类型。与布隆过滤器不同的是，这些衍生的数据结构在每个位置上存储的不再是单个 0/1 比特，而是有一定长度的 0/1 比特串。在相同的假正例率的情况下，当存储的内容不再是单个比特时，对应数据结构的存储空间利用率也就会相应提高。

布谷鸟过滤器是 OR 型过滤器的典型代表，它通过两个哈希函数计算元素的位置信息，每个位置上使用桶放置指纹。对于每个元素，需要保证至少有一个位置上存储着该元素对应的指纹信息。如果每个桶只能容纳一个指纹的话，布谷鸟过滤器的负载因子只有 0.5，在存储开销上并没有明显优势。但是当将桶的容量扩展为可以存储 4 个指纹时，负载因子可以提高到 0.95。布谷鸟过滤器的性能与桶的大小密切相关，已有的优化工作[14-15]都是从优化桶的角度来提升过滤器整体性能。由于布谷鸟过滤器存储的是完整的指纹信息，所以自然支持元素的动态插入和删除，这也是 OR 型过滤器相比其他过滤器来说最大的优势。

与 OR 型过滤器存储完整的指纹信息不同，XOR 型过滤器是将指纹信息（或 $f(x)$）拆分成若干份存放在不同的位置。因此 XOR 型过滤器在构造之后便不能执行插入和删除操作，也就是说 XOR 型过滤器针对的是不可变集合 (immutable sets)。对于所有的 XOR 型过滤器，它们在判断阶段的过程基本相同，即首先通过哈希函数计算出输入元素 x 的位置，再将这些位置上存储的值进行异或。如果异或结果为 $f(x)$，则返回 True。难点在于构造方式，而构造关键在于如何找到每个元素对应的 “独占位置”。Bloomier 过滤器提出使用贪心算法来确定每个元素对应的互不冲突位置，但这种方案复杂度太高，而且存储复杂度也太高。后续的异或过滤器和二进制引信过滤器不要求每个元素都存在互不冲突的位置，而是让每个元素当它在插入时存在互不冲突的位置。通过这种思路，异或过滤器和二进制引信过滤器采用按序扫描入栈再反向出栈的方式完成构造，实现了更优的计算和存储开销。后续的缎带过滤器也是采取这种思路，但它是基于高斯消元法来构造，相比异或过滤器进一步提高了性能。

不经意键值存储的概念脱胎于隐私集合求交协议，但它后续工作的构造思路可以说与 XOR 型过滤器殊途同归。许多文献[22,25,27]将混淆布隆过滤器看作是一种不经意键值存储结构，但实际上它更符合我们对 XOR 型过滤器的定义。尽管出发点不同，但是不经意键值存储和 XOR 型过滤器在设计思路上有相似的地方。单从构造来看，3H-GCT[22]可以对应到异或过滤器[18]，RB-OKVS[25]可以对应到缎带过滤器[6]。究其根源，它们都受到之前相关工作的启发。比如异或过滤器[18]和 3H-GCT[22]都受到 Botelho 等人[28-29]所提出的超图 (hypergraph) 构造的

启发。缎带过滤器[6]和 RB-OKVS[25]中的高斯消元法实际上源于同一篇快速消元法的工作[21]。从中我们也可以发现，这些想法都不是凭空出现的，有时我们需要 “站在巨人的肩膀上”，这样才能 “看得更远”。

第 3 章　在隐私保护上的应用

这一章我们主要介绍布隆过滤器及其衍生数据结构在隐私保护相关协议中的应用。其中涉及到的协议包括可搜索加密、隐私信息检索和隐私集合运算。

3.1　在对称可搜索加密方面的应用

3.1.1　背景介绍

随着近年来数据规模的不断增大，越来越多的个人和企业选择将本地文件外包到云平台（如 iCloud、Amazon S3）进行存储。存储在云端的文件不仅为用户节省了本地存储所需要的成本，避免文件丢失的风险，还能让用户随时随地通过互联网对文件进行搜索和访问，极大地提高了便利性。但是将文件直接存放在云服务器中也大大增加文件泄露的风险。一方面云服务提供者可以直接获取文件，另一方面由于云服务器处在公开的网络环境中，很容易受到外部攻击者的攻击。一旦文件遭到泄露，用户的隐私也受到威胁。保护用户文件隐私的直接方式是将文件在本地进行加密，再将加密后的文件进行上传。但是服务器无法在加密后的文件上执行搜索，用户需要搜索时只能把所有文件下载下来才能完成，这就丧失了将文件存储在云端的意义。

为了解决文件隐私和可搜索之间的矛盾，对称可搜索加密 (Searchable Symmetric Encryption，SSE)[30-31] 的概念被提出。对称可搜索加密通过为加密数据构造安全索引实现隐私保护的关键词搜索。如图所示，对称可搜索加密中包含用户和服务器两个实体。在上传阶段，用户不仅需要上传加密文件，还需要上传对应的安全索引。在搜索阶段，用户根据需要搜索的关键词生成搜索令牌 (tokens)，服务器使用搜索令牌检索得到加密的文件标识并返回给用户。

我们假设服务器是半诚实的 (semi-honest)，即服务器会诚实地执行协议，但它同时会对尝试分析输入输出信息来推断用户的隐私。相比其他隐私保护的搜索方案，对称可搜索加密方案能在效率和安全之间取得更好的平衡。一方面，基于属性保留加密 (Property-Preserving Encryption) 的方案[32] 可以直接保留密文中的相等关系，从而实现高效的搜索。但服务器可以通过分析密文上的相等信息来执行频率统计攻击并还原明文信息[33]。在另一方面，基于通用密码学工具（如同态加密、安全多方计算以及不经意随机访问机）虽然能够提供较强的安全性，但这些工具要么在计算上开销非常大，要么存在较大的通信开销，直接应用到加密搜索场景会面临效率问题[34]。而对称可搜索加密通过将搜索过程转移到安全索引上，在允许有限信息泄露的同时提供了高效的搜索。

对称可搜索加密允许泄露的信息也被称作模式信息 (pattern information)，这些信息包括：

- 搜索模式 (search pattern)，即两次搜索是否包含相同的搜索关键词。
- 访问模式 (access pattern)，即每次搜索能匹配到哪些加密结果。
- 数量模式 (volume pattern)，即每次搜索返回的结果数量。

对称可搜索加密协议的安全性是定义在给定的模式信息之上的，也就是说如果我们称一个对称可搜索加密协议是安全的，那么除了允许泄露的模式信息之外，它不会泄露其他的任何信息。近些年有大量工作[35-38]集中关注于如何利用这些模式信息来设计相应的攻击，这些攻击被统称为泄露滥用攻击 (Leakage-Abuse Attacks, LAAs)。而我们前面的介绍的过滤器及其衍生数据结构正好可以用来隐藏特定的模式信息，从而避免受到对应的攻击。以下我们将给出两个具体例子，介绍这些数据结构是如何用到对称可搜索加密之中的。

3.1.2　隐藏中间结果模式的 SSE

这一节我们介绍的是
HXT[39]

3.1.3　隐藏数量模式的 SSE

3.2　在隐私信息检索方面的应用

3.2.1　背景介绍

3.2.2　方案介绍

3.3　在隐私集合运算方面的应用

3.3.1　背景介绍

3.3.2　隐私集合求交的协议

3.3.3　隐私集合求并的协议

3.4　总结

参 考 文 献

[1] BLOOM B H. Space/time trade-offs in hash coding with allowable errors[J]. Communications of the ACM, 1970, 13(7): 422-426.

[2] GERAVAND S, AHMADI M. Bloom filter applications in network security: A state-of-the-art survey[J]. Computer Networks, 2013, 57(18): 4047-4064.

[3] LUO L, GUO D, MA R T B, et al. Optimizing Bloom filter: Challenges, solutions, and comparisons[J]. IEEE Communications Surveys & Tutorials, 2019, 21(2): 1912-1949.

[4] BOSE P, GUO H, KRANAKIS E, et al. On the false-positive rate of Bloom filters[J]. Information Processing Letters, 2008, 108(4): 210-213.

[5] CHRISTENSEN K, ROGINSKY A, JIMENO M. A new analysis of the false positive rate of a Bloom filter[J]. Information Processing Letters, 2010, 110(21): 944-949.

[6] DILLINGER P C, WALZER S. Ribbon filter: Practically smaller than Bloom and Xor: arXiv:2103.02515[M]. arXiv, 2021.

[7] FAN B, ANDERSEN D G, KAMINSKY M, et al. Cuckoo filter: Practically better than bloom [C]//Proceedings of the 10th ACM International on Conference on Emerging Networking Experiments and Technologies (CONEXT). ACM, 2014: 75-88.

[8] 张响鸽, 张聪, 刘巍然, 等. 隐私集合运算中的关键数据结构研究[J]. 密码学报, 2024, 11 (2): 263-281.

[9] Li Fan, Pei Cao, ALMEIDA J, et al. Summary cache: A scalable wide-area Web cache sharing protocol[J]. IEEE/ACM Transactions on Networking, 2000, 8(3): 281-293.

[10] MITZENMACHER M. Compressed Bloom filters[J]. IEEE/ACM Transactions on Networking, 2002, 10(5): 604-612.

[11] BONOMI F, MITZENMACHER M, PANIGRAHY R, et al. An improved construction for counting Bloom filters[C]//Proceedings of the 14th European Symposium on Algorithms (ESA): Vol. 4168. Springer, 2006: 684-695.

[12] PUTZE F, SANDERS P, SINGLER J. Cache-, hash-, and space-efficient Bloom filters[J]. ACM Journal of Experimental Algorithmics, 2009, 14(4): 4.4-4.18.

[13] PAGH R, RODLER F F. Cuckoo hashing[J]. Journal of Algorithms, 2004, 51(2): 122-144.

[14] BRESLOW A D, JAYASENA N S. Morton filters: Fast, compressed sparse cuckoo filters[J]. The VLDB Journal, 2020, 29(2): 731-754.

[15] WANG M, ZHOU M, SHI S, et al. Vacuum filters: More space-efficient and faster replacement for Bloom and cuckoo filters[J]. Proceedings of the VLDB Endowment, 2019, 13(2): 197-210.

[16] CHAZELLE B, KILIAN J, RUBINFELD R, et al. The Bloomier filter: An efficient data

structure for static support lookup tables[C]//Proceedings of the 15th ACM-SIAM Symposium on Discrete Algorithms (SODA). SIAM, 2004: 30-39.

[17] CHARLES D, CHELLAPILLA K. Bloomier filters: A second look[C]//Proceedings of the 16th European Symposium on Algorithms (ESA). Springer, 2008: 259-270.

[18] GRAF T M, LEMIRE D. Xor filters: Faster and smaller than bloom and cuckoo filters[J]. ACM Journal of Experimental Algorithmics, 2020, 25: 1.5:1-1.5:16.

[19] LI H, WANG L, CHEN Q, et al. ChainedFilter: Combining membership filters by chain rule [J]. Proceedings of the ACM on Management of Data, 2023, 1(4): 234:1-234:27.

[20] GRAF T M, LEMIRE D. Binary fuse filters: Fast and smaller than xor filters[J/OL]. ACM Journal of Experimental Algorithmics, 2022, 27(1.5): 1-15. DOI: 10.1145/3510449.

[21] DIETZFELBINGER M, WALZER S. Efficient gauss elimination for near-quadratic matrices with one short random block per row, with applications[C]//Proceedings of the 27th European Symposium on Algorithms (ESA). Springer, 2019: 39:1-39:18.

[22] GARIMELLA G, PINKAS B, ROSULEK M, et al. Oblivious key-value stores and amplification for private set intersection[C]//Proceedings of the 41st Annual Cryptology Conference (CRYPTO). Springer, 2021: 395-425.

[23] DONG C, CHEN L, WEN Z. When private set intersection meets big data: An efficient and scalable protocol[C]//Proceedings of the 20th ACM Conference on Computer & Communications Security. ACM, 2013: 789-800.

[24] PINKAS B, ROSULEK M, TRIEU N, et al. PSI from PaXoS: Fast, malicious private set intersection[C]//Proceedings of the 39th Annual International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT). Springer, 2020: 739-767.

[25] BIENSTOCK A, PATEL S, SEO J Y, et al. Near-optimal oblivious key-value stores for efficient PSI, PSU and volume-hiding multi-maps[C]//Proceedings of the 32nd USENIX Security Symposium (USENIX Security). USENIX Association, 2023.

[26] MOENCK R, BORODIN A. Fast modular transforms via division[C]//Proceedings of the 13th Annual Symposium on Switching and Automata Theory (SWAT). 1972: 90-96.

[27] RAGHURAMAN S, RINDAL P. Blazing fast PSI from improved OKVS and subfield VOLE [C]//Proceedings of the 29th ACM Conference on Computer and Communications Security (CCS). ACM, 2022: 2505-2517.

[28] BOTELHO F C, PAGH R, ZIVIANI N. Practical perfect hashing in nearly optimal space[J]. Information Systems, 2013, 38(1): 108-131.

[29] BOTELHO F C, PAGH R, ZIVIANI N. Simple and space-efficient minimal perfect hash functions[C]//Proceedings of the 10th Workshop on Algorithms and Data Structures (WADS). Springer, 2007: 139-150.

[30] SONG D X, WAGNER D, PERRIG A. Practical techniques for searches on encrypted data [C]//Proceedings of the 2000 IEEE Symposium on Security and Privacy (S&P). IEEE, 2000: 44-55.

[31] CURTMOLA R, GARAY J, KAMARA S, et al. Searchable symmetric encryption: Improved definitions and efficient constructions[C]//Proceedings of the 13th ACM Conference on Computer and Communications Security (CCS). ACM, 2006: 79-88.

[32] BELLARE M, BOLDYREVA A, O'NEILL A. Deterministic and efficiently searchable encryption[C]//Proceedings of the 27th International Cryptology Conference (CRYPTO). Springer, 2007: 535-552.

[33] NAVEED M, KAMARA S, WRIGHT C V. Inference attacks on property-preserving encrypted databases[C]//Proceedings of the 22nd ACM Conference on Computer and Communications Security (CCS). ACM, 2015: 644-655.

[34] REN K, WANG C. Searchable encryption: From concepts to systems[M]. Springer, 2023: 149-152.

[35] CASH D, GRUBBS P, PERRY J, et al. Leakage-abuse attacks against searchable encryption [C]//Proceedings of the 22nd ACM Conference on Computer and Communications Security (CCS). ACM, 2015: 668-679.

[36] BLACKSTONE L, KAMARA S, MOATAZ T. Revisiting leakage abuse attacks[C]// Proceedings of the 27th Annual Network and Distributed System Security Symposium (NDSS). ISOC, 2020.

[37] NING J, HUANG X, POH G S, et al. LEAP: Leakage-abuse attack on efficiently deployable, efficiently searchable encryption with partially known dataset[C]//Proceedings of the 28th ACM Conference on Computer and Communications Security (CCS). ACM, 2021: 2307-2320.

[38] KAMARA S, KATI A, MOATAZ T, et al. SoK: Cryptanalysis of encrypted search with LEAKER –a framework for LEakage AttacK Evaluation on Real-world data[C]//Proceedings of the 7th IEEE European Symposium on Security and Privacy (EuroS&P). IEEE, 2022: 90-108.

[39] LAI S, PATRANABIS S, SAKZAD A, et al. Result pattern hiding searchable encryption for conjunctive queries[C]//Proceedings of the 25th ACM Conference on Computer and Communications Security (CCS). ACM, 2018: 745-762.