中国科学技术大学

读书报告

布隆过滤器及其衍生数据结构在隐私保护协议中的应用

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：　二〇二四年十月十日

目　录

第 1 章　简　　介

1.1　布隆过滤器

在构造网络安全相关协议时，我们经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[2-3]。布隆过滤器是由 k 个哈希函数构造的数组结构，它的作用是快速判断元素是否属于某一集合 (membership query)，具有空间效率高、判断速度快的特点。布隆过滤器的结构如图 1.1 所示，数组中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，

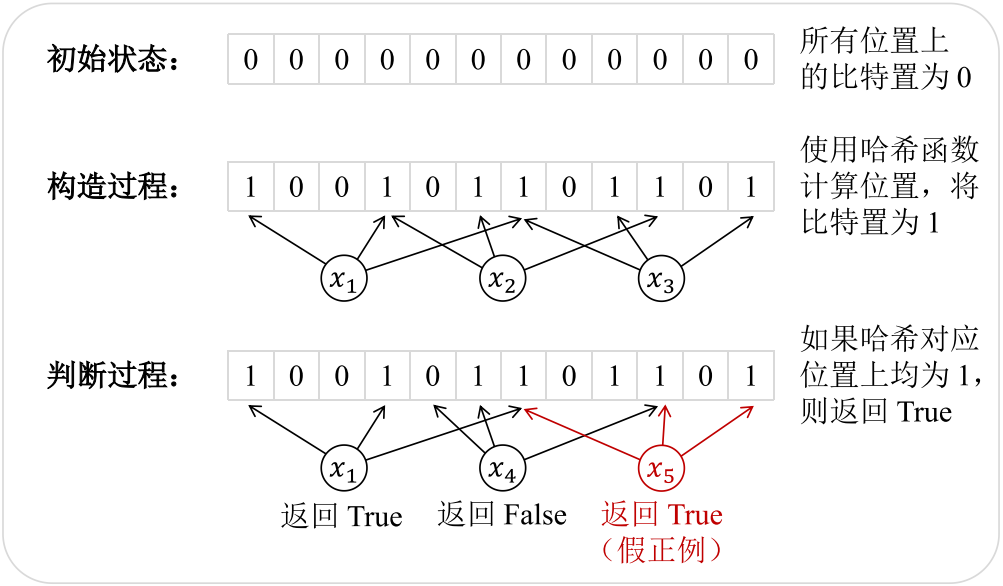


图 1.1　布隆过滤器示例（$k = 3$）

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特定位置的映射。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[4]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 $\epsilon \approx 2^{-k} \approx 0.6185^{m/n}$，过滤器的长度 $m \approx 1.44n \log_2 (1/\epsilon)$。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[5-6]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本身的大小无关，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 0/1 比特，所以过滤器存储大小也就是 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，所以检查一个元素的时间复杂度为 $O(k)$，与集合中元素的数量无关。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，尤其当元素数量较多的情况下优势更为明显。
- **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，并不支持检索功能。

1.2　定义及分类

在介绍布隆过滤器衍生的新型数据结构之前，我们需要对所介绍的数据结构进行分类。为此，我们首先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 $x \in S$，$\Pr[\mathsf{Evaluate}(x, \mathcal{H}, F) = \text{True}] = 1$ 始终成立；对于任意 $x' \notin S$，概率 $\Pr[\mathsf{Evaluate}(x', \mathcal{H}, F) = \text{True}]$ 为可忽略不计的。

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，即过滤器中会以极小的概率出现假正例。

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)，根据探测方式可以将过滤器分为以下三种类型[7]：

- **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)[8]，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)[8]，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，这类构造通常不支持动态更新。

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)[9]，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[10]，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。按照这种思路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 $\{(x_i, y_i)\}_{i\in[1,n]} \subseteq \mathcal{K} \times \mathcal{V}$，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 $\{y_i\}_{i\in[1,n]} \leftarrow \mathcal{V}$，若 $D \neq \perp$，则分布 $\{D | y_i \leftarrow \mathcal{V}, i \in [1, n], \mathsf{Encode}(\{(x_1^0, y_1), (x_2^0, y_2), \cdots, (x_n^0, y_n)\})\}$ 与分布 $\{D | y_i \leftarrow \mathcal{V}, i \in [1, n], \mathsf{Encode}(\{(x_1^1, y_1), (x_2^1, y_2), \cdots, (x_n^1, y_n)\})\}$ 计算不可区分。

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 $A = \{(x_i, y_i)\}_{i\in[1,n]} \subseteq \mathcal{K} \times \mathcal{V}$ 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。

我们主要关注这些数据结构在存储和计算方面的性能。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，我们主要关注构建/判断（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。在接下来的章节，我们将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

探测类型	名称	$f(x)$	负载因子
AND 型	布隆过滤器	1	$0.7k_{opt}^{-1}$
OR 型	布谷鸟过滤器[8]	x 的指纹值	0.5 – 0.95
XOR 型	异或过滤器[11]	x 的指纹值	0.81
	二进制引信过滤器[12]	x 的指纹值	0.88 – 0.93
	缎带过滤器[7]	x 的指纹值	0.96 – 0.99
	随机带状不经意键值存储[13]	键 x 对应的值 y	0.91 – 0.97

1.3　小结

布隆过滤器是一种近似成员查询过滤器 (approximate membership query filter)，即以一定的假正例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[14]。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)[15]，压缩布隆过滤器 (compressed Bloom filter)[16]，d-left 计数式布隆过滤器[17]，块布隆过滤器 (blocked Bloom filter)[18]等。这些变体本质都是在布隆过滤器构造的基础上提出的各种优化，关于这些结构的介绍不属于本文的讨论范畴。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 章对这些衍生新型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协议上的应用。

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三种新型数据结构，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[8] 于 2014 年提出的，其构造方式受到了布谷鸟哈希表 (cuckoo hash table)[19] 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

所谓的哈希表可以看作一个由多个桶 (bucket) 组成的数组结构。元素在数组中的位置由哈希函数决定，而桶的大小则决定了每个位置上能容纳的元素数量。布谷鸟哈希表与一般形式的哈希表最大的不同在于插入元素的方式。在布谷鸟哈希表中，桶的大小设为 1，即每个位置上只能容纳一个元素。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，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

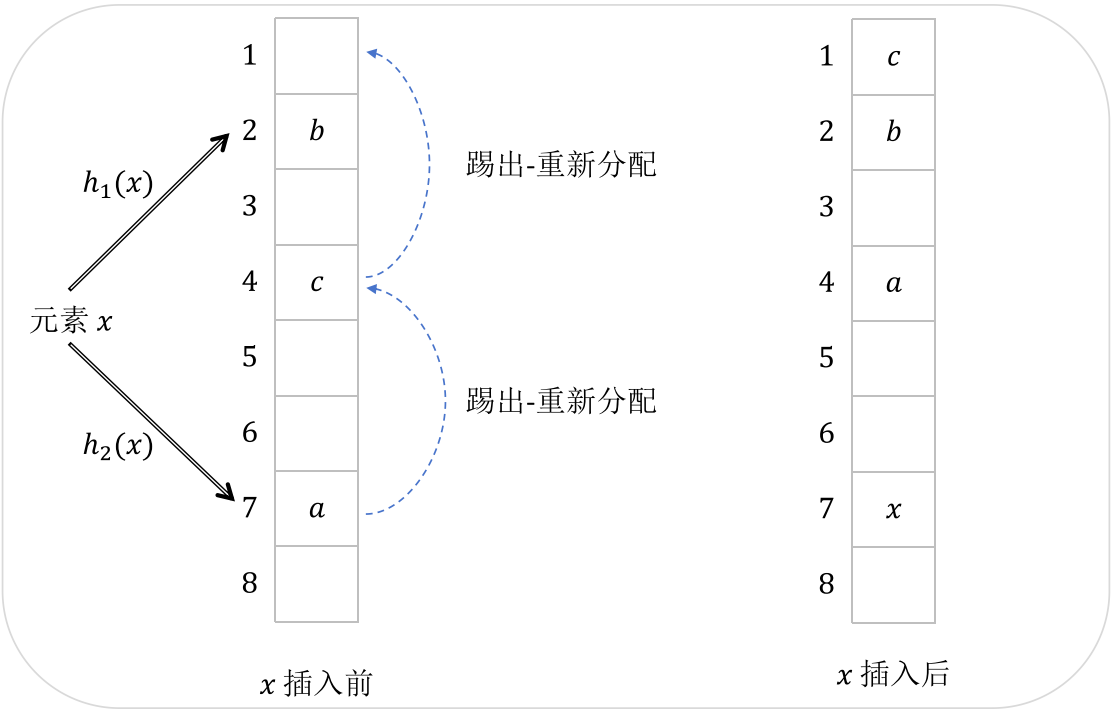


图 2.1　布谷鸟哈希表示例

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，并从该位置上踢出一个指纹，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，当踢出的指纹次数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，也是比较直接，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及优化

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 $(0 < \alpha \leqslant 1)$ 来表示布谷鸟过滤器的空间利用率，它的定义是过滤器中存储元素的数量 n 与过滤器长度 m 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 0.5，而当 $b = 4$ 或 $b = 8$ 时，负载因子随之增长为 0.95 和 0.98。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[8]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：

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

当 $b=4$ 时，此时均摊空间开销约为 $(\log_2(1/\epsilon)+3)/\alpha$，其中 $\alpha\approx 0.95$。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[8]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。

由于布谷鸟过滤器使用桶作为存储单元，桶的大小便成为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[8]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，那么就只有 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，平均为每个指纹节省了 1 比特。在这之后，Breslow 和 Jayasena[20]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[21]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 过滤器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。

2.2　异或型过滤器

Bloomier 过滤器[10,22] 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**定义 1.2** 中的不经意键值存储，而非**定义 1.1** 中过滤器。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[11,23]。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键值存储。

早期的 Bloomier 过滤器[10] 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 k 个位置信息 $\{h_1(x_i),h_2(x_i),\cdots,h_k(x_i)\}$，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 $C^4_{2^4+4-1}=3876$ 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 $\{h_1(x_i), \cdots, h_k(x_i)\}$，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[22]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[7,11-12]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，以下我们将对它们逐一进行介绍。

2.2.1　异或过滤器

首先介绍的是异或过滤器 (xor filter)[11]。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值正好与元素对应的指纹值相等。

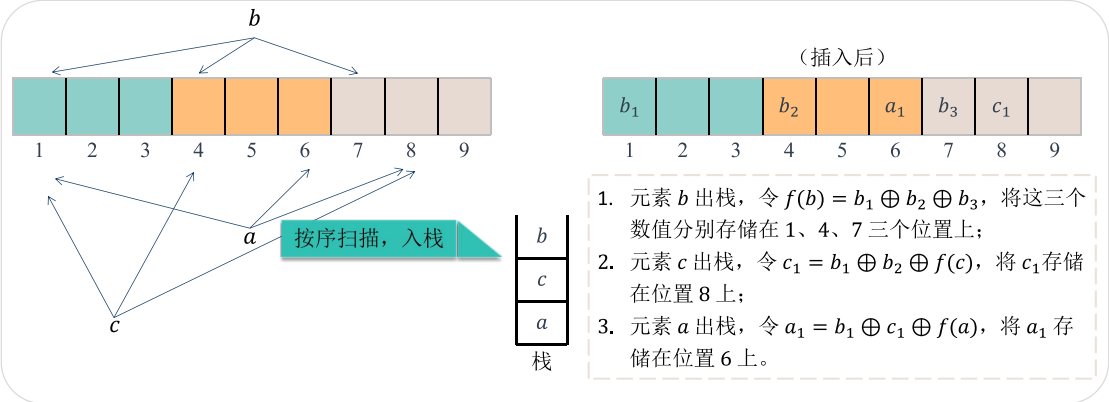


图 2.2　异或过滤器构造过程示例

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 $|f|$，那么假正例率 $\epsilon = 2^{-|f|}$。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[11]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 $|f|$ 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

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

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开销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[10]，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)[12]。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 $1.125n$。图 2.3 描述了二进制引信过滤器中哈希函数的映射方式。直观上看，这种方式可以让过滤器两端更容易出现只有一个元素的位置（如元素 a 和 c），而当两端元素被移除后，又会暴露出新的两端（如移除元素 a 后，左端元素为 b），依次可以不断移除直到所有元素都压入栈中。这种移除元素的过程就像将 “引信” 的一端点燃产生的连锁反应，因此得名引信过滤器。而其中的所有操作都是二进制的异或操作，故称为二进制引信过滤器。

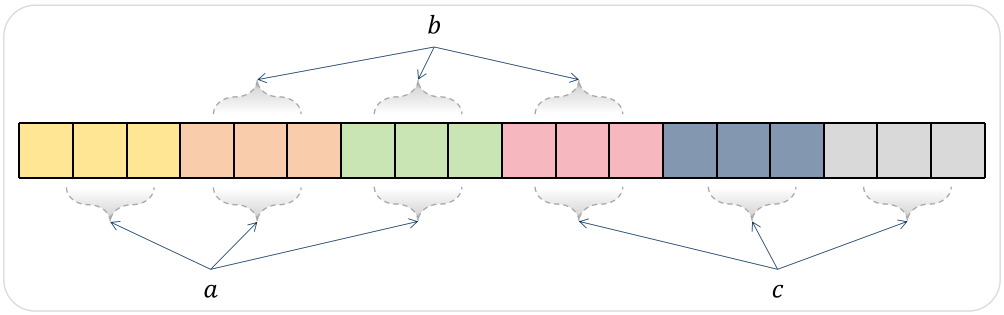


图 2.3　二进制引信过滤器哈希函数的映射方式

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

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

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[12]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[7]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[24]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：

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

与文献[24]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

[①]原文认为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[7]。

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 2.4 所示，因此可以直接使用向后替换法来快速对方程求解。

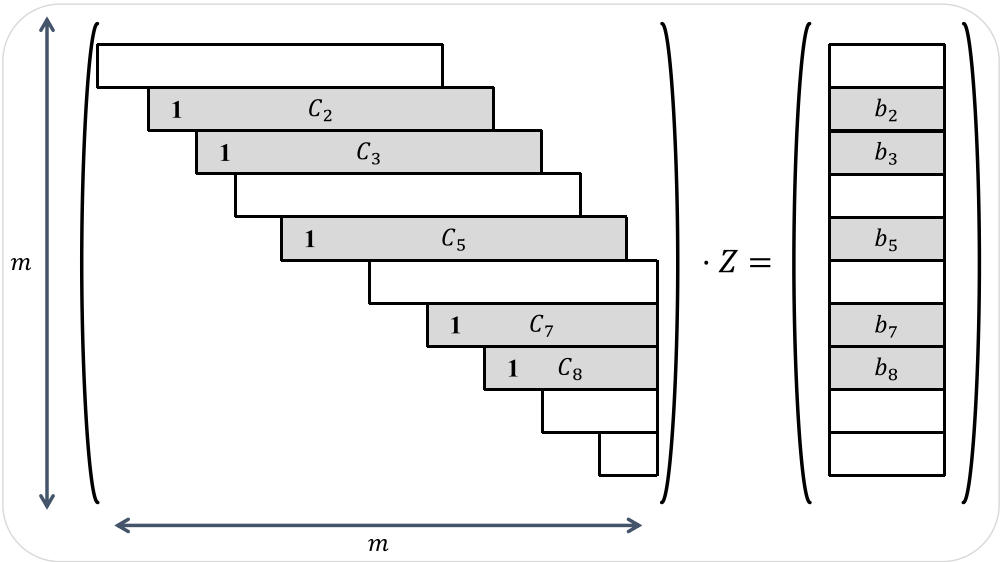


图 2.4　缎带过滤器构造过程中的矩阵示例

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，但实际中计算上的差距并不明显。

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

缎带过滤器的原文[7]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[25]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 D，当其存储的键值对 $\{k, v\}$ 中 v 为随机的数值，那么 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[9,26]）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[13]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[27]，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)[9]：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 $h_1, h_2, \cdots, h_k$ 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 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 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 D 的 k 个位置上。
- 如果 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 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，D 的长度 $m \approx 1.44kn$，对应的假正例率为 $1/2^k$。文献[25]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)[25-26]：混淆布谷鸟哈希表实际上是 Bloomier 过滤器[22] 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[10]。在 2008 年，Charles 等人[22]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，$D[h_1(x)]$ 和 $D[h_2(x)]$ 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 $f(x) = D[h_1(x)] \oplus D[h_2(x)]$。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[26] 和 3H-GCT[25]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[13]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 D 看作长度为 m 的向量时，那么构建 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 M：

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

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 $M \cdot D = v$ 使用高斯消元法求解，得到的 D 就是我们需要的编码结果。文献[25]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[28]通过使用矩阵变换的方式将 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[13]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[25]	1	$O(n^3)$	$O(n)$
混淆布隆过滤器[9]	$O(1/k)$	$O(n\lambda)$	$O(n)$
2H-GCT[26]	0.4	$O(n\lambda)$	$O(\lambda)$
3H-GCT[25]	$0.77-0.81$	$O(n\lambda)$	$O(\lambda)$
矩阵三角化算法[28]	$0.78-0.81$	$O(n\lambda)$	$O(\lambda)$
随机带状矩阵[13]	$0.91-0.97$	$O(n\lambda)$	$O(\lambda)$

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[13]，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[7]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。

如图 2.5 所示，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

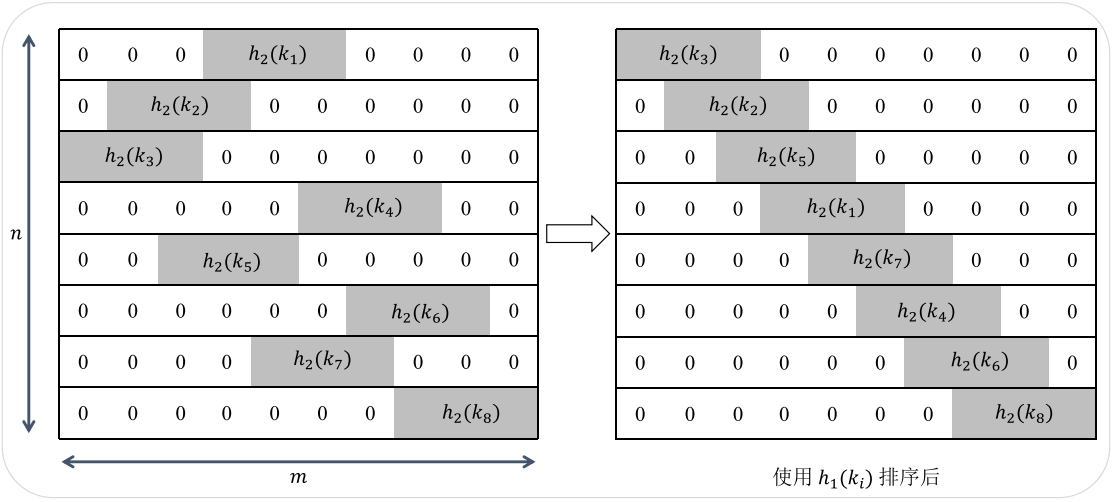


图 2.5　RB-OKVS 构造过程中的矩阵示例

运算，即：

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

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，所以 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[29]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[25]和基于矩阵三角化的方案[28]，RB-OKVS 具有更低的存储开销。

2.4　小结

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[20-21]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，但这种方案计算复杂度太高，而且需要两个哈希表。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，而是让每个元素在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[13,25,28]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[25]可以对应到异或过滤器[11]，RB-OKVS[13]可以对应到缎带过滤器[7]。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[11]和 3H-GCT[25]都受到 Botelho 等人[29-30]所提出的超图 (hypergraph) 构造的启发。缎带过滤器[7]和 RB-OKVS[13]中的高斯消元法实际上源于同一篇快速消元法的工作[24]。从中我们也可以发现，这些想法都不是凭空出现的。我们需要善于总结前人的经验，在已有工作的基础上做进一步的突破和创新。

第 3 章　在隐私保护协议中的应用

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。其中涉及的协议包括对称可搜索加密、隐私信息检索和隐私集合运算。

3.1　在对称可搜索加密协议中的应用

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)[31-32] 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 3.1 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

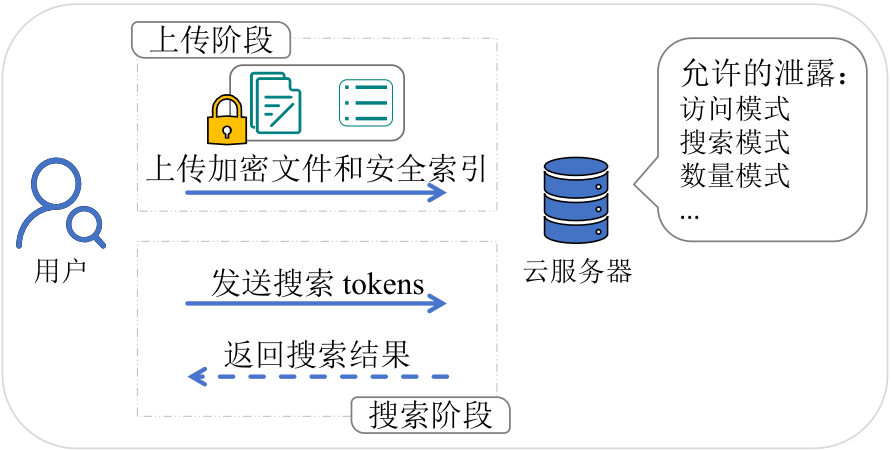


图 3.1　对称可搜索加密系统模型

相应的安全索引。在搜索阶段，用户根据需要搜索的关键词生成搜索 tokens，服务器使用搜索 tokens 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，但同时它会尝试通过分析输入输出信息来推断用户的隐私信息。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[33]虽然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[34]。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[35]。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。

目前安全索引大多以倒排索引形式构建，也就是使用关键词信息去匹配文件标识的形式。这样能够保证搜索效率只与关键词对应的文件数有关，而与整个数据库的大小无关。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，也就是说如果我们声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[36-41]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　HXT

这一节我们介绍的是 HXT (Hidden Cross-Tags) 协议[42]。HXT 是一个针对连接多关键词查询的可搜索加密协议。上一节介绍的背景都是默认只考虑单关键词查询，如果要扩展到多关键词并不是一件容易的事。最直接的方式是执行多次单关键词的查询，但这样做，一来服务器与用户之间交互次数太多，通信开销太大；二来服务器可以知道每次查询的各种模式信息。针对这些问题，Cash 等人[43]提出了次线性 (sublinear) 的连接多关键词协议 OXT (Oblivious Cross-Tags)，即对于任意形如 $w_1 \wedge w_2 \wedge \cdots \wedge w_n$ 的连接多关键词查询，OXT 的查询复杂度只与第一个关键词对应的文件数量有关（即次线性），与查询的关键词数量、数据库中文件数量无关。OXT 协议中包含两个索引，分别为 T-set 和 X-set，其中 T-set

与倒排索引类似，X-set 则是存储每个关键词 w 与文件标识 ind 配对的加密形式（称为 xtag）。搜索时，用户首先选取搜索关键词中对应预估文件数最低的关键词（称为 s-term）生成对应在 T-set 中的搜索 token（称为 stag），并生成其他搜索关键词（称为 x-term）对应的搜索 tokens（称为 xtokens）。以形如 $w_1 \wedge w_2 \wedge \cdots \wedge w_n$ 的连接多关键词为例，其中假设 s-term 为 w_1。服务器在收到这些搜索 tokens 之后，首先根据 stag 从 T-set 中检索出 w_1 对应的加密文件标识列表和盲化的关键词与文件标识配对列表。然后在上一步检索结果的基础上，使用 xtokens 还原出对应的 xtag，如果该 xtag 存在于 X-set 中，则返回对应的加密文件标识。图 3.2 对 OXT 协议的搜索过程进行了描述。从整体上来看，OXT 协议的核心思路是先通过 T-set 确定出一个较小的文件范围，再使用 X-set 从这些文件中筛选出符合条件的结果。这样就避免了前面提到的交互次数太多和泄露每个单独关键词模式信息的问题，而且能够实现次线性的搜索复杂度。但是在 OXT 协议中，因为需要计算出每个 x-term 与 s-term 对应的文件标识配对的 xtag，而服务器可以判断每一个 xtag 是否存在于 XSet 中。以图 3.2 中的查询为例，服务器可以判断出关键词 w_2 与 ind_1 配对的 xtag 存在于 X-set，而关键词 w_3 与 ind_1 配对的 xtag 不存在于 X-set。Lai 等人[42] 将这种信息定义为关键词对结果模式 (Keyword Pair Result Pattern, KPRP)。而 HXT 协议的提出就是为了解决 OXT 中 KPRP 信息泄露问题。

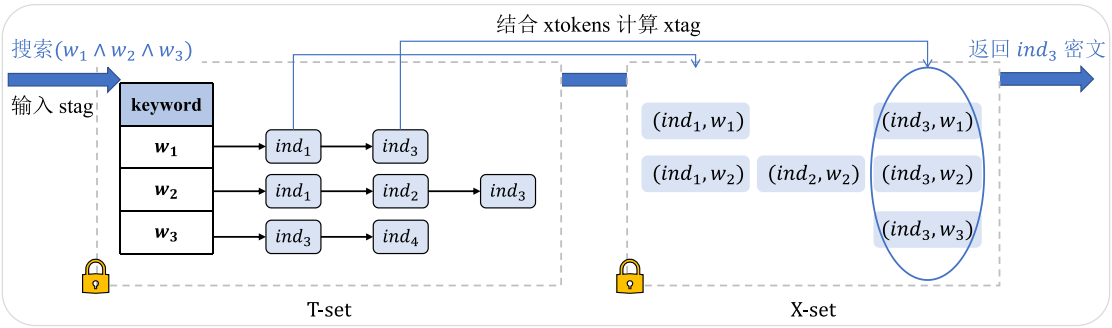


图 3.2　OXT 协议查询流程示例

HXT 协议相比 OXT 协议主要修改的是 X-set 构造。为了隐藏 KPRP，就需要保证服务器不能通过计算 xtag 来完成在 X-set 中的筛选，也就是说 X-set 中不能直接保存 xtag。HXT 协议采用的方法是将 X-set 使用布隆过滤器构造成长度为 m 的数组，再通过对称隐藏向量加密 (Symmetric Hidden Vector Encryption, SHVE) 进行加密，将加密结果作为 HXT 中的 X-set，转换过程如图 3.3 所示。

其中，SHVE 是一种针对 0/1 比特向量的加密算法，主要包含以下四个算法：

- SHVE.Setup(λ) $\rightarrow$ msk：通过输入安全参数 λ，输出主密钥 msk，并定义明文空间 $\mathcal{M}$。
- SHVE.KeyGen($msk, \mathbf{v} \in \{0, 1, *\}^m \rightarrow \mathbf{s}$：通过输入长度为 m 的预测向量 $\mathbf{v}$

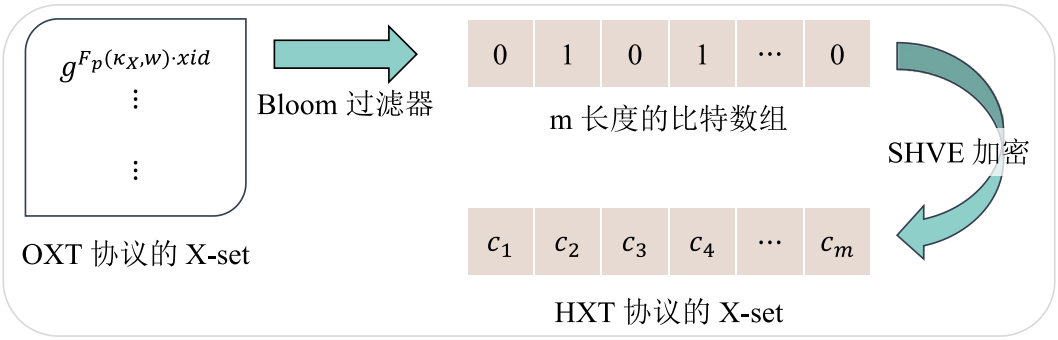


图 3.3　HXT 协议中 X-set 的构造

和主密钥，输出解密密钥 **s**。

- SHVE.Enc($msk, \mu \in \mathcal{M}, \mathbf{x} \in \{0,1\}^m$) $\rightarrow$ **c**：通过输入明文消息 μ 和长度为 m 的索引向量 **x**，输出密文 **c**。
- SHVE.Query(**s**, **c**) $\rightarrow \mu$ or $\perp$：通过输入预测向量 **v** 对应的解密密钥 **s** 和密文 **c**，如果 $P_{\mathbf{v}}^{\mathsf{SHVE}}(\mathbf{x}) = 1$，则返回明文消息 μ；否则返回 $\perp$。

其中 $P_{\mathbf{v}}^{\mathsf{SHVE}}(\mathbf{x}) = 1$ 的含义是向量 **v** 和 **x** 除了通配符 $*$ 所在位置上，其他所有位置上的值都相等，即

$$P_{\mathbf{v}}^{\mathsf{SHVE}}(\mathbf{x}) = \begin{cases} 1 & \forall 1 \leqslant i \leqslant m\ (v_i = x_i \text{ or } v_i = *), \\ 0 & \text{otherwise.} \end{cases} \tag{3.1}$$

关于方案的具体构造可以参考文献[42]，这里就不详细介绍。下面主要介绍 HXT 是如何结合 SHVE 来隐藏 KPRP 的。构造时，用户使用图 3.3 中的方式构造 X-set，在使用 SHVE 加密时，明文消息 μ 设置为 True。搜索时，用户还是首先生成 stag 和 xtoken，这部分 HXT 与 OXT 相同。服务器还是通过 stag 检索 T-set，再利用 xtoken 计算出一系列 xtag。由于此时 X-set 中并没有直接存储 xtag，所以服务器并不能直接通过 xtag 完成后续判断。对于服务器使用 stag 检索出的每个加密文件标识，后续的判断过程如下：

- 服务器使用布隆过滤器的哈希函数，计算 xtag 对应的位置，并返回给用户。
- 用户生成长度为 m 且所有位置为 $*$ 的向量 **v**，根据服务器返回的位置，将向量 **v** 上所有这些位置上的值设为 1。
- 用户调用 SHVE.KeyGen 将生成向量 **v** 对应的解密密钥 **s** 并发送给服务器。
- 服务器调用 SHVE.Query 来判断 **v** 是否能与 X-set 匹配，如果返回为 True，则将对应的加密文件标识返回给用户。

相比于 OXT 协议，HXT 在搜索时需要多一轮通信，目的是让用户生成判断向量 **v**。得益于布隆过滤器的特性，向量 **v** 记录了所有 x-term 的信息，服务器只能判断每个加密文件标识是否满足查询条件中的 x-term，并不能判断每个单独的 x-term 与加密文件标识的对应关系，从而避免 KPRP 信息的泄露。

3.1.3　XorMM

这一节介绍的 XorMM[44]是一个借助异或过滤器隐藏数量模式的对称可搜索加密协议。数量模式指的是在搜索过程中，服务器获取的查询结果数量信息。早期的可搜索加密方案中认为这类模式信息的泄露是可以接受的。但近年来的研究[37-39]指出，攻击者可以利用数量模式推测用户搜索的关键词甚至还原出用户存储的信息。因此隐藏数量模式就显得尤为重要。

要隐藏数量模式，最直接的思路是采用填充的方式，也就是将倒排索引中每个关键词对应的文件数量填充为相同的长度。这样无论是搜索哪个关键词，服务器查找得到的结果数量都相同，从而隐藏了每个关键词实际对应的数量模式信息。现有工作[45]认为，要实现隐藏数量模式信息的目的，填充后每个关键词对应的结果数量至少为填充前结果数量的最大值。这也就意味着填充后索引的存储开销会非常大，在实际中不可行。

为了解决这一问题，XorMM[44]提出将异或过滤器的结构作为可搜索加密中的索引。以关键词 w_1 对应文件标识 $\{ind_1, ind_2, ind_3\}$ 为例，XorMM 中将其看作 $\{(w_1||1, ind_1), (w_1||2, ind_2), (w_1||3, ind_3)\}$ 这样的键值型组合集合。这样一来便可以使用异或过滤器对其进行记录，比如将 ind_1 看作 $w_1||1$ 对应的指纹，查询时只需要输入 $w_1||1$，则返回对应的 ind_1。而对于不存在的键，比如 $w_1||4$，还是能够得到其对应的结果。以上描述的是明文索引的构造思路，对于安全索引，我们只需要将所有的关键词替换成对应的关键词 token，而文件标识也需要替换成对应的密文形式。这样构造的安全索引就能在不需要填充的情况下实现对任意查询都能返回一个确定的结果。

按照上述方式构造的安全索引能够实现较低的存储开销，但用户在查询时还存在通信开销上的问题。假设 ℓ 为数量模式的最大值，为了隐藏数量模式，就需要每次搜索时让服务器都返回 ℓ 个搜索结果。以查询关键词 w_1 为例，用户需要发送 $\{w_1||1, w_1||2, \cdots, w_1||\ell\}$ 对应的搜索 tokens，这会造成非常大的通信开销。为了解决这个问题，XorMM 采用受限伪随机函数 (Constrained Pseudorandom Function, CPRF) 让服务器可以根据用户给定的密钥独自生成 ℓ 个搜索 tokens，从而避免通信开销问题。图 3.4 给出了 XorMM 协议的查询流程，描述如下：

- 用户根据查询关键词生成对应的 CPRF 密钥 tk_{key}，以及设定的返回结果数量 ℓ，并发送给服务器。
- 服务器根据 tk_{key} 生成 ℓ 查询关键词对应的 ℓ 个搜索 tokens，并通过异或过滤器得到 ℓ 个对应的结果，最后将结果返回给用户。

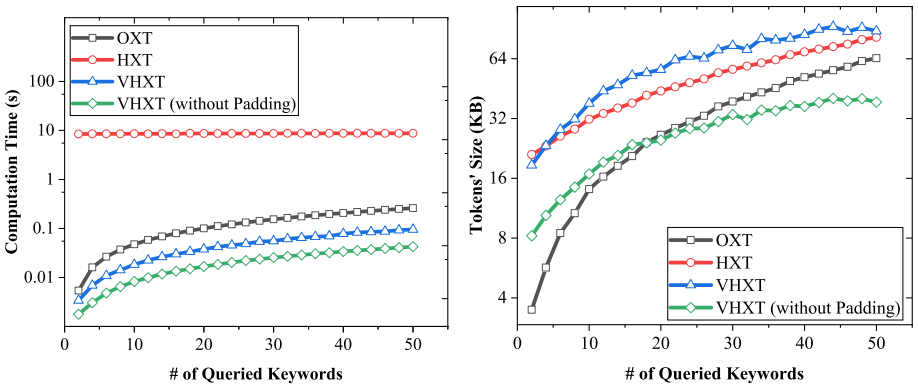
在 XorMM 的协议设计中，我们可以发现异或过滤器的作用更像是不经意键值存储，也就是说返回的结果是查询键对应的值，而不是简单的判断。因为对

图 3.4　XorMM 协议查询示例

于任意查询输入，不经意键值存储都会返回一个确定的值。所以在 XorMM 协议中，对于任意查询都返回 ℓ 个对应的结果，这就实现了在不需要填充的情况下，隐藏数量模式的信息。

既然 XorMM 协议中将异或过滤器看作不经意键值存储来使用，那么很自然可以想到使用其他的不经意键值存储来代替 XorMM 协议中的异或过滤器。Bienstock 等人[13]将 XorMM 中的异或过滤器替换为 RB-OKVS 设计了 RB-MM 协议，从而将存储开销从 1.23n 降低为 1.03n。

在了解了 HXT 和 XorMM 两个协议之后，会让我们思考这样一个问题：HXT 协议虽然能够隐藏 KPRP，但并不能隐藏数量模式；而 XorMM 虽然能够隐藏数量模式，但不支持连接多关键词查询。因此我在这些工作基础上，提出了一个新的协议 VHXT (Volume-Hiding Cross-Tags)，即在 HXT 的基础上，进一步隐藏数量模式。其中主要做的改进还是在 X-set 上。受到 XorMM 和 RB-MM 的启发，VHXT 协议中将 X-set 改造成由 xtag 到加密文件标识的映射形式（即键值型数据），这样一来就可以使用 RB-OKVS 存储 X-set 的信息。为了隐藏 T-set 的数量模式，VHXT 进一步采用了非负差分隐私机制对 T-set 进行填充。图 3.5 给出了



(a) 用户端搜索时间对比　　(b) 用户端通信开销对比

图 3.5　针对不同数量的搜索关键词，不同方案开销对比

VHXT 与 OXT 以及 HXT 在不同数量的搜索关键词情况下，用户端搜索时间与通信开销对比。得益于 RB-OKVS 的高效压缩性能，在不填充额外信息的情况下，我们提出的 VHXT 相比其他相关方案能取得更优的效率。

3.2　在隐私信息检索协议中的应用

3.2.1　背景介绍

隐私信息检索 (Private Information Retrieval, PIR) 是一种用于隐藏用户查询内容的隐私保护密码协议。如图 3.6 所示，隐私信息检索协议中存在用户和服务器两个实体。对于用户只需要输入检索条件，并能得到相应的检索结果。而检索

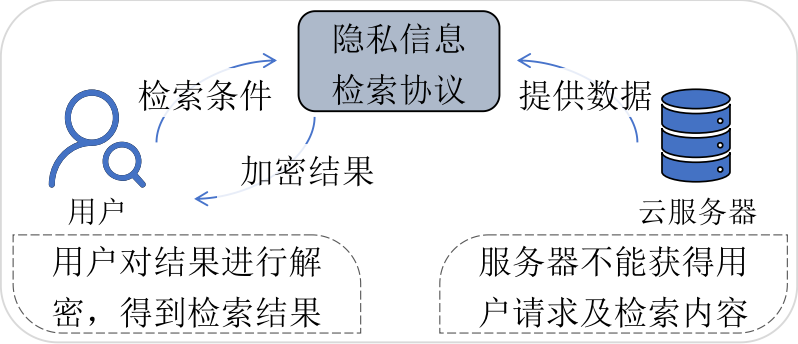


图 3.6　隐私信息检索系统模型

的数据则由服务器提供。与可搜索加密相比，隐私信息检索针对的场景和考虑的问题都有所区别。在场景上，可搜索加密针对的是外包数据存储场景，也就是云服务器中存储的是用户的数据，典型的是各种云盘服务；而隐私信息检索中所有的数据是由服务器自身掌握的，也就是数据本身并不是属于查询用户的，比如像查询火车票服务。由于数据的持有方不同，二者考虑的问题也就相差甚远。在可搜索加密中，因为数据都是以加密形式存储在服务器，所以服务器能够获取到的信息是非常有限，此时只需要考虑服务器不能根据查询时泄露的各种模式信息推断出用户的查询内容或存储内容；在隐私信息检索中，因为服务器是直接掌握所存储的数据，这就要求服务器不能获取访问模式和搜索模式相关的信息。如何保护用户的查询隐私成为隐私信息检索的主要目标。

根据实现方式，可以将隐私信息检索分为基于单服务器和基于双服务器两种方式。在基于单服务器的方案中，用户与服务器的通信开销与存储条目数量成正比。而服务器的计算开销也非常大，因为服务器必须要访问每一个存储条目，否则就会泄露当前用户没有查询到哪些内容的信息[46]。而基于双服务器的方案虽然相比单服务器的方案在计算开销上要更低，但是这类方案需要假设两个服务器是不共谋的，这种安全假设太强，难以在现实中部署。

根据查询方式，隐私信息检索又分为基于索引 (index-based PIR) 和基于关键词 (keyword-based PIR) 两种类型。顾名思义，在基于索引的隐私信息检索协议中，每条数据都对应一个索引 $i \in \{1, 2, \cdots, n\}$，查询时用户通过索引得到需要查询的条目，而服务器并不能推断用户发送的索引信息；而在基于关键词的隐私信息检索协议中，每条数据对应一个关键词，查询时用户通过关键词得到需要查询

的条目，服务器不能推断用户发送的关键词信息。

本文介绍的是基于关键词的单服务器隐私信息检索方案。我们假设服务器是半诚实的，即服务器会诚实地执行协议，但同时也会通过协议执行时获取的信息来推测用户的查询信息。早期的基于关键词的隐私信息检索协议是通过执行对数轮基于索引的隐私信息检索协议来实现的。近年来有一部分工作是借助全同态加密实现基于关键词的检索。但相比于基于索引的协议，目前大部分基于关键词的协议无论在通信上还是在计算上依然存在开销较高的问题。

3.2.2　ChalametPIR

ChalametPIR[47]是由 Celi 和 Davidson 结合同态加密和二进制引信过滤器提出的一种计算高效且通信开销低的隐私信息检索方案。从计算效率来看，ChalametPIR 相比现有基于关键词的隐私信息检索方案要快 6 到 11 倍。在介绍 ChalametPIR 之前，我们首先介绍构造该协议需要用到的 Regev 同态加密方案[48]。

Regev 加密[48]是一种基于 LWE (Learning with Errors) 困难问题的加密方案，它可以支持同态加法操作。简单来说，令 p, q, n, σ 为 Regev 加密机制 Σ_{lwe} 的参数，λ 为安全参数，它主要包含以下四个算法：

- $\Sigma_{\mathsf{lwe}}.\mathsf{KeyGen}(1^{\lambda}, q, p, n, \sigma) \rightarrow (pp, sk)$：通过输入预先设定的参数，生成公共参数 pp 和私钥 sk。
- $\Sigma_{\mathsf{lwe}}.\mathsf{Enc}(pp, sk, \mathbf{v} \in \mathbb{Z}_p) \rightarrow c$：根据公共参数和私钥，对输入明文 v 进行加密，输出密文结果 c。
- $\Sigma_{\mathsf{lwe}}.\mathsf{Eval}(pp, \mathbf{c} \in \mathbb{Z}_b^m, \mathbf{w} \in \mathbb{Z}_p^m) \rightarrow c'$：给定一组由相同密钥生成的密文 $\mathbf{c} = (c_1, c_2, \cdots, c_m)$，以及明文向量 $\mathbf{w}$，输出密文 c'。
- $\Sigma_{\mathsf{lwe}}.\mathsf{Dec}(pp, sk, c) \rightarrow v$：给定公共参数 pp，私钥 sk，以及密文 c，输出对应的明文 v。

相比一般形式的加密，Regev 加密机制中多了一个 $\Sigma_{\mathsf{lwe}}.\mathsf{Eval}$ 算法。其返回得到的密文 c' 在解密后得到的是向量 $\mathbf{c}$ 对应的明文 $\mathbf{v}$ 与向量 $\mathbf{w}$ 的内积。这是由于 Regev 加密机制满足加法同态的性质而得到的。关于 Regev 加密机制的具体构造这里就不展开介绍，感兴趣的可以参看文献[48]。

我们重点来说明 Regev 加密机制是如何用来构造隐私信息检索协议的。近年来有一系列工作[49-51]都是采用这种技术路线，这类协议也被称作 LWE-based PIR，下面就简记为 LWEPIR。LWEPIR 分为离线和在线两个阶段。在离线阶段，服务器需要执行生成全局公共状态信息 (global public state)，并将该信息共享给用户。注意这里的离线阶段只需要执行一次。在线阶段可以分为三个部分，首先由用户发起请求，然后服务器回应请求，最后用户将服务器返回的结果进行解密。LWEPIR 的构造思路是将服务器上的数据看作 $m_1 \times m_2$ 的矩阵，即一共有 m_1

行数据，每条数据长度为 m_2。在离线阶段，服务器预先生成 Σ_{lwe}.Enc 过程中用到的参数 A。在查询过程中，以请求第 i 行数据为例（假设我们这里考虑的是基于索引的协议），用户只需要生成长度为 m_1 的全零查询向量，其中第 i 个位置设为 1。用户再随机生成私钥，调用 Σ_{lwe}.Enc 并结合参数 A 对查询向量进行加密，将加密结果发送给服务器。服务器将本地 $m_1 \times m_2$ 的明文矩阵看作 m_2 个长度为 m_1 的列向量组成，对于每个列向量，将其与用户发送的加密向量作为输入，调用 Σ_{lwe}.Eval 计算得到加密结果。最后将所有的加密结果返回给用户。用户将结果解密便可以得到第 i 行数据的信息。

LWEPIR 的正确性很容易进行验证。因为 Σ_{lwe}.Eval 计算的结果是输入两个向量的内积。在该协议中，输入的两个向量分别是用户发送的加密查询向量和服务器上存储数据的列向量。由此可以推出，假设用户查询的第 i 行，服务器会依次得到第 i 行上所有列的加密数值（因为是内积运算，只有第 i 行为 1）。在这一过程中，只要用户每次查询都是随机生成加密密钥，那么用户发送的加密向量对于服务器来说就是随机的，服务器无法通过用户的查询获取查询模式相关的信息。而在查询过程中，服务器存储的数据上每一比特都参与了运算，因此服务器也无法获取访问模式相关的信息。也就是说这样的设计能够满足隐私信息检索协议的隐私保护需求。

但是以上的构造只是针对基于索引的协议来进行设计的。ChalametPIR 通过引入二进制引信过滤器，将上述构造改造成了基于关键词的协议。在使用关键词作为查询条件的情况下，服务器上存储的数据就可以看作键值型数据形式，即关键词作为键，对应的数据条目作为值。那么直接的思路就是可以将这种数据采用异或型过滤器来构造。ChalametPIR 协议的构造思路如图 3.7 所示，其中指纹函

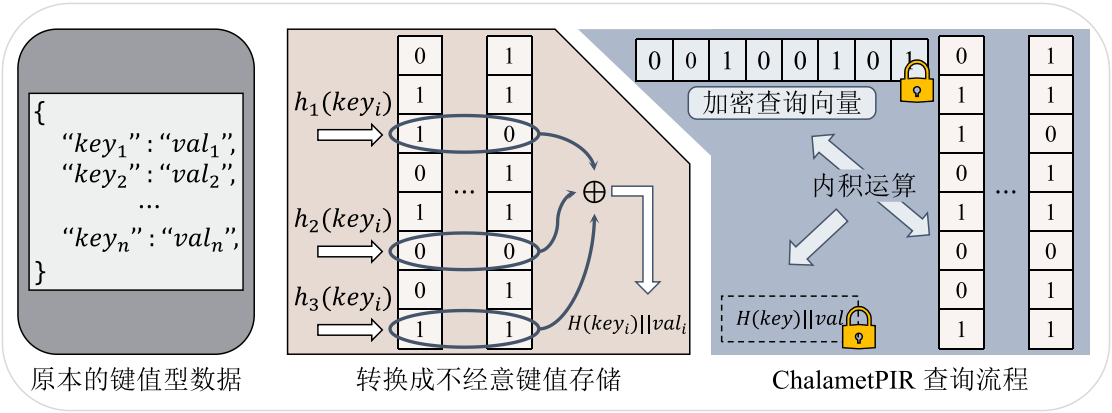


图 3.7　ChalametPIR 协议构造思路

数被定义为 $f(key) = H(key)||val$，H 为哈希函数。这样在离线阶段，服务器需要额外生成过滤器的相关参数信息。用户在查询时，与 LWEPIR 类似，也是先生成全零向量，再使用过滤器的哈希函数计算出所需要查询的关键词对应的位

置，并在查询向量中将这些位置上的比特置为 1。之后用户便采用相同的方法对向量进行加密，并把加密后的查询向量发送给服务器。服务器调用 $\Sigma_{\mathsf{lwe}}.\mathsf{Eval}$ 以相同的方式对加密查询向量与本地构造的过滤器进行计算，将结果返回给用户。用户将结果进行解密后便可以得到查询关键词对应的数据。由于异或型过滤器本身可以记录键值型数据，而且 Evaluate 的执行过程也可以看作内积操作，再得益于 Regev 加密算法对密文上求内积操作的支持，这些因素使得二者之间的组合可以实现基于关键词的隐私信息检索协议。

3.3　在隐私集合运算协议中的应用

3.3.1　背景介绍

隐私集合运算 (Private Set Operation, PSO) 是隐私保护计算领域中的一个重要分支，主要用于在不泄露集合数据的情况下，对两方或多个参与方各自私有集合执行安全运算。常见的隐私集合运算包括隐私集合求交 (Private Set Intersection, PSI) 和隐私集合求并 (Private Set Union, PSU) 两种。本文中我们主要介绍两方参

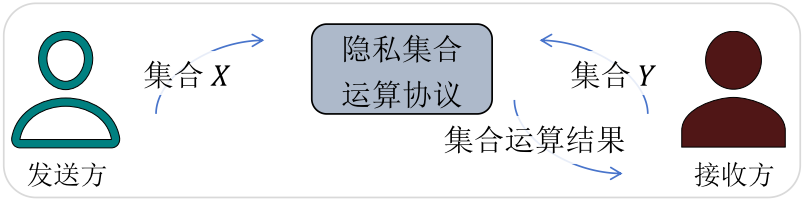


图 3.8　隐私集合运算系统模型

与的隐私集合运算协议，如图 3.8 所示，这两个参与方分别是发送方和接收方。在隐私集合求交协议中，发送方持有集合 X，接收方持有集合 Y。协议执行完成后，发送方不会获得任何信息，接收方仅能获得 $X \cap Y$ 的信息。同理，隐私集合求并协议要求接收方只能获得 $X \cup Y$ 的信息，而无法获得 $X \cap Y$ 的信息。隐私集合运算在现实中有着广泛的应用前景[14]，比如隐私集合求交可以用于联系人匹配、基因检测和模式匹配、传染病患者追踪等；隐私集合求并可以用于隐私保护数据聚合、合并 IP 黑名单进行网络风险评估等。

隐私集合运算协议常用的安全模型有半诚实模型和恶意模型。在半诚实模型中，所有参与方都诚实地执行协议，但是也会尝试从其他参与方的输入或协议的中间计算结果中推断隐私信息。在恶意模型中，参与方会主动去破坏协议的安全性，包括恶意篡改输入输出信息、拒绝参与协议、提前终止协议等。在本文中，如无特殊说明，默认各参与方是半诚实的。假设发送方和接收方分别持有集合 $X = \{x_1, x_2, \cdots, x_{n_X}\}$ 和 $Y = \{y_1, y_2, \cdots, y_{n_Y}\}$。实现隐私集合运算协议的直接方式是让发送方和接收方对各自集合中的所有元素进行隐私保护的两两比较。

但当双方集合规模较大时，这样做会带来非常严重的通信和计算开销。为了降低开销，隐私集合计算协议中通常需要引入各种特定的数据结构，如哈希表、过滤器和不经意键值存储[14]。

借助哈希表构造的协议流程如图所示。在该过程中，发送方和接收方利

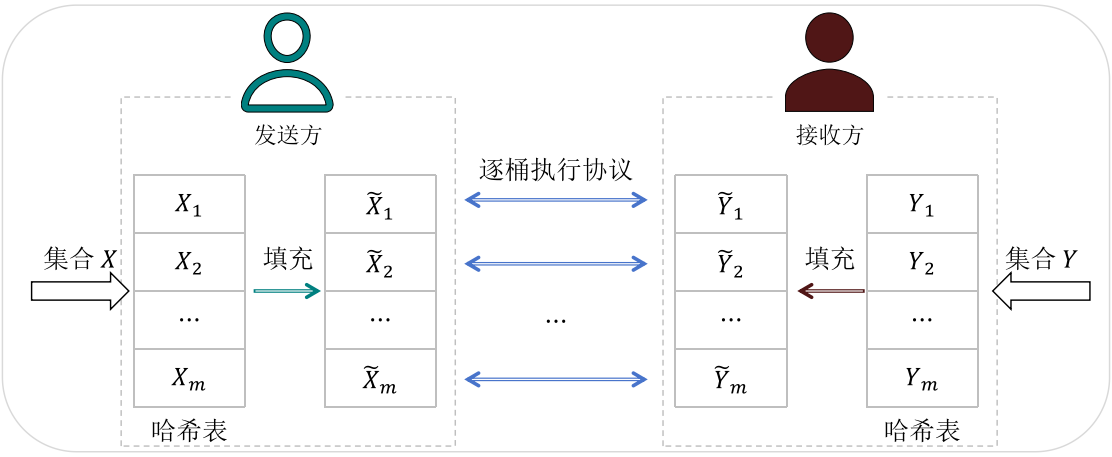


图 3.9　基于哈希表构造的隐私集合运算协议流程

用哈希表将集合 X 和 Y 分别划分成 m 个不相交的子集 $\{X_1, X_2, \cdots, X_m\}$ 和 $\{Y_1, Y_2, \cdots, Y_m\}$。为了确保相同的元素能够映射到相同序号的子集中，双方需要使用相同的哈希函数进行构造。这样双方就能通过哈希表完成集合元素的对齐。之后双方在每个子集 X_i 和 Y_i 中填充一定数量的哑元得到 $\tilde{X}_i$ 和 $\tilde{Y}_i$。最后双方在填充哑元后的 $\tilde{X}_i$ 和 $\tilde{Y}_i$ 上执行隐私集合计算协议。这样做的好处是双方不再需要对所有元素进行两两比对，从而降低计算复杂度。使用填充的目的是防止将桶中元素的数量暴露出来。此类协议根据参与双方使用的哈希表结构不同主要可以分为两类：一类是双方都使用哈希表，再根据划分的每个子集中的元素构建多项式。假设双方集合中的元素数量都为 n，构建的哈希表长度为 m，哈希表中每个桶的大小为 b，那么这种方法就把原本需要构造 n 次多项式转换成构造 m 个 b 次多项式。由于桶中容量最多为 $b = O(\log n)$，因此元素的比较次数从 n^2 降低为 $O(n \log^2 n)$。另一类是一方使用布谷鸟哈希表，另一方使用一般形式的哈希表。在前面我们也介绍过，布谷鸟哈希表中桶的大小为 1，而对于每个元素，它存放的位置可能是使用哈希函数映射后得到的位置中的一个。在比较时，一方使用布谷鸟哈希表将元素 x 存储在 k 个哈希函数对应位置中的一个，另一方使用一般形式的哈希表将元素 y 存储在 k 个哈希位置上。在逐个位置进行对比时，由于布谷鸟哈希中 $b = 1$，此时比较次数进一步降低为 $O(n \log n)$。

基于过滤器构造的协议主要是利用过滤器的成员测试功能避免双方交互过程中直接发送原始数据集，从而防止出现通信开销过大的问题。将过滤器应用到隐私集合运算主要有两种方式：一种是将数据集盲化后插入过滤器，其协议流程如图 3.10 所示。比如发送方将数据集 $\{x_i\}_{i\in[1,n]}$ 盲化后得到 $\{x_i^*\}_{i\in[1,n]}$ 并将盲化

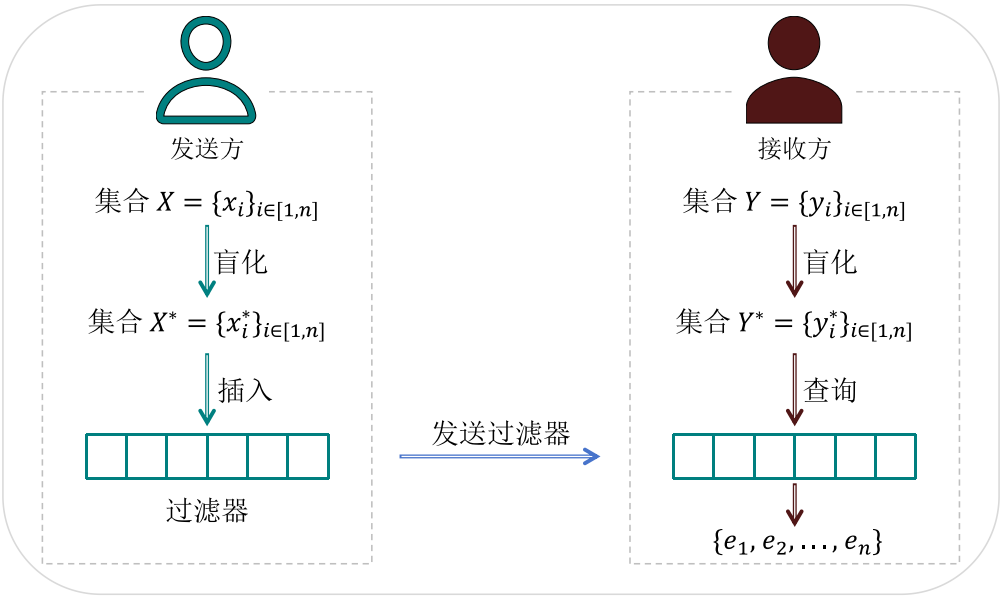


图 3.10　基于过滤器和盲化技巧构造的隐私集合运算协议流程

后的数据集插入过滤器。由于盲化后的数据集并不会暴露原始集合，因此发送方直接将得到的过滤器发送给接收方。接收方使用同样的方式对自己的集合进行盲化，并通过过滤器来判断哪些元素是与接收方相同的。另一种是将数据集以明文的形式插入过滤器中并将过滤器加密，其协议流程如图 3.11 所示。比如接收

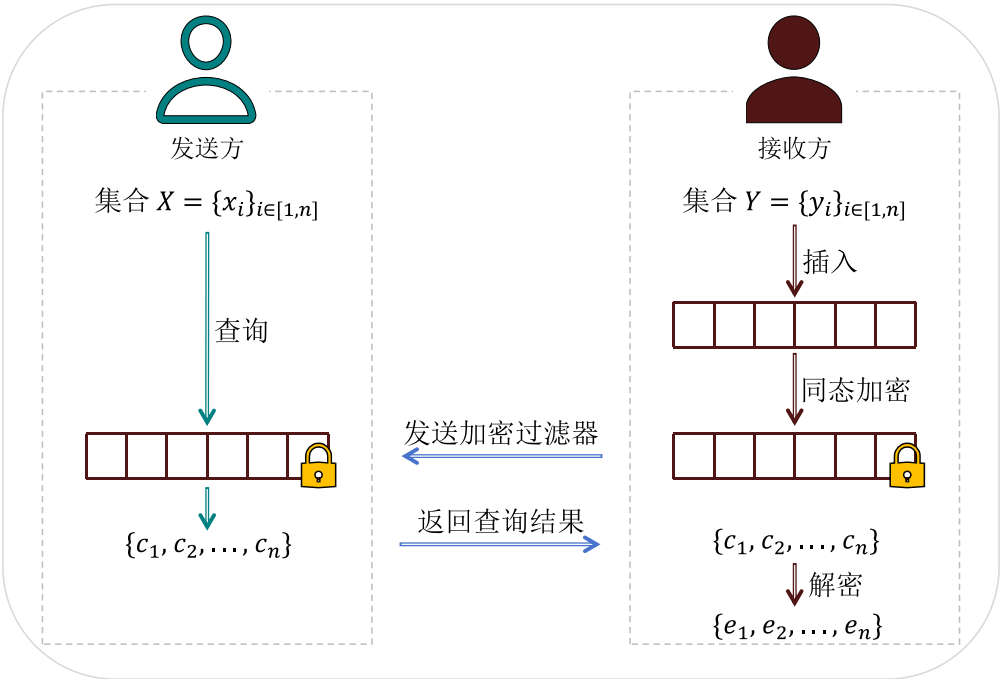


图 3.11　基于过滤器和同态加密构造的隐私集合运算协议流程

方将数据集 $\{y_i\}_{i\in[1,n]}$ 直接插入布隆过滤器中，再使用同态加密对布隆过滤器的每一个比特进行加密，并将加密结果发送给发送方。发送方对自己集合中每个元素 x_i 找到布隆过滤器上对应的位置，将对应位置上的密文进行相加，得到 c_i。如果 $x_i \in Y$，那么 c_i 应该是 k 的密文（或者为 0 的密文，取决于布隆过滤器是否逐

比特反转），其中 k 为布隆过滤器中哈希函数的数量。发送方再将得到的密文发送给接收方，接收方解密后便可以判断出交集信息。上述举得两个例子虽然都是计算交集，但也可以通过类似的方式改造成计算并集，这里就不详细介绍。感兴趣的可以阅读文献[52-53]。除了使用布隆过滤器之外，Resende 和 Aranha[54] 提出了基于布谷鸟过滤器的隐私信息检索方案，与基于布隆过滤器构造的方案相比，该方案的通信开销更低。

近些年来有越来越多的工作是在不经意随机存储结构上构造隐私集合运算协议。更准确地说，不经意键值存储结构本身就是从隐私计算求交协议[25]中总结出来的一种数据结构。有许多隐私计算操作协议[55-58]均显式或隐式地利用了不经意随机存储结构对集合数据进行编码。此类协议主要利用了不经键值存储的不经意性，也就是对于映射关系 $x_i \mapsto v_i$，当 v_i 为随机值时，不经意键值存储结构能够隐藏其对应的键 x_i。对应到隐私集合运算协议中，x_i 一般对应参与者集合中的元素，而 v_i 一般对应协议中的关键辅助信息。协议的执行流程如图 3.12 所示。接收方随机生成与元素对应的关键辅助信息 $\{v_i\}_{i\in[1,n]}$，以此构建键值对数

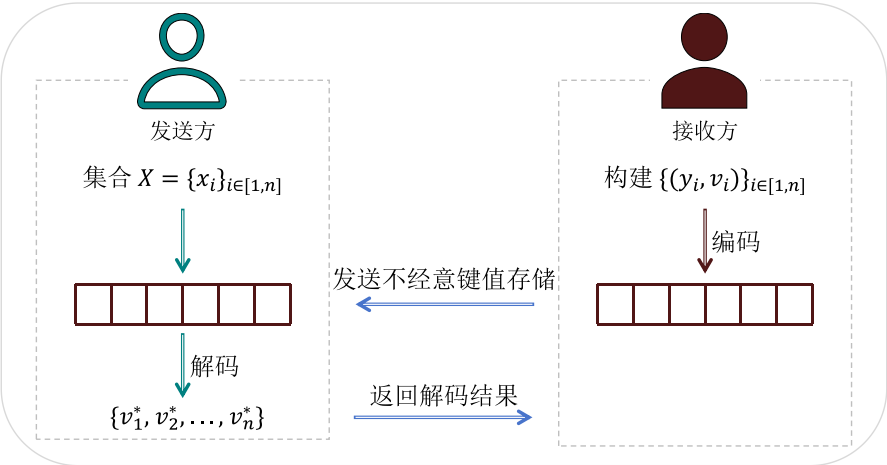


图 3.12　基于不经意键值存储构造的隐私集合运算协议流程

据 $\{(y_i, v_i)\}_{i\in[1,n]}$，并使用不经意随机存储编码，将编码后的结果发送给发送方。发送方将自己的集合元素作为输入，在收到的编码结果进行解码，得到一系列的 $\{v^*\}_{i\in[1,n]}$。最后双方可以根据集合 $\{v_i\}_{i\in[1,n]}$ 和 $\{v_i^*\}_{i\in[1,n]}$ 的信息进一步来完成求交集或并集的操作。另外，不经意键值存储也可以用于前面基于哈希表的构造中来提高安全性。在基于哈希表构建的隐私集合求交协议中，如果是在恶意安全模型假设下，恶意参与者可以选择将交集元素存放在其映射位置上的其中一个而非所有位置，这样就会对协议造成破坏。而如果使用不经意键值存储，因为所有的存储对象是以异或拆成多份存储的，可以避免上述问题的出现。接下来我们着重介绍基于不经意键值存储构造的隐私集合求交协议 VOLE-PSI[57]和隐私集合求并协议 SKE-PSU[58]。

3.3.2　VOLE-PSI

VOLE-PSI[57]是结合向量不经意线性评估 (Vector Oblivious Linear Evaluation, VOLE) 和不经意键值存储设计的隐私集合求交方案。其中的一个核心模块是向量不经意线性评估，它通常被用于构造各种高效的安全多方计算协议，特点是具有较低的通信复杂度。简单来说，它的作用是随机生成长度为 m 的向量 $\mathbf{A}, \mathbf{B}, \mathbf{C}$，以及元素 Δ，并且让 $\mathbf{A}, \mathbf{B}, \mathbf{C}$ 满足线性关系：$\mathbf{C} = \Delta \cdot \mathbf{A} + \mathbf{B}$。协议参与的接收方得到向量 $\mathbf{A}$ 和向量 $\mathbf{C}$，而发送方得到向量 $\mathbf{B}$ 和 Δ。

VOLE-PSI 协议是建立在 VOLE 协议之上的。也就是说在计算交集之前，参与计算的双方需要执行 VOLE 协议：接收方得到向量 $\mathbf{A}$ 和向量 $\mathbf{C}$，发送方得到向量 $\mathbf{B}$ 和 Δ，且满足 $\mathbf{C} = \Delta \cdot \mathbf{A} + \mathbf{B}$。整体协议流程如图 3.13 所示。对于持有集

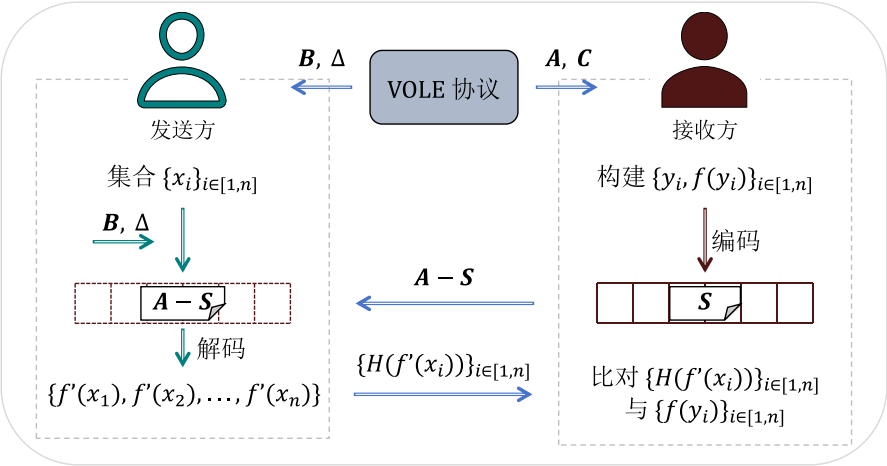


图 3.13　VOLE-PSI 协议流程

合 $Y = \{y_i\}_{i\in[1,n]}$ 的接收方，首先将集合中的每个元素看作键，并生成对应的值 $\{f(y_i)\}_{i\in[1,n]}$，并且满足：

$$f(y_i) = \mathsf{H}(\mathsf{Decode}(\mathbf{C}, y_i)), \tag{3.2}$$

其中 H 为随机谕言机 (random oracle)，Decode 为不经意键值存储中的解码算法。然后接收方可以将键值型数据 $\{(y_i, f(y_i))\}_{i\in[1,n]}$ 编码成不经意键值存储结构 $\mathbf{S}$。接收方将 $\mathbf{A} - \mathbf{S}$ 发送给发送方。发送方使用 $\mathbf{B}$ 和 Δ 计算 $\mathbf{B}'$，如下所示：

$$\mathbf{B}' = \mathbf{B} + \Delta \cdot (\mathbf{A} - \mathbf{S}). \tag{3.3}$$

对于集合 $X = \{x_i\}_{i\in[1,n]}$ 中的每一个元素 x_i，发送方使用不经意键值存储的 Decode 算法进行解码，计算 $f'(x_i)$，如下所示：

$$f'(x_i) = \Delta \cdot \mathsf{Decode}(\mathbf{S}, x_i) + \mathsf{Decode}(\mathbf{B}', x_i). \tag{3.4}$$

发送方使用与接收方相同的 H 得到新的集合 $\{\mathsf{H}(f'(x_i))\}_{i\in[1,n]}$，并将该集合返回给接收方。根据 VOLE 中的线性关系以及**定义 1.2** 中 Decode 算法的性质，当集

合 X 中存在与 Y 有交集的元素时，即 $x_i \in Y$ 时，有：

$$f'(x_i) = \Delta \cdot \mathrm{Decode}(\mathbf{S}, x_i) + \mathrm{Decode}(\mathbf{B}', x_i) \tag{3.5}$$

$$= \Delta \cdot f(x_i) + \mathrm{Decode}(\mathbf{B}, x_i) + \mathrm{Decode}(\Delta \cdot (\mathbf{A} - \mathbf{S}), x_i) \tag{3.6}$$

$$= \Delta \cdot f(x_i) + \mathrm{Decode}(\mathbf{B}, x_i) + \Delta \cdot \mathrm{Decode}(\mathbf{A}, x_i) - \Delta \cdot f(x_i) \tag{3.7}$$

$$= \mathrm{Decode}(\mathbf{B}, x_i) + \Delta \cdot \mathrm{Decode}(\mathbf{A}, x_i) \tag{3.8}$$

$$= \mathrm{Decode}(\mathbf{C}, x_i). \tag{3.9}$$

接收方只需要根据 $\mathbf{C}$ 可以直接计算 $\{f(y_i)\}_{i\in[1,n]}$，再与发送方返回的结果进行比对便能得到 $X \cap Y$ 的交集结果。

3.3.3　SKE-PSU

SKE-PSU[58] 是结合不经意传输 (Oblivious Transfer，OT) 和多点反向隐私成员测试 (Multi-Query Reverse Private Membership Test, mq-RPMT) 设计的隐私集合求并方案。其中的 mq-RPMT 构造使用到了不经意键值存储结构。在介绍具体方案之前，我们首先简单介绍不经意传输和多点反向隐私成员测试的含义。

不经意传输是一种常见的安全多方计算协议。最基础的不经意传输协议是 1-out-of-2 OT，其系统模型如图 3.14 所示。其中发送方持有两个消息，分别为 m_1

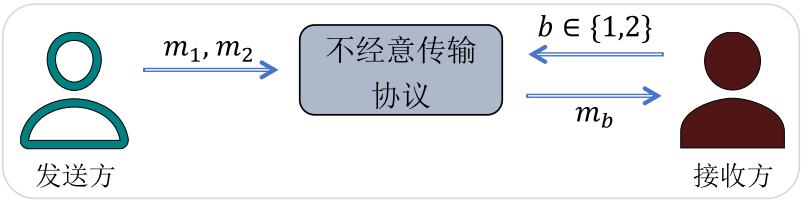


图 3.14　1-out-of-2 不经意传输系统模型

和 m_2，而接收方想获取其中的一个消息 $m_b, b \in \{1,2\}$。通过执行 1-out-of-2 OT 协议之后，接收方获得了消息 m_b 但并不知道另一个消息的信息，而发送方不知道接收方想要获取的是哪个消息。由 1-out-of-2 OT 可以扩展到 1-out-of-n OT，即接收方可以获取到发送方 n 个消息中的一个，但发送方并不能判断接收方获取的具体是哪个消息。

多点反向隐私成员测试是一个两方协议，它的作用是判断发送方提供的元素 x_i 是否属于接收方的集合 Y，并让接收方获得判断结果，而发送方不获得任何信息。在执行该协议之后，接收方便能知道哪些序号的元素是不在集合 Y 中的，之后就可以通过不经意传输协议获取这些元素，得到并集结果。早期的反向隐私成员测试协议[55] 一次只能判断一个元素，也被称为单点反向隐私成员测试。SKE-PSU[58] 中借助不经意键值存储结构和 VODE (Vector Oblivious Decryption-then-Matching) 协议实现了多点反向隐私成员测试协议，也就

是可以一次判断多个元素。这里的 VODE 也是一种作用于两方的协议，假设 $\mathcal{E} = (\mathsf{Setup}, \mathsf{KeyGen}, \mathsf{Enc}, \mathsf{Dec})$ 为一个抗选择明文攻击安全的对称加密机制，VODE 的功能描述如下：

- 接收方持有明文 s 以及由 KeyGen 生成的密钥 k。
- 发送方持有集合 $\{s_1^*, s_2^*, \cdots, s_n^*\}$，其中每个元素都来自 $\mathcal{E}$ 的密文空间。
- 对于 $i \in [1, n]$，计算 $s_i' = \mathsf{Dec}(k, s_i^*)$。如果 $s_i' = s$，令 $b_i = 1$，否则 $b_i = 0$。
- 接收方得到判断结果 $\{b_i\}_{i\in[1,n]}$。

在有了 VODE 协议的基础上，我们可以构建多点反向隐私成员测试，其流程如图 3.15 所示。我们依然假设发送方持有集合 $X = \{x_i\}_{i\in[1,n]}$，接收方持有集合

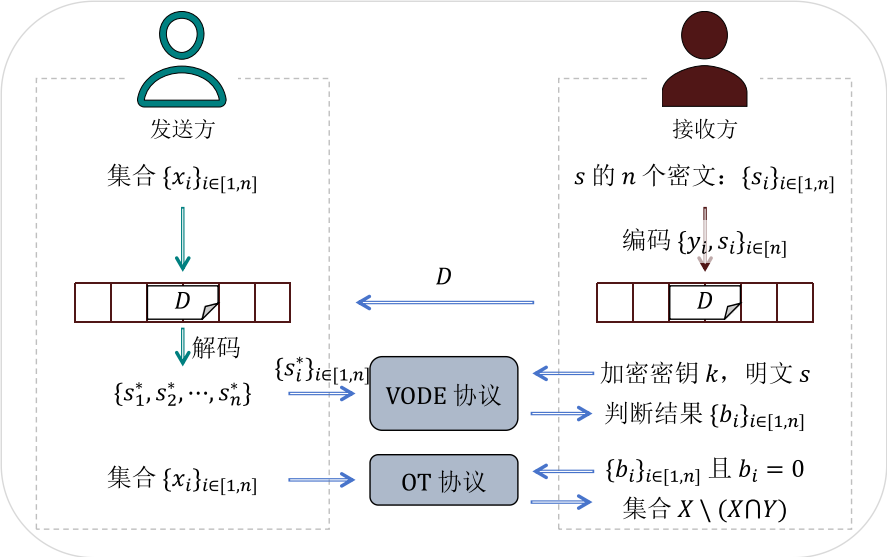


图 3.15　SKE-PSU 协议流程

$Y = \{y_i\}_{i\in[1,n]}$，协议流程描述如下：

- 接收方随机生成标识字符串 s，使用 KeyGen 生成密钥 k，并使用加密机制 $\mathcal{E}$ 将 s 加密 n 次，得到 $\{s_1, s_2, \cdots, s_n\}$。
- 接收方构造键值型数据 $\{(y_1, s_1), (y_2, s_2), \cdots, (y_n, s_n)\}$，并使用不经意键值存储的 Encode 编码成 D，将 D 发送给发送方。
- 发送方收到 D 之后，便可以用自己的集合 X 作为输入，使用不经意键值存储的 Dncode 算法解码成 $\{s_1^*, s_2^*, \cdots, s_n^*\}$。
- 接收方和发送方执行 VODE 协议，发送方输入 $\{s_1^*, s_2^*, \cdots, s_n^*\}$，接收方输入 (k, s)。因为 VODE 的作用是判断密文 s_i^* 与明文 s 能否匹配，所以最终接收方获取到一系列判断结果 $\{b_1, b_2, \cdots, b_n\}$。
- 在得到判断结果的输出后，接收方便能通过不经意传输协议，从发送方获取到不属于集合 Y 中的元素，最终得到 $X \cup Y$ 的结果。

得益于不经意键值存储的性质，当 $x_i \in Y$ 时，发送方计算出的 s_i^* 就是 s 的密文。

而不经意的性质确保发送方无法知道 x_i 是否属于 Y，避免发送方获取到额外的信息。而当计算出判断结果 $\{b_i\}_{i\in[1,n]}$ 之后，接收方也只知道当 $b_i = 1$ 时，对应集合 X 中序号为 i 的元素存在于集合 Y 中，但并不能知道 x_i 的具体信息，从而防止接收方获得交集信息。

3.4　小结

这一节我们主要介绍了布隆过滤器及其衍生的数据结构在对称可搜索加密、隐私信息检索和隐私集合运算上的应用。这些数据结构之所以能够广泛应用在各种不同的隐私保护协议之中，主要有以下几个方面的原因：

- 较低的空间开销：在设计隐私保护计算协议尤其涉及到双方交互的协议时，影响性能的因素之一便是通信开销。而过滤器或者不经意的键值存储本身就是一种压缩率非常高的结构，引入这些结构会大大降低协议的通信开销。比如在隐私集合运算过程中，交互双方并不会直接发送（加密后的）集合信息，而是采取将集合表示为过滤器或不经意键值存储结构后再发送。
- 灵活的编码形式：布隆过滤器能够将集合中的元素编码成 0/1 比特串的形式，从而可以直接应用到输入为比特向量的加密机制中（比如 HXT 中的 SHVE）；异或型过滤器和不经意键值存储能够将键值型数据重新编码。因此只要是任意键值型数据（比如 XorMM 中的索引，ChalametPIR 中的数据条目结构），都可以直接编码成向量的数据结构形式。
- 成员测试的能力：过滤器设计的初衷就是实现成员测试。正因为其拥有成员测试的能力，因此可以用于查询协议中做条件判断。比如在 HXT 中判断多个搜索关键词是否与文件标识匹配，ChalametPIR 中判断那些数据符合用户的请求。而对于异或型过滤器和不经意键值存储来说，它们还具有不经意的性质，也就是对任意输入都可以生成一个输出。比如在 XorMM 协议中服务器对 ℓ 个查询需要返回 ℓ 个输出，在 VOLE-PSI 协议中，发送方对集合中每个元素 x_i，都有对应的输出 $f'(x_i)$，在 SKE-PSU 协议中，发送方对集合中每个元素 x_i，都有对应的输出 s_i^*。这样不经意性保证让一方不获取任何消息的同时，为另一方提供成员测试的功能。

这一节的主要目的是希望通过介绍这些数据结构在隐私保护协议上的应用，让大家了解到现有的一些通用方法，为设计隐私保护协议提供新的思路。从我们列出的相关工作也可以看出，布隆过滤器和布谷鸟过滤器在隐私保护协议中已经应用得很成熟，而异或型过滤器和不经意键值存储的应用近几年才刚刚开始。我们可以多关注后两种数据存储结构，或许能为我们构建方案提供更好的帮助。

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] PATGIRI R, NAYAK S, MUPPALANENI N. Bloom filter: A data structure for computer networking, big data, cloud computing, internet of things, bioinformatics and beyond[M]. Academic Press, 2023.

[4] 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.

[5] 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.

[6] 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.

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

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

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

[17] 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.

[18] 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.

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

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

[21] 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.

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

[23] 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.

[24] 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.

[25] 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.

[26] 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.

[27] 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.

[28] 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.

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

[30] 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.

[31] 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.

[32] 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.

[33] 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.

[34] 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.

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

[36] 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.

[37] GRUBBS P, LACHARITE M S, MINAUD B, et al. Pump up the volume: Practical database reconstruction from volume leakage on range queries[C]//Proceedings of the 25th ACM Conference on Computer and Communications Security (CCS). ACM, 2018: 315-331.

[38] GUI Z, JOHNSON O, WARINSCHI B. Encrypted databases: New volume attacks against range queries[C]//Proceedings of the 26th ACM Conference on Computer and Communications Security (CCS). ACM, 2019: 361-378.

[39] 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.

[40] 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.

[41] 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.

[42] 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.

[43] CASH D, JARECKI S, JUTLA C, et al. Highly-scalable searchable symmetric encryption with support for boolean queries[C]//Proceedings of the 33rd Annual Cryptology Conference (CRYPTO). Springer, 2013: 353-373.

[44] WANG J, SUN S F, LI T, et al. Practical volume-hiding encrypted multi-maps with optimal overhead and beyond[C]//Proceedings of the 29th ACM Conference on Computer and Communications Security (CCS). ACM, 2022: 2825-2839.

[45] ANDO M, GEORGE M. On the cost of suppressing volume for encrypted multi-maps[J]. Proceedings on Privacy Enhancing Technologies, 2022, 2022(4): 44-65.

[46] BEIMEL A, ISHAI Y, MALKIN T. Reducing the Servers' Computation in PrivateInformation Retrieval: PIR with Preprocessing[J]. Journal of Cryptology, 2004, 17(2): 125-151.

[47] CELI S, DAVIDSON A. Call me by my name: Simple, practical private information retrieval for keyword queries[C]//Proceedings of the 31th ACM Conference on Computer and Communications Security (CCS). 2024.

[48] REGEV O. On lattices, learning with errors, random linear codes, and cryptography[J]. Journal of the ACM, 2009, 56(6): 1-40.

[49] DAVIDSON A, PESTANA G, CELI S. FrodoPIR: Simple, scalable, single-server private information retrieval[J]. Proceedings on Privacy Enhancing Technologies, 2023, 2023(1): 365-383.

[50] HENZINGER A, HONG M M, Corrigan-Gibbs H, et al. One server for the price of two: Simple and fast single-server private information retrieval[C]//Proceedings of the 32nd USENIX Security Symposium (USENIX Security). USENIX Association, 2023: 3889-3905.

[51] ZHOU M, LIN W K, TSELEKOUNIS Y, et al. Optimal single-server private information retrieval[C]//Proceedings of the 42nd International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT). Springer, 2023: 395-425.

[52] DAVIDSON A, CID C. An efficient toolkit for computing private set operations[C]// Proceedings of the 22nd Australasia Conference on Information Security and Privacy (ACISP). Springer, 2017: 261-278.

[53] CHEN Y, ZHANG M, ZHANG C, et al. Private set operations from multi-query reverse private membership test[C]//Proceedings of the 27th International Workshop on Practice and Theory in Public Key Cryptography (PKC). Springer, 2024: 387-416.

[54] RESENDE A C D, ARANHA D F. Faster unbalanced private set intersection[C]//Proceedings of the 22nd Financial Cryptography and Data Security (FC). Springer, 2018: 203-221.

[55] KOLESNIKOV V, ROSULEK M, TRIEU N, et al. Scalable private set union from symmetric-key techniques[C]//Proceedings of the 25th Annual International Conference on the Theory and Application of Cryptology and Information Security (ASIACRYPT). Springer, 2019: 636-666.

[56] GARIMELLA G, MOHASSEL P, ROSULEK M, et al. Private set operations from oblivious switching[C]//Proceedings of the 24th International Conference on Practice and Theory of Public Key Cryptography (PKC). Springer, 2021: 591-617.

[57] RINDAL P, SCHOPPMANN P. VOLE-PSI: Fast OPRF and Circuit-PSI from Vector-OLE [C]//Proceedings of the 40th International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT). Springer, 2021: 901-930.

[58] ZHANG C, CHEN Y, LIU W, et al. Linear private set union from multi-query reverse private membership test[C]//Proceedings of the 32nd USENIX Security Symposium (USENIX Security). USENIX Association, 2023: 337-354.