# 可验证对称加密搜索问题分析与研究

（申请清华大学工程硕士专业学位论文）

培 养 单 位: 计 算 机 科 学 与 技 术 系
学　　科: 计 算 机 技 术
申　请　人: 朱　洁
指 导 教 师: 李　琦　副　研　究　员

二〇一八年四月

# Analysis and Research of Verifiable Searchable Symmetric Encryption

Thesis Submitted to

**Tsinghua University**

in partial fulfillment of the requirement

for the professional degree of

**Master of Engineering**

by

**Zhu Jie**

**( Computer Technology )**

Thesis Supervisor : Professor Li Qi

**April, 2018**

# 关于学位论文使用授权的说明

本人完全了解清华大学有关保留、使用学位论文的规定，即：

清华大学拥有在著作权法规定范围内学位论文的使用权，其中包括：（1）已获学位的研究生必须按学校规定提交学位论文，学校可以采用影印、缩印或其他复制手段保存研究生上交的学位论文；（2）为教学和科研目的，学校可以将公开的学位论文作为资料在图书馆、资料室等场所供校内师生阅读，或在校园网上供校内师生浏览部分内容。

本人保证遵守上述规定。

**（保密的论文在解密后应遵守此规定）**

作者签名：_____　　导师签名：_____

日　　期：_____　　日　　期：_____

# 摘 要

云存储的发展使得用户可以方便地存储、获取与分享数据。但与此同时，云存储也带来了很多安全问题，例如，数据隐私泄露等等。对称加密搜索的提出解决了数据隐私泄露问题，同时也保证了数据的可搜索性。通过使用对称加密搜索方案，用户可以在上传数据到云服务器之前，对数据进行加密，同时云服务器可以在用户的加密数据上进行搜索，从而确保数据隐私。

然而，对称加密搜索的假定是云服务器是诚实且好奇的，即云服务器会遵守协议，但现实情况中云服务器往往是不可靠的。为了解决该问题，可验证对称加密搜索技术相应提出，它通过结果验证技术可以到检测云服务器的恶意行为。但是，现有的可验证对称加密搜索方案都不完善，例如，不支持用户数据动态更新，依赖于特定的对称加密搜索方案，只支持单用户读写等等。

针对以上的问题，本文提出了一种通用的可验证对称加密搜索框架，该框架普适于任何加密搜索方案，支持用户数据更新，并且能够同时在单用户和多用户的场景下工作。本文的主要工作和创新点包括：

- 提出了一种单用户场景下的可验证对称加密搜索框架，并在此基础上设计了结果验证算法，该算法能同时保证数据新鲜性和数据完整性。该框架支持用户数据动态更新，并且将验证索引从对称加密搜索方案中解耦，使其可以为任何加密搜索方案提供结果验证功能。
- 提出了一种多用户场景下的可验证对称加密搜索框架。该框架支持单用户写，多用户读，确保了多用户场景下的数据新鲜性，并实现了数据共享场景下的结果验证。
- 本文采用了一个开源数据集作为测试数据，在本地环境对该框架进行了实验测试。安全性分析和实验表明，本文提出的可验证对称加密搜索框架不泄露数据隐私，并且给对称加密搜索方案引入的额外计算开销和通信开销很小，几乎可以忽略不计。

**关键词：**对称加密搜索，结果验证，云存储

# Abstract

Cloud storage allows users to retrieve and share their data conveniently. Meanwhile, cloud storage also brings serious data privacy issues, i.e., the disclosure of private information. In order to ensure data privacy without losing data usability, Searchable Symmetric Encryption (SSE) has been proposed. By using SSE, users can encrypt their data before uploading to cloud services, and cloud services can directly operate and search over encrypted data, which ensures data privacy.

However, most SSE schemes only work with honest-but-curious cloud services that do not deviate from the prescribed protocols. This assumption does not always hold in practice due to the untrusted nature in storage outsourcing. To alleviate the issue, there have been studies on Verifiable Searchable Symmetric Encryption (VSSE), which functions against malicious cloud services by enabling results verification. But to our best knowledge, existing VSSE schemes exhibit very limited applicability, such as only supporting static database, demanding specific SSE constructions, or only working in the single-user model.

In this paper, we proposed a generic verifiable SSE framework in both single-user model and single-writer multiple-reader model, which provides verifiability for any SSE schemes and further supports data updates. In summary, our contributions are three-fold:

- We proposed a verifiable SSE framework in single-user model and designed the result verification algorithms. The framework seperate the verification index from the SSE construction and can provides generic verification for any SSE schemes. The algorithms guaranteed both data freshness and data integrity with support of data updates.
- We proposed the first verifiable SSE framework in the single-writer and multiple-reader model, which ensures data freshness across multiple users and provides result verification under data sharing scenario.
- We implemented our framework in a local enviroment and fed it with an open-source data set. Rigorous analysis and experimental evaluations show that our shceme is secure and introduces small overhead for result verification.

**Key words:** Searchable Symmetric Encryption; Result Verification; Cloud Storage

# 目 录

# 主要符号对照表

| | |
|---|---|
| SSE | 加密搜索 (Searchable Symmetric Encryption) |
| VSSE | 可验证加密搜索 (Verifiable Searchable Symmetric Encryption) |
| MPT | 默克尔帕特里夏树 (Merkle Patricia Tree) |
| IH | 增量哈希 (Incremental Hash) |
| $\mathcal{W}$ | 关键字集合 |
| $\|W\|$ | 关键字集合大小 |
| $w_i$ | 关键字，其中 $i \in \{1, \cdots, \|W\|\}$ |
| $\mathcal{D}$ | 明文文件集合 |
| $D_{w_i}$ | 包含关键字 $w_i$ 的明文文件集合 |
| $C$ | 密文文件集合 |
| $C_{w_i}$ | 包含关键字 $w_i$ 的密文文件集合 |
| $d$ | 明文文件 |
| $c$ | 密文文件 |
| $W_d$ | 文件 $d$ 包含的关键字集合 |
| $\|W_d\|$ | 文件 $d$ 包含的关键字集合大小 |
| $\tau$ | 搜索令牌 |
| $\lambda$ | 验证索引 |
| $\pi$ | 鉴别符 |

# 第 1 章　绪论

## 1.1　研究背景及选题意义

云存储使得用户可以随时随地地存取数据，并且极大地方便了用户之间的数据共享，降低了维护数据的成本[1–7]。但与此同时，云存储也带来了许多安全性问题，例如，数据丢失，数据隐私泄露等等。总体来说，云存储带来的安全性问题可以分为以下两类：

- 可用性问题。要求云服务器保证数据不丢失，用户可以将云端作为数据中枢进行数据备份和同步。目前，一般的云服务提供商都采用了多副本的方式保障数据的可用性，即将数据的多个副本分别写入其他的存储节点，当一个节点发生故障时，其他节点上的数据继续提供服务，同时通过其他节点中的数据副本，快速恢复故障节点上丢失的数据。目前，针对数据可用性的相关学术研究包括数据拥有证明 (Proof of Data Possession, PDP)[2,8–10] 以及数据可恢复性证明 (Proof of Retrievability, PoR)[1,5,11]。
- 隐私性问题。要求云服务器保证数据的隐私并且不泄露数据。目前，云服务提供商一般采用数据加密方式对隐私数据进行保护，但数据加密往往会导致数据可用性的降低，例如数据失去可搜索性。因此加密搜索 (Searchable Encryption, SE) 应运而生。加密搜索技术主要分为两类，一是对称加密搜索 (Searchable Symmetric Encryption, SSE)[12–16]，二是公钥加密搜索 (Public Key Encryption with Keyword Search, PEKS)[17]。

加密搜索的提出，使得用户可以在上传数据给云服务器之前，对其进行加密，并且使得云服务器可以在加密数据上进行搜索。从而既保证了数据隐私性，又保证了数据的可搜索性。目前，由于效率问题，应用较为广泛的为对称加密搜索技术。然而，大部分的对称加密搜索方案都基于服务器是诚实且好奇的假设[13–15]，即服务器会遵循协议但是可以从用户的查询中推断相关信息。这种假设在实际应用场景中往往是不成立的。因为云服务器可能会因为外部攻击，内部配置错误，软件错误等等问题而导致其违反原有协议[7,18]。这种协议违反所导致的最常见问题就是服务器返回的搜索结果不完整。例如，云服务器有可能为了节省计算开销和通信开销而返回少量搜索结果给用户，甚至有可能不返回搜索结果给用户。

为了解决该问题，可验证对称加密搜索技术也相应提出[3,18–24]。可验证对称加密搜索技术允许用户对搜索结果进行验证，从而来检测服务器的不诚信行为，保障加密搜索的正确性。然而，据我们所知，现有的可验证对称加密搜索方案都是不

完善的。例如，有的方案[19,20,23,24] 不支持数据更新，只能作用在静态数据库中，数据库若有变化则需要重建整个索引。有的方案[3,21,22] 无法防止服务器故意返回空结果来规避结果验证。换句话说，以上这些方案[3,21,22] 在用户提交的关键字不存在于数据库中时，是不返回任何搜索结果的，这就导致了服务器可以对任意关键字返回空结果来规避结果验证，除非用户在本地保留数据库的所有关键字集合。另外，大部分的可验证对称加密搜索方案[3,18–24] 仅仅支持在单用户场景下工作，即数据持有者自己写自己读的场景，而现实情况中，数据往往有共享需求，即一方写多方读的多用户场景①。表格 1.1比较了现有的可验证对称加密搜索方案。

## 1.2　本文的主要内容

本文基于默克尔帕特里夏树 (Merkle Patricia Tree, MPT) 和增量哈希 (Incremental Hash, IH) 技术，提出了一种单用户场景下的通用可验证对称加密搜索框架。该框架将验证索引从对称加密搜索方案中解耦，使其可以与任何对称加密搜索方案结合，包括但不限于论文[14,15,22] 中的方案。该验证索引基于支持动态更新的数据结构 MPT 构建，因此支持用户动态更新其数据集，而不需要重新构建验证索引。该验证索引将加密后的关键字和其对应的文件存储于叶子节点中，从而使得 MPT 的根节点成为用户数据完整性的见证，用于后续支持结果验证。同时，本文还提出了基于该验证索引的一系列验证机制，来确保数据完整性和数据新鲜性的验证。与以往的方案不同[3,21,22]，我们的方案要求服务器不管搜索关键字存在与否，都需要给用户返回一个“证明”，用于让用户验证服务器是故意返回了空结果还是搜索关键字的确不存在与现有数据集中。需要特别说明的是，我们的方案不需要用户在本地维护文件集对应的关键字集合。

此外，基于以上方案，本文利用时间戳链和公钥加密机制，首次提出了一种多用户场景下的通用可验证加密搜索框架。该框架通过时间戳链和公钥加密机制构建了出了与 MPT 根哈希相关的鉴别符，解决了多用户共享数据情况下的数据新鲜性验证问题，实现了多用户下的结果验证。

本文通过严格的安全性证明，确保了方案不泄露用户的数据隐私信息。另外，本文通过实验表明，单用户场景和多用户场景下的可验证加密搜索框架效率很高，与加密搜索法方案结合时，给加密搜索引入的额外开销很小，几乎可以忽略不计。

① 本文所述的多用户场景指一方写入，多方读取的场景，下文中若不做特别说明，均指这种情况。

表 1.1　现有可验证对称加密搜索方案比较

| | 动态性 [1] | 新鲜性 [2] | 完整性 [3] | 验证效率 [4] | 通用性 [5] |
|---|---|---|---|---|---|
| KPR11[3] | ✓ | ✓ | × | $O(\|W\|)$ | ✓ |
| KO12[19] | × | - | × | $O(n)$ | × |
| CG12[20] | × | - | ✓ | $O(log(\|W\|))$ | × |
| KO13[21] | ✓ | ✓ | × | $O(n)$ | × |
| SPS14[22] | ✓ | ✓ | × | $min\{\alpha + log(N), rlog^3(N)\}$ | × |
| CYGZR15[23] | × | - | × | $O(\|W\|) + O(r)$ | × |
| BFP16[18] | ✓ | ✓ | ✓ | $O(r)$ | ✓ |
| OK16[24] | × | - | ✓ | $O(r)$ | ✓ |
| 我们的方案 | ✓ | ✓ | ✓ | $O(log(\|W\|))$ | ✓ |

[1] 注意，动态性是指方案是否支持用户数据动态更新，由此可将可验证对称加密搜索方案分为静态和动态两种类型，后者在功能性上更完善。

[2] 注意，'×' 表示有实现的需求但是该方案没有实现, 而'-' 表示没有实现的需求。具体而言，静态的可验证对称加密搜索方案不存在数据新鲜性问题，因此方案[19,20,23,24] 也没有进行数据新鲜性验证的需求。

[3] 我们考虑各种数据完整性攻击，尤其包括服务器故意返回空结果来规避结果验证的场景。

[4] 验证效率是指服务器进行结果验证支持所需要的计算开销。对于表格中的非通用型方案[19–23] 来说，由于他们的方案并没有将验证索引从加密搜索方案中解耦，因此他们的验证效率和服务器进行加密搜索所需的计算开销是等价的。这里，$n$ 代表所有文件的数量, $|W|$ 表示所有关键字的数量, $r$ 表示包含某一特定关键字的文件数量, $\alpha$ 表示某一关键字历史上被加入到集合中的次数[22], $N$ 表示 (文件，关键字) 对的数量。

[5] 一个通用的可验证对称加密搜索方案是指该方案可以为任何加密搜索方案提供结果验证，而非通用的可验证对称加密搜索方案表示该方案仅支持在特定的加密搜索方案下工作。

## 1.3　本文的结构安排

本文的结构如下，第 1 章为绪论，介绍了研究背景、选题意义以及主要工作内容；第 2 章为相关研究综述，介绍了对称加密搜索、可验证对称加密搜索等相关工作的研究现状，并对本文用到的相关概念和先验知识进行了介绍；第 3 章为单用户下的可验证对称加密搜索方案研究，从适用场景、方案流程、算法分析、安全性证明和实验验证几个角度，完整的介绍了单用户场景下可验证对称加密搜索框架方

案；第 4 章为多用户下的可验证对称加密搜索方案研究，整体结构与第 3 章类似；第 5 章总结了全文，并对可验证加密搜索领域未来可能的发展方向进行了分析。

# 第 2 章　相关工作及问题定义

## 2.1　研究现状

### 2.1.1　安全云存储方案

可验证的云存储服务已经被广泛的研究过，例如，数据拥有性证明 (Proof of Data Possession, PDP)[2,8–10]，数据可取回证明 (Proof of Retrievability, POR)[1,5,11] 等等。这些方案主要侧重于云端存储数据的完整性验证，并支持丢失数据的恢复。注意，这些方案与加密搜索场景下的结果验证是不同的，因为加密搜索的结果验证不仅需要验证某个文件本身的完整性，还需要验证整个搜索结果集合是否完整。而这些方案只能单纯验证数据块的完整性，不支持对搜索结果完整性的验证。

### 2.1.2　安全加密搜索方案

加密搜索的概念首次由 Song 等人[12] 在 2000 年提出，他们的方案允许用户将加密后的数据集存储到云端，并同时保证用户在该加密数据集上进行搜索的能力。随后，加密搜索方案被广泛的研究，总体来说可以分为以下两个分支：对称加密搜索 (Searchable Symmetric Encryption, SSE) 和公钥加密搜索 (Public Key Encryption with Keyword Search, PEKS)。其中，最经典的对称加密搜索方案[13] 由 Curtmola 等人提出，他们的方案利用了明文搜索中的倒排索引的思想，并且他们对加密搜索的安全性进行了严格的定义，提出加密搜索方案至少要在面对一个被动敌手的情况下是安全可靠的。目前还有许多不同的对称加密搜索方案实现了不同的搜索功能。例如，动态对称加密搜索 (Dynamic SSE) 方案[14,15,22] 允许用户更新其数据集，支持关键字排序 (Ranked Keyword Search) 的对称加密搜索方案[25] 允许用户获取根据某一影响因子排序后的搜索结果。最经典的公钥加密搜索方案[17] 由 Boneh 等人提出，他们的方案利用了双线性映射技术。总体来说，公钥加密搜索方案的性能是远远低于对称加密搜索方案的。

### 2.1.3　可验证数据结构

可验证数据结构 (Authenticated data structure) 在不可信的云存储环境中，主要被用于验证数据块的完整性。典型的可验证数据结构包括：默克尔树 (Merkle Tree, MT)[26], 可验证哈希表 ( Authenticated Hash Table, AHT)[27] 以及可验证跳表 (Authenticated Skip List, ASL)[28,29]。其中，默克尔树是最常见的用于验证数据完整

性的数据结构，但是默克尔树对数据更新的支持不够灵活。采用默克尔树实现的可验证对称加密搜索方案[3] 由于没法在中间节点存储关键字信息，因此也不支持与关键字相关的搜索。可验证哈希表采用了 RSA 累加器 ① (RSA Accumulator) 方法来实现数据验证，但是它的搜索和更新性能都较低。具体而言，可验证哈希表的搜索与更新速度在毫秒级别，而我们采用的默克尔帕特里夏树的搜索更新速度在微秒级别。可验证跳表采用了类似多级链表的方式来实现，一定程度上提升了搜索性能，但如果它将关键字信息存储于搜索路径上，存储空间将比默克尔帕特里夏树大很多。

### 2.1.4　可验证对称加密搜索方案

由 Kamara 等人提出的 CS2 方案[3] 通过使用默克尔树构建验证索引来支持用户对搜索结果的验证。具体的做法是，以加密的关键字作为“键”，以该关键字对应的加密文件集合作为“值”，将该“键值对”存储在默克尔树的叶子结点上。用户在本地需要保留默克尔树的根哈希作为一个指纹信息。在进行结果验证时，用户需要通过其搜索的关键字本身及服务器返回的该关键字对应在默克尔树上的路径来重构出该根哈希，并与保留的根哈希进行比对，从而来进行结果验证。但是他们的方案无法检测服务器恶意返回空结果的情况。关键的原因是，当用户搜索的关键字不存在时，默克尔树上不会存在该关键字对应的路径，因此服务器无法返回任何信息给用户。解决该问题的一个简单的方法是在构建默克尔树时，将整个字典空间中所有可能的关键字集合都存储在默克尔树中，但这样做会导致大量的空间浪费。

近期，Kurosawa 等人提出了一系列可验证对称加密搜索方案[19,21,24]。但是他们的方案要么效率很低，要么不支持用户数据动态更新。其中方案[19] 需要线性搜索时间并且不支持数据动态更新。他们的扩展方案[21] 支持了用户数据更新，该方案通过消息验证码 (Message Authenticated Code, MAC) 来确保了数据完整性，通过 RSA 累计器确保了数据新鲜性，但是方案的搜索复杂度超过了线性时间，并且该方案需要用户在本地维护一个关键字集合来探测服务器故意返回空结果的情况，这将引入较大的空间开销。Ogata 等人也提出了一个通用的可验证对称加密搜索框架[24]，该方案可以为任何对称加密搜索方案提供结果验证服务，并且不需要用户自己在本地维护一个关键字集合，但是他们的方案仍然是一个静态的方案，即不支持用户数据更新。方案[20] [23] 也同样只是静态方案。

由 Stefanov 等人提出的方案[22] 采用了时间戳和消息验证码机制来实现了结

① RSA 为提出该算法的三个密码学家名字的首字母，分别为 Ron Rivest, Adi Shamir, 和 Leonard Adleman

果验证，但是他们的方案没法防御服务器故意返回空结果来规避结果验证的情况。Bost 等人提出的方案[18] 是目前为止最完善的普适性可验证对称加密搜索方案，但他们的方案在搜索时需要与服务器进行两轮通信，加密搜索和结果验证过程在服务器端无法并行进行，即用户需要在拿到加密搜索的结果后再与服务器进行通信来进行验证，这将导致较大的验证时延和通信开销，并且他们的方案同样也不支持多用户情况下的验证。

总体来说，一个完善的普适性可验证对称加密搜索方案首先应该支持数据新鲜性和数据完整性验证，尤其要关注搜索结果为空时的验证，这一点被大部分的方案忽略。其次该方案应该在支持结果验证的同时，尽量降低用户本身的存储和计算开销，例如不需要用户本身去维护一个本地关键字集合。另外，该方案还应该支持用户数据的更新，并且能够支持多用户场景下的结果验证。综上所述，在单用户场景中，现有的可验证加密搜索方案无法在保证验证效率的同时，完善地验证数据新鲜性和数据完整性。并且现有的方案都不能满足多用户场景下的对称加密搜索结果验证。这需要我们利用合理的数据结构，并设计合理的机制来设计一个普适的可验证对称加密搜索框架。

### 2.1.5　可验证公钥加密搜索方案

第一个可验证的非对称加密搜索方案[30] 由 Zheng 等人提出，他们的方案采用了基于属性的关键字（Attribute-based keyword，ABK），但是他们的方案也只适用于数据库静态的情况。基于他们的工作，Liu 等人又提出了一个更高效的可验证非对称加密搜索方案[31]，Sun 等人也提出了一个支持多关键字搜索的可验证公钥加密搜索方案[7]。然而，由于非对称加密本身的限制，他们的方案必不可少地需要引入一个可信第三方，并且搜索的性能大大低于可验证对称加密搜索方案。

### 2.1.6　多用户加密搜索方案

目前有一些多用户场景下的加密搜索方案[13,32–34]，但这些方案都不支持结果验证。Curtmola 等人在 2006 年即提出了一个基于广播加密的多用户加密搜索方案[13]，该方案允许数据持有者将数据分享给其他用户，并且数据持有者可以设定其他用户的访问控制权限，可以随时撤销或者新增用户。Yang 等人也通过双线性映射 (Bilinear Mapping) 技术提出了一种支持多用户读多用户写的方案[32]，但是该方案的搜索效率与数据集合的大小成正比，无法应用于数据量很大的场景中。Jerecki 等人随后又提出了一个多用户加密搜索方案[33]，然而该方案需要数据持有者和搜索用户进行频繁的交互，给数据持有者带来了很大的通信开销。近期，Sun 等人提

出了一个非交互式的多用户加密搜索方案[34]，该方案降低了数据持有者的通信开销，但他们的方案不支持用户数据更新。注意，这里我们需要强调，下文我们提出的多用户场景下的可验证对称加密搜索方案旨在为实现了多用户加密搜索的方案提供一个跨用户的结果验证，而不是旨在设计一个多用户方案。据我们目前所知，现有的可验证对称加密搜索方案都只支持单用户场景下的结果验证，而不支持多用户场景下的结果验证，因为多用户场景下的结果验证会面临更多的困难。例如，当数据在不同用户之间共享时，由于数据搜索用户无法探知数据持有者是否对数据集合进行了更新，因此一个恶意的服务器可以返回旧数据集合的搜索结果。除非数据持有者在每次更新时都通知所有的搜索用户，但这将会带来很大的通信开销。我们将在第 4 章具体讲述我们的多用户方案。

## 2.2　先验知识

### 2.2.1　增量哈希

增量哈希 (Incremental Hash, IH) 由 Bellare 等人提出[35]，并被已有的加密搜索方案[3] 所使用。增量哈希函数是一个抗碰撞的函数：$IH : \{0,1\}^* \rightarrow \{0,1\}^l$，两个随机字符串通过增量哈希函数相加或相减后，生成的哈希值不会产生碰撞。举例来说，假设 $D$ 是一个包含关键字 $w$ 的数据集合，它的增量哈希值为 $H$。当一个新数据 $d$ 加入到 $D$ 中后，新的数据集合变为 $D'$，即 $D + d$。对于原有数据集 $D$ 来说，数据 $d$ 的加入只是微小的变动。这使得增量哈希函数可以基于数据 $d$ 和现有哈希值 $H$，并通过“加法”操作快速的计算出新文件集 $D'$ 的一个抗碰撞哈希值，而不需要基于新文件集 $D'$ 重新计算哈希值，这使得哈希操作的性能得到了较大的提升。

### 2.2.2　默克尔帕特里夏树

默克尔帕特里夏树 (Merkle Patricia Tree, MPT) 最早在以太坊 [36,37] (Ethereum) 中提出，它将传统的字典树 (Trie Tree) 和默克尔树结合，使得该树同时具有查找和验证的功能。MPT 具有四种类型的节点，分别为空节点（Blank Node，BN），叶子节点（Leaf Node，LN），分支节点（Branch Node，BN）和扩展节点（Extension Node，EN）。其中空节点只是一个不存任何信息的节点，叶子节点存储了键值对（key-value pair），扩展节点也存储了键值对，但扩展节点的键值分别为其子节点的公共前缀和子节点的哈希值。分支节点有 17 个元素，其中前 16 个元素代表了该节点上有可能的分支，即 16 个十六进制数字，第 17 个元素为值。当某一个关键字在该分支节点匹配完成时，该关键字对应的值就存储该元素中。

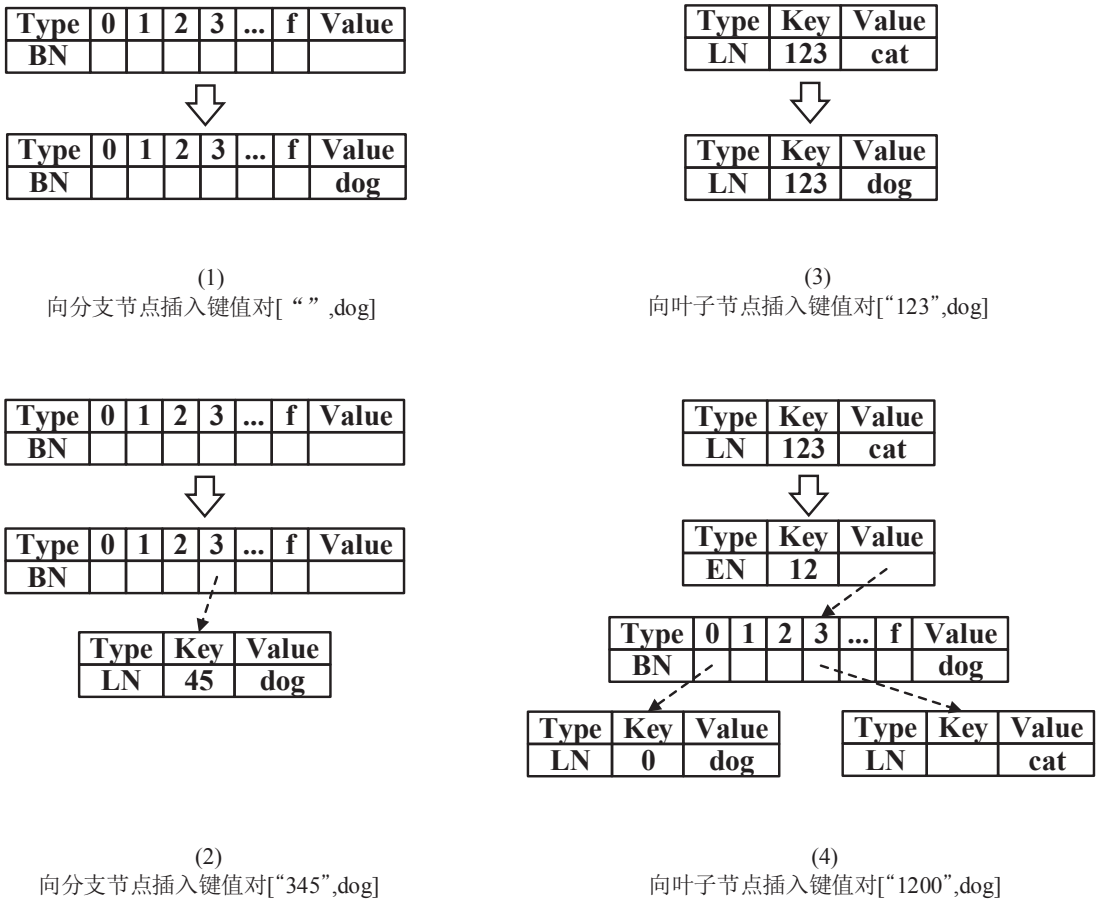


图 2.1　The Merkle Patricia Tree

图 2.1 通过四个简单的例子展示了 MPT 树的插入过程。首先是将一个“键值对”插入到分支节点，这分为两种情况。如果当前的键空间已经为空，我们可以直接将“值”插入到分支节点的第 17 个位置。否则，在经过了分支节点匹配后，键空间中“剩余键”和“值”将会存储在分支节点指向的一个新的叶子节点中。其次是将“键值对”插入到叶子节点，也分为两种情况。如果当前键空间中“剩余键”与叶子节点中的“键”正好匹配，直接将叶子节点中的“值”修改为新的“值”即可。否则，我们将找到当前键空间“剩余键”和叶子节点“键”的共同前缀，将其作为一个新建的扩展节点的“键”，并新建一个分支节点，将现有的叶子节点和新建的叶子节点作为子节点插入到分支节点对应的空间中。

注意，MPT 中的每一个节点都通过可递归长度前缀法[38](Recursive Length Prefix, RLP) 进行了编码并对编码值再进行了哈希。数据库中存储了每个节点的“键值对”键值对，其中“键”为该节点 RLP 编码的哈希，“值”为该节点的 RLP 编码。这样每个节点可以通过他的哈希值被引用，同时保证了 MPT 的可搜索性和可

验证性。通过这种方式，MPT 的根哈希成为了整棵树的指纹信息，根哈希的值由所有下层节点的哈希值所决定，任何节点的微小改变都会导致根哈希的值发生变化。此外，MPT 与默克尔树不同，MPT 是完全确定性的，即一组相同的“键值对”采用不同的顺序插入到 MPT 中，最终得到的根节点哈希值是相同的，而默克尔树不具有这个性质。

## 2.3　问题定义

在本节中，我们将正式定义方案的攻击模型，方案需要解决的问题以及方案需要实现的目标。

### 2.3.1　攻击模型

在单用户场景中，数据持有者和数据搜索用户是同一人，而在多用户场景中，这两者是分开的。我们假定数据持有者本身是可信的，而数据搜索用户不可信。此外，我们假定提供存储和搜索服务的云服务器是不可信的，即 1) 云服务器会试图从用户的加密数据和搜索请求中推断出一些隐私信息; 2) 云服务器有可能会因为外部攻击、配置错误、软件错误等原因背离原有协议，从而导致产生数据新鲜性攻击和数据完整性攻击，用以节省其自身的计算开销和通信开销。数据新鲜性攻击和数据完整性攻击的正式定义如下：

**定义 2.1 (数据新鲜性攻击)：** *在对称加密搜索中，数据新鲜性攻击是指一个恶意的云服务器试图从旧数据集中返回搜索结果，而不从最新的数据集中返回搜索结果。正式地，让 $\Delta_{n-1} = \{\delta_1, \delta_2, \cdots, \delta_{n-1}\}$ 代表用户数据集的历史版本，$\delta_n$ 代表用户的最新数据集，云服务器返回的搜索结果为 $\delta_i$ 的子集，其中 $1 \le i \le n-1$。*

**定义 2.2 (数据完整性攻击)：** *在对称加密搜索中，数据完整性攻击是指一个恶意的云服务器试图篡改搜索结果，阻止数据搜索用户获取到完整的搜索结果。正式地，让 $\tau$ 代表对称加密搜索方案中的搜索令牌，$\delta_i$ 代表数据集，其中 $1 \le i \le n$。对应的搜索结果应为 $\mathcal{F}(\delta_i, \tau)$，但云服务器返回的搜索结果为 $\mathcal{G}(\delta_i, \tau)$，其中 $\mathcal{G}(\delta_i, \tau) \neq \mathcal{F}(\delta_i, \tau)$。*

### 2.3.2　设计目标

本论文旨在设计一种普适的可验证加密搜索框架，即该方案可以和任意加密搜索方案相结合，包括但不限于[14,15,22]，使其能够完成结果验证的功能。本方案将现有的加密搜索方案当做黑盒，总体来说，需要满足以下几个需求：

1. **机密性：**数据和关键字的机密性是加密搜索最近本的安全需求。它保证了用户的明文数据和关键字信息无法被其他不可信第三方所推断。并且保证了敌手无法从方案的加密数据集，验证索引以及搜索关键字中推断出任何有用的隐私信息。
2. **可验证性：**一个可验证的对称加密搜索方案应该能够验证搜索结果的正确性和完整性，即防止重放攻击和数据完整性攻击。
3. **高效性：**一个可验证对称加密搜索方案应该达到次线性的计算复杂度，即对数复杂度 $O(log(|W|))$，其中 $|W|$ 是关键字的总数，并且应该在支持用户数据更新的情况下仍然能达到该复杂度。注意，这里的计算复杂度仅仅指服务器提供结果验证服务时所需的额外计算复杂度，不包括加密搜索方案本身带来的计算复杂度。

# 第 3 章　单用户下的可验证对称加密搜索方案研究

## 3.1　引言

本章提出了一种普适的可验证对称加密搜索框架 GS-VSSE，该框架可以在单用户场景下工作，与任意对称加密搜索方案结合后，可以为用户提供加密搜索的结果验证服务。本章的主要内容安排如下：首先介绍了单用户场景下的系统框架，介绍了该框架的参与方及其所承担的计算任务；接着，通过一个正式定义从抽象层面介绍了该框架工作的流程和每个参与方涉及到的算法。随后，对 GS-VSSE 框架涉及到的算法进行了详细分析，包括数据持有者构建和更新验证索引的算法，云服务器搜索验证索引并生成结果证明的算法，数据持有者进行结果验证的算法。随后通过一个简单的例子对这几个算法进行了详细的阐述。最后，通过安全性分析和实验结果分析证明 GS-VSSE 方案可以达到设计目标中的安全性要求和性能要求。

## 3.2　系统架构

单用户场景下的可验证对称加密搜索框架如图 3.1所示，数据持有者即为数据搜索用户本身。初始化时需要数据持有者对自身的数据集进行加密，并对该数据集构建加密的验证索引，用于后续结果验证。数据持有者将加密文件集和验证索引上传给云服务器存储，并在需要时更新数据集和验证索引。当用户需要进行关键字搜索时，他将会构建出一个与关键字相关的搜索令牌，提交给云服务器进行搜索。云服务器接收到该搜索令牌后，通过某个加密搜索方案取得加密搜索结果，同时通过搜索验证索引取得一个结果证明，将加密搜索结果和结果证明返回给用户。用户在收到搜索结果和结果证明后，对其进行结果验证，若验证失败，则丢弃该结果。

## 3.3　方案流程

**定义** 3.1 (**GS-VSSE 方案**)： *在 GS-VSSE 方案中，参与方有两个，分别为数据持有者本身和不可信的云服务器。数据持有者向云服务器提供加密数据集和验证索引，使得云服务器在用户搜索时可以向其返回结果证明，用于确保加密搜索结果的新鲜性和完整性。一个 GS-VSSE 方案是以下七个算法的集合:*

- $KGen(1^k) \rightarrow \{K_1, K_2\}$*: 是由数据持有者执行的秘钥生成算法。它将一个安全参数作为输入，输出对称秘钥* $K_1, K_2$。

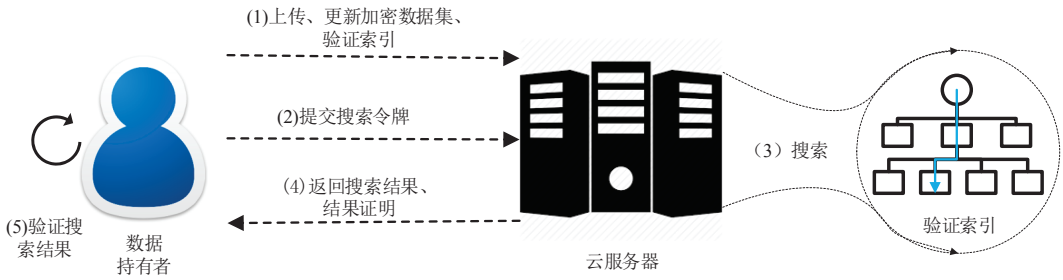


图 3.1　单用户场景下的可验证对称加密搜索框架 GS-VSSE

- $Init(K_1, K_2, \mathcal{D}) \rightarrow \{\lambda\}$: 是由数据持有者执行的初始化算法。它将对称秘钥 $K_1, K_2$ 和明文文件集 $\mathcal{D}$ 作为输入，输出验证索引 $\lambda$。数据持有者在本地保存验证索引 $\lambda$ 的根节点哈希 $rt$，并将验证索引 $\lambda$ 上传给云服务器。
- $UpdateToken(K_1, K_2, d) \rightarrow \{\tau_u\}$: 是由数据持有者执行的更新令牌生成算法。它将对称秘钥 $K_1, K_2$ 和需要更新的文件 $d$ 作为输入，输出一系列更新令牌 $\tau_u$。数据持有者将更新令牌 $\tau_u$ 上传给云服务器。
- $PreUpdate(\lambda, \tau_u) \rightarrow \{\lambda', \rho_u\}$: 是由云服务器执行的预更新算法。它将验证索引 $\lambda$ 和更新令牌 $\tau_u$ 作为输入，输出更新后的验证索引 $\lambda'$，和更新证明 $\rho_u$。云服务器将更新证明 $\rho_u$ 返回给用户。
- $Update(rt, \tau_u, \rho_u) \rightarrow \{rt'\}$: 是由数据持有者执行的更新算法。它将验证索引的根哈希 $rt$，更新令牌 $\tau_u$ 和服务器返回的更新证明 $\rho_u$ 作为输入，输出新的根哈希 $rt'$。若更新证明 $\rho_u$ 验证通过，则输出更新后的根哈希 $rt'$，若更新证明验证失败，则输出的根哈希 $rt'$ 与原始根哈希 $rt$ 相同。
- $SearchToken(K_1, w) \rightarrow \{\tau_w\}$: 是由数据持有者执行的搜索令牌生成算法。它将对称秘钥 $K_1$ 和某一关键字 $w$ 作为输入，输出与该关键字相关搜索令牌 $\tau_w$。数据持有者将该搜索令牌 $\tau_w$ 上传给云服务器进行搜索。
- $Prove(\lambda, \tau_w) \rightarrow \{\rho_s\}$: 是由云服务器执行的搜索算法。它将验证索引 $\lambda$ 和搜索令牌 $\tau_w$ 作为输入，输出结果证明 $\rho$。云服务器将结果证明 $\rho$ 返回给数据持有者。
- $Verify(K_1, K_2, C_w, \tau_w, rt) \rightarrow \{b\}$: 是由数据持有者执行的验证算法。它将对称秘钥 $K_1, K_2$，加密搜索结果 $C_w$，搜索令牌 $\tau_w$ 和保留的验证索引根哈希 $rt$ 作为输入，输出一个比特 $b$，代表接受或者拒绝该搜索结果。

注意，上述流程中的每一个算法 (除了 $Verify$ 算法)，都与加密搜索流程中的算法一一对应。例如 $Init$，$UpdateToken$，$PreUpdate$ 算法可以与加密搜索中的初始化

和更新操作同时进行，$Prove$ 算法可以与加密搜索中的搜索操作同时进行。该可验证加密搜索方案带来的额外算法是 $Verify$ 算法，它用于用户收到搜索结果后的验证操作。正式因为 GS-VSSE 方案的每一个算法都从加密搜索方案中解耦了出来，才使得该方案可以将加密搜索方案当做黑盒，并为任意加密搜索方案提供结果验证服务。

## 3.4　算法分析

本节，我们将具体阐述 GS-VSSE 方案，即单用户场景下的可验证加密搜索框架。首先我们将描述如何建立并更新验证索引，然后我们将给出服务器生成结果证明的方法，并详细解释用户如何利用结果证明来确保搜索结果的正确性。

### 3.4.1　构建及更新验证索引

**Algorithm 1** *Init* 算法

**Require:**

$K_1, K_2$: 对称秘钥; $\mathcal{D}$: 明文文件集合; $F, G : \{0,1\}^k \times \{0,1\}^* \rightarrow \{0,1\}^*$ 伪随机函数; $IH : \{0,1\}^* \rightarrow \{0,1\}^k$ 增量哈希函数; $H : \{0,1\}^* \rightarrow \{0,1\}^k$ 哈希函数

**Ensure:**

$\lambda$: 通过 MPT 构建的验证索引。

1: **for** each $w_i \in \Delta$, 其中 $\Delta$ 是包括了 $< w_i, D_{w_i} >$ 的倒排索引, $i \in \{1, \cdot, |W|\}$。 **do**
2: 　加密关键字作为“键” $\tau_{w_i} = F_{K_1}(w_i)$。
3: 　加密包含该关键字的文件集合作为“值” $V_{w_i} = \sum_{f_i \in D_{w_i}} IH(G_{K_2}(f_i))$。
4: 　向 MPT 插入键值对 $\lambda = \lambda.Insert(\tau_{w_i}, V_{w_i})$。
5: **end for**
6: **return** 返回从 MPT 构建得到的验证索引 $\lambda$。

算法 1 给出了建立验证索引的伪代码。首先数据持有者根据明文文件集 $\mathcal{D}$ 计算出倒排索引 $\Delta$，其中倒排索引 $\Delta$ 是指关键字 $w_i$ 与包含该关键字的文件 $D_{w_i}$ 组成的索引。对倒排索引中的每一个关键字 $w_i$，我们计算他的" 键值对"，其中“键”是每一个关键字通过伪随机函数生成的令牌，而“值”是包含该关键字的文件的增量哈希和。我们通过将这些“键值对”插入 MPT 中来形成验证索引。

对验证索引的更新操作支持三种方式，即插入、删除和编辑文件，其中编辑文件相当于删除一个文件后再新增一个文件。对于插入新文件操作，我们首先解析

**Algorithm 2** *Update* 算法

**Require:**

$rt$: 验证索引的根哈希；$\tau_u$：更新令牌；$\rho_u$: 更新证明；

**Ensure:**

$rt'$: 更新后的根哈希。

1: 将更新令牌 $\tau_u$ 解析为键值对 $(\tau_{w_i}, G_{K_2}(d))$，其中 $i \in \{1, \cdot, |W_d|\}$，$d$ 为待更新的文件。
2: **for** each $\tau_{w_i} \in \tau_u$ **do**
3: 　计算 $rt_t = Compute(\tau_{w_i}, \rho_u)$。
4: 　**if** $rt_t != rt$ **then**
5: 　　**return** 验证失败，返回原有根哈希 $rt$。
6: 　**end if**
7: **end for**
8: 计算 $rt' = Compute(\tau_u, \rho)$。
9: **return** 验证成功，返回更新后的根哈希 $rt'$。

该文件 $d$，得到该文件包含的关键字集合 $W_d$，对每一个关键字 $w_i \in W_d$，我们都用伪随机函数生成他的令牌 $\tau_(w_i)$，并将文件的伪随机结果 $G_{K_2}(d)$ 同时上传给云。云服务器收到后通过更新令牌 $\tau_(w_i)$ 找到对应的叶子节点，并将 $IH(G_{K_2}(d))$ 与原有的叶子节点的值相加。删除操作同样，只是将原有的叶子节点的值减去 $IH(G_{K_2}(d))$。云服务器在更新每一个令牌时，都需要将该令牌对应的搜索路径保存在更新证明 $\rho_u$ 中，用于后续发回给用户进行更新验证。

算法2展示了数据持有者在收到云服务器返回的更新证明 $\rho_u$ 后，执行的更新验证操作。由于数据持有者本身在本地并不保留验证索引 $\lambda$，只保留验证索引的根哈希 $rt$，因此在数据产生更新时，如何更新该根哈希 $rt$ 十分重要。因为云服务器是不可信的，数据持有者在提交了更新令牌 $\tau_u$ 后，无法确保服务器执行了正确的更新操作，因此他需要云服务器返回更新证明 $\rho_u$ 来进行验证。服务器返回的更新证明 $\rho_u$ 包含了更新令牌 $\tau_u$ 中每一个关键字令牌对应在验证索引 $\lambda$ 上的路径。用户在接受到该更新证明后，首先将自身生成的更新令牌 $\tau_u$ 解析为 $(\tau_{w_i}, G_{K_2}(d))$，随后对每一个令牌 $\tau_{w_i}$，验证是否根据更新证明 $\rho_u$ 生成原始根哈希 $rt$。若每个令牌都能验证成功，则用户通过更新令牌 $\tau_u$ 和更新验证 $\rho_u$ 构建新的根哈希 $rt'$，否则验证失败，用户保留原有根哈希 $rt$。在本章第3.4.4节，我们将通过一个例子来说明建立和更新验证索引的过程。

### 3.4.2　生成结果证明

如算法3所示，服务器根据用户提交的搜索令牌 $\tau_(w_i)$ 和验证索引 $\lambda$ 来生成结果证明 $\rho_s$。首先服务器根据搜索令牌 $\tau_(w_i)$ 来寻找搜索路径 $\sigma$。如果搜索令牌 $\tau_(w_i)$ 对应的叶子节点存在，即用户查询的关键字存在，则服务器从叶子节点的上一层节点开始，返回搜索路径上的“键”作为结果证明。注意对于分支节点，服务器还需要返回不在搜索路径上的“键值对”。如果搜索令牌 $\tau_(w_i)$ 对应的叶子节点不存在，即用户查询的关键字不存在，则服务器需要从搜索终结的节点开始向上返回搜索路径中的“键”作为结果证明，而对于搜索的终结节点，服务器需要返回完整的键值对。我们将在本章第3.4.4节，通过一个具体的例子来说明该过程。

**Algorithm 3** *Prove* 算法

**Require:** $\lambda$: 云服务器维护的验证索引; $\tau_{w_i}$: 用户提交的搜索令牌;

**Ensure:** $\rho_s$: 搜索结果的结果证明;

1: 查找搜索令牌 $\tau_{w_i}$ 在验证索引 $\lambda$ 上的对应路径 $\sigma = (n_0, \cdots, n_i, \cdots, n_m) \leftarrow \lambda.Search(\tau_{w_i})$, 其中 $n_i \in \{EN, BN, LN\}$, $n_0$ 为根节点。
2: **if** $t_{w_i}$ 在验证索引中存在 **then**
3:  **for** $i = m - 1$ to 0 **do**
4:  **if** $n_i = BN$ **then**
5:  $\rho_s = \rho_s \cup C_{n_i}$，其中 $C_{n_i}$ 包括分支节点中在搜索路径 $\sigma$ 上的" 键" 和不在搜索路径上的“键值对”。
6:  **else if** $n_i = EN$ **then**
7:  $\rho_s = \rho_s \cup C_{n_i}$，其中 $C_{n_i}$ 包括扩展节点中的“键”。
8:  **else**
9:  $\rho_s = \rho_s \cup C_{n_i}$，其中 $C_{n_i}$ 包括叶子节点中的“键值对”。
10:  **end if**
11:  **end for**
12: **else**
13:  **for** $i = m$ to 0 **do**
14:  Repeat steps 4-10
15:  **end for**
16: **end if**
17: **return** $\rho_s$

**Algorithm 4** Verify

**Require:** $K_1, K_2$: 对称秘钥; $C_w$: 加密搜索结果; $\rho_s$: 加密搜索结果证明; $\tau_w$: 用户提交的搜索令牌;$rt$：用户本身保留的根哈希；

**Ensure:** $b \in \{0,1\}$, 如果 $b = 1$, 表示结果验证成功，否则表示结果验证失败；

1: 计算剩余键 $\{remain\_key\}$ = String.match ($\tau_{w_i}$, keys in $\rho$)
2: **if** $C_w = \emptyset$ && $remain\_key = \emptyset$ **then**
3:　根据结果证明 $\rho_s$ 自底向上计算根哈希 $rt_t$；
4: **else if** $C_w \neq \emptyset$ && $remain\_key \neq \emptyset$ **then**
5:　计算 $\varphi = \sum_{f \in D_w} IH(G_{K_2}(f_i))$, 其中 $D_w$ 是 $C_w$ 对应的明文信息；
6:　计算叶子节点 $LN = Compute(\varphi, remain\_key)$
7:　根据结果证明 $\rho$ 和叶子节点 $LN$ 自底向上计算根哈希 $rt_t$.
8: **else**
9:　**return** 0
10: **end if**
11: **if** $rt = rt_t$ **then**
12:　**return** 1
13: **else**
14:　**return** 0
15: **end if**

### 3.4.3　结果验证

如算法 4 所示，当用户收到了结果证明 $\rho_s$ 时，就可以开始验证数据的新鲜性和完整性。首先用户通过搜索令牌 $\tau_{w_i}$ 与结果证明 $\rho_s$ 中的“键”进行匹配。如果结果证明 $\rho_s$ 中的“键”是搜索令牌 $\tau_{w_i}$ 的前缀，则 $remain\_key$ 存储搜索令牌 $\tau_{w_i}$ 与结果证明匹配完后剩余的键。如果结果证明 $\rho_s$ 中的“键”不是搜索令牌 $\tau_{w_i}$ 的前缀，那么 $remain\_key$ 就置为 $\emptyset$。如果搜索结果 $C_w$ 和 $remain\_key$ 都为空集，则我们通过结果证明 $\rho_s$ 直接计算出根哈希值 $rt_t$。如果二者都不为空，则我们首先通过搜索结果 $C_w$ 和 $remain\_key$ 生成叶子节点的哈希值，再通过结果证明 $\rho_s$ 重建出根哈希值 $rt_t$。除了这两种情况以外，我们就认为服务器故意返回了空结果或服务器篡改了结果证明的内容。最后，用户通过对比重建得到的根哈希 $rt_t$ 和用户本身保留的根哈希 $rt$ 是否相等来判断数据新鲜性和数据完整性。如果二者相等，则验证通过，如果二者不相等，则说明服务器少返回了搜索结果或者服务器篡改了结果证明。

### 3.4.4　实例分析

**The Inverted List**

| keyword | file | token | value |
|---|---|---|---|
| $w_1$ | $d_1$,$d_2$,$d_3$,$d_4$ | '43779' | $H_1$ |
| $w_2$ | $d_2$,$d_5$ | 'a5432' | $H_2$+IH($G_{K_2}(d_5)$) |
| $w_3$ | $d_1$,$d_2$,$d_3$ | 'a5cc1' | $H_3$ |
| $w_4$ | $d_1$,$d_2$,$d_4$ | 'ff48e' | $H_4$ |
| $w_5$ | $d_5$ | 'a5fab' | $H_5$=IH($G_{K_2}(d_5)$) |

图 3.2　一个简单的倒排索引

如图 3.2，图3.3，图3.4和图3.5所示，我们将通过一个解释性的实例来说明建立和更新验证索引 $\lambda$，生成更新证明 $\rho_u$ 和验证更新，以及生成结果证明 $\rho_s$ 和验证搜索结果的过程。

**建立并更新验证索引：**首先，我们假设数据持有者拥有四个文件，分别为 $d_1, d_2, d_3, d_4$，他们包含了四个关键字 $w_1, w_2, w_3, w_4$，其对应关系如图3.2中的倒排索引所示。由这四个文件构建的验证索引如图3.3所示。当包含关键字 $w_2$ 和 $w_5$ 的文件 $d_5$ 新增时，对于已经存在的关键字 $w_2$，云服务器只需将 $IH(G_{K_2}(d_5))$ 添加到原有的叶子节点上。而对于不存在的关键字 $w_5$，云服务器则需要创建一个新的叶子节点，并将 $IH(G_{K_2}(D_5))$ 作为他的节点值。

**生成更新证明和验证更新：**云服务器需要将两个涉及到更新的关键字 $w_2$,$w_5$ 对应的路径返回给用户，用以作为更新证明 $\rho_u$。如图 3.4所示，当用户拿到该更新证明后，对于每一个待更新关键字，首先需要确保该关键字对应的令牌或其前缀出现在更新证明中，然后需要根据更新证明重构出根节点哈希，用于跟用户持有的根哈希进行对比。只有当每一个待更新关键字对应的更新证明重构出的根哈希与原根哈希相同时，更新验证才通过。验证通过后，用户通过更新令牌 $\tau_u$ 和更新证明 $\rho_u$ 构造出更新后的根哈希 $rt'$，这确保了数据的新鲜性。

**生成结果证明和验证搜索结果：**结果证明 $\rho_s$ 的生成可以分为两种情况来讨论。第一种情况，假设用户想要搜索的关键字为 $w_2$，他提交的对应该关键字的挑战令牌为“a5432”。由于该关键字令牌在验证索引 $\lambda$ 中已经存在，云服务器可以找到与该令牌对应的搜索路径 BN1,EN1,BN2,LN3，根据 *Prove* 算法，服务器会返回除

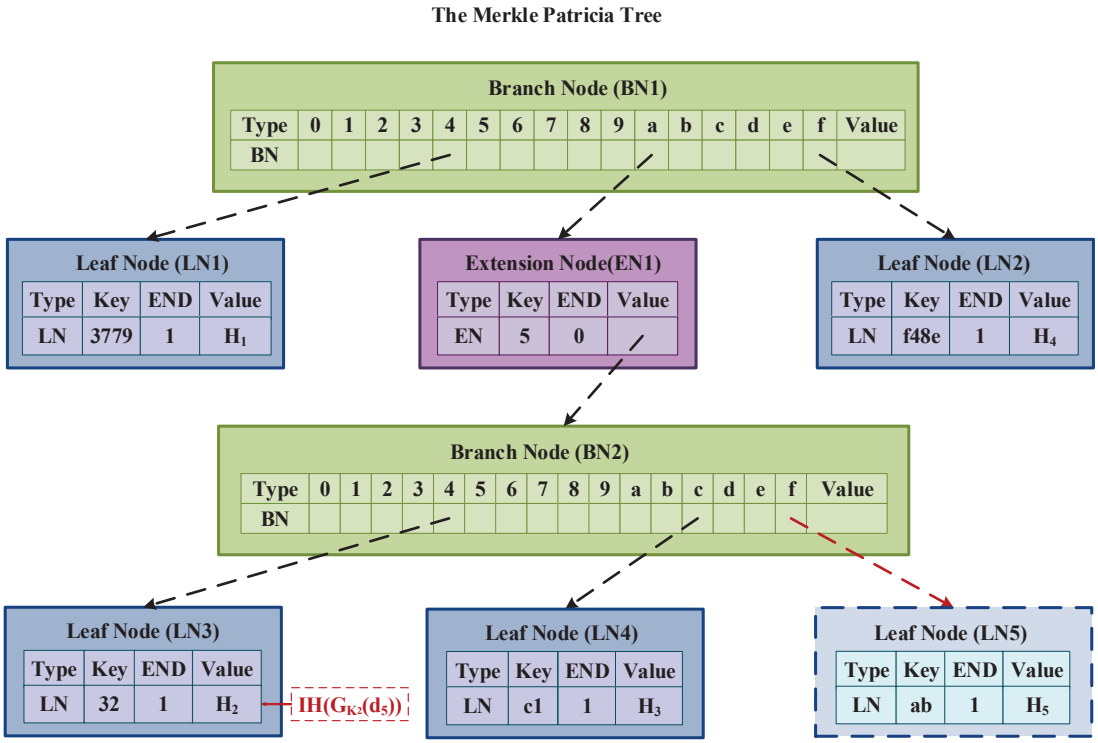


图 3.3　由倒排索引和 MPT 构建的验证索引

$LN3$ 以外的路径上的“键值对”作为结果证明，如 $C_{n2}, C_{n1}, C_{n0}$ 所示。用户在收到结果证明 $C_{n2}, C_{n1}, C_{n0}$ 以后，可以根据该证明 $\rho_s$ 和搜索结果 $C_w$ 重新构建根哈希。具体过程如下：首先用户将令牌“a5432”与结果证明中 $\rho_s$ 的“键”进行匹配，发现“a54”为令牌的前缀，因此剩余键 $remain\_key$ 为“32”。随后用户根据“32”以及搜索结果 $d_2, d_5$ 重新生成叶子节点 $LN3$，并通过结果证明 $\rho_s$ 自底向上构建冲根哈希的值。最后用户通过比较重构得到的根哈希和自身持有的根哈希，来判断数据是否完整。例如，假设云服务器只返回了文件 $d_2$，那么重构得到的根哈希将与正确的根哈希不匹配。第二种情况：假设用户搜索的关键字令牌为“a5433”，该令牌在验证索引 $\lambda$ 中不存在。根据云服务器的查找方法，其搜索路径与“a5432”相同，但不同的是，该令牌在叶子节点 $LN3$ 处发生了不匹配。因此云服务器需要从叶子节点 $LN3$ 开始自底向上生成结果证明，如图 3.5中的 $C_n3, C_n2, C_n1, C_n0$ 所示。用户在收到该结果证明以后，由于发现搜索令牌 $\tau_s$ 与结果证明 $\rho_s$ 中的“键”无法匹配，因此 $remain\_key$ 被置空。用户将直接根据结果证明 $\rho_s$ 重构根哈希。同样，用户将其与正确的根哈希进行对比，如果不相同，则说明服务器篡改了搜索结果或是结果证明，产生了恶意行为。

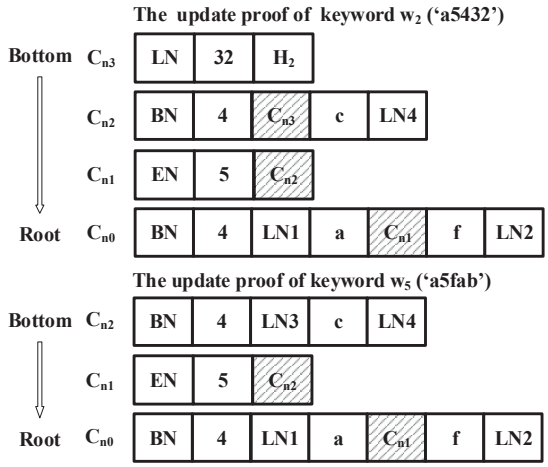


图 3.4　更新证明

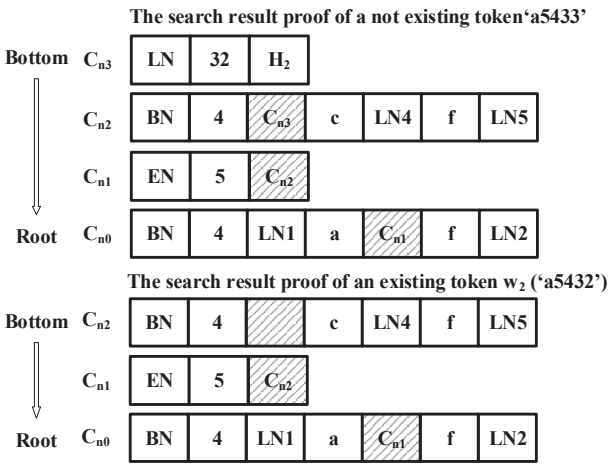


图 3.5　结果证明

## 3.5　安全性分析

在本节中，我们将对方案的安全性进行证明。方案的安全性主要分为两个部分，一个是机密性，另一个是可验证性。机密性是指敌手无法从验证索引 $\lambda$ 和用户发送的令牌 $\tau$ 中获取文件和关键字的明文信息。可验证性是指当服务器返回不完整或者错误的结果时，用户不会验证通过。

首先，我们采用基于仿真的博弈 (Simulation-based Game) 来证明方案的机密性。

**定义** 3.2 (GS-VSSE **机密性**)：令方案 *GS-VSSE* 是一个普适的、支持数据更新的可验证对称加密搜索方案，考虑以下概率性实验，其中 $\mathcal{A}$ 是一个有状态的敌手 *(state-fule adversary)*，$\mathcal{S}$ 是一个有状态的仿真者 *(stateful simulator)*，$\mathcal{L}$ 是有状态的泄露函数 *(leakage algorithms)*:

$\mathbf{Real}_{\mathcal{A}}(k)$: 一个挑战者采用 $KGen(1^k)$ 生成了对称秘钥 $K_1, K_2$。敌手 $\mathcal{A}$ 选择了一个文件集 $\mathcal{D}$ 让挑战者通过 $\{\lambda\} \leftarrow Init(K_1, K_2, \mathcal{D})$ 算法生成验证索引 $\lambda$。同时，敌手 $\mathcal{A}$ 生成了多项式数量级的自适应查询 $q = \{w, f\}$。对于每一个查询 $q$，敌手 $\mathcal{A}$ 从挑战者处收到一个搜索令牌 $\tau_w$ 和更新令牌 $\tau_u$，其中 $\tau_w \leftarrow SearchToken(K_1, w)$，$(\tau_u) \leftarrow UpdateToken(K_1, K_2, f)$。最后，敌手 $\mathcal{A}$ 返回一个比特 $b$。

$\mathbf{Ideal}_{\mathcal{A},\mathcal{S}}(k)$: 一个敌手 $\mathcal{A}$ 选择了一个文件集 $\mathcal{D}$。给定 $\mathcal{L}(\mathcal{D})$，仿真者 $\mathcal{S}$ 生成验证索引 $\lambda$ 发送给敌手 $\mathcal{A}$。敌手 $\mathcal{A}$ 生成了多项式数量级的自适应查询 $q = \{w, f\}$。对于每一个查询 $q$，仿真者 $\mathcal{S}$ 向其返回一个恰当的令牌 $\tau$。最后，敌手 $\mathcal{A}$ 返回一个比特 $b$。

如果对于任何概率多项式时间 *(Probabilistic Polynomial-Time, PPT)* 的敌手 $\mathcal{A}$,

始终存在一个概率多项式时间的仿真者 $\mathcal{S}$，使得，

$$|Pr[\mathbf{Real}_{A}(k)=1]-Pr[\mathbf{Ideal}_{A,S}(k)=1]|\leq negl(k). \tag{3-1}$$

则我们认为 *GS-VSSE* 是 $\mathcal{L}$-机密的。

在具体证明之前，我们首先给出敌手 $\mathcal{A}$ 能看到的内容，即泄露函数 $\mathcal{L}$ 的内容。$\mathcal{L}$ 定义如下：

$$\mathcal{L}(\mathcal{D})=(|\lambda|,\tau_q,\sigma) \tag{3-2}$$

其中 $|\lambda|$ 表示验证索引的大小，以叶子节点的数目来衡量。$\{\tau_q\}$ 表示由 $q$ 个查询产生的令牌，$\sigma$ 表示验证索引中的搜索路径。我们有以下的定理：

**定理** 3.1： 如果 $F$,$G$ 都是伪随机函数，那么方案 GS-VSSE 就是 $\mathcal{L}$-机密的。

**证明** 我们将证明，对于任何概率多项式时间内的敌手 $\mathcal{A}$，都存在一个概率多项式时间内的仿真者 $\mathcal{S}$，使得真实游戏 $\mathbf{Real}_{\mathcal{A}}(k)$ 和仿真游戏 $\mathbf{Ideal}_{\mathcal{A},\mathcal{S}}(k)$ 在计算上是无法区分的 (computationally indistinguishable)。

首先，给定 $\mathcal{L}(\mathcal{D})=(|\lambda|,\tau_q,\sigma)$，$\mathcal{S}$ 通过选择 $|\lambda|$ 个随机“键值对”插入 MPT 中生成一个仿真的验证索引 $\tilde{\lambda}$。由于在真实的验证索引中，“键值对”都是采用了伪随机函数 $F$,$G$ 进行了伪随机化的，因此敌手 $\mathcal{A}$ 将无法区分出真实的 $\lambda$ 和仿真的 $\tilde{\lambda}$。

模拟搜索令牌时，对于第一个令牌 $\tau_w$，如果它与 $\{\sigma\}$ 中的某一路径匹配，那么 $\mathcal{S}$ 就在 $\tilde{\lambda}$ 中选择任意一条路径作为令牌 $\tilde{\tau_w}$ 发送给敌手 $\mathcal{A}$，否则 $\mathcal{S}$ 就选择不在 $\tilde{\lambda}$ 路径中的随机字符串作为令牌 $\tilde{\tau_w}$ 发送给敌手 $\mathcal{A}$。对于后续的令牌，如果 $w$ 之前出现过，那么令牌 $\tilde{\tau_w}$ 就和之前发送给敌手 $\mathcal{A}$ 的维持一样。如果 $w$ 没出现过，那么令牌的生成方式就和第一个令牌的生成方式一样。由于令牌采用了伪随机函数 $F$ 进行了加密，因此敌手 $\mathcal{A}$ 也无法区分真实的令牌和仿真的令牌。

模拟更新令牌时，更新令牌被设置为 $\tilde{\tau_u}=(\tilde{\tau_{w_1}},\cdots,\tilde{\tau_{w_{|W_d|}}},\tilde{\tau_r})$。对每一个更新令牌 $\tilde{\tau_{w_i}}$，其中 $i\in\{1,\cdots,|W_d|\}$，$\mathcal{S}$ 的模拟方法与模拟搜索令牌方法相同。由于每一个更新令牌都采用了伪随机函数 $F$ 进行了加密，并且文件 $d$ 采用了伪随机函数 $G$ 进行了加密，因此敌手 $\mathcal{A}$ 无法区分真实的令牌、文件与仿真的令牌、文件。

因此我们可以得到结论：真实实验 $\mathbf{Real}_{\mathcal{A}}(k)$ 和仿真实验 $\mathbf{Ideal}_{\mathcal{A},\mathcal{S}}(k)$ 的输出结果是不可区分的。 □

GS-VSSE 方案的可验证性意味着该方案可以验证数据的新鲜性和完整性，即防御定义 2.1和 2.2提出的两种攻击。这里我们采用了一个基于游戏 (game-based) 的安全性定义来证明 GS-VSSE 方案的可验证性。

**定义** 3.3 (GS-VSSE **可验证性**)： 令方案 *GS-VSSE* 是一个普适的、支持数据更新的可验证对称加密搜索方案，考虑以下概率性实验，其中 $\mathcal{A}$ 是一个有状态的敌手 *(statefule adversary):*

$\mathbf{Vrf}_{\mathcal{A}}(k)$*:*

1. 挑战者通过 $KGen(1^k)$ 生成对称秘钥 $K_1, K_2$。
2. 敌手 $\mathcal{A}$ 给挑战者选择一个文件集合 $\mathcal{D}$。
3. 挑战者通过 $\{\lambda\} \leftarrow Init(K_1, K_2, \mathcal{D})$ 生成一个验证索引 $\lambda$。
4. 给定 $\lambda$ 和对算法 $SearchToken(K_1, w)$, $UpdateToken(K_1, K_2, d)$ 的预言权限, 敌手 $\mathcal{A}$ 输出一个关键字令牌 $\tau'_w$, 和一系列加密文件 $C'$，其中 $C' \neq C_w$，同时输出结果证明 $\rho'_s$.
5. 挑战者计算 $b := Verify(K_1, K_2, C', \rho'_s, \tau'_w)$.
6. 实验的输出为一个比特 $b$.

如果对于任何概率多项式时间 *(Probabilistic Polynomial-Time, PPT)* 的敌手 $\mathcal{A}$，

$$Pr[\mathbf{Vrf}_{\mathcal{A}}(k) = 1] \leq negl(k). \tag{3-3}$$

成立，则我们认为 *GS-VSSE* 方案是可验证的。

**定理** 3.2： 如果哈希函数 $H$ 和增量哈希函数 $IH$ 是抗碰撞的，并且 $G$ 是伪随机函数，那么 GS-VSSE 方案就是可验证的。

**证明** 考虑服务器返回的搜索结果为 $\tilde{D_w}$，而正确的搜索结果为 $D_w$，其中 $\tilde{D_w} \neq D_w$。但是用户的 $Verify$ 算法通过了 $\tilde{D_w}$ 作为正确搜索结果的情况。对于 $Verify$ 算法，这种情况的产生存在两种可能。第一种是在计算 $\tilde{D_w}$ 和 $D_w$ 对应的伪随机值和增量哈希值时产生了碰撞，第二种是在生成根哈希的路径中产生了哈希碰撞。不管是哪一种，都可以推出哈希函数产生了碰撞或者伪随机函数产生了碰撞，但是哈希函数产生碰撞或者伪随机函数产生碰撞的可能性是小于一个可忽略的值的，因此，我们的方案是可验证的。 □

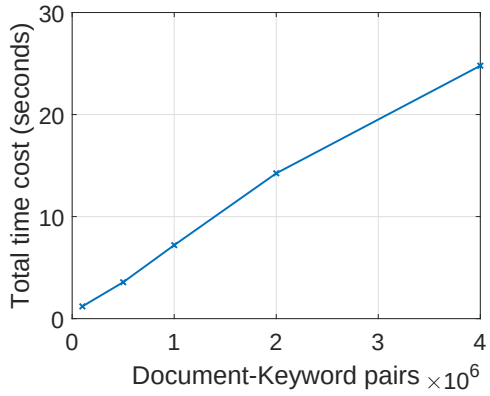


图 3.6　*Init* delays

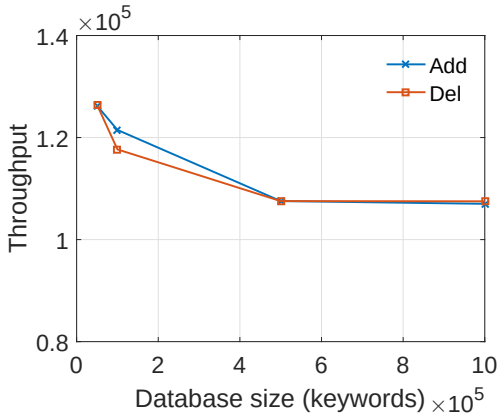


图 3.7　*PreUpdate* throughput

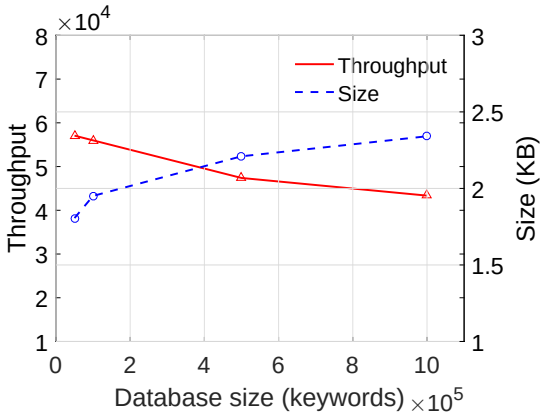


图 3.8　*Prove* cost

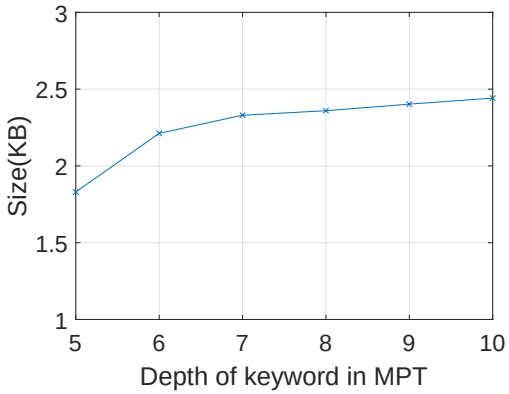


图 3.9　Proof cost of MPT

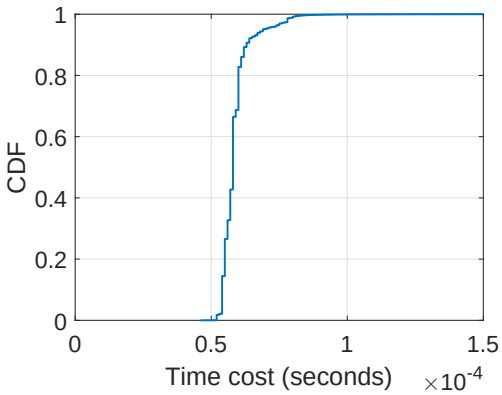


图 3.10　*Verify* performance

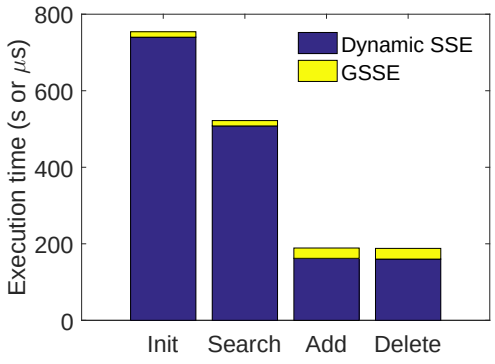


图 3.11　Comparison with SSE[15]

## 3.6　实验结果

### 3.6.1　实验设置

为了证明方案 GS-VSSE 的有效性，我们通过 Crypto++ 5.6.5 库实现了方案的原型。原型系统包含大约 2200 行代码。我们使用 HMAC-SHA256 作为两个随机预言 (random-oracle)，使用 SHA3-256 作为哈希函数，使用 MuHash 作为增量哈希函数。我们的实验在一台处理器为 Intel Core i5 2.5GHz，内存为 4G 的笔记本上进行，使用单线程实现。

我们使用一个开源数据集 Enron email dataset[39] 作为实验的测试数据集，使用了其中从“allen-p”到“kaminski-v”之间的数据。我们从该数据集中提取出了大量的“文件-关键字”对 (document-keyword pairs)，并通过 python 脚本为他们构建出了明文的倒排索引。注意，从文件中提取关键字的时延并没有被考虑在实验评估中，因为该问题与我们研究的 GS-VSSE 方案是一个独立的问题。

下文中，我们首先评估了 GS-VSSE 方案的算法效率，然后将 GS-VSSE 方案与一个著名的对称加密搜索方案[15] 进行了结合，以此来证明 GS-VSSE 方案引入的结果验证的开销并不大。注意，如无特别说明，下文中的每一个实验结果都是十次实验的平均值。

### 3.6.2　实验结果

首先，我们评估了 *Init* 算法的时延，即数据持有者生成验证索引 $\lambda$ 所需的时间。如图 3.6所示，生成验证索引 $\lambda$ 所需的时间与“文件-关键字”对的数量大小成正比，因为向 MPT 中执行的插入操作与“文件-关键字”对的数量相同。总体来说，*Init* 算法在“文件-关键字”对达到 400 万的时候，可以在 25 秒内执行完毕。由于 *Init* 算法仅需要在初始化时执行一次，因此这个开销是可以接受的。

云服务器更新验证索引的时间如图 3.7所示，更新的时延与数据库的大小有关，即与验证索引的大小有关，而验证索引的大小由它包含的关键字数量来衡量。严格意义上来书，一次更新的时延与 MPT 树的层数有关。为了更好的展示更新时延与验证索引大小的关系，我们使用了大量的关键字来评估更新时延。由于每一个文件包含的关键字个数不同，这里我们采用吞吐量 (throughput) 来衡量每秒钟云服务器可以更新的“文件-关键字”对。从图中可以看到，*Add* 和 *Del* 操作的性能几乎相同。当数据库的大小增大时，吞吐量将会降低。当数据库的大小为 100 万个关键字时，云服务器每秒钟可以同时支持 110,000 此更新操作。同时，我们也测量了用户端 $UpdateToken$ 算法引入的带宽开销，更新令牌 $\tau_u$ 中每一个关键字对应的密文带来的平均开销大小在 32 字节左右，而更新令牌的总大小与待更新文件 $d$ 包

含的关键字个数有关，这也是可以接受的。

如图 3.8所示，云服务器可以在数据库大小在 100 万关键字时，每秒钟进行 43,000 次 *Prove* 操作，这意味着云服务器可以同时支持 43,000 个来自用户的查询请求。由 *Prove* 操作产生的结果证明 $\rho_s$ 的大小也可以在图 3.8和图 3.9中看到，结果证明的大小随着 MPT 层数的增加而增长，但总体而言，只需要几千字节。

我们通过累计分布函数图 (Cumulative Distribution Function, CDF) 来对用户端进行的 $Verify$ 操作进行了测试。结果显示，几乎 99.7% 的情况下，用户端都可以在 0.1 毫秒内完成结果验证，这是可以接受的。

除此以外，我们还对 MPT 的存储空间进行了评估。如果我们使用一个 100 万个关键字的数据库，MPT 的存储空间大小大约为 82MB，而该数据库本身所占用的空间大小为 590MB，因此 MPT 带来的额外存储开销不算特别大。特别需要说明的是，这里我们评估数据库大小所采用为关键字密集型的数据，即邮件数据。如果数据持有者需要加密的是媒体类型的数据，例如图片或是音乐文件等等，这些文件只包含少量的关键字和属性，因此为这些类型的数据构建验证索引时，验证索引占原数据集的比例将会很小，甚至可以忽略。

### 3.6.3　与 SSE 方案的对比

我们将我们的普适性可验证对称加密搜索方案 GS-VSSE，与 Cash 等人提出的一个较为知名的动态对称加密搜索方案 (Dynamic Symmetric Searchable Encryption, DSSE)[15] 进行了结合，并展示了 GS-VSSE 为其提供结果验证服务带来的额外开销并不大。

为了公平的进行性能比较，我们使用了同样的数据集，并在同样的设备参数下对两个方案联合进行了实验。如图 3.11所示，我们测量了 *Init* 阶段，*Search* 阶段和 *PreUpdate* 阶段的性能开销。图中，*Init* 操作使用了 200 万个“文件-关键字”来分别构建 DSSE 方案[15] 和方案 GS-VSSE 用到的索引，时间单位为秒。而其他三个操作 *Search*, *Add*, *Delete* 的评估采用的数据库大小为 10,000 个关键字，时间单位为微妙。注意，DSSE 方案[15] 中的 *Search* 操作与 GS-VSSE 方案中的 *Prove* 操作对应。从图中可以看出，我们的 GS-VSSE 方案引入的额外开销非常小。其中，相较于 DSSE 方案[15] 而言，方案 GS-VSSE 在 *Init* 阶段引入的开销非常小，仅仅额外引入了 1.9% 的额外开销。而对于一次 *Search* (*Prove*) 操作来说，GS-VSSE 方案给云服务器引入的额外开销为 14 微秒，仅仅给 DSSE 方案带来了 2% 的额外开销。同样的，对于一次 *Add* 或 *Delete* 操作来说，我们的 GS-VSSE 方案仅仅引入了 27 微秒的时间，仅占方案[15] 的 17%。

在表 3.1中，我们还比较了二者的通信开销。由于数据的方差较大，每一个实验结果都是 50,000 次实验的平均值。结果显示，DSSE 方案[15] 的搜索结果大小平均为 53KB 左右，而我们的结果证明大小仅仅为 3KB 左右，即由 GS-VSSE 方案引入的额外开销低于 6%。此外，DSSE 方案[15] 生成的令牌大小平均为 390B，而 GS-VSSE 方案生成的令牌仅为 32B，即由 GS-VSSE 方案引入的额外开销低于 9%。这些实验结果充分表明了，GS-VSSE 方案是实用且高效的。

表 3.1　Comparison with the SSE scheme proposed by Cash et al.[15]

| Communication cost | SSE[15] | GS-VSSE |
|---|---|---|
| Search token | 390 Bytes | 32 Bytes |
| Search result/proof | 53 Kilobytes | 3 Kilobytes |

# 第 4 章　多用户下的可验证对称加密搜索方案研究

## 4.1　引言

## 4.2　系统架构

## 4.3　方案流程

三方

## 4.4　算法分析

### 4.4.1　构建时间戳链

In order to prevent cloud services from replaying previous authenticators and ensure the freshness of the root, we maintain a timestamp-chain for authenticators, such that users can trace authenticators in the chain and identify if the root is fresh. Here, the timestamp-chain scheme is different from the timestamp mechanism used in SSE[22]. Their schemes can only prevent servers from constructing data freshness attacks under the two-party model when the user holds the update information. Unfortunately, it may not be able to detect data freshness attack in our concerned three-party model. We show the details of our scheme below. Firstly, we will show how the data owner creates the authenticator by leveraging the timestamp-chain. In the very beginning, a data owner sets an interval for authenticator update①, and then the fixed update time points are set to be $\{up_1, up_2, \cdots, up_i, \cdots, up_m\}$ (see Fig. 4.2). Here, we use Network Time Protocol (NTP)[? ? ] run on cloud servers to synchronize the clocks among the data owners and the data users during their interactions with the servers. The clock synchronization accuracy can reach a few milliseconds or even tens of microseconds[? ? ? ], and thus the accuracy is enough for verification in GSSE. Note that, a malicious server can possibly fake a clock, but it cannot fake the timestamp-chain. If the server faked the clock, the timestamp will not allow a server to bypass the verification performed by users. The authenticator is uploaded to the server periodically at the update time point when there is no data update in an update interval. Otherwise, the data owner will additionally upload the authenticators along with the updating data.

① In the performance evaluation section, i.e., Section **??**, we will show the relationship between the update interval and the delays of detecting data freshness attacks.

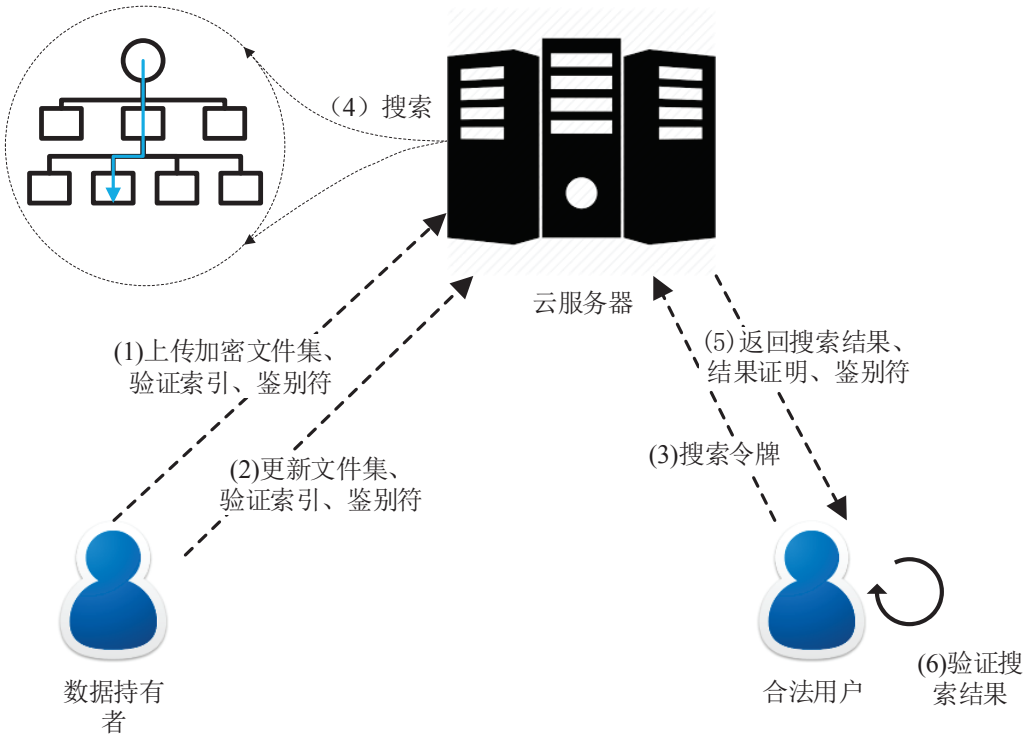


图 4.1　多用户场景下的可验证对称加密搜索框架 GM-VSSE

Intuitively, in order to prevent the authenticator from being replayed, we can simply set the authenticator $\pi$ as a concatenation of timestamp $tp$ and the root of the MPT, i.e., the proof index, encrypt it by using a symmetric key $K_3$, and then sign it with the secret key $ssk$. If the proof index is not updated during an update interval, the data owner only needs to update the timestamp at next update time point. If the document set of the data owner is modified within an update interval, which means the root of the proof index has been updated, then the data owner will calculate a new authenticator by using the latest root and the current timestamp and upload it to cloud services again. In this setting, when a data user generates a challenge, the server should send the latest authenticator to the data user. The data user can recover the root and the timestamp by decrypting the authenticator. If the timestamp is beyond the valid time, i.e., before the latest update time point, then the server is considered as malicious. This mechanism ensures that the server cannot mount a data freshness attack by using the data before the latest update time point.

However, cloud services may still be able to replay the authenticator between the latest update time point and the current query time. Specifically, if there is one or more

data updates happened after the latest update time point, then the server can cheat the data user by sending any authenticator uploaded after the latest update time point and then mounts a data freshness attack within the latest update interval. Therefore, we develop a timestamp-chain mechanism to detect those cheating behaviors. We modify the structure of the authenticator by chaining the value of the previous authenticator into the newly generated authenticator according to Equation (1). Note that, it will generate a new timestamp-chain of a new update interval, while the timestamp-chain ends at the beginning of the next update interval. In other words, the authenticators in each update interval are chained together, e.g., $\pi_{i,0}, \pi_{i,1}$ (see Fig. 4.2), but the authenticators are irrelevant in the two different update intervals. Here, the last authenticator in each update interval is uploaded at the next update time point. In this setting, the server needs to provide an authenticator at the query time and meanwhile an authenticator at the checkpoint, where the checkpoint is referred as the next update time point closest to the user's query time $t$, e.g., $up_{i+1}$ is the checkpoint in the update interval $(up_i, up_{i+1}]$.

$$\begin{cases} \pi_{i,0} = (\alpha_{i,0}, \mathsf{Sig}_{ssk}(\alpha_{i,0})), & up_i < tp_{i,0} \leq up_{i+1} \\ \alpha_{i,0} = Enc_{K_3}(rt_{i,0}||tp_{i,0}) & \\ \cdots & \\ \pi_{i,j} = (\alpha_{i,j}, \mathsf{Sig}_{ssk}(\alpha_{i,j})), & tp_{i,j-1} < tp_{i,j} \leq up_{i+1} \\ \alpha_{i,j} = Enc_{K_3}(rt_{i,j}||tp_{i,j}||\alpha_{i,j-1}) & \\ \cdots & \\ \pi_{i,n} = (\alpha_{i,n}, \mathsf{Sig}_{ssk}(\alpha_{i,n})), & tp_{i,n} = up_{i+1} \\ \alpha_{i,n} = Enc_{K_3}(rt_{i,n}||tp_{i,n}||\alpha_{i,n-1}) & \end{cases} \tag{4-1}$$

Here $i$ represents the $i$-th update interval and $j$ represents the $j$-th authenticator in the interval.

Let us consider the following cases (shown in Fig. 4.2) when a data user initiates a query at different time points: (i) the first case is that the query occurs at $t_1$, where $t_1 < tp_{i,0}$, the server can only send $\pi_{i-1,n}$ to the data user; (ii) the second case is that the query occurs at $t_2$ after the data update event at $tp_{i,0}$, and the authenticator that server sends to the user is $\pi_{i,0}$; (iii) the last case is that the query is generated at $t_2$, and the authenticator sent by the server is $\pi_{i-1,n}$. In the last case, a data freshness attack occurs, but it will be detected at the checkpoint $up_{i+1}$. The data user will obtain the last authenticator $\pi_{i,1}$ from

the server at the checkpoint to verify whether the data obtained at the query time has been replayed or not.

### 4.4.2　验证时间戳

Algorithm 5 shows the pseudo-code of the *Check* algorithm that is executed by a data user and verifies whether the authenticator has been replayed. Let $\pi_q^t$ denote the authenticator received at the query time $t$ and $\pi_c$ denote the authenticator received at the checkpoint, which is used to deduce the previous authenticators during the latest update interval. First, we need to verify the signature of $\pi_q^t$ and $\pi_c$ by using the public key $spk$ of the data owner. We check the authenticator $\pi_q^t$ received at the query time is not generated before the previous update time point by using $\alpha_q^t$ extract from $\pi_q^t$. Then, we decrypt the previous $rt_k||tp_k||\alpha_{k-1}$ concatenation by using $\alpha_k$ until it finds the first concatenation with timestamp $tp_k < t$ or $\alpha_k = \emptyset$. We compare $\alpha_k$ with $\alpha_q^t$ and $\emptyset$. If it is not equal to either of them, a data freshness attack is detected. Otherwise, $\alpha_q^t$ is considered correct. Now we use the three cases above to explain the algorithm. In the first case, $\pi_{i,1}$ and $\pi_{i,0}$ are received and $\alpha_{i,1}$ and $\alpha_{i,0}$ are extracted. We can find the field of $\alpha$ in the concatenation is $\emptyset$ after decrypting $\alpha_{i,0}$. Therefore, the *Check* algorithm outputs $b = 1$ and the authenticator $\pi_{i-1,n}$ received in the query time is considered correct. In the second case, $\alpha_{i,0}$ is also decrypted by $\alpha_{i,1}$ and the timestamp of $\alpha_{i,0}$ is less than $t_2$. We can find that $\alpha_{i,0}$ and $\alpha_q^{t_2}$ are equal. Hence $\alpha_q^{t_2}$ is considered correct, i.e., $\pi_q^{t_2}$ is correct. However, in the last case, we will detect a data freshness attack due to the mismatch between the correct authenticator $\pi_{i,0}$ and the received one $\pi_q^{t_2}$, i.e., $\pi_{i-1,n}$.

**Remark.** The update interval can be controlled by the data owner according to its update frequency. Normally, if data is frequently updated, the update interval can be set to a shorter period so that the length of the authenticator will decrease and the verification delays will be shorter. However, it will incur more communication overheads. In our experiments (see Section **??**), we will show that the verification delays and the bandwidth consumption for updating authenticators are acceptable.

### 4.4.3　实例分析

图 4.2　An illustration of the timestamp-chain mechanism

**Algorithm 5** Check

**Require:** $K_3$: the symmetric key; $spk$: the public key for verifying signature; $\pi_q^t$: the authenticator received in the query time $t$; $\pi_c$: the authenticator received in the checkpoint.

**Ensure:** $b \in \{0, 1\}$, if $b = 1$, the *Check* algorithm succeeds, otherwise, it fails.

let $\pi_q^t = \{\alpha_q^t, Sig_q^t\}$ and $\pi_c = \{\alpha_c, Sig_c\}$
**if** $\alpha_q^t \neq (Sig_q^t)_{spk}$ || $\alpha_c \neq (Sig_c)_{spk}$ **then**
    **return** $b = 0$
**end if**
$(rt_q^t, tp_q^t, \alpha) \leftarrow Dec_{K_3}(\alpha_q^t)$
**if** $tp_q^t$ is not before the previous update time point **then**
    let $\alpha_k = \alpha_c$
    **for** $\alpha_k \neq \emptyset$ **do**
        $(rt_k, tp_k, \alpha_{k-1}) \leftarrow Dec_{K_3}(\alpha_k)$
        **if** $tp_k < t$ **then**
        **end if**
        let $\alpha_k = \alpha_{k-1}$
    **end for**
    **if** $\alpha_k = \alpha_q^t || \alpha_k = \emptyset$ **then**
        **return** $b = 1$
    **else**
        **return** $b = 0$
    **end if**
**else**
    **return** $b = 0$
**end if**

## 4.5　安全性分析

## 4.6　实验结果

In Fig. 4.4, we evaluate the verification delays in data users. Note that an entire verification delay includes the delay of waiting for a checkpoint and the delay of executing the *Check* annd the *Generate* algorithms. Since the execution delay of the *Generate* algorithm is relatively stable, around 0.1 milliseconds, we do not plot it in Fig. 4.4. Here,

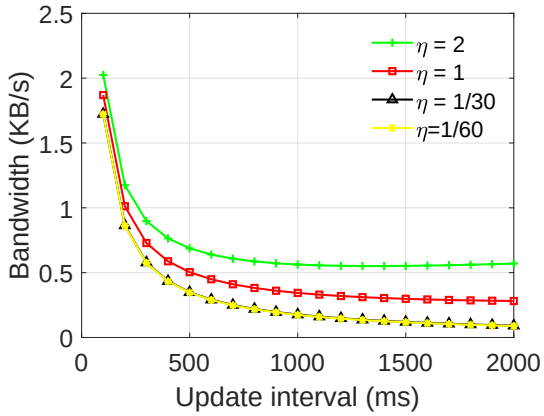


图 4.3　带宽开销

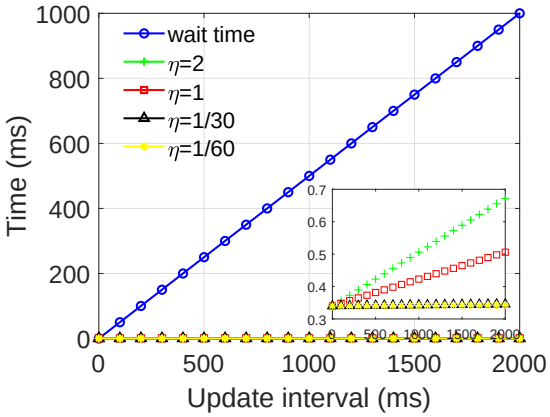


图 4.4　总验证时间开销

$\eta$ is the update frequency of the data owner. We assume that the time that a user initiates a query is uniformly distributed during an update interval, and then the user's waiting delays are also uniformly distributed. Therefore, the expected delay is half of the update interval and the verification delays are dominated by the waiting delays. The execution delay of the *Check* algorithm is negligible and is proportional to the update interval, which is mainly incurred by verifying the signature and decrypting authenticators. Kindly note that in above measurement, we do not take into account the network transmission and propagation delays, as they vary in different specific network contexts and do not reflect the essential extra cost directly introduced by our verification design. We do, however, report the communication overhead in terms of the message size, as shown in Fig. 4.3. In a later experiment, we will also show that we can set an update interval so as to make a trade-off between verification delays and communication overhead.

Fig. 4.3 shows the bandwidth costs for authenticator update. Here, the size of the first authenticator in each update interval is around 112 bytes, which includes 32 bytes of the root of MPT, 8 bytes of the timestamp, an 8 bytes AES-CBC extension and a 128 bytes RSA signature. Overall, the bandwidth of the authenticator includes two part: the overhead introduced by the fixed update time point and the overhead introduced by data update. We can observe that the bandwidth cost increases to about 2KB per second when the update interval decrease to zero, this is introduced by the fixed update time point which is inversely proportional to the bandwidth overhead. Moreover, the bandwidth gradually increases when the update interval becomes too long. This overhead is introduced by the length of the authenticator, because as the update interval grows, the length of the authenticator becomes larger. Overall, the cost should be acceptable to achieve GM-VSSE. According to the results, in order to make a decent tradeoff between verification delays

and bandwidth costs, we suggest choosing an update interval between 500 milliseconds and 1,500 milliseconds.

图 9 和图 10 评估了系统的总验证开销，包括计算开销和通信开销。我们主要考虑合法用户端的验证计算开销，和数据拥有者端的通信开销。图中的 表示数据拥有者的数据更新频率。首先，我们考虑合法用户端的验证开销。这部分开销包括用户等待检测点的时间以及执行 Check 算法和 Rebuild 算法的时间。由于 Rebuild 算法的开销可以忽略，因此图 9 中并未标注该部分开销。如图 9 所示，蓝色的曲线表示合法用户等待检测点的时间，这里我们假设合法用户的查询时刻在一个更新周期内呈均匀分布，那么用户等待时间的均值就是更新间隔的一半。其余的曲线表示 Check 算法的执行时间，可以看到，当更新频率在 2Hz 到 1/60Hz 不等时，Check 算法的执行时间也几乎可以忽略，因此合法用户端的验证开销主要取决于等待检测点所需的时间，即和更新间隔的设置有关。其次，我们考虑数据拥有者端的通信开销。我们主要考虑由鉴别符带来的开销。由于鉴别符的更新包含两种情况，固定更新导致的鉴别符更新和数据更新导致的鉴别符更新。图 10 中的曲线充分体现了这两种更新带来的开销。首先，当数据拥有者的更新间隔设置的很小时，通信开销较大，这是因为固定更新太频繁导致的鉴别符的通信开销增大；当数据拥有者的更新间隔设置的很大时，通信开销也较大，这是因为当更新频率一定时，更新间隔越大，在更新间隔内形成的鉴别符的长度会累积增大，最终导致了鉴别符的通信开销增大。综合以上两种情况，更新间隔可以设置为 1000ms 至 1500ms 之间。

# 第 5 章　总结与展望

## 5.1　论文工作总结

## 5.2　未来工作展望

# 参考文献


[1] Juels A, Kaliski Jr B S. Pors: Proofs of retrievability for large files[C]//Proc. of CCS. [S.l.: s.n.], 2007: 584–597.

[2] Ateniese G, Di Pietro R, Mancini L V, et al. Scalable and efficient provable data possession[C]// Proc. of Security and privacy in communication netowrks (SecureComm). [S.l.: s.n.], 2008.

[3] Kamara S, Papamanthou C, Roeder T. Cs2: A semantic cryptographic cloud storage system[R]. [S.l.]: Tech. Rep. MSR-TR-2011-58, Microsoft Technical Report (May 2011), http://research. microsoft. com/apps/pubs, 2011.

[4] Wang Q, Wang C, Ren K, et al. Enabling public auditability and data dynamics for storage security in cloud computing[J]. IEEE TPDS, 2011, 22(5): 847–859.

[5] Stefanov E, van Dijk M, Juels A, et al. Iris: A scalable cloud file system with efficient integrity checks[C]//Proc. of Annual Computer Security Applications Conference (ACSAC). [S.l.: s.n.], 2012.

[6] Kamara S, Papamanthou C. Parallel and dynamic searchable symmetric encryption[C]//Proc. of International Conference on Financial Cryptography and Data Security(FC). [S.l.: s.n.], 2013.

[7] Sun W, Liu X, Lou W, et al. Catch you if you lie to me: Efficient verifiable conjunctive keyword search over large dynamic encrypted cloud data[C]//Proc. of INFOCOM. [S.l.: s.n.], 2015.

[8] Ateniese G, Burns R, Curtmola R, et al. Provable data possession at untrusted stores[C]//Proc. of CCS. [S.l.: s.n.], 2007.

[9] Erway C C, Küpçü A, Papamanthou C, et al. Dynamic provable data possession[J]. ACM TISSEC, 2015, 17(4): 15.

[10] Zhu Y, Hu H, Ahn G J, et al. Cooperative provable data possession for integrity verification in multicloud storage[J]. IEEE TPDS, 2012, 23(12): 2231–2244.

[11] Bowers K D, Juels A, Oprea A. Proofs of retrievability: Theory and implementation[C]//Proc. of the workshop on Cloud computing security (SCC). [S.l.: s.n.], 2009.

[12] Song D X, Wagner D, Perrig A. Practical techniques for searches on encrypted data[C]//Proc. of S&P. [S.l.: s.n.], 2000.

[13] Curtmola R, Garay J, Kamara S, et al. Searchable symmetric encryption: improved definitions and efficient constructions[J]. Journal of Computer Security, 2011, 19(5): 895–934.

[14] Kamara S, Papamanthou C, Roeder T. Dynamic searchable symmetric encryption[C]//Proc. of CCS. [S.l.: s.n.], 2012: 965–976.

[15] Cash D, Jaeger J, Jarecki S, et al. Dynamic searchable encryption in very-large databases: Data structures and implementation.[C]//NDSS: volume 14. [S.l.: s.n.], 2014: 23–26.

[16] Wang Q, He M, Du M, et al. Searchable encryption over feature-rich data[J]. IEEE Transactions on Dependable and Secure Computing, 2016.

[17] Boneh D, Di Crescenzo G, Ostrovsky R, et al. Public key encryption with keyword search[C]// Proc. of EUROCRYPT. [S.l.: s.n.], 2004.

[18] Bost R, Fouque P A, Pointcheval D. Verifiable dynamic symmetric searchable encryption: Optimality and forward security.[J]. IACR Cryptology ePrint Archive, 2016, 2016: 62.

[19] Kurosawa K, Ohtaki Y. Uc-secure searchable symmetric encryption[C]//Proc. of International Conference on Financial Cryptography and Data Security (FC). [S.l.: s.n.], 2012.

[20] Chai Q, Gong G. Verifiable symmetric searchable encryption for semi-honest-but-curious cloud servers[C]//Proc. of International Conference on Communications (ICC). [S.l.: s.n.], 2012.

[21] Kurosawa K, Ohtaki Y. How to update documents verifiably in searchable symmetric encryption [C]//Proc. of International Conference on Cryptology And Network Security (CANS). [S.l.: s.n.], 2013.

[22] Stefanov E, Papamanthou C, Shi E. Practical dynamic searchable encryption with small leakage [C]//Proc. of NDSS. [S.l.: s.n.], 2014.

[23] Cheng R, Yan J, Guan C, et al. Verifiable searchable symmetric encryption from indistinguishability obfuscation[C]//Proc. of AsiaCCS. [S.l.: s.n.], 2015.

[24] Ogata W, Kurosawa K. Efficient no-dictionary verifiable sse[J]. IACR Cryptology ePrint Archive, 2016, 2016: 981.

[25] Wang C, Cao N, Li J, et al. Secure ranked keyword search over encrypted cloud data[C]//Proc. of ICDCS. [S.l.: s.n.], 2010.

[26] Merkle R C. A digital signature based on a conventional encryption function[C]//Proc. of EUROCRYPT. [S.l.: s.n.], 1987.

[27] Papamanthou C, Tamassia R, Triandopoulos N. Authenticated hash tables[C]//Proc. of CCS. [S.l.: s.n.], 2008.

[28] Pugh W. Skip lists: a probabilistic alternative to balanced trees[J]. Communications of the ACM, 1990, 33(6): 668–676.

[29] Goodrich M T, Tamassia R, Schwerin A. Implementation of an authenticated dictionary with skip lists and commutative hashing[C]//Proc. of DARPA Information Survivability Conference & Exposition (DISCEX). [S.l.: s.n.], 2001.

[30] Zheng Q, Xu S, Ateniese G. Vabks: verifiable attribute-based keyword search over outsourced encrypted data[C]//Proc. of INFOCOM. [S.l.: s.n.], 2014.

[31] Liu P, Wang J, Ma H, et al. Efficient verifiable public key encryption with keyword search based on kp-abe[C]//Proc. of Broadband and Wireless Computing, Communication and Applications (BWCCA). [S.l.: s.n.], 2014.

[32] Yang Y, Bao F, Ding X, et al. Multiuser private queries over encrypted databases[J]. International Journal of Applied Cryptography, 2009, 1(4): 309–319.

[33] Jarecki S, Jutla C, Krawczyk H, et al. Outsourced symmetric private information retrieval[C]// Proc. of CCS. [S.l.: s.n.], 2013.

[34] Sun S F, Liu J K, Sakzad A, et al. An efficient non-interactive multi-client searchable encryption with support for boolean queries[C]//Proc. of ESORICS. [S.l.: s.n.], 2016.

[35] Bellare M, Goldreich O, Goldwasser S. Incremental cryptography: The case of hashing and signing[C]//Proc. of CRYPTO. [S.l.: s.n.], 1994.

[36] Wood G. Ethereum: A secure decentralised generalised transaction ledger[J]. Ethereum Project Yellow Paper, 2014, 151: 1–32.

[37] Merkle patricia tree[M]. [S.l.: s.n.].

[38] Rlp code[M]. [S.l.: s.n.].

[39] Enron_email dataset[M]. [S.l.: s.n.].

# 致　谢

衷心感谢导师李琦教授，香港城市大学王聪教授，武汉大学王骞教授对本人的精心指导。他们的言传身教将使我终生受益。

感谢宋奇阳同学，以及实验室全体老师和同学们的热情帮助和支持，帮我节省了不少时间。本课题承蒙国家自然科学基金资助，特此致谢。

# 声　明

本人郑重声明：所呈交的学位论文，是本人在导师指导下，独立进行研究工作所取得的成果。尽我所知，除文中已经注明引用的内容外，本学位论文的研究成果不包含任何他人享有著作权的内容。对本论文所涉及的研究工作做出贡献的其他个人和集体，均已在文中以明确方式标明。

签　名：_____ 日　期：_____

# 个人简历、在学期间发表的学术论文与研究成果

## 个人简历

1993 年 02 月 02 日出生于浙江省海宁市。

2011 年 09 月考入北京邮电大学计算机学院计算机科学与技术专业。

2015 年 07 月本科毕业并获得工学学士学位。

2015 年 09 月免试进入清华大学计算机科学与技术系攻读工程硕士学位至今。

## 发表的学术论文

[1] Jie Zhu, Qi Li, Cong Wang, Xingliang Yuan, Qian Wang, Kui. Enabling Generic,Verifiable, and Secure Data Search in Cloud Services. (已被 IEEE Transactions on Parallel and Distributed Systems(TPDS) 录用)

## 研究成果

[1] 李琦, 朱洁, 王骞. 一种可验证的加密搜索方法: 中国, 201711277295.7. (中国专利申请号)

[2] Qi Li, Jie Zhu, Yanyu Chen, Lichun Li. System and Method for detecting routing loops in a Software Defined Network (SDN): SG, 10201703959R. (新加坡专利申请号)