國立臺灣師範大學
資訊工程研究所碩士論文

指導教授： 黃冠寰 博士

利用集合雜湊值達到有效率的雲端儲存系統稽核與證明違約協定

Efficient Auditing and Proof-of-Violation for Cloud Storage Systems in Aggregate hash

研究生： 吳承翰 撰

中華民國 一零七 年 七 月

摘 要

利用集合雜湊值達到有效率的雲端儲存系統稽核與證明違約協定

吳承翰

雲端儲存系統是一種執行在雲端運算平台上的儲存系統,使用者可以將檔案放在雲端儲存系統以節省使用者自己電腦的儲存空間,雲端儲存系統也不需要使用者維護，而是由雲端服務提供者負責安裝、維護這些儲存系統。然而，雲端儲存系統可能因為系統當機、系統毀損、錯誤操作，甚至是遭到駭客攻擊而導致資料遺失或資料被更動，進而使雲端服務提供者回傳給使用者的資料版本不一致。因此我們希望使用者在存取雲端儲存系統時都可以透過一些證據來實行即時性的稽核。此外，我們也希望雲端服務提供者與使用者在進行稽核時，能夠以較少的運算動作來完成，達到減少能耗的目的。在本篇論文中，我們提出了一個即時稽核架構,此架構可以讓使用者執行 query 動作之後能夠即時性的發現雲端服務提供者是否有不當疏忽造成儲存系統毀損或是被竄改,並且有效的降低稽核時運算上的能耗。當不幸發生儲存系統毀損或是被竄改的情況，可以再透過證明違約機制來釐清使用者與雲端服務提供者之間的違約責任歸屬。我們將使用 Aggregate hash 來實作此架構。

關鍵字：雲端儲存、雲端安全、不可否認性、即時稽核、證明違約

誌謝

首先要先感謝我的指導教授黃冠寰老師，老師很有耐心的教導我做研究的方法以及許多的資訊科學領域知識，很感謝老師引導我如何去發現問題、理解問題並且解決問題，這兩年讓我收穫很多。老師讓我能夠從一個原本完全不懂得活用知識的學生開始慢慢學會如何去思考，並且讓我能夠在報告時能有更系統化的架構，在這兩年當中我學到的不只是資訊科學的相關知識，更學到了許多如何做研究的態度以及方法，透過老師兩年的訓練，讓我在解決問題的能力上有不凡的改進。也謝謝我的家人默默的支持我念完這兩年的研究所，讓我能全心全力去完成我的學業。另外還要感謝兩年中遇到的學長以及朋友們，感謝鄭安傑學長在我不懂的時候可以耐心的教導我，以及給我很多建議和方向。感謝這兩年來跟我一起奮鬥的同學、學弟們，人恩、信德、聖翰還有培均，謝謝你們在這兩年中也教了我許多，還有一起陪伴我在實驗室奮戰的日子，謝謝大家讓我在這兩年的研究所生活覺得非常快樂。

吳承翰　誌於

國立臺灣師範大學資訊工程所

民國　一零七 年 七 月

目錄

附表目錄

附圖目錄

第一章 簡介

第一節 雲端儲存系統

雲端儲存系統（Cloud Storage Systems）[1]是一種運行在雲端運算平台上的遠端儲存系統。雲端儲存系統是由雲端服務提供者負責安裝、維護儲存系統，使用者不需要自己維護、管理儲存系統，僅須透過網路將欲儲存的檔案上傳至雲端儲存系統儲存即可。這樣的儲存方式可以減輕使用者電腦的負擔。目前著名的雲端儲存系統有 Google Drive[2]、Dropbox[3]、OneDrive[4]、iCloud[5]、SugarSync[6]等等。

第二節 雲端儲存系統遇到的問題

以目前比較著名的雲端儲存系統而言，使用者透過雲端服務商提供的服務無法直接存取雲端儲存系統實體以外的檔案，例如儲存系統的設定檔與日誌檔。就算使用者能夠取得設定檔或日誌檔，這些檔案也可能已經被雲端服務提供商造假。此外，儲存系統遭到駭客入侵，或者儲存系統毀損、回溯等等狀況都是現行的雲端儲存系統非常容易發生的情況，在這些情況下，使用者日後可能就會取得錯誤的檔案而無法驗證。所以為了讓使用者可以確保儲存系統的資料正確，使用者與雲端服務提供者之間必須要制定一些驗證機制，我們一般稱這些驗證動作為稽核（Auditing）。

然而，只有稽核是不夠的，使用者與雲端服務提供商之間還需要一些密碼學證據來釐清當問題發生時的責任歸屬，並訂定一個合約來保障雙方的權益，這個合約機制我們稱為行為違反驗證協定（Proof of Violation）[7]，在下一節中，我們會介紹行為違反驗證協定。

第三節　行為違反驗證協定

有鑑於上一節所述驗證方法的缺點，我們在先前發表的論文中提出了證明違約協定（Proof of Violation，以下簡稱 POV）來達到當發現檔案錯誤時的責任歸屬。POV 是指用戶將檔案上傳至雲端儲存之前，可以和雲端服務提供商建立一個合約。如果雲端服務提供商管理的資料與當初使用者上傳的版本不一致，則需根據合約內容賠償使用者相對應的金額。當用戶在使用雲端服務提供商的服務時，對每一個動作結果皆會產生一個密碼學證據（cryptographic proof），當用戶在使用服務上產生爭議時，可以利用密碼學證據來證明服務提供商是否有過失，反之，雲端服務提供商也可以透過密碼學證據證明自己是否有確實保存正確的資料。

在[7][8]論文中提到，POV 可以分為 Epoch-Based POV 以及 Real-Time POV 兩種。Epoch-Based POV 是指使用者與雲端服務提供商會協議一個固定的時期（Epoch），在這個時期內，雲端服務提供商會記錄使用者所有使用的服務動作及執行結果。當時期結束時，雲端服務提供商與使用者對這些服務動作與執行結果進行驗證，並將密碼學證據更新到最新版本。雖然 Epoch-Based POV 可以驗證使用者使用的服務動作與執行結果是否正確，但是使用者沒辦法在每次使用服務之

後立刻檢查雲端服務提供商是否有違約，只能等到每次時期結束時，才能夠進行驗證。Real-Time POV 是指使用者在使用服務之後，可以立刻在非常短的時間內透過使用者保存的密碼學證據檢查這次服務動作後的結果是否正確。

第四節　目標

為了避免使用者因為本章第二節所講述的雲端儲存問題而存取到錯誤的檔案，所以我們的目標是將 POV 機制加入整個雲端儲存系統架構中，達到[9]中所提出的四個雲端安全性質，即保密性(Confidentiality)、資料完整性(Data Integrity)、讀取資料的新鮮性(Read Freshness)以及寫入檔案的循序性(Write Serializability)，避免使用者拿到錯誤的資料，同時我們會使用本實驗室的研究成果 Aggregate hash 來達成 Real-time auditing。

第二章 過往作法

第一節 Merkle Hash Tree

目前許多對於雲端儲存系統稽核的相關研究都是建立在 Authenticated data structure 的基礎上，目前最常被使用的是 Merkle Hash Tree。接下來，我們先介紹[8]中提到的 Merkle Hash Tree 是如何達到驗證檔案完整性與正確性。

首先，使用者將欲儲存的檔案與資料夾上傳至雲端儲存系統後，雲端服務提供商會利用 SHA-256**錯誤! 找不到參照來源。**計算出這些檔案與資料夾的 Hash[11]值，並將這些 Hash 值建立一個如圖 1 所示的 Merkle Hash Tree 結構。在此結構中，每一層的 Hash 值都是從樹的 Leaf-Node 開始向上計算而得，例如 h（d3）=h（h（f2），h（d6），h（f3）），最後經過一層一層的計算後，會得到可以代表整棵樹狀態的 Root Hash。因為雜湊函數的特性是只要某個節點有被修改過，那麼整棵 Merkle Hash Tree 的 Root Hash 就會不同，因此使用者只需要保存 Root Hash 加上雙方的數位簽章（Digital Signature）[12]就可以驗證雲端儲存系統的完整性與正確性，而整個樹狀結構則由雲端服務提供商保存。日後若使用者想要更新或修改資料，雲端服務提供商會重新計算出 Root Hash 並回傳給使用者儲存，等使用者需要從雲端下載檔案時，只需要透過雲端服務提供商的 Merkle Hash Tree 來計算 Root Hash 並與自己保存的 Root Hash 做比對，如果相同，則表示檔案未再被更動，完成稽核。

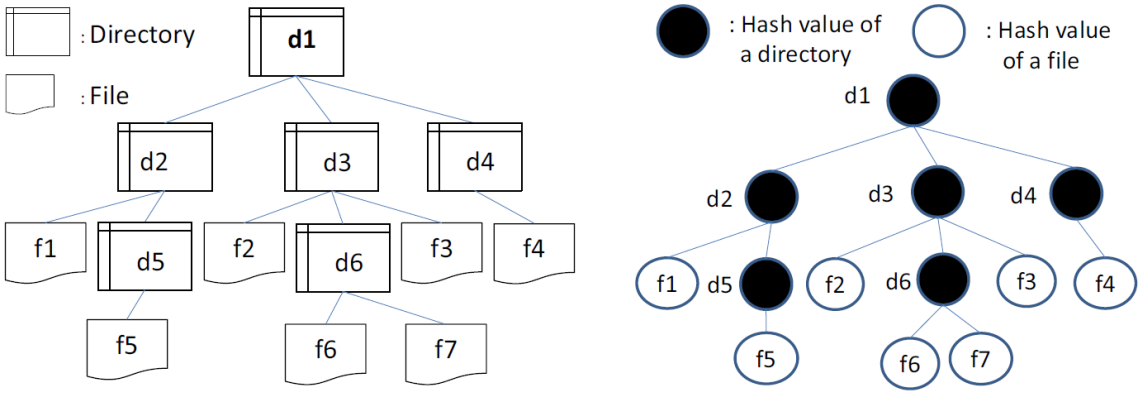


圖1 Merkle Hash Tree 結構

雖然這樣的做法確實可以驗證檔案的完整性，但當儲存的資料夾與檔案數量非常多的時候，Merkle Hash Tree 計算出 Root Hash 就會花費非常多的時間。所以[8]對於原本的 Merkle Hash Tree 稍加修改，形成 Partial Merkle Tree（pMT），讓使用者不用計算整棵 Merkle Hash Tree 也可以進行驗證。接下來我們介紹一下 Partial Merkle Tree 是如何運作的。

假設雲端服務提供商針對某位使用者所保存的 Merkle Hash Tree 如圖 2 所示:

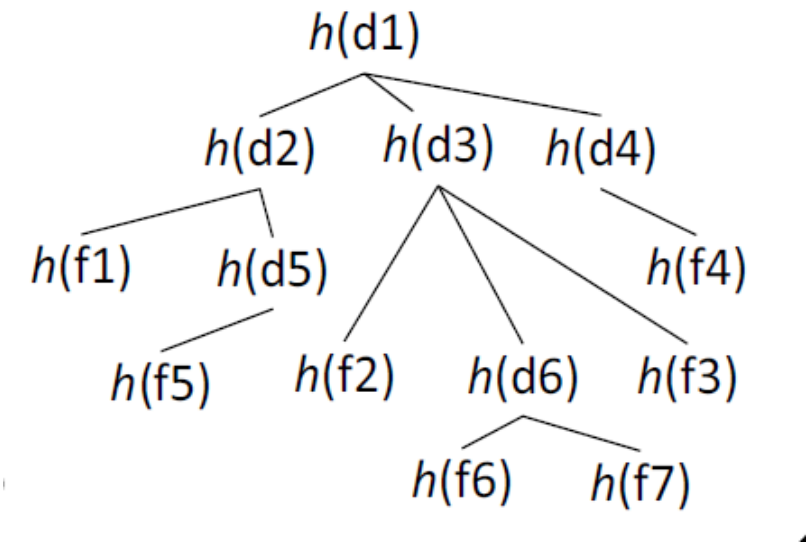


圖2 雲端服務提供商儲存的 Merkle hash Tree 範例

若這名使用者在某個時期只需要存取 f1 與 f5，表示 d3 與 d4 底下所有檔案應該是不會被更動的，也就是說 h（d3）與 h（d4）應該也是不會變的，所以當使用者想要進行稽核時，只需要計算如圖 3 所示的 Partial Merkle Tree 即可，d3 與 d4 底下未被更動的部分就不用再計算一次，如此一來，就可以加快使用者計算出 Root Hash 的時間。

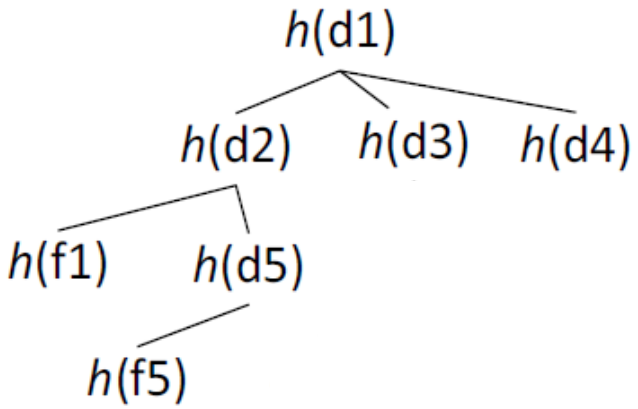


圖3 包含 f1 與 f5 的 Partial Merkle Tree

但是這樣的做法還是有個問題，如果使用者欲存取的資料所在的那一層資料數量非常多，就算只計算 Partial Merkle Tree，還是會花費使用者非常多的時間。

第二節　Full Binary Hash Tree

在[13]中提到了另一個對雲端儲存系統稽核的方法，這個方法是將 Merkle Hash Tree 稍作改良而成。當使用者上傳一系列的檔案到雲端儲存系統後，雲端服務提供商會產生每個檔案的 Hash 值，並將這些 Hash 值透過一個 Index function Γ 定位到 Hash Tree 中對應的 Leaf-Node 位置，最後一層一層向上運算，建立一個樹狀結構，此結構稱為 Full Binary Hash Tree （以下簡稱 FBHTree），

如下圖所示：

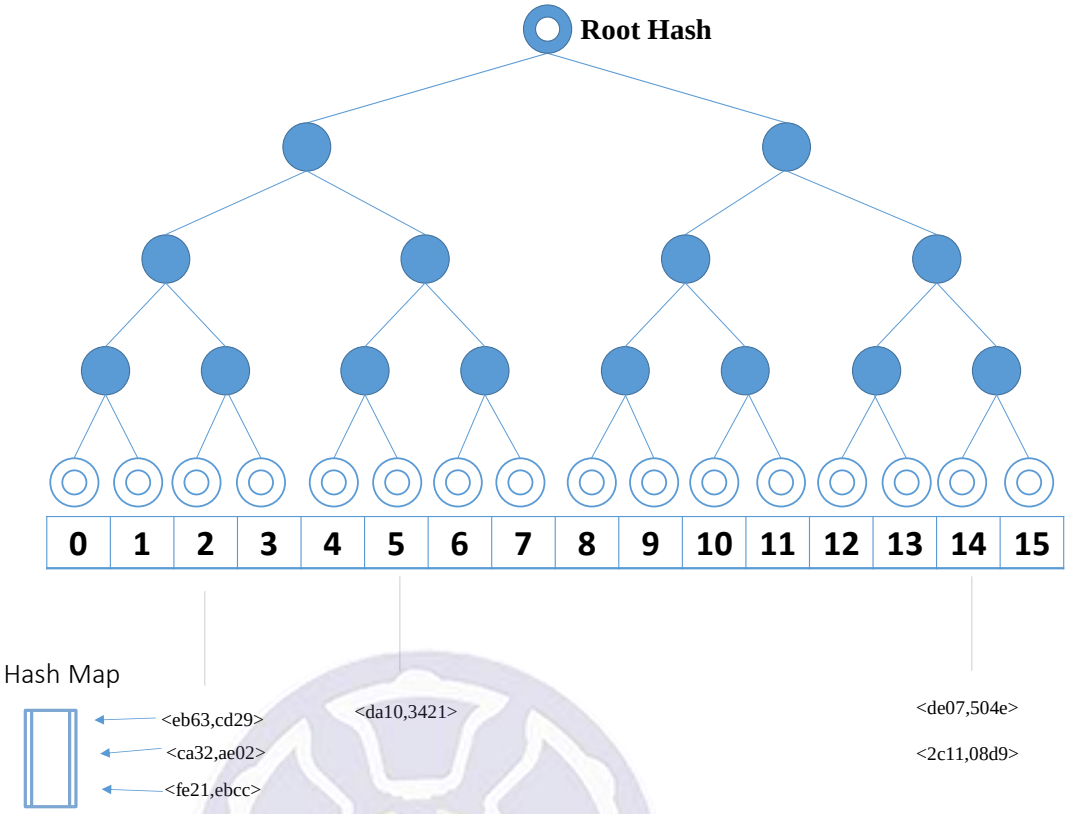


圖4 雲端服務提供商所儲存的 FBHTree 結構

當 FBHTree 建立完成後，雲端服務提供商會計算出 Root Hash，使用者只需要保存 Root Hash 就可以進行稽核動作。

雖然在 FBHTree 的稽核架構中，使用者只須要取得要稽核的檔案所在的 slice 並搭配自己保存的 Root Hash 即可進行稽核，確實解決傳統 Merkle Hash Tree 運算量可能過大的問題。但若是使用者想要將某個檔案從舊的資料夾搬移到新的資料夾，那麼雲端服務提供者就必須對 FBHTree 所對應的兩條 slice 進行更新才能夠計算出新的 Root Hash 以供使用者保存。下面圖 5 與圖 6 為使用者搬移一個檔案後，雲端服務提供商所須更新的 slice：

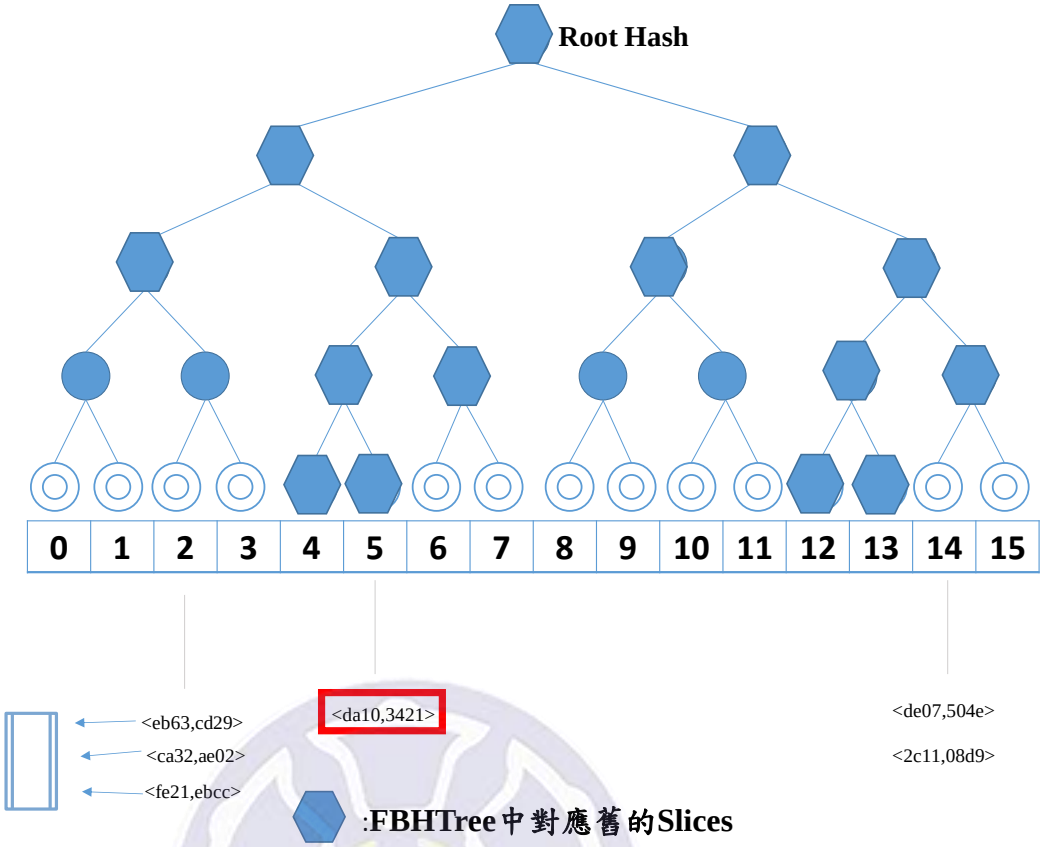


圖5 雲端服務提供商收到使用者搬移檔案的請求後，會先將 FBHTree 中對應舊的兩條 Slices 回傳給使用者作稽核

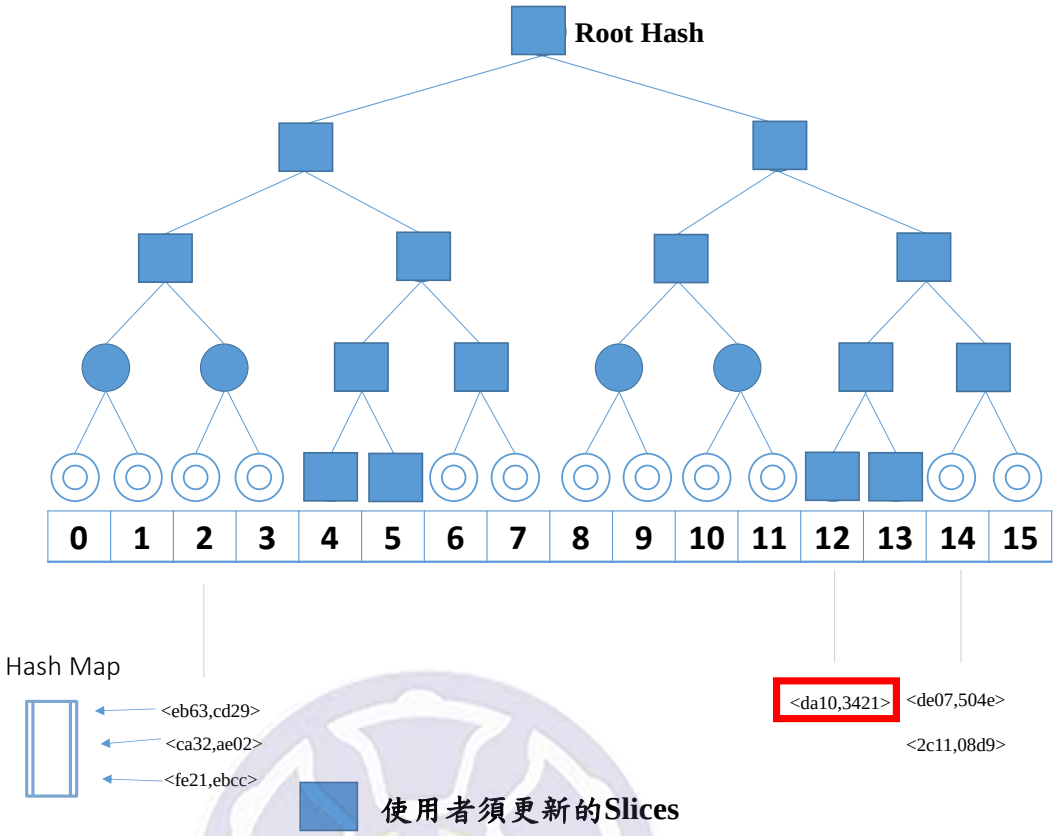


圖6 使用者搬移一個檔案後，雲端服務提供商計算出新的 Root Hash，使用者將舊的 Slices 更新後與雲端服務提供商的 Root Hash 比對以完成稽核

但對於一般電腦使用者來說，時常更動檔案的路徑是很常見的事情，如果使用者搬移了 N 個檔案，則需要 2N 次的 Slices 傳輸時間、4N 次 Slices 運算更新 Root Hash，對於使用者與雲端服務提供商而言，無論是在運算量與更新 Root Hash 的時間上都會造成非常大的負擔。

在接下來的章節中，我們會介紹本實驗室新的稽核架構，不僅能夠達到即時稽核，也能夠有效減少雲端提供商的運算負擔。

第三章 稽核架構

第一節 群與模反元素

在數學中，群（Group）是由一個集合以及一個二元運算所組成。一個群必須符合封閉性、結合律、單位元素、以及集合中所有元素都具有反元素等條件[14]。給定兩個整數 a、b，若這兩個整數除以正整數 m 所得到的餘數相等，則稱 a，b 對於模 m 同餘，記作 $a \equiv b \pmod{m}$。

即若 P 為質數，h_i （$0 \le i < n$）為 n 個整數， $h_i < P$ 且 h_i 與 P 互質，且 X 和此 n 個整數的乘積同餘於 P，如下：

$$X \equiv h_1 \times h_2 \times h_3 \times \ldots \times h_{k-1} \times h_k \times h_{k+1} \times \ldots \times h_{n-1} \pmod{P}$$

在此群中，以下的特性一定成立：

$$X \times h_n \equiv h_1 \times h_2 \times h_3 \times \ldots \times h_{k-1} \times h_k \times h_{k+1} \times \ldots \times h_{n-1} \times h_n \pmod{P}$$

$$X \times h_k^{-1} \equiv h_1 \times h_2 \times h_3 \times \ldots \times h_{k-1} \times h_{k+1} \times \ldots \times h_{n-2} \times h_{n-1} \pmod{P}$$

以下舉例：若 P 為 37，且 $X \equiv 3 \times 16 \times 21 \times 29 \times 38 \pmod{37}$，以下可以計算出 X 可能的值：（1） $3 \times 16 \times 21 \times 29 \times 38 = 1110816$；（2） $1110816 \text{ MOD } 37 =^{1} 2$。所以 X 可以為 2。

[1]此處的 MOD 為兩正整數的乘積求取除以 P 正整數的餘數計算，非同餘的意思。

我們看範例，$X\times13 \equiv 3\times16\times21\times29\times38\times13 \pmod{37}$，檢驗看看：（1）$3\times16\times21\times29\times38\times13 = 14440608$、$14440608 \text{ MOD } 37 = 26$；（2）$X\times13 \text{ MOD } 37 = 2\times13 \text{ MOD } 37 = 26$。

再一個範例，$X\times29^{-1} \equiv^{2} 3\times16\times21\times29\times38\times29^{-1} \pmod{37}$，如下檢驗：（1）$3\times16\times21\times29\times38\times29^{-1} \text{ MOD } 37 = 3\times16\times21\times38 \text{ MOD } 37 = 9$；（2）$X\times29^{-1} \text{ MOD } 37 = 2\times29^{-1} \text{ MOD } 37 = 2\times23 \text{ MOD } 37 = 9$。

第二節　Aggregate hash 與雲端物件稽核

藉由上一節所述的特性，我們可以加以運用實作雲端 Auditing 及 POV 機制。若有 n 個物件存在雲端系統，分別為 O_0、O_1、…、O_k、…、O_{n-1}，我們假定這些物件內都帶有此物件 hash value 及使用者對此物件 hash value 的電子簽章，這只有使用者能夠產出。其中，$h_0 = \text{hash}(O_0)$、$h_1 = \text{hash}(O_1)$、$h_2 = \text{hash}(O_2)$、…、$h_{n-1} = \text{hash}(O_{n-1})$。令 $X \equiv h_0\times h_1\times h_2\times\ldots\times h_{n-1} \pmod{P}$，我們稱 X 所有可能值的最小整數，為這 n 個物件的 Aggregate hash。要計算出 Aggregate hash 很容易，及將這些 hash values 相乘，所得乘積和 P 相除求其餘數，即 Aggregate hash $AH = h_0\times h_1\times h_2\times\ldots\times h_{n-1} \text{ MOD } P$。則每次新增一物件 O_i，$h_i = \text{hash}(O_i)$，新的 Aggregate

[2]因為 29 與 37 互質，根據擴展歐基里得函式可以得到 $29x+37y=1$，$29x+37y\equiv29x\equiv1 \pmod{37}$，根據模反元素的定義，x 即為 29 對於 37 的模反元素，我們可以找到 $x=23$，$y=(-18)$ 滿足 $29x+37y=1$，其中 23 即為 29 對於 37 的模反元素。但事實上，$23+37k$（k 為整數）都是 29 對於 37 的模反元素，故通常會取最小正整數解，即 23。

hash AH′表示 n+1 個物件，$AH' = AH \times h_i \text{ MOD } P$。若由這 n 個物件移除一個物件，新的 Aggregate hash AH″表示 n-1 個物件，$AH'' = AH \times h_j^{-1} \text{ MOD } P$。使用者和雲端系統對於儲存於系統物件的互相以電子簽章同意最後的 Aggregate hash，即建立了這些儲存物件的電子證據。

使用者在新增、編刪物件後可以自行運算得出新的 Aggregate hash， 雲端系統也可以得到相同的 Aggregate hash。但是在讀取（或下載）物件時，雲端系統必須將所有物件的 hash values 及使用者的簽章傳送給使用者，使用者先核對這些簽章是否正確，然後將所有的 hash values 相乘，所得乘積和 P 相除求其餘數的結果和現在的 Aggregate hash 比較是否相同，如果相同表示所得所有物件的 hash values 都是正確的。雲端系統無法欺瞞使用者，因為他無法假造使用者的電子簽章，所以也不可能產生一些 hash values 讓其 Aggregate hash 和使用者手上的 Aggregate hash 相同。

但是很明顯的這樣的方法沒有效率，因為每次使用者要讀取（或下載）單一物件時，雲端系統都必須將所有物件的 hash values 及使用者的簽章傳送給使用者，如果物件數目十分大時，效率回十分差，比如雲端儲存系統一個帳戶可以有數萬個檔案，雲端資料庫系統一個 table 可以有百萬個 rows。

解決的方法是建立一個協定將物件的 hash values 分群，同一群的物件將他們的 Aggregate hash 存在一個固定的位置。見圖 7 **錯誤! 找不到參照來源。**，我們

將物件分為 k 組，每一組都有自己的 Aggregate hash，分別為 AH_0、AH_1、…、AH_{k-1}。分組的方法可以將物件的 hash values 除以 k 取餘數。舉例而言，n 個物件中，O_0、O_1、…、O_k、…、O_{n-1}，其中，h_0 = hash（O_0）、h_1 = hash（O_1） 、h_2 = hash（O_2） 、…、h_{n-1} = hash（O_{n-1}）。若 h_i MOD k = 2，則 h_i 分在 AH_2 那一組。以此例而言，若分在第 j 組的有 q 個物件，O_{j1}、O_{j2}、…、O_{jq}。則協定規定，$AH_j = h_{j1} \times h_{j2} \times \ldots \times h_{jq}$。若 AH 為此 n 個物件的 Aggregate hash，則可以證明 AH = $AH_0 \times AH_1 \times \ldots \times AH_{k-1}$ MOD k。因為每個物件的 hash value 都分配到某一組，所以 $AH_0 \times AH_1 \times \ldots \times AH_{k-1}$ MOD k = $h_1 \times h_2 \times h_3 \times \ldots \times h_{k-1} \times h_k \times h_{k+1} \times \ldots \times h_{n-1}$ MOD k = AH，即得證。

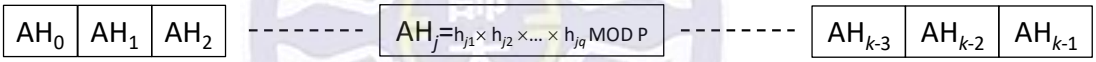


圖7 將物件分組並分別儲存每一組的 AH

在使用者的裝置中，只要儲存 k 個 hash values，AH_0、AH_1、…、AH_{k-1}，及 Aggregate hash。雲端系統除了要儲存此 k 個 hash values 及 Aggregate hash，還要記錄每組有哪些物件的 hash values。舉例，若使用者要自雲端讀取物件 O_{j2}，他先算出此物件的 hash values 存在第 j 組，根據協定雲端系統要傳遞 q 個物件的 hash values，h_{j1}、h_{j2}、…、h_{jq} 及 O_{j2}，給使用者。使用者算出 $h_{j1} \times h_{j2} \times \ldots \times h_{jq}$，並比對是否等於自己儲存的 AH_j，若相等則進一步檢查 O_{j2} 內的 tag 及簽章及其 hash value 就可以確認是否雲端系統回傳正確的物件。如果通不過稽核，持有的 AH 可以成

為 POV 的電子證據。

第三節　雲端檔案儲存系統的即時稽核架構

在第二節中，我們說明了如何使用 Hash 來完成雲端儲存系統中物件的稽核，接下來在本節中，我們會介紹如何將 Aggregate hash 應用在雲端的檔案儲存系統中。

在我們的即時稽核架構中，參與角色分為使用者與雲端服務提供者。首先，我們先介紹使用者如何建立與保有稽核證據。使用者在上傳檔案到雲端儲存之前，會在本機上先建立一個 Array 並且在 Array 的每個位置底下建立一個 hashmap，接著使用者計算出檔案路徑與檔案本身的 hash 值，然後再利用檔案路徑的 hash 值經過一個 Index function Γ 的運算來得到該檔案的檔案路徑與檔案本身的 hash 值應該放在 Array 中哪個位置的 hashmap，此 Index function Γ如下：

$$\Gamma(\text{File\_Path}) = \text{SHA-256}(\text{File\_Path}) \bmod 2^{N-1}。$$

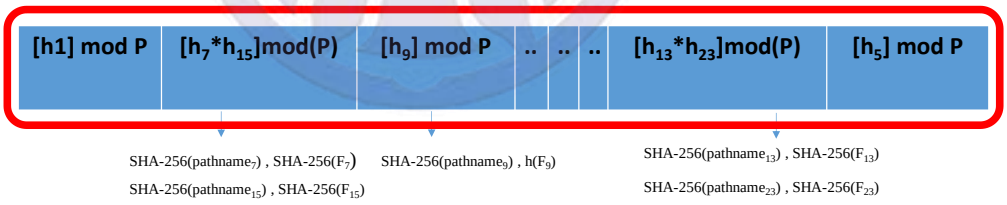
其中，N 為 Array 的長度，File_Path 為檔案路徑，例如：C:\User\123.txt。

當檔案路徑與檔案本身的 hash 值都放入 hashmap 之後，使用者會將檔案路徑的 hash 值與檔案本身的 hash 值進行 Π 運算，最後將運算結果 module 一個 256bits 的大質數 P 後儲存在 Array 中。此 Π 運算如下：

Π（SHA-256（File_Path）,SHA-256（File））=SHA-256[SHA-256（File_Path）

|| Sha-256（File）] mod P。

如果使用者的檔案數量很大，可能會發生 Array 某一個位置的 hashmap 會存放多對檔案路徑與檔案的 hash 值。若發生這樣的情況，使用者須將每一對檔案路徑與檔案的 hash 值進行 Π 運算後，將這些運算結果相乘起來得到該位置的 Aggregate hash，再將 Array 中每個位置內的 Aggregate hash 全部相乘起來即可得到代表整個 Array 狀態的證據 AH。最後，使用者只須要保存 Array 與 AH 即可在未來進行檔案稽核。使用者的稽核架構如下圖所示。

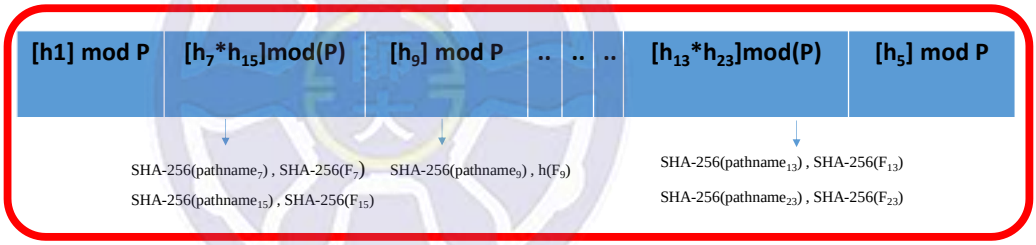


●Client calculates the total aggregate hash→AH=$(h_1 \times h_2 \times \ldots \times h_t) \bmod (P)$

：**使用者需要保存的稽核證據**

圖8 使用者欲進行檔案稽核前所須建立的稽核架構與證據

接下來，我們要介紹雲端儲存提供者所需要建立的稽核架構。雲端儲存提供者會接收到使用者的檔案之後，會仿照使用者一模一樣的步驟來建立稽核架構，不同的地方在於雲端端儲存提供者不僅要保存 Array，包括 Array 底下 hashmap 所儲存的內容也需要保存起來，雲端儲存提供者的稽核架構如下圖所示：



●Server calculates the total aggregate hash → AH=$(h_1 \times h_2 \times \ldots \times h_t) \bmod (P)$

：**雲端服務提供商需要保存的稽核證據**

圖9 雲端服務提供商須建立的稽核架構與證據

第四節　稽核檔案

在本節中，我們會延續上面幾節所介紹的稽核架構來說明使用者如何進行稽核。當使用者想要從雲端儲存系統取回檔案時，使用者必須先對檔案進行稽核以確保下載回來的檔案是沒有被竄改過的。假設使用者檔案的 hash 值存在 Array 中

的第 K 個位置，則稽核的步驟如下：

Step1: 使用者請雲端服務提供者回傳第 K 個位置內每一對檔案路徑與檔案 hash 值。

Step2: 使用者會將每一個檔案路徑的 hash 值 module 大質數 P 來確認回傳回來的資料是否都是放在 Array 中第 K 個位置。

Step3: 使用者將每一對檔案路徑與檔案 hash 值進行 Π 運算後得到的結果全部相乘起來 module 大質數 P，並與自己保存的第 K 個位置的 Aggregate hash 做比較，若一致，則稽核完成。

第五節　增刪檔案

雲端儲存系統的真實使用情況是使用者會常常需要新增檔案或是刪除舊的檔案，所以接下來我們會介紹使用者在新增或刪除檔案之後迅速得到稽核證據的步驟。

新增檔案

Step1: 使用者先將新的檔案路徑之 hash 值進行 Index function Γ的運算得到 Array 中對應的位置 I。

Steop2: 使用者將新的檔案路徑與檔案 hash 值進行 Π 運算後得到的結果與位置 I 中儲存的 Aggregate hash 相乘得到 I 位置新的 Aggregate hash。

Step3: 使用者將新的檔案路徑與檔案 hash 值進行 Π 運算後得到的結果與原本的 AH 相乘形成新的 $AH^{'}$。

Step4: 使用者保存新的 Array 與 AH'作為新的稽核證據。

Step5: 使用者將新的檔案上傳給雲端服務提供者保存，雲端服務提供者仿照使用者的方法，得到新的 Array、Array 底下的 Hashmap 與 AH。

Step6: 使用者與雲端服務提供者核對新的 AH 是否一致，雙方簽章後達成共識。

刪除檔案

Step1: 使用者將檔案之檔案路徑 hash 值進行 Index function Γ的運算得到 Array 中對應的位置 T。

Step2: 使用者先將檔案之檔案路徑與檔案 hash 值進行 Π 運算，然後使用者根據 Π 運算的結果找出對應的模反元素。

Step3: 使用者將位置 T 的 Aggregate hash 乘上模反元素後得到位置 T 新的 Aggregate hash。

Step4: 使用者將原本的 AH 乘上模反元素得到新的 AH"。

Step5: 雲端服務提供者接收使用者刪除檔案的請求後刪除檔案，並且仿照使用者的方法，得到新的 Array、Array 底下的 Hashmap 與 AH。

Step6: 使用者與雲端服務提供者核對新的 AH 是否一致，雙方簽章後達成共識。

第六節　　搬移檔案

在真實的個人電腦環境中，使用者常常會把某個檔案從原本的資料夾移動到新的資料夾。在這樣的情況下，檔案路徑的 hash 值會改變，導致進行 Index function Γ運算後，檔案路徑與檔案本身的 hash 值會存在 Array 中的其他位置。因此，若使用者想要更新 Array 與 AH，必須經過一次刪除與一次新增的動作。接下來，我們會介紹搬移檔案之後，使用者如何迅速得到稽核證據。

搬移檔案

Step1: 使用者將原檔案路徑的 hash 值進行 Index function Γ的運算得到 Array 中對應的位置 T。

Step2: 使用者將原檔案路徑與檔案 hash 值進行 Π 運算，然後使用者根據 Π 運算的結果找出對應的模反元素。

Step3: 使用者將位置 T 的 Aggregate hash 乘上模反元素得到新的 Aggregate hash。

Step4: 使用者將原本的 AH 乘上模反元素得到新的 AH_d。

Step5: 使用者將新檔案路徑的 hash 值進行 Index function Γ的運算得到 Array 中對應的位置 I。

Step6: 使用者將新檔案路徑與檔案 hash 值進行 Π 運算後得到的結果與位置 I 中儲存的 Aggregate hash 相乘得到新的 Aggregate hash。

Step7: 使用者將新的檔案路徑與檔案 hash 值進行 Π 運算後得到的結果與

AH_d 相乘形成新的 AH_i。

Step8: 使用者保存新的 Array 與 AH_i 作為新的稽核證據。

Step9: 雲端服務提供者收到使用者搬移檔案的請求後，仿照上述使用者的步驟更新 Array、Array 底下的 Hashmap 與 AH。

Step10: 雲端服務提供者更新完成後與使用者合對最新的 AH 是否一致，雙方簽章後達成共識。

第七節　另一種 Aggregate hash 稽核架構

在本章第三節中，我們介紹了利用 Aggregate hash 設計的稽核架構，從介紹中可以發現，將檔案分群的 Index Function 會讓位於同一個資料夾內的檔案分散到 Array 中不同的位置，許多人會認為如果讓同一個資料夾內的檔案能夠放在 Array 中同一個位置會如何，所以我們稍加改變了稽核架構，讓同一個資料夾內的檔案可以放到 Array 中同一個位置。雖然這個改變看似可以提升稽核效率，但實際上如果使用者的資料夾內有大量檔案，那麼在稽核時，反而會造成更大的負擔，在下一章我們會有兩種 Aggregate hash 架構以及 FBHTree 的實驗結果。

第四章 相關實驗數據

本章實驗的對象為 Index Merkle Tree 與 Aggregate hash。我們會再將 Aggregate hash 分成兩個版本，第三章第三節所介紹的版本，我們稱為 Aggregate hash（一）。第三章第七節所介紹的版本，我們稱為 Aggregate hash（二）。

在一系列的實驗中，我們使用 JAVA 來實作我們的即時稽核架構，以及 digest function 為 java.security.MessageDigest 中的 SHA-256 algorithm，在我們實做出來的 Aggregate hash（一）架構中，PB-Pairs 會透過一個 Index function Γ 來將所有 PB-Pairs 放入 Hashmap 中儲存，但儲存的過程中可能會發生碰撞（Collision）的情況，所以我們首先進行了一百萬筆檔案（一百萬筆 PB-Pairs）的 Index function Γ 的碰撞分析，實驗數據詳見表 1

表1 Aggregate hash（一）中 Index function Γ 的碰撞測試

#Array size：Aggregate hash 架構中的 Array size 總數量；
#AVG collision：在 Array 的 Slot 中儲存 PB-pairs 的平均值；
#MAX collision：在 Array 的 Slot 中儲存 PB-pairs 的最大值；
#MIN collision：在 Array 的 Slot 中儲存 PB-pairs 的最小值；

Array size	AVG collision	MIN collision	Max collision
8	125,000	124,541	125,441
128	7,812.5	7,671	7,987
512	1,953.13	1,831	2,097
2048	488.29	412	561
8192	122.13	87	167
131072	7.71	0	27
524288	2.25	0	11
1048576	1.56	0	8

表 2、表 3 為 FBHTree 與 Aggregate hash 對於 Client 與 Server 的儲存空間需求。

表2 FBHTree 所需的儲存空間

Tree Height	Client	Server
4	32 Bytes	152.7 MB
8	32 Bytes	152.7 MB
10	32 Bytes	152.8 MB
12	32 Bytes	152.9 MB
14	32 Bytes	153.2 MB
18	32 Bytes	161.1 MB
20	32 Bytes	186.3 MB
21	32 Bytes	219.4 MB

表3 Aggregate hash 所需的儲存空間

Array Size	Client	Server
8	0.7 KB	131.5 MB
128	8.6 KB	131.5 MB
512	34.7 KB	131.5 MB
2048	138.0 KB	131.5 MB
8192	552.0 KB	131.5 MB
131072	8.6 MB	131.5 MB
524288	29.7 MB	131.5 MB
1048576	44.3 MB	131.5 MB

表 2、表 3 的實驗數據是透過 java.io.ObjectOutputStream 中的 writeObject 將建構好的 FBHTree 與 Aggregate hash 輸出成一個物件檔儲存在電腦當中。其中，FBHTree 從 14 層之後，因為樹狀結構中的內節點數量大幅增加，所以雲端服務提供商要花較多的儲存空間。

表 4 為 FBHTree 與 Aggregate hash 執行 Auditing 時 ，平均所需要的運算量。

表4 FBHTree 與 Aggregate hash 在執行 Auditing 時平均所需要的運算數量

#Δ：一個 256bit 的 hash 值 mod Array size；

#α：兩個 256bit 的 hash 值做 concatenation 後再取 SHA-256；

#β：AH 乘上一個 256bit 的 hash 值後 mod P；

Tree Height	FBHTree	Aggregate hash	Array Size
4	250,002α	125,000Δ+125,000α + 125,000β	8
8	15,630α	7,812Δ + 7,812α + 7,812β	128
10	3,914α	1,953Δ + 1,953α + 1,953β	512
12	986α	488Δ + 488α + 488β	2048
14	256α	122Δ + 122α + 122β	8192
18	30α	7Δ + 7α + 7β	131072
20	21α	2Δ + 2α + 2β	524288
21	22α	1Δ + 1α + 1β	1048576

表 5、表 6 及表 7 為使用者將一百萬筆 PB-Pairs 儲存到雲端後，在不同網路環境下向雲端儲存系統發出不同 Auditing 的 Request 後，FBHTree、Aggregate hash（一）及 Aggregate hash（二）取得所有相關的 PB-Pairs 並計算出 Root Hash 與

AH 的平均時間。

表5 無網路傳輸時間，用戶端執行稽核的平均時間（ms）

Tree Height	FBHTree	Aggregate hash（一）	Aggregate hash（二）	Array Size
4	39.57	160.12	1033.97	8
8	2.64	23.44	1033.97	128
10	0.68	8.78	1033.97	512
12	0.12	2.68	1033.97	2048
14	0.13	0.95	1033.97	8192
18	0.12	0.063	1033.97	131072
20	0.13	0.027	1033.97	524288
21	0.14	0.014	1033.97	1048576

表6 相同網路區段，用戶端執稽核的平均時間（ms）

Tree Height	FBHTree	Aggregate hash（一）	Aggregate hash（二）	Array Size
4	70.07	191.62	1153.97	8
8	18.14	38.94	1153.97	128
10	11.98	20.28	1153.97	512
12	8.61	11.18	1153.97	2048
14	6.83	7.65	1153.97	8192
18	4.32	4.063	1153.97	131072
20	4.33	4.027	1153.97	524288
21	4.34	4.014	1153.97	1048576

表7 不同網路區段，用戶端執行稽核的平均時間（ms）

Tree Height	FBHTree	Aggregate hash（一）	Aggregate hash（二）	Array Size
4	259.57	380.62	1466.89	8
8	102.14	122.94	1466.89	128
10	63.68	64.48	1466.89	512
12	35.54	38.18	1466.89	2048
14	25.92	26.45	1466.89	8192
18	22.63	22.57	1466.89	131072
20	22.63	22.53	1466.89	524288
21	22.64	22.52	1466.89	1048576

從表 4、表 5、表 6 及表 7 的實驗數據可以發現 Tree Height 在 14 層以下、Array Size 在 8192 以下時，FBHTree 的 auditing 速度會比 Aggregate hash（一）快，原因是因為樹高越低、Array Size 越小的情況下，Collision 的 PB-Pairs 數量越多，導致 Aggregate hash（一）的運算量會比 FBHTree 多很多。而 Aggregate hash（二）的 auditing 時間會花這麼多的原因是一百萬個 PB-Pairs 存放在同一個 Slot 內，導致 auditing 時的運算量比 FBHTree 與 Aggregate hash（一）多上許多。

表 8 為 FBHTree 與 Aggregate hash 執行 Insert 後更新出 Root Hash 與 AH 平

均所需要的運算量。

表8 Insert 一筆檔案後更新 Root Hash 與 AH 平均所需要的運算數量

#α：兩個 256bit 的 hash 值做 concatenation 後再取 SHA-256；

#β：AH 乘上一個 256bit 的 hash 值相乘後 mod P；

Tree Height	FBHTree	Aggregate hash	Array Size
4	250,004α	1α + 1β	8
8	15,632α	1α + 1β	128
10	3,916α	1α + 1β	512
12	998α	1α + 1β	2048
14	258α	1α + 1β	8192
18	32α	1α + 1β	131072
20	23α	1α + 1β	524288
21	24α	1α + 1β	1048576

在表 9、表 10、表 11 中，我們實測了在不同網路環境下，Insert 一筆 PB-Pair 後，FBHTree 與 Aggregate hash 更新 Root Hash 及 AH 所需要的時間。

表9 無網路時間，用戶執行 Insert 後算出 Root Hash 及 AH 的平均時間（ms）

Tree Height	FBHTree	Aggregate hash（一）	Aggregate hash（二）	Array Size
4	40.20	0.0054	0.0054	8
8	2.86	0.0054	0.0054	128
10	0.81	0.0054	0.0054	512
12	0.51	0.0054	0.0054	2048
14	0.13	0.0054	0.0054	8192
18	0.12	0.0054	0.0054	131072
20	0.12	0.0054	0.0054	524288
21	0.13	0.0054	0.0054	1048576

表10　相同網路區段，用戶執行 Insert 後算出 Root Hash 及 AH 的平均時間（ms）

Tree Height	FBHTree	Aggregate hash（一）	Aggregate hash（二）	Array Size
4	44.84	4.21	4.21	8
8	6.57	4.21	4.21	128
10	4.69	4.21	4.21	512
12	4.67	4.21	4.21	2048
14	4.57	4.21	4.21	8192
18	4.25	4.21	4.21	131072
20	4.21	4.21	4.21	524288
21	4.24	4.21	4.21	1048576

表11　不同網路區段，用戶執行 Insert 後算出 Root Hash 及 AH 的平均時間（ms）

Tree Height	FBHTree	Aggregate hash（一）	Aggregate hash（二）	Array Size
4	62.21	22.51	22.51	8
8	24.74	22.51	22.51	128
10	22.81	22.51	22.51	512
12	22.67	22.51	22.51	2048
14	22.61	22.51	22.51	8192
18	22.57	22.51	22.51	131072
20	22.57	22.51	22.51	524288
21	22.76	22.51	22.51	1048576

從表 8、表 9、表 10 及表 11 可以看出 Aggregate hash 在 Insert 一筆 PB-Pair 後更新 AH 的時間是一樣的，原因是對於 Aggregate hash 而言，無論 Arrary size 多寡，Insert 一筆 PB-Pair 後更新 AH 只是經過一次 α 與 β 運算，而對於 FBHTree 來說，就必須更新整條 Slice，所花費的時間會較多。此外，由於 Aggregate hash 的運算次數較少，對於使用者及 Server 而言，電腦所需要的能耗也降低不少。

表 12 為 FBHTree 與 Aggregate hash 執行 Delete 後更新出 Root Hash 與 AH 平

均所需要的運算量。

表12　Delete 一筆檔案後更新 Root Hash 與 AH 平均所需要的運算數量

#α：兩個 hash 值做 concatenation 後再取 SHA-256；

#λ：AH 對一個 256bits 的 hash 值進行模反元素運算；

Tree Height	FBHTree	Aggregate hash	Array Size
4	250,000α	1λ+1α	8
8	15,628α	1λ+1α	128
10	3,912α	1λ+1α	512
12	984α	1λ+1α	2048
14	254α	1λ+1α	8192
18	28α	1λ+1α	131072
20	19α	1λ+1α	524288
21	20α	1λ+1α	1048576

在表 13、表 14 及表 15 中，我們實測了在不同網路環境下，Delete 一筆 PB-Pair 後，FBHTree 與 Aggregate hash 更新 Root Hash 及 AH 所需要的時間。

表13　無網路傳輸時間，用戶執行 Delete 後算出 Root Hash 及 AH 的平均時間（ms）

Tree Height	FBHTree	Aggregate hash（一）	Aggregate hash（二）	Array Size
4	40.09	0.028	0.028	8
8	2.86	0.028	0.028	128
10	0.81	0.028	0.028	512
12	0.51	0.028	0.028	2048
14	0.13	0.028	0.028	8192
18	0.12	0.028	0.028	131072
20	0.12	0.028	0.028	524288
21	0.13	0.028	0.028	1048576

表14　相同網路區段，用戶執行 Delete 後算出 Root Hash 及 AH 的平均時間（ms）

Tree Height	FBHTree	Aggregate hash（一）	Aggregate hash（二）	Array Size
4	44.84	4.52	4.52	8
8	6.57	4.52	4.52	128
10	4.69	4.52	4.52	512
12	4.67	4.52	4.52	2048
14	4.57	4.52	4.52	8192
18	4.25	4.52	4.52	131072
20	4.21	4.52	4.52	524288
21	4.24	4.52	4.52	1048576

表15　不同網路區段，用戶執行 Delete 後算出 Root Hash 及 AH 的平均時間（ms）

Tree Height	FBHTree	Aggregate hash（一）	Aggregate hash（二）	Array Size
4	62.14	22.23	22.23	8
8	24.64	22.23	22.23	128
10	22.73	22.23	22.23	512
12	22.57	22.23	22.23	2048
14	22.51	22.23	22.23	8192
18	22.47	22.23	22.23	131072
20	22.47	22.23	22.23	524288
21	22.66	22.23	22.23	1048576

從表 12、表 13、表 14 及表 15 中可以看出 Aggregate hash 在 Delete 一筆 PB-Pair 後更新 AH 的時間是一樣的，原因是對於 Aggregate hash 而言，無論 Arrary size 多寡，Delete 一筆 PB-Pair 後更新 AH 都只會經過一次 λ 和一次 α 的運算，而對於 FBHTree 來說，就必須更新整條 Slice。此外，由於 Aggregate hash 的運算次數較少，對於使用者及 Server 而言，電腦所需要的能耗也降低不少。

在檔案系統中，除了 Insert 與 Delete 檔案外，可能會發生使用者想要將檔案從舊目錄搬移到新目錄的情況。所謂的搬移檔案，對於 FBHTree 與 Aggregate hash 所

保存的 PB-Pairs 而言，就是進行許多次的 Insert 與 Delete。

表 16 為 FBHTree 與 Aggregate hash 搬移檔案後更新出 Root Hash 與 AH 平均所需要的運算量。

表16 FBHTree 與 Aggregate hash 搬移一筆檔案後更新 Root Hash 與 AH 平均所需要的運算數量

#α：兩個 hash 值做 concatenation 後再取 SHA-256；

#λ：AH 對一個 256bit 的 hash 值進行模反元素運算；

Tree Height	FBHTree	Aggregate hash	Array Size
4	500,004α	2α+1β+1λ	8
8	31,260α	2α+1β+1λ	128
10	7,828α	2α+1β+1λ	512
12	1,972α	2α+1β+1λ	2048
14	512α	2α+1β+1λ	8192
18	60α	2α+1β+1λ	131072
20	42α	2α+1β+1λ	524288
21	44α	2α+1β+1λ	1048576

在表 17、表 18 及表 19 中，我們實測了在不同網路環境下，搬移一百筆檔案

後 FBHTree 與 Aggregate hash 計算出新的 AH 與 Root Hash 所需要的平均時間。

表17　無網路傳輸時間，用戶搬移一百筆檔案後後算出 Root Hash 及 AH 的平均時間（ms）

Tree Height	FBHTree	Aggregate hash（一）	Aggregate hash（二）	Array Size
4	8,016.32	1.86	1.86	8
8	572.46	1.86	1.86	128
10	162.72	1.86	1.86	512
12	102.36	1.86	1.86	2048
14	26.28	1.86	1.86	8192
18	26.93	1.86	1.86	131072
20	24.35	1.86	1.86	524288
21	26.14	1.86	1.86	1048576

表18　相同網路區段，用戶搬移一百筆檔案後算出 Root Hash 及 AH 的平均時間（ms）

Tree Height	FBHTree	Aggregate hash（一）	Aggregate hash（二）	Array Size
4	8,021.32	5.63	5.63	8
8	577.46	5.63	5.63	128
10	168.72	5.63	5.63	512
12	106.36	5.63	5.63	2048
14	31.28	5.63	5.63	8192
18	30.93	5.63	5.63	131072
20	29.35	5.63	5.63	524288
21	30.14	5.63	5.63	1048576

表19　不同網路區段，用戶搬移一百筆檔案後算出 Root Hash 及 AH 的平均時間（ms）

Tree Height	FBHTree	Aggregate hash（一）	Aggregate hash（二）	Array Size
4	8,176.32	22.23	22.23	8
8	634.46	23.37	23.37	128
10	202.72	23.37	23.37	512
12	163.36	23.37	23.37	2048
14	47.28	23.37	23.37	8192
18	46.93	23.37	23.37	131072
20	46.35	23.37	23.37	524288
21	47.14	23.37	23.37	1048576

從上面三個表格可以看出 Aggregate hash 在搬移一百筆檔案後更新 AH 的時間是一樣的，原因是對於 Aggregate hash 而言，無論 Arrary size 多寡，搬移一百筆 PB-Pair 後更新 AH 只是經過一百次 Insert 與 Delete 的運算，而對於 FBHTree 來說，搬移一筆檔案須要更新兩條 Slice，所以搬移一百筆檔案則必須更新兩百條 Slice。

在表 20、表 21 及表 22 中，我們實測了在不同網路環境下，搬移一千筆檔案

後 FBHTree 與 Aggregate hash 計算出新的 AH 與 Root Hash 所需要的平均時間。

表20　無網路傳輸時間，用戶搬移一千筆檔案後後算出 Root Hash 及 AH 的平均時間（ms）

Tree Height	FBHTree	Aggregate hash（一）	Aggregate hash（二）	Array Size
4	80,163.20	20.03	20.03	8
8	5,724.63	20.03	20.03	128
10	1,627.21	20.03	20.03	512
12	1,023.64	20.03	20.03	2048
14	262.84	20.03	20.03	8192
18	269.37	20.03	20.03	131072
20	243.54	20.03	20.03	524288
21	261.47	20.03	20.03	1048576

表21　相同網路區段，用戶搬移一千筆檔案後算出 Root Hash 及 AH 的平均時間（ms）

Tree Height	FBHTree	Aggregate hash（一）	Aggregate hash（二）	Array Size
4	80,167.23	5.63	5.63	8
8	5,728.63	5.63	5.63	128
10	1,631.34	25.11	25.11	512
12	1,027.83	25.11	25.11	2048
14	266.75	25.11	25.11	8192
18	254.47	25.11	25.11	131072
20	246.54	25.11	25.11	524288
21	264.47	25.11	25.11	1048576

表22　不同網路區段，用戶搬移一千筆檔案後算出 Root Hash 及 AH 的平均時間（ms）

Tree Height	FBHTree	Aggregate hash（一）	Aggregate hash（二）	Array Size
4	80,367.47	42.57	42.57	8
8	5,848.59	42.57	42.57	128
10	1,697.21	42.57	42.57	512
12	1,058.64	42.57	42.57	2048
14	282.84	42.57	42.57	8192
18	289.37	42.57	42.57	131072
20	283.54	42.57	42.57	524288
21	281.47	42.57	42.57	1048576

在表 23、表 24 及表 25 中，我們實測了在不同網路環境下，搬移一萬筆檔案後 FBHTree 與 Aggregate hash 計算出新的 AH 與 Root Hash 所需要的平均時間。

表23　無網路傳輸時間，用戶搬移一萬筆檔案後後算出 Root Hash 及 AH 的平均時間（ms）

Tree Height	FBHTree	Aggregate hash（一）	Aggregate hash（二）	Array Size
4	801,632	138	138	8
8	57,246	138	138	128
10	16,272	138	138	512
12	10,236	138	138	2048
14	2,628	138	138	8192
18	2,693	138	138	131072
20	2,435	138	138	524288
21	2,614	138	138	1048576

表24　相同網路區段，用戶搬移一萬筆檔案後算出 Root Hash 及 AH 的平均時間（ms）

Tree Height	FBHTree	Aggregate hash（一）	Aggregate hash（二）	Array Size
4	801,662	5.63	5.63	8
8	57,285	143	143	128
10	16,312	143	143	512
12	10,279	143	143	2048
14	2,668	143	143	8192
18	2,546	143	143	131072
20	2,467	143	143	524288
21	2,643	143	143	1048576

表25　不同網路區段，用戶搬移一萬筆檔案後算出 Root Hash 及 AH 的平均時間（ms）

Tree Height	FBHTree	Aggregate hash（一）	Aggregate hash（二）	Array Size
4	801,873	161	161	8
8	57,425	161	161	128
10	16,373	161	161	512
12	10,309	161	161	2048
14	2,688	161	161	8192
18	2,567	161	161	131072
20	2,489	161	161	524288
21	2,667	161	161	1048576

從一系列搬移檔案的實驗中，我們可以發現在搬移一個檔案後更新出 Root Hash 及 AH 所需要的運算量及運算時間差距是非常大的，也就是說 Aggregate hash 只需要一點能耗與運算量就能做出與 FBHTree 同等級的密碼學稽核架構。

第五章 結論

現行的雲端服務越來越多樣，也越來越普及，讓使用者可以將許多重要文件與檔案存放在雲端上，不僅便利，也讓使用者節省了需多硬體設置與維護的成本。但雲端服務仍然會遭受到駭客甚至是雲端服務提供商本身惡意的攻擊，就算讓使用者檢查雲端服務系統的監控紀錄或者日誌黨也未必能夠確定這些紀錄與檔案是否是正確的，所以我們實驗室所提出的稽核機制與證明違約協定就可以改善類似的情況發生。

這篇論文所介紹的稽核機制 Aggregate hash，根據實驗數據可以發現，使用者只需要花費一些儲存空間來保存稽核證據，不僅在稽核運算方面非常快速，運算量也節省許多，這對於使用者或是雲端服務商而言，都可以節省許多能耗，又能達到過往稽核機制同等級的功能。

第六章 參考著作

[1] *Cloud Storage*.Received from: https://en.wikipedia.org/wiki/Cloud_Storage/

[2] *Google Drive*. Received from: https://www.google.com/drive/

[3] *Dropbox*. Received from: https://www.dropbox.com

[4] *OneDrive*. Received from: https:// onedrive.live.com/about/zh-tw/

[5] *iCloud*. Received from: https:// www.icloud.com/

[6] *SugarSync*. Received from: https:// www2.sugarsync.com/

[7] Gwan-Hwan Hwang,Wei-Sian Huang, Jenn-Zjone Peng and Yu-Wei Lin. (2014). *Fulfilling mutual nonrepudiation for cloud storage. CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE*.

[8] Gwan-Hwan Hwang, Wei-Sian Huang, and Jenn-Zjone Peng. (2014). *Real-time Proof of Violation for Cloud Storage. IEEE 6th International Conference on Cloud Computing Technology and Science.*

[9] R. A. Popa and J. R. Lorch. (2011). *Enabling Security in Cloud Storage SLAs with CloudProof. USENIX Annual Technical Conference （USENIX）, pp. 31.*

[10] *SHA-256*. Received from Wikipedia: https://en.wikipedia.org/wiki/SHA-2

[11] *Hash*. Received from Wikipedia: https://en.wikipedia.org/wiki/Hash_function

[12] *Digital Signature*. Received from Wikipedia: https://en.wikipedia.org/wiki/Digital_signature

[13] Gwan-Hwan Hwang, Wei-Sian Huang, and Jenn-Zjone Peng. (2014). *Real-time Proof of Violation for Cloud Storage. IEEE 6th International Conference on Cloud Computing Technology and Science.*

[14] Joseph.A.Gallian.*Contemporary Abstract Algebra 8th edition*.