Abstract

In electronic commerce applications, such as e-banking, e-voting, software authentication, etc., we need to sign many messages at one time. So the technique of batch verification is developed. We can use batch verification scheme to verify a large amount of signatures fast. But some serious problem may happen while the design of batch verification isn't secure enough. The attacker may forge a group of signatures which can be passed by the batch verification, but the individual signature can't be passed the verification. So this paper studied the weaknesses of batch verification and aimed to improve the security of batch verification. Next, some batch verifications are base on Digital Signature Algorithm (DSA)，DSA needs to compute inverse. Computing inverse needs a large of computation costs. Our proposed scheme can reduce inversion computation. And we also simulate the performance of our proposed scheme.

Keywords: Digital Signature Algorithm, RSA, ElGamal Signature Algorithm, Schnorr Signature Algorithm, Batch Verification, Inverse, Small Exponents Test

摘要

在電子商務上，電子簽章的應用非常的頻繁。這些簽章的應用包括電子錢、電子銀行、電子投票、軟體保護及電子公文等等。在某些應用當中，常常需要在短時間內驗證大量的簽章，例如電子錢的驗證及電子投票的驗證等等，因此而發展出整批式驗證簽章的方法，整批式簽章驗證的方法能夠快速的在短時間之內同時驗證大量的簽章，但是整批式簽章驗證的方法當中，可能因為架構設計的問題而使得整批式驗證的過程出現嚴重的錯誤，也就是簽章者有可能偽造出一批假的簽章，可以通過整批式簽章驗證。所以在本論文當中除了對於整批式簽章驗證的方法的弱點作一些探討之外，我們也試著發展出可靠度較高的驗證方法。另外一方面，在整批式簽章驗證的方式當中，有些採用 Digital Signature Algorithm (DSA) 作為基礎的整批式簽章驗證架構，而 DSA 的架構當中需要計算乘法反元素，但是計算乘法反元素需要耗費龐大的計算量，它需要運用到大量的除法運算。所以在我們的論文當中也運用了整批式簽章驗證的特性，將 DSA 的整批式驗證架構當中，原本需要計算大量的乘法反元素的架構經過修改之後，以乘法運算來取代掉乘法反元素的運算，並且利用程式的模擬與其他學者所提出的整批式驗證架構做效能的比較。

關鍵字：數位簽章演算法，RSA，ElGamal 簽章演算法，Schnorr 簽章演算法，整批式驗證，反元素，小指數測試。

目錄

圖表目錄

第一章 導論

在 1991 年，美國聯邦政府提出了數位簽章標準 (Digital Signature Standard, DSS) [5] ，數位簽章的技術也日益的重要，而數位簽章的價值不管是在商業或是在其他方面都有其應用的價值，比如說電子錢、軟體保護、SSL、電子公文及電子投票的應用等等，都可以看出電子簽章技術的重要性。在現今的電子商務當中，簽章技術的應用更是非常的頻繁且常常需要驗證大量的簽章，例如電子錢的簽章及驗證，但是在電子錢的應用當中常常需要在短時間內驗證大量的簽章，這也考驗著伺服器的能力是否有足夠的運算能力，以及是否能提供更快的服務。因此 Naccache [6] 等人在 1994 年提出了整批式驗證的架構，此架構是以 Digital Signature Algorithm (DSA)為基礎，主要是運用乘法群的觀念，在指數次項以加法運算來取代指數運算，達到減少指數運算的目的。但是在同年 Lim 和 Lee [7] 兩個人提出了能夠偽造簽章的攻擊方法，此方法可以偽造出能夠通過整批式驗證的簽章，但是這些偽造的簽章在個別的驗證是不正確的。接著韓亮 [9] 在 1995 年提出了簽章架構類似 DSA 架構的整批式驗證的方法，它的簽章架構為十八個 ElGamal 簽章架構當中的一種，韓亮所提出的方法能夠有效的防止 Lim 和 Lee 兩位提出來的攻擊，但是韓亮的方法當中存在著設計的錯誤，這個問題在第三章會提到，並且提出改進。之後又有許多的

攻擊被提出來，例如 Boyd 和 Pavlovski [12] 所提出的攻擊，同樣能夠偽造出能通過整批式驗證的簽章，Hwang [15] 等人也提出了能夠偽造簽章的攻擊，這些攻擊的提出都考驗著整批式簽章的安全性。另外一方面，如何防止簽章偽造攻擊的方法也一一的被提出，像是 Bellare [13] 等人提出了數個測試方法，能夠有效的預防簽章偽造的攻擊成立，Boyd 和 Pavlovski [12] 也提出了防止簽章偽造的方法。在這些整批式簽章驗證的研究當中，我們針對了效能的問題以及安全性的問題方面，我們提出了高效能且高安全度的方法。在效能的問題上，我們提出了在 DSA 簽章架構底下的整批式驗證架構能夠有效減少計算乘法反元素的方法，而在安全性的問題上，我們提出了一個高安全性的檢驗方法，能夠防止各種攻擊的成立。接下來介紹本論文的編排，首先在第二章當中，我們會介紹到與本論文研究相關的基礎密碼學知識。在第三章我們會將其他人提出來的整批式驗證方法作介紹，也會將整批式驗證的攻擊方法做個介紹以及將如何防止攻擊的方法作介紹。接下來在第四章當中，會介紹本論文所提出來的整批式驗證方法，以及討論我們所提出的方法對整批式驗證的效能所做的改進。接下來做方法的效能分析以及安全性的分析，並且將我們所提出的方法與其他的方法做一個比較，以及利用程式模擬做效能的分析比較。在第五章，我們提出了能有效防止簽章偽造攻擊的測試方法，並

且分析其效能及安全度。最後在第六章，我們會為整批式驗證的研究做一些結論。

第二章 理論基礎

第一節 ElGamal 簽章架構

ElGamal [1,2] 的架構不只是應用在加解密上，也可以應用在簽章技術上。ElGamal 架構的安全度是建立在解離散對數的困難問題上，所謂解離散對數的問題，常利用來設計公開金鑰系統，接著我們在下面做個介紹。

為了產生一對金鑰，首先我們選擇一個質數，p，以及兩個亂數，g 和 x，並且使得這兩個亂數小於 p，然後計算

$$y=g^x \bmod p$$

y、g 和 p 是所謂的公開金鑰，g 和 p 可以分享給同一個群組的使用者，x 則是所謂的秘密金鑰。像這樣的公開金鑰系統，當攻擊者得知公開金鑰，也就是 y、g 和 p，卻很難解出秘密金鑰 x，這就是所謂的離散對數的難題。

接下來我們就介紹一下 ElGamal 的簽章架構，此簽章架構當中的公開金鑰為 y、g 和 p 以及私密金鑰為 x，假設現在有一個訊息 M 需要被簽署，則我們執行以下的動作。首先我們選取一個亂數，k，並且使得 k 與 p-1 互質，接著計算，

$$a = g^k \bmod p$$

並且用歐基里得演算法 (extended Euclidean Algorithm)以及下列式子

來解出 b，

$$M = (xa + kb) \bmod (p\text{-}1)$$

而 a 和 b 是我們所求得的 M 的簽章對，其中 k 是必須保密的。

接著驗證者要驗證簽章，可以使用下列的方程式，

$$y^a a^b \bmod p = g^M \bmod p$$

假如這個方程式等式成立，我們就可以說簽章 a 及 b 是正確的。

第二節 Schnorr 簽章架構

Schnorr [3,4] 的簽章架構同樣是建構在解離散對數的困難度上，接下來我們介紹 Schnorr 的簽章架構，以下為此簽章架構所必須要的共同參數：

p：一個大質數。

q：一個$(p\text{-}1)$的質因數。

a：一個不為 1 的數，且會讓下列方程式，$a^p \equiv 1$，成立。

M：欲簽章的訊息。

$H(\cdot)$：雜湊函數。

接下來為 Schnorr 簽章演算法的簽章及驗證過程。

簽章過程：

步驟一：簽章者選擇一亂數，r，且此亂數小於 q，接著計算

$x = a^r \bmod p$ 。

步驟二：接著將 M 及 x 連結在一起並用雜奏函數計算出結果，

$$e = H(M,x)$$

步驟三：計算 $y = (r + se) \bmod q$，最後將簽章 e 及 y 送給驗證者。

驗證過程：

驗證者計算 $x' = a^y v^e \bmod p$ ，接著計算 $e = H(M, x')$ ，假如計算結果相等，則可以確定簽章正確無誤。

第三節 數位簽章演算法 (Digital Signature Algorithm)

美國聯邦政府在 1991 年的八月，提出數位簽章演算法(DSA) [5]的標準。而這個提案充滿了爭議性，也被廣泛的討論，以及當時密碼學另外一個的簽章標準 RSA 則普遍被業界所接受。不過不管爭議如何，DSA 目前也提供了一個簽章的標準，現在我們就來介紹一下 DSA，DSA 是結合了 Schnorr 以及 ElGamal 簽章演算法的變形，以下定義了 DSA 所使用到的參數：

p：一個長度為 512 到 1024 位元之間的質數，且其位元長度必須為 64 的倍數。

q：p-1 的質因數且長度為 160 位元。

g：一個利用方程式，$g = h^{(p-1)/q} \bmod p$，計算出來的原根，且序為

q，其值必須大於 1，而 h 是小於 p-1 的數。

x：是一個小於 q 的數，而且是簽章者的秘密金鑰。

y：以方程式 $y = g^x \bmod p$ 產生的數，而且是簽章者的公開金鑰。

$H(\cdot)$：一個標準的雜湊函數演算法。

m：要簽章的訊息。

簽章過程：

步驟一：簽章者產生一個小於 q 的亂數，k。

步驟二：簽章者利用下列兩個方程式來產生簽章，

$$r = (g^k \bmod p) \bmod q \tag{1}$$

$$s = (k^{-1}(H(m) + xr)) \bmod q \tag{2}$$

則 r，s 為訊息 m 的簽章對，並且將簽章對送給驗證者。

驗證過程：

驗證者將所收到的簽章 r，s，利用下列方程式來做驗證，

$$w = s^{-1} \bmod q$$

$$u_1 = (H(m) * w) \bmod q$$

$$u_2 = (rw) \bmod q$$

$$v = ((g^{u_1} y^{u_2}) \bmod p) \bmod q \tag{3}$$

假如 $v = r$，則簽章通過驗證。

第三章 文獻探討

第一節 整批式驗證架構

一、Naccache 等人的整批式驗證方法

Naccache [6] 等人所提出的整批式驗證方法是以 DSA 為基礎，以下介紹此方法的參數、簽章過程以及驗證過程。首先我們定義一些整批式驗證的參數：

p：一個大質數。

q：$p-1$的質因數。

g：在 GF (p)裡一個序q的元素。

x：簽章者的私鑰並且此私鑰在 GF (q)當中。

y：簽章者的公鑰並且$y = g^x \bmod p$。

k_i：一群小於 q 的亂數，$i = 1,2,\ldots,t$。

簽章過程：

在準備好簽章的參數之後，簽章者準備好 t 個訊息 $m_1, m_2, \ldots, m_t$，以及定義好的參數來產生 t 個簽章 $(r_1, s_1), (r_2, s_2), \ldots, (r_t, s_t)$。這些簽章$(r_i, s_i)$是以下面兩個方程式所產生的，$r_i = g^{k_i} \bmod p$ 以及 $s_i = k_i^{-1}(m_i + xr_i) \bmod q$ 當 $i = 1,2,...,t$。

驗證過程：

簽章者產生完所有簽章之後將這些簽章送給驗證者。接下來

驗證者會依照下列的方程式來檢驗這一群簽章是否正確：

$$\prod_{i=1}^{t} r_i \stackrel{?}{\equiv} g^{\sum_{i=1}^{t} m_i s_i^{-1} \bmod q} y^{\sum_{i=1}^{t} r_i s_i^{-1} \bmod q} \pmod p \qquad (4)$$

假如上述的方程式等式成立，則所有簽章都是正確的，假如方程式等式不成立，則至少有一個以上的簽章錯誤。

二、韓亮的整批式驗證方法

韓亮 [8,9] 所提出的整批式簽章主要是以 ElGamal 簽章架構當中的其中一種為基礎。以下為此整批式驗證的方法，首先定義參數如下：

p：一個大質數。

q：$p-1$的質因數。

g：在 GF (p)裡一個序q的元素。

x：簽章者的私鑰並且此私鑰在 GF (q)當中。

y：簽章者的公鑰並且$y = g^x \bmod p$。

k_i：一群小於 q 的亂數，$i = 1,2,\ldots,t$。

簽章過程：

當所有簽章所需的參數準備好了之後，簽章者利用這些參數、t 個訊息 $m_1, m_2, \ldots, m_t$ 以及下面兩個簽章的方程式 $r_i = (g^{k_i} \bmod p) \bmod q$ 和 $s_i = (r_i k_i - m_i x) \bmod q$ 當 $i = 1,\ldots,t$，來產生所有的

簽章 (r_i, s_i) 當 $i = 1,...,t$ 。

驗證過程：

當驗證者收到所有的簽章後，他可以利用下列方程式來檢驗所有的簽章是否都正確：

$$\prod_{i=1}^{t} r_i \overset{?}{\equiv} (g^{\sum_{i=1}^{t} s_i r_i^{-1} \bmod q} y^{\sum_{i=1}^{t} m_i r_i^{-1} \bmod q} \bmod p) (\bmod q) \quad (5)$$

假如上述的方程式等式成立，則所有簽章都是正確的，假如方程式等式不成立，則至少有一個以上的簽章錯誤。

三、對於韓亮整批式驗證方法的評論

根據我們的觀察，韓亮的整批式驗證方法在設計上，出現了會使得驗證方程式無法成立的狀況，以下的證明說明了驗證方程式為何無法成立，

左式為

$$(r_1 r_2 \cdots r_t) \bmod q = ((g^{k_1} \bmod p)(g^{k_2} \bmod p) \cdots (g^{k_t} \bmod p)) \bmod q$$

右式為

$$\begin{aligned}
&(g^{s_1 r_1' + s_2 r_2' + ... + s_t r_t'} y^{m_1 r_1' + m_2 r_2' + ... + m_t' r_t'}) \bmod p \bmod q \\
&= ((g^{(r_1 k_1 - m_1 x) r_1'} g^{x m_i r_1'} \bmod p)(g^{(r_2 k_2 - m_2 x) r_2'} g^{x m_2 r_2'} \bmod p) \cdots (g^{(r_t k_t - m_t x) r_t'} g^{x m_t r_t'} \bmod p)) \bmod p \bmod q \\
&= ((g^{k_1} \bmod p)(g^{k_2} \bmod p) \cdots (g^{k_t} \bmod p)) \underline{\bmod p} \bmod q
\end{aligned}$$

而左右兩式經過推導之後明顯的看出來，左右兩式不相等，這會使得整批式驗證方程式無法成立，假如將整批式驗證方程式

改成以下的方程式，

$$(r_1 r_2 ... r_t) \bmod p \stackrel{?}{=} g^{s_1 r_1' + s_2 r_2' + ... + s_t r_t'} y^{m_1 r_1' + m_2 r_2' + ... + m_t r_t'} \bmod p \quad (6)$$

則等式會成立，以下為證明，

左式為

$$(r_1 r_2 \cdots r_t) \bmod p = ((g^{k_1} \bmod p)(g^{k_2} \bmod p) \cdots (g^{k_t} \bmod p)) \bmod p$$

右式為

$$\begin{aligned}
&(g^{s_1 r_1' + s_2 r_2' + ... + s_t r_t'} y^{m_1 r_1' + m_2 r_2' + ... + m_t r_t'}) \bmod p \\
&= ((g^{(r_1 k_1 - m_1 x) r_1'} g^{x m_1 r_1'} \bmod p)(g^{(r_2 k_2 - m_2 x) r_2'} g^{x m_2 r_2'} \bmod p) \cdots (g^{(r_t k_t - m_t x) r_t'} g^{x m_t r_t'} \bmod p)) \bmod p \\
&= ((g^{k_1} \bmod p)(g^{k_2} \bmod p) \cdots (g^{k_t} \bmod p)) \bmod p
\end{aligned}$$

因為左式等於右式，所以韓亮的整批式驗證方程式經過修改之後仍然可以正確的驗證整批式的簽章。

四、Yen 和 Laih 的整批式驗證方法

Yen 和 Laih [17] 提出了以 RSA [18] 為基礎的整批式驗證架構，接下來我們就針對這個方法做介紹，首先定義整批式驗證的參數：

N：RSA 的模數。

e：RSA 的公開金鑰。

d：RSA 的私密金鑰。

m_i：欲簽署的訊息，$i = 1,2,\ldots,t$。

S_i：所有訊息的簽章，$S_i = m_i^d \bmod N$ ，$i = 1,2,\ldots,t$。

s_i：長度為 l 位元的亂數，$i = 1,2,\ldots,t$。

驗證過程：

$$\left(\prod_{i=1}^{t} S_i^{s_i}\right)^e \equiv \prod_{i=1}^{t} m_i^{s_i} \bmod N \qquad \textbf{(7)}$$

假如以上方程式等式成立，則驗證成功。

第二節 整批式驗證的攻擊方法

一、Lim 和 Lee 的攻擊方法

Lim 和 Lee [7] 兩個人在 1994 年提出了對 Naccache 等人提出的整批式驗證方法的攻擊方法，此攻擊方法可以產生一群偽造的簽章，但是這群被偽造的簽章依然能夠通過整批式驗證的方程式，使得攻擊者達到偽造的整批式簽章被驗證通過的目的。以下我們就來介紹一下 Lim 和 Lee 的攻擊方法是如何執行的。首先驗證所需的參數與先前本章第一節所介紹的 Naccahe 整批式驗證方法的參數是相同的，接著簽章者能夠以下列步驟偽造簽章。

步驟一：在 GF(q)當中亂數選擇 $2t$ 個數 u_i, v_i 並且計算

$r_i = g^{u_i} y^{v_i} \bmod p$ 當 $i = 1,2,\ldots,t$。

步驟二：計算 $s_i = v_i^{-1} r_i \bmod q$ ，當 i=1,2,…,t-2。

步驟三：依照下列兩個方程式求出 s_{t-1}, s_t 的解，

$$S_{t-1}^{-1} m_{t-1} + s_t^{-1} m_t \equiv \sum_{i=1}^{t} u_i - \sum_{i=1}^{t-2} s_i^{-1} m_i (\bmod q)$$

$$s_{t-1}^{-1}r_{t-1}+s_t^{-1}r_t \equiv v_{t-1}+v_t(\bmod q)$$

則依照這些步驟所求出來的所有簽章將可以通過整批式驗證的方程式。

二、Hwang 等人的攻擊方法

Hwang [15] 等人在 2001 年針對於韓亮提出的整批示驗證方法的攻擊，他們所提出的攻擊方法也是以簽章的偽造做攻擊，以下就對 Hwang 等人提出的攻擊方法做介紹，首先簽章所需的參數 p,q,g,x,y 的條件和本章第一節當中提到的韓亮的整批式驗證方法一樣，接著惡意的簽章者利用原本產生的合法簽章(r_i,s_i)當 $i=1,2,\ldots,t$，來產生偽造的簽章$(r_1,s_1'),(r_2,s_2'),\ldots,(r_t,s_t')$，當 $i=1,2,\ldots,t$，而偽造的簽章產生的方法如下：

首先簽章者產生 $a_1,a_2,\ldots,a_t$，且符合 $\sum_{i=1}^{t}a_i=0$ 的條件，然後利用這群準備好的數來產生 $s_i'=s_i+a_ir_i(\bmod q)$，當 $i=1,2,\ldots,t$，這樣就可以偽造出假的簽章，且這樣子的簽章依然能夠通過整批式驗證的方程式：

$$\prod_{i=1}^{t}r_i \overset{?}{\equiv}(g^{\sum_{i=1}^{t}s_i'r_i^{-1} \bmod q}y^{\sum_{i=1}^{t}m_ir_i^{-1} \bmod q} \bmod p)(\bmod q) \qquad (8)$$

但是個別驗證卻不正確，因為 $r_i \neq (g^{s_i'r_i^{-1}}y^{m_ir_i^{-1}} \bmod p) \bmod q$。此攻擊方法不只對於韓亮的攻擊方法有效，而且也對 Naccache 等人提出的

DSA 整批式驗證方法有效，同樣利用 $a_1, a_2, ..., a_t$ ，且 $\sum_{i=1}^{t} a_i = 0$ ，並且計算 $r_i' = r_i + s_i a_i \pmod{q}$ ，來產生偽造的簽章，$r_1', r_2', ..., r_t'$ ，就可以通過整批式驗證方程式，$\prod_{i=1}^{t} r_i \stackrel{?}{\equiv} g^{\sum_{i=1}^{t} m_i s_i^{-1} \bmod q} y^{\sum_{i=1}^{t} r_i' s_i^{-1} \bmod q} \pmod{p}$ ，但是個別的驗證卻不會通過，因為 $r_i = (g^{m_i s_i^{-1}} y^{r_i' s_i^{-1}}) \bmod p$ 。

三、Boyd 及 Pavlovski 的攻擊方法

Boyd 及 Pavlovski [12] 在其論文當中提出了對於整批式驗證的攻擊方法，分別是針對 Yen 和 Laih 的整批式驗證方法、韓亮的整批式驗證方法，以下就一一的作介紹。

首先介紹針對 Yen 和 Laih 的整批式攻擊方法做介紹，Yen 和 Laih 的整批式驗證方程式為 $\left(\prod_{i=1}^{t} S_i^{s_i}\right)^e \equiv \prod_{i=1}^{t} m_i^{s_i} \bmod N$，當我們將某些 S_i 換成-S_i 以及 m_i 換成-m_i，則驗證方程式仍然有 1/2 的機率能通過驗證。

接下來介紹針對韓亮的整批式驗證攻擊方法，韓亮的整批式驗證方程式為 $\prod_{i=1}^{t} r_i \stackrel{?}{\equiv} (g^{\sum_{i=1}^{t} s_i r_i^{-1} \bmod q} y^{\sum_{i=1}^{t} m_i r_i^{-1} \bmod q} \bmod p) \pmod{q}$，現在假設攻擊者偽造了 t-1 個簽章 $s_1', s_2', ..., s_{t-1}'$ ，則攻擊者可以利用解下列的方程式來解 s_t' ，

$$\sum_{i=1}^{t} s_i' r_i^{-1} \bmod q = \sum_{i=1}^{t} s_i r_i^{-1} \bmod q \qquad (9)$$

我們以下面的例子來說明如何使這樣的偽造簽章攻擊能夠成立，

首先我們定義三個數 A，B，C，使得 $A = (g^B y^C \bmod p) \bmod q$，接著攻擊者選擇兩個訊息 m_1，m_2，然後解下面兩個方程式來達到攻擊的目的：

步驟一：利用下列兩個方程式來解 r_1，r_2

$$r_1 r_2 \equiv A \bmod q$$
$$m_1 r_1^{-1} + m_2 r_2^{-1} \equiv C \bmod q$$

步驟二：利用下列方程式解 s_1，s_2

$$s_1 r_1^{-1} + s_2 r_2^{-1} \equiv B \bmod q$$

接著討論步驟一能夠找到解的機率有多少，在步驟一的兩個方程式能夠化減為 $(m_2 / A) r_1^2 - C r_1 + m_1 \bmod q$，其判別式為 $C^2 - 4 m_1 m_2 / A$，假如 m_1，m_2 是亂數選擇的話，則有 1/2 的機率有解。

第三節 防止簽章偽造攻擊的方法

在上一節當中所介紹到對於整批式驗證的攻擊方法，都是針對簽章的偽造，因此在 1998 年 Bellare [13] 等人在其論文當中提出了三種驗證過程的檢驗方法，這些方法可以在一定的機率底下防止簽章的偽造，接下來我們就對這三種方法做介紹。

首先定義一些參數：

G：是一個序 q 的群。

q：質數。

g：G 的原根。

(x_i,y_i)：$x_i \in Z_p$ ，$y_i \in G$ ，i=1,2,…,n。

l：安全度參數的位元數。

一、亂數子集合測試 (Random Subset Test)

此測試方法依照以下的步驟來執行。

步驟一：亂數選取 $b_i \in \{0,1\}$ ，i=1,2,…,n。

步驟二：製作一集合 $S = \{i:b_i=1\}$。

步驟三：計算 $x = \sum_{i \in S} x_i \bmod q$ 以及 $y = \prod_{i \in S} y_i$ 。

步驟四：假如 $g^x=y$ 則通過驗證。

二、小指數測試 (Small Exponents Test)

以下為小指數測試方法的步驟：

步驟一：亂數選擇 $s_1,s_2,\ldots,s_n \in \{0,1\}^l$ 。

步驟二：計算 $x = \sum_{i=1}^{n} x_i s_i \bmod q$ 以及 $y = \prod_{i=1}^{n} y_i^{s_i}$ 。

步驟三：假如 $g^x=y$ 則通過驗證。

三、槽狀測試 (Bucket Test)

首先我們假設 m 為安全參數，並且 $m \geq 2$ ，使得 $M = 2^m$，接下來

介紹一下槽狀測試的方法：

步驟一：亂數選擇 $t_i \in \{1,2,...,M\}$，$i = 1,2,\ldots,n$。

步驟二：使得 $B_j = \{i : t_i = j\}$，$j = 1,2,\ldots,M$。

步驟三：計算 $c_j = \sum_{i \in B_j} x_i \bmod q$ 以及 $d_j = \prod_{i \in B_j} y_i$ 。

步驟四：接著使用小指數測試方法來測試這些整批式驗證實體，$(c_1,d_1), (c_2,d_2),\ldots,(c_M,d_M)$，假如所有整批式實體通過驗證則所有簽章正確。

四、Bellare 等人的整批式驗證方法

Bellare [13] 等人所提出的方法是將原本的 DSA 整批式驗證方法加上 *Small Exponents Test* 來抵擋簽章偽造的攻擊，接下來就針對此方法訂定一些參數，定義如下：

p：一個大質數。

q：$p-1$ 的質因數。

g：在 GF (p)裡一個序 q 的元素。

x：簽章者的私鑰並且此私鑰在 GF (q)當中。

y：簽章者的公鑰並且 $y = g^x \bmod p$ 。

簽章過程：

當產生以上參數之後，簽章者利用訊息 $m_1, m_2, \ldots, m_t$ 來產生

一群簽章$(r_1,s_1),(r_2,s_2),\ldots(r_t,s_t)$，這些簽章是以 DSA 簽章演算法來產生的，$r_i = g^{k_i} \bmod p, s_i = k_i^{-1}(m_i + xr_i) \bmod q$ 。

驗證過程：

當驗證者收到所有的簽章之後，驗證者利用下面步驟來驗證所有的簽章。

步驟一：亂數選擇$b_1,b_2,\ldots,b_t \in \{0,1\}^l$ 。

步驟二：利用下列的方程式來驗證所有的簽章是否正確，

$$\prod_{i=1}^{t} r_i^{b_i} \pmod p \stackrel{?}{=} (g^{\sum_{i=1}^{t} m_i s_i^{-1} b_i \bmod q} y^{\sum_{i=1}^{t} r_i s_i^{-1} b_i \bmod q}) \bmod p \text{ 。} \quad (10)$$

第四章 有效率的整批式驗證架構

第一節 整批式驗證方法

在本論文當中，提出了具有高效率及高安全度的整批式驗證的方法，我們所提出的方法最主要是減少乘法反元素的運算以提高效能，以下就介紹一下本架構如何運作，首先我們定義整批式驗證所需要的參數如下：

p：一個大質數。

q：$p-1$的質因數。

g：在 GF (p)裡一個序q的元素。

x：簽章者的私鑰並且此私鑰在 GF (q)當中。

y：簽章者的公鑰並且 $y = g^x \bmod p$ 。

簽章過程：

當產生以上參數之後，簽章者利用訊息 $m_1, m_2, \ldots, m_t$ 來產生一群簽章 $(r_1, s_1), (r_2, s_2), \ldots (r_t, s_t)$，這些簽章是以 DSA 簽章演算法來產生的。

驗證過程：

當驗證者收到所有的簽章之後，驗證者利用下面步驟來驗證所有的簽章。

步驟一：亂數選擇 $b_1, b_2, \ldots, b_t \in \{0,1\}^l$ 。

步驟二：先計算出 $S_j = \prod_{i=1 \wedge i \neq j}^{t} s_i \bmod q$ ， 當 $j = 1,2,...,t$ ， 然後計

算出 $S = s_1 S_1 \bmod q$ 。

步驟三：利用下列的方程式來驗證所有的簽章是否正確。

$$\left(\prod_{i=1}^{t} r_i^{b_i}\right)^{S} \bmod p \stackrel{?}{=} \left(g^{\sum_{i=1}^{t} m_i S_i b_i \bmod q} y^{\sum_{i=1}^{t} r_i S_i b_i \bmod q}\right) \bmod p \qquad (11)$$

由驗證的這些步驟我們可以看到，原本 DSA 簽章演算法的驗證過程需要計算大量的乘法反元素，但是我們的架構當中是利用少量的乘法來取代了原本的乘法反元素計算，而如何快速的計算在驗證過程當中步驟二所提到的 S_j，在下一節我們會提到一個加速的演算法來加速 S_j 的計算。在這裡我們先解釋一下如何能夠能夠快速的算出 S_j，首先我們先算出 2(t-1)的值，如圖 1，

$A_1 = s_1$	$B_1 = s_t$
$A_2 = s_1 s_2$	$B_2 = s_{t-1} s_t$
$A_3 = s_1 s_2 s_3$	$B_3 = s_{t-2} s_{t-1} s_t$
.	.
.	.
.	.
$A_{t-1} = s_1 s_2 s_3 \ldots s_{t-1}$	$B_{t-1} = s_2 s_3 s_4 \ldots s_t$

圖 1 加速演算法說明一

接著利用以上的值來計算 S_j，如圖 2 所示

$S_1 = B_{t\text{-}1} = \; s_2 s_3 s_4 \ldots s_t$

$S_2 = A_1 * B_{t\text{-}2} = \; s_1 * s_3 s_4 \ldots s_t$

$S_3 = A_2 * B_{t\text{-}3} = \; s_1 s_2 * s_4 s_5 \ldots s_t$

.

$S_{t\text{-}1} = A_{t\text{-}2} B_1 = \; s_1 s_2 \ldots s_{t-2} * s_t$

$S_t = A_{t\text{-}1} = \; s_1 s_2 \ldots s_{t-1}$

圖 2 加速演算法說明二

從圖中我們可以看出，計算 $S_1, S_2, \ldots, S_t$ 是利用圖 1 當中所示的 $2(t\text{-}1)$ 個值來產生的，所以我們可以利用這樣的方式來做加速運算的動作，利用這樣子的概念，我們可以利用計算乘法的方法來替代原本需要計算乘法反元素的方法。而由於我們的整批式驗證方法使用小指數測試方法，所以我們的整批式驗證架構的安全度可以達到最少 2^{-l}。關於我們所提出的方法的安全度分析以及效能分析，我們將會在第四章當中做一個詳細完整的分析。

接著，我們要證明我們所提出的整批式驗證方法與 Bellare 等人所提出的整批式驗證方法是一樣的，並且擁有相同的安全度。

證明：

$$\left(\prod_{i=1}^{t} r_i^{b_i}\right)^{S} \bmod p = \left(g^{\sum_{i=1}^{t} m_i S_i b_i \bmod q} y^{\sum_{i=1}^{t} r_i S_i b_i \bmod q}\right) \bmod p$$

$$\Rightarrow \left(\prod_{i=1}^{t} r_i^{b_i}\right) \bmod p = \left(g^{\sum_{i=1}^{t} m_i S_i b_i \bmod q} y^{\sum_{i=1}^{t} r_i S_i b_i \bmod q}\right)^{S^{-1}} \bmod p$$

$$\Rightarrow \left(\prod_{i=1}^{t} r_i^{b_i}\right) \bmod p = \left(g^{\sum_{i=1}^{t} m_i S_i S^{-1} b_i \bmod q} y^{\sum_{i=1}^{t} r_i S_i S^{-1} b_i \bmod q}\right) \bmod p$$

$$\Rightarrow \left(\prod_{i=1}^{t} r_i^{b_i}\right) \bmod p = \left(g^{\sum_{i=1}^{t} m_i s_i^{-1} b_i \bmod q} y^{\sum_{i=1}^{t} r_i s_i^{-1} b_i \bmod q}\right) \bmod p$$

由以上證明，可以得知我們所提出來的架構，基本上與 Bellare 等人所提出的方法是相同的，但是我們的方法經過修改後，可以將原本的乘法反元素運算取代成乘法運算，而使得效能得以提升。

第二節 效能分析

首先先介紹一下能夠快速計算 S_i 的演算法，接下來的效能分析也會依照這個演算法來分析。

Algorithm1 (Speed-up computation of S_j)
INPUT: q, s[1,t] // q is a modulus, s is an signature array of (s_1, s_2, …, s_t) OUTPUT: S[1,t] // S is an array of (S_1, S_2, …, S_t) Step: 1. define array temp_1[1, t-1] and array temp_2[1, t-1]; 2. temp_1[1] = s[1], temp_2[1] = s[t]; 3. **for** i = 2 to t-1 3.1 temp_1[i] = s[i] * temp_1[i-1] mod q; 3.2 temp_2[i] = s[t-i-1] * temp_2[i-1] mod q; 4. S[1] = temp_2[t-1], S[t] = temp_1[t-1]; 5. **for** j = 2 to t-1 5.1 S[j] = temp_1[j-1] * temp_2[t-j] mod q; 6. return S[1,t];

接著所介紹的第二個演算法能夠快速的計算 $\prod_{i=1}^{t} r_i^{b_i} \bmod p$ 。

Algorithm2 (Square-and-Multiply)
INPUT: $r_1,r_2,\ldots,r_t,b_1,b_2,\ldots,b_t$ //a_i's and b_i's are binary strings and every string's //length are l OUTPUT: result // the result of $\prod_{i=1}^{t} r_i^{b_i} \bmod p$ Step: 1. result = 1; 2. **for** $j = l$ downto 1 3. result = result^2; 4. **for** i = 1 to t **if** $b_i[j]$ = 1 **then** result = result $* r_i$; 5. return result;

在這節當中，我們要分析所提出的整批式驗證架構，並且與第二章第四節當中，Bellare 等人所提出的方法先做效能上面的比較，而我們比較的方式是比較這些方法當中的驗證方程式的效能，因為基本上兩個方法的簽章過程都是相同的，不同之處是在於驗證過程，所以我們會比較驗證方程式所會用到的各種運算之次數，如指數運算、乘法運算以及乘法反元素運算等等，以下就來分析一下兩個方法的效能。為了容易的對照我們的比較，我們再次分別將兩個方法的驗證方程式

列在下面。

Bellare 等人的驗證方程式：

$$\prod_{i=1}^{t} r_i^{b_i} \pmod p \stackrel{?}{=} (g^{\sum_{i=1}^{t} m_i s_i^{-1} b_i \bmod q} y^{\sum_{i=1}^{t} r_i s_i^{-1} b_i \bmod q}) \bmod p \tag{12}$$

本論文提出的驗證方程式：

$$\left(\prod_{i=1}^{t} r_i^{b_i}\right)^{S} \bmod p \stackrel{?}{=} (g^{\sum_{i=1}^{t} m_i S_i b_i \bmod q} y^{\sum_{i=1}^{t} r_i S_i b_i \bmod q}) \bmod p \tag{13}$$

首先分析指數運算的次數，在 Bellare 等人的方法當中需要 2 次的指數運算來計算 $(g^{\sum_{i=1}^{t} m_i s_i^{-1} b_i \bmod q} y^{\sum_{i=1}^{t} r_i s_i^{-1} b_i \bmod q}) \bmod p$ ，而本論文當中的方法則需要 3 次的指數運算，除了計算 $(g^{\sum_{i=1}^{t} m_i S_i b_i \bmod q} y^{\sum_{i=1}^{t} r_i S_i b_i \bmod q}) \bmod p$ 需要 2 次指數運算，在計算 $\left(\prod_{i=1}^{t} r_i^{b_i}\right)^{S}$ 還需要額外的 1 次指數運算。在乘法的次數，Bellare 等人的方法需要 $l + t(4 + l/2)$ 次的乘法，這麼多次的乘法當中，有 $l + tl/2$ 次的乘法是用來計算 $\prod_{i=1}^{t} r_i^{b_i} \pmod p$ 的值，而在這算這個值當中我們有運用到 square-and-multiply [13] 的演算法，4t 次的乘法則是用來計算 $\sum_{i=1}^{t} m_i s_i^{-1} b_i \bmod q$ 以及 $\sum_{i=1}^{t} r_i s_i^{-1} b_i \bmod q$ ，本論文所提出的方法則總共需要用到 $l + t(7 + l/2) - 5$ 次的乘法，其中計算 $\prod_{i=1}^{t} r_i^{b_i}$ 需要 $l + tl/2$ 次，計算 S_i 則需要 3(t-2) 次的乘法，以及一次的乘法來計算 S，最後我們來比較一下兩個方法當中所需要計算的乘法反元素次數，Bellare 等人的方法需要計算 t 次的乘法反元素計算，而本論文所提出的方法則不需要計算任何的乘法反元素。以下表 1 為出兩方法之間各種運算所需要的次數。

表 1 Bellare 等人與本論文的方法運算效能比較

(l 為 b_i 的位元數，t 為簽章的個數)

運算 / 方法	指數運算	模數乘法運算	乘法反元素運算
Bellare 等人的方法	2	$l+t(4+l/2)$	t
本論文提出的方法	3	$l+t(7+l/2)-5$	0

我們除了針對本論文提出的方法做分析與 Bellare 等人提出的方法做比較，也對兩個方法分別利用程式去測試其效能，我們所採用的函式庫是 Multiprecision Integer and Rational Arithmetic C/C++ Library (MIRACL) [16]，而所使用的環境是 MSVC 的編譯器，作業系統為微軟的 Windows XP 作業系統，x86 架構的主機，P4-2G 的 CPU 以及 2G 的記憶體，以下的表 2 以及圖 1 則是我們實驗的結果以及將我們實驗的結果畫成折線圖以方便我們觀察比較。

表 2 實驗結果

簽章的個數	Bellare 等人的方法	本論文的方法
5000	39765	5954

10000	83375	11828
15000	149157	17906
20000	186109	24156
25000	256015	30688
30000	309937	37296
35000	379953	44219
40000	434250	50860
45000	507219	58016
50000	575766	65266

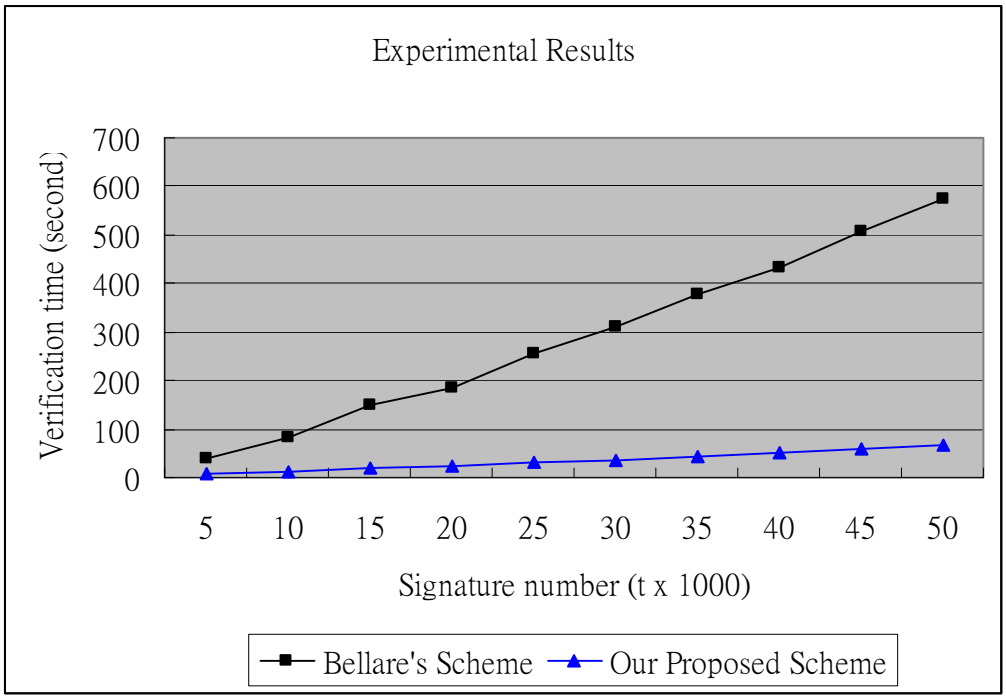

圖 3 實驗結果的比較

第三節 安全度分析

接下來我們分析一下本方法的安全度，由於我們的方法當中也有引用小指數測試 (*Small Exponents Test*)，所以我們將從亂數的大小來分析安全度，首先我們假設原本的 t 個正確的簽章 $s_1,s_2,\ldots,s_t$，可能會被偽造，而偽造過後的簽章為 $s_1',s_2',\ldots,s_t'$，我們假設在這群被偽造的簽章當中，我們不知道有哪些是被偽造的，但是第一個簽章一定是被偽造的，接下來假如攻擊者要使這些偽造的簽章能成功的通過驗證，這些偽造的簽章必須要滿足下列的聯立方程式。

$$\begin{cases} m_1 s_1^{-1} b_1 + m_2 s_2^{-1} b_2 + \cdots + m_t s_t^{-1} b_t = m_1 s_1'^{-1} b_1 + m_2 s_2'^{-1} b_2 + \cdots + m_t s_t'^{-1} b_t \\ r_1 s_1^{-1} b_1 + r_2 s_2^{-1} b_2 + \cdots + r_t s_t^{-1} b_t = r_1 s_1'^{-1} b_1 + r_2 s_2'^{-1} b_2 + \cdots + r_t s_t'^{-1} b_t \end{cases}$$

$$\Rightarrow$$

$$\begin{cases} m_1 b_1 (s_1^{-1} - s_1'^{-1}) = (m_2 s_2'^{-1} b_2 + m_3 s_3'^{-1} b_3 + \cdots + m_t s_t'^{-1} b_t) - (m_2 s_2^{-1} b_2 + m_3 s_3^{-1} b_3 + \cdots + m_t s_t^{-1} b_t) \\ r_1 b_1 (s_1^{-1} - s_1'^{-1}) = (r_2 s_2'^{-1} b_2 + r_3 s_3'^{-1} b_3 + \cdots + r_t s_t'^{-1} b_t) - (r_2 s_2^{-1} b_2 + r_3 s_3'^{-1} b_3 + \cdots + r_t s_t^{-1} b_t) \end{cases}$$

而我們可以看到假如要使這樣的聯立方程式成立，要依照 b_1 的值來決定，b_1 的大小是 l 位元，所以最多只 2^{-l} 的機率會通過驗證，所以我們可以證明我們的方法錯誤的機率為 2^{-l}。

第五章 高安全度的整批式測試方法

第一節 整批式子實體的測試方法

我們所提出的這個測試方法主要是利用編碼的概念，利用我們所設計的矩陣當作驗證過程的一個輔助工具，且我們所提出的方法適用於所有的整批式驗證架構，以下就將我們所提出的方法做詳細的介紹，現在我們假設有 n 個驗證實體所組成的整批式驗證實體，並且將之定義成 x =($(x_1,y_1),(x_2,y_2), \ldots, (x_n,y_n)$)，而 $n = 2^k$-1 且 k 為正整數，而驗證實體個別的驗證方式如同第三章第三節所提到的方法一樣，

$y_i = g^{x_i} \bmod p$ ，當 $i = 1,2, \ldots,n$。

接著我們定義 H 為一個 k 列 n 欄的檢驗矩陣，定義如下

$$H = \begin{bmatrix} h_{1,1} & h_{1,2} \cdots & h_{1,n} \\ \vdots & \ddots & \vdots \\ h_{k,1} & \cdots & h_{k,n} \end{bmatrix}, h_{i,j} \in \{0,1\}$$ ，當 $i = 1,2,\ldots,k$，$j = 1,2,\ldots,n$，$h_{i,j}$ 表示一個二進位的值，並且這個檢查矩陣的每一欄都是不相同的，但是不包含全部為”0”的欄。

接著我們定義 GT(x, n)為一般的整批式驗證方法，包含所有的整批式驗證方法，x 為整批簽章的實體，n 為此次簽章驗證的最大個數，GT(x, n)定義如下：

假如 $\prod_{i=1}^{n} y_i \equiv g^{\sum_{i=1}^{n} x_i \bmod q} \bmod p$ 成立，則 GT(x, n) = 1，否則 GT(x, n) = 0。

接下來就介紹一下我們所提出的整批式驗證測試方法，以下為整批式驗證的步驟：

步驟一：

從整批簽章實體 x 當中建立 k 個整批簽章的子實體，舉例如下。

$$x_i = \{(m_j, s_j) \mid h_{i,j} = 1\}$$

x_i 是一個從整批簽章實體 x 的簽章當中挑選出來的整批簽章的子實體，其挑選方式是依照 $h_{i,j}$ 的值來決定是否要把簽章挑選出來，i=1,2,…,k，j=1,2,…,n，當 $h_{i,j}$ 的值為 1 時，我們就將第 j 個簽章挑選出來並且成為 x_i 的一部分，當 $h_{i,j}$ 的值為 0 時則不挑選，最後依照這樣的規則來完成 k 個整批簽章實體的建立。

步驟二：

接下來利用 $\mathrm{GT}(x_i, 2^{k-1})=\sigma_i$ 來對個別的整批簽章的子實體做驗證的動作，假如驗證通過，則 σ_i=0，假如驗證不通過則 σ_i=1，當我們驗證到第一個 σ_i=1 的情況出現時我們就停止驗證的動作，並且知道這次的整批式驗證當中有錯誤的簽章出現，假如 σ_i 全部驗完，並且全部為零，也就是說沒有錯誤的情況發生，則我們可以說這次的整批式驗證當中沒有錯誤發生。

當使用了這樣子的檢測方法，我們可以不需使用小指數測試(*Small Exponents Test*)來增加整批式驗證的安全度。

舉例：

以下舉例如何執行本論文所提出的測試方法，首先我們假設有七個驗證實體需要驗證，分別是$(x_1, y_1), (x_2, y_2),\ldots, (x_7, y_7)$，接著我們使用一個檢查矩陣如下列所示

$$\begin{bmatrix} 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 1 & 0 & 1 \end{bmatrix}$$

接著我們依序對下列三個整批驗證的子實體作整批式驗證，

分別為

$x_1 = \{(x_4, y_4), (x_5, y_5), (x_6, y_6), (x_7, y_7)\}$

$x_2 = \{(x_2, y_2), (x_3, y_3), (x_6, y_6), (x_7, y_7)\}$

$x_3 = \{(x_1, y_1), (x_3, y_3), (x_5, y_5), (x_7, y_7)\}$

以及做三次測試，GT(x_1,4)，GT(x_2,4)，GT(x_3,4)，只要做到有一個整批式簽章子實體未驗證通過，就停止。本論文所提出的方法與小指數測試方法的不同在於小指數測試方法是機率式的測試方法，也就是說測試會有一定的錯誤機率；而我們所提出的測試方法是確定性的測試方法，能夠確定整批式驗證是否正確。

第二節 效能分析

首先我們分析一下這個測試方法的效能，一個 k 列 t 欄的最佳化的檢查矩陣應該是符合 $n = 2^k$-1 的條件，所以我們推導出 $k = \left| \log_2^{n-1} \right|$ 的條件，而依照檢查矩陣的每一列所選出來並準備檢驗的簽章有 2^{k-1} 個，所以我們以第四章所提出來的方法作為例子來分析其效能及安全度，當一個像下列這樣的整批式驗證方程式：

$$\prod_{i=1}^{n} y_i \equiv g^{\sum_{i=1}^{n} x_i \bmod q} \bmod p \qquad (14)$$

總共需要 k 次指數運算，大約是 $O(\log n)$ 次指數運算，接著我們分析乘法的次數，在計算乘法的個數前，我們需要知道每次的整批式子實體的簽章個數，以利我們做分析，而每個整批式子實體的簽章個數大約為 2^{k-1} 個，接著我們分析方程式(14)的左式，需要 $k \cdot 2^{k-1}$ 次的乘法，所以乘法的次數大約是 $O(k \cdot 2^{k-1}) = O(n \log n)$。接著我們將 Bellare 等人所提出的小指數測試方法與我們所提出的方法做效能上的比較，表 3 為兩方法所需要的各種運算次數的比較表。

表 3 整批式驗證測試方法之效能比較

運算 / 方法	乘法	指數
小指數測試方法	$l(n/2+1)$	1
整批式子實體測試方法	$n/2 \cdot \log n$	$\log n$

第三節 安全度分析

接下來分析一下其安全度，因為我們的檢測方法運用了檢查矩陣，當只要有一個以上的錯誤發生的時候，就應該被我們檢測出來。為了方便分析，我們把簽章錯誤的情況分成兩種，現在我們假設第一種狀況是所有簽章裡面有一個以上的錯誤簽章，且所有簽章不能通過整批式驗證，這種狀況的簽章在一般的整批式驗證方程式就可以被檢驗出來了。第二種狀況是所有簽章當中有一個以上的錯誤簽章，但是所有簽章卻可以通過驗證，像是這樣的狀況假如以 DSA 整批式驗證方法來看，通常是個別的簽章是錯誤的，但是所有簽章的總值在計算過後是正確的，才會使得偽造的簽章也通過認證。我們現在假設所有簽章當中只有一個錯誤，並且會通過整批式驗證，但是這樣子的假設不成立，因為只有一個簽章發生錯誤，其他的簽章是正確的話，所有的簽章無法通過整批式驗證。假如所有的簽章當中，有兩個簽章發生錯誤，但是這兩個錯誤的簽章的計算總合和原本正確的簽章一樣，則所有的簽章仍然能通過整批式驗證，假如這樣的情況要成立的話必須要使得所有簽章子實體當中都包含這兩個錯誤的簽章，或者是都不包含，也就是說檢查矩陣要找到兩欄相同的位元字串，但是檢查矩陣經過設計後，當中的每一欄都是不相同的，找不到兩欄相同的，所以說這樣的情況不存在。推廣到三個以上錯誤簽章的情況，以此類推，找

不到三個以上相同的欄位，所以我們可以說，假如使用檢查矩陣一定可以檢測出發生錯誤的狀況。所以假如偽造簽章的攻擊者想要攻擊簽章的機率為 $(s_1 \times s_2 \times \ldots \times s_t)^{-1}$ 。

第六章 結論

在這篇論文當中，我們針對了所有有關整批式驗證的論文做了重點式的探討及研究，並且我們提出了整批式驗證架構的弱點以及如何修補整批式驗證方法。首先，整批式驗證的方法能在同時間驗證大量簽章的情況下，有效且快速的驗證簽章。但是當整批式驗證的方法在設計過程當中有瑕疵，會造成偽造的整批簽章也能通過驗證的情況發生，其中有些論文提出了解決方法，但是相對的必須使得整批式驗證的成本提高，且仍然有一定的機率會發生錯誤，也就是機率式的測試方法，所以我們提出了確定性的整批式驗證方法，能夠將偽造的簽章卻能通過整批式驗證的機率降為零。第二，在我們的研究當中，觀察到有些整批式驗證架構是使用 DSA 來做為整批式驗證的基礎，而 DSA 當中會用到乘法反元素運算，乘法反元素的運算是很耗費時間的，尤其是在整批式驗證過程更是需要計算大量的乘法反元素，這些計算需要龐大的除法運算，假如在整批式驗證方法當中計算乘法反元素會相當的耗時並且降低服務效能。所以本論文當中提出了可以將反元素運算替換成模數乘法的運算的架構，並且有很好的效能。最後，在我們參考的論文當中，也有做有關於在整批式驗證當中尋找偽造或錯誤簽章的方法，但是我們覺得在整批式驗證當中尋找錯誤簽章的成本較難以控制，且錯誤的簽章個數難以判定，這可以是未來研究及努

力的目標。未來對於整批式驗證的研究，在於尋找錯誤及偽造的簽章的方法上，以及整批式驗證的效能提升上，還是有值得研究的地方。

參考

[1] T. ElGamal, “A Public-Key Cryptosystem and a Signature Scheme based on Discrete Logarithms,” Proceedings of Advances in Cryptology – CRYPTO’ 84, Springer Verlag, 1985, pp. 10

[2] T. ElGamal, “A Public-Key Cryptosystem and a Signature Scheme based on Discrete Logarithms,” IEEE Transactions on Information Theory, Vol. IT-31, No. 4, 1985, pp. 469-472.

[3] C.P. Schnorr, “Efficient Signature Generation for Smart Cards,” Proceedings of Advances in Cryptology – CRYPTO’ 89, Springer Verlag, 1990, pp. 239-252.

[4] C.P. Schnorr, “Efficient Signature Generation for Smart Cards,” Journal of Cryptology, Vol. 4, No. 3, 1991, pp. 161-174.

[5] Proposed Federal Information Processing Standard for Digital Signature Standard, Federal Register 56 (169) (1991) 42980-42982.

[6] D. Naccache, D. M’Raihi, D. Rapheali, and S. Vandenay: “Can DSA be Improved: Complexity Trade-offs with the Digital Signature Standard,” Proceedings of Advances in Cryptology - EUROCRYPT ’94, LNCS 950, 1995, pp. 77-85.

[7] C.H. Lim, and P.J. Lee: “Security of Interactive DSA Batch

Verification," Electronics Letters, 1994, Vol. 30, No. 19, pp. 1592-1593.

[8] L. Harn: "DSA Type Secure Interactive Batch Verification Protocols," Electronics Letters, 1995, Vol.31, No.4, pp.257-258.

[9] L. Harn: "Batch Verifying Multiple DSA-type Digital Signatures," Electronics Letters, 1998, Vol.34, No.9, pp.870-871.

[10] D. Knuth: "The Art of Computer Programming: Volume 2, Semi-numerical Algorithms," 2nd edition, Addison-Wesley, 1981.

[11] Stuart F. Oberman, Michael J. Flynn: "Division Algorithms and Implementations," IEEE Transactions on Computers, 1997, Vol.46, No.8, pp.833-854.

[12] C. Boyd, C. Pavlovski: "Attacking and Repairing Batch Verification Schemes," Proceedings of Advances in Cryptology - ASIACRYPT 2000, LNCS 1976, 2000, pp. 58-71.

[13] M. Bellare, J.A. Garay, T. Rabin: "Fast Batch Verification for Modular Exponentiation and Digital Signatures," Proceedings of Advances in Cryptology - EUROCRYPT '98, LNCS 1403, 1998, pp.236-250.

[14] Z. Shao: "Batch Verifying Multiple DSA-type Digital Signatures,"

Computer Networks, 2001, 37, pp.383-389.

[15] M.S. Hwang, C.C. Lee, Y.L. Tang: “Two Simple Batch Verifying Multiple Digital Signatures,” Proceedings of International Conference on Information and Communications Security, ICICS 2001, LNCS 2229, 2001, pp.233-237.

[16] Multiprecision Integer and Rational Arithmetic C/C++ Library (MIRACL) from http://indigo.ie/~mscott/

[17] S. Yen, C.Laih: “Improved Digital Signature Suitable for Batch Verification,” IEEE Transactions on Computers, Vol. 44, No. 7, pp. 957-959, July, 1995.

[18] R. Rivest, A. Shamir and L. Adleman: “A Method for Obtaining Digital Signatures and Public Key Cryptosystems,” Comm. ACM, Vol. 21, No. 2, 1978.

[19] B. Schneier: “Applied Cryptography,” 2nd edition, Wiley, 1996.

附錄 整批式驗證程式原始碼

載入金鑰的程式碼

GenKey.h

```
#ifndef GENKEY_H
#define GENKEY_H

#include "big.h"

using namespace std;

class GenKey {
public:
	GenKey(miracl* mip);
	GenKey(Big p, Big q, Big g, miracl* mip);
	void getParameters(Big& p, Big& q, Big& g, Big& pri, Big& pub);
	//pri is the private key and pub is the public key
	void genK(Big* K, int size);
	void genK(const char* file, int size);
	~GenKey();
private:
	int size;
	Big p, q, g, x, y;
};

#endif
```

GenKey.cpp

```
#include <iostream>
#include <fstream>
#include "GenKey.h"

//--------------------------------------------------------------------------------
GenKey::GenKey (miracl* mip) {
	mip->IOBASE = 16;
	ifstream common("common.dss");		/* construct file I/O streams
```

```
*/
      ifstream prikey("private.dss");
      ifstream pubkey("public.dss");
      common >> size;
      common >> p >> q >> g;
      mip->IOBASE = 10;
      prikey >> x;
      pubkey >> y;
      Big t_y=pow(g,q,p);
   if (t_y!=1)      {
            cout << "Problem - generator g is not of order q" << endl;
       exit(0);
   }
}
//--------------------------------------------------------------------------------
GenKey::GenKey (Big p, Big q, Big g, miracl* mip) {
      this->p = p;
      this->q = q;
      this->g = g;
      Big y=pow(g,q,p);
   if (y!=1)   {
            cout << "Problem - generator g is not of order q" << endl;
       exit(0);
   }
}
//--------------------------------------------------------------------------------
GenKey::~GenKey () {
}
//--------------------------------------------------------------------------------
void GenKey::getParameters (Big& p, Big& q, Big& g, Big& pri, Big&
pub) {
      pri = this->x;
      pub = this->y;

      p = this->p;
      q = this->q;
      g = this->g;
```

```
}
//--------------------------------------------------------------------------------
void GenKey::genK(Big* K, int size) {
      long seed;
      cout << "Enter 9 digit random number seed (Generate vector K) = ";
      cin >> seed;
   irand(seed);

      for (int i = 0; i < size ; i++) {
            *(K + i) = rand(this->q);
      }

      cout<< "finished!!" << endl;

}
//--------------------------------------------------------------------------------
void GenKey::genK(const char* file, int size) {
      long seed;
      cout << "Enter 9 digit random number seed (Generate vector K) = ";
      cin >> seed;
   irand(seed);

      ofstream file_k(file);
      for (int i = 0; i < size ; i++) {
            file_k << rand(this->q) << endl;
      }

      cout<< "finished!!" << endl;

}
//--------------------------------------------------------------------------------
```

Extended Euclidean Algorithm 的程式碼

XGCD.h

```
#ifndef XGCD_H
#define XGCD_H
```

```
#include "big.h"

using namespace std;

class XGCD {
public:
	XGCD();
	Big xgcd(Big modular, Big b);
	~XGCD();
private:
};

#endif
```

XGCD.cpp

```
#include <iostream>
#include "XGCD.h"

//--------------------------------------------------------------------------------
XGCD::XGCD () {
}
//--------------------------------------------------------------------------------
Big XGCD::xgcd(Big modular, Big b) {
	Big a1 = 1, a2 = 0, a3 = modular;
	Big b1 = 0, b2 = 1, b3 = b;
	Big t1, t2, t3;
	Big Q;

	while (true) {
		if (a3 == 0) return a3;
		if (b3 == 1) {
			return (b2 + modular) % modular;
		}
		Q = a3/b3;
		t1 = a1 - Q*b1;
		t2 = a2 - Q*b2;
```

```
            t3 = a3 - Q*b3;
            a1 = b1;
            a2 = b2;
            a3 = b3;
            b1 = t1;
            b2 = t2;
            b3 = t3;
        }
}
//---------------------------------------------------------------------------
```

數位簽章程式碼

DSS.h

```
#ifndef DSS_H
#define DSS_H

#include "big.h"

using namespace std;

class DSS {
public:
    DSS(Big p, Big q, Big g, Big pri, Big pub);
    DSS();
    void getSignatures(const char* file_r, const char* file_s, const char*
file_m,
    const char* file_k, int size);   //pri is the private key and pub is the
public key
    ~DSS();
private:
    int size;
    Big p, q, g, y, x;
};

#endif
```

DSS.cpp

```
#include <iostream>
#include <fstream>
#include "DSS.h"

//-------------------------------------------------------------------------------------
DSS::DSS () {
      ifstream common("common.dss");        /* construct file I/O streams
*/
      common >> size;
      common >> p >> q >> g;
      ifstream prikey("private.dss");
      prikey >> x;
      ifstream pubkey("public.dss");
      pubkey >> y;
}
//-------------------------------------------------------------------------------------
DSS::DSS (Big p, Big q, Big g, Big pri, Big pub) {
      this->p = p;
      this->q = q;
      this->g = g;
      Big y=pow(g,q,p);
   if (y!=1)   {
            cout << "Problem - generator g is not of order q" << endl;
       exit(0);
   }
      this->x = pri;
      this->y = pub;
}
//-------------------------------------------------------------------------------------
DSS::~DSS () {
}
//-------------------------------------------------------------------------------------
void DSS::getSignatures (const char* file_r, const char* file_s, const
char* file_m,
                         const char* file_k, int size) {
      Big m;
```

```
    Big k;
    ifstream vec_m(file_m);
    ifstream vec_k(file_k);
    ofstream vec_r(file_r);
    ofstream vec_s(file_s);

    cout << "Start generating signatures!!" << endl;

    int unit = size/100;
    int index = 0;
    int count = 0;
    Big temp_r, temp_s;

    for (int i = 0; i < size ; i++ ) {

        if (count == unit) {
            cout << ++index <<"%"<<endl;
            count = 0;
        }

        vec_m >> m;
        vec_k >> k;

        temp_r = pow(g, k, p);
        vec_r << temp_r << endl;
        temp_s = ( ((temp_r*k - m*x) % q) + q) % q;
        vec_s << temp_s << endl;

        count++;
    }


}
//--------------------------------------------------------------------------------
```

整批式驗證程式碼

DSV.h

```
#ifndef DSV_H
#define DSV_H

#include "big.h"

using namespace std;

class DSV {
public:
	DSV(Big p, Big q, Big g, Big pub, miracl* mip);
	DSV();
	//pri is the private key and pub is the public key
	bool verifySignaturesByNormal(Big* R, Big* S, Big* M,Big* B,int
size);
	//pri is the private key and pub is the public key
	bool verifySignaturesByBatch(Big* R, Big* S, Big* M,Big* B,int
size);

	Big* getInversesByNormal(Big* r, int size);
	Big* getInversesByBatch(Big* r, int size);
	Big xgcd(Big b, Big modular);
	Big fast_mult(Big* a, Big* b, int num, int bit_len);
	~DSV();
private:
	int size;
	Big p, q, g, y;
	miracl* mip;
};

#endif
```

DSV.cpp

```
#include <iostream>
#include <fstream>
#include "DSV.h"
```

```
//--------------------------------------------------------------------------------
DSV::DSV () {
      ifstream common("common.dss");        /* construct file I/O streams
*/
      common >> size;
      common >> p >> q >> g;

      ifstream pubkey("public.dss");
      pubkey >> y;
}
//--------------------------------------------------------------------------------
DSV::DSV (Big p, Big q, Big g, Big pub, miracl* mip) {
      this->p = p;
      this->q = q;
      this->g = g;
      Big t_y=pow(g,q,p);

   if (t_y!=1)       {
            cout << "Problem - generator g is not of order q" << endl;
       exit(0);
   }

      this->y = pub;
      this->mip = mip;
}
//--------------------------------------------------------------------------------
DSV::~DSV () {
}
//--------------------------------------------------------------------------------
bool DSV::verifySignaturesByNormal (Big* R, Big* S, Big* M,Big* B,
                                                int size) {

      int unit = size/100;
      int index = 0;
      int count = 0;
      Big left_val = 1;
      Big right_val = 0;
      Big g_exp = 0, y_exp = 0;
```

```
    Big temp_rp, temp_m, temp_s;
    Big* RP = new Big[size];

    RP = this->getInversesByNormal(R, size);

    left_val = fast_mult(R, B, size, 60);

    for (int i = 0; i < size ; i++ ) {

        g_exp = ( g_exp + modmult(modmult(S[i], RP[i], q),B[i], q) ) %
        q;
        y_exp = ( y_exp + modmult(modmult(M[i], RP[i], q),B[i], q) ) %
        q;
        count++;
    }

    right_val = modmult(pow(g,    g_exp, p), pow(y, y_exp, p), p);

    if (left_val == right_val) {
        cout << "All signatures are valid!!" << endl;
        return true;
    } else {
        return false;
    }
}
//-------------------------------------------------------------------------------
bool DSV::verifySignaturesByBatch (Big* R, Big* S, Big* M,Big* B,int
size) {

    int unit = size/100;
    int index = 0;
    int count = 0;
    Big left_val = 1;
    Big right_val = 0;
    Big g_exp = 0, y_exp = 0;
    Big temp_rp, temp_m, temp_s;
    Big* RP = new Big[size];
```

```
    RP = this->getInversesByBatch(R, size);

    left_val = fast_mult(R, B, size, 60);
    left_val = pow(left_val, RP[size], p);

    for (int i = 0; i < size ; i++ ) {

        g_exp = ( g_exp + modmult(modmult(S[i], RP[i], q),B[i], q) ) %
        q;
        y_exp = ( y_exp + modmult(modmult(M[i], RP[i], q),B[i], q) ) %
        q;

        count++;
    }

    right_val = modmult(pow(g,    g_exp, p), pow(y, y_exp, p), p);

    if (left_val == right_val) {
        cout << "All signatures are valid!!" << endl;
        return true;
    } else {
        cout << "There are at least one signature invalid!!" << endl;
        return false;
    }

}
//-------------------------------------------------------------------------------------
Big* DSV::getInversesByNormal(Big* r, int size) {
    Big* rp = new Big[size];
    for (int i = 0; i < size; i++) {
        rp[i] = xgcd(r[i], q);
    }
    return rp;
}
//-------------------------------------------------------------------------------------
Big* DSV::getInversesByBatch(Big* r, int size) {
    Big* temp_1 = new Big[size-1];
    Big* temp_2 = new Big[size-1];
```

```
    Big det_r;
    Big* rp = new Big[size+1];

    for (int j = 0; j < size-1; j++) {
        if (j==0) {
            temp_1[j] = r[0];
            temp_2[j] = r[size-1];
        } else {
            temp_1[j] = (r[j] * temp_1[j-1]) % q;
            temp_2[j] = (r[size-j-1] * temp_2[j-1]) % q;
        }
    }

    rp[size] =     (temp_1[0] * temp_2[size-2]) % q;

    for (int k = 0; k < size; k++) {
        if (k==0) {
            rp[0] = temp_2[size-2];
        } else {
            if (k==(size-1)) {
                rp[size-1] = temp_1[size-2];
            } else {
                rp[k] =      (temp_1[k-1] * temp_2[size-2-k]) % q;
            }
        }
    }

    return rp;
}
//---------------------------------------------------------------------------
Big DSV::xgcd(Big b, Big modular) {
    Big a1 = 1, a2 = 0, a3 = modular;
    Big b1 = 0, b2 = 1, b3 = b;
    Big t1, t2, t3;
    Big Q;

    while (true) {
        if (a3 == 0) return a3;
```

```
            if (b3 == 1) {
                return (b2 + modular) % modular;
            }
            Q = a3/b3;
            t1 = a1 - Q*b1;
            t2 = a2 - Q*b2;
            t3 = a3 - Q*b3;
            a1 = b1;
            a2 = b2;
            a3 = b3;
            b1 = t1;
            b2 = t2;
            b3 = t3;
        }
    }
    //-------------------------------------------------------------------------------
    Big DSV::fast_mult(Big* a, Big* b, int num, int bit_len) {
        char **c_b = new char*[num];

        mip->IOBASE = 2;
        for(int k=0; k<num; k++) {
            c_b[k] = new char[bit_len+2];
            c_b[k] << b[k];
        }
        mip->IOBASE = 10;

        //square multiply
        Big result("1");
        for(int i=0; i<bit_len; i++) {
            result = pow(result, 2, p);
            for (int j=0; j<num; j++) {
                if (c_b[j][i] == '1') {
                    result = modmult(result, a[j], p);
                }
            }
        }

        return result;
```

```
}
//--------------------------------------------------------------------------------
```

整批式驗證應用程式的程式碼

BatchVerificationApp.h

```
#ifndef BATCHVERIFICATION_H
#define BATCHVERIFICATION_H
#include "big.h"

using namespace std;

class BatchVerificationApp {
public:
    BatchVerificationApp();
    ~BatchVerificationApp();
private:
};

#endif
```

BatchVerificationApp.cpp

```
#include <iostream>
#include <fstream>
#include <time.h>
#include <windows.h>
#include "big.h"
#include "BatchVerificationApp.h"
#include "GenKey.h"
#include "DSS.h"
#include "DSV.h"

Miracl precision = 100;

//--------------------------------------------------------------------------------
```

```
BatchVerificationApp::BatchVerificationApp () {
}
//--------------------------------------------------------------------------------
BatchVerificationApp::~BatchVerificationApp () {
}
//--------------------------------------------------------------------------------

int main () {
    miracl *mip=&precision;
    Big p, q, g, private_key, public_key;

    GenKey *generator = new GenKey(mip);
    //get the parameters that batch verification needs
    generator->getParameters(p, q, g, private_key, public_key);
    mip->IOBASE = 10;
    cout<<"private key: "<<private_key<<endl;
    cout<<"public key: "<<public_key<<endl;
    cout<<"p: " << p << endl;
    cout<<"q: " << q << endl;
    cout<<"g: " << g << endl;

    int size = 100;
    cout << endl << "Enter the signatures size you want: ";
    cin >> size;

    ifstream M("M.dss");
    ifstream R("R.dss");
    ifstream S("S.dss");
    ifstream B("random.dss");
    Big* m = new Big[size];
    Big* r = new Big[size];
    Big* s = new Big[size];
    Big* b = new Big[size];

    //prepare parameters for DSA batch
    for (int i = 0; i < size ; i++) {
        M >> m[i];
        R >> r[i];
```

```
        S >> s[i];
    }

    mip->IOBASE = 2;
    for (int j=0; j<size; j++) {
        B >> b[j];
    }
    mip->IOBASE = 10;

    //new DSA batch verification class
    DSV *dsv = new DSV(p, q, g, public_key, mip);

    cout << "start batch!!" << endl;
    SYSTEMTIME* start1 = new SYSTEMTIME();
    GetSystemTime(start1);
    dsv->verifySignaturesByBatch(r, s, m, b, size);

    //Big* r1 = dsv->getInversesByBatch(r, size);
    SYSTEMTIME* end1 = new SYSTEMTIME();
    GetSystemTime(end1);

    long total_millis1 = (end1->wHour - start1->wHour) * 3600000;
    total_millis1 += (end1->wMinute - start1->wMinute) * 60000;
    total_millis1 += (end1->wSecond - start1->wSecond) * 1000;
    total_millis1 += (end1->wMilliseconds - start1->wMilliseconds);
    cout << "finish batch!!" << endl;
    cout << "batch: " << total_millis1 << endl;

    cout << "start normal!!" << endl;
    dsv->verifySignaturesByNormal(r, s, m, b, size);
    //Big* r2 = dsv->getInversesByNormal(r, size);
    SYSTEMTIME *end2 = new SYSTEMTIME();
    GetSystemTime(end2);
    long total_millis2 = (end2->wHour - end1->wHour) * 3600000;
    total_millis2 += (end2->wMinute - end1->wMinute) * 60000;
    total_millis2 += (end2->wSecond - end1->wSecond) * 1000;
    total_millis2 += (end2->wMilliseconds - end1->wMilliseconds);
    cout << "end normal!!" << endl;
```

```
    cout << "Normal: " << total_millis2 << endl;

    return 0;
}
```