植基於 Java Card 與 HMAC-SHA-2 的檔案完整性稽核軟體設計與實現

張惟淙
高雄師範大學資訊教育所
bug@icemail.nknu.edu.tw

楊中皇
高雄師範大學資訊教育所
chyang@nknucc.nknu.edu.tw

摘要

檔案稽核的主要功用在於檢驗檔案是否遭到變更而失其原有的完整性，其在資訊安全管理上是不可或缺的重要工作之一。

一般檔案稽核軟體僅使用雜湊函數演算法計算檔案的訊息摘要，以作為確認完整性的依據。而本研究所開發的軟體，主要以 HMAC 來取代訊息摘要，成為檔案完整性稽核應用的改進技術。HMAC 是將金鑰結合檔案內容以雜湊函數演算法計算而得。本研究參考 HMAC 相關標準及 GnuPG 的 SHA-2 原始碼實作 HMAC 模組，並以 Borland C++ Builder 6 開發視窗介面軟體；此外，基於便利性及金鑰安全保存的考量，將 HMAC 金鑰透過 Java Card 存取，使用者必須持有卡片才能執行檔案稽核工作。

關鍵詞：系統稽核、雜湊函數、HMAC、SHA-2、Java Card

一、前言

就確認檔案完整性的實質作用而言，檔案稽核最重要的功能就是偵測出系統中遭到變更的重要檔案[6]。無論是單一主機或網路系統，絕對有某些重要的系統檔案不得隨意變更，例如系統管理權限密碼檔、伺服器組態檔、入侵偵測系統或防火牆的設定檔…等。諸如這些檔案，如果遭到惡意變更就會影響系統運作，甚至危害系統的整體安全。

由於沒有任何防範機制能完全保證系統檔案的完整性不會因受到入侵攻擊而遭破壞。因此在建置網路安全系統時，通常會考慮使用檔案稽核軟體為重要的系統檔案實施例行性的完整性檢查工作，以確保系統檔案的異常變動能夠及早發現並儘速採取補救措施，使系統在最短時間內恢復正常。

西元 1992 年時，美國普渡大學 Spafford 博士及其學生 Gene Kim 發表了著名的檔案稽核軟體 Tripwire[1]。原先 Tripwire 是開放原始碼軟體，但之後成為商業軟體。其他類似的軟體，還有承自 Tripwire 開放原始碼版本而發展的 AIDE(Advanced Intrusion Detection Environment)[4]，以及跨平台的視窗軟體 wxChecksums[18]…等。這些軟體共同的主要功能就是以雜湊函數演算法計算檔案的訊息摘要，作為日後比對確認檔案完整性的依據。

本研究所開發的軟體，則是採用 HMAC 演算法作為計算檔案檢查碼的主要技術。HMAC 是以雜湊函數演算法，加上一把由亂數產生的金鑰，結合檔案內容計算而得[7]。不同金鑰將使得同一檔案產生出不同的 HMAC 雜湊值，因此就其意義而言，無論 HMAC-SHA-1 或 HMAC-MD5 均較單純使用 SHA-1 或 MD5 來得安全。然而由於 MD5 已為中國山東大學的王小雲教授等學者所破解[19]，美國國家標準技術研究院(NIST)亦宣佈將於 2010 年後不再繼續支持處理流程類似於 MD5 的 SHA-1[10]。因此我們選擇較新的 SHA-2 作為用以計算 HMAC 的雜湊函數演算法。

在本研究中以 Java Card 存取 HMAC 所用的金鑰。使用者只要合法持有金鑰卡片，便能進行檔案稽核及維護的安全管理工作。不僅使用時存取便利，同時亦較為安全。

二、文獻探討

2.1 HMAC 演算法

2.1.1 HMAC 的運算流程

依據 NIST 所制定的第 198 項聯邦處理標準(FIPS PUB 198)[13]規範之 HMAC 處理流程總共有十個步驟(如表一)。相關的參數說明如下：

B：每個處理區段的位元組數。
H：選用的雜湊函數演算法。
L：雜湊函數 H 輸出的雜湊值長度，單位為位元組。
K：原始使用的金鑰。
K_0：經過前置處理後的金鑰，其大小應等於 B 個位元組。
0xN：表示 16 進位值為 N 的字元，以 1 個位元組儲存。

ipad： B 個 16 進位值為 0x36 的字元組成的訊息。
opad： B 個 16 進位值為 0x5C 的字元組成的訊息。
t：輸出結果的 HMAC 碼長度。
text：未經任何處理的原始輸入訊息。
||：將欲處理的訊息接續。
⊕：執行 XOR 位元運算。

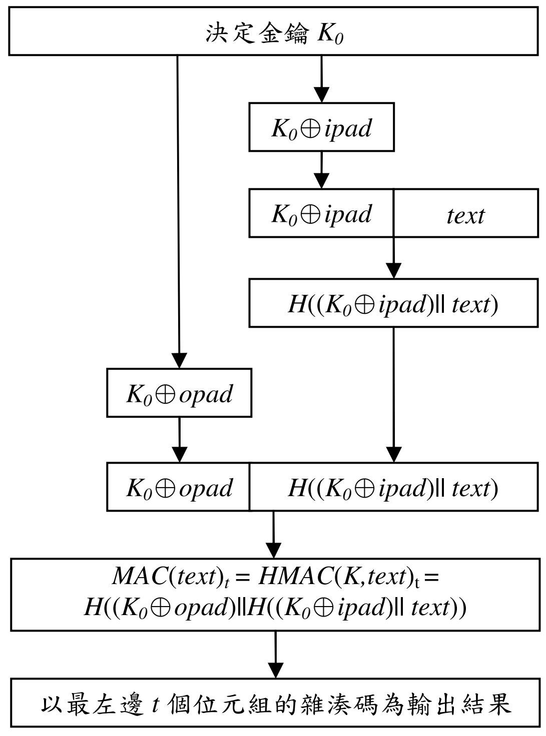
表一：HMAC 演算法運算流程

步驟	說明
1	以亂數產生 K，如果 K 的長度等於 B，設 K_0=K，直接跳到步驟 4。
2	如果 K 的長度大於 B，則對 K 作雜湊函數運算，再將其輸出結果附加 B-L 個 0x00 字元成為 K_0。
3	如果 K 的長度小於 B，則將 K 附加 B-K 個 0x00 字元成為 K_0。
4	將 K_0 與 *ipad* 作⊕運算，產生一串長度為 B 的訊息：$K_0 \oplus ipad$。
5	將步驟 4 所產生的結果接續原始輸入訊息成為：$(K_0 \oplus ipad) \| text$。
6	將步驟 5 的結果執行雜湊函數運算：$H((K_0 \oplus ipad) \| text)$。
7	將 K_0 與 *opad* 作⊕運算，產生另串長度為 B 的訊息：$K_0 \oplus opad$。
8	將步驟 6 的結果接續到步驟 7 的後面：$K_0 \oplus opad \| H((K_0 \oplus ipad) \| text)$。
9	將步驟 8 的結果執行雜湊函數運算：$H((K_0 \oplus opad) \| H((K_0 \oplus ipad) \| text))$。
10	依據步驟 9 的結果，輸出最左邊的 t 個位元組作為最終輸出的 HMAC 雜湊訊息確認碼。

以 HMAC 演算法計算訊息確認碼的計算方程式可以表示如下：

$$MAC(text)_t = HMAC(K,text)_t = H((K_0 \oplus opad) \| H((K_0 \oplus ipad) \| text)) \quad (1)$$

詳細的處理流程圖如圖一所示。最後的結果是以最左邊 t 個位元組的雜湊碼為輸出值，這亦是定義於 HMAC 標準。舉例來說，假設我們使用的處理區段為 32 位元組，而選擇的 HMAC 雜湊函數演算法為 SHA-256，在真正執行完所有運算後，結果的雜湊碼長度仍然與原設定的處理區段長度相同，均為 32 位元組，亦即 256 個位元。在相同的安全性之下，我們可以選擇僅輸出部分作為要使用的 HMAC 碼，例如輸出 20 個位元組即可。這樣做並不會失去原先選用的 SHA-256 演算法之安全性，而且又可以節省儲存空間。但在本研究中，我們開發的軟體，目前仍以原處理區段的長度作為 HMAC 碼輸出長度，以方便驗證計算的正確性。



圖一：HMAC 演算法運算流程圖

2.1.2 HMAC 的安全性

HMAC 的安全強度主要取決於所選用的雜湊函數演算法。實作 HMAC 時，可以將雜湊函數程式碼的部分視為單一模組，這意味著假如我們需要效率更高的演算法時，不用變更其他的處理流程，只要依需要置換雜湊函數原始碼即可。最重要的是，如果原先使用的雜湊函數原始碼已有安全上的疑慮，藉由此設計優點，輕易就可以改成更安全的雜湊函數演算法(例如將 MD5 或 SHA-1 置換為 SHA-2)。

此外，金鑰的使用亦能增加 HMAC 的安全強度。在 HMAC 的處理過程中，$K_0 \oplus ipad$ 及 $K_0 \oplus opad$ 這兩個步驟均會改變金鑰 K 中一半的位元值，而且改變的部分不同，而再將其導入雜湊函數的運算之後，就如同隨機產生另外兩把金鑰，因此大幅提升 HMAC 作為檔案訊息確認碼的安全性[17]。

2.2 SHA-2 雜湊函數演算法

2.2.1 SHA-2 簡介

雜湊函數演算法可用於產生訊息或檔案的指紋值(Fingerprint)。此類函數會將任意長度的訊息，運算出固定長度的雜湊數值，該數值一般稱為訊息摘要(Message Digest)，或雜湊值(Hash Value)[3]。雜湊函數演算法在密碼學的應用領域中相當廣泛，通常配合數位簽章使用[11]，以確認訊息在傳送過程中未遭竄改，藉此驗證訊息的完整性。

SHA(Secure Hash Algorithm)演算法是由 NIST 所發展出來，並在 1993 年成為第 180 項聯邦處理標準(FIPS PUB 180)。目前最新的修訂版是 FIPS PUB 180-2，其中新增包含 SHA-256、SHA-384 及 SHA-512 等三種雜湊演算法[12]。

2.2.2 SHA-2 演算法的前置處理

SHA-2 輸入的訊息長度不能超過 2^{64} 個位元，SHA-256 的處理區段為 512 個位元，SHA-384/512 則是 1024 個位元，在實際進行計算之前的前置處理有三個步驟，以 SHA-256 為例說明如下：

1. 訊息附加位元：這個步驟是必須的，就 SHA-256 而言，在訊息之後附加位元，使訊息的位元長度能在取 512 的同餘之後等於 448。附加的方式是先加 1 個 1，再用 0 補到所需的長度。之後再於訊息最後加上一段 64 個位元的資料，這段資料用以表示原始訊息的長度。經此步驟後，訊息長度就變成為 512 個位元的倍數。
2. 將附加位元後的訊息分解成數個區段：SHA-256 以 512 個位元為區段長度進行分解。
3. 設定初始的雜湊值：SHA-2 處理中的區段資料會放在一個訊息摘要緩衝區(MD buffer)內。而在前置處理時，首先會在該緩衝區中設定以 16 進位值表示長整數的 8 個初始值。

2.2.3 SHA-2 演算法的參數

$a, b, c, ..., h$：*MD* 緩衝區的 8 個字元變數。

$H^{(i)}$：第 i 個雜湊值，由 $a,b,c,...,h$ 所組成。$H^{(0)}$ 表示初始值，$H^{(N)}$ 表示訊息摘要結果。

$H_j^{(i)}$：第 i 個雜湊值的第 j 個字元。

K_t：第 t 個計算回合所用的常數。

k：在附加位元時增加的 0 值位元數。

l：欲計算之訊息 M 的長度。

m：一個訊息區段 $M^{(i)}$ 的位元長度。

M：前置處理後的訊息。

$M^{(i)}$：第 i 個訊息區段。

$M_j^{(i)}$：第 i 個訊息區段的第 j 個字元。

n：字元的旋轉位數或移位位數。

N：M 所分解的區段數。

T：雜湊計算過程中暫存的字元。

w：字元的位元數。SHA-256 的字元是 32 個位元，SHA-384/512 的字元則為 64 個位元。

W_t：根據目前輸入區段導出的字元。

2.2.4 SHA-2 演算法的字元處理函數

SHA-2 用到兩個字元處理函數，一是字元向右移位運算 *SHR*，另一則是字元向右旋轉運算 *ROTR*，如下：

$$SHR^n(x) = x >> n \tag{2}$$

字元 x 向右移 n 個位元

$$ROTR^n(x) = (x >> n) \vee (x << w-n) \tag{3}$$

字元 x 向右旋轉 n 個位元

2.2.5 SHA-2 演算法的邏輯函數

SHA-2 演算法共有六個邏輯函數，SHA-256 的如下：

$$Ch(x,y,z) = (x \wedge y) \oplus (\neg x \wedge z) \tag{4}$$

$$Maj(x,y,z) = (x \wedge y) \oplus (x \wedge z) \oplus (y \wedge z) \tag{5}$$

$$\textstyle\sum_0^{\{256\}}(x) = ROTR^2(x) \oplus ROTR^{13}(x) \oplus ROTR^{22}(x) \tag{6}$$

$$\textstyle\sum_1^{\{256\}}(x) = ROTR^6(x) \oplus ROTR^{11}(x) \oplus ROTR^{25}(x) \tag{7}$$

$$\sigma_0^{\{256\}}(x) = ROTR^7(x) \oplus ROTR^{18} \oplus SHR^3(x) \tag{8}$$

$$\sigma_1^{\{256\}}(x) = ROTR^{17}(x) \oplus ROTR^{19} \oplus SHR^{10}(x) \tag{9}$$

上列每一個函數的輸入變數 x, y, z 都是 32 位元的字元，處理完之後輸出另一個 32 位元的字元。SHA-384/512 的邏輯函數前兩個，與 SHA-256 的相同，亦即方程式(4)及(5)，只是字元的位元長度不同。在 SHA-384/512 運算中，x, y, z 變數的長度為 64 位元。SHA-384/512 的其他四個邏輯函數如下：

$$\textstyle\sum_0^{\{512\}}(x) = ROTR^{28}(x) \oplus ROTR^{34}(x) \oplus ROTR^{39}(x) \tag{10}$$

$$\textstyle\sum_1^{\{512\}}(x) = ROTR^{14}(x) \oplus ROTR^{18}(x) \oplus ROTR^{41}(x) \tag{11}$$

$$\sigma_0^{\{512\}}(x) = ROTR^1(x) \oplus ROTR^8 \oplus SHR^7(x) \tag{12}$$

$$\sigma_1^{\{512\}}(x) = ROTR^{19}(x) \oplus ROTR^{61} \oplus SHR^6(x) \tag{13}$$

2.2.6 SHA-2 演算法的常數

SHA-256 在每次的訊息區段計算過程中，會一直用到 64 個常數，其定義的來源是前 64 個質數的立方根小數部分的前 32 個位元；而 SHA-384/512 則會用到 80 個常數，其來源是前 80 個質數的立方根小數部分的前 64 個位元。

2.2.7 SHA-2 演算法的運算流程

在訊息經過前置處理完畢之後，會被分解成 N 個訊息區段，以 $M^{(1)}$、$M^{(2)}$、… 、$M^{(N)}$表示，SHA-256 的運算流程如表二。

表二：SHA-256 的運算流程

For i = 1 to N: //目前處理的區段以 i 表示
{
//根據第 i 個輸入區段導出 64 個 32 位元的字元，以 W_t 表示個別的字元

$$Wt = \begin{cases} M_t^i & 0 \le t \le 15 \\ \sigma_1^{\{256\}}(W_{t-2}) + W_{t-7} + \sigma_0^{\{256\}}(W_{t-15}) + W_{t-16} & 16 \le t \le 63 \end{cases}$$

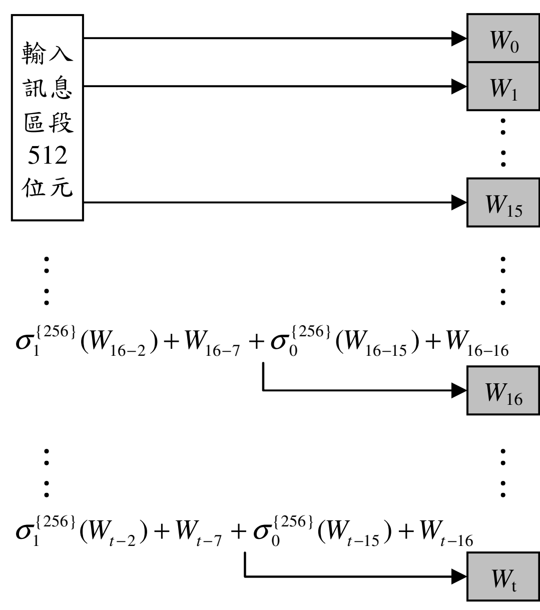
//設定前個訊息區段計算完的雜湊值到 MD buffer 中的 8 個字元變數

$a = H_0^{(i-1)}$　　$b = H_1^{(i-1)}$
$c = H_2^{(i-1)}$　　$d = H_3^{(i-1)}$
$e = H_4^{(i-1)}$　　$f = H_5^{(i-1)}$
$g = H_6^{(i-1)}$　　$h = H_7^{(i-1)}$

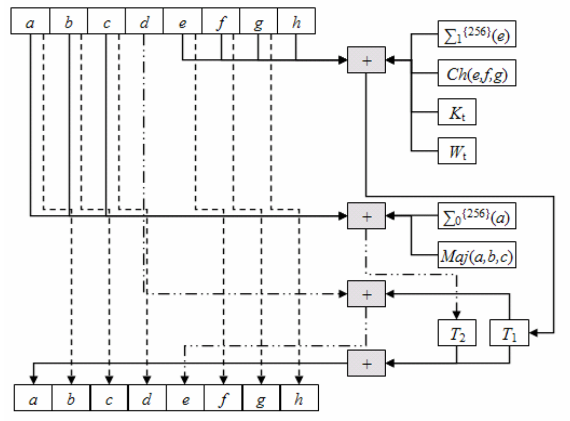
//每個區段要經過 64 回合的基本運算
For t = 0 to 63:
{
$T_1 = h + \sum_1^{\{256\}}(e) + Ch(e,f,g) + K_t^{\{256\}} + W_t$
$T_2 = \sum_0^{\{256\}}(a) + Maj(a,b,c)$
$h = g$
$g = f$
$f = e$
$e = d + T_1$
$d = c$
$c = b$
$b = a$
$a = T_1 + T_2$
}
//得出目前訊息區段的雜湊值結果
$H_0^{(i)} = a + H_0^{(i-1)}$　　$H_1^{(i)} = b + H_1^{(i-1)}$
$H_2^{(i)} = c + H_2^{(i-1)}$　　$H_3^{(i)} = d + H_3^{(i-1)}$
$H_4^{(i)} = e + H_4^{(i-1)}$　　$H_5^{(i)} = f + H_5^{(i-1)}$
$H_6^{(i)} = g + H_6^{(i-1)}$　　$H_7^{(i)} = g + H_7^{(i-1)}$
}

圖二是 SHA-256 計算處理一個訊息區段，並導出 64 個字元的方式。前 16 個字元是由輸入的 512 個位元訊息區段分解而成(512/32 = 16)。後續的字元必須經過邏輯函數運算與同餘加法計算而得。

每個訊息區段計算導出的 64 個字元，在訊息區段的基本運算中會用到，每個訊息區段會計算 64 個回合的基本運算，每次基本運算會各用到一個導出的字元以及事先定義好的常數。單一回合計算的方式請參考圖三。



圖二：由一個訊息區段導出 64 個字元



圖三：SHA-256 單一回合的基本運算流程

如果是 SHA-384/512 的話，處理一個訊息區段，是導出 80 個字元，每個字元是 64 位元，前 16 個字元是由輸入的 1024 個位元訊息區段分解而成（1024/64 = 16）。後續的第 17 個到第 80 個字元亦必須經過 SHA-384/512 的邏輯函數運算與同餘加法計算而得。

SHA-384 與 SHA-512 只除了初始的雜湊值有所不同，使用的常數、邏輯函數及運算流程均完全相同。而在最後的訊息摘要雜湊值輸出，SHA-384 是輸出雜湊值結果中的前 6 個 64 位元的字元，而 SHA-512 則是輸出全部 8 個 64 位元的字元。

2.3 Java Card

Java Card 是一種標準的智慧卡(Smart Card)，智慧卡在當今網路安全應用最重要的三個特點就是：確認性、保密性及便利性。其內部具有源於

Java 技術的 Java Card 虛擬機器(JCVM, Java Card Virtual Machine)及 Java Card 執行環境(JCRE, Java Card Runtime Environment)[5]。

此外，Java Card 尚提供一套具有物件導向程式設計特色的 API，它所包含的最重要部分就是與密碼學有關的 API，其中有 3DES、RSA、SHA-1、MD5 等密碼技術[15]。這使得 Java Card 本身可以在卡片內部進行資料的加解密及數位簽章的產生與驗章等安全性功能[2]。本研究所開發的軟體中用以存取 HMAC 金鑰的 Java Card 是 IBM JCOP20 卡，它完全符合 Visa OpenPlatform Card 架構及 Java Card API 2.1.1 版規範，其主要特色是對 RSA 公開金鑰演算法的支援[8]。

三、軟體實作

3.1 HMAC 模組實作

由於本研究所開發的檔案稽核軟體，乃是採用 HMAC 演算法所產生的 HMAC 雜湊訊息確認碼作為比對檔案完整性的檢查碼。因此必須先實作 HMAC 模組這個核心功能。

HMAC 中雜湊函數演算法的部分，我們參用 GnuPG 1.4.1 版[16]中 SHA-2 演算法的原始碼，並依 RFC2202 文件 [14]內所附的 HMAC-SHA-1 程式碼，共設計了六個 HMAC-SHA-2 模組。如表三：

表三：本研究實作的 HMAC-SHA-2 模組

模組名稱	功能說明
hmac_sha256	對輸入的訊息字串計算並輸出 HMAC-SHA-256 碼。
hmac_sha256_file	對選取的檔案計算並輸出 HMAC-SHA-256 碼。
hmac_sha384	對輸入的訊息字串計算並輸出 HMAC-SHA-384 碼。
hmac_sha384_file	對選取的檔案計算並輸出 HMAC-SHA-384 碼。
hmac_sha512	對輸入的訊息字串計算並輸出 HMAC-SHA-512 碼。
hmac_sha512_file	對選取的檔案計算並輸出 HMAC-SHA-512 碼。

為了測試所實作的 HMAC 模組是否能夠正確運算 HMAC 碼，必須以較有公信力的測試資料作為模組的測試樣本。然而，由於目前 RFC 文件中尚未正式制定有關 HMAC-SHA-2 標準規範。因此我們採用 RSA Security 公司的 M. Nystrom 先生所撰的 RFC 草稿” Identifiers and Test Vectors for HMAC-SHA-224, HMAC-SHA-256, HMAC-SHA -384, and HMAC-SHA-512”中[9]的測試樣本。該份草稿於 2005 年 6 月 27 日釋出，總共提供包含七個 HMAC 測試樣本，並各使用四種雜湊函數演算法，SHA-256/384/512 均囊括在內。經過對表三所列六個模組測試的結果，無論是輸入訊息字串，或以檔案作計算，均符合測試樣本的結果。圖四是我們參考 RFC2202 標準所設計的 HMAC-SHA-256 檔案處理模組。

```
int hmac_sha256_file
(
    char*    seckey,    /* secret key */
    int      kenlen,    /* length of the key in bytes */
    char*    fname,     /* file name */
    SHA256_CONTEXT *ictx,
    SHA256_CONTEXT *octx
)
{
    char    buffile[512] ;
    char    buf[SHA256_BLOCKSIZE] ;
    int     i, n ;
    FILE *fp ;

    if (kenlen > SHA256_BLOCKSIZE) {

       SHA256_CONTEXT   tctx ;

       sha256_init(&tctx) ;
       sha256_write(&tctx, seckey, kenlen) ;
       sha256_final(&tctx) ;

       //----key size
       for (i = 0 ; i < SHA256_BLOCKSIZE ; ++i)
            seckey[i] = tctx.buf[i] ;

       kenlen = SHA256_DIGESTSIZE ;
    }

    /**** Inner Digest ****/

    sha256_init(ictx) ;

    /* Pad the key for inner digest */
    for (i = 0 ; i < kenlen ; ++i) buf[i] = seckey[i] ^ 0x36 ;
    for (i = kenlen ; i < SHA256_BLOCKSIZE ; ++i) buf[i] = 0x36
```

圖四：hmac_sha256_file 模組(部分)

3.2 應用流程設計

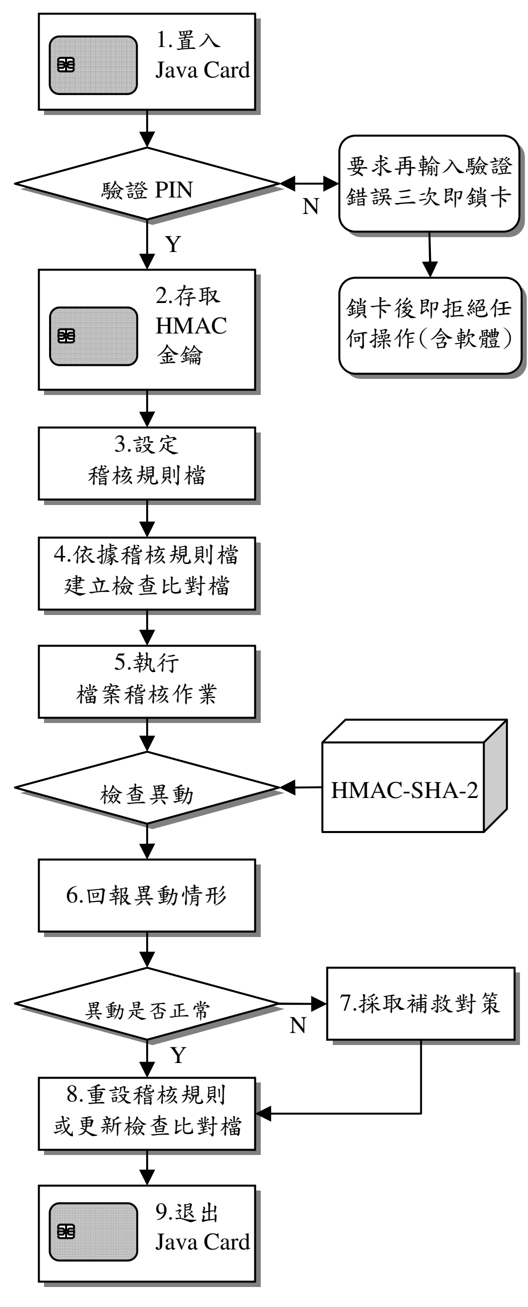
使用者必須先持有內建 HMAC 金鑰存取程式的 Java Card，才能使用這套軟體。本軟體的操作步驟說明如表四：

表四：檔案稽核的應用流程說明

步驟	說明
1	將內建有 HMAC 金鑰存取程式的 Java Card 置入讀卡機。
2	讀取卡片之後，會要求使用者輸入 PIN 碼驗證。驗證成功後，可用軟體產生金鑰存入卡片，若偵測出卡片內已存有金鑰，則提示使用者是否置換。請注意置換金鑰會影響已建立的檢查比對檔案。
3	無論是否變更金鑰，只要能存取金鑰，便可設定稽核規則檔。
4	依據稽核規則檔的設定，建立檢查比對檔以供日後檢驗檔案完整性之用。使用者可自行針對檔案的性質分類，建立多個稽核規則檔。
5	執行檔案稽核作業，通常這個步驟是例行性工作。

6	檢查如果發現檔案有異動，則回報異動情形。使用者依異動報告再檢查異動是否屬於正常變更。
7	檔案遭到異常變更，應立即採取具安全性的補救對策。相關作為必須有賴使用者的智慧，軟體本身並不判別何種變更屬於異常，亦不具備檔案修復功能。
8	檢查過後，使用者應重設稽核規則或更新檢查比對檔。
9	退出 Java Card，結束所有作業，並清除軟體執行時暫存金鑰內容的變數。

應用流程如圖五：

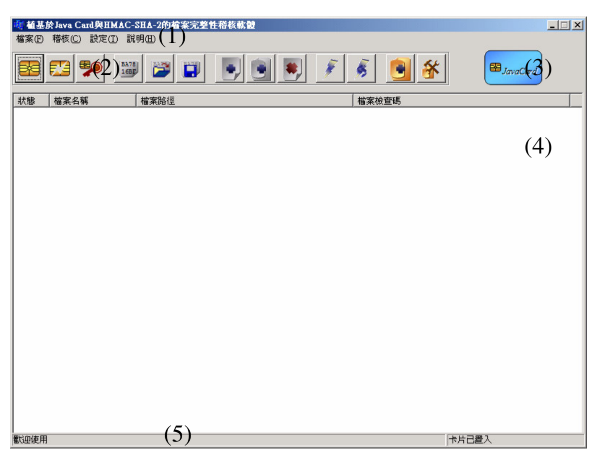


圖五：檔案稽核的應用流程圖

3.3 軟體使用簡介

3.3.1 操作介面說明

我們使用 C++ Builder 6 開發，它不僅能快速建立視窗介面，且能整合 C/C++原始碼。非常有助於應用開放原始碼開發視窗軟體。主畫面如圖六。



圖六：軟體主畫面

主畫面大致劃分以下五個區塊說明：

(1) 主選單：有檔案、稽核、設定及求助等四個子選單。

(2) 工具列：表五是各工具圖示及相關的功能說明。

表五：工具列的使用說明

圖示	說明
	置入 Java Card，與卡片連線。
	退出 Java Card，與卡片斷線。
	存入或取出 HMAC 金鑰。
BA78 16BF	建立檢查比對檔，預設使用 hmac_sha256_file 模組，副檔名為.sha2。
	開啟檢查比對檔。
	儲存目前的檔案檢查碼資料到現行的檢查比對檔。
	新增欲檢查比對的檔案。
	新增欲檢查比對的目錄。
	從檢查比對檔中刪除資料。
	就現行的檢查比對檔執行稽核作業，檢驗檔案是否具完整性。

	就現行的檢查比對檔重新執行稽核作業，可反覆執行。
	依稽核規則檔新增欲檢查比對的目錄或檔案。
	建立稽核規則檔，定義要檢查的目錄及檔案。

(3) 卡片狀態圖：若顯示彩色表示卡片目前連線使用中，若為黑白則尚未置入卡片。

(4) 稽核資料列表：列示使用者所開啟的檢查比對檔，執行稽核功能後，對檔案的檢查結果亦會顯示出來。

(5) 狀態列：前段顯示檔案稽核相關訊息，後段則是卡片使用的狀態。

3.3.2 HMAC 金鑰的存取

通過卡片的 PIN 驗證之後，接著就要開啟金鑰的存取功能，如圖七：

圖七：HMAC 金鑰存取功能畫面

畫面上會顯示卡片是否已存放金鑰，若是則可讀取金鑰以便執行稽核相關功能。針對已存放的金鑰可以予以置換或移除，但要特別注意置換及移除的動作會影響之前以該把金鑰建立的檢查比對檔無效。若尚未存放金鑰，則應點選新增按鈕以亂數產生一把金鑰，並存入卡片，再讀取使用。使用者亦可透過此功能畫面修改卡片中的 PIN 碼。

3.3.3 檔案稽核使用

初次使用時首先須建立一個新的檢查比對檔，使用者可以逐一選取要檢查的檔案或目錄新增到該檔中。日後就可以依據檢查比對檔執行稽核功能，確認檔案是否遭竄改或其他異常的變更。

稽核的結果區分四種情況，分別以圖示標示於每一列資料的第一個欄位，說明如下：

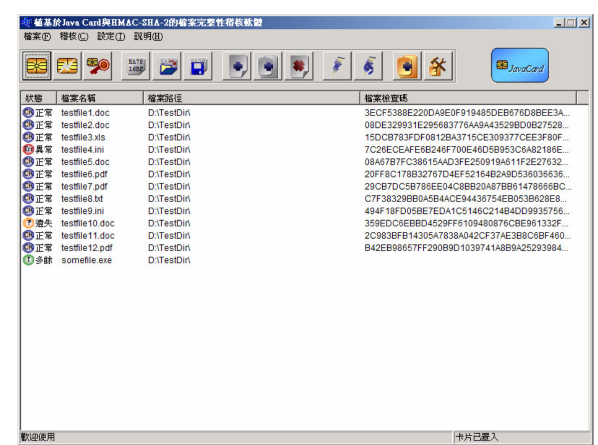
OK 正常：檢查的檔案雜湊碼與比對檔的相符，檔案並未遭到變更。

Err 異常：顯示檔案已遭變更，使用者必須自行檢查檔案的變更是否異常。

! 多餘：在檢查的目錄中發現多出於檢查比對檔內資料的檔案。

? 遺失：要檢查的檔案已遺失，應予檢查是否遭竊取或移除。

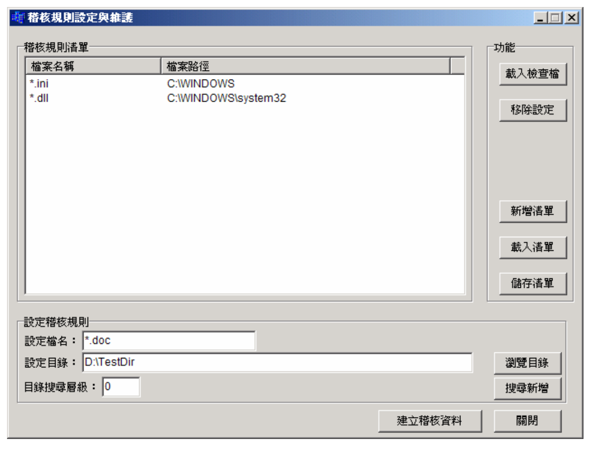
通常執行檔案稽核之後，畫面會如圖八所示：



圖八：執行稽核的結果

3.3.4 設定稽核規則檔

稽核規則檔可以允許使用者定義要搜尋哪些目錄的哪些檔案，搜尋的檔案名稱可以使用萬用字元符號"?"或"*"表示，並可將設定好的規則儲存成稽核規則清單檔（副檔名預設為*.s2c），以利日後便於執行例行性的稽核工作。使用者亦能對稽核規則檔修改維護，例如將多個檔案儲存成一個檔案。此項功能畫面如圖九。

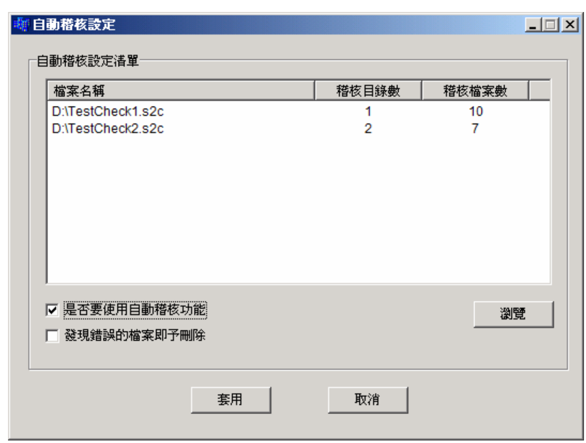


圖九：稽核規則設定與維護畫面

3.3.5 自動稽核功能

一般的稽核軟體安裝於系統中之後，通常會加入系統工作排程，以便自動執行例行性的檔案稽核檢查作業。然而由於我們設計的稽核軟體必須使用

Java Card 才能運行，因此在沒有卡片的情況之下，軟體本身並無法自動進行稽核作業。但為了讓使用者在應用上更為方便，我們亦設計自動稽核的功能，使用者可以設定要用以自動稽核的稽核規則檔。設定好之後，爾後只要一置入 Java Card 並經 PIN 碼驗證成功，軟體便會自動依據預先設定的自動稽核設定清單執行檔案稽核檢查作業。設定畫面如圖十所示。



圖十：自動稽核設定畫面

瀏覽選取並設定稽核規則檔後，此功能便會統計該規則檔中包含的目錄數及檔案數。使用者可以勾選決定是否要使用此項功能。

3.4 使用限制

我們所設計的檔案稽核軟體，除了在檔案完整性檢驗技術上改用計算 HMAC 碼取代訊息摘要值外，其餘功能均參考同質的開放原始碼軟體，如 AIDE[4]及 wxChecksums[18]等，因此在使用特性上大同小異。這類軟體只能檢查出檔案是否變更，既無法防範入侵，亦不能阻止不當的檔案存取；且僅能檢查出檔案遭到修改，無法追蹤檔案如何被修改及修改的內容為何，更不能將檔案修復。

四、結論

通常檔案稽核軟體多應用於針對入侵偵測系統、防火牆、重要的網路服務伺服器等軟體組態檔或系統設定檔，定期執行稽核工作，以確保重要的系統檔案在遭到竄改或竊取後，能及早發現並迅速採取補救措施，以期使系統恢復正常運作。

檔案稽核的主要功能雖僅在於檢查出檔案是否變更，然而其所發揮的效用可謂「小兵立大功」。若系統管理者不予以善加利用之，光是要找出哪些檔案是被攻擊者竄改或是後門程式，就得曠日費時，所必須付出的成本及代價實在難以估計。

目前一般的檔案稽核軟體，多數僅以 SHA-1 或 MD5…等其他有安全疑慮的雜湊函數法計算檔案訊息摘要作為確認完整性的依據。本研究以 HMAC-SHA-2 取而代之則更安全，雖然在執行效率上一定會比單純計算訊息摘要值來得慢，但安全還是首要考量。此外我們以 Java Card 儲存 HMAC 金鑰，使用者藉由卡片存取不僅便利，而且金鑰不存留在電腦中，避免遭他人竊取破解的可能，亦因此提高金鑰使用上的安全。

五、致謝

本研究部分成果承蒙國科會計畫經費補助（NSC 93-2213-E-017-001），特此致謝。

六、參考文獻

[1] 伊原秀明著，蘇秉豐譯，蔣大偉校編，*Tripwire for Linux 系統稽核*, O'Reilly, 2001 年 11 月初版.

[2] 陳志群著, 周利欽、翁御舜譯, *智慧卡技術實務 -使用 Java Card*, 碁峰, 2002 年初版.

[3] 謝續平, *網路安全概論*, http://dsns.csie.nctu.edu.tw/course/intro-security/2005/index.html, 2005 年.

[4] AIDE, http://www.cs.tut.fi/~rammer/aide.html.

[5] C.E. Ortiz, "An Introduction to Java Card Technology", http://developers.sun.com/techtopics/mobility/javacard/articles/javacard1/, May 29, 2003.

[6] E.H. Spafford and G.H. Kim, "The Design and Implementation of Tripwire: A FileSystem Integrity Checker", http://ftp.cerias.purdue.edu/pub/papers/Tripwire/Tripwire.pdf, February 23, 1995.

[7] H. Krawczyk, M. Bellare, R. Canetti, "RFC-2104 HMAC: Keyed-Hashing for Message Authentication",ftp://ftp.rfc-editor.org/in-notes/rfc2104.txt, February 1997.

[8] IBM, "JCOP20 Technical Brief", http://www.zurich.ibm.com/jcop/download/specs/JCOP20-Brief.pdf.

[9] M. Nystrom, "Identifiers and Test Vectors for HMAC-SHA-224, HMAC-SHA-256, HMAC-SHA-384, and HMAC-SHA-512: draft-nystrom-smime-hmac-sha-02", Internet-Draft, http://www.ietf.org/internet-drafts/draft-nystrom-smime-hmac-sha-02.txt, June 27, 2005.

[10] NIST, "NIST Brief Comments on Recent Cryptanalytic Attacks on SHA-1", http://csrc.nist.gov/hash_standards_comments.pdf, February 18, 2005.

[11] NIST, "Digital Signature Standard(DSS)", FIPS PUB 186-2, http://csrc.nist.gov/publications/fips/fips186-2/fips186-2-change1.pdf, January 26, 2001.

[12] NIST, "Secure Hash Standard", FIPS PUB 180-2, http://csrc.nist.gov/publications/fips/fips180-2/fips180-2withchangenotice.pdf, August 1, 2002.

[13] NIST, "The Keyed-Hash Message Authentica tion Code (HMAC)", FIPS PUB 198, http://csrc.nist.gov/publications/fips/fips198/fips-198a.pdf, March 6, 2002.

[14] P. Cheng, R. Glenn, "RFC2202 Test Cases for HMAC-MD5 and HMAC-SHA-1", ftp://ftp.rfc-editor.org/in-notes/rfc2202.txt, September 1997.

[15] Sun Microsystems, "Java Card 2.2.1 Platform Specification", http://java.sun.com/products/javacard/specs.html.

[16] The GNU Privacy Guard, http://www.gnupg.org/.

[17] W. Stallings, Cryptography and Network Security, Third Edition, Prentice-Hall, Inc., November 2002.

[18] wxChecksums, http://sourceforge.net/projects/wxchecksums/.

[19] X. Wang, D. Feng, X. Lai and H. Yu, "Collisions for Hash Functions MD4, MD5, HAVAL-128 and RIPEMD", http://eprint.iacr.org/2004/199.pdf, August 17, 2004.