國立交通大學

資訊管理研究所

碩士論文

於 Auto-ID 的環境中使用加密傳輸

Secure Data transmission under Auto-ID

研 究 生： 陳俊麟

指導教授： 羅濟群 博士

中華民國 九十三 年 六 月

於 Auto-ID 的環境中使用加密傳輸
Secure Data transmission under Auto-ID

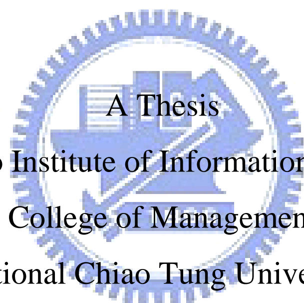
研 究 生： 陳俊麟　　Student: Jun-Lin Chen

指導教授： 羅濟群 博士　　Advisor: Chi-Chun Lo

國立交通大學

資訊管理研究所

碩士論文



A Thesis

Submitted to Institute of Information Management

College of Management

National Chiao Tung University

in Partial Fulfillment of the Requirements

for the Degree of

Master of Business Administration

in

Information Management

June 2004(畢業年)

Hsinchu, Taiwan, the Republic of China

中華民國　九十三　　年　六　月

於 Auto-ID 的環境中使用加密傳輸

學生 陳俊麟　　　　　　　　　　　　　　　　　指導教授 羅濟群 博士

國立交通大學資訊管理研究所 碩士班

摘要:

本論文主要是針對原有 Auto-ID 的架構中，因使用明碼傳輸所造成的安全議題加以討論並改良，其方法為使用一對稱式加解密演算法-TEA 來保護傳輸的 EPC，並針對此加密電路的實行性作了研究。

Auto-ID 在物流管理上所造成的影響，可比當年條碼正式實用後所造成的影響，而目前 auto-ID 的所有應答機制均為明碼傳輸，在未來， auto-ID 被廣為使用後，在安全上一定會面臨很大的問題，因此，如何達成一個安全的傳輸環境，實為本論文的探討主題。又有鑒於無線應答機(tag)晶片為節省成本，其核心晶片並無強大運算能力與大量的記憶體。為解決安全上的議題，將傳輸過程加密為一最可行的方式。然因應無強大運算能力的應答器核心，欲達成加密傳輸之目地，需對常見之加解密演算法加以研究，結論為擇出 TEA 演算法。TEA 演算法特性為只使用加法(ADD)與位移(shift)，並以簡單的數位電路就可以完成設計。

本論文也重新設計且定義了讀碼器、應答器、金鑰管理以及所使用的資料結構並針對原有的 Auot ID 傳輸協定進行改良，使改良後的系統在面對各種常見的惡意攻擊時具有足夠的強度。另外，本論文針對所提出的架構中應答器核心所需的加密電路進行了數位電路設計，以 VHDL 來實際模擬實現該加密電路所需的成本變估與實際可行性。

Secure Data transmission under Auto-ID

Student Jun-Lin Chen Advisor: Dr. Chi-Chun Lo

Institute of Information Management

National Chiao Tung University

Abstract:

The present paper provides an improved architecture for encryption/decryption based on the original Auto-ID. The traditional architecture in Auto-ID transmits the EPC code between tags and reader in plain text that causes the risk of theft of data during transmission.

The architecture provided by this paper is based on a symmetric encryption /decryption. Due to the matter of cost, the Auto-id is a tag with less memory and poor computing ability. For the capability of the original auto- ID. The encryption/ decryption algorithm is designed to be applied by Add and shift digital Algorithm.

The present paper provides a new architecture and the responding data structure. Also redesign the structure of tag, readers, and a new protocol for the new encrypted data transmission. Furthermore, a discussion for realization the encryption circuit on FPGA and VHDL environment is provided.

Keyword: Auto-ID, TEA, encryption algorithm

誌謝

論文完成後，也代表研究所的學習生活差不多告一個段落了，這二年來，發生了很多事情，首先，要把這篇論文獻給我過世的父親，您是走的這麼的突然，來不及見到我完成這二年的學業。而在論文構思與寫作期間，提供我經濟與靈感來源的專利事務所的所長與同事，在此要謝謝你們讓我有能力與勇氣繼續完成第二年的學業。當然，其中最重要的，要感謝的是我的指導老師羅濟群老師，課業上，羅老師在 Meeting 時總是會給我們許多寶貴的觀念及意見，指引我們正確的思考方向。讓我能在作研究時能準確掌握了該有的方向。

此外，我相當感謝同班的同學們，謝謝大家在遇到困難時可以互相扶持、互相勉勵，而與丁武、永鑫討論本論文相關的想法也對本論文的產生有相當大的助益；當然，我還要特別感謝我的家人，尤其是我的母親，要不是她這麼支持我，我無法這麼幸福地在這裡求學，真的謝謝妳，辛苦了。

目　次

摘要: .... **I**

ABSTRACT: .... **II**

誌謝 .... **III**

目　次 .... **IV**

圖目錄 .... **VII**

表目錄 .... **IX**

第一章　緒論 **1**

1.1 研究動機 1

1.2 研究目的 2

1.3 研究方法 3

1.4 章節介紹: 3

第二章　相關文獻分析 **4**

2.1AUTOID 介紹 4

2.1.2　Auto ID 的特色: 5

2.1.3　Auto-ID 應答器 8

2.2 無線傳輸的安全與 TEA 12

2.2.1　無線傳輸上的安全 12

2.2.2　TEA 演算法 15

2.2.3　TEA 的強度分析 17

2.2.4　TEA 與其他演算法的比較 19

2.2.5　TEA 的其他討論: 24

第三章 改良式 AUTO-ID 系統 **25**

3.1 現有系統問題分析 25
3.1.2 竊聽(Eavesdropping)： 25
3.1.3 偽裝(Masquerade)： 25
3.1.4 重送攻擊(Replay)： 25
3.1.5 訊息竄改(Message Modification)： 26
3.1.6 阻斷服務(Denial-of-Services)： 26
3.2 安全 AUTO-ID 傳輸環境。 26
3.2.1 安全傳輸的實現 27
3.2.2 應答器的設計 27
3.2.4 資料結構 30
3.2.5 安全資料傳輸的運作方式 33
第四章 安全性分析 36
4.1 與原有系統比較 36
4.1.1 竊聽(Eavesdropping)： 36
4.1.2 偽裝(Masquerade)： 36
4.1.3 重送攻擊(Replay)： 37
4.1.4 訊息竄改(Message Modification)： 37
4.1.4 阻斷服務(Denial-of-Services)： 37
第五章 加密電路的實現 39
5.1 元件分析: 39
5.1.1 8bit 的 XOR: 40
5.1.2 5 位元的右位移與 4 位元的左位移: 41
5.1.3 8 bit 的全加器 42
5.1.4 8 bit 的暫存器: 43
5.2 加密電路的界面設計: 44
5.2.1 TEA 加密核心所需的界面: 44
5.2.2 資料流協定 45

5.2.3 電路運作的順序: 45
5.2.4 結果測試: 46
5.3 成本評估: 46
5.3.1 Magic 的模擬 46
5.3.2 在 FPGA 的模擬: 47
第六章 結語 48
6.1 研究價值 48
6.2 未來方向 48
參考文獻 49
附錄 50
A. BENCH.CPP 50

圖目錄

圖 1 AUTO-ID的架構[1] 5
圖 2 READER 的工作原理 6
圖 3 EPC碼的格式 6
圖 4 一個應答器的構成[2] 8
圖 5 應答器的TREE TRAVERSAL 9
圖 6 CLASS 0 的運作協定[2]。 11
圖 7 OPEN認證 13
圖 8 SHARED-KEY 認證方式 13
圖 9 TEA 的單一攪拌示意圖[4] 16
圖 10 DES的演算法 20
圖 11 TEA 與BLOW FISH 的演算法比較 21
圖 12 RC5 與TEA的比較 22
圖 13 XTEA 流程圖 24
圖 14 核心IC的設計圖 28
圖 15 整個應答器的組成 30
圖 16 ID3 資料格式 31
圖 17 M(ID2,ID0) 31
圖 18KEY 值的格式設定 32
圖 19 MODE 3 的通信協定圖 34
圖 20 金鑰管理機制 35
圖 21 TEA 加密運算的元件綜覽 39

圖 22 CARY CHAIN 串聯 40
圖 23 8 BIT XOR匣 41
圖 24 5 位右位移。 41
圖 25 右 4 位移電路 42
圖 26 全加器 42
圖 27 八BIT的全加器 43
圖 28 8 位元的暫存器。 43
圖 29 TEA加密電路所需INTERFACE腳位 44
圖 30 WRITING CYCLE 45
圖 31 READING CYCLE 45
圖 32 核心電路的CORE時序圖。 46
圖 33 XOR 的IC LAYOUT 47
圖 34 4BITS 的全加器 47

表目錄

表 1　研究方法 3

表 2　TEA對不同攻擊的強度表 19

表 3　TEA與其他演算法的效能比較 23

表 4　AUTO-ID 與改良後AUTO-ID面對各種攻擊時比較 38

表 5　使用FPGA XC2048XL 為目標以VHDL模擬所得資料 47

第一章　緒論

1.1 研究動機

與研究方向相關

個人自進入交大資管所以來，一直將研究的重心著重在無線傳輸的相關領域，在無線資料傳輸的相關技術中，又對數位無線資料傳輸有著相當大的興趣。在無線傳輸的領域中，包括了目前最受注目的無線區域網路，也包括了像 Auto-ID 這種全新的領域。第一次接觸到有關 Auto-ID 的相關資料時，就有感於這個技術會像 20 年前的條碼(Bar code)一樣，深深的改變了商業模式與生活模式，乃開始虔心研究 Auto-ID 相關技術。而 Auto-ID 是一個嶄新的技術規格，雖可預見將會對人類作出改變，而欲完成此目地的先決條件是必需有更多的人參與此項技術的研發與改進，方能將其改善至更好以造福人群。

與修讀課程相關

個人於交大資管所研讀期間，不但修習了關網路通訊方面相關課程。也深知一個安全的網路才是一個真正有使用價值的網路，在無線網路傳輸方更是使然。故也多方修習次訊安全方面相關課程，使自已在關於建造一個安全的網路方面不會只流於技術面的安全，而忽略掉在管理層面所需下的功夫。而資管所的供應鍊課程，更是讓我體會到現行物流系統的極限在哪，而如果這個極限被突破後，整個物流系統的改變會比現在更方便，更有效率。也因此，選擇了 Auto-ID 上安全不足的問題來作研討。希望能為改善這技術貢獻一份心力。

與社會動態相關

2003 可以說是 RFID 光輝璀璨的一年，首先是美國國防部宣佈將採用無線化的物料管理系統，接著是全世界最大的零售流通商威名百貨(Wall-Mart) 也宣佈在 2005 年起，所有的的零售供應商必需配合其使用 RFID 的供貨系統。利之所趨，一線的軟體與服務提供商如微軟(Microsoft)，IBM，昇陽(Sun)，與 ERP

軟體巨擘 SAP 皆同聲宣佈提供 RFID 的發展平台。可知無線化的識別系統是一股擋不住的潮流。而在台灣，因應此潮流，已在高雄港及航空貨運園區建立起 RFID 的物料管理系統。相信在不久的未來。無線識別系統將成為市場的主流，並帶來龐大的商機。

1.2 研究目的

根據 Auto-ID 的技術規格，在讀碼器(Reader)與應答器(Tag)之間所傳遞的電子產品碼(EPC)，實為一明文傳輸之架構，有心人士得以監聽(sniffering)無線電波之方法輕易複製出具有同樣電子產品碼(EPC)之應答器，而形成應答器假冒。而在物流系統中，應答器假冒之問題可能會造成物料的盤點不正確而造成物料供應的不正常或是財產的損失。

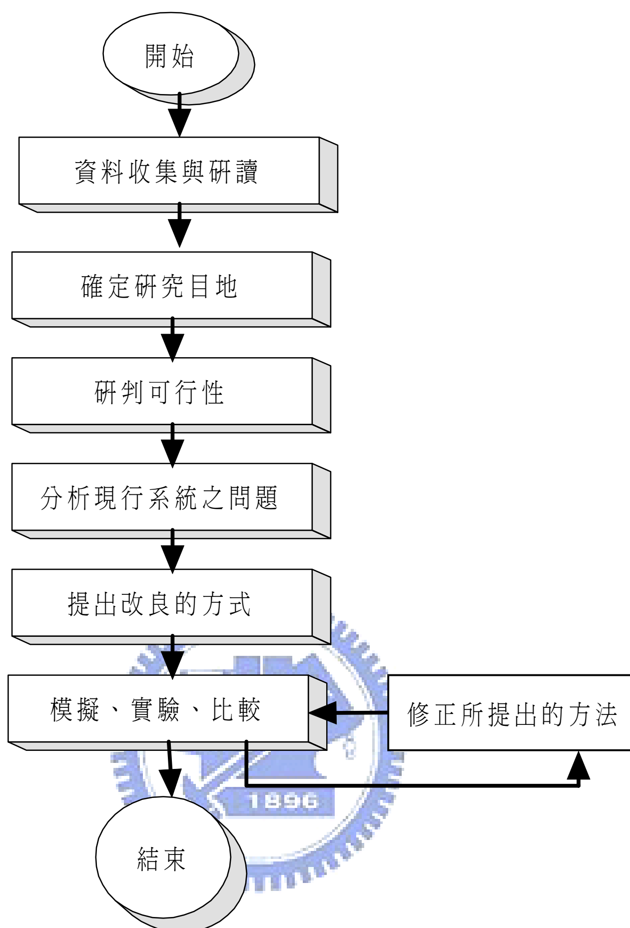
而另一方面，在無線傳輸上，已具有相當多的加/解密演算架構，如 DES 這種對稱式加/解密架構，或是 RC4 此類已在無線網路廣為使用的演算法。雖然將傳輸的資料加密可保障資料於無線網路傳輸時不被人竊用。然上列之演算法需要強大的運算能力與大量的記憶體空間。為了因應應答器的低成本特性，故需提出一有效且不需大量計算能力與記憶體之演算法。

且為與原有之未加密應答器相容，仍如何將此演算法在不大量變更架構的前提下，以模組或是指令集的的方式加入原有之核心電路架構，實為一重要之課題。故本論文提出以下研究目地

- 研究現行之加/解密演算法，並擇出不需要大量計算能力與記憶體之演算法。
- 因應該演算法與相容性，提出相對應的改良應答流程。
- 因應該演算法，提出一合理之金鑰管理機制。
- 測試該演算法之效能與設計是否適用於 Auto-ID 的環境

1.3 研究方法

表 1　研究方法



1.4 章節介紹:

在第二章中，針對與本論文主題相關的研究做文獻探討包括：Auto-ID 的架構及應答器的分類、以及所選定的加解密演算法介紹，在第三章中，先對之前 Auto-ID 問題分析後，接著提出改良的方法，在第四章中，對本論文所提的方法做分析與比較，包括實行面與安全面；在第 5 章中以 VHDL 與 FPGA 來進行數位電路的模擬與估算。最後，在第六章中對本論文做結論及未來的研究方向。

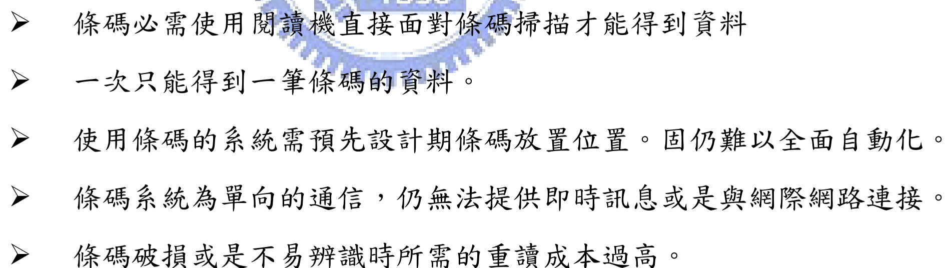
第二章　相關文獻分析

2.1AutoID 介紹

2.1.1　自動識別技術 (Auto-ID)總論

1999 年起，美國麻省理工學院 MIT 推出了 Auto-ID 的技術規格。由 Sanjay Sarma 教授帶領的團隊提出了一個完整的物件辯識系統，而這個系統同時整合以了數個現行的技術特性。Auto-ID 的前身是無線射頻識別(RFID)，其發展歷史已經有數十年了。

在 20 年前，條碼(Bar Code) 開始大量出現在物流管理上。全球化的物流與大型流通場所的出現使追蹤物料的需求開始大量增加。也應此。條碼可說是目前最被廣為使用與不可少的物料追蹤工具。但是，條碼目前也面臨了以下的問題。



- 條碼必需使用閱讀機直接面對條碼掃描才能得到資料
- 一次只能得到一筆條碼的資料。
- 使用條碼的系統需預先設計期條碼放置位置。固仍難以全面自動化。
- 條碼系統為單向的通信，仍無法提供即時訊息或是與網際網路連接。
- 條碼破損或是不易辨識時所需的重讀成本過高。

Auto-ID 技術被期待於使用極低成本的無線射頻(RF)應答器(Tag)-或稱標簽來管理物品以解決物料管理清晰度不高的問題。應答器被附著在箱子，盒子，或是被含在製成的材料中，在供應鍊管理系中提供管理者雙向的資料流與資訊流。而 Auto-ID 的最大改進處為使用與 XML 相容的 PML 透過網路來收集與提供從應答器所收集的數據。而這與傳統的條碼相較之下，可以大大的削減在收集資料的時間成本與人力成本。

RFID 使用主動式的應答器，並以自訂的系統來進行收集後的資料管理。在過去的數十年內，RFID 並未發展出一大量採用的標準，故於推行上碰到了

問題，那就是使用者面對不同的系統時，必需針對相對應的系統進行改造。另一個問題在於應答器的成本上，RFID 使用的應答器為主動式應答，所以必需有一電池以提供無線電波的發射，使得在範圍內的接收器都能接收。但是距離越遠，干擾越重，定位精確度也更差。而電池本身也有其使用期限與成本上的問題。目前的 RFID 應答器依功率大小的差異，訂價從美金二元起跳，也因此使得 RFID 發展至今仍未廣為使用。

2.1.2 Auto ID 的特色:

2.1.2.1 多階層式的物件識別系統:

Auto-ID 主要的貢獻是提供一種前端與後端應用技術的整合，其主要的原理為引入電子產品碼(EPC, Electronic Product Code)作為識別物件的基礎，使用一稱為 Savant 的中介轉譯服務程式，將 EPC 轉為一符合 PML 格式的要求(Request)。搭配一物件名稱伺服器(ONS)作為解析(Resolve)要求網路上相對應的資訊。這些資訊可能包括有物品的名稱，描述，製造資訊或甚至是食譜，經由 Savant 傳至客戶端以茲系統使用[1]。整個架構如圖 1 所示。

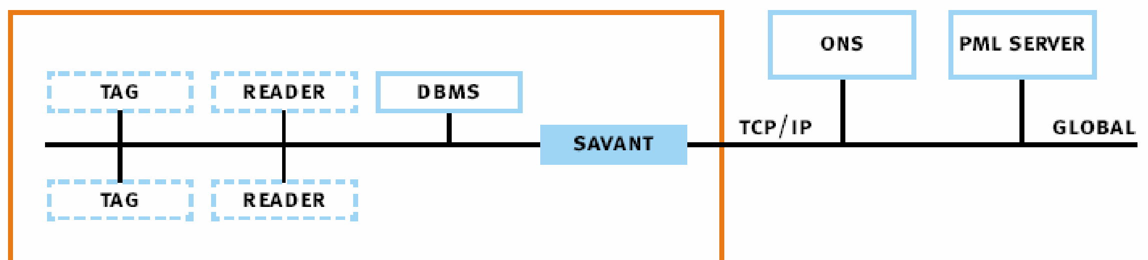


圖 1 Auto-ID 的架構[1]

以下則針對圖 1 的系統中所有元件進行解說。

TAG:

又稱為無線應答器或簡稱應答器，在 Auto-ID 的系統中，應答器存放能識別物件的 EPC 碼，並在讀碼器實行呼叫時，回應該讀碼器並將所儲存的 EPC 碼傳遞至讀碼器

READER:

讀碼器是一個多頻，且多重運作模式的資料收集工具，在 Auto-ID 的定義裏，讀碼器必需連接上後端的某種 TCP/IP 網路。一個讀碼器的工作原理如圖 2 所示。

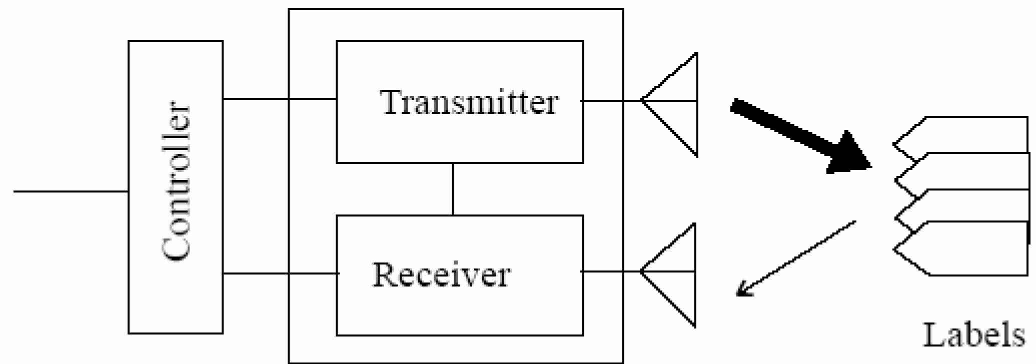


圖 2 READER 的工作原理

EPC:

EPC 是個四個欄位的編碼技術，一共具有 96 個 bit。這 96 bit 幾乎可以用來表示地球上的每一個物件[1]。這四個欄位所代表的意思與其欄位的大小如下：

1. 檔頭(Head), 8 bits
2. 製造商序號(Manufacturer), 28 bits，
3. 產品品別號(Product), 24 bits
4. 產品序號(Serial Number), 36 bits

比如現在用的滑鼠的 EPC 可能如圖 3 所示：

F127.C238.DF1B.17CC

圖 3 EPC 碼的格式

PML:

PML(Physical Markup Language)是一種基於 XML 用來描述物件資訊的語言。PML 是 XML 的再延伸。這種語言的重點在於定義如何正確的呈現(display)一個物件。

ONS:

物件命名服務(Object Naming Service, ONS)是一個使用類似 DNS 的觀念的產品。它負責將來查詢的 EPC 碼，轉成提供此項物件資訊的網路位址。

SAVANT:

SAVANT 是一種智慧型的代理程式(intelligent agent) 在 Auto-ID 的整個資料流裏擔任管理的角色，並負有讓資料流與外部程式溝通的責任。Savant 藉由 ONS 的幫助，將 EPC 的數字轉化為 PML 伺服器所了解的 PML 格式查詢(Query)向提供該物件的伺服器查詢，並接收回應加以處理。

Savant 又分為 Savant 與被動式(idiot) Savant 二種。其中 Savant 是一用以將「合成」過的資料送給應用程式的主動元件。而被動式(idiot) Savant 則只能經由 XQL 的查詢提供「有限」的資料給應用程式。

2.1.2.2 被動式應答:

使用被動式的應答機制，使得應答器在能量上的消耗與成本上大大的降低，而其發射電波所需的能量則來自讀碼器呼叫時所產生的電磁波。Auto-ID 本身所訂定的規格中，將讀碼器與應答器的有效距離訂為 3 公尺。當然，主動式的應答器一定優於被動式的應答器，而被動式應答的器的優點在於可以減少干擾與省去電池的成本，使得標簽的應用市場更大。而在 Auto-ID 的技術定義裏，目前已經定義有 Class 0 與 Class 1 這二種標籤的規格。其中 Class0 是一個單純的呼叫與回應的機制[2]。Class 1 是用於定義了較複雜的系統交握過程與控制命令，乃是為了因應要制定更高等級的標籤用於更嚴格要求的系統的呼聲。由於 Class 1 的標準目前仍未有詳細的規定，以下將只針對 Class 0 的應答器作一簡單描述。

2.1.3 Auto-ID 應答器

根據 MIT 的 Auto-ID 技術規格[2]，一個被動式的應答器由以下四個部份所形成。

- A.核心 IC
- B.天線(Antenna)
- C.IC 與天線之間的連接匯流排 (connection between IC an antenna)
- D.放置天線的基座。(the substrate on which the antenna resides)

而在實用中，這樣的應答器很可能在生產的過程中就直接加在容器裏。例如在包裝物品的厚紙箱中。如圖 4 所示:

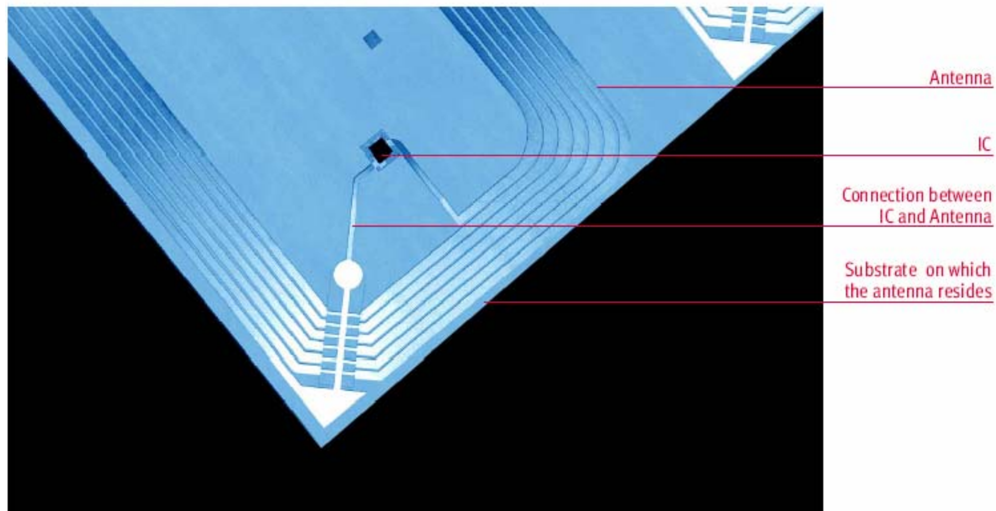


圖 4 一個應答器的構成[2]

2.1.3.1Class0 應答器技術特性

Auto ID 的 Class 0 初步技術規格中[2]裏定義了 auto-ID class 0 應答器所必需具備的技術特色:

- 1 必需出廠時就至少內建有 EPC 碼，一個 24 bit 的中止碼，以及其他需要的資料。
- 2 可被讀碼器解讀。
- 3 可被視為被群體選擇的一部份。

- 4 可以被單獨的破壞(destroy)。

而在設計一個 Class 0 的應答器時，在實務上所需滿足的條件為:

- 1 必需是非常低價。
- 2 應答器在設計上就必需考量到可能會受到群體選擇的情況。
- 3 必需具有夠長的工作距離。
- 4 要能不受鄰近的其他類似的讀碼系統干擾。

2.1.3.2 運作方式:

由於以上的限制，在[2]裏提出了一個(Reader talk first) 的運作方式。需要被讀取的應答器全部可以表示成一個二元樹。在此我們假設這樣的一個二元樹足以表示所有的 EPC 碼，如圖 5 所示

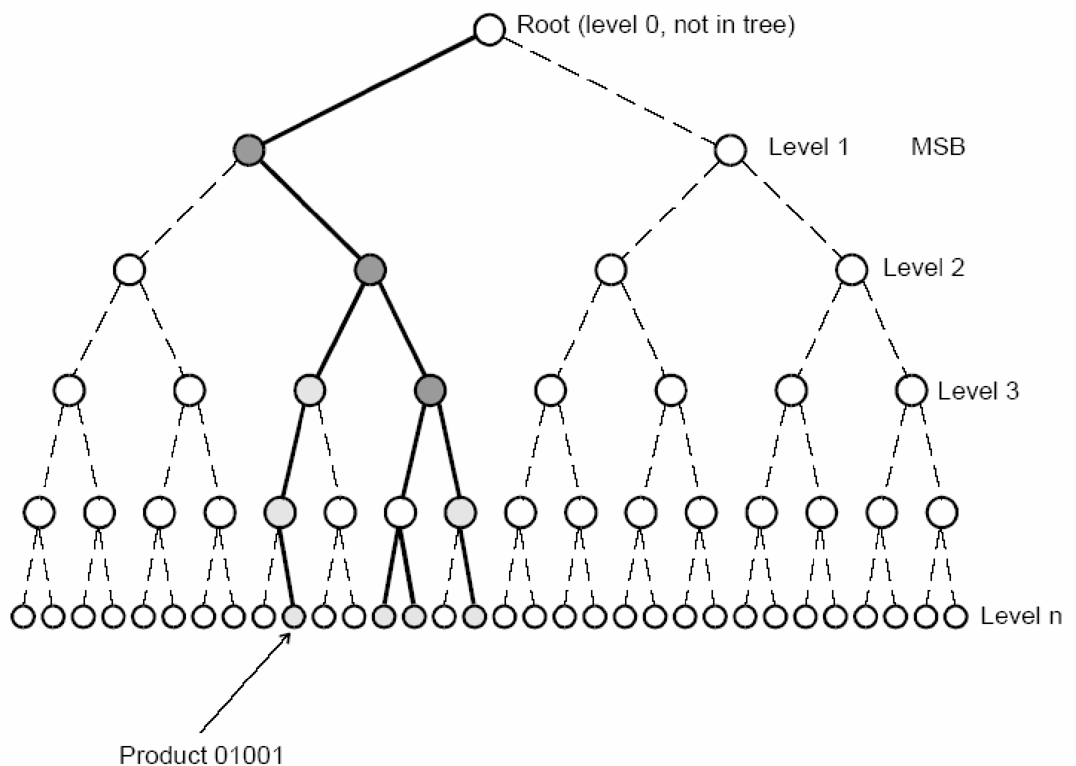


圖 5 應答器的 Tree traversal

如果讀碼器會依某個路徑下的子節點數量分為 single populated 與 multiple populated，而讀碼器會依二元樹的方式去整個掃描所範圍內的 EPC 碼，並依

EPC 碼叫號。而當應答器收到它獨有的 EPC 碼時，就產生回應並將 EPC 碼傳回讀碼器。這樣的叫法雖然很直接，但是很明顯的在效率上與安全會有很大問題。在效率上，因無線電的電波發送最多只能一次進行數千次的發送而已，用於本系統中，讀碼器一次掃描能進行大約數千次的掃描就已經是極限了。[2]

因此在 Auto-ID 的 Class 0 應答器技術規格裏[2]，創造出三種存在應答器裏的代號，並依此定義了以下三種叫號的依據:

- ID0 是一個單純的亂數，使用於工作模式 mode 0，其建議長度為 10bit，在每次叫號並回應後就予以重設(Reset)，而讀碼器會先行設定所有的應答器進行 Mode0 的叫號方式後，開始叫號，而應答器在收到叫號後，如果與 ID0 相同，則將 ID2 送出。並重設 ID0。
- ID1 是一個基於 EPC 碼所產生的靜態碼，使用於工作模式 Mode 1，其建議長度為 10bit。用以識別或是用回復一個已經建立的連線。ID1= Hash(EPC),是一個出廠時就燒在其中的固定值。而應答器接收到來自讀碼器的 ID1 叫號後，如果與 ID1 相同，則回應且將 ID2 送出。
- ID2 為 96bit 的碼，使用於 Mode 2 的工作模式，該值為該應答器所對應的 EPC 碼。

在圖 6 展示了 Class 0 應答器三種模式的詳細運作流程[2]

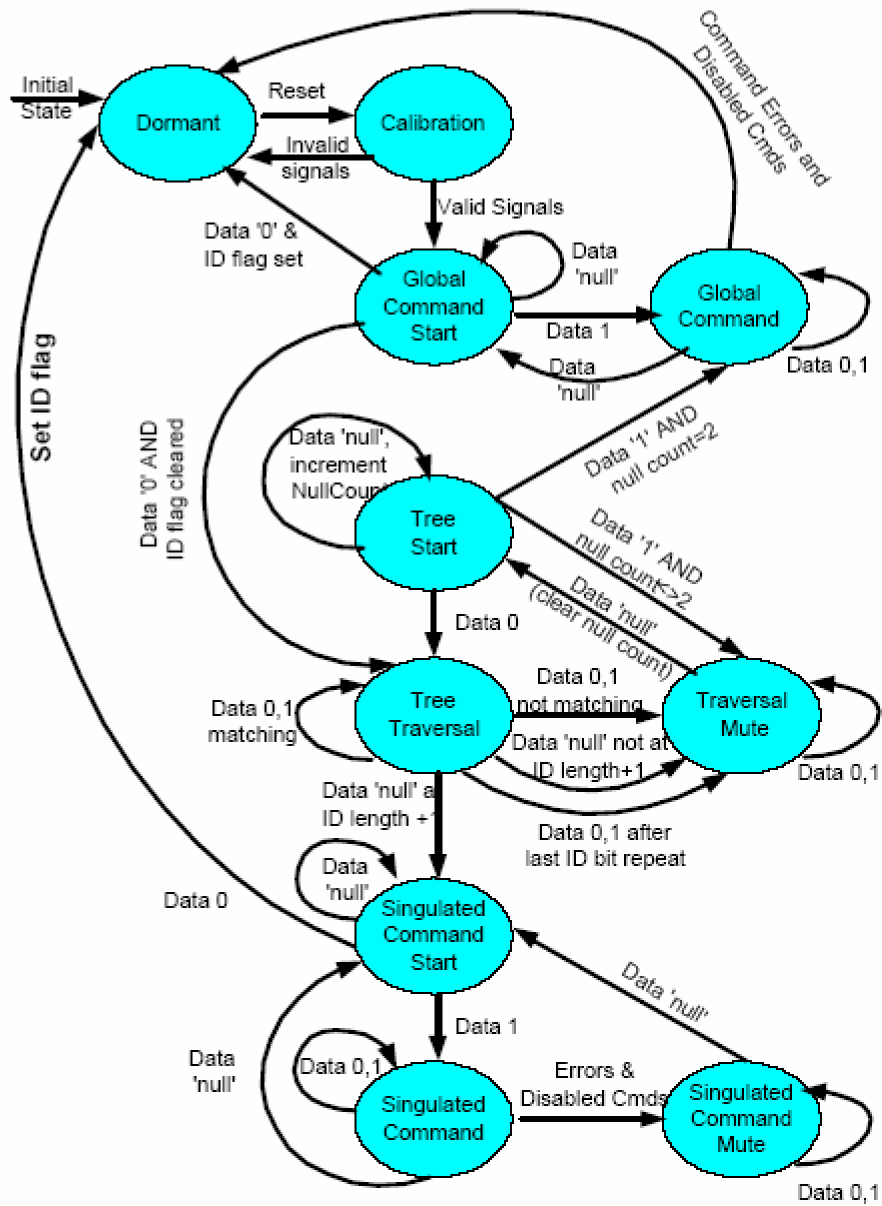


圖 6 Class 0 的運作協定[2]。

2.2 無線傳輸安全與 TEA

2.2.1 無線網路安全

無線網路，汎指所有不使用實體電纜為介質而構成的網路。而這樣的架構讓無線網路在安全性上先天上就與有線網路有許多差異。欲控管有線網路的安全性僅需監控接入區域網路中的所有實體節點即可。因此有線網路的存取安全控制被視為區域網路節點的實體控制。因為有線網路上資料的傳輸是直接送至一個特定的目標位址，其間並不會產生所謂存取授權(privacy) 的權限問題，除非有心人士以特定的儀器加以攔截，因此有線區域網路的安全性考量主要在於區域網路實體連結的破壞。

但對無線網路而言，資料的收送主要經由無線電波透過空氣介質來廣播傳輸，所以它會被某一特定服務區域內的所有無線網路用戶端所接收。由於無線電波可穿透天花板、樓板及牆壁等結構體，因此資料傳輸可能會被不同樓層甚至不同建築物中的非預期接收者所得到。因此，架設一個無線網路就好像將無數的網路節點隨意置於各處，甚至是停車場中。此時資料的存取授權就會變成一個無線網路中主要的考量。因為對無線網路而言，我們並沒有辦法要求無線電波只單點傳輸至某一接收端手中。也因此。無線網路採取了以下的二個方式來保障傳輸上的安全。

2.2.1.1 網路使用者認證:

以 IEEE 802.11 為例，就定義了二種型態的認証方法：Open 及 Shared-key。[3]而認証的方法必須被設定於每個使用端，這些設定值必須與無線存取基地台中的設定值相符。

整個 Open 認証方式的認証流程均以無加密的文字方式進行，而且使用端甚至不須要提供正確的 WEP 金鑰就可與無線基地台互相連繫。而 Shared-key 認証方式，無線基地台會送出要求登錄的無加密文字的封包給使用端，而使用端則必須將此一文字與本身持有的 WEP 金鑰一起加密後，再傳回無線基地台，

而由無線基地台來判別是否此一使用端可存取網路資源。圖 7,8 展示這二種認證方式的運作。

圖 7 open 認證

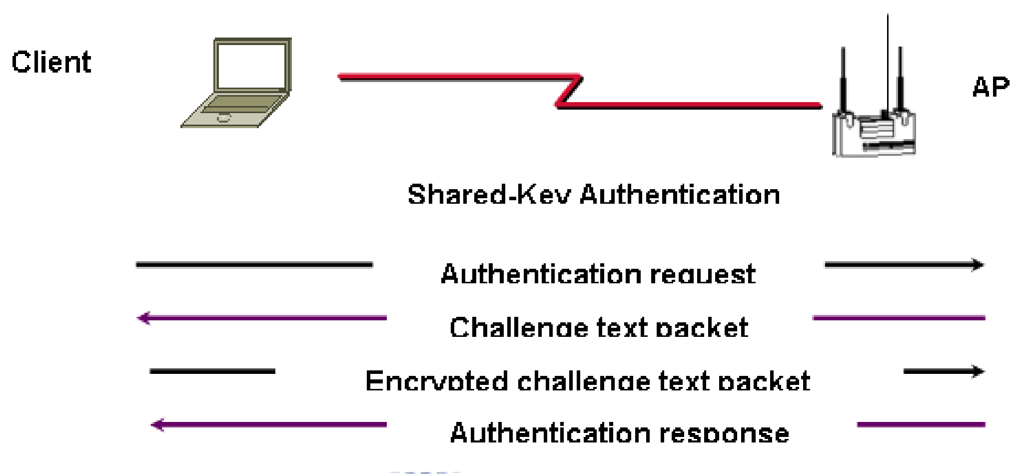


圖 8 shared-key 認證方式

有些無線區域網路設備供應商也支援由網路卡實體位址(MAC address)來執行認證的動作，一個使用端僅當它的 MAC address 符合無線基地台中儲存的認證表列時，才被允許存取網路資源。

2.2.1.2 傳輸資料加/解密:

以 IEEE 802.11b 標準為例，其中制定了一個可選用的加密機制，稱之為 Wired Equivalent Privacy 或 WEP，可提供安全的維護無線區域網路的資料傳送。WEP 使用對稱的加/解密機制，即在資料加/解密的過程中均使用相同的加密金鑰。使用 WEP 的主要目的[3]是：

- 存取控制：避免非法且無正確 WEP 金鑰的使用者，任意存取網路資源。

- 存取授權：利用加密的機制，保護無線區域網路的資料傳送，並且僅允許持有正確 WEP 金鑰的使用者才可解讀。

雖然 WEP 是一個以 RC4 演算法為基底的選用的安全選項[4]，但若要通過由 WECA 檢驗的“Wi-Fi”認証，就必須具備 40-bit 的加密金鑰機制，所以所有 WECA 的成員均支援 WEP 設定。至於 WEP 的做法，有些無線網路設備製造商利用軟體來完成加/解密的動作。而另一些設備製造商，如思科(Cisco)，則利用硬體加速器來完成資料加/解密的流程，以避免機器本身處理效能的大幅下降。

IEEE 802.11b 提供二種方式去定義使用於無線區域網路的 WEP 金鑰。第一種方式是可設定一組最多 4 把內定的加密金鑰，並由所有的無線存取設備(如用戶端及無線基地台)所共享。當使用者取得內定的金鑰後，就能安全地在同一無線區域網路子系統中與其他使用者溝通。但問題是當持有內定金鑰的使用者數量太多時，這時就不可避免的會產生安全上的顧慮。

第二種方式，每個使用端會與其他的使用端建立一個”金鑰對映(key mapping)”的關係，這是一種較為安全的方式，因為較少的使用者會持有對映的金鑰。但同樣當使用者持續增加時，則這種單點傳送的金鑰散佈方式就變成極為困難。

另以全球行動系統(Global System for Mobile communication)為例。其為史上第一個具有複合安全機制的通訊系統。它制定了以下的安全標準，包括了具有匿名性、可認證性’訊號保護與使用者資料保護。GSM 系統中的加密也只是指無線路徑上的加密，是指基地台(Base Transceiver Station，BTS)和行動服務交換中心(Mobile services Switching Centre，MSC)之間交換客戶資訊和客戶參數時，不被非法個人或團體所得或監聽。在鑒權程式中，當客戶側計算 SRES 三參數組的提供時，同時用 A8 演算法計算出密鑰 Kc。根據 MSC／VLR 發送出的加密命令，BTS 側和 MS 側均開始使用 Kc。在 MSC 側，由 Kc、TDMA(Time Division Multiple Access，分時多工)時槽號和加密命令 M 一起經 A5 演算法，對客戶資訊資料流程進行加密(也叫擾碼)，在無線路徑上傳送。在 BTS 側，把

從無線通道上收到加密資訊資料流程、TDMA 時槽號和 Kc，再經過 A5 演算法解密後，傳送 BSC(基地台控制中心 Base Station Centre)和 MSC。 而在 GSM 系統中所使用的 A3, A5 , A8 系統均為一未公開的加/解密演算法。所以其安全性與強度一直受到大家的質疑與挑戰。

2.2.2 TEA 演算法

TEA 是由 Cambridge Computer Lab 的 David wheeler 與 Roger Needham 於 1994 年首度發表。[4] TEA 的特性包括有

- 64 Bit 的 block cipher 能力。
- 128Bit 的金鑰
- 具有 Feistel network 的特性。可以一直遞迴加密以增加強度。
- 圖 9 表示了 TEA 的加密原理[4]

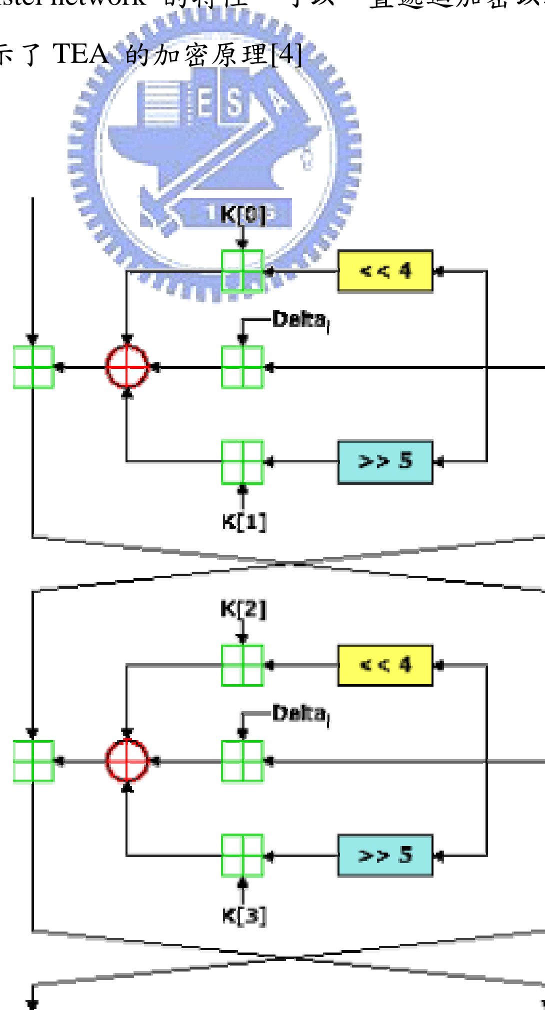


圖 9 TEA 的單一攪拌示意圖[4]

而將圖示轉化成數學示。看到 TEA 的每一個攪拌表示如下

- 明文本身 p 為 64bit 長，均分為二個相等的部份 P1 與 P2
- 金鑰 K 本身為 128 bit 長，均分為四個相等的部份 k[0],k[1],k[2],k[3]
- Delta[i] 為一任意數，可以任意設置，主要是用來累加作為 key extension 使用，讓每次攪拌所使用的 Key 值會不同。
- 每一次的攪拌(round)是由二次的循環 (cycle) 所完成。

在下列的方程式 1 ,2,3,4 中分別列出了 TEA 的加/解密方程式與使用 C 語言來表示的原始碼:

```
P1 =  P1 +(  (P2 <<4)+k[0] ) XOR  (p2+Delta(i) )  XOR  ( (p2>>5) + k[1] )  )

P2 =  P2 +(  (P1 <<4)+k[2] ) XOR  (p1+delta(i) )  XOR  ( (p1>>5) + k[3] )  )
```

方程式 1 Tea 的加密方程式

```
void encipher(unsigned long *const v,unsigned long *const w,
    const unsigned long *const k)
{
    register unsigned long        y=v[0],z=v[1],sum=0,delta=0x9E3779B9,
                        a=k[0],b=k[1],c=k[2],d=k[3],n=32;

    while(n-->0)
        {
        sum += delta;
        y += (z << 4)+a ^ z+sum ^ (z >> 5)+b;
        z += (y << 4)+c ^ y+sum ^ (y >> 5)+d;
        }

  w[0]=y; w[1]=z;
}
```

方程式 2 TEA 的 C 程式碼，以 32 個 round 為例

而 TEA 的解密方式只要反其道而行，在方程式 3 與 4 裏展示了 TEA 的解密算示與 C 原始碼。

```
P2 =  P2 -(  (P1 <<4)+k[2] ) XOR  (P1+Delta(i) )  XOR  ( (P1>>5) + k[3] )  )

P1 =  P1 -(  (P2 <<4)+k[0] ) XOR  (P2+delta(i) )  XOR  ( (P2>>5) + k[1] )  )
```

方程式 3 TEA 的解密演算

```
void decipher(unsigned long *const v,unsigned long *const w,
   const unsigned long *const k)
{
   register unsigned long       y=v[0],z=v[1],sum=0xC6EF3720,
                    delta=0x9E3779B9,a=k[0],b=k[1],
                    c=k[2],d=k[3],n=32;

   /* sum = delta<<5, in general sum = delta * n */

   while(n-->0)
      {
      z -= (y << 4)+c ^ y+sum ^ (y >> 5)+d;
      y -= (z << 4)+a ^ z+sum ^ (z >> 5)+b;
      sum -= delta;
      }

   w[0]=y; w[1]=z;
}
```

方程式 4 TEA 的解密 C Source Code

2.2.3 TEA 強度分析

2.2.3.1 Equivalent keys

針對此演算的第一個目標就是本演算法中將金鑰分為四個子金鑰的特色。此四個子金鑰而且分別在奇數與偶數回合使用。雖然每次回合會加入一個變異數 Delta，使得每次的子金鑰得以變化。所以本演算法在針對 slide attack [5] 時具有一定的強度[6].。

而針對 TEA 最有效的攻擊應該是 Equivalent keys 攻擊[7].。詳細的證明引用如下:

Lemma 1, a 是一個 n bit 的字元，$\boxplus$定義了 n bit 的加法，而$\oplus$表示 XOR。所以針對 MSBs 位元更換運算就相等於 n bit 的加法

$$a \oplus 2^{n-1} \equiv a \boxplus 2^{n-1}.$$

Lemma 2, 如果 a 與 b 是 n bit 的字元。則以下成立

$$a \boxplus (b \oplus 2^{n-1}) \equiv a \boxplus (b \boxplus 2^{n-1}) \equiv (a \boxplus b) \boxplus 2^{n-1} \equiv (a \boxplus b) \oplus 2^{n-1}.$$

Proposition 1，如果$\mathrm{TEA}_i(z, K_0, K_1)$ 表示在 TEA 中的第 i 回合。其中 z 為

32bit 的本文與二個子金鑰K_0與K_1，則

$$
\begin{aligned}
\mathrm{TEA}_i(z, K_0 \oplus 2^{31}, K_1 \oplus 2^{31}) &= [(z \ll 4) \boxplus (K_0 \oplus 2^{31})] \oplus \\
&\quad [(z \gg 5) \boxplus (K_1 \oplus 2^{31})] \oplus [z \boxplus \delta_i] \\
&= [(z \ll 4) \boxplus K_0] \oplus 2^{31} \oplus \\
&\quad [(z \gg 5) \boxplus K_1] \oplus 2^{31} \oplus [z \boxplus \delta_i] \\
&= [(z \ll 4) \boxplus K_0] \oplus \\
&\quad [(z \gg 5) \boxplus K_1] \oplus [z \boxplus \delta_i] \\
&= \mathrm{TEA}_i(z, K_0, K_1)
\end{aligned}
$$

同理可知，替換 K2 與 K3 的 MSB 對此並無助益。所以。在 TEA 演算法中，每一個金鑰都有三把等效的金鑰，一為具有 K1 與 K2 替換的 MSBs，另一為 K2 與 K3 替換，最後一為 k0 k1 k2 k3 都被替換。也因此，原本2^{128}的金鑰數量空間變成只有2^{126}個。這個問題使得 TEA 不適用於某些 Hash 上，例如 Davis-Meyer。

2.2.3.2 Related-key

在[7]中有提到 TEA 面對 Related-key 攻擊時的討論，而根據它的結論，在最佳的情境下，只需要2^{32}個挑選過的明文，並以 2 個相關金鑰來進行攻擊，只要次計算2^{32}次就有可能破譯出 TEA。

2.2.3.3 SAC Distinguisher 與 Differential Attack

在[8]中，TEA 被用以作 Strict Avalanche certification. 在少於 5 個回合的攪拌中，TEA 可以經由挑選2^{25}個選定文字，以亂數的方式來辯別出來。不過超過 5 次以後的攪拌就無法以 SAC Distinguisher 的方式來加以分辨。又另外在[9]中，指出使用一個經個 11 次攪拌後的 TEA，使用 impossible differential 攻擊法加以攻擊。經證明需要$2^{52.5}$個選定文字與2^{84}個加密後文字來作比較才得以辨別。TEA 對於 truncated differential 的攻擊強度在[10]有被提出。機率為 1 的 Truncated differentials 用來攻擊一個 17 回合攪拌過的 TEA 需要 1920 個選定明文與與2^{84} 次的計算時間。

在表 2 裏整理了以上所提的 TEA 面對攻擊時的強度。

表 2 TEA 對不同攻擊的強度表

攻擊類別	攪拌回合數	所需的選定明文	計算時間	資料來源
Equivalent Keys	Any	1	2^{126}	[6]
Related-Key	64	2^{23}	2^{32}	[7]
SAC Distinguisher	10	2^{25}	-	[8]
Impossible Differential	11	$2^{52.5}$	2^{84}	[9]

2.2.4 TEA 與其他演算法的比較

由於目前的加/解密演算法多如繁星。且不只 TEA 演算法只使用加法與位移。故有必要於針對一些其他的演算法與 TEA 作比較。

2.2.4.1 演算法挑選基準

我們透過以下的二個篩選標來挑選作為 TEA 的對照組:

- 條件一: 同樣為 symmetric block cipher

經過尋找，我們找到了 IDEA, DES, Triple-DES (DES-EDE2 and DES-EDE3), DESX (DES-XEX3), RC2, RC5, Blowfish, Diamond2, SAFER, 3-WAY, GOST, SHARK 等皆為 symmetric key cipher 的演算法。

- 條件二: 同為 Festiel network 性質，且只能使用 Add 與 shift 與 XOR 。不得使用乘法與指數。

符合此條件的只剩下 DES , BLOWFISH, RC5

2.2.4.2 加解密演算法分析

DES :

該演算法主要的不便之處在於 DES 需要大量的空間來準備多個 table 的值以供存取，包括有 Initial Permutation , Inverse Initial Permutation，Expansion permutation1，以及 Sbox，這樣的非線性化結構使得 DES 在數位電路的實現上需要大量的記憶空間來進行，對設計一個在成本上斤斤計較的 Auto-ID 應答器來說是十分不適合的，另外，DES 演算法是 RSA Data security Co. Ltd 的財產，如欲將其使用在商業應用上會增加成本。在圖 10 中對 DES 的演算法作了詳細的說明

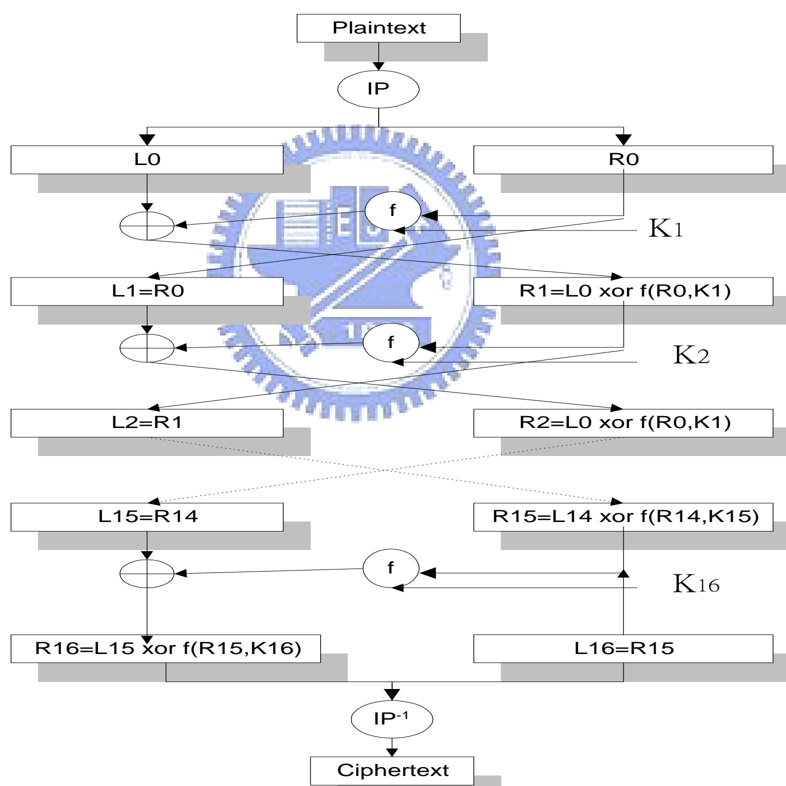


圖 10 DES 的演算法

BLOW FISH:

這是另一個適合拿來本系統作為加密的選擇。Blowfish 與 TEA 類似，都是使用 Fesitel 網路的架構，而且都只使用了加法與 Xor，由於此二系統實在過於相似，我們將此二系統的數運算流程在圖 11 裏作一比較

BlowFish algorithm　　TEA algorith

圖 11　TEA 與 BLOW FISH 的演算法比較

由圖可以清楚得知 Blow FISH 總共需要三個 XOR 與二個 ADD 閘。而 TEA 則是需要三個 ADD，二個 XOR 以及二個 shift 閘。雖然表面上看來 XOR 閘的設計難度要比 ADD 匣簡單。但 Blow fish 在設計上需要有 18 個 32bit 的 sub key 與 4 個 8*32 的 S-box 的空間需求[11]，也就是 1024 個 32bit 的需求。由於 TEA 在 key expansion 上只始用累加的方式(delta)，故不需要多準備做 key-expansion 所需要的空間 ，在電路設計上只需使用一個全加器就可以完成這樣的運算。這樣的空間優勢使得 TEA 在設計使用 auto-ID 的應答器上更具有優勢。此外，TEA 使用 shift 作了攪拌的功能，在使用數位電路進行設計時，位移電路可以直接使用 buffer 來進行，比其他任何電路還要方便。

RC5:

RC5 是一個可以自定明文長度與金鑰長度的加/解密演算法，與 TEA 一樣，只使用了 ADD, OR,SHIFT 作為運算的基礎[12]。同樣的，我們將其與 TEA 的比較展示於圖 12。

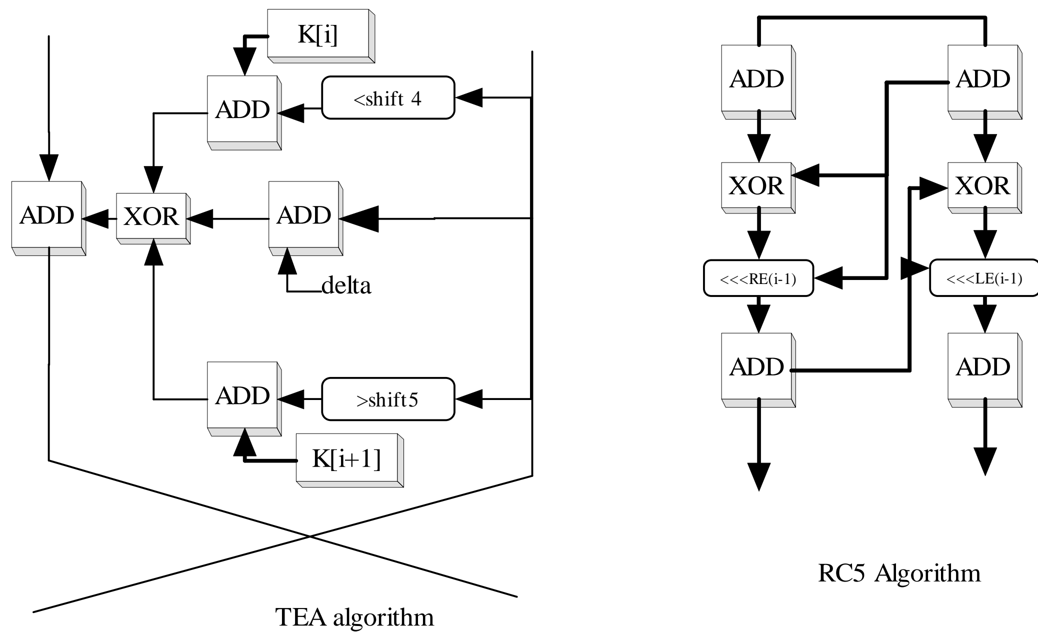


圖 12 RC5 與 TEA 的比較

RC5 雖然具有可不固定明文與金鑰長度的特色，可惜在 Auto-ID 的應答器這種固定長度資料的環境中，並未具有優勢。另與 TEA 的不同點在於，RC5 在 shift 時的位移數量是不固定的，而在數位電路上設計一個不固定位移量的位移器，會比直接用 buffer 來設計一個固定位移量的位移器來複雜上數倍，也因此，TEA 比 RC5 較符合本系統的需求。另外在實現上， RC5 是 RSA Data Security 的產品，如欲將其使用於商業產品上，需要有授權的成本考量。

2.2.4.3 程式模擬:

為了比較 TEA 與其他三項演算法的效能，以證明 TEA 的效能的卓著，現以 C 語言在電腦進行加解密的流通量(throughput)效能測試(benchmark)測試環境如下，並將模擬的結果呈現於圖表 4 中。

測試環境:

- CPU: Pentium 4 2.4G
- RAM 512MB
- OS: Windows XP professional
- Compiler : DEV C++ (compatible for gc++)

➢ 測試程式:列於附錄 1

表 3 TEA 與其他演算法的效能比較

演算法	Bytes proceed	Time taken	Throughput(MB/sec)
TEA (32 rounds)	67108864	2.554	25.059
DES	33554432	1.442	22.191
BLOWFISH(16rounds)	134217728	3.003	42.244
RC5(32 rounds)	134217728	6.32	20.253

分析表 4，首先，可以看到 BLOW FISH 不愧是最快的加/解密演算法，幾乎是其他對照演算法的二倍數度，而其主要原因為 Blow FISH 只需進行 16 次的攪拌(round)，其他的演算法均進行了 32 次的攪拌。而 TEA 演算法在與其他進行 32 rounds 的演算法來比，具有較高的流通量。

2.2.4.5 小結

由以上的內容與討論，可以得知 TEA 相較於其他演算法，具有以下的特性

- 使用累加作 Key Extension,不需大量的空間。
- 混合內容時使用固定的位移量，在數位電路的設計上具有優勢
- 使用位移作為混合，設計簡單。
- 效能卓越
- 無專利問題，成本低廉。

而這些特性，在適用本論文的前提—以數位電路來設計加密電路上具有絕大的優勢。也因此選擇以 TEA 來進行改良 Auto-ID 系統的加密演算法是一個絕佳的選擇。

2.2.5 TEA 的其他討論:

TEA 目前仍是 public domain 的版權，並未受任何的專利的保護。在製造產品上，可以省下授權費用。此點於應用在需要超低成本的 Auto-ID 應答器上具有相當的優勢。

而 TEA 還有後續的變化型，TEA 的原作者在 1997 年公開發表了 XTEA[4]，主要是針對 TEA 面對 Related-key 攻擊時相形較為脆弱進行了改善，圖 13 展示了 XTEA 的流程。另外在同一篇論文中，作者另外提出了 BLOCK TEA 的演算法，利用 XTEA 為運算區塊來組成陣列，可以一次處理較長串的訊息。Block TEA 的主要目的是增加 TEA 在處理較長訊息時的效率。

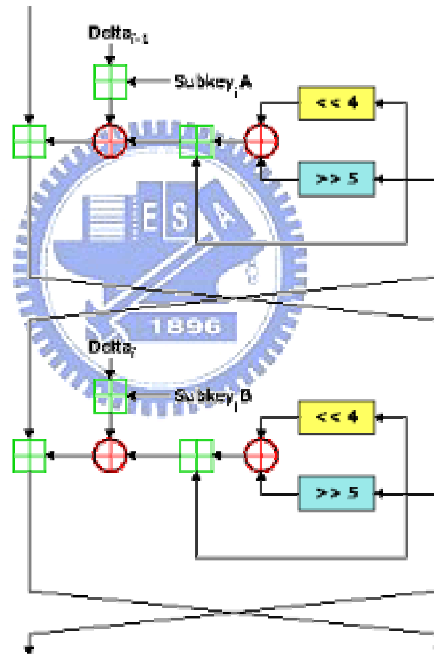


圖 13 XTEA 流程圖

第三章 改良式 Auto-ID 系統

3.1 現有系統問題分析

Auto-ID 基本上仍是是無線傳輸的一種架構。而無線傳輸因為相較起來於更容易被頡取與收集，故受到攻擊的機會又比有線傳輸來得大。不論是主動式或是被動式的攻擊，均會嚴重的影響到此系統資料傳輸的安全。接下來以常見的攻擊方式來檢視原有的 Auto-ID 架構所面臨的安全議題。

3.1.2 竊聽(Eavesdropping)：

竊聽可能是最簡單、也是最有效的攻擊方式，因為它不會留下任何線索。Auto-ID 本身的特性為讀碼機會發出控制訊號。此為較長時間的傳輸。而在應答器回應叫號的過程中，是以二個天線在很短的時間內發出資料。然後就保持在聆聽的狀態。雖然這樣增加了使用者竊聽有價值資料(EPC) 的難度，但如果竊聽設備非常靠近一個應答器，且有足夠的時間，竊聽仍然可以很輕易的取得 EPC 碼。如果能將 EPC 碼將某種方式加以編碼，雖不能阻止竊聽，但也可以防止惡意使用者取得有用的資訊。

3.1.3 偽裝(Masquerade)：

攻擊者會欺騙系統，非法取得資源。在 Auto-ID 中此為最嚴重的問題，因為惡意使用只需持有一台讀碼機，或是相容的產品。就可以直接收集到所有接收範圍內應答器的 EPC 碼。或是於取得 EPC 碼後，複製一個具有同樣 ECP 碼的應答器用來干擾系統的運作。

3.1.4 重送攻擊(Replay)：

攻擊者將網路截取的某些通訊內容再重新發送，最常見的莫過於重新發送認證資訊以假冒合法的使用者。此類攻擊便是傷害使用者身份驗證系統的功能。而在 Auto-ID 的系統中，如果只是單純的將 EPC 碼加以編碼、重組、或是

加密。惡意使用者仍可以將之固定的訊息錄製並進行重送攻擊。雖然惡意使用者不能輕易的得知內容，但如果是應用於即時倉儲管理的 Auto-ID 系統，惡意使用者就可以使用重送攻擊來製造某項物件還在倉庫，但其實該物件已經不存在的假象。

3.1.5 訊息竄改(Message Modification)：

攻擊者企圖竄改無線網路通訊內容，此類攻擊主要是傷害資料的完整性。而在原有的 Auto-ID 環境中，會發生的狀況是惡意使用者使用收集到的 EPC 碼後加以修改，製造出某個不存在的物件或是擾亂原有的物件。這點是無法預防與阻止的。

3.1.6 阻斷服務(Denial-of-Services)：

阻斷服務是最常見的攻擊手段之一。因為它會對遠端系統進行攻擊，藉此達到使系統無法提供服務或是使系統降低服務品質。而這樣的問題，事實上存在於所有的無線網路系統，在 Auto-ID 的系統中。由於大部份的資料都是由讀碼器向著應答器。在碰到惡意的干擾使用時。就確實只能比雙方的敏感度與訊號強度了。而阻斷服務的另一個目地是要消耗能源。這點對 Auto-ID class 0 的應答器是不起作用的，因為它的能量來源是來自天線收到訊號時所產生的電場，可說是半永久性的。

由以上的分析可得知，原始的 Auto-ID 在先天上對惡意使用者攻擊是很脆弱的。但它的 Class 0 應答器因受成本的考量，不可能具有太多的計算能力與儲存空間。也因此無法像其他的無線網路環境，像是 IEEE 802.11、GSM 等，可以在傳輸的二端使用昂貴的運算晶片來完成加解密與金鑰的傳遞以提供傳輸上的保障。

3.2 安全的 Auto-ID 傳輸環境。

為了解決 3.1Auto-ID 系統所面臨的安全議題，在本論文中將在第二章所提

出的 TEA 引入 Auto-ID 的傳輸過程中。本論文將以外加模組並與原來的運作模式相同的前提下，達成在 Auto-ID 的環境中完成安全傳輸的目地。本論文提出將原有的應答器架構上加入一加密電路，並定義一以原有的叫號 ID 基礎上的加密內容。另外，在與原架構相容的前提下，修改了讀碼器的運作模式以期能跟與修改過後的應答器相容。為此，提出一個適用於本架構的新操作協定也是本論文為完成此系統所不可少的。這以上三者的配合，可使改良後的 Auto-ID 傳輸環境能免於安全上的顧慮，同時達到應答器低成本的要求。

3.2.1 安全傳輸的實現

由於本論文選擇了使用 TEA 作為本系統中的加/解密演算法，且設計目地為可以附加在原有 Auto-ID Class0 的原有架構上。選用 TEA 演算法及稱為加密電路而不稱為加密晶片的原因有二，首先為忠實呈現 TEA 演算法的特色—只使用了加法與位移—這樣的特色使得整個 TEA 的電路可以直接使用數位電路的邏輯閘設計例如全加器、暫存器、位移器來達成，而不需要使用較高階、具有乘法與指數功能的運算晶片。在一般的情沿下，單一乘法器所需要的電晶體數量就數倍於同位元數的全加器。而減少使用的羅輯閘數量是減少生產與設計成本的保證。第二個原因是為了要達成 TEA 演算法的電路可以在很小的 DICE 上設計完成，可以在不改變原有的應答器核心 IC 的 DICE 大小下將原本的核心以指令集的方式將其與核心 IC 電路整合設計一起。以下將針對整個系統的各個元件設計作詳細說明:

3.2.2 應答器的設計

由於在前文已提及應答器是由四個部份所組成的，以下就一一針對各部份元件的設計提出修改，並將本論文所提出的加密模組晶片加入應答器的設計中。

3.2.2.1 核心晶片:

為了因應本改良系統的設計。本系統在核心 IC 中設置有一隨機存取記憶體區塊，用以儲存 ID0—這是原始規格中定義為當接收器收到重設的訊號時，會由亂數產生器產生的呼叫代號。另一個在 RAM 區儲存的資料則為本論文所提

出的加密後的資料碼 ID3。ID3 的資料格式將在下一節中仔細的介紹。RAM 區尚有一個存放目前工作模式的 sta 暫存器。而在核心晶片中另有一亂數產生器 rnd。此外還有一個開關電路 sw，用以關啟或關閉外部的電力供應來源(P)。而在 ROM 區有 ID1,ID2，以及本論文所特別提出的 Key--用於加密時所需的資料。Key 值是出廠時，依據 ID1 與 key ring serial 所對應的金鑰值，換言之。具有同樣 ID1 與同一個 Key ring serial 的應答器具有同樣的 key 值。圖 14 為應答器核心晶片的示意圖。

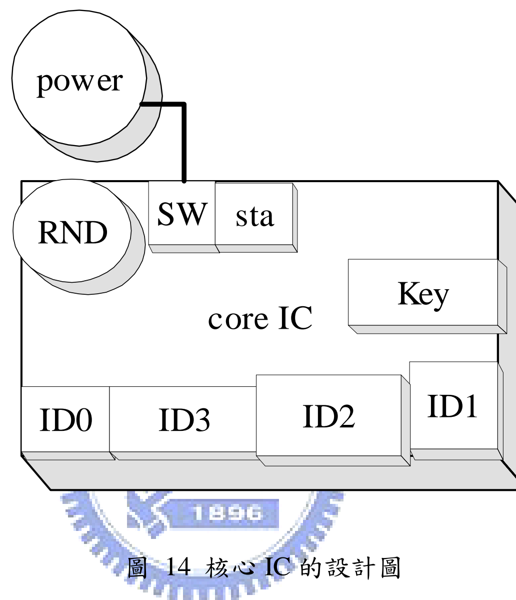


圖 14 核心 IC 的設計圖

圖示說明:

- ID0: class 0 應答器所定義的一亂數，在每次呼叫後會重設
- ID1:一個以該應答器的 EPC 碼為基本所產生的固定值
- ID2:該應答器所含的 ECP 碼
- ID3: 本論文所提出的一個數值，為經加密後的數值，會在每次叫號完成後變動。
- RND:亂數產生器
- Sw:外接電源控制開關
- Sta:工作模式暫存器
- Key: 加密用的金鑰值。

3.2.2.2RF 模組:

在應答器中原有之 RF 模組，增設有 RDY 腳位與核心 IC 相連接。在 RDY 設為低位元時會進行接收及傳送的動作。而 RDY 腳位設為高位元時，就會暫時停止接收或是傳送(圖 12)。

3.2.2.3 底座:

在本論文中，IC 底座並不需要進行大量或是重大的重新設計，如在現實上因為加入加密模組使得整個應答器的尺寸增加時才會變更底座的大小

3.2.2.4 連接匯流排:

匯流排連接了核心 IC、加密電路模組、以及 RF 模組，在圖 12 中展示了連接匯流排與其他模組的連接方式。

3.2.2.5 加密電路模組:

此為本論文所提出的特別模組。此模組的目地在使用 TEA 演算法加密原有之資料，產生一變動的加密後數值 ID3，在這個模組中，可選擇性備有一外接電源供應線路以預防經由接收讀碼器所產生的電場不足以完成加密電路的運算。此電路的開關 sw 由核心 IC 所控制。只在每次完成 ID3 的傳遞後，才會打開並進行加密的運算。在此論文所提出的系統中，為了密碼強度的設計，TEA 演算法將被會進行 32 次的攪拌並且只需要進行加密的方向。在圖 15 中，加密模組是直接附著在核心晶片上。並直接將加密後的資料回傳核心 IC。

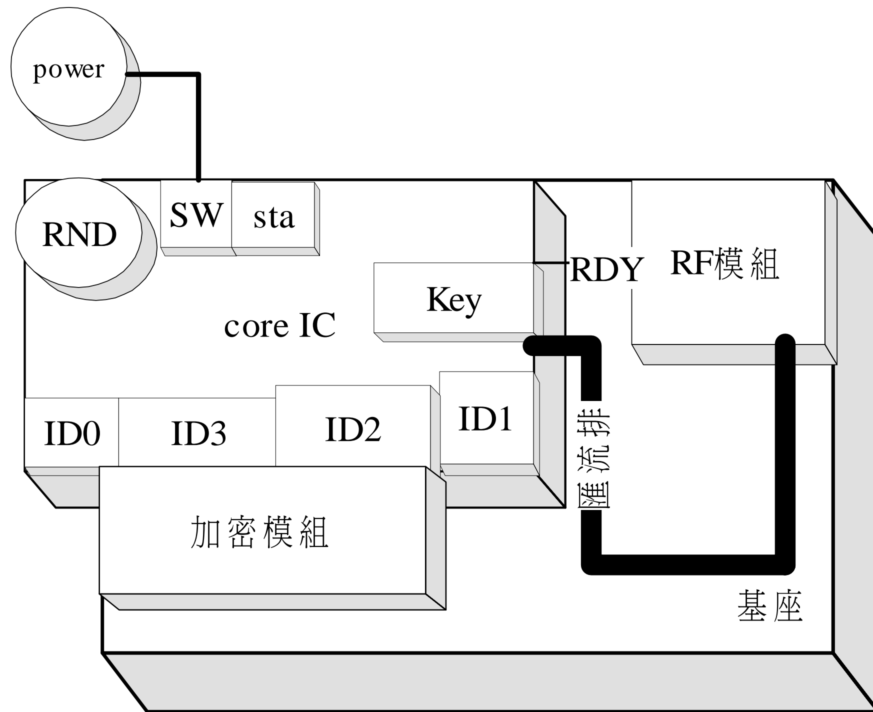


圖 15 整個應答器的組成

ES

3.2.3 讀碼器的設計

本論文所提出的系統中。讀碼器負責發送控制命令、依不同的工作模式依ID 叫號。並接收應答器所回應的資料。與原有系統不同的是，本論文所提出的讀碼器多了一個 Mode 3 的工作模式。在此模式中。讀碼器負責解密的工作。解碼器依據 ID1 叫號與 ID3 所含的 Key ring serial，找出對應的金鑰並加以解密。

3.2.4 資料結構

3.2.4.1 ID3

ID3 是本論文所提出的系統中獨有的資料。其資料格式如圖 16 所示是一長度為 138 bits 的資料:

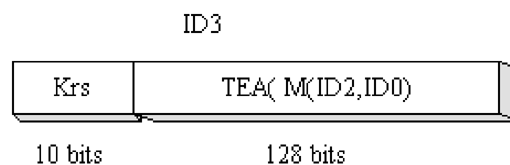


圖 16　ID3 資料格式

其中

- Krs 為 10 bit 的 key ring serial 。
- TEA(　M(ID2, ID0))為 ID2 與 ID0 的混合後經 TEA 演算法使用 Key 值作為金鑰加密 M(ID2,ID0)所產生的 128bit 數值。
- M(ID2, ID0) 為 ID0 與 ID1 的 128bit 的切割混合函數。

假設輸出值 M(ID2, ID0)=y 為 128bits，且將 y 分為 y[0]y[1]二個 64bits 的值。ID2 的 96 bits 等分為 ID2[0] ID2[1]二個 48 bits 的值，以及 ID0 10 bit 等分為 ID0[0]ID[1] 二個 5bit 的值 。則

Y[0]=ID2[0]||ID0[1]||ID0(0)||C0，Y[1]=C1[1]||ID0[0]||ID0[1]||ID2[1]

方程式 5　混合方程式 M

其中 C0 為一依據 ID2[0]與 ID0[0]所產生的 6 bit 的檢查碼。C1 為一依據 ID2[1]與 ID0[1]所產生的 6 bit 的檢查碼。將這子字串連接起來後，各自形成一個 64 bit 的字串 Y[0]與 Y[1]。而將 Y[0]與 Y[1]連接起來後，就形成了 M(ID2, ID0)。圖 17 展示了混合函式 M(ID2,ID0)如何將 ID0 與 ID1 加以混合。

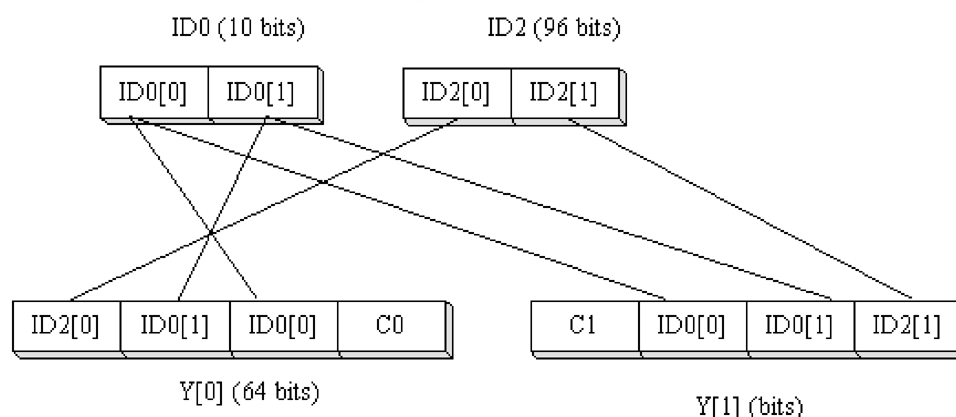


圖 17　M(ID2,ID0)

ID3 (138 bits) 比 ID2 (96 bits) 多了 42 bit . ID3 會成為 128 bit 的原因是因為 TEA 裏每次都是以 64 個 bit 為一個區塊(block)作區塊加密。所以就將原來的 96 個 bit 擴充為 128 個 bit。而混合函式 M 又適當的將亂數 ID0 混合於 ID2

這個固定值，使得 TEA (M (ID2,ID0)) 在每次 ID0 RESET 後都會重新產生。讓每次的叫號後 ID3 值都會不同。

3.2.4.2 key

Key 值的資料格式如圖 18 所示為一 138 bit 的值。具有以下五個部份。分別為 Key ring serial, k[0], k[1],k[2],k[3] 這 5 個元件。而 Key ring serial 則會依出廠的 tag 批號或是日期的不同而不同。而 k[0][,k[1],k[2],k[3]這四個子金鑰 (subkey) 為加密使用的金鑰等四分切為各 32bit 的值，其目地在提供 TEA 演算法時所需要的 4 個子金鑰值。key 值在每個應答器出廠時就已經燒錄在內，為一固定值。

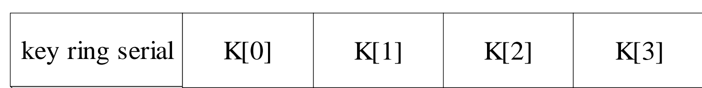


圖 18Key 值的格式設定

本論文所提出的系統，在金鑰的傳遞上是使用金鑰環(Key ring) 的觀念。每個 key ring 有一個 key ring serial，而每一 key ring 對映一 1024 欄的 table，裏頭有對應於這 1024 個 ID1 值所使用的金鑰。這些 Key ring 是儲存於讀碼器中。而 key ring 的生成與維護，是由生產此具有加密傳輸功能讀碼器與應答器的製造商來維護。並確保經授權的讀碼器使用者可以經由安全的網路連線或是其他安全的方式來取得這些 Key Ring。

當讀碼器完成某一 ID1 的叫號後，將所收集到所有 D3 值作解析。由於在 ID3 中以明文傳遞了 Key ring serial，讀碼器將比對現有的 Key ring serial。如果目前在讀碼器中有相對應的 key ring serial，則根據叫號的 ID1 值找出相對應的金鑰。

如果所收到的 ID3 值所含的 key ring serial 值，並不在讀碼器現存的 key ring serial 清單裏。由於 Auto-ID 的定義，不論是 Savant 或是讀碼器都必需連接在 TCP/IP 網路上。如此可以讓讀碼器可以直接，或是經由 savant 經由網路取得對應的 key ring。這樣的金鑰傳遞過程可以透過現有的多樣技術來達成安

全傳遞的目地，例如使用適當的認證程序以防止未經授權的讀碼機取得 key rings，或是防止以加密儲存在讀碼機中的 key ring 被解密還原。

3.2.5 安全資料傳輸的運作方式

3.2.5.1 mode 3 的運作流程。

假設一讀碼器與以 Mode 3 的方式收集應答器的資料，其傳遞過程如下

- Step1: 讀碼機廣播設定運作狀態為 Mode 3。在應答器收到此控制指令後。將 IC 中的 sta 設定為 3 (sta=3)；
- Step2: 而讀碼機在廣播工作模式封包後一段時間，就依據某種法則，例如二元樹-開始叫號(ID1 tree traversal(n) ；
- Step3: 而應答器在接收到與其 ID1 同號的叫號封包後，延遲一亂數時間後將 ID3 的資料送至讀碼機) (IF ID 1 tree traversal(ID1)= ID1, transmit ID3) ；
- Step 4: 讀碼機在收到 ID3 後，將其存入一 Buffer 以供後續的程式進行解密；
- Step 5: 讀碼器完成存入後，對同一 ID1 的應答器發出 RESET 的控制封包(RESET) ；
- Step 6: 應答器於收到此重設封包後，首先重設 ID0，接下來開啟 sw 以啟動外接電源(選用)來進行加密電路的演算。產生下一次呼叫的 D3 值。並將 ID0 與 ID3 存入 IC 的 RAM 區裏。完成以上動作後。關閉 sw ，並重設 RF 單元上的 RDY 腳位。(rdy=1, reset ID0 ,sw=1 ,generate D3 , save ID0 , ID3, then sw=0, rdy=0) ；
- Step7:準備接收下一次的呼叫。而讀碼器在送出 RESET 信號後。就重覆進行下一個 ID1 值的叫號流程。直到所有 ID1 值都叫號完為止。

在圖 19 中詳細的將 Mode 3 的流程進行了描述：

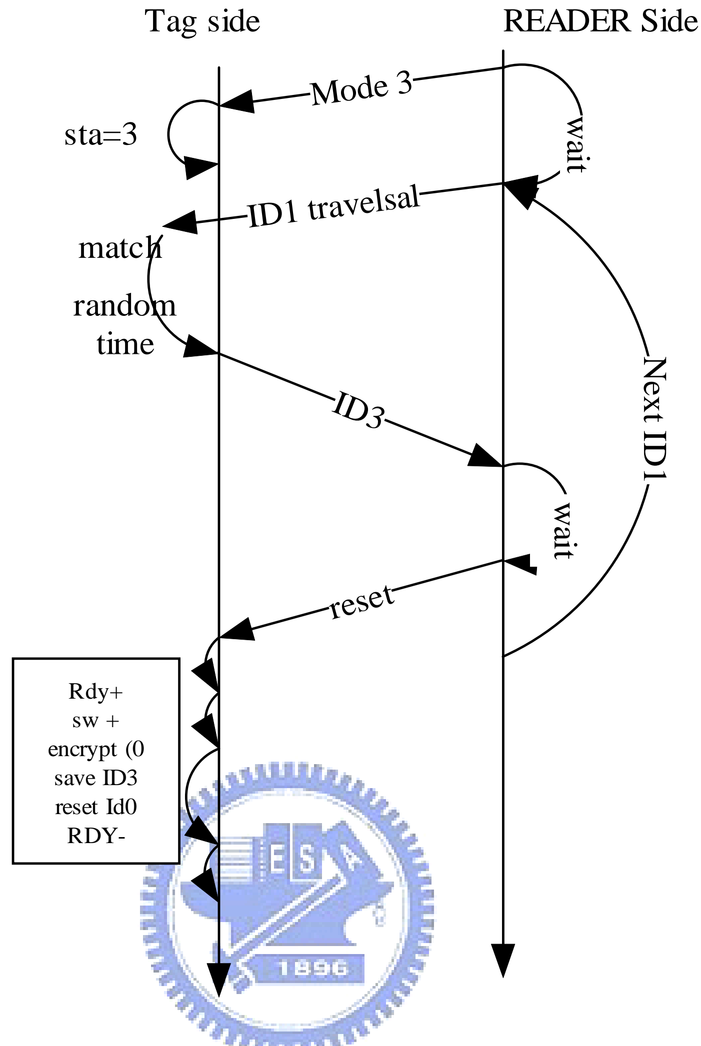


圖 19 Mode 3 的通信協定圖

當讀碼器接收 ID3 資料後，將進行以下步驟：

- Step1:當讀碼器接收了來自應答器的 ID3 資料後，首先經由 ID3 的前 10 個 bit 尋找對應的 key ring，
- Step2:並由該對應的 Key table 中，尋找對應 ID1 的金鑰。將 D3 的後 128 位元以所查詢得到的金鑰進行解密。得到 Y = M(ID2,ID0)。
- Step3:將 Y 的最前 48bit (ID2[0]) 與最後的 48bit (ID2[1]) 組合就可以得到 ID2。也就是 EPC 碼。

3.2.5.2 金鑰的傳遞與管理

由於在本系統中，金鑰的管理是透過所謂的實體傳遞(physical delivery)的方式來傳達。在應答器與讀碼器的任何通訊封包中，並未有任何的金鑰傳送，

所以本系統可以完全防止惡意使用者使用竊聽的手段得到金鑰。

為了安全起見，一個機制是 key ring 的有效時間設計，在生產批號相同的應答器上，可以設定某一 key ring 於何時過期。而這組 key ring 一但過期後，就算惡意使用者取得了，在有效期間過後就再也無效了。

而在取得key ring 的過程中，由於Auto-ID本身就定義讀碼器或是SAVANT必需連接在 TCP/IP 網路上，所以在製造商端就必需使用適當的認證程序以防止未經授權的讀碼機取得 key rings。或是在傳輸的過程中使用 SSL 加密、亦或是直接傳送編碼後的 key rings 到讀碼器，這些手段都可以確保讀碼器在取得 key rings 的過程中不會將 key rings 洩漏出去。而在讀碼器上的 key rings 儲存設計，由於在現實中，已經有很多有效的設計來保護類似的資料，例如 Smart Card ，所以儲存在讀碼器中的 Key rings 可以得到相當的保護。在此就直接引用該類技術以確保金鑰在傳輸的過中不易被竊取。

從應答器上取得該應答器所含金鑰的可能性反而是比較值得擔心的部份，其弱點在同一批號或是同一段時間製造的應答器皆使用同一組 key ring。如果使用者將應答器送去作反向工程以取得存在 ROM 裏的 Key 值，使用電子顯微鏡去觀察是有可能得到此應答器的金鑰，但反向工程曠日廢時。當解出這個應答器的金鑰時，該組 key rings 已經失效了。當然，較為積極的作法是設計一個見光死的機制，一但偵測到進行反向工程，就將 ROM 裏的所有資料，包括 ID1, ID2, 與 Key 全部洗掉，以防止金鑰與 EPC 碼被惡意使用者得知。

整個金鑰的傳遞與預防措施如圖 20 所示：

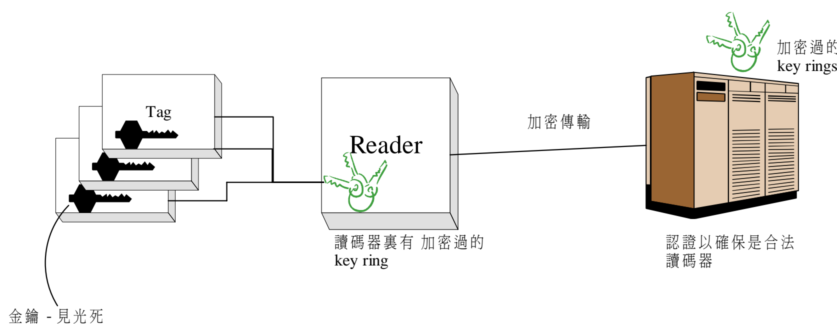


圖 20 金鑰管理機制

第四章 安全性分析

4.1 與原有系統比較

茲將改良後架構與在 3.1 章中 Auto-ID 所提出的安全問題，作一比較分析。

4.1.1 竊聽(Eavesdropping)：

竊聽可能是最簡單、也是最有效的攻擊方式，因為它不會留下任何線索。Auto-ID 本身的特性為讀碼機會發出控制訊號 0。此為較長時間的傳輸。而在應答器回應叫號的過程中，是以二個天線在很短的時間內發出資料。然後就保持在聆聽的狀態。雖然這樣增加了使用者竊聽有價值資料(EPC) 的難度。但如果竊聽設備非常靠近一個應答器，且有足夠的時間，竊聽的方式仍然可以很輕易的取得 EPC 碼。而在改良後的架構中，對於竊聽雖仍無法防止，但是因為 EPC 碼已經被加密過了。所以就算惡意使用者取得了應答器所發出的資料，在無法取得資料的混合與金鑰的情況下，也只是無意義的亂碼。而如果使用者想要在無金鑰的環境下破解 TEA 演算法，是非常困難的。在 2.2.3 裏已經有討論過 TEA 演算法的強度。

4.1.2 偽裝(Masquerade)：

攻擊者會欺騙系統，非法取得資源。在原有 Auto-ID 中此為最嚴重的問題，因為惡意使用只需持有一台讀碼機，或是相容的產品。就可以直接收集到所有接收範圍內應答器的 EPC 碼。或是取得 EPC 碼後，複製一個具有同樣 ECP 碼的應答器用來干擾系統的運作。而在改良過的 auto-ID 系統中，由於讀碼器需要透過 SAVANT 或是直接向 Key ring 提供者持續索取 Key rings，而在這索取的過程中可以加入現行的認證與傳輸機制來保證該讀碼器為合法的讀碼機。而非合法的讀碼機如果無法取得 Key Ring 的話，就算收集了再多的 ID3 資料。也只是一堆無意義的亂碼。

4.1.3 重送攻擊(Replay)：

攻擊者將網路截取的某些通訊內容再重新發送，最常見的莫過於重新發送認證資訊以假冒合法的使用者。此類攻擊便是傷害使用者身份驗證系統的功能。而在 Auto-ID 的系統中，如果只是單純的將 EPC 碼加以編碼、重組、或是加密，惡意使用者仍可以將之固定的訊息錄製並進行重送攻擊。雖然惡意使用者不能輕易的得知內容，但如果是應用於即時倉儲管理的 Auto-ID 系統，惡意使用者就可以使用重送攻擊來製造某項物件還在倉庫,但其實已經不在的假象。

在改良過後的 Auto-ID 系統中，由於 ID3 的主要成分是 M(ID2,ID0)。其中 ID0 是一個亂數值，而且是每次叫號完成就會加以重設的值，非常適合拿來作 nonce 來攪亂 ID3，而適當又簡易的 M 設計可以讓輸出值在不同的 ID0 時可以得到不同的結果。面對這樣的一個系統設計，惡意使用者將無法使用重送攻擊來假冒應答器，達成偷竊物件的成果。

4.1.4 訊息竄改(Message Modification)：

攻擊者企圖竄改無線通訊內容，此類攻擊主要是傷害資料的完整性。而在原有的 Auto-ID 環境中,會發生的狀況是惡意使用者使用收集到的 EPC 碼後加以修改,製造出某個不存在的物件或是擾亂原有的物件。這點在原有的 Auto-ID 環境下是無法預防與阻止的。

在改良式的 auto-ID 環境下，由於每個應答器內都必需要內建有適合該應答器的金鑰值與對應該金鑰值的 key ring Serial。在惡意使用者未能取得 key ring 的情況下，光是憑 EPC 碼就要造出不存在的物件是不可能的。而未經正確加密的 ID3 資料也會被讀碼器捨棄。所以惡意使用者在改良過的系統上不能竄改資料。

4.1.4 阻斷服務(Denial-of-Services)：

阻斷服務是最常見的攻擊手段之一。因為它會對遠端系統進行攻擊，藉此達到使系統無法提供服務或是使系統降低服務品質。而這樣的問題，事實上存在於所有的無線網路系統，在 Auto-ID 的系統中，由於大部份的資料流向都是

由讀碼器向著應答器，在碰到惡意的干擾使用時。就確實只能比雙方的敏感度與訊號強度了。而阻斷服務的另一個目地是要消耗能源，這點對 Auto-ID class 0 的應答器是不起作用的，因為它的能量來源是來自天線收到訊號時所產生的電場，可說是半永久性的。

表 4　Auto-ID 與改良後 Auto-ID 面對各種攻擊時比較

攻擊方式	Auto-ID	改良後 auto-ID
竊聽	不能防止	可以防止
偽裝	不能防止	可以防止
重送攻擊	不能防止	可以防止
訊息竄改	不能防止	可以防止
阻斷攻擊	部份防止	部份防止

表 3 將原有的系統與改良後的系統加比總結比較，由以上的分析可得知，本系統所提出的改良式 Auto-ID 架構讓 Auto-ID 在安全上大大的提升。可有效的解決原有的 auto-ID 在安全上所碰到的問題。

第五章　加密電路的實現

本節討論如何初步的要在數位電路中實現 TEA 演算法，其目地在證明如何在不量更改核心 IC 設計與不增加生產成本的情況下來將第三章所提及之加密電路以數位電路設計的方向與以實現。在本文中，使用 MAXPLUS 來進行邏輯元件的安排與設計，並使用 VHDL 元件與語法來進行模擬。

5.1 元件分析:

TEA 的運算式如方程式 1 所示

$$P1 = P1 + ((P2 << 4) + k[0])\ XOR\ (p2 + Delta(i))\ XOR\ ((p2 >> 5) + k[1]))$$

$$P2 = P2 + ((P1 << 4) + k[2])\ XOR\ (p1 + delta(i))\ XOR\ ((p1 >> 5) + k[3]))$$

為了實現 TEA 的加密電路，接著將一一對圖 21 中所使用的元件加以討論

圖 21 TEA 加密運算的元件綜覽

其中

- 32 bit 的記憶空間為存放欲加密的內容，也就是 ID2。因為 TEA 需將其割為一半，故需要二個 32bit 的存放空間。可使用 4 個 8 位元的隨機存取記憶體來達成
- 32 bit 全加器可以用 32 個全加器跟一個半加器來達成 一共需要 3 個
- 32 bit XOR 可以用 32 個 XOR 匣來達成，一共需要 3 個
- 而往左 Shit 4 位及往右 Shift 5 位的位移器可直接用 buffer 來完成，各需要一個。
- Data latch 裏有一個 counter 來計算進行的回合數。每次把 AB 交換位置後就把 counter 數減 1

以下就是各元件的數位邏輯匣圖。大多是使用 8bit 為單位，接著使用如圖 22 的 Carry Chain 的方式來實現 32bit 的電路。

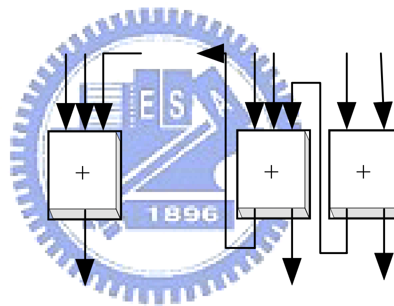


圖 22 Cary Chain 串聯

5.1.1 8bit XOR:

這是另一個易於實現的元件，處理 16 個 bit 的輸入只需要 8 個 XOR 匣，而本設計中，將使用四個 8 bit 的 XOR 匣來進行並聯。

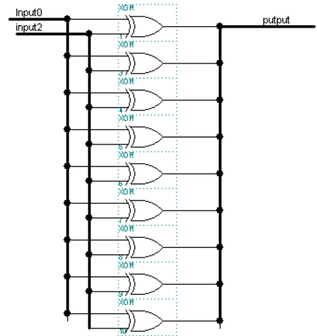


圖 23 8 bit XOR 匣

5.1.2 5 bit 的右位移與 4 bit 的左位移:

這是另一組易於實現的元件，只需要單純的將資料重排其順序就可，而 TEA 所使用的固定位移特色使得位移器設計起來更是簡易。

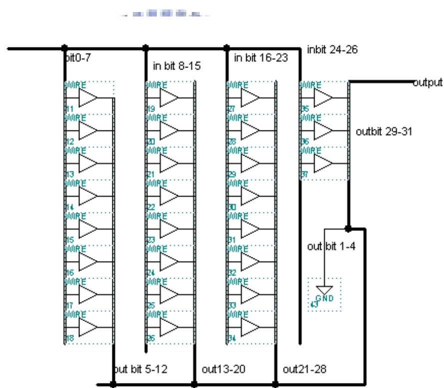


圖 24 右 5bit 位移。

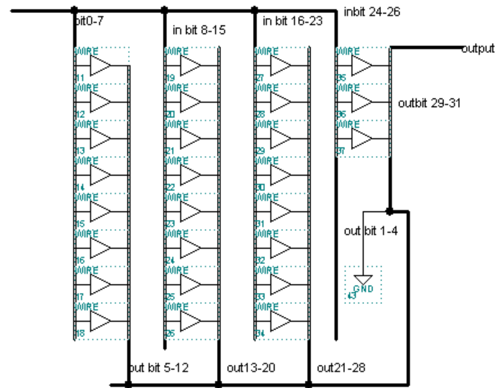


圖 25 左 4bit 位移電路

5.1.3 8 bit 的全加器

32bit 的全加器可由 4 個 8 bit 的全加器來設計完成，在圖 26 中展示了如何設計一個全加器，圖 27 則是 8 位元全加器的設計範例，其中單一全加器是以方塊來表示:

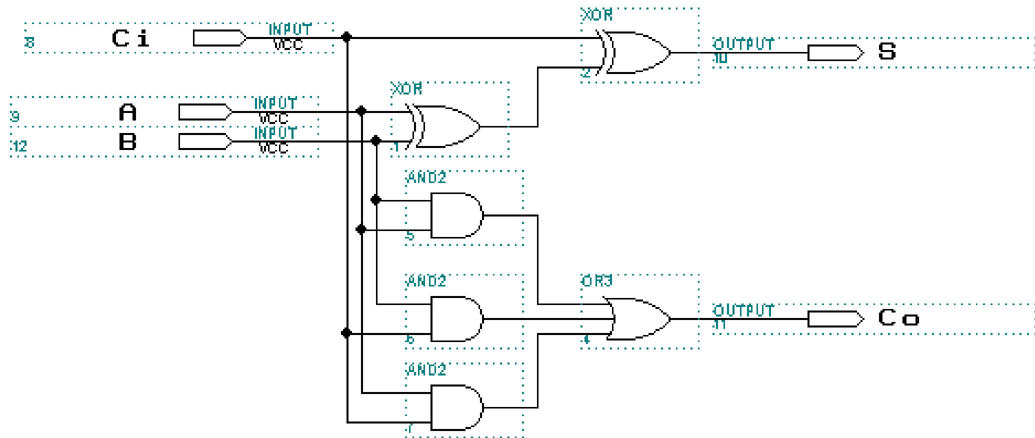


圖 26 全加器

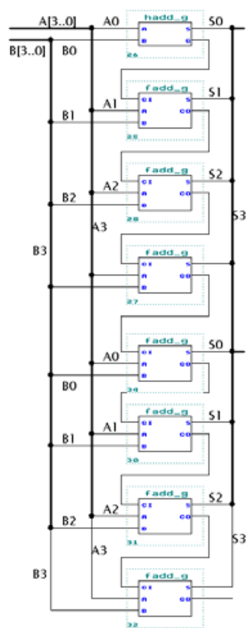


圖 27 八 bit 的全加器

5.1.4 8 bit 的暫存器:

在本圖例中，需要使用到數個 32 或是 64 位元的暫存器，而實現此暫存器可由圖 28 的 8 位元暫存器並聯數個加以實現，而事實上，在數位電位中要設計暫存器是非常容易的，只需將正反器並聯成所需的數目即可。

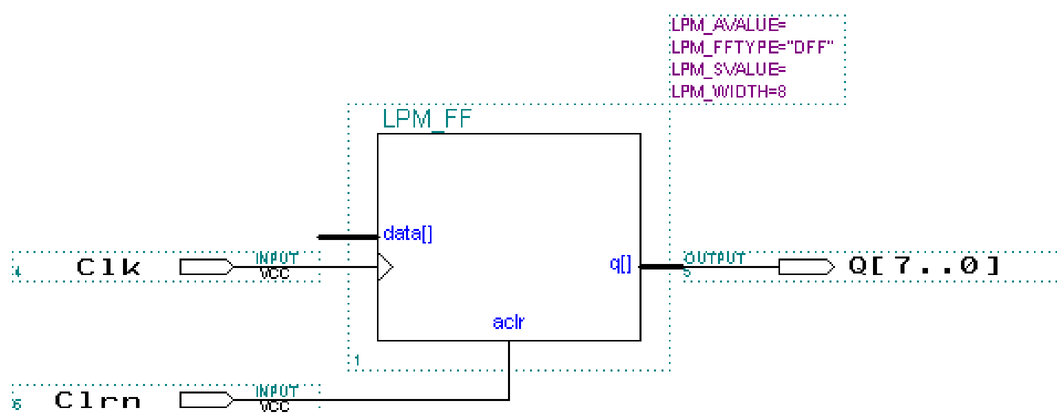


圖 28 8 位元的暫存器。

5.2 加密電路的界面設計:

在 5.1 中已經設計了 TEA 的核心加密電路，核心加密電路可以被重複的使用來進行多回合的攪拌。TEA 中可以省略 Key expansion 的電路，那是因為 TEA 使用重複的子金鑰來進行加密，所以在本設計中，所有的 TEA 加密金鑰與異變值 delta 均來自加密電路以外的部份。

5.2.1 TEA 加密核心所需的界面:

由於外部的界面電路部份為整合加密模組與核心 IC 的部份，故仍需視核心 IC 的製程方法加以適當的改變。在此不考慮製程上的限制與電力消耗的效能考量。所有需要用的到界面腳位在圖 29 中皆有定義。

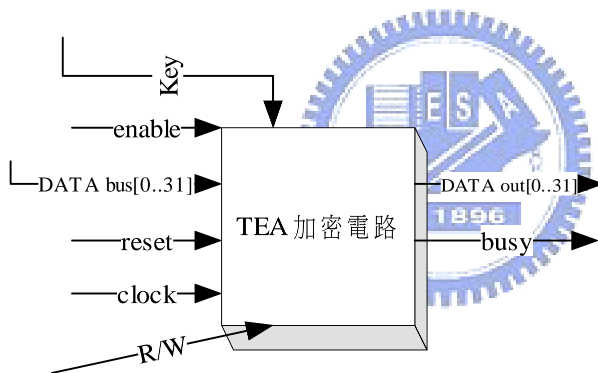


圖 29 TEA 加密電路所需 Interface 腳位

其中:

- Key:由核心 IC 而來，傳送在 TEA 加密過程中所需的 key 值
- Enable: 由此啟動 TEA
- DATA Bus [0,31]: 是一個 32 位元的匯流排，用以傳遞 3.2 中所提的 M(ID0,ID1)
- Reset: 中斷整個加密流程，清除加密電路的所有暫存器
- R/W 控制電路中的暫存器是讀入資流還是輸出資料
- Busy: 對外輸出忙碌訊號，以免外部再度輸入欲加密的資料。
- Clock: 時脈電路，用以控制正反器。

5.2.2 資料流協定

將資料寫入加密電路:

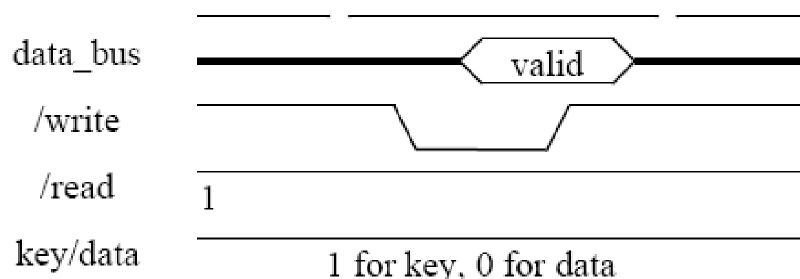


圖 30 writing cycle

將 write 腳位設為低電位，在低緣觸發時，開始輸入 DATA

從加密電路讀出資料:

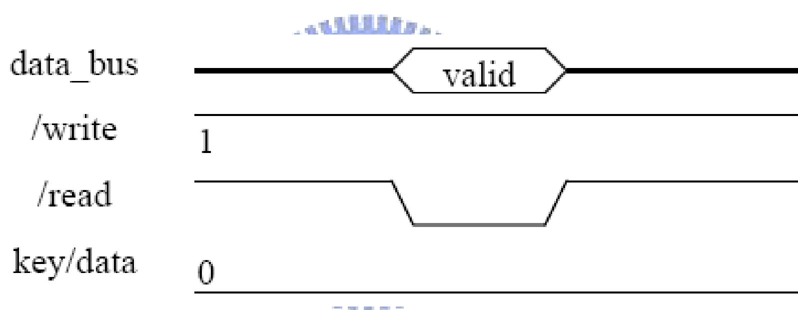


圖 31 reading cycle

圖 30 與圖 31 非常接近，當 DATA 腳位上有資料時，read 腳位在低緣觸發時，開始輸出 DATA 回核心電路。

5.2.3 電路運作的順序:

為了要更有效的使用該核心加密電路，如何正確的安排使用的流程是一門重要的學問。以下說明如何進行核心電路的流程

- step1： 由核心電路讀入所需的資料，包括 M(ID0,ID1)與 key，存在電路的暫存器中，並將 busy 腳位設為 1
- Step2:進行 32 次的攪拌(round)
- Step3:將加密後的資料由暫存器寫回核心 IC，並將 busy 腳位設為 0 。

5.2.4 結果測試:

將此核心電路以 VHDL 加以模擬，可以得到如圖 32 的時序圖

圖 32 核心電路的 Core 時序圖。

由以上的數位電數設計範例，可以大略的算出欲實現此加密電路的困難度不高，因為我們可以全部使用非時脈的元件來完成整個加密的動作。不需時脈就能進行設計，表示整個電路，只要在匯流排的大小相容的情況下，不需更動原設計的情況下就可以順利的加入原有的核心 IC，成為指令集的一部份。

5.3 成本評估:

5.3.1 Magic 的模擬

如果在要 IC 設計時，加入一個像這樣的指令集。需要多少的成本呢。目前仍未有一個明確的資料，原因是因目前仍未有真正量產化的 auto-ID 應答器晶片。雖然在目前，能推估出主要元件所使用的邏輯閘數量大約為 900 個 CELL，理論上可將其全部代換成 NAND 閘的數量。以求出建構所有主要元件需要的 NAND 閘數量，而如果有了準確的 NAND 閘數量。要推測出使用的電晶體數量就不難了。如果能取得核心 IC 的大小與規格，相信不難估出增加這些閘會影響這顆 IC 在生產上的成本的影響。以下是以 MAGIC 這設計軟體來展示 XOR(圖 33)與 4bit 全加器(圖 34)的電晶體圖層。

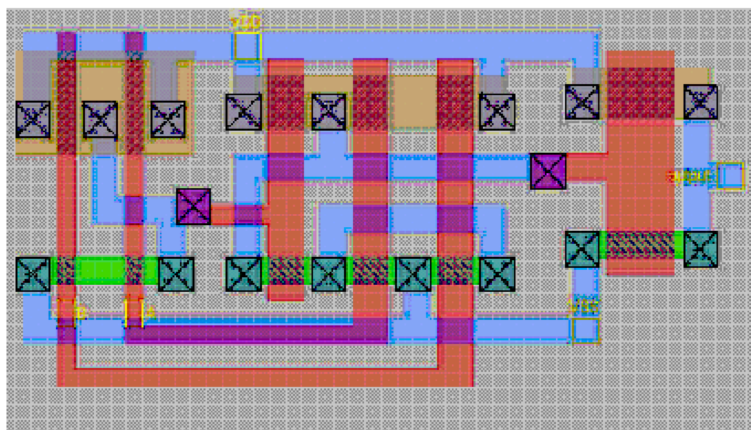
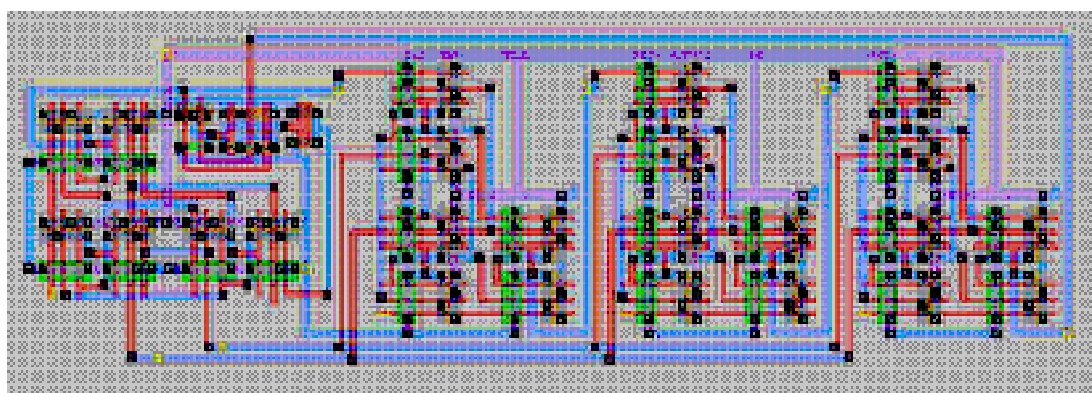


圖 33　XOR 的 IC layout



1896

圖 34 四 bits 的全加器

5.3.2　在 FPGA 的模擬:

經由 VHDL 的計算，以 FPGA 的 XC2048XL 為例來計算，可大約估算要完成該電路所需的單元數目

表 5 使用 FPGA XC2048XL 為目標以 VHDL 模擬所得資料

最佳化目標	速度	尺寸
使用單元數	911	888
每處理一次 data 所需 clock 數	16	16

由表 5 可知，欲完成該加密核心電路只需要使用 900 多個單元，對 FPGA XL2048XL 裏有將近 200,000 個 Cell 的容量，實在是只佔用小小的資源。

第六章 結語

6.1 研究價值

縱觀本論文，在第一章中，由研究目地與動機昭示了本論文的目地在提出一可行性高的低成本 Auto-ID 應答器。也如第二章所介紹的 Auto-ID 的特色與原理，了解了 Auto-ID 這樣的技術對未來的影響以及在設計 Auto-ID 應答器上的限制。第二章也對 TEA 演算法加以研究，可以得知 TEA 演算法不但具有簡易，快速，與安全的特性、以及 TEA 演算法與其他相似演算法的分析比較也證明了 TEA 是一個非常適合用於數位電路設計的演算法。在三章中，本研究提供了一個改良 Auto-ID 傳輸上安全問題的環境，在此我們針對其原本以明碼傳遞的方式，提出另一個以 TEA 為加密演算法的通訊架構，從而定義了該架構使用的資料格式、操作流程。而這個新的架構不但是具有足夠的安全性，更因為它是基於原有架構所提出的，故新的架構仍然與原有的架構相容。所以在第四章中，針對所提出的新架構與原有的架構進行安全性進行分析比較。接下來在第 5 章中，以 VHDL 的元件設計出本架構的實作，並進一步加以估算所需的單元數量，以證明本論文所提出的架構確實是符合低價可行的目地。

6.2 未來方向

由於 TEA 的安全強度在針對 related key 的攻擊上仍未能滿意。故可以考慮改用 XTEA 或是其他適合作成數電路的演算法。

由於在目前，市面上仍未有一量產的應答器晶片，固無法針對現有的 IC Layout 設計進行改造。期冀本論文，能提供在未來進行應答器時的核心 IC 設計時，能將本論文使用的電路以指令集的方式設計出真正的實作。

參考文獻

[1]Sanjay Sarma, Towards the 5 ¢ Tag, Auto-ID center, 2002

[2] Auto-ID Center, Draft protocol specification for a 900 MHz Class 0 Radio Frequency Identification Tag , Auto-ID Center, 2002

[3] 唐政, 802.11 無線區域網路通訊協定及應用,文魁資訊,1991

[4] Roger M. Needham and David J. Wheeler., Tea extensions, Technical report, Computer Laboratory, University of Cambridge, October 1997.

[5] . Biryukov and D. Wagner, Slide attacks, In Lars Knudsen, editor, Fast software encryption: 6th International Workshop, FSE'99, Rome, Italy, March 24-26, 1999

[6] Roger Fleming., An attack on a weakened version of TEA, URL http://groups.google.com/groups?selm=roger_sf-2210961222000001%40mg4-50.its.utas.edu.au.

[7] John Kelsey, Bruce Schneier, and David Wagner, Key-schedule cryptanalysis of IDEA, G-DES, GOST, SAFER, and Triple-DES, Lecture Notes in Computer Science, 1109: 237-251, 1996. ISSN 0302-9743.

[8] Julio César Hernández, José María Sierra, Arturo Ribagorda, Benjamín Ramos, and J. C. Mex-Perera. Distinguishing TEA from a random permutation: Reduced round versions of TEA do not have the SAC or do not generate random numbers. In Proceedings of the IMA Int. Conf. on Cryptography and Coding 2001, pages 374-377, 2001.

[9] Dukjae Moon, Kyungdeok Hwang, Wonil Lee, Sangjin Lee, and Jongin Lim, Impossible differential cryptanalysis of reduced round XTEA and TEA. *Lecture Notes in Computer Science*, 2365: 49-60, 2002. ISSN 0302-9743

[10] Deukjo Hong, Youngdai Ko, Donghoon Chang, Wonil Lee, and Jongin Lim. Differential cryptanalysis of XTEA. Technical Report TR03_13, Center for the Information Security and Technologies (CIST), Seoul, Korea, 2003a

[11] Wei Dai, Crypto++™ Library 5.1,cryptopp.com,2002

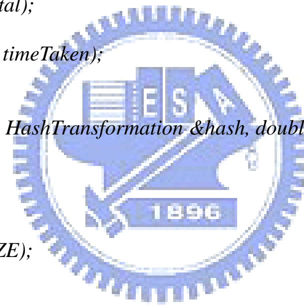
[12] William Stallings, Cryptography and network security – principles and practice, pretence HALL, ISBN 0-13-869017-0

附錄

A. Bench.cpp

```
// bench.cpp

#include "pch.h"
#include "crc.h"
#include "des.h"
#include "blowfish.h"
#include "rc5.h"
#include "tea.h"
#include "bench.h"
#include <time.h>
#include <math.h>
#include <iostream>
#include <iomanip>

USING_NAMESPACE(CryptoPP)
USING_NAMESPACE(std)
#ifdef CLOCKS_PER_SEC
static const double CLOCK_TICKS_PER_SECOND = (double)CLOCKS_PER_SEC;
#elif defined(CLK_TCK)
static const double CLOCK_TICKS_PER_SECOND = (double)CLK_TCK;
#else
static const double CLOCK_TICKS_PER_SECOND = 1000000.0;
#endif
static const byte *const key=(byte
*)"0123456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ00000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000";
static double logtotal = 0;
static unsigned int logcount = 0;
void OutputResultBytes(const char *name, unsigned long length, double timeTaken)
{
   double mbs = length / timeTaken / (1024*1024);
   cout << "<TR><TH>" << name;
   cout << "<TD>" << length;
   cout << setiosflags(ios::fixed);
   cout << "<TD>" << setprecision(3) << timeTaken;
   cout << "<TD>" << setprecision(3) << mbs << endl;
   cout << resetiosflags(ios::fixed);
   logtotal += log(mbs);
   logcount++;
}

void OutputResultOperations(const char *name, const char *operation, bool pc, unsigned long iterations, double
timeTaken)
{
   cout << "<TR><TH>" << name << " " << operation << (pc ? " with precomputation" : "");
   cout << "<TD>" << iterations;
   cout << setiosflags(ios::fixed);
   cout << "<TD>" << setprecision(3) << timeTaken;
   cout << "<TD>" << setprecision(2) << (1000*timeTaken/iterations) << endl;
   cout << resetiosflags(ios::fixed);

   logtotal += log(iterations/timeTaken);
   logcount++;
}

void BenchMark(const char *name, BlockTransformation &cipher, double timeTotal)
{
   const int BUF_SIZE = RoundDownToMultipleOf(1024U, cipher.OptimalNumberOfParallelBlocks() *
cipher.BlockSize());
   SecByteBlock buf(BUF_SIZE);
   const int nBlocks = BUF_SIZE / cipher.BlockSize();
```

```
	clock_t start = clock();

	unsigned long i=0, length=BUF_SIZE;
	double timeTaken;
	do
	{
		length *= 2;
		for (; i<length; i+=BUF_SIZE)
			cipher.ProcessAndXorMultipleBlocks(buf, NULL, buf, nBlocks);
		timeTaken = double(clock() - start) / CLOCK_TICKS_PER_SECOND;
	}
	while (timeTaken < 2.0/3*timeTotal);

	OutputResultBytes(name, length, timeTaken);
}

void BenchMark(const char *name, StreamTransformation &cipher, double timeTotal)
{
	const int BUF_SIZE=1024;
	SecByteBlock buf(BUF_SIZE);
	clock_t start = clock();

	unsigned long i=0, length=BUF_SIZE;
	double timeTaken;
	do
	{
		length *= 2;
		for (; i<length; i+=BUF_SIZE)
			cipher.ProcessString(buf, BUF_SIZE);
		timeTaken = double(clock() - start) / CLOCK_TICKS_PER_SECOND;
	}
	while (timeTaken < 2.0/3*timeTotal);

	OutputResultBytes(name, length, timeTaken);
}
```



```
void BenchMark(const char *name, HashTransformation &hash, double timeTotal)
{
	const int BUF_SIZE=1024;
	SecByteBlock buf(BUF_SIZE);
	LC_RNG rng(time(NULL));
	rng.GenerateBlock(buf, BUF_SIZE);
	clock_t start = clock();

	unsigned long i=0, length=BUF_SIZE;
	double timeTaken;
	do
	{
		length *= 2;
		for (; i<length; i+=BUF_SIZE)
			hash.Update(buf, BUF_SIZE);
		timeTaken = double(clock() - start) / CLOCK_TICKS_PER_SECOND;
	}
	while (timeTaken < 2.0/3*timeTotal);

	OutputResultBytes(name, length, timeTaken);
}

void BenchMark(const char *name, BufferedTransformation &bt, double timeTotal)
{
	const int BUF_SIZE=1024;
	SecByteBlock buf(BUF_SIZE);
	LC_RNG rng(time(NULL));
	rng.GenerateBlock(buf, BUF_SIZE);
	clock_t start = clock();

	unsigned long i=0, length=BUF_SIZE;
	double timeTaken;
	do
	{
		length *= 2;
		for (; i<length; i+=BUF_SIZE)
			bt.Put(buf, BUF_SIZE);
		timeTaken = double(clock() - start) / CLOCK_TICKS_PER_SECOND;
	}
```

```
    while (timeTaken < 2.0/3*timeTotal);

    OutputResultBytes(name, length, timeTaken);
  }

  void BenchMarkEncryption(const char *name, PK_Encryptor &key, double timeTotal, bool pc=false)
  {
    unsigned int len = 16;
    LC_RNG rng(time(NULL));
    SecByteBlock plaintext(len), ciphertext(key.CiphertextLength(len));
    rng.GenerateBlock(plaintext, len);

    clock_t start = clock();
    unsigned int i;
    double timeTaken;
    for (timeTaken=(double)0, i=0; timeTaken < timeTotal; timeTaken = double(clock() - start) /
CLOCK_TICKS_PER_SECOND, i++)
            key.Encrypt(rng, plaintext, len, ciphertext);

    OutputResultOperations(name, "Encryption", pc, i, timeTaken);

    if (!pc && key.GetMaterial().SupportsPrecomputation())
    {
            key.AccessMaterial().Precompute(16);
            BenchMarkEncryption(name, key, timeTotal, true);
    }
  }

  void BenchMarkDecryption(const char *name, PK_Decryptor &priv, PK_Encryptor &pub, double timeTotal)
  {
    unsigned int len = 16;
    LC_RNG rng(time(NULL));
    SecByteBlock ciphertext(pub.CiphertextLength(len));
    SecByteBlock plaintext(pub.MaxPlaintextLength(ciphertext.size()));
    rng.GenerateBlock(plaintext, len);
    pub.Encrypt(rng, plaintext, len, ciphertext);

    clock_t start = clock();
    unsigned int i;
    double timeTaken;
    for (timeTaken=(double)0, i=0; timeTaken < timeTotal; timeTaken = double(clock() - start) /
CLOCK_TICKS_PER_SECOND, i++)
            priv.Decrypt(rng, ciphertext, ciphertext.size(), plaintext);

    OutputResultOperations(name, "Decryption", false, i, timeTaken);
  }

  void BenchMarkSigning(const char *name, PK_Signer &key, double timeTotal, bool pc=false)
  {
    unsigned int len = 16;
    LC_RNG rng(time(NULL));
    SecByteBlock message(len), signature(key.SignatureLength());
    rng.GenerateBlock(message, len);

    clock_t start = clock();
    unsigned int i;
    double timeTaken;
    for (timeTaken=(double)0, i=0; timeTaken < timeTotal; timeTaken = double(clock() - start) /
CLOCK_TICKS_PER_SECOND, i++)
            key.SignMessage(rng, message, len, signature);

    OutputResultOperations(name, "Signature", pc, i, timeTaken);

    if (!pc && key.GetMaterial().SupportsPrecomputation())
    {
            key.AccessMaterial().Precompute(16);
            BenchMarkSigning(name, key, timeTotal, true);
    }
  }

  void BenchMarkVerification(const char *name, const PK_Signer &priv, PK_Verifier &pub, double timeTotal, bool
pc=false)
  {
    unsigned int len = 16;
    LC_RNG rng(time(NULL));
```

```
	SecByteBlock message(len), signature(pub.SignatureLength());
	rng.GenerateBlock(message, len);
	priv.SignMessage(rng, message, len, signature);

	clock_t start = clock();
	unsigned int i;
	double timeTaken;
	for (timeTaken=(double)0, i=0; timeTaken < timeTotal; timeTaken = double(clock() - start) / 
CLOCK_TICKS_PER_SECOND, i++)
		pub.VerifyMessage(message, len, signature, signature.size());

	OutputResultOperations(name, "Verification", pc, i, timeTaken);

	if (!pc && pub.GetMaterial().SupportsPrecomputation())
	{
		pub.AccessMaterial().Precompute(16);
		BenchMarkVerification(name, priv, pub, timeTotal, true);
	}
}

void BenchMarkKeyGen(const char *name, SimpleKeyAgreementDomain &d, double timeTotal, bool pc=false)
{
	LC_RNG rng(time(NULL));
	SecByteBlock priv(d.PrivateKeyLength()), pub(d.PublicKeyLength());

	clock_t start = clock();
	unsigned int i;
	double timeTaken;
	for (timeTaken=(double)0, i=0; timeTaken < timeTotal; timeTaken = double(clock() - start) / 
CLOCK_TICKS_PER_SECOND, i++)
		d.GenerateKeyPair(rng, priv, pub);

	OutputResultOperations(name, "Key-Pair Generation", pc, i, timeTaken);

	if (!pc && d.GetMaterial().SupportsPrecomputation())
	{
		d.AccessMaterial().Precompute(16);
		BenchMarkKeyGen(name, d, timeTotal, true);
	}
}

void BenchMarkKeyGen(const char *name, AuthenticatedKeyAgreementDomain &d, double timeTotal, bool pc=false)
{
	LC_RNG rng(time(NULL));
	SecByteBlock priv(d.EphemeralPrivateKeyLength()), pub(d.EphemeralPublicKeyLength());

	clock_t start = clock();
	unsigned int i;
	double timeTaken;
	for (timeTaken=(double)0, i=0; timeTaken < timeTotal; timeTaken = double(clock() - start) / 
CLOCK_TICKS_PER_SECOND, i++)
		d.GenerateEphemeralKeyPair(rng, priv, pub);

	OutputResultOperations(name, "Key-Pair Generation", pc, i, timeTaken);

	if (!pc && d.GetMaterial().SupportsPrecomputation())
	{
		d.AccessMaterial().Precompute(16);
		BenchMarkKeyGen(name, d, timeTotal, true);
	}
}

void BenchMarkAgreement(const char *name, SimpleKeyAgreementDomain &d, double timeTotal, bool pc=false)
{
	LC_RNG rng(time(NULL));
	SecByteBlock priv1(d.PrivateKeyLength()), priv2(d.PrivateKeyLength());
	SecByteBlock pub1(d.PublicKeyLength()), pub2(d.PublicKeyLength());
	d.GenerateKeyPair(rng, priv1, pub1);
	d.GenerateKeyPair(rng, priv2, pub2);
	SecByteBlock val(d.AgreedValueLength());

	clock_t start = clock();
	unsigned int i;
	double timeTaken;
	for (timeTaken=(double)0, i=0; timeTaken < timeTotal; timeTaken = double(clock() - start) /
```

```
CLOCK_TICKS_PER_SECOND, i+=2)
        {
                d.Agree(val, priv1, pub2);
                d.Agree(val, priv2, pub1);
        }

        OutputResultOperations(name, "Key Agreement", pc, i, timeTaken);
    }

    void BenchMarkAgreement(const char *name, AuthenticatedKeyAgreementDomain &d, double timeTotal, bool pc=false)
    {
        LC_RNG rng(time(NULL));
        SecByteBlock spriv1(d.StaticPrivateKeyLength()), spriv2(d.StaticPrivateKeyLength());
        SecByteBlock epriv1(d.EphemeralPrivateKeyLength()), epriv2(d.EphemeralPrivateKeyLength());
        SecByteBlock spub1(d.StaticPublicKeyLength()), spub2(d.StaticPublicKeyLength());
        SecByteBlock epub1(d.EphemeralPublicKeyLength()), epub2(d.EphemeralPublicKeyLength());
        d.GenerateStaticKeyPair(rng, spriv1, spub1);
        d.GenerateStaticKeyPair(rng, spriv2, spub2);
        d.GenerateEphemeralKeyPair(rng, epriv1, epub1);
        d.GenerateEphemeralKeyPair(rng, epriv2, epub2);
        SecByteBlock val(d.AgreedValueLength());

        clock_t start = clock();
        unsigned int i;
        double timeTaken;
        for (timeTaken=(double)0, i=0; timeTaken < timeTotal; timeTaken = double(clock() - start) /
CLOCK_TICKS_PER_SECOND, i+=2)
        {
                d.Agree(val, spriv1, epriv1, spub2, epub2);
                d.Agree(val, spriv2, epriv2, spub1, epub1);
        }

        OutputResultOperations(name, "Key Agreement", pc, i, timeTaken);
    }

    //VC60 workaround: compiler bug triggered without the extra dummy parameters
    template <class T>
    void BenchMarkKeyed(const char *name, double timeTotal, T *x=NULL)
    {
        T c;
        c.SetKeyWithIV(key, c.DefaultKeyLength(), key);
        BenchMark(name, c, timeTotal);
    }

    //VC60 workaround: compiler bug triggered without the extra dummy parameters
    template <class T>
    void BenchMarkKeyedVariable(const char *name, double timeTotal, unsigned int keyLength, T *x=NULL)
    {
        T c;
        c.SetKeyWithIV(key, keyLength, key);
        BenchMark(name, c, timeTotal);
    }

    //VC60 workaround: compiler bug triggered without the extra dummy parameters
    template <class T>
    void BenchMarkKeyless(const char *name, double timeTotal, T *x=NULL)
    {
        T c;
        BenchMark(name, c, timeTotal);
    }

    //VC60 workaround: compiler bug triggered without the extra dummy parameters
    template <class SCHEME>
    void BenchMarkCrypto(const char *filename, const char *name, double timeTotal, SCHEME *x=NULL)
    {
        FileSource f(filename, true, new HexDecoder());
        typename SCHEME::Decryptor priv(f);
        typename SCHEME::Encryptor pub(priv);
        BenchMarkEncryption(name, pub, timeTotal);
        BenchMarkDecryption(name, priv, pub, timeTotal);
    }

    //VC60 workaround: compiler bug triggered without the extra dummy parameters
    template <class SCHEME>
    void BenchMarkSignature(const char *filename, const char *name, double timeTotal, SCHEME *x=NULL)
```

```
{
   FileSource f(filename, true, new HexDecoder());
   typename SCHEME::Signer priv(f);
   typename SCHEME::Verifier pub(priv);
   BenchMarkSigning(name, priv, timeTotal);
   BenchMarkVerification(name, priv, pub, timeTotal);
}

//VC60 workaround: compiler bug triggered without the extra dummy parameters
template <class D>
void BenchMarkKeyAgreement(const char *filename, const char *name, double timeTotal, D *x=NULL)
{
   FileSource f(filename, true, new HexDecoder());
   D d(f);
   BenchMarkKeyGen(name, d, timeTotal);
   BenchMarkAgreement(name, d, timeTotal);
}

void BenchMarkAll(double t)
{
#if 1
   logtotal = 0;
   logcount = 0;

   cout << "<TABLE border=1><COLGROUP><COL align=left><COL align=right><COL align=right><COL
align=right>" << endl;
   cout << "<THEAD><TR><TH>Algorithm<TH>Bytes Processed<TH>Time Taken<TH>Megabytes(2^20
bytes)/Second\n<TBODY>" << endl;

   BenchMarkKeyless<CRC32>("CRC-32", t);
   BenchMarkKeyed<DES::Encryption>("DES", t);
   BenchMarkKeyed<RC5::Encryption>("RC5 (r=16)", t);
   BenchMarkKeyed<Blowfish::Encryption>("Blowfish", t);
   BenchMarkKeyed<TEA::Encryption>("TEA", t);
   cout << "</TABLE>" << endl;

   cout << "<TABLE border=1><COLGROUP><COL align=left><COL align=right><COL align=right><COL
align=right>" << endl;
   cout << "<THEAD><TR><TH>Operation<TH>Iterations<TH>Total Time<TH>Milliseconds/Operation" << endl;
   cout << "<TBODY style=\"background: yellow\">" << endl;
   BenchMarkKeyAgreement<XTR_DH>("xtrdh171.dat", "XTR-DH 171", t);
   BenchMarkKeyAgreement<XTR_DH>("xtrdh342.dat", "XTR-DH 342", t);
   BenchMarkKeyAgreement<DH>("dh1024.dat", "DH 1024", t);
   BenchMarkKeyAgreement<DH>("dh2048.dat", "DH 2048", t);
   BenchMarkKeyAgreement<LUC_DH>("lucd512.dat", "LUCDIF 512", t);
   BenchMarkKeyAgreement<LUC_DH>("lucd1024.dat", "LUCDIF 1024", t);
   BenchMarkKeyAgreement<MQV>("mqv1024.dat", "MQV 1024", t);
   BenchMarkKeyAgreement<MQV>("mqv2048.dat", "MQV 2048", t);
   cout << "</TABLE>" << endl;
   cout << "Throughput Geometric Average: " << setiosflags(ios::fixed) << exp(logtotal/logcount) << endl;
   time_t endTime = time(NULL);
   cout << "\nTest ended at " << asctime(localtime(&endTime));
#endif

   }
```