BZA05 - MNG

Model 2019

Bezpečná zařízení

Část 5

Útoky na postranní kanály

Post 18/19

Souhrnné materiály

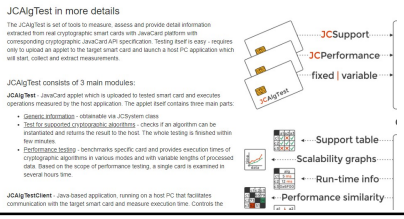
Ver 0.1

Útoky na postranní kanály

Kamil Malinka
BZA 2021

Malý návrat ke smartkartám

- https://www.fi.muni.cz/~xsvenda/jcalgtest/



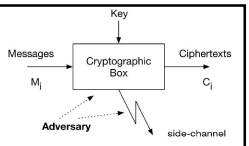
Experiment

Obsah

- Úvod do výkonové analýzy
- Časová analýza (TA)
- Výkonová analýza (PA)
- RSA algoritmus v praxi
- Další příklady
- Ochrana proti útokům na postranní kanály

- Vycházíme z knihy: Koc C.K. (2009) About Cryptographic Engineering. In: Koc C.K. (eds) Cryptographic Engineering. Springer, Boston, MA
- Kap. Side-Channel Attacks and Countermeasures

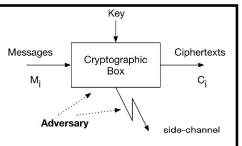
Postranní kanál



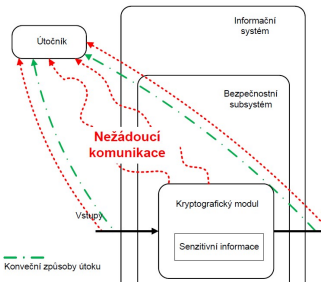
- Klasické útoky:

- Přístup k systému jako k black boxu
- Vstup, výstup
- Útok na algoritmus

Postranní kanál



- Kryptografické algoritmy musí běžet na reálných zařízeních
- A ty mají nějaké fyzikální vlastnosti
- Zařízení neúmyslně vyzařují informace týkající se kryptografických algoritmů, klíčů a zpráv
- Útok na implementaci
- K útoku tedy využíváme informace, které se nám podaří získat sledováním různých charakteristik zařízení
- Často využíváme statistiku
- Z těchto informací se snažíme zjistit nějaké informace o systému, o klíči atd.



Zdroj: Martinásek: Kryptoanalýza postranními kanály

Postranní kanál

- Provozovatel systému nikdy neví, jaké všechny postranní kanály jdou monitorovat a zneužít k útoku
- Stále přichází nové útoky

- Věděli jste, že Vás jde odposlouchávat pomoci reproduktorů?
- **Appendix A:** Mordechai Guri, Yosef Solewicz, Andrey Daidakulov, Yuval Elovici. *SPEAKE(a)R: Turn Speakers to Microphones for Fun and Profit*). 2017
- https://www.youtube.com/watch?v=ez3o8aIZCDM

8

Postranní kanál

- Typy postranních kanálů
  - Časový
  - Odběrový/výkonový
  - Elektromagnetický
  - Chybový
  - Optický
  - Akustický
  - Atd..

9

Postranní kanál

Existují dvě základní analýzy:

- jednoduchá (Simple Analysis, SA)
  - Útočník se snaží určit klíč přímo např. ze změřené spotřeby
  - Musí existovat přímá nebo nepřímá závislost proudové spotřeby na hodnotě šifrovacího klíče.- single/multishot analysis – jeden proud vs. více proudů
- diferenciální (Differential Analysis, DA)
  - Používá se tam, kde jednoduchá analýza selhává kvůli přílišnému šumu v nasnímanych datech
  - Srovnává data nasnímaná během mnoha běhů systému, ne jenjednoho
  - Využívá matematický aparát, statistické metody, neuronové sítě, atd.

10

Postranní kanál

- vs. skrytý kanál:
- Postranní je důsledkem fyzicke implementace
- Skrytý je v rámci systému, např. sdílený soubor, zátěž procesoru, registry...

11

Malý návrat do kryptografie – chybový postranní kanál

- S. Vaudenay. *„CBC Padding: Security flaws in SSL, IPSEC, WTLS“.* EUROCrypt'02
- Konkrétní útoky realizovány nad SSL, IPSec, několika webovými frameworky včetně JavaServer Faces, Ruby on Rails a ASP.NET, i SW (např. Steam klient).
- Rozšíření i na bezpečný HW
- **Appendix B** (2012, efektivní i na kryptografický HW): Romain Bardou, Riccardo Focardi, Yusuke Kawamoto, Lorenzo Simionato, Graham Steel, et al. *„Efficient Padding Oracle Attacks on Cryptographic Hardware“*. [Research Report] RR-7944, 2012, pp.19. ffhal-00691958v2f
- POZN: paper obsahuje i Bleichenbacherův útok, budete probírat v KRY

12

CBC Padding

- SSL/TLS, IPSEC, WTLS – zpráva je nejdříve předformátována
- po dešifrování je formát opět zkontrolován
  - postranní kanál
    - chybová zpráva – chyba při dešifrování
    - chybová zpráva – chybný formát
  - složitost útoku O(NbW)
    - N – počet bloků, b – počet slov v bloku, W – počet možných slov (usually 256)
      - 8 kB msg => 1000 blocks * 8 words * 256

RFC2040

- RC5-CBC-PAD, každý blok má 8 slov o 8 bitech
  - zpráva je doplněna n slovy s hodnotou n
  - získáme sekvenci bloků $x_1, \ldots, x_n$
  - šifrování $y_1=C(IV \oplus x_1)$, $y_i = C(y_{i-1} \oplus x_i)$
- dešifrování:
  - dešifrování, kontrola paddingu, odstranění paddingu
    - kontrolu korektnosti paddingu použijeme jako "věštírnu" (z angl. oracle)

Nepravděpodobný útok

- předpokládejme že šifrové bloky y_i, y_j jsou shodné
  - potom platí $y_{i-1} \oplus y_{j-1} = x_i \oplus x_j$
  - dále využijeme redundance v textu k získání x_i, x_j z y_{i-1}, y_{j-1}
  - pravděpodobnost – narozeninový teorém
    - $p \approx 1 - e^{-N.N/(2m)} = 1 - e^{-N.N/(2.8*8)}$
    - rozumnou pravděpodobnost 39% získáme až pro data o délce 32 GB

Účinný útok I

- používáme "orákulum" *O* – vrací 1, pokud má dešifrovaný text korektní padding, jinak vrací 0
- *O* je definováno šifrovým textem a IV
- chceme spočítat poslední slovo bloku *y*
  - nechť $r=r_1 \ldots r_b$ jsou náhodná slova, podvrhneme šifrovaný text $r|y$
  - pokud $O(r|y)=1$ pak $C^{-1}(y) \oplus r$ skončí s platným paddingem
  - nejpravděpodobnější padding je 1 (odpovída pouze 1 byte), můžeme ověřit použitím $r'= r_1' \ldots r_b', r_b = r_b'$

Účinný útok II

- $O(r|y) = 1 \Rightarrow C^{-1}(y) \oplus r = 01$ resp. 02 atd..

$C^{-1}(y_i) = (y_{i-1}) \oplus p_i$ $\quad C^{-1}(y) \oplus r = 01$

$((\underline{y_{i-1}}) \oplus p_i) \oplus \underline{r} = \underline{01}$

Účinný útok III

- nyní chceme dešifrovat blok – block decryption oracle
  - $a = a_1 \ldots a_b <= C^{-1}(y)$
  - použijeme *O* k získání posledního slova a_b
  - co s a_{b-k}? – a_j
    - začátek je stále náhodný $r_1 \ldots r_{j-1}$, konec je $r_k=a_k \oplus (b-j+2)$
    - použijeme padělaný šifrovaný text $r|y$ pro dešifrování
    - druhý blok je $r \oplus a$, tedy poslední b-j+1 blok je b-j+2
    - pokud $O(r|y) = 1$ tak víme, že $r_j \oplus a_j = b-j+2$ => tedy získáme a_j

Příklad

b=8 W=256 $y_i = C(y_{i-1} \oplus x_i)$, $x_i = C^{-1}(y_i) \oplus y_{i-1}$

- C = 3c 68 74 6d 6c 3e 3c 68 | 65 61 64 3e 3c 6c 69 6e
- r = 00 00 00 00 00 00 00 00 = $r_1\ r_2\ r_3\ r_4\ r_5\ r_6\ r_7\ r_8$
- zkoušíme
  - *O*(65 61 64 3e 3c 6c 69 6e | 00 00 00 00 00 00 00 r_8)
  - kde r_8 běží od 0 -> **ff**, dostaneme *O*()=1 pro r_8 = **f3** a dešifrování posledního bajtu "6e" je s největší pravděpodobností 01 ⊕ f3 = a_8
  - další bajt – $r_8 = a_8 \oplus 2$ a dále zkoušíme všechny hodnoty r_7
  - *O*(65 61 64 3e 3c 6c 69 6e | 00 00 00 00 00 00 r_7 **f2**)
  - pokud *O*()=1 => $r_7 \oplus a_7 = 2$ => $a_7 = r_7 \oplus 2$

19

Jiný příklad

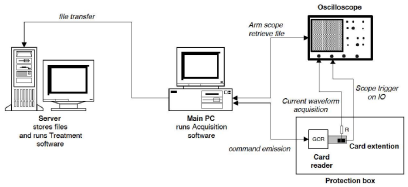
- ŠIFRA: 4D A8 B5 17 64 59 26 B7 | FA 0B 47 2C 04 04 62 33
- Inicializační vektor: 12 12 12 12 12 12 12 12
- r_1: 4C50585A5F31545C
- r_2: 07FDF633645E20B2

$p_{18} = 12 \oplus 5C \oplus 01 = 4F$ $p_{28} = B7 \oplus B2 \oplus 01 = 04$
$p_{17} = 12 \oplus 54 \oplus 02 = 44$ $p_{27} = 26 \oplus 20 \oplus 02 = 04$
$p_{16} = 12 \oplus 31 \oplus 03 = 20$ $p_{26} = 59 \oplus 5E \oplus 03 = 04$
...
p11 = 12 ⊕ 4C ⊕ 08 = 56 $p_{21} = 4D \oplus 07 \oplus 08 = 42$

VELMI DOBRE!

20

Experimentální prostředí pro analýzy



21

Časový postranní kanál

- Využívá toho, že stejné operace nad různými daty (hlavně klíčem) mohou trvat různě dlouho
- Informace o trvání uniká a jsme schopni ji měřit
- Často musíme provést mnoho měření, jejich výsledky ukládat a statisticky zpracovat
- Musíme znát implementaci (jinak nevíme, k čemu naměřené hodnoty použít)

22

TA – Ověření hesla

- Útok na aplikace se základní implementaci kontroly hesel
- Algoritmus:

```
for i = 0 to 7 do
if ( E[i] != P[i]) then return false
end for
return true
```

– E – 8 bytové pole obsahující uživatelem zadané heslo
– P – 8 bytové pole obsahující správné heslo

23

TA – Ověření hesla

- Jak dlouho trvá ověření hesla...
  - když uživatel zadá správné heslo?
  - Když uživatel zadá špatné heslo, které…
    - se liší znakem na prvni pozici?
    - se liší znakem někde uprostřed?

```
for i = 0 to 7 do
if ( E[i] != P[i]) then return false
end for
return true
```

24

TA – Ověření hesla

- Na výstupu dostanu jen true or false
- Ve skutečnosti mam ale ještě další informaci: jak dlouho trvalo ověřování hesla?
- Pokud vím, že je ověřeni implementováno podle tohoto algoritmu, mohu to využit

25

TA – Ověření hesla

- Algoritmus útoku:

1. Pro 0..n..255 vyzkoušim hesla E=(n,0,0,0,0,0,0,0) a ukladam si,jak dlouho pro které n trvalo ověřeni
2. To n, pro které ověřeni trvalo nejdéle, je to správné. Tím jsem určil první bajt – P[0].
3. Pokračuji obdobně, ale využiji znalost prvního bajtu – zkouším pro 0..n..255 hesla E=(P[0],n,0,0,0,0,0,0)...

26

TA – Ověření hesla - efektivita

- Brute-force: 256^8 = 2^64
- Tento útok: 256*8 = 2^11
- … a přitom jsme neobjevili žádnou chybu v algoritmu

27

TA – Ověření hesla - obrana

- Zkusíme přidat do funkce náhodné zpoždění
- Pomůže to?

28

TA – Ověření hesla - obrana

- Zkusíme přidat do funkce náhodné zpoždění
- Útok: pro každé n na každé pozici měříme čas x-krát, průměrujeme, porovnáváme průměry => Stačí nám x-krát tolik pokusů
- Správně: zajistit konstantní čas bez ohledu na zadané heslo

29

TA - RSA

- Útok na RSA podpis
- Útočník může volit vstup, který mu má systém podepsat
- Máme zprávu, spočítáme z ní hash a ten podepíšeme...

30

TA-RSA

- … umocněním na soukromý exponent modulo modul:
  - S = h(m)^d mod n
- Algoritmus podpisu:
  - Square and multiply

```
res = 1; tmp = h
for i = k-1 downto 0 do
tmp = tmp^2 mod n
if d[i]==1 then res = res*tmp mod n
end for
return res
```

- h = hash zprávy
- d = d[k-1]..d[0] = soukromý exponent
- n = modul

31

TA – RSA – Square and multiply

```
res = 1; tmp = h
      for i = k-1 downto 0 do
      tmp = tmp^2 mod n
      if d[i]==1 then res = res*tmp mod n
      end for
      return res
```

- Algoritmus pro umocňování
- x69 binarně^69 = x^64 * x^4 * x^1
- x^69 = (x^64)^**1** * (x^32)^**0** * (x^16)^**0** * (x^8)^**0** * (x^4)^**1** * (x^2)^**0** * (x^1)^**1**
- = **1000101**

- https://asecuritysite.com/encryption/sqm

32

TA – RSA – Montgomery

```
res = 1; tmp = h
      for i = k-1 downto 0 do
      tmp = tmp^2 mod n
      if d[i]==1 then res = res*tmp mod n
      end for
      return res
```

- Montgomeryho násobení modulo
  - Efektivnější a rychlejší než naivní vynásobení a operace modulo, ale…
  - … vrací hodnoty mezi 0 a 2n, kde n je modul, takže…
  - … je v konečné fázi **někdy** potřeba odečíst n…

  … a to trvá nějakou měřitelnou dobu

33

TA – RSA

- Chceme zjistit soukromý exponent d
- Útok využívá různou dobu výpočtu montgomeryho algoritmu pro různé vstupy
- Útok je iterativní, hádáme bit po bitu

34

TA – RSA

- Algoritmus:

1. Předpokládáme, že už známe n bitů, chceme získat bit d[k-n]
2. Tipneme si, že d[k-n]==**1**
3. Náhodně zvolím t zprav m1…mt a rozdělím je na dvě skupiny
   A) Montgomeryho multiplikace **povede** k odečtu
   B) Montgomeryho multiplikace **nepovede** k odečtu
   … což umím, protože už znám předchozích n bitů
4. Každou zprávu nechám podepsat. Počítám průměrný čas podpisu, pro A a B zvlášť
5. Pokud průměry jsou **podobné**, znamená to, že se multiplikace neprováděla. Tedy jsem tipnul špatně a bit je **0**. Pokud se liší zhruba o dobu, jakou trvá odčítaní, multiplikace se prováděla. Tipnul jsem tedy správně **1**.
6. n=n+1, goto 1)

35

Efektivita a opatření

- Pro 128 bitů, útok získá 2 bity/sec pro množinu zpráv L = 10 000
- Pro 512 bitů, útok získá 1 bity/sec pro množinu zpráv L = 100 000

- Efektivitu lze zvyšovat při zapojení dalších postranních kanálů
- Nefunguje pouze pro smartkarty, ale i na servery, počítače a další..

- Opatření:

36

Efektivita a opatření

- Pro 128 bitů, útok získá 2 bity/sec pro množinu zpráv L = 10 000
- Pro 512 bitů, útok získá 1 bity/sec pro množinu zpráv L = 100 000

- Efektivitu lze zvyšovat při zapojení dalších postranních kanálů
- Nefunguje pouze pro smartkarty, ale i na servery, počítače a další..

- Opatření:
- Zajištění konstantního času zpracování případně alespoň nějaká randomizace

37

Odběrový postranní kanál

- Spotřeba energie se u zařízeni mění v závislosti na
  - Prováděných instrukcích
  - Zpracovávaných datech

- Spotřebu můžeme monitorovat pomocí osciloskopu vložením rezistoru zapojeného v sérii se zemí

38

Odběrový postranní kanál

- Existují různé modely závislosti spotřeby na datech/instrukcích
- Hammingova váha dat nebo kódu instrukce
  - počet "1" bitů
  - H(0) = 0
  - H(1) = H(2) = H(4) = H(8) = … = 1
  - H(3) = H(5) = H(6) = H(9) = … = 2

- Hammingova vzdálenost mezi daty v současném stavu a daty v nějakém minulém stavu (např. před provedením posledním instrukce)
  - hd=hw(stav1 XOR stav2) → počet různých bitů
- …

39

PA – zjištění instrukce

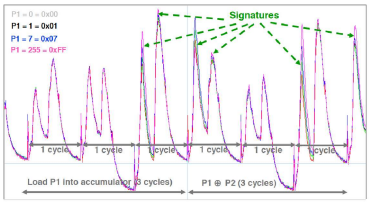
- Známe model závislosti spotřeby – Hammingova vzdálenost
- Mějme neznámou funkci f:
  - V určitém okamžiku smart karta načte hodnotu x do akumulátoru a aplikuje binární operátor (instrukci) nad x a 0
- Chceme zjistit, jakou instrukci zařízení provádí
  - Víme, že zařízení má v registru vstupní data a provede nad nimi operaci.

- Pro hodnoty 0, 1, 7 a 255 (0b, 1b, 111b, 11111111b) provedeme instrukci a zaznamenáváme průběh napětí
- Naneseme tyto průběhy do jednoho grafu (x, spotřeba(x))

40

PA – zjištění instrukce

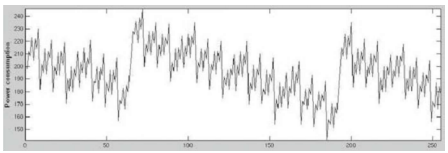
- Nahraj hodnotu do P1 a proveď XOR s P2 = 0 tak, že P1 = 0; 1; 7; 255



41

PA – zjištění instrukce

- Vybereme si nějakou signaturu
- Pro tuto signaturu změříme spotřebu pro všechny vstupní hodnoty 0..255 a zaneseme do grafu (x,spotřeba(x))



42

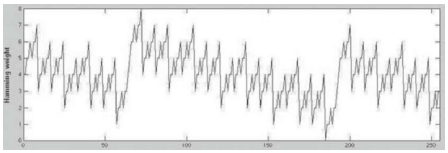
PA – zjištění instrukce

- Tento graf budeme porovnávat s druhým grafem:
- Pro každou z hodnot i=0..255, nakreslíme graf (x,hw(i XOR x))
- Víme přeci, že modelem spotřeby je Hammingova vzdálenost…
- … a že Hammingova vzdálenost je počet různých bitů…
- … a to je to samé, jako počet bitů, které se změní při nahrazení jedné hodnoty druhou…
- … a to je přesně to, co se děje v registru, kde data před vykonáním instrukce, nahradíme daty po vykonání instrukce.
- Z grafů vybereme ten, ktery je nejpodobnější původnímu grafu
- Tím zjistíme správné **i**, což je kód použité instrukce

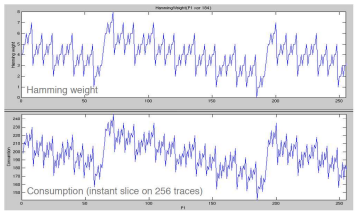
43

PA – zjištění instrukce

- Jeden z 256 grafů (x,hw(i,x))



44



45

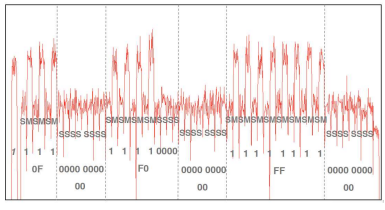
SPA – útok na RSA

- Záznam spotřeby energie square and multiply algoritmu
- Nižší odběr → Jen umocnění → 0
- Vyšší odběr → Umocnění a násobení → 1

46

SPA – útok na RSA

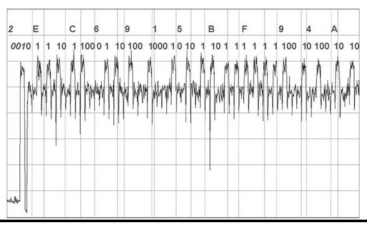
- Testovací hodnota klíče: 0F 00 F0 00 FF 00



47

SPA – útok na RSA

- Testovací hodnota klíče : 2E C6 91 5B F9 4A



48

Diferenciální odběrová analýza

- Používá se tam, kde obyčejná odběrová analýza selhává kvůli přílišnému šumu v nasnímaných datech
- Využívá statistické metody
- Srovnává data nasnímaná během mnoha běhů systému, ne jen jednoho

49

DPA – Bit Tracing

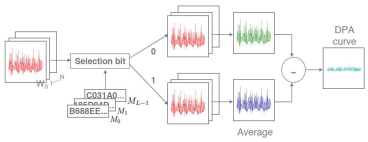
- Slouží k nalezení místa, kdy se vykonává určitý kód – tedy sleduji průběh spotřeby zařízeni a chci z něj vyčíst, v jakou chvíli probíhá nějaká operace
- Např. data jsou nejdříve zašifrována a pak někam kopírována. Která část grafu odpovídá kopírovaní?

50

DPA – Bit Tracing

- Mějme funkci D, která pro každý vstup x vrací 0 nebo 1
- DPA má dvě fáze:
- Nasbírám několik záznamů spotřeby
- Vytvořím množiny:
- L0={x|D(x)=0}
- L1={x|D(x)=1}

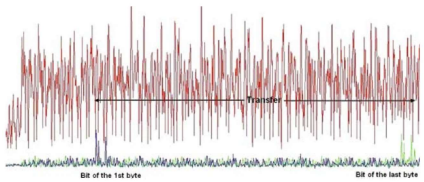
51



52

DPA – Bit Tracing

- (čas,avg(spotřeba(t,L1)) – avg(spotřeba(t,L0)))



53

DPA – Bit Tracing

- Pro 8bit zařízení, které se chova podle modelu hammingovy vahy:
- D definujeme tak, že vrací hodnotu nejnižšího bitu
- L0 tedy má průměrnou hammingovu váhu 3.5
  - Protože jeden bit je určitě 0 a ostatní nevíme
- L1 ma průměrnou hammingovu vahu 4.5
  - Protože jeden bit je určitě 1 a ostatní nevíme
- Tento rozdíl v Hammingově váze se promítne i ve spotřebě
- Dokážeme záznamy rozdělit na ty odpovídající L0 a L1, protože kryptosystém nám vrátí ciphertext

54

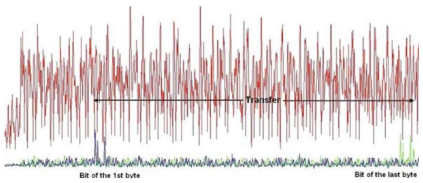
DPA – Bit Tracing

- Spustili jsme tedy systém na mnoha vzorcích, ty potom zpětně rozdělili do dvou kategorii i se záznamy jejich spotřeby. Záznamy jsme zpracovali tak, že v grafu vidíme rozdíl mezi dvěma kategoriemi. Tam, kde se liší, bylo evidentně pracováno s bitem, ze kterého vychází funkce D
- Při pohledu na graf vidíme, se kterým bitem bylo manipulováno ve kterém místě. Víme, že jde o ciphertext, tedy data už byla zašifrovaná, takže jde o kopírovaní. Našli jsme tedy, kdy jsou data v systému kopírovaná

55

DPA – Bit Tracing

- (čas,avg(spotřeba(t,L1)) – avg(spotřeba(t,L0)))



56

Elektromagnetický kanál

- Využívá obecnější a detailnější data o průtoku proudu, než jen celková spotřeba zařízení
- Sleduje, kterou částí zařízení proud protéká a jak spolu proudy interferují

57

Obrana proti side-channel útokům

- Nevětvit program na základě tajných hodnot. Bez ohledu na data by program měl provést vždy to samé
- Dvě skupiny opatření:
- Skrývání
  - Zařízení vyrobeno tak, že bude mít náhodnou spotřebu
  - Např. přidáním šumu, čekání, Dummy operací, …
- Maskování
  - přimaskování náhodných hodnot k hodnotám ve výpočtech

58

Trezor One – HW kryptopeněženka

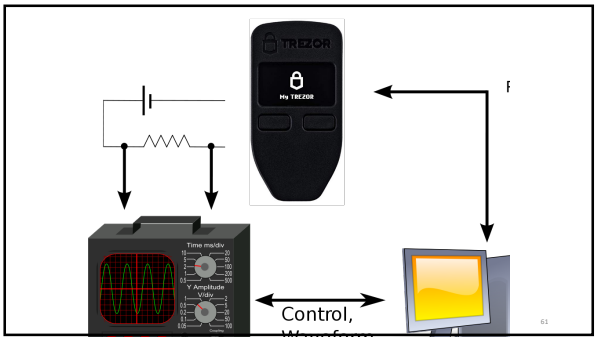
- Realizace dvou profilovaných útoků bočního kanálu na zařízení vedoucí k:
  - získání PIN kódu odcizeného zařízení
  - získávání významných částí skaláru použitého během násobení eliptické křivky (což vede k obnovení soukromého klíče)
- Útok zveřejněn po opravě firmware
- https://medium.com/ledger-on-security-and-blockchain/details-about-the-side-channel-attacks-on-trezor-one-hardware-wallet-62e2d278e803

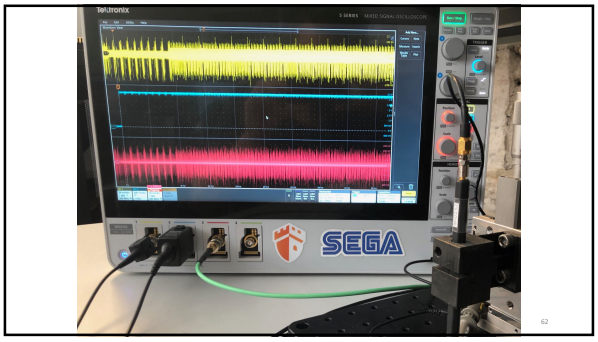
59

Trezor One – HW kryptopeněženka

- Hardwarové peněženky byly navrženy tak, aby chránily soukromé klíče používané pro přístup k účtům kryptoměny
- Tato tajemství mají nikdy neopustit zařízení
- Bylo použito zařízení s modifikovaným firmwarem pro podporu profilování, což umožnilo poslat více požadavků na cílené bezpečnostní funkce, např. zápisy, které snižují počet pokusů o PIN (PTC), jsou pro automatizaci deaktivovány
- Získáno 3000 časových vzorků na stopu
- Stačil by jednoduchý osciloskop se šířkou pásma 100 MHz a vzorkovací frekvencí 1 G / s, který by stál kolem 1 kB $.

60





The PIN verification function code extracted from the firmware source code

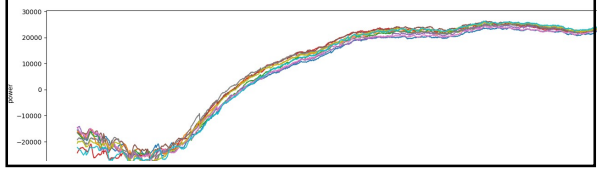
```
1 /* Check whether pin matches storage. The pin must be
2 * a null-terminated string with at most 9 characters.
3 */
4 bool storage_containsPin(const char *presented_pin)
5 {
6     /* The execution time of the following code only depend
7     * (public) input. This avoids timing attacks.
8     */
9     char diff = 0;
10    uint32_t i = 0;
11    while (presented_pin[i]) {
12        diff |= storageRom->pin[i] - presented_pin[i];
13        i++;
```

Pár komentářů ke kódu

- `StorageRom-> pin` je hodnota, kterou hledáme: hodnota PIN uživatele. Skládá se z tabulky N číslic a každá z číslic může nabývat 9 hodnot (číslici 0 nelze použít)
- Jak je napsáno v komentářích, funkce není citlivá na načasování útoků: ale čas není postranní kanál, který je použit
- Číslice ve `storageRom-> pin` jsou zpracovány ve funkci jedna po druhé v hlavní smyčce. To znamená, z pohledu postranního kanálu, že se může cílit na každou číslici nezávisle na ostatních (strategie Divide& Conquer)
- Existuje určitá citlivá hodnota, která závisí na tajemství a které vypadá zajímavě: odčítání v každém kroku smyčky
  `storageRom-> pin [i] - present_pin [i] pro 0 ≤ i <4`
- Tato citlivá hodnota zpracovává jak tajný klíč, tak vstupní hodnotu, což otvírá útočníkovi možnosti využití postranního kanálu

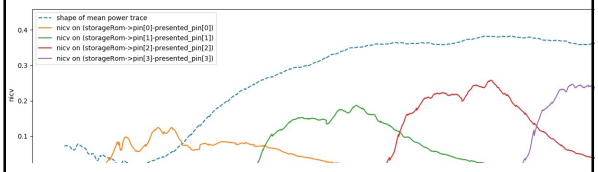
SPA

- Jak vypadají data z postranního kanálu?
- Příklad: Odběr odpovídající 10 verifikacím PINu



Prokázání závislosti na spotřebě

- 200 tisíc měření (traces) s náhodným, ale známým vstupem
- Pro každé měření vypočítána množina hodnot k odečtení (lze, protože známe všechny vstupy)
- Model potvrdil závislost (podle velikosti odečítání) a ukázal i, kdy jsou jednotlivé číslice zpracovávány

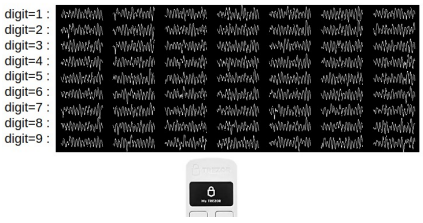


67

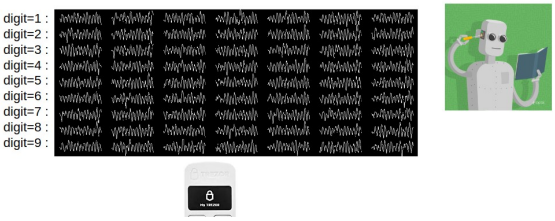
Profilovací fáze

- Pro každou číslici PIN získána odpovídající charakterizace (NICV - Normalized Inter-Class Variance)
- Každá stopa může být označena (pro každou číslici PIN) hodnotou příslušného odečtení
- Fáze profilování je v zásadě instancí Machine Learning Classification
- Známá ohodnocená data používáme k sestavení jednoho klasifikátoru na každou číslici PIN. Klasifikátor je rozhodovací funkce určená k ohodnocení nové neznámé stopy
- V tomto případě je označení stopy hodnotou odčítání. Protože vždy známe hodnotu `present_pin`, znalost odčítání znamená znát hodnotu `storageRom-> pin`.
- Klasifikátory následně použity pro hledání neznámého PINu (15 PINů, 300 měření

68



69



70

Efektivnost

- Průměrováním byl PIN uhodnut (worst case zlepšen z 10 pokusů na 5 vs. aktuálně použitý limit je 16 pokusů)

71

Díky za pozornost!

72

Efficient Padding Oracle Attacks on Cryptographic Hardware

Romain Bardou, Riccardo Focardi, Yusuke Kawamoto, Lorenzo Simionato, Graham Steel, Joe-Kai Tsay

Efficient Padding Oracle Attacks on Cryptographic Hardware

Romain Bardou, Riccardo Focardi, Yusuke Kawamoto, Lorenzo Simionato, Graham Steel, Joe-Kai Tsay

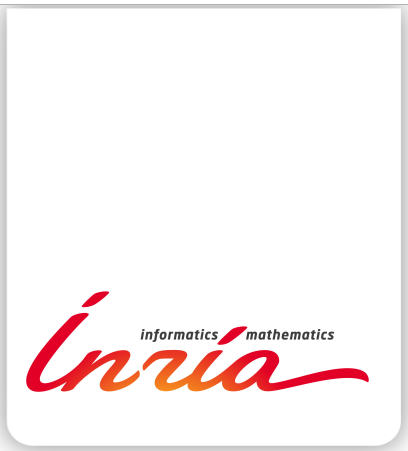
RESEARCH REPORT

N° 7944

Avril 2012

Project-Team Prosecco

ISSN 0249-6399 ISRN INRIA/RR--7944--FR+ENG



Efficient Padding Oracle Attacks on Cryptographic Hardware

Romain Bardou*, Riccardo Focardi†, Yusuke Kawamoto‡,
Lorenzo Simionato†§, Graham Steel*, Joe-Kai Tsay¶

Project-Team Prosecco

Research Report n° 7944 — Avril 2012 — 19 pages

* INRIA Project Prosecco, France
† University of Venice Ca' Foscari, Italy
‡ University of Birmingham, UK
§ Now at Google Inc.
¶ Norwegian University of Science and Technology (Norges Teknisk-Naturvitenskapelige Universitet), Norway

RESEARCH CENTRE
PARIS – ROCQUENCOURT

Domaine de Voluceau, - Rocquencourt
B.P. 105 - 78153 Le Chesnay Cedex

Abstract: We show how to exploit the encrypted key import functions of a variety of different cryptographic devices to reveal the imported key. The attacks are padding oracle attacks, where error messages resulting from incorrectly padded plaintexts are used as a side channel. In the asymmetric encryption case, we modify and improve Bleichenbacher's attack on RSA PKCS#1v1.5 padding, giving new cryptanalysis that allows us to carry out the 'million message attack' in a mean of 49 000 and median of 14 500 oracle calls in the case of cracking an unknown valid ciphertext under a 1024 bit key (the original algorithm takes a mean of 215 000 and a median of 163 000 in the same case). We show how implementation details of certain devices admit an attack that requires only 9 400 operations on average (3 800 median). For the symmetric case, we adapt Vaudenay's CBC attack, which is already highly efficient. We demonstrate the vulnerabilities on a number of commercially available cryptographic devices, including security tokens, smartcards and the Estonian electronic ID card. The attacks are efficient enough to be practical: we give timing details for all the devices found to be vulnerable, showing how our optimisations make a qualitative difference to the practicality of the attack. We give mathematical analysis of the effectiveness of the attacks, extensive empirical results, and a discussion of countermeasures and manufacturer reaction.

Key-words: Chosen ciphertext attack, padding oracles, PKCS#11, HSMs, electronic ID cards

Attaques Efficaces sur Appareils Cryptographiques par Oracle de Padding

Résumé : Nous montrons comment exploiter l'interface de plusieurs appareils cryptographiques pour extraire leurs clés cryptographiques. Nos attaques sont effectué par oracle de padding.

Mots-clés : Cartes à puces, Chosen ciphertext attack, padding oracles, PKCS#11, HSMs

1 Introduction

Tamper-resistant cryptographic security devices such as smartcards, USB keys, and Hardware Security Modules (HSMs) are an increasingly common component of distributed systems deployed in insecure environments. Such a device must offer an API to the outside world that allows the keys stored on the device to be used for cryptographic functions and permits key management operations, but without compromising security. The most commonly used standard for designing cryptographic device interfaces, RSA PKCS#11 [24], is known to have vulnerabilities if the attacker is assumed to have access to the full API, and can therefore make attacks by combining commands in unexpected ways [4, 5,7]. In this paper, we describe a different way to attack keys stored on the device using only decryption queries performed by a single function, usually the `C_UnwrapKey` function for encrypted key import. These attacks are cryptanalytic rather than purely logical, and hence require multiple command calls to the interface, but the attacker only needs access to one seemingly innocuous command, subverting the typical countermeasure of introducing access control policies permitting only limited access to the interface.

We will show how the `C_UnwrapKey` command from the PKCS#11 API is often implemented on commercially available devices in such a way that it offers a 'padding oracle', i.e. a side channel allowing him to see whether a decryption has succeeded or not. We give two varieties of the attack: the first for when the imported key is encrypted under a public key using RSA PKCS#1 v1.5 padding, which is still by far the most common and often the only available mechanism on the devices we obtained, and the second for when the key is encrypted under a symmetric key using CBC and PKCS#5 padding. The first attack is based on Bleichenbacher's well-known attack [2]. Although commonly known as the 'million message attack', in practice Bleichenbacher's attack requires only about 215 000 oracle calls on average against a 1024 bit modulus when the ciphertext under attack is known to be a valid PKCS#1 v1.5 block. This is however not efficient enough to be practical on low power devices such as smartcards which perform RSA operations rather slowly. We give a modified algorithm which results in an attack which is 4 times faster on average than the original, with a median attack time over 10 times faster. We also show how the implementation details of some devices can be exploited to create stronger oracles, where our algorithm requires only 9400 mean (3800 median) calls to the oracle. At the heart of our techniques is a small but significant theorem that allows not just multiplication (as in the original attack) but also division to be used to manipulate a PKCS#1 v1.5 ciphertext and learn about the plaintext. In the second attack we use Vaudenay's technique [26] which is already highly efficient. Countermeasures to such chosen ciphertext attacks are well known: one should use an encryption scheme proven to be secure against them. We discuss the availability of such modes in current cryptographic hardware and examine what other countermeasures could be used while such modes are still not available.

In summary, our contributions are the following: i) new results on PKCS#1 v1.5 cryptanalysis that, when combined with the 'parallel threads' technique of Klima-Pokorny-Rosa [25] (which on its own contributes a 38% improvement on mean and 52% on median) results in an improved version of Bleichenbacher's algorithm giving a fourfold (respectively tenfold) improvement in mean (respectively median) attack time compared to the original algorithm (measured over 1000 runs with randomly generated 1024 bit RSA keys and randomly generated conforming plaintexts); ii) demonstration of the attacks on a variety of cryptographic hardware including USB security tokens, smartcards and the Estonian electronic ID card, where we found various implementations of the oracle, and adapted our algorithm to each one, resulting in attacks with as few as 9400 mean (3800 median) oracle calls on the most vulnerable devices; iii) analysis of the complexity of the attacks, empirical data, and manufacturer reaction.

In the next section, we describe the padding attacks relevant to this work and describe our modifications to Bleichenbacher's algorithm. The results on commercial devices are described in section 3. We discuss countermeasures in section 4. Finally we conclude with a discussion of future work in

section 5.

2 Padding Oracle Attacks

A padding oracle attack is a particular type of side channel attack where the attacker is assumed to have access to an oracle which returns true just when a chosen ciphertext corresponds to a correctly padded plaintext under a given scheme.

2.1 Bleichenbacher's Attack

Bleichenbacher's padding oracle attack, published in 1998, applies to RSA encryption with PKCS#1 v1.5 padding [2]. Let n, e be an RSA public key and d be the corresponding private key, i.e. $n = pq$ and $ed \equiv 1 \ (mod \ \phi(n))$. Let k be the byte length of n, so $2^{8(k-1)} \leq n < 2^{8k}$. Suppose we want to encrypt a plaintext block P where P is l bytes long. Under PKCS#1 v1.5 we first generate a pseudorandom non-zero padding string PS which is $k - 3 - l$ bytes long. We allow l to be at most $k - 11$, so there will be at least 8 bytes of padding. The block for encryption is now created as

$$\texttt{0x00}, \texttt{0x02}, PS, \texttt{0x00}, P$$

We call a correctly padded plaintext and a ciphertext that encrypts a correctly padded plaintext *PKCS conforming* or just *conforming*. For the attack, imagine, as above, that the attacker has access to an oracle that tells him just when an encrypted block decrypts to give a conforming plaintext, and assume he is trying to obtain the message $m = c^d \ mod \ n$, where c is an arbitrary integer. He is going to choose integers s, calculate $c' = c \cdot s^e \ mod \ n$ and then send c' to the padding oracle. If c' is conforming then he learns that the first two bytes of $m \cdot s$ are $\texttt{0x00}, \texttt{0x02}$. Hence, if we let $B = 2^{8(k-2)}$, $2B \leq m \cdot s \ mod \ n < 3B$. The idea is to repeat the process for many values of s until only a single plaintext is possible.

2.2 Improving the Bleichenbacher Attack

Let us first review in a little more detail the original attack algorithm. We are trying to obtain message $m = c^d \ mod \ n$ from ciphertext c. In step 1 (Blinding), we search for a random integer value s_0 such that $c(s_0)^e \ mod \ n$ is conforming, by accessing the padding oracle. We let $c_0 = c(s_o)^e \ mod \ n$ and $m_0 = (c_0)^d \ mod \ n$. Note that $m_0 = ms_0 \ mod \ n$. Thus, if we recover m_0 we can compute the target m as $m_0(s_0)^{-1} \ mod \ n$. If the target ciphertext is already conforming, we can set s_0 to 1 and skip this step.

We let $B = 2^{8(k-2)}$. If c_0 is conforming, $2B \leq m_0 < 3B$. Thus, we set the initial set M_0 of possible intervals for the plaintext as $\{[2B, 3B - 1]\}$. In step 2, we search for s_i such that $c(s_i)^e \ mod \ n$ is conforming. In step 3, we apply the s_i we found to narrow the set of possible intervals M_i containing the value of the plaintext, and in step 4 we either compute the solution or jump back to step 2.

We are interested in improving step 2, i.e. the search for s_i. We give step 2 of the original algorithm below, and omit the other steps (in the appendix we give our modified algorithm, of which step 1.a equals step 1 of the original algorithm, whereas steps 3 and 4 are unchanged from the original).

Step 2a If $i = 1$ (i.e. we are searching for s_1), search for the smallest positive integer $s_1 \geq n/(3B)$ such that $c_0(s_1)^e \ mod \ n$ is conforming. It can be shown that smaller values of s_1 never give a conforming ciphertext.

Step 2b If $i > 1$ and $|M_{i-1}| > 1$, search for the smallest positive integer $s_i > s_{i-1}$ such that $c_0(s_i)^e \ mod \ n$ is conforming.

Step 2c If $i > 1$ and $|M_{i-1}| = 1$, i.e. $M_{i-1} = \{[a, b]\}$, choose small r_i, s_i such that

$$r_i \geq 2\frac{bs_{i-1}-2B}{n} \qquad \text{and} \qquad \frac{2B+r_in}{b} \leq s_i < \frac{3B+r_in}{a}$$

until $c_0(s_i)^e \ mod \ n$ is conforming. Intuitively, the bounds for s_i derive from the fact that we want $c_0(s_i)^e \ mod \ n$ conforming, i.e. $2B \leq m_0s_i - r_in < 3B$, for some r_i, and from the assumption $a \leq m_0 \leq b$. As explained in the original paper, the constraint on r_i aims at dividing the remaining interval in half so to maximize search performance.

Some features of the algorithm's behaviour were already known from the original paper. For example, step 2a/b will in general be executed only very few times (in roughly 90% of our trials, step 2b was executed a maximum of once, and in 32% of cases not at all). However, a lot of the expected calls are here, since each time we just search naïvely for the next s_i, which takes an expected $1/Pr(P)$ calls where $Pr(P)$ is the probability of a random ciphertext decrypting to give a conforming block. Step 2c, meanwhile, is highly efficient, but is only applicable if there is only one interval left. Furthermore it cannot be directly applied to the original interval $\{2B, 3B - 1\}$ (since the bound on r_i, s_i collapses and we end up with the same search as in step 2a). Based on this observation, we devised a new method for narrowing down the initial interval so that 'step 2c-like' reasoning could be applied to speed up the search for s_1.

Trimming M_0 First observe that as well as multiplying the value of the decrypted plaintext ($mod \ n$) by some integer s, we can also divide it by an integer t by multiplying the original ciphertext by $t^{-e} \ mod \ n$. Multiplication modulo n is a group operation on $(\mathbb{Z}_n)^*$, so inverses are unique. If the original plaintext was divisible by t, the result $m_0 \cdot t^{-1} \ mod \ n$ will just be m_0/t, otherwise it will be some other value in the group that we in general cannot predict without knowing m_0. The following holds.

Proposition 1. *Let u and t be two coprime positive integers such that $u < \frac{3}{2}t$ and $t < \frac{2n}{9B}$. If m_0 and $m_0 \cdot ut^{-1} \ mod \ n$ are PKCS conforming, then m_0 is divisible by t.*

Proof. We have $m_0u < m_0\frac{3}{2}t < 3B\frac{3}{2}t < n$. Thus, $m_0u \ mod \ n = m_0u$. Let $x = m_0 \cdot ut^{-1} \ mod \ n$. We know $x < 3B$ since it is conforming. Thus $xt < 3Bt < n$ and $xt \ mod \ n = xt$. Now, $xt = xt \ mod \ n = m_0u \ mod \ n = m_0u$ which implies t divides m_0. □

By Proposition 1, if we find coprime positive integers u and t, $u < \frac{3}{2}t$ and $t < \frac{2n}{9B}$ such that for a PKCS conforming m_0, $m_0 \cdot ut^{-1} \ mod \ n$ is also conforming, then we know that m_0 is divisible by t and $m_0 \cdot ut^{-1} \ mod \ n = m_0\frac{u}{t}$. As a consequence

$$2B \cdot t/u \leq m_0 < 3B \cdot t/u.$$

Note that since we already know $2B \leq m_0 < 3B$ we can restrict our search to t and u such that $2/3 < u/t < 3/2$. We apply this by constructing a list of suitable fractions u/t that we call 'trimmers'. In practice, we use a few thousand trimmers and take $t \leq 2^{12}$ as the implementations typically satisfy $n \geq 2^{8k-1}$. For each trimmer u/t, we submit $c_0u^et^{-e}$ to the padding oracle. If the oracle succeeds, we can trim the bounds of M_0.

A large denominator t allows for a more efficient trimming. The trimming process can be thus optimised by taking successful trimming fractions $u_1/t_1, \ldots, u_n/t_n$, computing the lowest common multiple t' of $t_1, \ldots, t_n$, using this value as a denominator and then searching for the highest and lowest numerators u_h, u_l that imply a valid padding, giving $2B \cdot t'/u_l \leq m < 3B \cdot t'/u_h$.

Skipping Holes In the original algorithm step 2a, the search for the first s_1 starts at the value $\lceil n/3B \rceil$. However, note that to be conforming we require in fact that $m \cdot s \geq n+2B$. Since $3B-1 \geq m$ we get $(3B-1)s \geq n+2B$. So we can start with $s = \lceil (n+2B)/(3B-1) \rceil$. On its own this does not save us much: about 8000 queries depending on the exact value of the modulus. However, when we have already applied the trimming rule above to reduce the upper bound on M_0 to some b, this translates immediately into a better start bound for s_1 of $(n+2B)/b$.

Observe that in general for a successful s we must have $2B \leq ms - jn < 3B$ for some natural number j. Given that we have trimmed the first interval M_0 to the range $[a, b]$, this gives us a series of bounds

$$\frac{2B+jn}{b} \leq s < \frac{3B+jn}{a}$$

Observe further that when

$$\frac{3B+jn}{a} < \frac{2B+(j+1)n}{b}$$

we have a 'hole' of values where a suitable s cannot possibly be. When used in combination with the trimming rule, we found that we frequently obtain a list of many such holes. We use this list to skip out the holes during the search for the s_1. Note that this is similar to the reasoning used to calculate s values in step 2c, except that here we are concerned with finding the smallest possible s_1 in order to have the fewest possible intervals remaining when searching for s_2. As we show in the results below, the combination of the trimming and hole skipping techniques is highly effective, in particular against more permissive oracles than a strict PKCS padding oracle.

2.3 Existing Optimisations

In addition to our original modifications, we also implemented changes proposed by Klima, Pokorny and Rosa (KPR) [25]. These are mainly aimed at improving performance in step 2b, because they were concerned with attacking a weaker oracle where most time was spent in step 2b (see below). They are therefore naturally complementary to our optimisation of step 2a.

Parallel thread method The parallel thread method consists of omitting step 2b in the case where there are several intervals in M_{i-1}, and instead forking a separate thread for each interval and using the method of step 2c to search for s_i. As soon as one thread finds a hit, all threads are halted and the new intervals are calculated. If there is still more than one interval remaining, new threads are launched. In practice, since access to the oracle may not be parallelisable, the actions of each thread can be executed stepwise. This heuristic is quite powerful in practice, as we will see below.

Tighter bounds and Beta Method KPR were concerned with attacking the weaker 'bad version' oracle found in implementations of SSL patched against the original vulnerability. This meant that when the oracle succeeds, they could be sure of the length of the unpadded plaintext, since it must be the right length for the SSL 'pre-master secret'. This allowed them to tighten the $2B$ and $3B-1$ bounds. We also implemented this optimisation where possible, since it has no significant cost, but its effects are not significant. We implemented a further proposal of KPR, the so-called 'Beta Method' that we do not have space to describe here(see appendix A), but again found that it caused little improvement in practice.

2.4 Stronger and Weaker Oracles

In order to capture behaviour found in real devices (see section 3), we define stronger and weaker Bleichenbacher oracles, i.e. oracles which return true for a greater or smaller proportion of values x such that $2B \leq x < 3B$. We characterise them by three Booleans specifying the tests they apply or

skip on the decrypted plaintext. The first Boolean corresponds to the test for a 0 somewhere after the first ten bytes. The second Boolean corresponds to the check for 0s in the non-zero padding. The third Boolean corresponds to a check of the plaintext length against some specific value (e.g. 16 bytes for an encrypted AES-128 key). More precisely, we say an oracle is FFF if it returns true only on correctly padded plaintexts of a specific fixed length, like the the KPR 'bad version' oracle found in some old versions of SSL. An oracle is FFT if it returns true on a correctly padded plaintext of any length. This is the standard PKCS oracle used by Bleichenbacher. An oracle is FTT if it returns true on a correctly padded plaintext of any length and additionally on an otherwise correctly padded plaintext containing a zero in the eight byte padding. An oracle is TFT if if returns true on a correctly padded plaintext of any length and on plaintexts containing no 0s after the first byte. The most permissive oracle, TTT, returns true on any plaintext starting with `0x00, 0x02`. We will see in the next section how all these oracles arise in practice.

In Table 1, we show performance of the standard Bleichenbacher algorithm on these oracles, apart from FFF for which it is far too slow to obtain meaningful statistics. Attacking the strongest oracles TTT and TFT is substantially easier than the standard oracle. We can explain this by observing that for the original oracle, on a 1024 bit block, the probability $Pr(P)$ of a random ciphertext decrypting to give a conforming block is equal to the probability that the first two blocks are `0x00, 0x02`, the next 8 bytes are non-zero, and there is a zero somewhere after that. We let $Pr(A)$ be the probability that the first two bytes are `0x00, 0x02`, i.e $Pr(A) \approx 2^{-16}$. We identify $Pr(P|A)$, the probability of a ciphertext giving a valid plaintext provided the first two bytes are `0x00, 0x02`, i.e

$$\left(\frac{255}{256}\right)^8 . \left(1 - \left(\frac{255}{256}\right)^{118}\right) \approx 0.358$$

$Pr(P)$ is therefore $0.358 \cdot 2^{-16}$. Bleichenbacher estimates that, if no blinding phase is required, the attack on a 128 byte plaintext will take

$$2/Pr(P) + 16 \cdot 128/Pr(P|A)$$

oracle calls. So we have

$$(2 \cdot 2^{16} + 16 \cdot 128)/Pr(P|A) = 371843$$

In the case of, say, the TTT oracle, $Pr(P|A)$ is 1, since any block starting `0x00, 0x02` will be accepted. Hence we have

$$2^{17} + 16 \cdot 128 = 133120$$

oracle queries. This is higher than what we were able to achieve in practice in both cases, but the discrepancy is not surprising since the analysis Bleichenbacher uses is a heuristic approximation of the upper bound rather than the mean. However, it gives an explanation of why the powerful oracle gives such a big improvement in run times: improvements in the oracle to $Pr(P|A)$ make a multiplicative difference to the run time. Additionally, the expected number of intervals at the end of step 2a is $\lceil s_1 \cdot B/n \rceil$ [2, p. 7], so if s_1 is less than 2^{16}, the expected number of intervals is one. For the FFT oracle, the expected value of s_1 (calculated as $1/2 \cdot 1/Pr(P)$) is about 91 500, between 2^{16} and 2^{17}, whereas for TTT it is 2^{15}. That means that in the TTT case we can often jump step 2b and go straight to step 2c, giving a total of

$$2^{16} + 16 \cdot 128 = 34816$$

i.e. the TTT oracle is about 10 times more powerful than the FFT oracle, which is fairly close to what we see in practice (our mean for FFT is about 5.5 times that for TTT).

In comparison, if the modulus is 2048 bit long, then $Pr(P|A) \approx 0.599$. Because the modulus is longer, the probability that `0x00` appears after the 8 non-zero bytes is higher than in the 1024 bit case. Furthermore, following the same argument as above, we obtain that the attack on a 2048 bit plaintext will take about 335 065 calls to the FFT oracle, fewer than in the 1024 bit case. Note however that RSA private key operations slow down by roughly a factor of four when key length is doubled.

Oracle	Original algorithm		Modified algorithm			
	Mean	Median	Mean	Median	Trimmers	Mean skipped
FFF	-	-	18 040 221	12 525 835	50 000	7 321
FFT	215 982	163 183	49 001	14 501	1 500	65 944
FTT	159 334	111 984	39 649	11 276	2 000	61 552
TFT	39 536	24 926	10 295	4 014	600	20 192
TTT	38 625	22 641	9 374	3 768	500	18 467

Table 1: Performance of the original and modified algorithms.

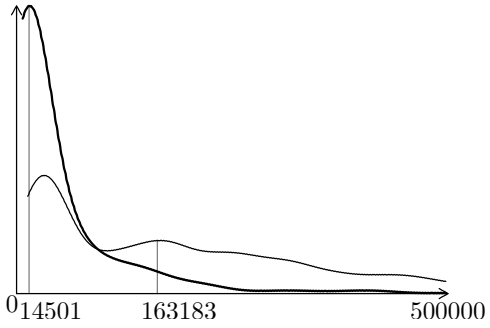


Figure 1: Graph comparing distribution of oracle calls for original (lower peak, thinner line) and optimised version of the algorithm on the FFT oracle. Median is marked for each.

2.5 Performance of the Modified Algorithm

Referring again to Table 1, we give a summary of our experiments with our modified algorithm. As well as mean and median, we give the number of trimming fractions tried and the average number of oracle calls saved by the hole skipping modification we presented in section 2.2. Observe that as the oracles become stronger, the contribution of the KPR 'parallel threads' method becomes less significant and our hole skipping technique more significant. This is to be expected, since as discussed above, for the stronger oracles, fewer runs need to use step 2b. Similarly, when trimming the first interval M_0, we find that more fractions can be used because of the more permissive oracle, hence we find more holes to skip. For the most restrictive oracle, FFF, the addition of our trimming method slightly improves on the results of KPR (which were 20 835 297 mean and 13 331 256 median). Note also that the trimming technique contributes more than just the oracle calls saved by the hole skipping, it also slightly improves performance on all subsequent stages of the algorithm. We know this because we can compare performance using only the parallel threads optimisation, where we obtain a mean of 113 667 and a median of 78 674 (on the FFT oracle). In Figure 1, we give the density distribution for 1000 runs of the original algorithm and our optimised algorithm on the classical FFT oracle, with medians marked. Notice the change in shape: we have a much thinner tail.

2.6 Vaudenay's Attack

Vaudenay's attack on CBC mode symmetric-key encryption [26] is somewhat simpler and highly efficient. Recall first the operation of CBC mode [8]: given some block cipher with encryption, decryption functions $E(.), D(.)$ and a fixed block size of b bytes, suppose we want to encrypt a message P of length $l = j \cdot b$ for some integer j, i.e. $P = P_1, \ldots, P_j$. In CBC mode, we first choose a fresh *initialisation vector* IV. The first encrypted block is defined as $C_1 = E(IV \oplus P_1)$, and subsequent blocks as $C_i = E(C_{i-1} \oplus P_i)$. The need for padding arises because l is not always a multiple of b. Suppose $l = j \cdot b + r$. Then we need to encrypt the last r bytes of the message in a b bytes block in

Device	PKCS#11 version	PKCS#1 v1.5 Attack		CBC-PAD Attack	
		Token	Session	Token	Session
Aladdin eTokenPro	2.01	✓	✓	✓	✓
Feitian ePass 2000	2.11	×	×	N/A	N/A
Feitian ePass 3003	2.20	×	×	N/A	N/A
Gemalto Cyberflex	2.01	✓	N/A	N/A	N/A
RSA Securid 800	2.20	✓	N/A	N/A	N/A
Safenet Ikey 2032	2.01	✓	✓	N/A	N/A
SATA DKey	2.11	×	×	×	×
Siemens CardOS	2.11	✓	✓	N/A	N/A

Table 2: Attack Results on Tokens

such a way that on decryption, we can recognise that only the first r bytes are to be considered part of the plaintext. One way to do this is the so-called RC5 padding, also known as PKCS padding and described in RFC 5652 [11]. The r bytes are encoded into the leftmost bytes of the final block, and then the final $b - r$ bytes are filled with the value $b - r$. Under this padding scheme, if the plaintext length should happen to be an exact multiple of the block size, then we add a whole block of padding bytes b.

To effect Vaudenay's attack, suppose that the attacker has some ciphertext $C_1, \ldots, C_n$ and access to an oracle that returns true just when a ciphertext decrypts with valid padding. To attack a given block C_i, we first prepend a random block $R = r_1, \ldots, r_b$. We then ask the padding oracle to decrypt $R \mid C_i$. If the padding is valid most probably the final byte is 1, hence the final byte p_m of the plaintext P_i satisfies $p_b = r_b \oplus 1$. If the padding is not accepted, we iterate over i setting $r'_b = r_b \oplus i$ and retrying the oracle until eventually it is accepted. There is a small chance that the final byte of an accepted block is not 1, but this is easily detected. Having discovered the last byte, it is easy to extend the attack to obtain p_{b-1} by tweaking r_{b-1}, and so on for the whole block. Given this 'block decryption oracle' we can then apply it to all the blocks of the message. Overall, the attack requires $O(nb)$ steps, and hence is highly efficient.

Since the original attack appeared, many variations have been found on other padding schemes and block cipher modes [1, 6, 13, 16, 19, 21]. Bond and French recently showed that the attack could be applied to the `C_UnwrapKey` command as implemented on a hardware security module (HSM) [3]. We will show in the next section that many cryptographic devices are indeed vulnerable to variants of the attack.

3 Attacking Real Devices

We applied the optimised versions of the attacks of Bleichenbacher and Vaudenay presented in section 2 to the unwrap functionality of PKCS#11 devices. RSA PKCS#11, which describes the 'Cryptoki' API for cryptographic hardware, was first published in 1995 (v1.0). The latest official version is v2.20 (2004) which runs to just under 400 pages [24]. Adoption of the standard is almost ubiquitous in commercial cryptographic tokens and smartcards, even if other additional interfaces are frequently offered. In a PKCS#11-based API, applications initiate a *session* with the cryptographic token, by supplying a PIN. Once a session is initiated, the application may access the *objects* stored on the token, such as keys and certificates. Objects are referenced in the API via *handles*, which can be thought of as pointers to or names for the objects. In general, the value of the handle, e.g. for a secret key, does not reveal any information about the actual value of the key. Objects have *attributes*, which may be bitstrings e.g. the value of a key, or Boolean flags signalling properties of the object, e.g. whether the

key may be used for encryption (`CKA_ENCRYPT`[1]), or for encrypting other keys, for signing, verification, and other uses. New objects can be created by calling a key generation command, or by *unwrapping* an encrypted key packet using the `C_UnwrapKey` command, which takes a handle, a ciphertext and a *template* as input. A template is a partial description of the key to be imported, giving notably its length. The device attempts to decrypt the ciphertext using the key referred to by the handle. If it succeeds, it creates a new key on the device using the extracted plaintext and the template, and returns a new handle.

Observe that a padding check immediately following the decryption could give rise to an oracle that may be used to determine the value of the newly stored key. To test for such an oracle on a device, we create a key with the `CKA_UNWRAP` attribute set to allow the `C_UnwrapKey` operation, create encrypted key packets with deliberately placed padding errors, call the function on these ciphertexts and observe the return codes. For the case of asymmetric key unwrapping, constructing test ciphertexts is easy since the public key of the pair is always obtainable via a query to the PKCS#11 interface. For symmetric key unwrapping, it is not quite so trivial since the device may create unwrapping keys marked with the Boolean key attribute `CKA_SENSITIVE` which prevents them from being read via the PKCS#11 interface. In this case there are various tricks we can use: we can try to set the attribute `CKA_ENCRYPT` and then use the PKCS#11 function `C_Encrypt` to construct the test packets if a suitable mode is available, or if the device does not allows this, we can explicitly try to create a key with `CKA_SENSITIVE` set to false, assuming the same unwrap algorithm will be used as for sensitive keys. In the event, we were always able to find some way to do this with the devices under test.

3.1 Smartcards and Security Tokens

In Table 2 we give results from implementing the attacks on all the commercially available smartcards and USB tokens we were able to obtain that offer a PKCS#11 interface and support the unwrap operation. A tick means not only that we were able to construct a padding oracle, but that we were actually able to execute the attack and extract the correct encrypted key. A cross notes that the attack fails. We explain these failures below. Not applicable (N/A) means that the token did not support the cryptographic mechanisms and/or unwrap modes required for this attack. Note that relatively few devices support unwrap under symmetric key algorithms. We tested the attacks using both token keys and session keys for the unwrapping. The exact semantics of the difference between these key types is not completely clear from the standard: there is an attribute `CKA_TOKEN` which when set to true indicates a token key and when false indicates a session key. Session keys are destroyed when the session is ended, whereas token keys persist. However, we have noticed that devices often enforce very different policies for token keys and session keys, so it seemed pertinent to test both types.

In Table 3 we give the class of padding oracle found in each device in the PKCS#1 v1.5 case. To obtain this table we construct padded plaintexts with a single padding error and observed the return code from the token (the exact return codes are in the appendix, Table 4). Note that we give separate entries for token and session keys in this table only when there is a difference in the device's behaviour in the two cases. We report median attack time, computed from the results of table 1 and from a measure of the unwrap rate of the hardware. Notice how the tenfold improvement in median attack time of our modified algorithm makes attacks even against FFT oracles on slow devices quite practical. Unwrap calls using session keys are often many times faster than token keys though it is not clear why, unless perhaps these devices are carrying out session key operations in the driver software rather than on the card.

We will briefly discuss each line of Table 2 in turn. The **Aladdin eToken Pro** supports both unwrapping modes required, though the `CBC_PAD` unwrap mode does not conform to the standard: a

[1]Throughout the paper we will refer to commands, attributes, return codes and mechanisms by their names as defined in the PKCS#11 standard, so `C_` prefixes a (cryptoki) command, `CKA_` prefixes a cryptoki attribute, `CKR_` prefixes a cryptoki return code and `CKM_` prefixes a cryptoki mechanism.

Device	Token		Session	
	Oracle	Time	Oracle	Time
Aladdin eTokenPro	FTT	21m	FTT	17m
Gemalto Cyberflex	FFT	92m	N/A	N/A
RSA Securid 800	TTT	13m	N/A	N/A
Safenet Ikey 2032	FTT	88m	FTT	17m
Siemens CardOS	TTT	21m	FFT	89s

Table 3: Oracle Details and Median Attack Times

block containing a final byte of `0x00` is accepted. According to the standard, if the final byte of the plaintext is zero and it falls at the end of a block, then an entire block of padding should be added (see section 2). This causes a small problem for the attack since it gives us an extra possibility for the last byte, but we easily adapted the attack to take account of this. The PKCS#1 v1.5 padding implementation ignores zeros in the first 8 bytes of the padding and gives a separate error when the length of the extracted key does not match the requested one (`CKR_TEMPLATE_INCONSISTENT`). Based on this we can build an FTT oracle. The **Feitian** tokens do not support `CBC_PAD` modes. They also do not implement PKCS#1 v1.5 padding correctly as shown in Table 4: in our tests, any block with `0x02` in the second byte was accepted, except for very large values (e.g. for one key, anything between `0x00` and `0xE2` in the first byte was accepted). The result is that the attack does not succeed. The **Gemalto Cyberflex** smartcard does not allow unwrapping under symmetric keys. However, it seems to implement standard PKCS#1 v1.5 padding correctly, and the Bleichenbacher attack succeeds (FFT oracle, since the length is ignored). The **RSA SecurID** device does not support unwrapping using symmetric keys, hence the Vaudenay attack is not possible. However, the Bleichenbacher attack works perfectly. In fact, the RSA token implements a perfect TTT oracle. The device also supports OAEP, but not in a way that prevents the attack (see next paragraph). The **Safenet ikey2032** implements an asymmetric key unwrapping. The padding oracle derived is more accepting than the Bleichenbacher oracle since the 0s in the first 8 bytes of the padding string are ignored (FTT oracle). The **SATA DKey** does not implement standard padding checks. In `CBC_PAD` mode, only the last byte is checked: it seems that as long as the last byte n is less than the number of bytes in a block, the padding is accepted and the final n bytes discarded. This means we cannot use the attack to recover the whole key, just the final byte. In PKCS#1 v1.5 mode, many incorrectly padded blocks were accepted, and we were unable to deduce the rationale. For example, any block with the *first* byte equal to `0x02` is accepted. The wide range of accepted blocks prevents the attack. The **Siemens CardOS** supports only unwrapping under asymmetric keys. The Bleichenbacher attack works perfectly: with token keys the oracle is TTT, while with session keys it is FFT.

Attacking OAEP Mode Unwrapping A solution to the Bleichenbacher attack is to use OAEP mode encryption, which was first added to PKCS#1 in v2.0 (1998) and is recommended for all new applications since v2.1 (2002). RSA OAEP was included as a mechanism in PKCS#11 in version 2.10 (1999). However, out of the tokens tested (all of which are currently available products), only one, the RSA SecureID, supports OAEP encryption. The standard PKCS#1 v2.1 notes that it is dangerous to allow two mechanisms to be enabled on the same key [23, p. 14], since "an opponent might be able to exploit a weakness in the implementation of RSAES-PKCS1-v1_5 to recover messages encrypted with either scheme.". An examination of the developer's manual for the RSA SecurID reveals that for private keys generated by the token, the relevant attribute "`CKA_ALLOWED_MECHANISMS` is always set to the following mechanism list : `CKM_RSA_PKCS`, `CKM_RSA_PKCS_OAEP`, and `CKM_RSA_X_509.`". We created a key wrapped under OAEP and then performed Bleichenbacher's attack on it using a PKCS#1 v1.5 unwrap oracle. The attack is only slightly complicated by the fact that the initial encrypted block

does not yield a valid block when decrypted, requiring us to use the 'blinding phase' where many ciphertexts are derived form the original to obtain one that passes the padding oracle. In our tests this added only a few hundred seconds to the attack.

3.2 HSMs

Hardware Security Modules are widely used in banking and similar sectors where a large amount of cryptographic processing has to be done securely at high speed (verifying PIN numbers, signing transactions, etc.). A typical HSM retails for around 20 000 Euros hence is unfortunately too expensive for our laboratory budget. HSMs process RSA operations at considerable speed: over 1000 decryptions per second for 1024 bit keys. Even in the case of the FFF oracle, which requires 12 000 000 queries, this would result in a median attack time of 12 000 seconds, or just over three hours.

We hope to be able to give details of HSM testing soon.

3.3 Estonian ID Card

Estonia's Citizenship and Migration Board completed the issuing of more than 1 million national electronic ID (eID) cards in 2006 [15]. The eID is the primary national identification document in Estonia and it is mandatory for all Estonian citizens and alien residents 15 years and older to have one [9]. The card contains two RSA key pairs [12]. One key pair is intended to be mainly used for authentication (e.g., for mutual authentication with TLS/SSL) but can also be used for encrypting and signing email (e.g., with S/MIME). The other key pair is attributed only to be used for digital signatures. Only this latter key pair can be used for legally binding digital signatures [15]. Since January 1, 2011, the eID cards contain 2048 bit RSA keys, therefore these cards comply with NIST's recommendation [17]. However, cards issued before January 1, 2011 continue to use 1024 bit keys.

Attack Vector Unlike the cryptographic devices discussed above, the Estonian eID card does not allow the import of keys, so our attack here does not rely on the unwrap operation. Instead we consider attacks using the padding oracle provided by the decryption function of the DigiDoc software, part of the official ID software package developed by the Estonian Certification Center, Estonia's only CA [10]. We note that the attack succeeds with any application that returns whether decryption with the eID card succeeds. Our experiments were conducted using the Java library of DigiDoc, called *JDigiDoc*. DigiDoc encrypts data using a hybrid encryption scheme, where a 128-bit AES key is encrypted under a public key. First we tested the Estonian ID card's decryption function using raw PKCS#11 calls and confirmed that it checks padding correctly. We then observed that with the default configuration, when attempting to decrypt, e.g., an encrypted email, JDigiDoc writes a log file of debug information that includes the padding errors for the 128-bit AES key that is encrypted under the public key. This behavior has been observed with JDigiDoc version 2.3.19, and the latest version (3.6.0.157) does not seem to change it. Any application built on JDigiDoc, that reveals whether decryption succeeds, e.g., by leaking the contents of the log file, provides an attacker with a suitable padding oracle. The information in JDigiDoc's log file gives an attacker access to essentially an FFT oracle but with additional length information. The length information allows us to adjust the $2B$ and $3B - 1$ bounds used in the attack, though in our experiments this made little difference.

In tests, the Estonian ID card, using 2048 bit keys, was able to perform 100 decryptions in 340 seconds. This means that for our optimised attack, where 28 300 decryptions are required, we would need about 96 200 seconds, or about 27 hours to decrypt an arbitrary valid ciphertext. For ID cards using 1024 bit keys, each decryption should be four times faster, while 49 000 decryptions are required; therefore we estimate a time of about 41 700 seconds, or about 11 hours and 30 minutes to decrypt an arbitrary valid ciphertext. To forge a signature, we require, due to the extra blinding step, a mean of 109 000 oracle calls and a median of 69 000 oracle calls to get a valid signature on an arbitrary

message, giving an expected time of 103 hours on a 2048 bit Estonian eID. On a card using 1024 bit keys, we require a mean of 203 000 calls and a median of 126 000 calls; therefore expect to sign an arbitrary message in around 48 hours.

4 Countermeasures

A general countermeasure to the Bleichenbacher and Vaudenay attacks has been well known for years: use authenticated encryption. There are no such modes for symmetric key encryption in the current version of PKCS#11, but version 2.30, which is still at the draft stage, includes GCM and CCM (mechanisms `CKM_AES_GCM` and `CKM_AES_CCM`). While these modes have their critics [22], they do in theory provide secure authenticated encryption and hence could form the basis of secure symmetric key unwrap mechanisms. Unfortunately, in the current draft (v7), they are given only as modes for `C_Encrypt`. Adoption of these modes for `C_UnwrapKey` would provide a great opportunity to give the option of specifying authenticated data along with the encrypted key to allow secure transfer of attributes between devices. This would greatly enhance the flexibility of secure configurations of PKCS#11. To prevent the Bleichenbacher attack one must simply switch to OAEP, which is already in the standard. PKCS#11 should follow PKCS#1's long-held position of recommending OAEP exclusively for all new applications. Care must also be taken to remind developers not to allow the two modes to be used on the same key, as is the case in RSA's own SecureID device. In fact, the minutes of the 2003 PKCS workshop suggest that there was a consensus to include the single mechanism recommendation in version 2.20 [20], but it does not appear in the final draft. Note that care must be taken when implementing OAEP as otherwise there may also be a padding oracle attack which is even more efficient than our modified Bleichenbacher attack [14], though we are yet to find such an oracle on a PKCS#11 device.

If unauthenticated unwrap modes need to be maintained for backwards compatibility reasons, there are various options available. For the CBC case, Black and Urtubia note that the 10^* padding, where the plaintext is followed by a single 1 bit and then only 0 bits until the end of the block, leaks no information from failed padding checks while still allowing length of the plaintext to be determined unambiguously [1]. Paterson and Watson suggest a refinement that additionally preserves a notion of indistinguishability, by ensuring that no padded blocks are invalid [18]. They also give appropriate security proofs for the two schemes. If PKCS#1 v1.5 needs to be maintained, we have seen that an implementation of the padding check that rejects anything other than a conforming plaintext containing a key of the correct length with a single error code gives the weakest possible (FFF) oracle. This may be enough for some applications, but one is well advised to remember the maxim that attacks only get better, never worse. An alternative approach would be to adopt 'SSL style' countermeasures, proceeding to import a randomly generated key in the case where a block contains invalid padding. However, this may not fix the hole: if an attacker is able to replay the same block and detect that two different keys have been imported, he knows there is a padding error. One could also decide to ignore padding errors completely and always import just the number of bytes corresponding to the size of the key required, but this looks dangerous: if the same block can be passed off as several different kinds of key, this might open the possibility of attacking weaker algorithms to obtain keys for stronger ones. Thus it seems clear that authenticated encryption is by far the superior solution.

We detail manufacturer responses in Appendix C. There is a broad spectrum: while some manufacturers offer mitigations and state a clear need to get authenticated encryption into the standard and adopted as soon as possible, others see their responsibility as ending as soon as they conform to the PKCS#11 standard, however vulnerable it might be.

5 Conclusions

We have demonstrated a modified version of the Bleichenbacher RSA PKCS#1 v1.5 attack that allows the 'million message attack' to be carried out in a few tens of thousands of messages in many cases. We have implemented and tested this and the Vaudenay CBC attack on a variety of contemporary cryptographic hardware, enabling us to determine the value of encrypted keys under import. We have shown that the way the `C_UnwrapKey` command from the PKCS#11 standard is implemented on many devices gives rise to an especially powerful error oracle that further reduces the complexity of the Bleichenbacher attack. In the worst case, we found devices for which our algorithm requires a median of only 3 800 oracle calls to determine the value of the imported key. Vulnerable devices include eID cards, smartcards and USB tokens.

While some theoreticians find the lack of a security proof sufficient grounds for rejecting a scheme, some practitioners find the absence of practical attacks sufficient grounds for continuing to use it. We hope that the new results with our modified algorithm will prompt editors to reconsider the inclusion of PKCS#1 v1.5 in contemporary standards such as PKCS#11.

References

[1] John Black and Hector Urtubia. Side-channel attacks on symmetric encryption schemes: The case for authenticated encryption. In Dan Boneh, editor, *USENIX Security Symposium*, pages 327–338. USENIX, 2002.

[2] D. Bleichenbacher. Chosen ciphertext attacks against protocols based on the RSA encryption standard. In *Advances in Cryptology: Proceedings of CRYPTO '98*, volume 1462 of *LNCS*, pages 1–12, 1998.

[3] Mike Bond and George French. Hidden semantics: why? how? and what to do? Presentation at Fourth Analysis of Security APIs workshop (ASA-4), July 2010.

[4] Matteo Bortolozzo, Matteo Centenaro, Riccardo Focardi, and Graham Steel. Attacking and fixing PKCS#11 security tokens. In *Proceedings of the 17th ACM Conference on Computer and Communications Security (CCS'10)*, Chicago, Illinois, USA, October 2010. ACM Press.

[5] J. Clulow. On the security of PKCS#11. In *5th International Workshop on Cryptographic Hardware and Embedded Systems (CHES 2003)*, pages 411–425, 2003.

[6] Jean Paul Degabriele and Kenneth G. Paterson. On the (in)security of ipsec in mac-then-encrypt configurations. In Ehab Al-Shaer, Angelos D. Keromytis, and Vitaly Shmatikov, editors, *ACM Conference on Computer and Communications Security*, pages 493–504. ACM, 2010.

[7] S. Delaune, S. Kremer, and G. Steel. Formal analysis of PKCS#11. In *Proceedings of the 21st IEEE Computer Security Foundations Symposium (CSF'08)*, pages 331–344, Pittsburgh, PA, USA, June 2008. IEEE Computer Society Press.

[8] M. Dworkin. Recommendation for block cipher modes of operation: Modes and techniques. NIST Special Publication 800-38A, December 2001.

[9] Estonian Certification Center. The estonian ID card and digital signature concept, principles and solutions. `http://www.id.ee/public/The_Estonian_ID_Card_and_Digital_Signature_Concept.pdf`, March 2003.

[10] Estonian Informatics Center. Estonian ID-software. `https://installer.id.ee/?lang=eng`.

[11] R. Housley. Cryptographic Message Syntax (CMS). RFC 5652 (Standard), September 2009.

[12] ID Süsteemide AS. EstEID specification v2.01. `http://www.id.ee/public/EstEID_Spetsifikatsioon_v2.01.pdf`.

[13] T. Jager and J. Somorovsky. How to break xml encryption. In *Proceedings of the 18th ACM Conference on Computer and Communications Security (CCS)*, pages 413–422, 2011.

[14] James Manger. A chosen ciphertext attack on RSA optimal asymmetric encryption padding (OAEP) as standardized in PKCS #1 v2.0. In Joe Kilian, editor, *Advances in Cryptology CRYPTO 2001*, volume 2139 of *Lecture Notes in Computer Science*, pages 230–238. Springer Berlin / Heidelberg, 2001.

[15] Tarvi Martens. eID interoperability for PEGS, national profile estonia, European Commission's IDABC programme. `http://ec.europa.eu/idabc/en/document/6485/5938`, November 2007.

[16] Chris J. Mitchell. Error oracle attacks on CBC mode: Is there a future for CBC mode encryption? In J. et al. Zhou, editor, *ISC 2005*, number 3650 in LNCS, pages 244–258, 2005.

[17] National Institute of Standards and Technology. NIST special publication 800-57, recommendation for key management. `http://csrc.nist.gov/publications/PubsSPs.html`, March 2007.

[18] Kenneth G. Paterson and Gaven J. Watson. Immunising cbc mode against padding oracle attacks: A formal security treatment. In Rafail Ostrovsky, Roberto De Prisco, and Ivan Visconti, editors, *SCN*, volume 5229 of *Lecture Notes in Computer Science*, pages 340–357. Springer, 2008.

[19] K.G. Paterson and A. Yau. Padding oracle attacks on the ISO CBC mode encryption standard. In T. Okamoto, editor, *RSA '04 Cryptography Track*, number 2964 in LNCS, pages 305–323. Springer, 2004.

[20] Minutes from the April, 2003 PKCS workshop. Available at `ftp://ftp.rsa.com/pub/pkcs/03workshop/minutes.txt`, 2003.

[21] Juliano Rizzo and Thai Duong. Practical padding oracle attacks. In *Proceedings of the 4th USENIX conference on Offensive technologies*, WOOT'10, pages 1–8, Berkeley, CA, USA, 2010. USENIX Association.

[22] Phillip Rogaway. Evaluation of some blockcipher modes of operation. `http://www.cs.ucdavis.edu/~rogaway`, February 2011. Evaluation carried out for the Cryptography Research and Evaluation Committees (CRYPTREC) for the Government of Japan.

[23] RSA Security Inc., v2.1. *PKCS #1: RSA Cryptography Standard*, June 2002.

[24] RSA Security Inc., v2.20. *PKCS #11: Cryptographic Token Interface Standard.*, June 2004.

[25] T. Rosa V. Klima, O. Pokorny. Attacking RSA-based sessions in SSL/TLS. In *5th International Workshop on Cryptographic Hardware and Embedded Systems (CHES 2003)*, pages 426 – 440. Springer-Verlag, 2003.

[26] Serge Vaudenay. Security flaws induced by CBC padding - applications to SSL, IPSEC, WTLS ... In Lars R. Knudsen, editor, *EUROCRYPT*, volume 2332 of *Lecture Notes in Computer Science*, pages 534–546. Springer, 2002.

A Modified Bleichenbacher Algorithm

We present the algorithm of the optimised Bleichenbacher attack. It incorporates existing and new optimisations as presented in section 2.2. Notation is as before.

Step 1 - Initialization

Step 1.a - Blinding For an integer c, choose different random integers s_0 and check whether $c \cdot (s_0)^e \ mod\ n$ is PKCS conforming, by accessing the padding oracle. (If $c \ mod\ n$ is conforming then choose $s_0 \leftarrow 1$ instead.) For the first successful value s_0, set $c_0 \leftarrow c \cdot (s_0)^e \ mod\ n$, $M_0 \leftarrow \{[2B, 3B-1]\}$, $i \leftarrow 1$.

Step 1.b - Trimming M_0 Generate pairs of coprime integers and, for each pair (u, t), check whether $c_0 u^e t^{-e} \ mod\ n$ is PKCS conforming. For successful pairs $(u_1, t_1), (u_2, t_2), \ldots, (u_q, t_q)$, compute the lowest common multiple t' of $t_1, t_2, \ldots, t_q$, search for the smallest integer $u_{\min}$ and the largest integer $u_{\max}$ such that $c_0 u_{\min}^e t'^{-e} \ mod\ n$ and $c_0 u_{\max}^e t'^{-e} \ mod\ n$ are PKCS conforming. Set

$$\begin{aligned} a &\leftarrow 2B \cdot t'/u_{\min} \\ b &\leftarrow (3B-1) \cdot t'/u_{\max} \\ M_0 &\leftarrow \{[a, b]\}. \end{aligned}$$

Step 2 - Searching for PKCS conforming message

Step 2.a - Starting the search while Skipping Holes If $i = 1$, then search for the smallest positive integer $s_1 \geq \lceil (n + 2B)/b \rceil$ such that $c_0 \cdot s_1^e \ mod\ n$ is PKCS conforming. While searching for s_1, skip all values s' such that

$$(3B + jn)/a \leq s' < (2B + (j+1)n)/b$$

and do not access the padding oracle to check whether $c_0 \cdot s'^e \ mod\ n$ is PKCS conforming.

Step 2.b - Searching with more than one interval left If $i > 1$ and $|M_{i-1}| > 1$, then

Step 2.b.i - Parallel Threads Method If $|M_{i-1}| \leq P_{\max}$[2], then for each interval $I_j \in M_{i-1}$, start its own thread T_j following Step 2.c, for $j = 1, 2, \ldots, |M_{i-1}|$. The threads T_j take rounds making each one oracle call per round. If one of the threads finds a s_i such that $c_0 \cdot s_i^e \ mod\ n$ is PKCS conforming, then go to Step 3.

Step 2.b.ii - Beta Method [3] If $|M_{i-1}| > P_{\max}$, then search for the smallest integer $2 \leq \beta \leq \beta_{\max}$[4] such that for

$$s_i \leftarrow \beta s_{i-1} - (\beta - 1) s_0$$

$c_0 \cdot s_i^e \ mod\ n$ is PKCS conforming. If failed to find s_i, go to Step 2.b.iii.

Step 2.b.iii - No optimisation If Step 2.b.ii failed, then search for the smallest integer $s_i > s_{i-1}$ such that $c_0 \cdot s_i^e \ mod\ n$ is PKCS conforming. If such a s_i is found, go to Step 3.

Step 2.c - Searching with one interval left If $i > 1$ and $|M_{i-1}| = 1$, i.e., $M_{i-1} = \{[a, b]\}$, then choose small integers r_i, s_i such that

$$\begin{aligned} r_i &\geq 2\frac{bs_{i-1} - 2B}{n} \\ \frac{2B + r_i n}{b} &\leq s_i < \frac{3B + r_i n}{a} \end{aligned}$$

until $c_0 \cdot s_i^e \ mod\ n$ is PKCS conforming.

[2] In practice we take $P_{\max} = 40$.

[3] We did not use beta method for most experiments. (See section 2.5.)

[4] In practice we take $\beta_{\max} = 40$.

Step 3 - Narrowing the set of solutions After s_i is found, let

$$M_i \leftarrow \bigcup_{(a,b,r)} \{ \lceil \max(a, \lceil \frac{2B+rn}{s_i} \rceil), \min(b, \lfloor \frac{3B-1+rn}{s_i} \rfloor) \rceil \}$$

for all $[a,b] \in M_{i-1}$ and $\frac{as_i-3B+1}{n} \leq r \leq \frac{bs_i-2B}{n}$.

Step 4 - Computing Solution If $M_i = [a,a]$, then set $m \leftarrow a(s_0)^{-1} \ mod \ n$, and return m as solution of $m \equiv c^d \ mod \ n$. Otherwise, set $i \leftarrow i+1$ and continue with Step 2.b or Step 2.c.

B Actual Padding Errors Reported by Smartcards and USB Tokens

Table 4 reports actual padding errors returned by the devices we tested.

Device	First byte not `0x00`	Second byte not `0x02`	`0x00` in first 8 bytes padding	No `0x00` from byte 3 to 128	Length incorrect
Aladdin eToken PRO	1	1	4	1	4
Feitian epass 2000	0	5	5	5	0
Feitian epass 3003	0	3	5	5	5
Gemalto Cyberflex	2	2	2	2	0
RSA SecureID 800	1	1	0	0	0
Safenet Ikey 2032	1	1	4	1	4
SATA Dkey (session)	1	0	5	5	1
SATA Dkey (token)	1	1	5	5	1
Siemens CardOS (session)	5	5	5	5	0
Siemens CardOS (token)	5	5	0	0	5

Table 4: Variations found on PKCS#1 v1.5 Padding Tests. Error 0 = `CKR_OK` (key is imported), Error 1 = `CKR_ENCYRYPTED_DATA_INVALID`, Error 2 = `CKR_WRAPPED_KEY_INVALID`, Error 3 = `CKR_DATA_LEN_RANGE`, Error 4= `CKR_TEMPLATE_INCONSISTENT`, Error 5 = `CKR_FUNCTION_FAILED`, `CKR_GENERAL_ERROR`, `CKR_DEVICE_ERROR` or similar.

C Manufacturer Reaction

We have notified all manufacturers of our findings and we summarize their reactions so far.

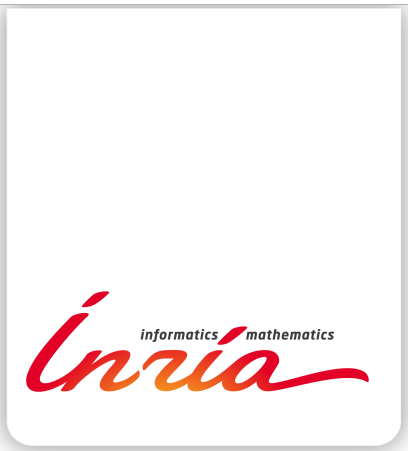
SafeNet is planning to release a security bulletin where they confirm the vulnerability on eToken Pro, eToken Pro Smartcard, eToken NG-OTP, eToken NG-FLASH, iKey 2032 using Aladdin eToken PKI Client or SafeNet Authentication Client software. As a workaround they suggest to use SafeNet Authentication Client 8.0 or later to enable PKCS#1 v2.1 padding for RSA and to avoid wrapping symmetric keys using other symmetric keys. They plan enhancements in their products for enabling symmetric keys wrapping with other symmetric keys using GCM and CCM modes of operation (discussed in section 4). They also plan to add a key wrapping policy that enforces the usage of only GCM and CCM modes of operation for symmetric encryption, and PKCS#1 v2.1 padding for RSA encryption.

RSA recognises that an attacker can obtain the corresponding plaintext through a padding oracle attack against RSA SecureID faster than would be possible with standard Bleichenbacher attack. They however claim that "this attack is unnecessary since the prerequisites to the attack are already enough to call `C_UnwrapKey` and `C_GetAttributeValue` and receive the same plaintext". Instead,

they regard these flaws as incomplete compliance with the standard and they are planning to fix this. Our perspective is that (1) full compliance with the standard would only slow down the attacks and not prevent them; (2) the attacker could have indirect attacks to the unwrapping functionality without accessing other functionalities such as `C_GetAttributeValue` and without knowing the PIN, e.g. though a network protocol

Siemens has also recognised the flaws and we have been informally told that they have fixed the verification of the padding and added a check of the obtained plaintext with respect to the given key template in the most recent version.

We filed a vulnerability report of our attack on the Estonian eID card to the Estonian Certification Center. They showed concern about the vulnerability of the card we reported and informed CERT Estonia about the flaw. However, according to the Estonian Certification Center the authentication certificate is mainly used for authentication with SSL (in 95% of the cases), and our attack would be too slow to forge an SSL client response before a server timeout. At the time of our communication they had not decided on any countermeasures. The most recent release (v3.6.0.157) of digiDoc does not change the default output to the debug file.



RESEARCH CENTRE
PARIS – ROCQUENCOURT

Domaine de Voluceau, - Rocquencourt
B.P. 105 - 78153 Le Chesnay Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399

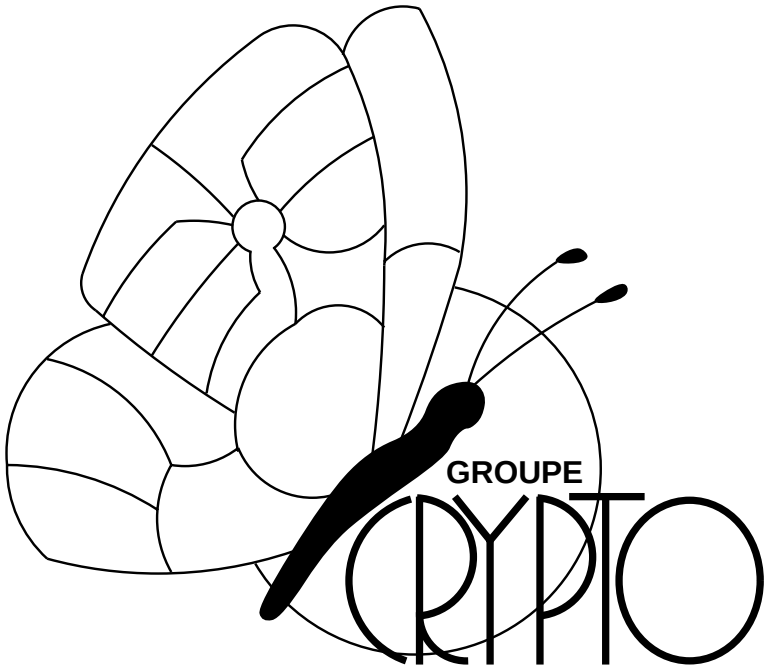
UCL Crypto Group Technical Report Series

UCL
Université
catholique
de Louvain

REGARDS

A practical implementation of the timing attack

J.-F. Dhem, F. Koeune, P.-A. Leroux, P. Mestré,
J.-J. Quisquater and J.-L. Willems



http://www.dice.ucl.ac.be/crypto/

Technical Report
CG–1998/1

Place du Levant, 3
B-1348 Louvain-la-Neuve, Belgium

Phone: (+32) 10 472541
Fax: (+32) 10 472598

A practical implementation of the timing attack*

J.-F. Dhem[1)], F. Koeune[3)], P.-A. Leroux[3)], P. Mestré[2)], J.-J. Quisquater[3)] and J.-L. Willems[3)]

June 15, 1998

[1)] Belgacom Multimedia & Infohighways,
Bld E. Jacqmain 177,
B-1030 Brussels, Belgium
E-mail: dhem@belbone.be

[2)] PWI,
Rue du Planeur, 10, B-1130 Brussels, Belgium
E-mail: mestre.p@banksys.be

[3)] Département d'Électricité (DICE), Université catholique de Louvain
Place du Levant, 3, B-1348 Louvain-la-Neuve, Belgium
E-mail: fkoeune,leroux,jjq,willems@dice.ucl.ac.be

Abstract. When the running time of a cryptographic algorithm is non-constant, timing measurements can leak informations about the secret key. This idea, first publicly introduced by Kocher, is developed here to attack an earlier version of the CASCADE smart card†. We propose several improvements on Kocher's ideas, leading to a practical implementation that is able to break a 512-bit key in a few minutes, provided we are able to collect 300 000 timing measurements (128-bit keys can be recovered in a few seconds using a personal computer and less than 10 000 samples). We therefore show that the timing attack represents an important threat against cryptosystems, which must be very seriously taken into account.

Keywords: timing attack, cryptanalysis, RSA, smart card.

*This work has been done when J.-F. Dhem and P. Mestré were research assistants at the UCL Crypto Group.

†Later modified to resist against it.

CG–1998/1

1 Introduction

Implementations of cryptographic algorithms often perform computations in non-constant time, due to performance optimizations. If such operations involve secret parameters, these timing variations can leak some information and, provided enough knowledge of the implementation is at hand, a careful statistical analysis could even lead to the total recovery of these secret parameters (fig. 1).

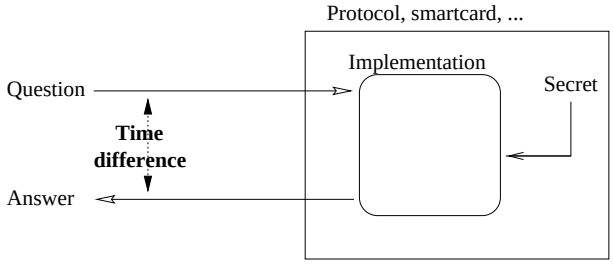


Figure 1: The timing attack principle.

This idea was first presented by Kocher [Koc96], who laid the foundations of the basic ideas exploited in this paper. However, the results of Kocher were quite theoretical[‡] and we found them rather difficult to exploit in practice. This paper presents an effective and efficient attack of a cryptographic algorithm running on a smart card. The first practical timing attack to our knowledge was described at the rump session of CRYPTO'97 by Lenoir. Our paper, however, develops quite different ideas.

Another problem of the attack presented by Kocher is that the attacker needs a very detailed knowledge of the implementation of the system he is attacking, as he has to be able to compute the partial timings due to the known part of the key. As for this paper, the knowledge needed is very limited, which makes the attack quite general and easy to carry out.

Last but not least, some completely new ideas, such as the attack of the square rather than the multiply, are presented.

We begin by presenting the model we are attacking and the characteristics it must present to be vulnerable. We then describe the attack that was carried out, as well as its results and possible improvements. Finally, we describe some countermeasures that would allow to defeat it.

[‡]Kocher identified several targets for the timing attack and ran simulations to determine what the success rate would be for an attack of the modular multiplication, but did apparently not carry out the attack itself.

We try to present both a formal and an intuitive view of the timing attack, our goal being to make the principle of the attack easy to understand, but also to provide a detailed enough description to allow the reader to implement it without encountering major problems.

2 The general framework

We here give the characteristics that the system must present to be vulnerable, and briefly formalize the model in which our attack will be drawn.

Given a message m as input, an algorithm A performs a computation (that we call a signature) using a secret key k. We note:

M, the set of messages,
K, the set of keys,
S, the set of signed messages,
$A : M \times K \to S : (m, k) \to A(m, k)$, the signature of m with the secret key k,
$B = \{0, 1\}$,
$T : M \times K \to \mathcal{R} : m \to t = T(m, k)$, the time taken to compute $A(m, k)$.
$O : M \to B : m \to O(m)$, an oracle, based on our knowledge of the implementation, that provides us some information about the details of the computation of $A(m, k)$.

Remark: It may look surprising that the oracle does not depend on the key k, although the computation of $A(m, k)$ does, but this is precisely the idea of this paper: typically, we want to build a decision criterion (formalized by the oracle) that will be meaningful or not, *depending on the actual value of some bit of the key*. By observing the meaningfulness of our criterion, we will deduce the bit value.

The scenario of our attack is the following: Eve disposes of a sample of messages and, for each of them, the time needed to compute the signature of the message with the key k. Her goal is to recover k, which can thus be considered as an unknown parameter rather than as a variable. To simplify our notations, we will thus simply note $T(m)$ instead of $T(m, k)$.

To attack the bit i of the key k, Eve will use an oracle O to build two subsets of messages $M_1, M_2 \subseteq M$. We will denote the corresponding timings by the functions:

$$F_1 : M_1 \to \mathcal{R} : m \to F_1(m) = T(m)$$

$$F_2 : M_2 \to \mathcal{R} : m \to F_2(m) = T(m)$$

Suppose these two functions have the following properties:

$$\begin{cases} \text{If } k_i = 0, \text{ then } & F_1 \text{ is a random variable } v_1^0 \\ & F_2 \text{ is a random variable } v_2^0 \\ \\ \text{If } k_i = 1, \text{ then } & F_1 \text{ is a random variable } v_1^1 \\ & F_2 \text{ is a random variable } v_2^1 \end{cases}$$

and suppose that, for a parameter of these random variables $\phi(v)$ (e.g. the mean or the variance) we have:

$$\phi(v_1^0) = \phi(v_2^0) \quad \text{and} \quad \phi(v_1^1) > \phi(v_2^1)$$

then with the following statistical test:

$$H_0 : \phi(F_1) \stackrel{?}{=} \phi(F_2)$$
$$H_1 : \phi(F_1) \stackrel{?}{>} \phi(F_2)$$

we deduce that if H_0 is accepted with error probability α, then $i = 1$ with error probability α.

In other words, this means that Eve is able to construct two samples of messages and two functions whose statistical behaviours will depend on the actual value of the bit i. By observing the relative behaviours of the two functions, Eve will be able to determine, with a certain error probability, the value of the bit i. Of course, couples of ciphertexts / decryption timings, with the same properties, could also be used.

3 Towards a practical attack

3.1 The implementation

We have attacked an RSA computation (without CRT), performed in an earlier version of the cryptographic library we developed for the CASCADE [Cas] smart card:

The computation in the smart card was: $m^k \bmod n$.

The algorithm is the left to right square and multiply (fig. 3.1).

Both the multiplication and the square are done using the Montgomery algorithm. The time for a Montgomery multiplication is constant, independently of the factors, except that, if the intermediary result of the multiplication is greater than the modulus, then an additional subtraction (called a reduction) has to be performed.

```
x = m
for i = n − 2 downto 0
    x = x^2
    if (k_j == 1) then
        x = x · m
endfor
return x
```

Figure 2: Square and multiply

3.2 A first attempt: attacking the multiply

The most obvious way to take advantage of this knowledge is to aim our attack at the *multiply* step of the square and multiply. The idea is the following:

We start by attacking k_2, the second bit[§] (MSB first) of the secret key. Performing the Montgomery algorithm step-by-step, we see that, if that bit is 1, then the value $m \cdot m^2$ will have to be computed during the square and multiply.

Now, for some messages m (those for which the intermediary result of the multiplication will be greater than the modulus), an additional reduction will have to be performed during this multiplication, while, for other messages, that reduction step will not be necessary. So, we are able to divide our set of samples in two subsets: one for which the computation of $m \cdot m^2$ will induce a reduction and another for which it will not. If the value of k_2 is really 1, then we can expect the computation times for the messages from the first set to be slightly higher than the corresponding times for the second set.

On the other hand, if the actual value of k_2 is 0, then the operation $m \cdot m^2$ will not be performed. In this case, our "separation criterion" will be meaningless: there is indeed no reason for which a m inducing a reduction for the operation $m \cdot m^2$, would also induce a reduction for $m^2 \cdot m^2$, or for any other operation. Therefore, the separation in two subsets should look random, and we should not observe any significant difference in the computation times.

Let us rewrite this a little more formally:

The algorithm $A(m, k)$ could be split into $L(m, k)$ and $R(m, k)$ where $L(m, k)$ is the computation due to the additional reduction at the multiplication phase for bit k_2 and $R(m, k)$ the remaining computations. This gives for the computation times: $T(m) = T^L(m) + T^R(m)$, where $T^L(m)$, $T^R(m)$

[§]We can of course suppose that the first bit of the key is always 1.

are the times to compute $L(m,k)$ and $R(m,k)$ respectively.

The oracle O is:

$$O : m \to \begin{cases} 1 \text{ if } m \cdot m^2 \text{ is done with a reduction,} \\ 0 \text{ if } m \cdot m^2 \text{ is done without a reduction.} \end{cases}$$

As in section 2, define

$$\begin{aligned} M_1 &= \{m \in M : O(m) = 1\}, \\ M_2 &= \{m \in M : O(m) = 0\}, \\ F_1 &: M_1 \to R : m \to F_1(m) = T(m), \\ F_2 &: M_2 \to R : m \to F_2(m) = T(m). \end{aligned}$$

We have

$$\begin{cases} F_1 = T^R & \text{if } k_2 = 0 \\ F_1 = T^R + T^L & \text{if } k_2 = 1 \end{cases}$$

while,

$$F_2 = T^R$$

independently of the value of k_2.

Now, analyzing the mean as parameter ϕ, and testing:

$$\begin{aligned} H_0 &: \phi(F_1) \stackrel{?}{=} \phi(F_2) \\ H_1 &: \phi(F_1) \stackrel{?}{\neq} \phi(F_2) \end{aligned}$$

should reveal the value of k_2.

Once this value is known, we can simulate the computation up to the multiplication due to bit k_3, attack it in the same way as described above, and so on for the next bits.

3.3 Problems

Using the previous attack, we were able to recover 128-bit keys by observing samples of 50 000 timings.

However, this method is not fully satisfying. Mainly two problems arise: Firstly, the operations we observe are multiplications by a constant value m, and these operations seem to be much more correlated than expected. Rather surprisingly, we observed that, while the probability for an additional reduction to be necessary when the two factors and the modulus are random is about 0.17, this probability will, when the modulus and one factor are fixed, vary between 0 and 0.5, depending on modulus and factor (see figure 3). So,

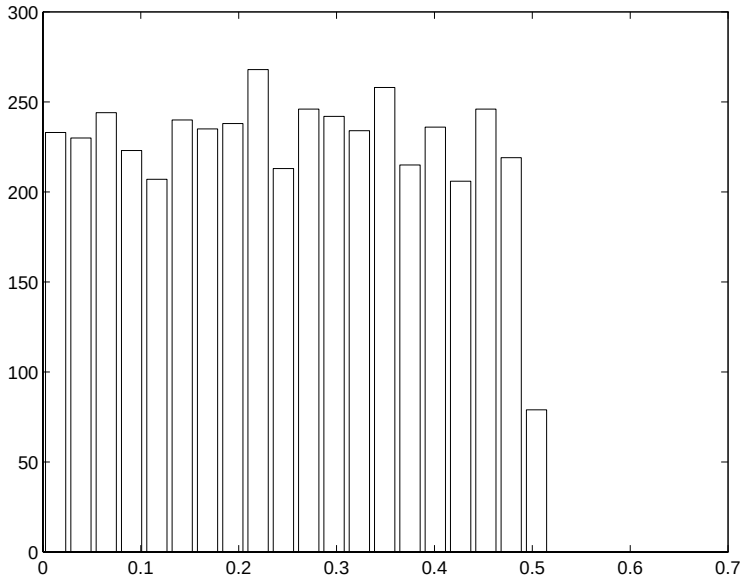


Figure 3: With fixed modulus, probability for an additional reduction to take place when multiplying by a constant factor. The test has been carried out with 4 500 factors and, for each of them, with 20 000 multiplications.

although it seems difficult to explain theoretically, our selection criterion seems to be highly biased.

Secondly, the decision we have to make is of the type: "are these two samples different *or not*". That is, we have to decide, on the basis of a finite set of measures, whether the differences we observe between the two sets is significant or not. Statistics can be of some help, but not as much as could be expected. As we said before, Montgomery multiplication by a constant seems to be a biased operation, so that the two subsets we build *always* appear different, even if the corresponding bit is 0. The answer to the statistical tests we tried is *always* positive, at a very high level of confidence. So, the question we have to ask is rather: "are these two samples 'very' different, or simply different ?".

Luckily, statistics can though be used to answer that question: we can simply decide that our two subsets are very different when the observed value for the statistic is 'very high', and that they are simply different when the value is 'not so high'. The problem will now be to decide what "very high" and "not so high" mean[¶], and we will have to tune up some swap value for

[¶]For example, attacking a 128-bit key using 50 000 samples and the χ^2 test, a typical

each key we attack. Some heuristics can help us in this tuning operation, but we will not describe them here, as there is a more efficient approach:

Aiming our attack at the *square* operation solves both of these problems.

3.4 Attacking the square

There is a more subtle way to take advantage of our knowledge of the Montgomery algorithm: instead of the multiplication phase, we could turn ourselves to the *square* phase.

The idea is quite similar to that of section 3.2: suppose we know the first $i-1$ bits of the key and attack the ith. We begin by executing the first $i-1$ steps of the square and multiply algorithm, stopping just before the possible - but unknown - multiplication by m due to bit k_i ; we denote by m_{temp} the temporary value we obtain.

First, we suppose k_i is set. If this is the case, the two next operations to be performed are

1. multiply m_{temp} by m,
2. square the result,

and both of these operations will be done using the Montgomery algorithm. We simply execute the multiplication and then, for the square, determine whether an additional reduction will be necessary or not. Doing this for every message, we divide our samples set in two subsets M_1 (additional reduction) and M_2 (no reduction).

Next, we suppose $k_i = 0$. In this case, no multiplication will take place, and the next operation will simply be

$$m_{temp}^2.$$

Once again, we divide the samples set in two subsets M_3 and M_4, depending on whether this square requires a reduction or not.

Clearly, only one of these separations makes sense, depending on the actual value of k_i. All we have to do now is to compare the separations: if the timing difference between M_1 and M_2 is more important than that between M_3 and M_4, then conclude $k_i = 1$, otherwise, conclude $k_i = 0$.

observed value for the statistic was about 4300, which is much, much higher than $\chi^2_{0.95}$. We can however decide that values above 4320 correspond to "very different", while values beyond 4320 correspond to "simply different". This may look tedious on a theoretical point of view, but works well in practice and allowed us to recover the secret key.

Back to formalization, to attack bit k_i knowing bits $k_0, \ldots, k_{i-1}$, we split the algorithm $A(m,k)$ into $L(m,k)$, which is the computations due to the additional reduction at the square phase at step $i+1$, and $R(m,k)$, the remaining computations.

Compute

$$m_{temp} = (m^b)^2 \qquad \text{where } b = k_0 k_1 \ldots k_{i-1}.$$

We need two oracles,

$$O_1 : m \to \begin{cases} 1 \text{ if } (m_{temp} \cdot m)^2 \text{ is done with a reduction,} \\ 0 \text{ if } (m_{temp} \cdot m)^2 \text{ is done without a reduction,} \end{cases}$$

$$O_2 : m \to \begin{cases} 1 \text{ if } (m_{temp})^2 \text{ is done with a reduction,} \\ 0 \text{ if } (m_{temp})^2 \text{ is done without a reduction.} \end{cases}$$

Define

$$\begin{aligned} M_1 &= \{m \in M : O_1(m) = 1,\} \\ M_2 &= \{m \in M : O_1(m) = 0,\} \\ M_3 &= \{m \in M : O_2(m) = 1,\} \\ M_4 &= \{m \in M : O_2(m) = 0,\} \\ F_k &: M_k \to R : m \to F_k(m) = T(m), \qquad \text{for } 1 \le k \le 4. \end{aligned}$$

If $k_i = 1$, we have

$$\begin{cases} F_1 = T^R + T^L \\ F_2 = T^R \\ F_3 = F_4 \end{cases} \qquad (= T^R + T^L \cdot O_1, \text{ but this is not so important})$$

and thus $\mu(F_1) > \mu(F_2)$, while $\mu(F_3) = \mu(F_4)$.

On the other hand, if $k_i = 0$, we have

$$\begin{cases} F_1 = F_2 \\ F_3 = T^R + T^L \\ F_4 = T^R \end{cases}$$

and thus $\mu(F_3) > \mu(F_4)$, while $\mu(F_1) = \mu(F_2)$.

Testing which of these conditions is true should reveal the value of k_i.

Remark: The last bit cannot be revealed by this attack and must thus be guessed.

This attack does not suffer from the problems mentioned in previous section:

- Firstly, the operation we are observing (i.e. the square) does not involve a constant factor, and its behaviour appears to be much less biased than for the multiplication.
- Secondly, we do not have anymore to decide whether a separation makes sense or not: we have now to compare two separations and decide which is the most significant. We are thus relieved of the difficult task to tune up an appropriate swap value for a given key.

Using this attack, we were able to recover 128-bit keys with 20 000 timings. Some keys were disclosed with only 12 000 timings.

4 Statistics

We have not yet said very much about the statistics we have to use to compare samples, and that is mainly because they were not very useful in practice. We tried several of the tools that statistics offers to compare two samples, such as the Chi-square, Student, Hotteling, and even a test from non-parametric statistics, the Wald-Wolfowitz [Sie56] test. None of them offered really efficient results. Chi-square and Student provided criteria upon which a right decision could be made, but a simple comparison of the means of the two populations allowed the same decision, with a similar, if not better, success rate.

There is, however, a possible use for statistics: as the Chi-square test, for example, does not seem to yield the same errors as the means comparison, it can be used to provide us with some sort of level of confidence in the goodness of our choices. When both tests agree on the value of some bit, it is more probably right than when they disagree, and this can help us to detect erroneous deductions. We implemented this in practice, and it appeared to be of some help, but did not produce any significant breakthrough in efficiency.

The reason for this uselessness of statistics is probably the one we mentioned before, that is, Montgomery multiplications with constant modulus are not independent events, and our decision criteria are thus biased[‖]. Perhaps a better understanding of *why* this bias appears would allow to derive an useful statistical test? Up to now, however, they seems to be limited to a role of confirmation.

[‖]This is also true for square, even it is at a much less extend than for multiplication by a constant factor.

5 Error-detection

One remarkable property of our attack is that it has an error-detection property. This is easy to understand on an intuitive point of view: remember that the attack basically consists in simulating the computations until some point, then build two decision criteria, with only one of them making sense, depending on the searched value, and finally decide the bit value by observing which criterion actually makes sense. Also note that each step of the attack relies on the previous ones (we need the previous bits values to simulate the computation).

Now, suppose we made an erroneous decision for the value of bit k_i. In the following step, we will not correctly simulate the computations, so that the value m_{temp} we will obtain will not be the one involved in step $i+1$. Our attempts to decide whether the Montgomery multiplications will involve an additional reduction or not will thus not make sense, and the criteria we will build will *both* be meaningless. This remains true for the following bits.

In practice, this translates to abnormally close values for the two separations: while, as long as the choices were right, the two separations were generally** easy to distinguish, one of them being clearly more significant than the other, they appear much more similar (and both bad) after an erroneous choice has been made. This fact is well illustrated in figure 4, showing the attack of a 512-bit key on the basis of 350 000 observations. The decision criterion is simply the difference between the mean times for the two subsets, and the graph shows the absolute value of $diff_1$ (the difference between M_1 and M_2) minus $diff_2$ (difference between M_3 and M_4). Clearly, an error has occured near bit 149.

Once an error has been detected, it is not difficult to take back, make a different choice for the last chosen bit, and go ahead a few steps to see if things go better; if they do not, then we go back two steps, change the bit value, and so on.

In practice, this error-correction scheme allowed us to reduce significantly the amount of measures needed. Samples of 10 000 timings, for example, were sufficient to recover 128-bit keys, and some of them were revealed with as few as 6 000 timings.

**There are however some tedious cases, were the two criteria are uneasy to differentiate although no error has been made. That is why it is better to wait until several contiguous low values are observed before to conclude to an error.

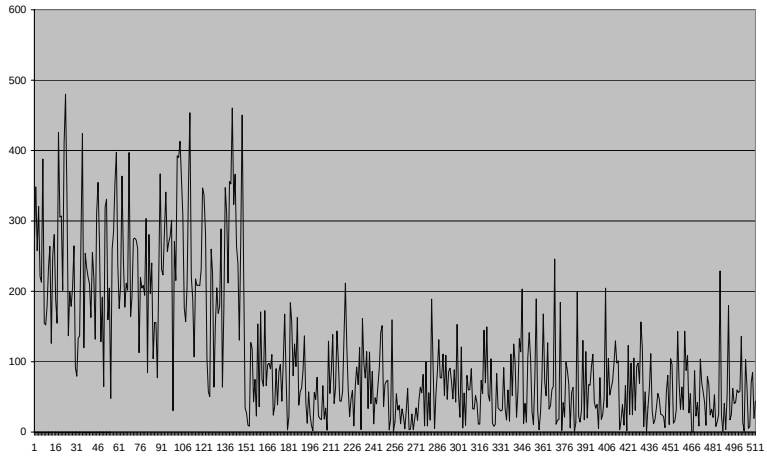


Figure 4: Detection of an error for a 512-bit key

6 Practical results

Our attack was first implemented in Visual C++ 4.2 on a 200 MHz PentiumPro PC under Windows NT.

Timings were collected using an emulator of the CASCADE smart card, that was able to monitor the number of cycles between two points. This seems to be quite a realistic scenario: the amount of measures required for a real attack of the electronic device would probably be slightly larger, to filter out additional noise, but we believe it should not grow too much.

With about 10 000 samples (couples messages, time for modular exponentiation), we were able to break 128-bit keys, at a rate of about 4 bits/s. The speed for a 512-bit key was of a little more than 1 bit/minute and approximately 350 000 samples were needed. The implementation was not optimized for speed.

Our results summarize as follows:

Key size	Result			
	without error correction		with error correction	
	sample size	speed	sample size	speed
64	1 500–6 500	> 20 bits/s	1 500–4 500	> 20 bits/s
128	12 000–20 000	2 bits/s	6 000–10 000	4 bits/s
256	70 000–80 000	1 bit/4s	15 000–50 000	1 bit/2s
512	±350 000	1 bit/65s		

However, these results correspond to a very high success rate: the error-correction algorithm we implemented was very simple and allowed us to correct errors only if they occur for a very small percentage of the bits. Experiments showed that the sample size grows very fast with the desired success rate (see e.g. figure 5).

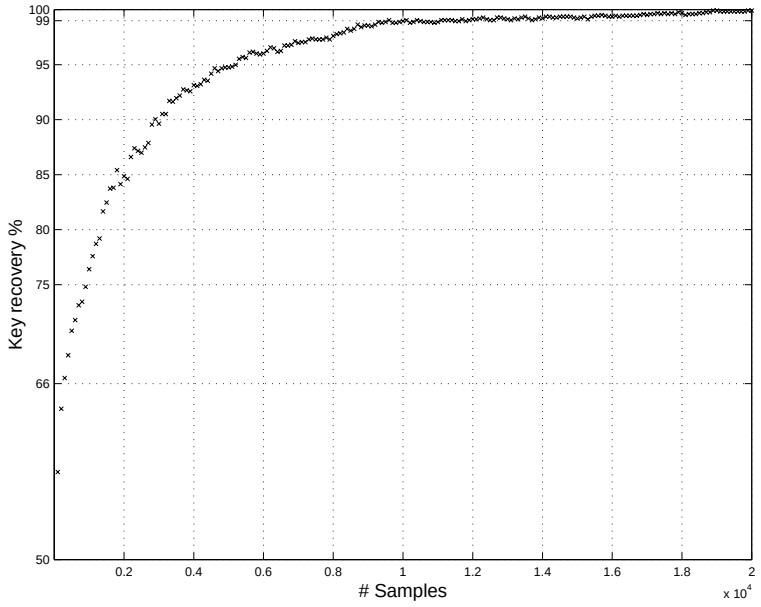


Figure 5: Sample size / success rate dependence

A more sophisticated algorithm, that would for example explore several choices when a possible error is detected, handle the case of two successive errors, ..., would probably resist to a higher error rate, thus allowing to reduce drastically the amount of measures needed.

As most of the computational effort consists in simulating the exponentiation steps for a large amount of data, the attack would also be very easy to parallelize. Experiments on a network of 20 Sparc Ultra-5 workstations (with is, nowadays, a reasonable computing power) showed that 512-bit keys could be recovered in few minutes.

7 Remarks about the attack

7.1 Accuracy of measures

The timing variation we are basing our attack on is that of one modular reduction. This timing is of course very small regarding the total computation. For example, a 512-bit exponentiation on the CASCADE chip takes about 7 400 000 cpu cycles, and the time for a modular reduction is 422 cycles!

The accuracy in measures is therefore of great importance. As the rounding effect induced by less accurate measures can be considered as noise, a greater amount of measures would still make the attack possible, but the sample size would rapidly grow.

7.2 Choice of messages

It is worth noting that we simply need to be able to determine the value of $O(m)$ for each message: we do not have to *build* messages for which the oracle will have a given value. This is important because many protocols will not accept to sign arbitrary message, but will require for these to have a specific format (e.g. to exhibit some redundancy). As long as we are able to trace the transformations preceding the modular exponentiation, the attack can be carried out.

7.3 Knowledge of the implementation

We also insist on the fact that we do not need too many details about the implementation of the cryptographic algorithm itself. All we have to know is that the exponentiation is square-and-multiply with Montgomery multiplication. It is amazing to note that one of the authors began with the timing-attack program before the CASCADE library was available for him. He thus decided to build his own exponentiation routine as a first target. When he finally received the CASCADE library, he discovered that his attack program did not need any modification to work against it.

7.4 Possible improvements

Even if the number of samples at disposal in not sufficient to discover the complete key, the attack is likely to reveal a part of it : we observed that samples twice as small as the required size for complete recovery could reveal 3/4 of the key before the first error occurs (as we have seen, nothing

significant is produced afterwards). Therefore, any method allowing Eve to guess a part of the key before carrying out the attack or to deduce the last bits once the first ones are know can dramatically improve performances.

Consider for example the frequent case where:

- the public exponent $e = 3$,
- the secret exponent k is calculated such that $k \cdot e = 1 \bmod (p-1)(q-1)$ (the important point is that we do not use the $lcm(p-1, q-1)$),
- p and q have the same l-bit length.

We have:

$$k \cdot 3 - s(p-1)(q-1) = 1, \quad \text{with} \quad s = 1 \text{ or } 2,$$

$$k \cdot 3 = \begin{cases} 1 + 1[n - (p+q) + 1] \\ 1 + 2[n - (p+q) + 1] \end{cases}$$

$$k = \begin{cases} [n - (p+q) + 2]/3 \\ [2n - 2(p+q) + 2]/3 \end{cases}$$

Because the length of n is twice the length of p and q, there is a great probability that the $l-3$ most significant bits (depending on the length of the propagation of a possible carry due to the subtraction by $p+q$) of the key k are the $l-3$ first bits of $n/3$. Eve can thus start the attack at the first unknown bit.

8 Other targets and further research

The attack could easily be extended against some variants of the square and multiply algorithm. The implementation of RSAREF, for example, processes the bits two by two, performing two square followed by a multiplication by 1, m, m^2 or m^3, depending on whether the bits are 00, 01, 10 or 11. The attack could quite easily be adapted to this case.

Other cryptosystems involving a modular exponentiation are of course subject to the same attack. Consider for example the Diffie-Hellman key exchange protocol: to build a common secret parameter, Alice and Bob exchange the values g^x and g^y, where g is public and x, y are secret values, known only by Alice and Bob, respectively, but often kept constant. The common parameter, that only Alice and Bob can compute, is obtained by computing $(g^y)^x$ or $(g^x)^y$.

Now, suppose Alice wants to discover Bob's secret parameter y. She chooses several random values $x_1, \ldots, x_N$, sends the values g^{x_i} (e.g. pretending to be someone different each time) and collects the corresponding response times (which can be, for example if Bob is a smart card, measured with very good accurancy). Clearly, the conditions of our attack are fullfilled. As a typical value for a Diffie-Hellman key size is 160 bits, a few thousand exchanges would suffice to discover the secret key.

Other protocol could probably be attacked in the same way. It must however be noted that the basis of the exponentiation (i.e., the parameter x in x^k) has to be known for the attack to be carried out as described here. Therefore, systems such as DSS, ... seem less vulnerable, although a more detailed study should have to be carried out.

One important weakness of our attack is that it cannot be carried out against systems using the Chinese Remainder Theorem for modular exponentiation. Kocher [Koc96] proposes some leads for a timing attack of the CRT, but it is not known whether such an attack would be practicable.

When developing the attack, we faced many difficulties on building a rigorous mathematical model explaining *why* things work. In fact, we encountered more than once the strange situation of building a model which should reveal some information, implementing it, and discovering that the system behaves differently than expected, *although the information is well revealed*. It seems that other researchers interested in the timing attack have faced the same problems with theory. A complete theoretical model would of course be useful, although we believe it is a real challenge.

9 Countermeasures

Three countermeasures come to mind when we try to protect ourselves against the above attack.

The first one is to modify the Montgomery algorithm so that an additional subtraction is always carried out, even if its result is simply discarded afterwards. This modification is very easy to carry out, does not decrease the performances very much and clearly defeats this attack. However, it cannot be guaranteed that it makes the system immune to *any* type of timing attack, only against those which exploit the reduction of the multiplication algorithm.

Another countermeasure, suggested by [Koc96], would be to use some blinding: before computing the modular exponentiation, choose a random

pair[††] (v_i, v_f) such that $v_f^{-1} = v_i^e$; multiply the message by $v_i \pmod n$ and multiply back the output by $v_f \pmod n$ to obtain the searched result. As Eve can no more simulate the internal computations, she can hardly exploit her knowledge of the timing measurements.

It is worth noting that the attack is quite general, in the sense that it was not directed against the peculiar case of a Montgomery multiplication, but against the fact that this algorithm is constant-time, except for a potential final reduction. This means that the use of other modular multiplication schemes, such as the standard versions of the Barrett or Quisquater algorithms, would not protect the system against our timing attack.

However, Dhem [Dhe98] recently proposed an improvement of these multiplications schemes, allowing several modular multiplications to be chained with only *one* extra reduction being performed after the last multiplication. This scheme seems to be especially interesting here, as it would suppress our attack's main target.

10 Conclusion

This paper shows that the timing attack represents a practical, important threat against cryptosystems implementations, namely in the case of a smart card, where the attacker can quite easily collect large amount of message decryptions.

It is important to note that the attack is quite general, in the sense that it does not require a very detailed knowledge of the implementation: all we have to know, besides some general hardware characteristics such as the word size, is that the modular exponentiation is done using the square and multiply and Montgomery algorithm. Computation details, timings necessary for specific operations, ..., are not necessary. This is an important improvement on Kocher's attack.

As shown in previous section, the attack is also general in the sense that it could have been directed against other classical modular multiplication schemes, if they are used in there standard form and not with Dhem's improvement.

In view of these results, the design of the CASCADE smart card has been modified to make it immune against the timing attack. It is however our belief that few smart cards take care of this, and that similar attacks could be successfully conducted against many of them.

[††] [Koc96] proposes a way to generate such pairs at a reasonable cost.

11 Acknowledgements

The authors wish to thank Gael Hachez for usefull comments and technical help, and HP Labs, Bristol, UK for the grant of the PCs which were used to carry out the attack.

References

[Cas] Cascade (Chip Architecture for Smart CArds and portable intelligent DEvices). Project funded by the European Community, see http://www.dice.ucl.ac.be/crypto/cascade.

[Dhe98] J.F. Dhem. *Design of an efficient public-key cryptographic library for RISC-based smart cards.* PhD thesis, Université catholique de Louvain - UCL Crypto Group - Laboratoire de microélectronique (DICE), May 1998.

[Koc96] P. Kocher. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In N. Koblitz, editor, *Advances in Cryptology - CRYPTO '96, Santa Barbara, California*, volume 1109 of *LNCS*, pages 104–113. Springer, 1996.

[Ler98] P.-A. Leroux. Timing cryptanalysis : Breaking security protocols by measuring transaction times. Master's thesis, Université catholique de Louvain - UCL Crypto Group, June 1998.

[RSA78] R.L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. In *Proc. Communications of the ACM*, volume 21, pages 120–126. ACM, February 1978.

[Sie56] S. Siegel. *Nonparametric Statistics.* McGraw-Hill, 1956.

[Wil98] J.-L. Willems. Timing attack of secured devices (in French). Master's thesis, Université catholique de Louvain - UCL Crypto Group, June 1998.

Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems

Paul C. Kocher

Cryptography Research, Inc.
607 Market Street, 5th Floor, San Francisco, CA 94105, USA.
E-mail: paul@cryptography.com.

Abstract. By carefully measuring the amount of time required to perform private key operations, attackers may be able to find fixed Diffie-Hellman exponents, factor RSA keys, and break other cryptosystems. Against a vulnerable system, the attack is computationally inexpensive and often requires only known ciphertext. Actual systems are potentially at risk, including cryptographic tokens, network-based cryptosystems, and other applications where attackers can make reasonably accurate timing measurements. Techniques for preventing the attack for RSA and Diffie-Hellman are presented. Some cryptosystems will need to be revised to protect against the attack, and new protocols and algorithms may need to incorporate measures to prevent timing attacks.

Keywords: timing attack, cryptanalysis, RSA, Diffie-Hellman, DSS.

1 Introduction

Cryptosystems often take slightly different amounts of time to process different inputs. Reasons include performance optimizations to bypass unnecessary operations, branching and conditional statements, RAM cache hits, processor instructions (such as multiplication and division) that run in non-fixed time, and a wide variety of other causes. Performance characteristics typically depend on both the encryption key and the input data (e.g., plaintext or ciphertext). While it is known that timing channels can leak data or keys across a controlled perimeter, intuition might suggest that unintentional timing characteristics would only reveal a small amount of information from a cryptosystem (such as the Hamming weight of the key). However, attacks are presented which can exploit timing measurements from vulnerable systems to find the entire secret key.

2 Cryptanalysis of a Simple Modular Exponentiator

Diffie-Hellman[2] and RSA[8] private-key operations consist of computing $R = y^x \bmod n$, where n is public and y can be found by an eavesdropper. The attacker's goal is to find x, the secret key. For the attack, the victim must compute $y^x \bmod n$ for several values of y, where y, n, and the computation time are known to the attacker. (If a new secret exponent x is chosen for each operation,

the attack does not work.) The necessary information and timing measurements might be obtained by passively eavesdropping on an interactive protocol, since an attacker could record the messages received by the target and measure the amount of time taken to respond to each y. The attack assumes that the attacker knows the design of the target system, although in practice this could probably be inferred from timing information.

The attack can be tailored to work with virtually any implementation that does not run in fixed time, but is first outlined using the simple modular exponentiation algorithm below which computes $R = y^x \bmod n$, where x is w bits long:

```
Let s_0 = 1.
For k = 0 upto w - 1:
  If (bit k of x) is 1 then
    Let R_k = (s_k · y) mod n.
  Else
    Let R_k = s_k.
  Let s_{k+1} = R_k^2 mod n.
EndFor.
Return (R_{w-1}).
```

The attack allows someone who knows exponent bits 0..(b-1) to find bit b. To obtain the entire exponent, start with b equal to 0 and repeat the attack until the entire exponent is known.

Because the first b exponent bits are known, the attacker can compute the first b iterations of the `For` loop to find the value of s_b. The next iteration requires the first unknown exponent bit. If this bit is set, $R_b = (s_b \cdot y) \bmod n$ will be computed. If it is zero, the operation will be skipped.

The attack will be described first in an extreme hypothetical case. Suppose the target system uses a modular multiplication function that is normally extremely fast but occasionally takes much more time than an entire normal modular exponentiation. For a few s_b and y values the calculation of $R_b = (s_b \cdot y) \bmod n$ will be extremely slow, and by using knowledge about the target system's design the attacker can determine which these are. If the total modular exponentiation time is ever fast when $R_b = (s_b \cdot y) \bmod n$ is slow, exponent bit b must be zero. Conversely, if slow $R_b = (s_b \cdot y) \bmod n$ operations always result in slow total modular exponentiation times, the exponent bit is probably set. Once exponent bit b is known, the attacker can verify that the overall operation time is slow whenever $s_{b+1} = R_b^2 \bmod n$ is expected to be slow. The same set of timing measurements can then be reused to find the following exponent bits.

3 Error Correction

If exponent bit b is guessed incorrectly, the values computed for $R_{k \geq b}$ will be incorrect and, so far as the attack is concerned, essentially random. The time

required for multiplies following the error will not be reflected in the overall exponentiation time. The attack thus has an *error-detection* property; after an incorrect exponent bit guess, no more meaningful correlations are observed.

The error detection property can be used for error correction. For example, the attacker can maintain a list of the most likely exponent intermediates along with a value corresponding to the probability each is correct. The attack is continued for only the most likely candidate. If the currently-favored value is incorrect, it will tend to fall in ranking, while correct values will tend to rise. Error correction techniques increase the memory and processing requirements for the attack, but can greatly reduce the number of samples required.

4 The General Attack

The attack can be treated as a signal detection problem. The "signal" consists of the timing variation due to the target exponent bit, and "noise" results from measurement inaccuracies and timing variations due to unknown exponent bits. The properties of the signal and noise determine the number of timing measurements required to for the attack.

Given j messages $y_0, y_1, ..., y_{j-1}$ with corresponding timing measurements $T_0, T_1, ..., T_{j-1}$, the probability that a guess x_b for the first b exponent bits is correct is proportional to

$$P(x_b) \propto \prod_{i=0}^{j-1} F(T_i - t(y_i, x_b))$$

where $t(y_i, x_b)$ is the amount of time required for the first b iterations of the $y_i^x \bmod n$ computation using exponent bits x_b, and F is the expected probability distribution function of $T - t(y, x_b)$ over all y values and correct x_b. Because F is defined as the probability distribution of $T_i - t(y_i, x_b)$ if x_b is correct, it is the best function for predicting $T_i - t(y_i, x_b)$. Note that the timing measurements and intermediate s values can be used improve the estimate of F.

Given a correct guess for x_{b-1}, there are two possible values for x_b. The probability that x_b is correct and x_b' is incorrect can be found as

$$\frac{\prod_{i=0}^{j-1} F(T_i - t(y_i, x_b))}{\prod_{i=0}^{j-1} F(T_i - t(y_i, x_b)) + \prod_{i=0}^{j-1} F(T_i - t(y_i, x_b'))}.$$

In practice, this formula is not very useful because finding F would require extraordinary effort.

5 Simplifying the Attack

Fortunately it is generally not necessary to compute F. Each timing observation consists of $T = e + \sum_{i=0}^{w-1} t_i$, where t_i is the time required for the multiplication and squaring steps for bit i and e includes measurement error, loop overhead,

etc. Given guess x_b, the attacker can find $\sum_{i=0}^{b-1} t_i$ for each sample y. If x_b is correct, subtracting from T yields $e + \sum_{i=0}^{w-1} t_i - \sum_{i=0}^{b-1} t_i = e + \sum_{i=b}^{w-1} t_i$. Since the modular multiplication times are effectively independent from each other and from the measurement error, the variance of $e + \sum_{i=b}^{w-1} t_i$ over all observed samples is expected to be $\mathrm{Var}(e) + (w-b)\mathrm{Var}(t)$. However if only the first $c < b$ bits of the exponent guess are correct, the expected variance will be $\mathrm{Var}(e) + (w-b+2c)\mathrm{Var}(t)$. Correctly-emulated iterations decrease the expected variance by $\mathrm{Var}(t)$, while iterations following an incorrect exponent bit each increase the variance by $\mathrm{Var}(t)$. Computing the variances is easy and provides a good way to identify correct exponent bit guesses.

It is now possible to estimate the number of samples required for the attack. Suppose an attacker has j accurate timing measurements and has two guesses for the first b bits of a w-bit exponent, one correct and the other incorrect with the first error at bit c. For each guess the timing measurements can be adjusted by $\sum_{i=0}^{b-1} t_i$. The correct guess will be identified successfully if its adjusted values have the smaller variance.

It is possible to approximate t_i using independent standard normal variables. If $\mathrm{Var}(e)$ is negligible, the expected probability of a correct guess is

$$P\left(\sum_{i=0}^{j-1}\left(\sqrt{w-b}X_i + \sqrt{2(b-c)}Y_i\right)^2 > \sum_{i=0}^{j-1}\left(\sqrt{w-b}X_i\right)^2\right)$$
$$= P\left(2\sqrt{2(b-c)(w-b)}\sum_{i=0}^{j-1} X_iY_i + 2(b-c)\sum_{i=0}^{j-1} Y_i^2 > 0\right)$$

where X and Y are normal random variables with $\mu = 0$ and $\sigma = 1$. Because j is relatively large, $\sum_{i=0}^{j-1} Y_i^2 \approx j$ and $\sum_{i=0}^{j-1} X_iY_i$ is approximately normal with $\mu = 0$ and $\sigma = \sqrt{j}$, yielding

$$P\left(2\sqrt{2(b-c)(w-b)}(\sqrt{j}Z) + 2(b-c)j > 0\right) = P\left(Z > -\frac{\sqrt{j(b-c)}}{\sqrt{2(w-b)}}\right)$$

where Z is a standard normal random variable. Finally, integrating to find the probability of a correct guess yields $\Phi\left(\sqrt{\frac{j(b-c)}{2(w-b)}}\right)$, where $\Phi(x)$ is the area under the standard normal curve from $-\infty$ to x. The required number of samples (j) is thus proportional to the exponent size (w). The number of measurements might be reduced if attackers choose inputs known to have extreme timing characteristics at exponent locations of interest.

6 Experimental Results

Figure 1 shows the distribution of 10^6 modular multiplication times observed using the RSAREF toolkit[10] on a 120-MHz Pentium^{TM} computer running MSDOS^{TM}. The distribution was prepared by timing one million ($a \cdot b \bmod n$) calculations using a and b values from actual modular exponentiation operations

with random inputs. The 512-bit sample prime #1 from the RSAREF Diffie-Hellman demonstration program was used for n. A few wildly aberrant samples (which took over 1300μs) were discarded. The Figure 1 distribution has mean $\mu = 1167.8\mu$s and standard deviation $\sigma = 12.01\mu$s. The measurement error is small; the tests were run twice and the average measurement difference was found to be under 1μs. RSAREF uses the same function for squaring and multiplication, so squaring and multiplication times have identical distributions.

RSAREF precomputes y^2 and y^3 mod n and processes two exponent bits at a time. In total, a 512-bit modular exponentiation with a random 256-bit exponent requires 128 iterations of the modular exponentiation loop and a total of about 352 modular multiplication and squaring operations. Each iteration of the modular exponentiation loop does two squaring operations and, if either exponent bit is nonzero, one multiply. The attack can be adjusted to append pairs of exponent bits and to evaluate four candidate values at each exponent position instead of two.

Since modular multiplications consume most of the total modular exponentiation time, it is expected that the distribution of modular exponentiation times will be approximately normal with $\mu \geq (1167.8)(352) = 411,065.6\mu s$ and $\sigma \geq 12.01\sqrt{352} = 225.3\mu s$. Figure 2 shows measurements from 5000 actual modular exponentiation operations using the same computer and modulus, which yielded $\mu = 419,901\mu s$ and $\sigma = 235\mu s$.

FIGURE 1: RSAREF Modular Multiplication Times

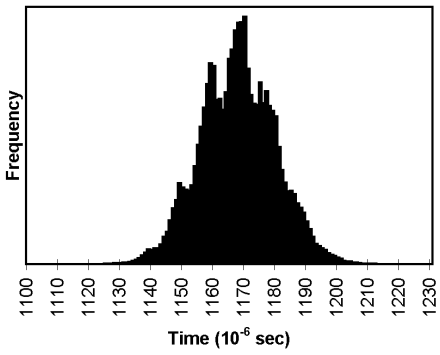
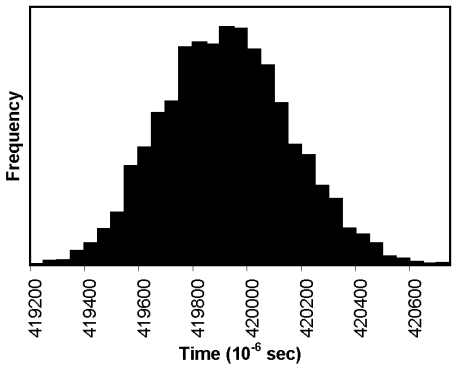


FIGURE 2: RSAREF Modular Exponentiation Times



With 250 timing measurements, the probability that subtracting the time for a correct modexp loop iteration from each sample will reduce the total variance more than subtracting an incorrect iteration is estimated to be $\Phi\left(\sqrt{\frac{j(b-c)}{2(w-b)}}\right)$, where $j = 250$, $b = 1$, $c = 0$, and $w = 127$. (There are 128 iterations of the RSAREF modexp loop for a 256-bit exponent, but the first iteration is ignored.) Correct guesses are thus expected with probability $\Phi\left(\sqrt{\frac{250(1-0)}{2(126)}}\right) \approx 0.84$. The

5000 samples from Figure 2 were divided into 20 groups of 250 samples each, and variances from subtracting the time for incorrect and correct modexp loop iterations were compared at each of the 127 exponent bit pairs. Of the 2450 trials, 2168 produced a larger variance after subtracting an incorrect modexp loop time than after subtracting the time for a correct modexp loop, yielding a probability of 0.885. The first exponent bits are most difficult, since b becomes larger as more exponent bits become known and the probabilities should improve. (The test above did not take advantage of this property.) It is important to note that accurate timing measurements were used; measurement errors which are large relative to the total modular exponentiation time standard deviation will increase the number of samples needed.

The attack is computationally quite easy. With RSAREF, the attacker has to evaluate four choices per pair of bits. Thus the attacker only has to do four times the number of operations done by the victim, not counting effort wasted by incorrect guesses.

7 Montgomery Multiplication and the CRT

Modular reduction steps usually cause most of the timing variation in a modular multiplication operation. Montgomery multiplication[6] eliminates the mod n reduction steps and, as a result, tends to reduce the size of the timing characteristics. However, some variation usually remains. If the remaining "signal" is not dwarfed by measurement errors, the variance in t_b and the variance of $\sum_{i=b+1}^{w-1} t_i$ would be reduced proportionally and the attack would still work. However if the measurement error e is large, the required number of samples will increase in proportion to $\frac{1}{\mathrm{Var}(t_i)}$.

The Chinese Remainder Theorem (CRT) is also often used to optimize RSA private key operations. With CRT, (y mod p) and (y mod q) are computed first, where y is the message. These initial modular reduction steps can be vulnerable to timing attacks. The simplest such attack is to choose values of y that are close to p or q, then use timing measurements to determine whether the guessed value is larger or smaller than the actual value of p (or q). If y is less than p, computing y mod p has no effect, while if y is larger than p, it is necessary to subtract p from y at least once. Also, if the message is very slightly larger than p, y mod p will have leading zero digits, which may reduce the amount of time required for the first multiplication step. The specific timing characteristics depend on the implementation. RSAREF's modular reduction function with a 512-bit modulus the Pentium computer with y chosen randomly between 0 and $2p$ takes an average of 42.1μs if $y < p$, as opposed to 73.9μs if $y > p$. Timing measurements from many y could be combined to successively approximate p.

In some cases it may be possible to improve the Chinese Remainder Theorem RSA attack to use known (not chosen) ciphertexts, reducing the number of messages required and making it possible to attack RSA digital signatures. Modular reduction is done by subtracting multiples of the modulus, and exploitable timing variations can be caused by variations in the number of compare-and-subtract

steps. For example, RSAREF's division loop integer-divides the uppermost two digits of y by one more than the upper digit of p, multiplies p by the quotient, shifts left the appropriate number of digits, then subtracts the result from y. If the result is larger than p (shifted left), a extra subtraction is performed. The decision whether to perform an extra subtraction step in the first loop of the division algorithm usually depends only on y (which is known) and the upper two digits of p. A timing attack could be used to determine the upper digits of p. For example, an exhaustive search over all possible values for the upper two digits of p (or more efficient techniques) could identify value for which the observed times correlate most closely with the expected number of subtraction operations. As with the Diffie-Hellman/non-CRT attack, once one digit of p has been found, the timing measurements could be reused to find subsequent digits.

It is not yet known whether timing attacks can be adapted to directly attack the mod p and mod q modular exponentiations performed with the Chinese Remainder Theorem.

8 Timing Cryptanalysis of DSS

The Digital Signature Standard[5] computes $s = (k^{-1}(H(m) + x \cdot r)) \bmod q$, where r and q are known to attackers, k^{-1} is usually precomputed, $H(m)$ is the hash of the message, and x is the private key. In practice, $(H(m) + x \cdot r) \bmod q$ would normally be computed first, then is multiplied by $k^{-1} \pmod q$.

If the modular reduction function runs in non-fixed time, the overall signature time should be correlated with the time for the $(x \cdot r \bmod q)$ computation. The attacker can calculate and compensate for the time required to compute $H(m)$. Since $H(m)$ is of approximately the same size as q, its addition has little effect on the reduction time. The most significant bits of $x \cdot r$ are typically the first used in the modular reduction. These depend on r, which is known, and the most significant bits of the secret value x. There would thus be a correlation between values of the upper bits of x and the total time for the modular reduction. By looking for the strongest probabilities over the samples, the attacker would try to identify the upper bits of x. As more upper bits of x become known, more of $x \cdot r$ becomes known, allowing the attacker to proceed through more iterations of the modular reduction loop to attack new bits of x. If k^{-1} is precomputated, DSS signatures require just two modular multiplication operations, potentially making the amount of additional timing noise which must be filtered out relatively small.

9 Masking Timing Characteristics

The most obvious way to prevent timing attacks is to make all operations take exactly the same amount of time. Unfortunately this is often difficult. Making software run in fixed time, especially in a platform-independent manner, is hard because compiler optimizations, RAM cache hits, instruction timings, and other

factors can introduce unexpected timing variations. If a timer is used to delay returning results until a pre-specified time, factors such as the system responsiveness or power consumption may still change detectably when the operation finishes. Some operating systems also reveal processes' CPU usage. Fixed time implementations are also likely to be slow; many performance optimizations cannot be used since all operations must take as long as the slowest operation. (Note: Always performing the optional $R_i = (s_i \cdot y) \bmod n$ step does not make an implementation run in constant time, since timing characteristics from the squaring operation and subsequent loop iterations can be exploited.)

Another approach is to make timing measurements so inaccurate that the attack becomes unfeasible. Random delays added to the processing time do increase the number of ciphertexts required, but attackers can compensate by collecting more measurements. The number of samples required increases roughly as the square of the timing noise. For example, if a modular exponentiator whose timing characteristics have a standard deviation of 10 ms can be broken successfully with 1000 timing measurements, adding a random normally distributed delay with 1 second standard deviation will make the attack require approximately $\left(\frac{1000ms}{10ms}\right)^2 (1000) = 10^7$ samples. (Note: The *mean* delay would have to be several seconds to get a standard deviation of 1 second.) While 10^7 samples is probably more than most attackers can gather, a security factor of 10^7 is not usually considered adequate.

10 Preventing the Attack

Fortunately there is a better solution. Techniques used for blinding signatures[1] can be adapted to prevent attackers from knowing the input to the modular exponentiation function. Before computing the modular exponentiation operation, choose a random pair (v_i, v_f) such that $v_f^{-1} = {v_i}^x \bmod n$. For Diffie-Hellman, it is simplest to choose a random v_i then compute $v_f = (v_i^{-1})^x \bmod n$. For RSA it is faster to choose a random v_f relatively prime to n then compute $v_i = (v_f^{-1})^e \bmod n$, where e is the public exponent. Before the modular exponentiation operation, the input message should be multiplied by $v_i \pmod n$, and afterward the result is corrected by multiplying with $v_f \pmod n$. The system should reject messages equal to $0 \pmod n$.

Computing inverses mod n is slow, so it is often not practical to generate a new random (v_i, v_f) pair for each new exponentiation. The $v_f = (v_i^{-1})^x \bmod n$ calculation itself might even be subject to timing attacks. However (v_i, v_f) pairs should not be reused, since they themselves might be compromised by timing attacks, leaving the secret exponent vulnerable. An efficient solution to this problem is update v_i and v_f before each modular exponentiation step by computing $v_i' = v_i^2$ and $v_f' = v_f^2$. The total performance cost is small (2 modular squarings, which can be precomputed, plus 2 modular multiplications). More sophisticated update operations using exponents other than 2, multiplication with other (v_i, v_f) pairs, etc. can also be used, but do not appear to offer any advantages.

If (v_i, v_f) is secret, attackers have no useful knowledge about the input to the modular exponentiator. Consequently the most an attacker can learn is the general timing distribution for exponentiation operations. In practice, distributions are close to normal and the 2^w exponents cannot possibly be distinguished. However, a maliciously-designed modular exponentiator could theoretically have a distribution with sharp spikes corresponding to exponent bits, so blinding does not *provably* prevent timing attacks.

Even with blinding, the distribution will reveal the average time per operation, which can be used to infer the Hamming weight of the exponent. If anonymity is important or if further masking is required, a random multiple of $\varphi(n)$ can be added to the exponent before each modular exponentiation. If this is done, care must be taken to ensure that the addition process itself does not have timing characteristics which reveal $\varphi(n)$. This technique may be helpful in preventing attacks that gain information leaked during the modular exponentiation operation due to electromagnetic radiation, system performance fluctuations, changes in power consumption, etc. since the exponent bits change with each operation.

11 Further Work

Timing attacks can potentially be used against other cryptosystems, including symmetric functions. For example, in software the 28-bit C and D values in the DES[4] key schedule are often rotated using a conditional which tests whether a one-bit must be wrapped around. The additional time required to move nonzero bits could slightly degrade the cipher's throughput or key setup time. The cipher's performance can thus reveal the Hamming weight of the key, which provides an average of $\sum_{n=0}^{56} \frac{\binom{56}{n}}{2^{56}} \log_2 \left(\frac{2^{56}}{\binom{56}{n}} \right) \approx 3.95$ bits of key information. IDEA[3] uses an $f()$ function with a modulo $(2^{16}+1)$ multiplication operation, which will usually run in non-constant time. RC5[7] is at risk on platforms where rotates run in non-constant time. RAM cache hits can produce timing characteristics in implementations of Blowfish[11], SEAL[9], DES, and other ciphers if tables in memory are not used identically in every encryption.

Additional research is needed to determine whether specific implementations are at risk and, if so, the degree of their vulnerability. So far, only a few specific systems have been studied in detail and the attacks against CRT/Montgomery RSA and DSS are currently theoretical.

Further refinements to the attack may also be possible. A direct attack against p and q in RSA with the Chinese Remainder Theorem would be particularly important.

12 Conclusions

In general, any channel which can carry information from a secure area to the outside should be studied as a potential risk. Implementation-specific timing

characteristics provide one such channel and can sometimes be used to compromise secret keys. Vulnerable algorithms, protocols, and systems need to be revised to incorporate measures to resist timing cryptanalysis and related attacks.

13 Acknowledgements

I am grateful to Matt Blaze, Joan Feigenbaum, Martin Hellman, Phil Karn, Ron Rivest, and Bruce Schneier for their encouragement, helpful comments, and suggestions for improving the manuscript.

References

1. D. Chaum, "Blind Signatures for Untraceable Payments," *Advances in Cryptology: Proceedings of Crypto 82,* Plenum Press, 1983, pp. 199-203.
2. W. Diffie and M.E. Hellman, "New Directions in Cryptography," *IEEE Transactions on Information Theory*, IT-22, n. 6, Nov 1976, pp. 644-654.
3. X. Lai, *On the Design and Security of Block Ciphers,* ETH Series in Information Processing, v. 1, Konstanz: Hartung-Gorre Verlag, 1992.
4. National Bureau of Standards, "Data Encryption Standard," Federal Information Processing Standards Publication 46, January 1977.
5. National Institute of Standards and Technology, "Digital Signature Standard," Federal Information Processing Standards Publication 186, May 1994.
6. P.L. Montgomery, "Modular Multiplication without Trial Division," *Mathematics of Computation,* v. 44, n. 170, 1985, pp. 519-521.
7. R.L. Rivest, "The RC5 Encryption Algorithm," *Fast Software Encryption: Second International Workshop, Leuven, Belgium, December 1994, Proceedings*, Springer-Verlag, 1994, pp. 86-96.
8. R.L. Rivest, A. Shamir, and L.M. Adleman, "A method for obtaining digital signatures and public-key cryptosystems," *Communications of the ACM*, **21**, 1978, pp. 120-126.
9. P.R. Rogaway and D. Coppersmith, "A Software-Optimized Encryption Algorithm," *Fast Software Encryption: Cambridge Security Workshop, Cambridge, U.K., December 1993, Proceedings*, Springer-Verlag, 1993, pp. 56-63.
10. RSA Laboratories, "RSAREF: A Cryptographic Toolkit," Version 2.0, 1994, available via FTP from `rsa.com`.
11. B. Schneier, "Description of a New Variable-Length Key, 64-bit Block Cipher (Blowfish)," *Fast Software Encryption: Second International Workshop, Leuven, Belgium, December 1994, Proceedings*, Springer-Verlag, 1994, pp. 191-204.