Časovni in drugi napadi s stranskim kanalom

Matija Polajnar

Podljubelj, 12. avgust 2009

Kazalo

1 Uvod

Ste na službenem potovanju in sodelavcu morate sporočiti pomemben zaupen podatek. Vesta, da telefonu lahko prisluškuje konkurenca, zato sta se vnaprej dogovorila za ključ Vernamove šifre (t.i. *one-time pad*). Med telefonskim pogovorom črko za črko kriptirate sporočilo tako, da v mislih *seštevate* črke in kriptogram izgovarjate kolegu. Kmalu izveste, da je konkurenca prestregla podatek. Kako? Saj sta vendar ključ uporabila samo enkrat, generiran je bil naključno, Vernamova šifra pa ima lastnost popolne tajnosti. Kje sta se uštela? Odgovor bo zainteresiran bralec našel med branjem sledečih poglavij, sicer pa je pojasnjen v zaključku.

Vgrajeni (angl. embedded) računalniki so vedno pogostejši v raznih napravah, njihova informacijska varnost pa lahko močno vpliva na fizično in gmotno varnost posameznikov. Kriptoanaliza s pomočjo informacij kot so

pari kriptogramov in golih besedil, več kriptogramov istega besedila, javni del ključa ipd. je študentom računalništva, morebitnim bodočim snovalcem tovrstnih sistemov, običajno dobro znana, manj pa vedo o prednostih, ki jih (morda zlonamernim) kriptoanalitikom prinašajo dodatne fizikalne informacije, ki jih pridobimo z metodami kot so [1]:

- meritve časov (de)kriptiranja (*časovni napad*),
- meritve električne porabe vezja,
- meritve prepuščenega sevanja iz komponent vezja,
- meritve oddanega zvoka (*akustična kriptoanaliza*),
- diferencialna analiza napak (tj. dobivanje informacij o npr. ključu s pomočjo različnih računskih napak, ki so bile narejene med ponovitvami iste kriptografske operacije),
- opazovanje odsevov (v steklih, žlicah, očeh, ...) s teleskopi itd.

V projektu skušam predstaviti nekatere metode napadov s stranskim kanalom in načine zaščite pred njimi.

Potrebno je poudariti, da je za napade s stranskim kanalom včasih potrebno imeti fizičen dostop do naprave ali jo celo razstaviti ali delno uničiti. Napadi so odvisni od implementacije (programske in strojne) in ne le od kriptografske sheme, ki jo neka oprema implementira. Niso pa vsi *neklasični* napadi napadi s stranskim kanalom: mučenje osebe, ki pozna iskano informacijo (npr. dešifrirni ključ), na primer **ni** napad s stranskim kanalom.

Namen projekta ni izčrpna predstavitev napadov s stranskim kanalom, temveč osveščanje o tovrstnih napadih s predstavitvijo nekaterih zanimivih primerov ter poskus ugotovitve trenutnega stanja zaščite pred tovrstnimi napadi.

2 Načini napadov s stranskim kanalom

Napadi s stranskim kanalom so osredotočeni na konkretno (znano ali neznano) implementacijo kriptografske sheme, ne le na shemo samo. [2] Delimo jih glede na fizikalno količino, s katero si pomagamo (čas, električna energija, elektromagnetno sevanje, ...). V tem projektu je poseben poudarek na časovnih napadih, zato sledi daljše podpoglavje o le-teh; kasneje so v tem poglavju opisani še napadi, ki temeljijo na drugih količinah.

2.1 Časovni napadi

2.1.1 Uvod

Časovni napadi (*timing attacks*) so osnovani na merjenju časa, ki ga naprava porabi za določene kriptografske operacije. Na podlagi tega je včasih možno dobiti nekaj informacij o ključu. Razlogi za različna trajanja operacij so različni:

- hitrostne optimizacije z izpuščanjem odvečnih računskih operacij,
- vejitve in pogojni stavki,
- lokalnost podatkov (zadetki oziroma zgrešitve v predpomnilniku),
- različna trajanja operacij (npr. množenja) glede na vrednosti operandov itd.

Običajno gre pri času izvajanja za odvisnost tako od ključa kot od besedila, ki ga šifriramo ali dešifriramo. Pri določenih algoritmih tako lahko z opazovanjem trajanja operacij nad različnimi besedili in statistično obravnavo izmerjenih vrednosti ,,uganemo" bite ključa.

Zanimiv (preprost) primer ranljivosti na časovni napad so nekoč predstavljali Unix terminalski strežniki, ki so zgostitveno funkcijo nad geslom (za primerjanje z zgoščeno vrednostjo, zapisano v bazi gesel) izračunali le, če je vpisano uporabniško ime dejansko obstajalo v sistemu. Na tedanjih računalnikih je izračun zgostitvene funkcije trajal zadosti dolgo, da je bilo na ta način mogoče ugotoviti, če neko uporabniško ime obstaja. Tako si je napadalec s slovarskim ugibanjem lahko nabral seznam obstoječih uporabniških imen na neznanem sistemu in s tem močno zožil iskalni prostor za poskus vdora z golo silo (torej preizkušanjem vseh kombinacij slovarskih imen in gesel).

2.1.2 Primer napada

O časovnem napadu na RSA (in nekatere druge kriptosisteme) je že leta 1996 pisal Kocher [3]. V članku omenja tudi, da je pri uporabi Montgomeryjevega algoritma za modularno množenje časovni napad precej otežen (potrebuje mnogo natančnejše meritve časa), saj tak algoritem odpravi potrebo po časovno potratnih redukcijah po modulu. V nadaljevanju je opisan napad na ravno tak sistem (RSA z Montgomeryjevim modularnim množenjem) [4]. V poglavju 3 je tudi poročilo moje izvedbe simulacje tega napada.

Splošen formalni sistem Opazujemo kriptografsko napravo, ki podpisuje besedila. Če so M, K in S zaporedoma množice besedil, ključev in podpisanih besedil, je $A : M \times K \to S$ algoritem, ki računa podpis.

Naj bo $T : M \times K \to \mathbb{R}$ čas za izračun $A(m, k)$. Kot napadalci želimo $k \in K$ najti (ga ne poznamo), zato opazujemo kriptografske operacije s tem (vedno istim) ključem nad različnimi besedili $m \in M$; posledično bomo poenostavljeno pisali $T(m)$ namesto $T(m, k)$.

Predpostavili bomo, da poznamo *preroka* $O : M \to \{1, 2\}$, s pomočjo katerega bomo lahko zgradili dve disjunktni množici besedil $M_1, M_2 \subseteq M$. Označimo $\forall m \in M_1 : F_1(m) = T(m)$ in $\forall m \in M_2 : F_2(m) = T(m)$.

Recimo, da ko ugotavljamo k_i, tj. i-ti bit ključa, velja $F_j = v_j^{k_i}$, kjer so $v_1^0, v_2^0, v_1^1, v_2^1$ verjetnostne spremenljivke. Predpostavimo, da za nek parameter ϕ verjetnostnih spremenljivk (npr. za srednjo vrednost) velja $\phi(v_1^0) = \phi(v_2^0)$ in $\phi(v_1^1) > \phi(v_2^1)$ (torej je pri $k_i = 1$ vrednost $\phi(T)$ pri besedilih iz M_1 značilno večja kot pri besedilih M_2, pri $k_i = 0$ pa te razlike ni).

Če nam potem statistični test s stopnjo zaupanja $1 - \alpha$ potrdi hipotezo $\phi(F_1) = \phi(F_2)$ napram hipotezi $\phi(F_1) > \phi(F_2)$, velja $k_i = 0$ z verjetnostjo $1-\alpha$. Če torej s pomočjo *preroka* zgradimo dve podmnožici besedil in merimo čase podpisovanja, bomo z uporabo statističnih testov lahko dobili en bit ključa.

Konkreten primer Želimo napasti RSA podpis, torej izračun $y = m^k \bmod n$, pri čemer vemo, da se potenciranje izračuna tako:

- uporablja se algoritem *kvadriraj-in-zmnoži* (*square-and-multiply*; algoritem 1),
- za množenje (5. vrstica algoritma) in kvadriranje (2. vrstica algoritma) se uporablja Montgomeryjev algoritem za modularno množenje [5] (glej tudi 2. poglavje [6]).

Glede Montgomeryjevega algoritma je dovolj vedeti, da števila transformira v tako reprezentacijo, da množenje traja konstanten čas (ne glede na vhod), **vendar** lahko po koncu množenja dobimo za modul n prevelik rezultat, torej je treba rezultatu odšteti n. Temu rečemo *redukcija*, česar ne smemo pomotoma enačiti z *redukcijo poljubno velikega celega števila po modulu*; ta pri uporabi Montgomeryjevega množenja ni potrebna.

Najprej si poglejmo, katere komponente sestavljajo celoten čas izvajanja algoritma 1. Označimo:

- T_M: konstanten čas Montgomeryjevega množenja po modulu n brez morebitne končne redukcije,

- T_R: konstanten čas redukcije na koncu Montgomeryjevega množenja,
- $\delta_i(k, m)$: indikator redukcije za i-to Montgomeryjevo množenje:
  - 0. množenje je kvadriranje v prvi iteraciji algoritma,
  - 1. množenje je množenje v prvi iteraciji algoritma (izvede se le, če je $k_0 = 1$),
  - 2. množenje je kvadriranje v drugi iteraciji algoritma itd.;

  indikator ima vrednost 1, če je potrebno opraviti redukcijo, sicer 0; odvisen je od ključa in sporočila.

Čas T izvajanja algoritma torej razpade na vsoto naslednjih komponent:

kvadriranje (vrstica 2)	$\sum_{i=0}^{b-1} (T_M + \delta_{2\cdot i}(k, m) \cdot T_R)$
množenje (vrstica 5)	$\sum_{i=0}^{b-1} k_i \cdot (T_M + \delta_{2\cdot i+1}(k, m) \cdot T_R)$
ostale vrstice algoritma	Δ

$$T = \sum_{i=0}^{b-1} (T_M + \delta_{2\cdot i}(k, m) \cdot T_R) \ + \ \sum_{i=0}^{b-1} k_i \cdot (T_M + \delta_{2\cdot i+1}(k, m) \cdot T_R) \ + \ \Delta$$

Opazimo, da so pri fiksnem ključu k in modulu n edine od m odvisne komponente tiste, ki pripadajo redukcijam, torej:

$$\sum_{i=0}^{b-1} (\delta_{2\cdot i}(k, m) \cdot T_R \ + \ k_i \cdot \delta_{2\cdot i+1}(k, m) \cdot T_R)$$

To dejstvo bomo uporabili za napad.

Algoritem 1 Kvadriraj in zmnoži.

Vhod: celoštevilski modul m, celoštevilsko sporočilo $m < n$, celoštevilski b-bitni ključ $k = (k_0 k_1 \dots k_{b-1})_2$

Rezultat: $x = m^k \mod n$

1: $x \leftarrow 1$
2: **za** i naraščajoč od 0 do $b - 1$ **naredi**
3: $\quad x \leftarrow x^2 \mod n$
4: $\quad$ **če** $k_i = 1$ **potem**
5: $\quad\quad x \leftarrow x \cdot m \mod n$
6: **vrni** x

Ideja 1: Napad na množenje. Pri tem napadu se osredotočamo na druge člene v zgornji vsoti, tj. tiste, pri katerih nastopajo δ z lihimi indeksi in torej pripadajo množenjem (vrstici 5) v algoritmu 1.

Predpostavimo, da že poznamo prvih $j-1 > 0$ bitov ključa k (če ne gre drugače, lahko prvi bit ključa ugibamo). Naključno si izberemo neko množico besedil $M' \subset M$ in uporabimo tak postopek (*prerok*) za razbitje množice na $M_1 \cup M_2 = M'$:

- izvajamo algoritem *kvadriraj-in-zmnoži* do zadnjega znanega bita ključa,
- pri predpostavki, da je naslednji bit ključa enak 1 ($k_j = 1$), torej se množenje v 5. vrstici algoritma izvede, opazujemo, če je po Montgomeryjevem množenju potrebno opraviti končno odštevanje (redukcijo).

Tako dobimo podmnožici besedil:

- $M_1 = \{m \in M' : \text{pri množenju zaradi } k_j = 1 \text{ opravimo redukcijo}\}$
- $M_2 = \{m \in M' : \text{pri množenju zaradi } k_j = 1 \text{ ne opravimo redukcije}\}$

Če poznane bite ključa k in predpostavko, da je j-ti bit enak 1, označimo $\kappa^{j=1}$, lahko to zapišemo bolj kompaktno:

- $M_1 = \{m \in M' : \delta_{2j+1}(\kappa^{j=1}, m) = 1\}$
- $M_2 = \{m \in M' : \delta_{2j+1}(\kappa^{j=1}, m) = 0\}$

Izmerimo čase, ki jih naprava potrebuje za podpis posameznih sporočil, in s statističnim testom poskušamo sprejeti hipotezo $\phi(F_1) = \phi(F_2)$ napram $\phi(F_1) > \phi(F_2)$. Če je hipoteza sprejeta, je $k_j = 0$, saj očitno množenja sploh nismo opravljali; sicer je $k_j = 1$.

Avtorji [4] so na ta način s 50.000 izmerjenimi časi uspeli razbiti 128-bitni RSA ključ, vendar naštevajo tudi pomanjkljivosti postopka.

- Pri potenciranju besedila m je to besedilo pri opazovanih operacijah množenja ($x \cdot m \mod n$) vedno eden izmed faktorjev. Izkazalo se je, da obstaja velika korelacija med temi operacijami (bodisi se redukcija za dano besedilo zgodi pri mnogih, bodisi pri zelo redkih množenjih). Posledično je težko izolirati časovno razliko, ki pripada redukciji pri obravnavi iskanega bita ključa, od časovnih razlik zaradi ostalih redukcij.

- Statistični testi zelo težko ugotovijo, če je neka razlika res statistično značilna, sploh v luči prej omenjene korelacije, zaradi katere dokaj izrazita razlika v časih podpisovanja besedil iz ene in druge podmnožice *vedno* obstaja (tudi ko $k_j = 0$), vprašanje je potem le, če je dovolj velika.

Oba problema sta rešena z naslednjo idejo.

Ideja 2: Napad na kvadriranje. Nekoliko manj očitna (a sorodna) je ideja napada na fazo kvadriranja. Tokrat prav tako izvajamo algoritem *kvadriraj-in-zmnoži* do zadnjega znanega bita ključa. Za naslednji (j-ti) bit predpostavimo vsako izmed možnih vrednosti in za vsako opazujemo, če bo potrebna operacija redukcije pri kvadriranju, ki se brezpogojno zgodi na začetku ***naslednje*** iteracije zanke. Tako vsako besedilo iz M' razporedimo v eno izmed množic M_1, M_2 (pri predpostavki $k_j = 1$) in eno izmed množic M_3, M_4 (pri predpostavki $k_j = 0$):

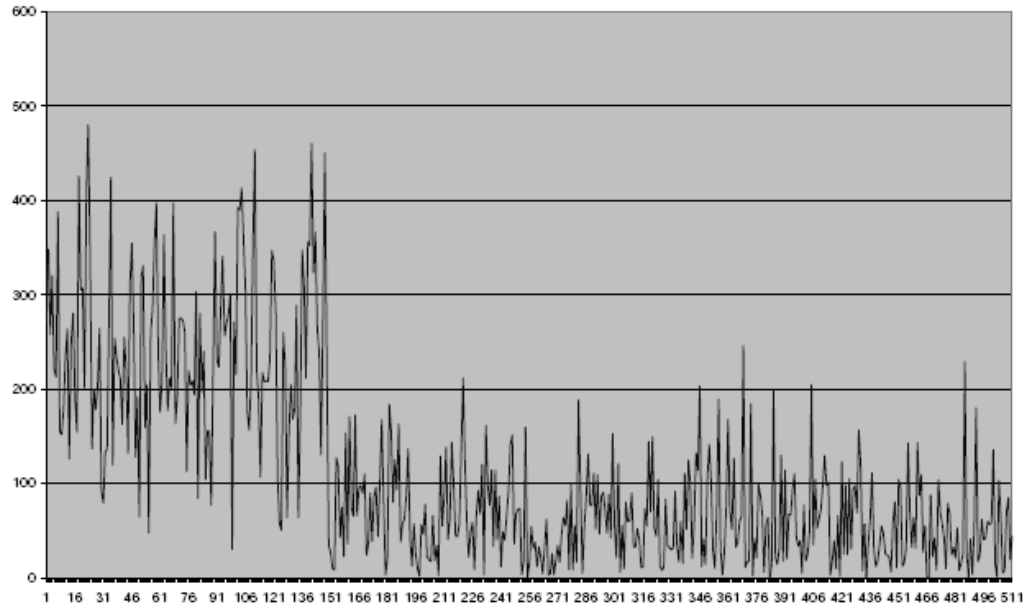
- $M_1 = \{m \in M' : \delta_{2j+2}(\kappa^{j=1}, m) = 1\}$
- $M_2 = \{m \in M' : \delta_{2j+2}(\kappa^{j=1}, m) = 0\}$
- $M_3 = \{m \in M' : \delta_{2j+2}(\kappa^{j=0}, m) = 1\}$
- $M_4 = \{m \in M' : \delta_{2j+2}(\kappa^{j=0}, m) = 0\}$

Če je bila predpostavka $k_j = 1$ pravilna, bo statistično značilno $\phi(F_1) > \phi(F_2)$, v nasprotnem primeru pa $\phi(F_3) > \phi(F_4)$. Opazimo dve pomembni dejstvi.

- Noben faktor v opazovanem Montgomeryjevem množenju ni fiksen (v vsaki iteraciji zanke ima običajno x drugačno vrednost, zato bo izračun $x^2 = x \cdot x$ vsakič *unikaten*). S tem smo odpravili korelacijo med dogodki, s katero smo imeli težave pri prejšnji ideji.

- Ni se nam več potrebno spraševati o *zadosti veliki* statistični značilnosti razlike, torej primerjati dogodkov $\phi(F_1) > \phi(F_2)$ in $\phi(F_1) = \phi(F_2)$, temveč preprosto preverimo, katera razlika med $\phi(F_1)$ in $\phi(F_2)$ ali med $\phi(F_3)$ in $\phi(F_4)$) je *bolj* statistično značilna ali preprosto večja.

Avtorji napada so uspeli 128-bitni ključ RSA razbiti z 20.000 besedili (merjenji časa). (V resnici nam te številke povejo bolj malo, saj so močno odvisne od natančnosti meritev, torej tudi hitrosti kriptografske naprave in razmerja med trajanjem redukcije T_R ter preostalih delov Montgomeryjevega množenja T_M.)



Slika 1: Absolutna vrednost razlike med razliko povprečnega časa za podpis besedil iz M_1 in M_2 ter razliko povprečnega časa za podpis besedil iz M_3 in M_4 (v številu potrebnih procesorskih ciklov). Ključ je 512-bitni, uporabljenih je bilo 350.000 besedil. Očitno smo bit 148 uganili narobe, ker se magnituda razlike od tam naprej nenadoma zmanjša (naš *prerok* je delal neosnovane odločitve). Vir: [4].

Sprotno zaznavanje in odprava napak Še ena zanimiva lastnost opisanega postopka je možnost sprotnega zaznavanja napak. Pri konstrukciji *preroka* smo namreč uporabili predpostavko, da pravilno poznamo prvih $j-1$ bitov ključa. Če ta predpostavka ne drži (torej če smo v postopku razbijanja ključa bit-po-bit pri nekem bitu naredili napako), bo delitev besedil v podmnožice naključna, zato bosta obe razliki časov (med M_1 in M_2 ter med M_3 in M_4) podobni. Primer je prikazan na grafu na sliki 1: bit 148 je bil določen narobe, zato pri nadaljnjem izvajanju algoritma ni več tako izrazitih razlik v časih podpisovanja.

Ko opazimo ta pojav, se lahko vrnemo na zadnji bit in njegovo vrednost negiramo; če se situacija ne izboljša, poskusimo enako z bitom pred njim itd. Na ta način so avtorji napada uspeli 128-bitni RSA ključ razbiti z manj kot 10.000 meritvami časov podpisovanj.

2.1.3 Protiukrepi

Konstanten čas izvajanja Najbolj očiten protiukrep proti napadom z merjenjem časa izvajanja je, da poskrbimo, da se algoritem vedno izvaja enako dolgo. To seveda pomeni izgubo časa, saj bomo v mnogih primerih morali po izračunu rezultata *čakati*, preden ga vrnemo.

Napadu iz prejšnjega primera bi uničili glavni vir informacij, če bi Montgomeryjev algoritem implementirali tako, da se končna redukcija (odštevanje) vedno izračuna, vendar se rezultat zavrže, če redukcija ni potrebna. Če bi poleg tega še algoritem *kvadriraj-in-zmnoži* popravili tako, da bi vedno opravil množenje v vrstici 5, vendar bi rezultat množenja zavrgel, če je opazovani bit ključa enak 0 (takemu algoritmu pravimo kvadriraj in vedno zmnoži – *square and* ***always*** *multiply*), bi se celotno potenciranje opravilo v konstantnem času.

Diskretizacija časov izvajanja (*bucketing*) Namesto enega samega časa izvajanja (kar je zelo razsipen način zakrivanja informacije o času računanja), avtorji [7] predlagajo določitev več različnih možnih trajanj računanja $t_1 < t_2 < \cdots < t_o$. Ko izračunamo rezultat, z vračanjem počakamo do trenutka, ko dosežemo prvo izmed teh trajanj (npr., če rezultat računamo $t_2 + \epsilon < t_3$ časa, ga vrnemo po poteku t_3 časa od začetka računanja).

Formalizem so avtorji še posplošili na katerokoli merljivo količino (ne le čas), tako da je O množica možnih izidov opazovanja računanja in je $o = |O|$. S pomočjo tega so matematično definirali napad s stranskim kanalom in z uporabo teorije informacij pokazali, da z n opazovanji napadalec dobi največ $|O| \cdot \log_2(n+1)$ bitov informacije o ključu.

Avtorji podajajo tudi postopke za iskanje optimalne diskretizacije časov pri danem številu o za izbran kriterij (bodisi minimalni časovni pribitek bodisi minimalno izdajanje informacije).

Naključen čas izvajanja Po izračunu rezultata lahko pred vračanjem letega počakamo naključen čas in s tem napadalčevo meritev zašumimo. To napadalcu močno poveča število potrebnih merjenj, da dobi iskano informacijo.

Naključno maskiranje (*blinding*) Včasih so kriptografske sheme take, da je možno čas izvajanja naključno spremeniti s spreminjanjem operandov na tak način, da končni rezultat ostaja enak. RSA je lep primer take sheme:

- pred potenciranjem si izberemo naključno število $r, 1 \leq r < n$,
- besedilo m množimo s tem številom: $m' = m \cdot r \mod n$,

- kriptografsko operacijo izvedemo (s poljubnim algoritmom) nad spremenjenim besedilom: $y' = (m')^k \mod n$,
- z REA izračunamo $r^{-k} \pmod n$ in to primnožimo rezultatu: $y \equiv y' \cdot r^{-k} \equiv (m')^k \cdot r^{-k} \equiv m^k \cdot r^k \cdot r^{-k} \equiv m^k \pmod n$

Napadalec ne more vedeti, katera števila so vključena v potenciranje (še več: če večkrat opazuje potenciranje z istimi operandi, bo zaradi naključnosti vedno izmeril drugačne čase), zato algoritmov za to operacijo ne more časovno napadati.

2.2 Napadi z merjenjem porabe energije

2.2.1 Uvod

Tovrstni napadi (*power (consumption) attacks*) so osnovani na merjenju porabe električne energije kriptografske naprave med izvajanjem kriptografske operacije. Ker so integrirana vezja sestavljena iz logičnih vrat, tok skozi napravo glede na njeno stanje teče po različnih poteh, od koder izvirajo razlike v električni porabi. Fizičnega dostopa do tuje kriptografske naprave ni vedno težko dobiti; primer so pametne bančne kartice, ki jih lahko napadamo s prirejenimi terminali.

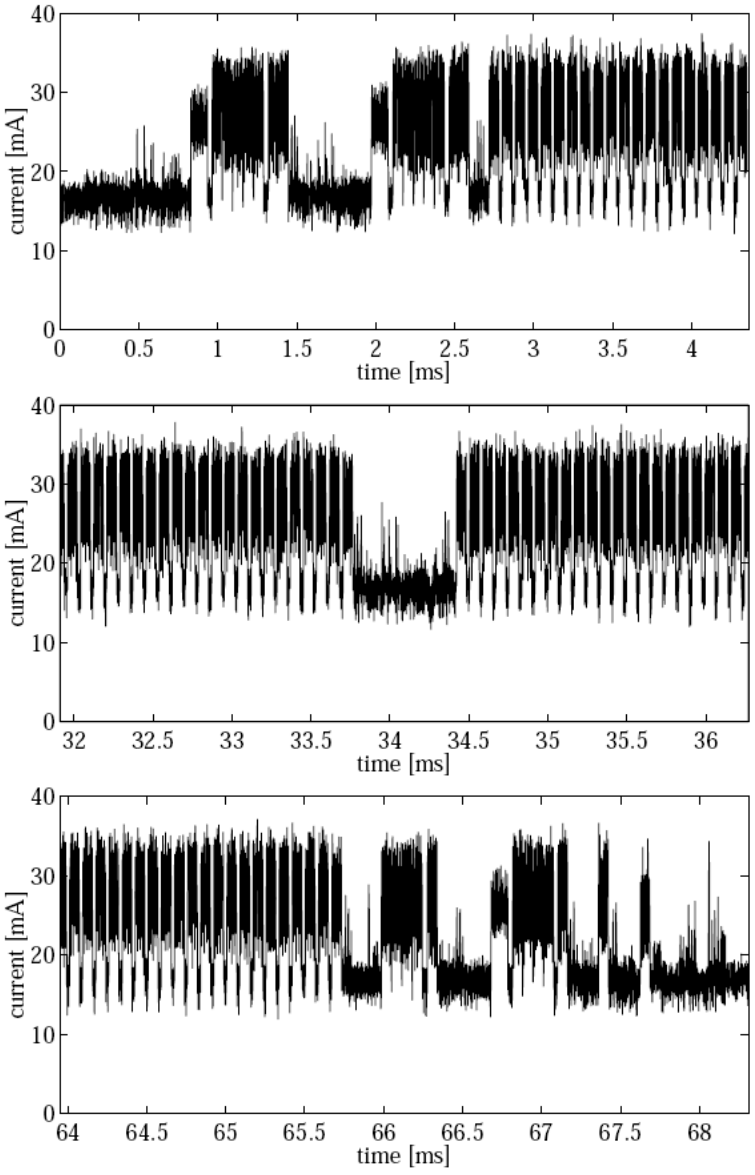
Laboratorijski merilniki električnega toka lahko zajemajo vzorce s frekvencami preko 1 GHz, nekoliko slabši pa so finančno dosegljivi tudi fizičnim osebam.

V splošnem ločimo dva načina napada:

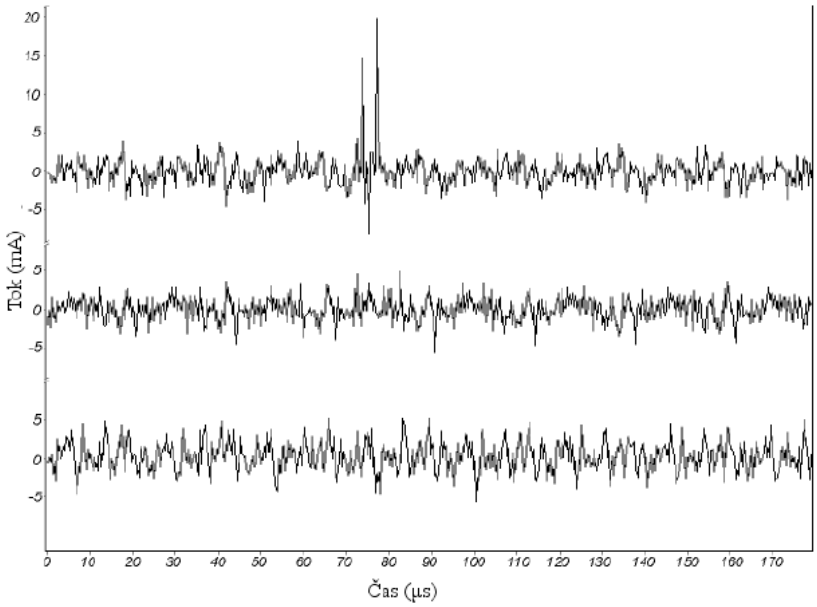
- enostavna analiza porabe (*simple power analysis, SPA*),
- diferencialna analiza porabe (*differential power analysis, DPA*).

Enostavna analiza porabe Napadalec si ogleda graf nihanja električnega toka preko naprave med izvajanjem kriptografske operacije in skuša najti značilnosti, ki bi mu pomagale razbiti ključ. Ta način je uporaben pri algoritmih, kjer je pot izvajanja močno odvisna od vhoda in je posamezne dele algoritma na grafu lahko prepoznati.

Na sliki 2 je primer grafa nihanja električnega toka na pametni kartici med izvajanjem RSA dekripcije [8]. Merjenje so avtorji opravili z nizkocenovno opremo. Dva daljša periodična vzorca sta dve potenciranji, kar nakazuje na uporabo kitajskega izreka o ostankih. Vsak vzorec vsebuje toliko periodičnih nihanj, kolikor je bilo opravljenih kvadriranj in množenj (torej za 2 manj od števila bitov ključa plus števila enic v ključu; spet glej algoritem 1), ni pa možno razločiti množenj od kvadriranj.



Slika 2: Nekaj izsekov grafa nihanja električnega toka med izvajanjem potenciranja za RSA podpisovanje/dekripcijo. Vir: [8].



Slika 3: Primer diferencialnih sledi nihanja električnega toka. Prva sled prikazuje diferencialno sled s pravilnim prerokom, spodnji dve pa z nepravilnim. Vir: [9].

Avtorji [8] so nato ugotovili, da gre za Garnerjev algoritem potenciranja s kitajskim izrekom o ostankih, ki mora vmesnemu rezultatu, če je ta negativen, prišteti modul p. To prištevanje so uspeli tudi identificirati na grafu nihanja električnega toka, matematično pa so odkrili lastnosti besedil, pri katerih je to prištevanje potrebno. Posledično so lahko z $\frac{1}{2}\log_2 n$ dekripcijami z izbranim kriptogramom odkrili število p in s tem celoten zasebni ključ.

Diferencialna analiza porabe Pri diferencialnih analizah imamo opravka z večjim številom meritev, preroki in statistično analizo [9] – primer smo že videli pri časovnih meritvah. Večje število potrebnih meritev je seveda slabost, vendar so posledično metode mnogo odpornejše na šum in pogosto delujejo tudi, ko ne poznamo podrobnosti implementacije.

Splošna ideja je, da imamo na voljo množico meritev električnega toka skozi čas pri kriptografski operaciji nad različnimi besedili. Nato izberemo preroka, ki meritve (oziroma besedila) razdeli na dve podmnožici, in izračunamo srednjo vrednost električnega toka za vsako podmnožico. Razlika med tema dvema bo praktično nična, če je bil prerok napačen, sicer pa bo razlika dosegala izrazite maksimume v točkah, kjer se izvajajo operacije nad bitom, po katerem prerok ločuje besedila (glej sliko 3).

V [9] je opisan primer napada z diferencialno analizo porabe na šifro DES.

2.2.2 Protiukrepi

Programske prilagoditve Redke izključno programske prilagoditve pomagajo pri merjenju električne porabe. Lahko omejimo število kriptografskih operacij, ki jih bo naprava izvedla preden se bo izključila oziroma uničila (npr. milijon transakcij z bančno kartico zadostuje vsakomur), in s tem omejimo število meritev, ki jih bo napadalec lahko opravil. Naključne zakasnitve v vsaki točki algoritma (oziroma *šumen* urin generator procesorja) otežijo iskanje korespondenc med različnimi meritvami, vendar tudi podaljšajo čas izvajanja. Iskanje korespondenc lahko popolnoma onemogočimo z naključnim vrstnim redom izvajanja ukazov algoritma, vendar zaradi sekvenčne narave postopkov to seveda ni možno; z zamenjavo le nekaterih ukazov (ki jih je dejansko možno, brez da bi to vplivalo na rezultat) pa iskanje korespondenc spet le otežimo.

Matematično dokazano učinkovita programska zaščita je naključno maskiranje vrednosti v pomnilniku, vkolikor je to za dano kriptografsko shemo mogoče. Konkretne zahteve za učinkovitost te metode pri DES (kjer je posledično med drugim potrebno spreminjati tudi S-škatle – *S-boxes*) so podane v [9].

Podobno kot proti napadom z merjenjem časa tudi proti enostavni analizi porabe deluje preprečevanje vejitev, kot je npr. uporabljeno v algoritmu *kvadriraj in vedno zmnoži*. Avtorji [10] predlagajo še več izboljšav tega algoritma, ki delujejo podobno hitro kot nezaščiten algoritem *kvadriraj in zmnoži*, vendar so odporni proti SPA napadom. (DPA napade uspešno preprečuje programsko naključno maskiranje vhodnih operandov.)

Strojne prilagoditve Ker je izvor ranljivosti, ki jih izkoriščajo ti napadi, ravno strojna oprema, ki z nihanjem električne porabe napadalcu izdaja informacije, so strojne prilagoditve bolj eleganten protiukrep. Obstaja več načinov:

- alternativne komponente vezij (namesto CMOS), npr. SABL ali WDDL, ki imajo na račun višje porabe in počasnejšega delovanja skoraj konstantno porabo,

- asinhronska vezja so varčnejša od sinhronskih, hkrati pa preko porabe izdajo manj informacij (žal več informacij izdajo preko oddanih sevanj, ker v teh sistemih ni urinega generatorja, ki s stališča sevanja povzroča veliko šuma),

- filtri za glajenje nihanj porabe,

- ločitev napajanja od kriptografskega vezja (z dvema kondenzatorjema; izmenično enega polnimo in drugega praznimo, tako da napadalec dobi zelo *zglajeno* krivuljo porabe) in onemogočanje komunikacije med izvajanjem kriptografskih postopkov, da iz napetosti na V/I vratih ni mogoče razbrati stanja kondenzatorjev,

- dodajanje šuma porabi (vgrajevanje dodatnega vezja z namenom ustvarjati naključno porabo),

- naključno maskiranje vrednosti v pomnilniku na hardverskem nivoju (to je splošnejši pristop od programskega in zato razbremeni snovalca algoritma).

2.3 Napadi na podlagi elektromagnetnih signalov

2.3.1 Uvod

Zgodnji napadi na sisteme z uporabo zaznavanja nenamerno prepuščenih sevanj segajo v 1. svetovno vojno, kjer so za vojaške telefone zaradi prihranka na teži napeljevali le en vodnik, za drugega pa uporabili zemljo. Kmalu so ugotovili, da na ta način lahko prisluškujejo nasprotniku, ki uporablja enako tehnologijo; začel se je razvoj protiukrepov.

Danes je povsem jasno, da je vsem obstoječim komunikacijskim kanalom možno (lažje ali težje) prisluškovati, zato zaupno vsebino kriptiramo. Bolj zahrbtni so napadi na podlagi merjenja sevanja, ki ga odda ena sama kriptografska naprava med izvajanjem kriptografske operacije.

Napadi z merjenjem sevanja so sorodni napadom z merjenjem porabe električne energije, a pogosto še bolj informativni, saj lahko (brez razstavljanja naprave) opazujemo tudi lokacijo in frekvenco izsevanja.

2.3.2 Protiukrepi

Nekateri protiukrepi, omenjeni pri prejšnjih dveh vrstah napadov, delujejo tudi proti napadom z merjenjem sevanja (npr. naključno maskiranje podatkov, neenakomeren urin generator in naključen vrstni red izvajanja algoritma).

Bolj specifični protiukrepi so npr. oklepi za preprečevanje prepuščanja sevanja (Faradayeva kletka) in namerno oddajanje šuma.

2.4 Napadi z izkoriščanjem računskih napak

2.4.1 Uvod

Večina napadov z izkoriščanjem računskih napak (*fault attacks*) spada v kategorijo diferencialne analize napak (*Differential Fault Analysis, DFA*). Ti napadi za odkrivanje bitov ključa uporabijo napačne rezultate, ki jih vrne kriptografska naprava. V ta namen se lahko izkorišča znana tovarniška strojna ali programska napaka, pogosto pa se napake umetno povzroča, za kar je običajno potrebno dobro poznavanje strojne opreme.

Že ime *diferencialna analiza napak* implicira, da bomo kriptografski postopek opazovali večkrat, vendar tokrat pri istem besedilu in ključu, razlika pa bo v rezultatu zaradi povzročenih računskih napak.

(Obstaja tudi nediferencialna analiza napak, pri kateri nas ne zanima pravilen rezultat kriptografske operacije, pač pa namerno uničimo del naprave ali pa uporabimo že narobe delujočo napravo in poskušamo iz nje izluščiti ključ.)

Primeri dejanj, ki lahko povzročijo računske napake, so: [11]

- znižanje (ali zvišanje) napajalne napetosti preko tovarniško določenih mejnih vrednosti,
- izpostavljanje naprave močnim elektromagnetnim sevanjem,
- dajanje neustreznega takta (ure) itd.

Ker ta dejanja vedno vplivajo le na nekatere komponente vezja ali procesorja, je z nekaj eksperimentiranja možno doseči predvidljive posledice. Idejno preprosta je na primer preprečitev spreminjanja števca v rutini za prepisovanje dela pomnilnika na izhodna vrata. Če procesor ne šteje, koliko podatkov je že poslal, nam bo v neskončni zanki prekopiral kar celoten pomnilnik; če je zasebni ključ v istem naslovnem prostoru kot glavni pomnilnik, smo ga s tem že pridobili!

Nekateri kriptografi napadov z izkoriščanjem računskih napak ne štejejo med napade s stranskim kanalom. V vsakem primeru pa so za sinhronizacijo napada (tj. povzročanja računskih napak) bistvenega pomena informacije s stranskih kanalov.

2.4.2 Primer napada na DES

Prvi napadi z diferencialno analizo napak (Boneh, Demillo in Lipton, 1996) so bili naperjeni proti shemam z javnimi ključi (kot je RSA). Dve leti kasneje sta Biham in Shamir predstavila tovrsten napad na poljubno simetrično

kriptografsko shemo (in konkretno na DES). [12] Možno je celo rekonstruirati S-škatle neznane kriptografske sheme, ki je podobna DES-u. Tu bo predstavljen le napad na DES.

Avtorja sta predpostavila, da v kriptografski napravi (pametni kartici) zaradi napadalčeve aktivnosti prihaja do negiranja vrednosti naključnih bitov v registrih, pri čemer so te napake tako redke, da se pri eni kriptografski operaciji redko ,,pokvarita" dva bita ali več. Pri tem velja še predpostavka, da napadalec ne pozna niti lokacije bita, ki je bil negiran, niti točnega časa, ko je prišlo do te napake (in če sploh je).

Pri svoji simulaciji napada na DES sta avtorja predpostavila še, da do napake pride v enem izmed 16 krogov v registru, ki hrani desnih 32 bitov podatkov. Možnih lokacij napak je torej $16 \cdot 32 = 512$. Ko napadalec dobi par različnih kriptogramov istega besedila, predpostavi, da je en pravilen, v drugem pa je prišlo do take napake. Potem poskuša ugotoviti, v katerem krogu je prišlo do napake, in iz razlike izluščiti bite zadnjega podključa.

- Če se je napaka pojavila v zadnjem (16.) krogu, je (pred končno permutacijo) v desni polovici napačen le en bit, v levi pa tisti biti, ki so izhod ene ali dveh S-škatel, ki sta na vhod dobili pokvarjen bit. Ker so vhodi v škatle 6-bitni (izhodi pa 4-bitni), ne moremo deterministično določiti vhoda vanje, ampak v povprečju ostanejo le 4 možni vhodi, tako da sčasoma eliminiramo vse, razen enega.

- Če se je napaka pojavila v 15. krogu, bo v desni polovici več napak, s poskušanjem pa najdemo tiste bite, ki bi lahko ob okvari v 15. krogu povzročili napako, ki smo jo zaznali pri primerjavi pravega in napačnega kriptograma. Isto lahko naredimo še za 14. krog. V teh dveh primerih za vhode v S-škatle uporabimo glasovalno metodo: za vse možne vhodne kombinacije, ki dajo pravi izhod, glasujemo, na koncu pa kot pravilno upoštevamo kombinacijo z največ glasovi.

Na ta način sta s simuliranjem tovrstnih napak s kriptiranjem od 50 do 200 besedil našla celoten zadnji podključ, ki je znana permutacija 48 izmed 56 bitov ključa. (Preostalih 8 bitov lahko nato uganemo s poskušanjem, ali pa izračunamo stanje pred zadnjim krogom in od tam nadaljujemo isti napad, s čimer je možno napasti tudi trojni DES s tremi neodvisnimi ključi.) Avtorja sta eksperimentalno dokazala, da napad deluje tudi z manj restriktivnimi predpostavkami glede napak (npr. da se napaka lahko pojavi tudi v levem delu podatkov ali celo tik pred vhodom v Feistelovo funkcijo), če pa napadalec povzročanje napak uspe časovno uskladiti s potekom enkripcije tako, da se napake pojavljajo le v zadnjih treh krogih, pa potrebuje še manj besedil

(okoli 10). Če napadalec lahko izbere še točno lokacijo napake, dobi seveda še dodatno prednost in lahko ključ najde z le tremi besedili.

2.4.3 Protiukrepi

Programski protiukrepi

Večkratno računanje Najbolj očiten protiukrep je, da kriptografsko operacijo izvedemo dvakrat in primerjamo rezultata; če je napaka naključna (kar povzročene napake praviloma so), jo bomo na ta način morda odkrili. Slabosti tega načina je več: več kot dvakrat daljše trajanje izvajanja, možnost uvedbe dodatnih ranljivosti v kodi za primerjanje rezultatov in včasih premajhna verjetnost odkrivanja napake (napadalec morda dobro časovno usklajuje svoj napad z delovanjem naprave).

Opravljanje preizkusa Bolj zanesljiva metoda ugotavljanja pravilnosti rezultata je preizkus z inverzno operacijo, npr. preverjanje podpisa (z javnim ključem) po podpisovanju (s privatnim). Napadalec bo mnogo težje, če sploh lahko, povzročil ravno take napake v sistemu, da preizkus ne bo odkril napake.

CRC Namesto preverjanja rezultata po zaključenem računanju lahko preverjamo že vse vmesne izračune z uporabo koda za odkrivanje (ali odpravljanje) napak, npr. CRC.

Programsko preverjanje vmesnih rezultatov Med razvojem (kakršnekoli) programske opreme včasih preverjamo smiselnost vmesnih rezultatov z uporabo trditev, ki morajo veljati (*assertions*), da morda odkrijemo napake v kodi. V produkcijskih verzijah običajno ta preverjanja izključimo, saj v program ne vgrajujemo logike, ki bi take napake znala smiselno obravnavati. V kriptografski programski opremi je zaradi izogibanja napadom, ki izkoriščajo računske napake, smiselno taka preverjanja ohraniti in v primeru odkrite napake ponoviti ali opustiti izračun. Na ta način odpravimo možnost, da bi ena sama napaka povzročila odpoved (*single point of failure*). Cena za to je seveda dodatno računanje in s tem počasnejše delovanje.

Dodajanje naključne vsebine Metode diferencialne analize napak temeljijo na večkratnem opravljanju iste kriptografske operacije. Pri podpisovanju se temu lahko izognemo tako, da pred podpisovanjem besedilu

vedno na konec dodamo naključne bite. S tem dosežemo, da nikoli ne podpisujemo istega sporočila večkrat, vseeno pa lahko preverimo podpis (dodane naključne bite ignoriramo). [13]

Strojni protiukrepi

Oteževanje razstavljanja in mikroskopiranjja Tipično so v pametnih karticah uporabljeni nekateri ukrepi proti enostavnemu izpostavljanju vgrajenega računalnika, kar bi omogočilo priključitev elektronskih naprav na notranja vodila ali mikroskopiranje in analizo delovanja. Kljub temu je z osnovnim znanjem kemije in javno dostopnimi kemikalijami možno uspešno razstaviti pametno kartico, z naprednejšo laboratorijsko opremo pa celo selektivno motiti ali uničevati komponente. [11]

3 Simulacija časovnega napada na RSA

Napad, ki je v poglavju 2.1.2 opisan v odstavku **Ideja 2**, sem tudi praktično izvedel oziroma simuliral. Avtorji napada so v svoji simulaciji z uporabo razhroščevalnika merili število porabljenih procesorskih ciklov; jaz sem uporabil manj natančno merilno metodo in sicer sem meril porabljen procesorski čas, kot ga za proces izmeri operacijski sistem. Izkazalo se je, da so tovrstne meritve mnogo premalo natančne, predvsem zaradi diskretizacijskega šuma. Meritev sem namreč izvajal na relativno hitrem procesorju (1,7 GHz). Problem sem rešil z dvema prilagoditvama okolja:

- meril sem čas večkratne (100-kratne) dekripcije istega besedila,
- napadenim operacijam, tj. redukcijam pri potenciranju, sem umetno podaljšal čas izvajanja s prazno zanko, ki je nekaj časa obremenjevala procesor.

Prva prilagoditev ni sporna, saj le 100-krat poveča preciznost meritve, kar je ekvivalentno napadu na 100-krat počasnejši računalnik (npr. pametno kartico) ali uporabi 100-krat natančnejše merilne opreme, kar za profesionalnega napadalca ne bi smelo biti problem. Druga prilagoditev je bolj sporna, saj močno poudari razlike med operacijami, ki se izvedejo brez redukcije, in tistimi, ki redukcijo potrebujejo. Kljub vsemu bi se z bistveno večjim številom meritev (torej bistveno večjim številom podpisanih sporočil) verjetno dalo napad izvesti brez te prilagoditve, torej tudi v realnem okolju in ne le simulaciji.

Izvorna koda aplikacije v jeziku C++, ki simulira napad, je v prilogi dokumenta. Simuliran je 64-bitni RSA.

Napad je pri 50.000 izmerjenih časih dekriptiranj uspel, torej našel vse bite ključa, ki jih je teoretično zmožen najti. Zadnjega bita ne more najti, saj mu ne sledi kvadriranje, prav tako pa mu je potrebno povedati za lokacijo prve enice v ključu, saj šele za njo kvadriranje pride do izraza (prej se kvadrira število 1 in redukcija nikoli ni potrebna). To lokacijo bi bilo seveda možno tudi ugibati oziroma preizkusiti vseh 64 položajev.

Pri 20.000 izmerjenih časih dekriptiranj napad ni uspel – narobe je določil 13. bit ključa, zato so bili *preroki* od tam naprej nesmiselni in bi teoretično morali obe predpostavki (da je naslednji bit 0 ali 1) dati približno enako razliko med besedili, ki naj bi redukcijo potrebovala, in tistimi, ki naj je ne bi. Posledično bi morala biti krivulja absolutne razlike med tema dvema razlikama bistveno nižja kot pred tem bitom. Graf 4 kaže, da sprememba ni tako drastična, da bi izdala točno lokacijo napake, je pa na grafu vseeno opazna (rdeča krivulja je od 13. bita naprej praviloma nižja od modre).

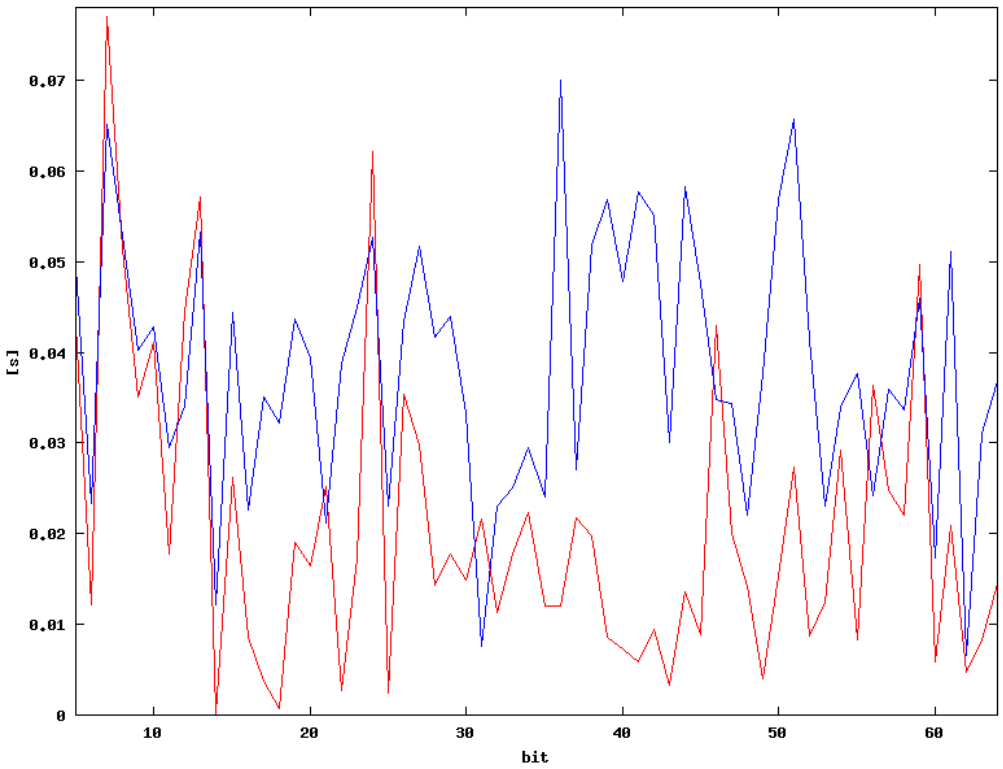
4 Zaključek

Napadi s stranskim kanalom so pomembna veja kriptoanalize, ki ji morajo snovalci sistemov pri vsaki imlementacijo posvetiti pozornost. To skupaj s konkretnimi rešitvami, ki pogosto zahtevajo tudi prilagoditve strojne opreme[1], v končni fazi podraži kriptografsko opremo.

Mnogi protiukrepi (za konkretne algoritme) delujejo proti različnim napadom s stranskim kanalom, a mnogi protiukrepi napada ne onemogočijo, pač pa ga le otežijo (da napadalec potrebuje več meritev ali natančnejše meritve).

Zaenkrat tako tudi na tem področju izgleda, da bitk med kriptografi in kriptoanalitiki še ne bo kmalu konec, vseeno pa je bilo nekaj dela na posplošitvi in matematični obravnavi napadov s stranskim kanalom že narejenega. Ideja v delu *Physically Observable Cryptography* [14] je, da si zamislimo model strojne opreme, ki *prepušča* kvečjemu določene informacije, nato pa zasnujemo tako kriptografsko programsko opremo, da napadalec ob njenem izvajanju na predpostavljeni strojni opremi ne bo dobil nobene informacije o skrivnosti, npr. ključu. Razvoj te teorije naj bi kasneje tudi usmerjal razvoj strojne opreme, saj bodo razvijalci bolj vedeli, na kaj morajo biti pozorni.

[1] za motenje meritev ali pa hitrejšo strojno opremo zaradi prilagoditve programske opreme, ki zato postane počasnejša



Slika 4: Absolutna vrednost razlike med razliko povprečnega časa za podpis besedil iz M_1 in M_2 ter razliko povprečnega časa za podpis besedil iz M_3 in M_4 (v sekundah) – podobno kot graf 1. Ključ je 64-bitni; pri modri krivulji je bilo uporabljenih 50.000 besedil in je napad uspel, pri rdeči pa 20.000 besedil in je prišlo do napake pri 13. bitu, zato je krivulja od tam naprej praviloma nižja od modre.

V članku [15] je na podlagi te teorije predstavljena transformacija programov v programskem jeziku C (v *zaščitene* programe v istem programskem jeziku), ki na predpostavljeni strojni opremi napadalcu ne prepušča informacij. Predpostavljena je taka strojna oprema, ki prepušča le vrednost programskega števca; napadalec torej pozna kriptografski program in za vsak korak enkripcije lahko izmeri vrednost programskega števca.

Zdaj pa še odgovor na vprašanje v uvodu: kje sta se s sodelavcem uštela? Preprosto, kriptoanalitik konkurenčnega podjetja si je beležil, koliko časa potrebujete za kriptiranje posameznega znaka. Ker črk najbrž ne znate seštevati intuitivno, ker se tega niste učili v šoli, pač pa štejete črko po črko, bo računanje trajalo dlje, če bo znak v ključu ,,večji", torej kasnejši v abecedi. Na podlagi tega si je kriptoanalitik močno omejil iskalni prostor, saj je za vsak znak moral preveriti le nekaj črk, njihovo približno lokacijo v abecedi pa je že poznal.

Literatura

[1] Wikipedia. Side-channel attack — Wikipedia, The Free Encyclopedia, 2009. [Na internetu; uporabljeno 10. julija 2009]. Dosegljivo na: `http://en.wikipedia.org/w/index.php?title=Side-channel_attack&oldid=301383326`.

[2] Bar-el, H. Introduction to side channel attacks, 2003. Dosegljivo na: `http://www.discretix.com/PDF/Introduction%20to%20Side%20Channel%20Attacks.pdf`.

[3] Kocher, P. C. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. *Advances in Cryptology – CRYPTO '96* (1996), 104–113.

[4] Dhem, J.-F., et al. A practical implementation of the timing attack. *UCL Crypto Group Technical Report Series*, 1 (1998). Dosegljivo na: `http://users.belgacom.net/dhem/papers/CG1998_1.pdf`.

[5] Montgomery, P. L. Modular multiplication without trial division. *Mathematics of Computation 44*, 170 (1985), 519–521.

[6] Fan, J., Sakiyama, K., and Verbauwhede, I. Montgomery modular multiplication algorithm for multi-core systems. *Proceedings of the IEEE Workshop on Signal Processing Systems: Design and Implementation (SIPS 2007)* (2007), 261–266.

[7] Köpf, B., et al. A provably secure and efficient countermeasure against timing attacks. *22nd IEEE Computer Security Foundations Symposium (CSF)* (2009). Dosegljivo na: `http://eprint.iacr.org/2009/089.pdf`.

[8] Novak, R. SPA-Based Adaptive Chosen-Ciphertext Attack on RSA Implementation. *Inštitut Jožefa Štefana* (2002). Dosegljivo na: `http://www-e6.ijs.si/~novak/papers/PKC2002.pdf`.

[9] Štimec, A. Diferencialna analiza električne aktivnosti in protiukrepi (projekt pri predmetu KTK2). FRI, UNI-LJ, 2008.

[10] Joye, M., et al. Recovering lost efficiency of exponentiation algorithms on smart cards. *Electronics Letters 38* (2002), 200–202.

[11] Anderson, R., and Kuhn, M. Tamper Resistance – A Cautionary Note. USENIX Association, 1996. Dosegljivo na: `http://www.cl.cam.ac.uk/~mgk25/tamper.pdf`.

[12] Biham, E., and Shamir, A. Differential Fault Analysis of Secret Key Cryptosystems. Springer-Ferlag, 1998. Dosegljivo na: `http://dsns.csie.nctu.edu.tw/research/crypto/HTML/PDF/C97/513.PDF`.

[13] Boneh, D., et al. On the Importance of Checking Computation. Math and Cryptography Research Group, 1996. Dosegljivo na: `http://www.demillo.com/PDF/smart.pdf`.

[14] Micali, S., and Reyzin, L. Physically observable cryptography. Cryptology ePrint Archive, Report 2003/120, 2003. Dosegljivo na: `http://eprint.iacr.org/`.

[15] Molnar, D., Piotrowski, M., Schultz, D., and Wagner, D. The program counter security model: Automatic detection and removal of control-flow side channel attacks. Cryptology ePrint Archive, Report 2005/368, 2005. Dosegljivo na: `http://eprint.iacr.org/`.

Sledi priloga: koda programa za časovni napad na RSA

```cpp
#include <iostream>
#include <cassert>
#include <iostream>
#include <vector>
#include <cmath>
#include <string>
#include <cstdlib>
#include <sys/resource.h>

using namespace std;


/* RSA parametri: 64-bit
 *  p = 0xc1052d29
 *  q = 0x50cd0449
 *M=pq= 0x3cec327d190384b1
 */
const string M_str = "3cec327d190384b1";  // modul M (v heksadecimalni obliki)
const unsigned int Mi = 907787183;   // minus inverz M v Z(2**32)
const int n = 64; // bitov modula
const int s = 2;  // ==n/32 : st. 32-bitnih segmentov
const string d_str = "3b1057b33d9ee250";  // zasebni ključ d

unsigned int M[s+2];  // globalni RSA modul M (se inicializira na zacetku programa)

const int w = 32; // velikost strojne besede unsigned int (v bitih)


/***********************************
 * DELO S STEVILI                  *
 ***********************************/

void clearNumber(unsigned int t[], int size) {
   for (int i=0; i<size; i++)
      t[i] = 0;
}

void copyNumber(unsigned int src[], unsigned int dest[], int size) {
   for (int i=0; i<size; i++)
      dest[i] = src[i];
}

void ltrim(string& str, const locale& loc = locale()) {
  string::size_type pos = 0;
  while (pos < str.size() && isspace(str[pos], loc)) pos++;
  str.erase(0, pos);
}

void rtrim(string& str, const locale& loc = locale()) {
  string::size_type pos = str.size();
  while (pos > 0 && isspace(str[pos - 1], loc)) pos--;
  str.erase(pos);
}

void btrim(string& str, const locale& loc = locale()) {
  ltrim(str, loc);
  rtrim(str, loc);
}

void parseHexNumber(string hex, unsigned int dest[]) {
   string s(hex);
   btrim(s);
   int j=0;
   while (s.length() > 0) {
      int i = s.length()-8;
      if (i<0) i=0;
      string sub = string( s, i, s.length()-i );
      s.erase(i, s.length()-i);
      btrim(s);
      sscanf(sub.c_str(), "%x ", &dest[j++]);
   }
}

void bitShiftR(unsigned int t[], int size) {
   bool carry = false;
   for (int i=0; i<size; i++) {
      carry = t[i] & 1;
      t[i] = t[i]>>1;
      if (carry && (i>0))
      t[i-1] += (1<<31);
   }
}

void bitShiftL(unsigned int t[], int size) {
   bool carry = false;
   for (int i=size-1; i>=0; i--) {
      carry = ((t[i] & (1<<31)) > 0);
      t[i] = t[i]<<1;
      if (carry) {
         assert(i<size-1);
         t[i+1]++;
      }
   }
}

void wordShiftR(unsigned int t[], int size) {
   for (int i=0; i<size-1; i++)
      t[i] = t[i+1];
   t[size-1] = 0;
}

// Pristevanje.
void add(unsigned int to[], unsigned int what[], int size) {
```

```
   bool carry = false;
   for (int i=0; i<size; i++) {
      if (carry) {
         to[i]++;
         carry = (to[i] == 0);
      }
      unsigned int sum = to[i] + what[i];
      carry = carry || (sum < to[i]) || (sum < what[i]);
      to[i] = sum;
   }
   if (carry) {
      to[size]++;
      assert(to[size]>0);
   }
}

// Odstevanje.
void subtract(unsigned int from[], unsigned int what[], int size) {
   bool carry = false;
   for (int i=0; i<size; i++) {
      if (carry) {
         carry = (from[i] == 0);
         from[i]--;
      }
      unsigned int sub = from[i] - what[i];
      carry = carry || (sub > from[i]);
      from[i] = sub;
   }
   assert(!carry);  // Negativno stevilo.
}

// Pristevanje 1.
void inc(unsigned int X[], int size) {
   for (int i=0; i<size; i++) {
      X[i]++;
      if (X[i]>0)
         return;
   }
   assert(false);
}

// true, ce je a vecji od b
bool isGreater(unsigned int a[], unsigned int b[], int size) {
   for (int i=size-1; i>=0; i--) {
      if (a[i]>b[i])
         return true;
      if (a[i]<b[i])
          return false;
   }
   return false;
}

// Heksadecimalni izpis stevila.
void printBig(unsigned int a[], int size) {
   for (int i=size-1; i>=0; i--)
      printf("%08x ", a[i]);
   cout << "\n";
}

/* result += Y * X
 * X je velikosti xSize, result mora biti vsaj size+1.
 */
void multAdd(unsigned int X[], int xSize, unsigned int Y, unsigned int result[], int resultSize){
   assert(resultSize>xSize);
    unsigned int tmp[resultSize];
   clearNumber(tmp, resultSize);
    copyNumber(X, tmp, xSize);
    for (int j=0; j<w; j++) {
      if (Y&1)
        add(result, tmp, resultSize);
      Y = Y>>1;
      bitShiftL(tmp, resultSize);
    }
}

// X <- X mod M
void mod(unsigned int X[], int xSize, unsigned int M[], int mSize) {
   unsigned int tmpRemainder[xSize];
   clearNumber(tmpRemainder, xSize);
   unsigned int Mx[xSize];
   clearNumber(Mx, xSize);
   copyNumber(M, Mx, mSize);
   for (int i=xSize-1; i>=0; i--) {
      for (unsigned int mask = 0x80000000; mask; mask >>= 1) {
         bitShiftL(tmpRemainder, xSize);
         if ( (X[i] & mask) != 0 )
            inc(tmpRemainder, xSize);
         if (!isGreater(Mx,tmpRemainder,xSize))
            subtract(tmpRemainder, Mx, xSize);
      }
   }
   copyNumber(tmpRemainder, X, xSize);
}

/*
 * Montgomeryjeva multiplikacija X in Y po modulu M (globalna spremenljivka).
 * Ce je delayReduction==true, koncna redukcija umetno traja nekoliko dlje
 * (sicer pri tej natancnosti meritev ni mozno uspesno izvesti napada ...)
 */
bool multMontgomery(unsigned int X[], unsigned int Y[], unsigned int result[],
                    bool delayReduction) {
   unsigned int Z[s+2];
   clearNumber(Z, s+2);
```

```
   for (int i=0; i<s; i++) {
      unsigned int T = (Z[0] + X[0] * Y[i]) * Mi;

      // Z = Z + X*Yi
      unsigned int Yi = Y[i];
      multAdd(X, s, Yi, Z, s+2);

      // Z = Z+M*T
      multAdd(M, s, T, Z, s+2);

      // Z = Z / r
      wordShiftR(Z, s+2);
   }

   // Koncna redukcija.
   bool reduction = false;
   if ((Z[s] | Z[s+1]) || isGreater(Z, M, s)) {
      subtract(Z, M, s+2);
      reduction = true;
      // Da malo dlje traja, da je merljivo ...
      if (delayReduction) for (int i=0; i<99999; i++) ;
   }
    if (Z[s] | Z[s+1]) cout << "whoa";
   copyNumber(Z, result, s);
   return reduction;
}

// Pretvorba stevila v montgomeryjevo obliko.
void toMontgomery(unsigned int X[]) {
   unsigned int tmp[2*s];
   clearNumber(tmp, 2*s);
   copyNumber(X, &tmp[s], s);   // Shift (mnozimo z 2^n).
   mod(tmp, 2*s, M, s);
   copyNumber(tmp, X, s);
}

// Pretvorba stevila iz montgomeryjevo v navadno binarno obliko.
void fromMontgomery(unsigned int X[]) {
   unsigned int Y[s];
   clearNumber(Y, s);
   Y[0]=1;
   multMontgomery(X,Y,X,false);
}

// Algoritem kvadriaj-in-zmnozi z uporabo Montgomeryjevega mnozenja. X <- X^exp.
void squareAndMultiply(unsigned int X[], unsigned int exp[], int eSize) {
   unsigned int ORIG[s];
   copyNumber(X, ORIG, s);

   clearNumber(X, s);
   X[0] = 1;

   toMontgomery(X);
   toMontgomery(ORIG);
   for (int i=eSize-1; i>=0; i--)
      for (unsigned int mask=0x80000000; mask; mask>>=1) {
         multMontgomery(X, X, X,true);  // square

         if (exp[i]&mask)
            multMontgomery(X, ORIG, X,false);  // multiply
      }
   fromMontgomery(X);
}




/***********************************
 * PROGRAM ZA DEKRIPTIRANJE        *
 ***********************************/

// Za 10.000 nakljucnih kriptogramov izvede dekripcijo (100-krat, da je cas merljiv) in
// vsak kriptogram ter cas dekripcije izpise na standardni izhod.
void prepareData() {
   unsigned int d[s];
   clearNumber(d, s);
   parseHexNumber(d_str, d);

   const int N = 10000;
   const int repeats = 100;
   for (int k=-5; k<N; k++) {
      unsigned int X[s];
      unsigned int X_tmp[s];
      clearNumber(X, s);
      for (int i=0; i<s; i++)
         X[i] = rand() * rand();
      if (k>=0)
         printBig(X, s);

      int who = RUSAGE_SELF;
      struct rusage start;
      getrusage(who, &start);

      for (int i=0; i<repeats; i++) {
         copyNumber(X, X_tmp, s);
         squareAndMultiply(X, d, s);
         if (i<repeats-1) copyNumber(X_tmp, X, s);
      }
      struct rusage end;
      getrusage(who, &end);

      if (k>=0) {  // Najprej smo jih nekaj kriptirali za "ogrevanje", da se cacheira, kar se.
         printf("%ld\n", (end.ru_utime.tv_sec - start.ru_utime.tv_sec)*1000000 +
```

```
                          (end.ru_utime.tv_usec - start.ru_utime.tv_usec));
            cout.flush();
            if ((k%100)==0) {
               cerr << k;
               cerr << "\n";
               cerr.flush();
            }
         }
      }
   }

   /**********************************
    * PROGRAM ZA NAPAD                *
    **********************************/

   // Kriptogram 'message' je dekriptiran v casu time.
   struct attackedMessage {
      unsigned int message[s];
      int time;
      unsigned int state[s];      // Stanje po zadnjem ugotovljenem bitu.
      unsigned int tmpstate0[s];    // ... ce je bil zadnji bit 0
      unsigned int tmpstate1[s];    // ... ce je bil zadnji bit 1
   };

   // Za kriptograme v vektorju 'message' izracuna razliko povprecnih casov dekriptiranja
   // med tistimi, pri katerih je potrebno opraviti redukcijo, in tistimi, pri katerih je
   // ni potrebno, pri predpostavki, da je naslednji bit kljuca enak 'thisBit'.
   // V polju 'state' teh kriptogramov so ze delno dekriptirani kriptogrami (s toliko
   // biti kljuca, kolikor jih je pred tem 'ugibanim' bitom). Stanje po naslednjem
   // kvadriranju in mnozenju ta metoda shrani v tmpstate0 oziroma tmpstate1 (glede na
   // thisBit); klicoca metoda ustrezno izmed teh dveh polj prekopira v 'state' za
   // naslednjo iteracijo.
   double timeDiff(vector<attackedMessage*> messages, bool thisBit) {
      long long int timeSum[2];
      int count[2];
      timeSum[0]=0; timeSum[1]=0;
      count[0]=0; count[1]=0;
      for (int i=0; i<messages.size(); i++) {
         attackedMessage *m = messages[i];
         unsigned int X[s];
         copyNumber(m->state, X, s);

         multMontgomery(X, X, X, false);  // square

         if (thisBit)
            multMontgomery(X, m->message, X, false);  // multiply

         copyNumber( X, thisBit ? m->tmpstate1 : m->tmpstate0, s);

         bool r = multMontgomery(X, X, X, false);  // square
         timeSum[r] += m->time;
         count[r]++;
      }
      double avgTime[2];
      for (int i=0; i<2; i++)
         avgTime[i] = (double)timeSum[i] / count[i];
      printf("0: %10lld/%5d=%10e    1: %10lld/%5d=%10e   ", timeSum[0], count[0],
             avgTime[0], timeSum[1], count[1], avgTime[1]);
      return avgTime[1] - avgTime[0];
   }

   // Podprogram za napad.
   void attack() {
      // S standardnega vhoda nalozi, kar je pripravila metoda 'prepareData'.
      vector<attackedMessage*> messages;
      cout << "Nalagam ...\n";

      while (!cin.eof()) {
         string msg_str;
         string time_str;
         getline(cin, msg_str);
         if (cin.eof() || (msg_str.length()==0))
            break;
         getline(cin, time_str);

         attackedMessage *m = new attackedMessage;

         parseHexNumber (msg_str, m->message);
         toMontgomery(m->message);
         sscanf(time_str.c_str(), "%d", &m->time);
         clearNumber(m->state, s);
         m->state[0] = 1;
         toMontgomery(m->state);

         messages.push_back(m);
      }

      // d je zasebni kljuc; tega "ugibamo"
      unsigned int d[s];
      clearNumber (d, s);
      // Pravi zasebni kljuc nalozimo le za izpis -- v resnici ga "ne vemo", a to je simulacija.
      unsigned int d_orig[s];
      parseHexNumber (d_str, d_orig);

      cout << "Napadam ...";

      int firstOne = 2;  // Na 3. mestu je prva enica; to nam je nekdo prisepnil ...

      for (int bg=s-1; bg>=0; bg--)     // za vsako besedo kljuca ...
         for (int b=w-1; b>=0; b--) {    // za vsak bit besede kljuca ...
            bool bit;

            printf("\n  bit %d, %02d ... ", bg, b);
```

```
413             if ((bg==0) && (b==0)) continue;    // Zadnjega bita ne moremo ugotoviti.
414             if ((bg==s-1)&&(b>=w-1-firstOne)) { // Prvih nekaj bitov vemo (do vkljucno prve 1).
415                bit = (b==w-1-firstOne);
416                timeDiff(messages, bit);
417                printf("%d", bit);
418             } else {
419                // Predpostavimo vrednost 0.
420                int diff0 = timeDiff(messages, false);
421
422                // Predpostavimo vrednost 1.
423                unsigned int zerodbg = d[bg];
424                d[bg] |= 1<<b;
425                int diff1 = timeDiff(messages, true);
426
427                // Izberemo ustrezno vrednost bita.
428                bit = true;
429                if (diff0>diff1) {
430                   bit = false;
431                   d[bg] = zerodbg;
432                }
433                printf("%d (pravilno %d, razlika: %e)", bit, ((d_orig[bg]>>b)&1),
434                       fabs(diff0-diff1));
435             }
436
437             // Shranimo stanje sporočil.
438             for (int i=0; i<messages.size(); i++)
439                copyNumber(bit ? messages[i]->tmpstate1 : messages[i]->tmpstate0,
440                           messages[i]->state, s);
441          }
442       }
443
444
445
446
447    /*****************************************************************************
448     * GLAVNI PROGRAM                                                            *
449     * Poklice podprogram za dekripcijo ('prepareData'), ce je podan parameter 'prepare'. *
450     * Poklica podprogram za napad ('attack'), ce je podan parameter 'attack'.           *
451     *****************************************************************************/
452
453    int main(int argc, char *argv[]) {
454       srand(time(0));
455
456       clearNumber(M, s+2);
457       parseHexNumber(M_str, M);
458
459       if (argc<2) {
460          cout << "parameter 'prepare' ali 'attack'\n";
461          return 1;
462       } else if (string(argv[1])=="prepare") {
463          prepareData();
464       } else if (string(argv[1])=="attack") {
465          attack();
466       } else {
467          printf("parameter 'prepare' ali 'attack', ne pa '%s'\n", argv[1]);
468          return 1;
469       }
470    }
```