Anpassning av GNU Nettle för inbyggda system

Niels Möller, South Pole AB

April 2013

Innehåll

1 Inledning

Projektet "Anpassning av GNU Nettle för inbyggda system" har pågått under våren 2013, med finansiering från Internetfonden.

GNU Nettle är ett minimalistiskt bibliotek som tillhandahåller kryptografiska byggstenar: hashfunktioner, som MD5, SHA1, och SHA256, krypteringsalgoritmer, som DES, AES, Camellia, Salsa20, och digitala signaturer, som RSA och DSA. Nettle är licensierat under GNU Lesser General Public License (LGPL), och releaser är tillgängliga på `ftp://ftp.gnu.org/gnu/nettle`.

Inbäddade och mobila enheter blir allt vanligare. Samtidigt är avlyssning av Internetkommunikation på frammarsch. Effektiv kryptering även på mobila enheter är därför avgörande för att skydda privat kommunikation.

Många inbäddade och mobila enheter, bland annat smartphones och hemmaroutrar, använder processorer ur ARM-familjen. GNU Nettle har fram tills det här projektet saknat optimeringar specifikt för dessa processorer.

2 Mål och syfte

Projektets övergripande syfte är att göra kryptering med hjälp av biblioteket GNU Nettle mer effektivt för inbyggda system och mobila enheter. De två konkreta delmålen för projektet är:

- Att implementera digitala signaturer baserade på elliptiska kurvor (ECC). Jämfört med traditionella digitala signaturer med RSA, så ger elliptiska kurvor motsvarande säkerhet med betydligt mindre processortid. Även minnesåtgång, både för själva beräkningen, och för nycklar och signaturer, är betydligt mindre.
- Att optimera andra viktiga kryptografiska byggstenar, som krypteringsalgoritmen AES och hashfunktionerna SHA1 och SHA256, för ARM-arkitekturen.

3 Projektbeskrivning

3.1 Utvecklingssystem

För ARM-utveckling, har jag använt en Pandaboard med en ARM Cortex-A9 processor, `http://www.pandaboard.org/content/platform`, med operativsystemet Debian GNU/Linux.

3.2 Elliptiska kurvor

För kryptografiska ändamål är en elliptisk kurva en algebraisk grupp av "punkter på kurvan". För de kurvor som jag har arbetat med, så definieras kurvan av ekvationen

$$y^2 = x^3 - 3x + b \pmod{p}.$$

Koordinaterna x och y ska alltså räknas modulo ett primtal p, och "punkterna på kurvan" är helt enkelt de par av (x, y) som uppfyller ekvationen. Kurvan definieras av primtalet p och konstanten b. Det som gör elliptiska kurvor intressanta för kryptografiska tillämpningar är att man kan konstruera en sorts

”addition”, som givet två punkter på kurvan ger en ny punkt. Om man dessutom kompletterar med en oändlighetspunkt, så får man något som uppfyller algebrans axiom för en ”grupp”.

3.3 Designval för elliptiska kurvor

Elliptiska kurvor är ett brett område. I litteraturen finns många olika föreslagna kurvor och en uppsjö av olika formler och algoritmer. I inledningsfasen av projektet var det därför ett antal val som behövde göras.

3.3.1 Val av kurvor

Det första beslutet gällde vilka kurvor som ska stödjas. Den vanligaste algoritmen för digitala signaturer baserade på elliptiska kurvor är ECDSA. I NIST:s specifikation, `http://csrc.nist.gov/publications/fips/fips186-3/fips_186-3.pdf`, rekommenderas fem kurvor, med olika säkerhetsnivå. Motsvarande primtal p är valda för att reduktion modulo p ska kunna göras enkelt med ett fåtal skift och additioner. Samtliga dessa kurvor har implementerats. NIST rekommenderar också fem kurvor av en annan typ, där aritmetik på koordinaterna inte görs med vanlig modulo; dessa har inte implementerats.

3.3.2 Sidokanaler

En ”sidokanalsattack” på ett kryptosystem är en attack som utnyttjar annan information än de överförda meddelandena. Till exempel observation av tidsåtgång och strömförbrukning för den maskin som utför krypteringen. En annan typ av attack är möjlig om man har flera processer som kör på samma maskin, och delar processorns cacheminne. Genom att själv göra operationer som påverkar cachens interna tillstånd, och sedan mäta hur lång tid olika minnesaccesser tar, kan en process få information om vilka minnesadresser en *annan* process läser. Och om den andra processen till exempel använder delar av en hemlig nyckel som index i en tabell, så kan den här typen av attacker ge information om den hemliga nyckeln.

Både attacker baserade på mätning av tidsåtgång, och attacker via processorns cache, går att skydda sig mot genom att skriva de rutiner som hanterar hemligt data så att de exekverar precis samma instruktionssekvens och läser precis samma minnesadresser, oavsett innehållet i nyckel och meddelande. Man behöver undvika alla villkorliga hopp som beror på innehåll, och en enkel tabelluppslagning måste ersättas med kod som läser *hela* tabellen från början till slut, och använder logiska operationer för att extrahera rätt element.

Timing-attacker är relevanta för de flesta system som kommunicerar över Internet, och i synnerhet sådana som är ganska långsamma. Cache-baserade attacker kan vara relevanta för till exempel smartphones, där många program av olika grad av pålitlighet körs på samma enhet. Jag bestämde mig därför för att sikta på att implementera elliptiska kurvor på ett sådant sätt att den inte ska läcka information via timing eller cache. Strömförbrukning har jag däremot inte tittat på. Att skydda sig mot läckor den vägen kräver troligen ingående kunskap om aktuell hårdvara.

3.3.3 Koordinatrepresentation

Addition av punkter kan skrivas på lite olika sätt, baserat på modulo-p-aritmetik på koordinaterna. Det enklaste sättet kräver även en division modulo p, vilket är en mycket dyrare operation än multiplikation och addition. Man kan eliminera nästan alla divisioner genom att införa en extra koordinat; en sammanställning av olika varianter finns på `http://www.hyperelliptic.org/EFD/g1p/auto-shortw.html`.

Nästa designval gällde representation av koordinaterna. Ett effektivt och förhållandevis enkelt sätt är så kallade Jacobi-koordinater. Istället för att räkna med de två koordinaterna (x, y) inför man tre koordinater (X, Y, Z), som hänger ihop med de ursprungliga koordinaterna genom variabelbytet

$$x = X/Z^2 \qquad\qquad y = Y/Z^3.$$

Poängen med den redundanta koordinaten Z är att när man med de ursprungliga koordinaterna skulle behöva dividera, så kan man i stället multiplicera Z med motsvarande tal.[1]

3.3.4 Algoritmer för skalärmultiplikation

För kryptografiska ändamål räcker det inte att kunna addera två punkter, man behöver även en ”skalärmultiplikation” som tar ett heltal k och en punkt P och räknar ut en ny punkt kP genom att i princip addera P till sig självt upprepade gånger. Baserat på addition och dubblering ($2P = P + P$) kan detta göras på många olika sätt. För att till exempel multiplicera en punkt P med talet 7, är några alternativ:

$7P = P + P + P + P + P + P + P$	6 additioner
$7P = 2(2(2P)) - P$	3 dubbleringar, en subtraktion
$7P = 2(2P + P) + P$	2 dubbleringar, 2 additioner

Den sista varianten ovan kan generaliseras till en enkel och effektiv algoritm som multiplicerar en punkt med ett godtyckligt tal. Om detta tal är n bitar, så kan man genom att arbeta på en bit i taget multiplicera med hjälp av högst n additioner och n dubbleringar.

Skalärmultiplikation är användbar för kryptografi för att det är en ”enkelriktad funktion”: Det är förhållandevis enkelt att räkna ut kP för vilket heltal k och vilken punkt P som helst. Men om man bara får punkten P och resultatet av multiplikationen, punkten kP, så är det i praktiken omöjligt att räkna ut k.

Ett nyckelpar i ECDSA består av en hemlig nyckel, ett heltal k, och en publik nyckel som är punkten $V = kG$, där G är generatorn för den elliptiska kurva man valt att använda. Säkerhetsnivån för en elliptisk kurva är ett sätt att kvantitativt beskriva hur svårt det är att givet bara den publika nyckeln knäcka systemet genom att räkna ut motsvarande hemliga nyckel.

För ECDSA finns det två fall av multiplikation, som kan optimeras på olika sätt. För att skapa en signatur, behöver man multiplicera en fix punkt, gruppens

[1] Det enklare ”projektiva” variabelbytet $x = X/Z$, $y = Y/Z$ ger också denna fördel, men Jacobi-koordinater har även andra fördelar. En ännu effektivare representation är ”Edwards-koordinater”, men tyvärr fungerar de inte för de standardkurvor som rekommenderas för ECDSA.

generator, med ett slumpmässigt tal. Här kan man vinna mycket tid på att förberäkna tabeller med väl valda multipler av generatorn. Man behöver en tabell per kurva, som sedan kan användas för alla signaturer man vill göra baserat på den kurvan.

För att verifiera en signatur, behöver man göra två skalärmultiplikationer, den ena med generatorn, och den andra med den publika nyckeln som ju är olika för varje användare. Det här gör att det är en betydligt dyrare operation att verifiera än ECDSA-signatur än att skapa den.

Så åter till algoritmer för skalärmultiplikation. En bra översikt finns på `http://cr.yp.to/papers/pippenger.pdf`. För skalärmultiplikation med gruppens generator, som är arbetshästen för skapande av ECDSA-signaturer, har jag valt att använda ett specialfall av Pippengers algoritm från 1976. Algoritmen beskrivs kortfattat i artikeln, och tas även upp i Handbook of Applied Cryptography, avsnitt 14.6.3, under namnet "Comb method", se `http://cacr.uwaterloo.ca/hac/about/chap14.pdf`. Algoritmen har senare återupptäckts och till och med patenteras (US Patent 5999627, som rimligen är ogiltigt).

Storleken på de förberäknade tabellerna är en trade-off mellan minne och processortid. Jag har valt att använda måttligt stora tabeller, ca 16 KByte vardera för de fem kurvor som stöds.

För skalärmultiplikation med en allmän punkt, en operation som behövs för att verifiera ECDSA-signaturer, används en rätt basal fönsterbaserad metod som först räknar ut en liten tabell, vid run-time, och sedan slår upp 4 bitar i taget ur talet man ska multiplicera med.

I grunden bygger båda de implementerade algoritmerna på upprepade dubbleringar och additioner, precis som den enklaste binära algoritmen för skalärmultiplikation. Den fönsterbaserade metoden använder tabelluppslagning för att reducera antalet additioner, men den kräver fortfarande lika många dubbleringar. Poängen med förberäkningen i Pippengers algoritm är att den reducerar *både* antalet dubbleringar och antalet additioner.

3.4 ARM-optimeringar

För många av de operationer som görs i Nettle, så kan man vinna en hel del prestanda på att implementera de mest kritiska funktionerna i assembler. Det kan handla om 10%-30% uppsnabbning tack vara att övergripande förståelse för algoritmen kan ge bättre registerallokering än C-kompilatorn klarar av. I vissa fall, när det finns vinster i att använda instruktioner eller typer som C-kompilatorn inte känner till, så kan assemblerimplementation ge mer dramatiska uppsnabbningar, uppemot fem gånger är inte ovanligt.

3.4.1 ECC-operationer

För implementationen av elliptiska kurvor, används biblioteket GNU GMP för operationer på koordinaterna, och GMP inkluderar optimerad assemblerkod för många processortyper. Men det är ändå ett par funktioner som varit aktuella för assemblerimplementation i detta projekt, och det är kod för reduktion modulo p. Varje gång ECC-implementationen behöver multiplicera två koordinater, räknar den först ut en vanlig produkt med hjälp av GMP:s funktioner. Men sen behöver produkten reduceras modulo p.

	Vikt	2^{288}	2^{256}	2^{224}	2^{192}	2^{160}	2^{128}	2^{96}	2^{64}	2^{32}	1
x	$=$	x_9	x_8	x_7	x_6	x_5	x_4	x_3	x_2	x_1	x_0
	$=$					x_5	x_4	x_3	x_2	x_1	x_0
	$+$					x_9	x_8	x_7	x_6		
	$+$							x_9	x_8	x_7	x_6

Tabell 1: Reduktion modulo $p = 2^{192} - 2^{64} - 1$. De fyra mest signifikanta 32-bits talen, $x_9, \ldots, x_6$, adderas till de mindre signifikanta orden två gånger, på positioner motsvarande vikterna 1 och 2^{64}.

För reduktion modulo p vill man utnyttja den struktur som finns i p. För att ta ett konkret exempel, så använder den minsta kurvan primtalet $p = 2^{192} - 2^{64} - 1$. Då kan vi göra reduktion genom ”vikning”, 128 bitar i taget. Låt x vara ett tal på 320 bitar, vilket på en 32-bits arkitektur som ARM representeras som en array av 10 stycken 32-bits tal x_i. Genom att utnyttja att $2^{192} = 2^{64} + 1 \pmod{p}$, kan vi vika enligt schemat i Tabell 1. För att reducera en produkt på 384 bitar till 192 behöver man vika på det här sättet två gånger.

Den här strukturen går förstås att utnyttja även för implementation i C, men det blir en del overhead eftersom det blir många anrop till GMP, var och en för rätt små operationer. En assemblerimplementation ger mindre overhead. Dessutom kan assemblerkod utnyttja att talen är så pass små att de ryms i tillgängliga register, och utnyttja add-with-carry-instruktionen till fullo. Implementationen i ARM-assembler består av 28 additionsinstruktioner samt några load och store, och ger drygt tre gångers uppsnabbning jämfört med implementationen i C och med GMP-anrop.

3.4.2 Andra kryptoprimitiver

Andra funktioner som har optimerats för ARM är memxor, AES, Salsa20, SHA1, SHA256, SHA512 och SHA3. ARM-arkitekturen har 14 stycken generella register om 32 bitar. Vissa processorer, bland andra Cortex-A9, implementerar också en utökning med vektorinstruktioner, kallad ”Neon”. Med Neon får man 16 stycken 128-bitars register, som kan representera vektorer av flyttal eller heltal av storlek upp till 64 bitar. Neon-instruktioner ger en dramatisk uppsnabbning för de algoritmer som behöver 64-bitars operationer, som hashfunktionerna SHA512 och SHA3. Även krypteringsalgoritmen Salsa20 vinner en del på användning av Neon-instruktioner som hanterar en vektorer av fyra stycken 32-bitars tal parallellt.

3.5 Övrigt

3.5.1 Optimeringar för x86_64

Under projektets gång har det varit naturligt att även göra en del optimeringar för Intel-arkitekturen x86_64. Dels för ECC-primitiver, dels för SHA256 och SHA512 för vilka det inte fanns assemblerkod sedan tidigare.

		Nettle		OpenSSL	
Algoritm	Bitstorlek	Sign./s	Ver./s	Sign./s	Ver./s
RSA	1024	495	9140	95	1901
RSA	2048	83	2683	16	590
DSA	1024	982	500		
ECDSA	192	1464	553		
ECDSA	224	1039	411	184	156
ECDSA	256	806	319		
ECDSA	384	343	136	70	59
ECDSA	521	184	74	26	21

Tabell 2: Prestanda för digitala signaturer med RSA, DSA och ECDSA, mätt som antal signaturer och antal verifieringar per sekund. Mätt på en 1 GHz ARM Cortex-A9.

3.5.2 UMAC

Även UMAC har implementerats, en förhållandevis ny algoritm för autenticering av meddelanden. UMAC beskrivs i RFC 4418, och algoritmen är konstruerad för höga prestanda på moderna datorarkitekturer med effektiv heltalsmultiplikation. Implementationen har optimerats för både ARM och x86_64, och den är drygt 10 gånger snabbare än alternativ som HMAC-SHA256 och HMAC-SHA512.

4 Leverabler

Det viktigaste leverabeln är releasen av Nettle-2.7, som inkluderar den kod som utvecklats under projektet, samt dokumentation av nya funktioner. Manualen finns även online på `http://www.lysator.liu.se/~nisse/nettle/nettle.html`. Utöver denna slutrapport har tre statusrapporter skickats till Internetfonden och till mailinglistan[2].

5 Resultat

5.1 ECDSA-signaturer

Implementationen av av digitala signaturer med ECDSA och elliptiska kurvor hanterar NIST:s fem standardkurvor, ”secp192r1”, ”secp224r1”, ”secp256r1”, ”secp384r1” and ”secp521r1”, där secp256r1 troligen är den som är mest relevant för tillämpningar. Prestanda mäts i antal signaturer och verifieringar per sekund. Se Tabell 2.

Om vi jämför 2048-bitars RSA med 224-bitars ECDSA (som anses ge samma eller till och med en aning högre säkerhet, se `http://www.keylength.com/en/3/`), är signering med ECDSA hela 12.5 gånger snabbare än med RSA. Med ECDSA är verifiering av signaturer en dyrare operation än att skapa en signatur, omkring 2.5 gånger långsammare. För RSA är det tvärtom, verifiering är många gånger

[2]Se `http://lists.lysator.liu.se/pipermail/nettle-bugs/2013/002591.html`, `http://lists.lysator.liu.se/pipermail/nettle-bugs/2013/002615.html`, `http://lists.lysator.liu.se/pipermail/nettle-bugs/2013/002667.html`.

snabbare än signering[3]. Det gör att om man jämför prestanda för verifiering är ECDSA 6.5 gånger långsammare än RSA (men verifiering av ECDSA är fortfarande betydligt snabbare än *signering* med RSA).

Det är också intressant att jämföra prestanda med OpenSSL. Här är Nettle 5–7 gånger snabbare än OpenSSL för signering med ECDSA, och 2–3 gånger snabbare för verifiering.

Mätningar på Intel x86_64 ger liknande resultat, fast med lite mindre skillnad mot OpenSSL.

5.2 Sidokanalstysta operationer

För de flesta rutiner har det varit ganska enkelt att skriva sidokanalstyst kod, och till en ganska liten prestandaförlust. Det finns två undantag.

Det ena är tabelluppslagning, där man för att undvika sidokanaler måste läsa igenom hela tabellen för att plocka ut ett element. Det blir ändå begränsad kostnad, tack vare att de största tabeller som används, för Pippengers algoritm, accessas delvis sekventiellt som en del av algoritmen, och därmed är det för varje uppslagning bara en deltabell som behöver läsas igenom helt.

Det andra undantaget är modulär invertering, alltså att givet a och p bestämma ett x sådant att $ax = 1 \pmod{p}$. Standardmetoden är att gå via en algoritm för största gemensamma delare, antingen Euklides' algoritm eller Steins binära algoritm. Men båda algoritmerna innehåller steg som är svåra att skriva utan villkorliga hopp som beror av indata. Jag har inte hittat någon sidokanalstyst inverteringsalgoritm i litteraturen, så den algoritm jag använder, en villkorsfri variant av den binära algoritmen för största gemensamma delare, är troligen ny. Den är uppemot 50 gånger långsammare än GMP:s vanliga rutiner för största gemensamma delare, och tar ca 30%-40% av tiden för att skapa en ECDSA-signatur.

5.3 Optimering av andra kryptoprimitiver

I Tabell 3 visas prestanda för ett urval av de kryptoprimitiver som implementeras i Nettle. Arcfour och Triple-DES är äldre men vanliga krypteringsalgoritmer, AES och Salsa20 är två nyare. De olika SHA-funktionerna är kryptografiska hashfunktioner. UMAC, som nämndes tidigare, är en hashfunktion med nyckel som används för autenticering av meddelanden. XOR representerar bitvis "exklusivt eller" av två minnesblock. Bland applikationer som är i drift idag är AES och SHA1 de två viktigaste av funktionerna på listan. Framöver kan Salsa20 och UMAC bli vanliga för tillämpningar där höga prestanda är viktigt.

För de kanske viktigaste algoritmerna, SHA1 och AES, har optimeringen gett en måttlig uppsnabbning, knappt 10% för SHA1 och knappt 30% för AES. Den största uppsnabbningen ser vi för SHA512 och SHA3, som båda bygger på 64-bits operationer. Även för UMAC blev det mer än en fördubbling av prestanda, tack vare användning Neon-instruktioner. En assemblerrutin för UMAC i ARM-assembler *utan* Neon-instruktioner skulle troligen också kunna ge en rätt bra uppsnabbning, och vore till stor nytta för de ARM-processorer som saknar Neon.

[3]Detta gäller om man väljer en liten publik exponent för RSA, i de här mätningarna användes $e = 65537$.

Namn	C-kod MByte/s	ARM-assembler MByte/s	Skillnad, %
AES-128	17.3	22.1	27
AES-192	14.5	18.5	28
AES-256	12.5	16.0	28
Arcfour	46.8		
Triple-DES	4.7		
Salsa20	39.8	58.2	46
SHA1	55.7	60.7	9
SHA256	24.6	31.7	29
SHA512	7.8	30.4	291
SHA3-256	5.5	26.0	378
SHA3-512	2.9	13.9	380
UMAC-32	379.6	932.1	146
UMAC-128	149.8	349.0	133
XOR	987.3	1905.9	93

Tabell 3: Prestanda för några av Nettles kryptografiska algoritmer. Mätt på en 1 GHz ARM Cortex-A9.

En erfarenhet är att Neon-instruktionerna är användbara och trevliga att arbeta med, i synnerhet i jämförelse med det ganska hemska hopkok av vektorinstruktioner som finns för x86-processorer, med SSE2 och relaterade utökningar.

6 Utvärdering och analys

6.1 Utvärdering av resultat

Den funktionalitet som planerades har blivit klar. Det fanns inga kvantitativa mål för prestanda, men för ECDSA är det ganska tillfredsställande att signering går en storleksordning snabbare än RSA, och att implementationen dessutom är ett par gånger snabbare än OpenSSL.

På ARM-sidan har utvecklingen varit begränsad till en plattform, en Pandaboard med ARM Cortex-A9. Den Raspberry Pi som också har funnits tillgänglig har det inte blivit tid att utveckla för, men även den plattformen bör kunna dra en del nytta av projektet, med undantag för de assemblerrutiner som använder Neon-instruktioner.

Projektets kostnader är i huvudsak arbetstid, planerat 420 timmar. Det har dragit över planen ett par dagar, i skrivande stund har 440 timmar lagts på projektet. Förseningen beror delvis på implementationen av UMAC, som inte var planerad då projektet startade, men som ändå känns relevant för inbyggda tillämpningar som behöver höga prestanda på begränsad hårdvara.

6.2 Förslag på förbättringar

- Design och dokumentation av underliggande ECC-primitiver, för att göra det enkelt att implementera andra saker än just digitala signaturer enligt ECDSA.

- För vissa ECC-operationer, i synnerhet verifiering av ECDSA-signaturer, behandlas inga hemliga data. Då gör det inget om information om datat läcker genom sidokanaler. Det skulle gå att skriva alternativa, lite snabbare, rutiner för denna användning.
- Konfigurering vid run-time av vilka assembler-rutiner som ska användas, beroende på vilken processormodell som programmet körs på. För ARM handlar det främst om att välja om rutiner som använder Neon-instruktioner kan användas eller inte. Men en sådan mekanism vore till stor nytta även på x86, där vissa processorer har särskilda instruktioner för till exempel AES.
- Test och optimering för andra processorer i ARM-familjen.
- Det finns troligen en del utrymme kvar för optimering av de olika assemblerrutinerna, till exempel genom mer genomtänkt schemaläggning av instruktionerna.

7 Framtida arbeten

För nästa release av Nettle planeras en mindre uppstädning av programmeringsgränssnittet. Större ändringar som kan bli aktuella i kommande releaser:

- Konfigurering vid run-time av vilka assembler-rutiner som ska användas, som nämndes i föregående avsnitt. Med detta maskineri på plats blir det enklare att skriva och haka in nya assemblerrutiner för att dra nytta av de särskilda kryptoinstruktioner som finns i vissa processorer.
- Design av gränssnitt för konstruktioner som ger både kryptering och autenticering.
- Stöd för digitala signaturer där den privata nyckeln lagras i separat hårdvara, till exempel ett smartcard.
- Möjlighet att använda mini-gmp (en liten och mindre effektiv implementation av en den viktigaste funktionaliteten i GNU GMP), som en nödlösning ifall GNU GMP av någon anledning inte är tillgängligt.