MASARYKOVA UNIVERZITA
FAKULTA INFORMATIKY

Generování náhodných sekvencí v mobilních zařízeních

DIPLOMOVÁ PRÁCE

Jiří Žižkovský

Brno, Jaro 2009

Prohlášení

Prohlašuji, že tato diplomová práce je mým původním autorským dílem, které jsem vypracoval samostatně. Všechny zdroje, prameny a literaturu, které jsem při vypracování používal nebo z nich čerpal, v práci řádně cituji s uvedením úplného odkazu na příslušný zdroj.

Vedoucí práce: Mgr. et Mgr. Jan Krhovják, Ph.D.

Poděkování

Mé poděkování patří Mgr. et Mgr. Janu Krhovjákovi, Ph.D., jehož odborné vedení a cenné rady významně přispěly při vzniku této práce.

Shrnutí

Cílem této práce je prostudování možností generování náhodných sekvencí v mobilních zařízeních založených na operačním systému Symbian.

V teoretické části se práce zabývá generováním (pseudo)náhodných sekvencí s důrazem kladeným na jeho bezpečnost. Dále pak představí důležité aspekty programování pro Symbian.

V praktické části popíšeme vybrané aspekty související s integrací a implementací tří aplikací pro mobilní telefony s operačním systémem Symbian. Na základě identifikovaných zdrojů náhodnosti byly zvoleny a implementovány dva generátory pseudonáhodných sekvencí – ANSI X9.31 a Fortuna – které tyto zdroje efektivně využívají.

V závěrečné části práce je popsán výběr a implementace protokolu pro bezpečné ustavení tajného klíče sdíleného mezi několika mobilními zařízeními komunikujícími přes Bluetooth.

Klíčová slova

ANSI X9.31 PRNG, Bluetooth, Burmester-Desmedt protokol, CSPRNG, Fortuna PRNG, Nokia N73, PRBG, PRNG, Symbian

Obsah

Úvod

Při současném pokroku informačních technologií už nestačí zaměřit se pouze na tradiční zařízení jako jsou osobní počítače. Ty už dnes nejsou dostatečně „osobní“. Od statických topologií komunikace přecházíme k dynamicky se měnícímu prostředí, ve kterém mají místo přenosná zařízení jako jsou kapesní počítače, mobilní telefony nebo notebooky.

Tyto změny sebou přinášejí zcela nové podmínky, na které musí vývojáři reagovat. Bez nároku na úplnost zmíníme dva důležité rozdíly. Přenosná zařízení většinou nejsou permanentně připojena ke zdroji elektrické energie a musí vystačit s vlastním zdrojem, baterií s omezenou kapacitou. Proto je výkon některých komponent omezen tak, aby byl poměr výkon/výdrž co nejlepší. I při vývoji aplikací musí být kladen podobný důraz na efektivitu a úsporu energie.

Druhý problém plyne přímo z principu popisovaných zařízení. Mobilita a malé rozměry značně usnadňují ztrátu nebo ukradení přístroje. S tím, jak jsou tato zařízení stále výkonnější, také roste množství citlivých informací, které jsou na nich uloženy, a stávají se velmi lákavým cílem. Proto je třeba klást velký důraz na bezpečnost přístupu k zařízení (zajištěnou PINem nebo jiným kódem) i práce s uloženými daty (kalendář, kontakty, dokumenty).

Použití prostředků kryptografie je jednou z možností, jak bezpečnost zajistit. Základem mnoha kryptografických protokolů jsou náhodná nebo pseudonáhodná data, proto jsme se v této práci zaměřili na možnosti generování dat na přenosných zařízeních, konkrétně na mobilních telefonech s operačním systémem Symbian (dále jen Symbian OS).

Po analýze podmínek pro generování (pseudo)náhodných dat bude výstupem implementace dvou zvolených generátorů pseudonáhodných čísel a protokolu pro bezpečné ustavení tajného klíče sdíleného několika mobilními telefony komunikujícími přes Bluetooth.

Rozvržení kapitol

V první kapitole práce se budeme zabývat teoretickými aspekty generování náhodných a pseudonáhodných čísel. Uvedeme definici generátoru (pseudo)náhodných bitů a požadavky kladené na jeho kryptografickou bezpečnost.

Ve druhé kapitole budeme prezentovat základní koncepty operačního systému Symbian nutné k pochopení implementačních detailů v dalších dvou kapitolách, které postupně popisují generátor pseudonáhodných čísel ANSI X9.31 a Fortuna.

Poslední kapitola pojednává o možnostech ustavení sdílného klíče mezi několika mobilními telefony přes Bluetooth. Popíšeme použité technologie, vybraný protokol a jeho implementaci.

Kapitola 1

Generování (pseudo)náhodných sekvencí

Na úvod se budeme zabývat teoretickou stránkou generování náhodnosti, ať už se jedná o náhodné (resp. pseudonáhodné) bity, sekvence nebo čísla.

„Náhodně vybraná“ čísla jsou potřebná v mnoha oblastech. Nyní uvedeme jen některé příklady.

První oblastí, která bývá zmiňována, je využití náhodných čísel při simulacích přirozených dějů, které jsou předmětem výzkumu vědců různých oborů. Jedná se například o studium kolizí částic zkoumaných nukleárními fyziky či modelování chování skupin lidí v sociálních vědách, či-li využíváme náhodných čísel pro zkoumání ve své podstatě náhodných dějů. Ukážeme si, že často se naopak náhodnosti některých přirozených dějů využívá k získání náhodných dat.

Oblast simulací a modelování není jedinou oblastí, ve které jsou náhodná čísla potřebná. S ohledem na zaměření této práce zmíníme ještě oblast informatiky. Jedním s příkladů využití náhodných čísel jsou náhodnostní algoritmy. Jde o speciální nedeterministické algoritmy s velmi bohatým teoretickým pozadím a také s praktickým využitím. Na rozdíl od „klasických“ deterministických algoritmů není jejich průběh pro pevně daný vstup vždy stejný, ale může se lišit. Velmi zjednodušeně se dá říci, že algoritmus si občas hodí mincí, to znamená, že se rozhodne na základě nějakého náhodného čísla, kterým směrem se bude další výpočet ubírat. Výsledky takovýchto výpočů je pak nutno interpretovat pomocí teorie pravděpodobnosti. Tyto algoritmy jsou často silnější než jejich deterministické protějšky, proto je tato oblast cílem intenzivního výzkumu.

Ovšem pro tuto práci nejdůležitějším využitím náhodných dat je speciálně oblast *kryptografie*. Využití náhodnosti v této oblasti je opravdu velmi bohaté. Jen velmi zřídka se v některém kryptografickém protokolu nebo algoritmu nevyužívá náhodných dat, ať už se jedná o fázi návrhu nebo praktické použití. Jako příklad můžeme zmínit náhodnou bitovou sekvenci použitou jako klíč v šifrovacím schématu *One-Time Pad*, náhodná čísla zasílaná v *challenge-response* protokolech či náhodné exponenty v *Diffie-Hellmann Key Agreement* protokolu.

Nyní se ale vrátíme k samotnému pojmu náhodnosti, který jsme zatím nijak nedefinovali. Obzvláště se zaměříme na speciální požadavky, které jsou na náhodná data kladeny v kryptografii.

Nejdříve definujme „náhodnost“ čísel velmi neformálně. Ve skutečnosti termín *náhodné číslo* není příliš přesný. Můžeme se ptát, je-li číslo 5 náhodné? Zřejmě vidíme, že je tato otázka nesmyslná. Pokud tedy budeme chtít být korektní, budeme mluvit spíše o *posloupnosti* či *sekvenci náhodných čísel* s určitým rozložením hodnot, přičemž každý člen posloupnosti byl vybrán náhodně s určitou pravděpodobností odpovídající požadovanému rozložení a je nezávislý na svém předchůdci i následníku.

Nejčastěji se setkáme s požadavkem na rovnoměrně rozložená náhodná čísla z ur-

čeného intervalu. Takovou posloupnost je možné generovat například pomocí házení pravidelnou kostkou. Toto řešení je samozřejmě možné zvolit pouze v omezeném počtu případů, většinou potřebujeme náhodná čísla generovat automatizovaně a poněkud rychleji, princip však zůstává zachován. Využijeme nějakého přirozeně náhodného děje a jeho sledováním získáme náhodná data.

Není však vůbec triviální docílit toho, aby použitý zdroj náhodnosti splňoval všechny podmínky, které jsou na něj kladeny. Jde zejména o tyto:

1. Zdroj náhodnosti musí generovat náhodná nezávislá data.
2. Sběr dat musí být automatizovaný.
3. Výkonnost generování musí být pro naše účely dostatečná.
4. Zdroj náhodnosti musí být chráněn před neoprávněnými manipulacemi a ovlivňováním potenciálním útočníkem.

Většina zdrojů náhodnosti, které se nabízejí, neprodukují data v takové formě, jak by bylo třeba. Generovaná data jsou ovlivněná (v bitové sekvenci není pravděpodobnost bitu 1 rovna $0{,}5$) nebo jsou korelovaná (pravděpodobnost generovaného bitu je závislá na předchozím), proto je nutné generovanou sekvenci ještě pomocí různých technik upravovat. Toto filtrování radikálně snižuje rychlost generování, kvůli tomu se pak většinou poruší třetí podmínka.

Vzhledem k tomu, že požadujeme, aby byl sběr dat plně automatický, využívají se zdroje náhodnosti především ze dvou skupin, nazveme je *hardwarové* a *softwarové*. Hardwarové zdroje využívají náhodnosti, která je přítomna v mnoha fyzikálních dějích. Příkladem může být rozpad radioaktivních prvků, frekvenční nestabilita volně běžícího oscilátoru nebo zvuk zachycený mikrofonem či obrazová data z kamery.

Všechny tyto zdroje je nutné důkladně testovat (nejlépe online) a případně generovaná data filtrovat tak, aby nevykazovala známky výše uvedených defektů. Je zřejmé, že tyto zdroje náhodnosti budou většinou relativně velká zařízení, která nebude jednoduché chránit před běžnými poruchami hardwaru ani proti případnému útočníkovi, jehož cílem je dané zařízení vyřadit z provozu nebo (ještě lépe) ovlivnit tak, aby generovaná data byla jednoduše zneužitelná.

Z důvodů snazší implementace se často používají softwarové zdroje náhodnosti. Nejčastěji se využívá systémový čas, uživatelský vstup (čas stisku kláves nebo pohyb myši) nebo různé aktuální hodnoty zaznamenávané operačním systémem (zátěž systému, síťové statistiky). Nevýhodou je, že mnohé z nich se dají poměrně jednoduše odhadnout (nebo alespoň značně omezit množinu jejich potenciálních hodnot) či přímo ovlivnit (některé i vzdáleně), proto se většinou používá jejich kombinace a výstupy se určitým způsobem „smíchají“ dohromady. Tento proces je v návrhu generátoru kritický a výrazně určuje výsledné vlastnosti generované sekvence.

Z předchozích odstavců vyplývá, že návrh a použití generátoru vhodného pro kryptografické účely není triviálním úkolem, v některých aplikacích se navíc jedná o příliš silný nástroj a je výhodnější použít *generátor pseudonáhodných sekvencí*.

Neformálně lze říci, že sekvence, která je výstupem generátoru pseudonáhodných sekvencí, pouze „vypadá“ jako náhodná. Generování je deterministický proces, náhodnost se do této operace vnáší pouze na začátku pomocí náhodného *semínka*.

Jako první představil koncept generátorů pseudonáhodných čísel John von Neumannn v roce 1946. Generovaná sekvence vznikala tak, že následující člen posloupnosti byl prostředními číslicemi druhé mocniny předchozího čísla. Na tomto jednoduchém generátoru ukážeme principy fungování a výhody i nevýhody generátorů pseudonáhodných sekvencí.

Z výše popsaného „algoritmu" je vidět, že náhodnost je do celého procesu generování vnesena pouze na začátku, při volbě prvního členu posloupnosti (semínka). Zbytek procesu je deterministický algoritmus, který je jednoduchý na výpočet a při tom poměrně dobře „napodobuje" náhodnou posloupnost. To jsou nejdůležitější vlastnosti generátorů sekvencí pseudonáhodných čísel, náhodné semínko a deterministický algoritmus generování, výpočetně jednoduchý s „dostatečně náhodným" výstupem. Ačkoliv tyto požadavky budou stačit v případě použití v oblasti simulací, je zřejmé, že v kryptografii by nebyl takový generátor bezpečný. I jeden člen posloupnosti dostačuje k rekonstrukci celé následující posloupnosti.

Další zkoumání tohoto generátoru ukázalo, že trpí zásadním nedostatkem, posloupnost má po jisté době tendenci degenerovat na různě dlouhé cykly, jejichž detekce generátor komplikuje. Tento problém ukazuje důležitost intenzivního zkoumání algoritmu generování, aby se odhalily podobné nedostatky, které nebývají na první pohled zřejmé. Analýza algoritmu, který má „generovat náhodnost", bývá principiálně náročná, ale ukazuje se, že je velmi potřebná, protože většina běžně používaných generátorů má zásadní nedostatky. Příkladem mohou být generátory ve známých knihovnách nebo operačních systémech.

Analýzu generátoru usnadňuje, pokud je návrh algoritmu nějakým způsobem teoreticky podložen, ukazuje se, že příliš „náhodné" algoritmy jsou velmi těžko teoreticky verifikovatelné a jejich testování se provádí až na výsledných sekvencích pomocí různých statistických testů, které zmíníme později. Samozřejmě toto empirické testování nemůže nikdy plně nahradit teoretickou analýzu vlastností generátoru.

Empirické testování generátorů (pseudo)náhodných čísel je postaveno na teorii pravděpodobnosti a matematické statistice, konkrétně se jedná o testování statistických hypotéz. Vzhledem k použitému aparátu je nutné výsledky interpretovat pomocí teorie pravděpodobnosti. Závěrem jednotlivých testů nikdy nemůže být výrok o korektnosti generátoru, spíše se jedná o výsledek typu: „Tímto testem nebyl odhalen žádný defekt generátoru."

Nyní budeme předchozí úvahy formalizovat a definujeme pojmy, které se budou často používat v dalším textu. Začněme s definicí *entropie*. To je číselná hodnota určující míru neujistoty o hodnotě, kterou se náhodná proměnná X realizuje, dá se také říci, že určuje množství informace nesené proměnnou X. Existuje několik různých definic, uvedeme tu, která bývá označována jako *Shannonova* [MvOV96, str. 56, def. 2.39] a pro účely této práce je naprosto dostačující, většinou se nám totiž bude jednat spíše o její odhad než přesné stanovení její hodnoty.

Definice 1.1 *Nechť X je diskrétní náhodná proměnná nabývající hodnot $x_1, \ldots, x_n$ s pravděpodobností $p_1, \ldots, p_n$, kde $0 \leq p_i \geq 1$ a $\sum_{i=1}^{n} p_i = 1$. Pak* entropie *náhodné proměnné X je definována jako*

$$H(X) = -\sum_{i=1}^{n} p_i \log p_i$$

Zmíníme ještě následující důležité vlastnosti [MvOV96, str. 56, tvrzení 2.40]:

- Hodnoty, kterých může $H(X)$ nabývat, jsou omezeny, platí:

$$0 \leq H(X) \leq \log n$$

- $H(X) = 0$ právě tehdy, když $\exists i : p_i = 1$ a $\forall j \neq i : p_j = 0$.
- $H(X) = \log n$ právě tehdy, když $\forall i : p_i = \frac{1}{n}$

Je důležité si uvědomit, že množství entropie ve zkoumaných datech je silně závislé na znalostech potenciálního útočníka. Uveďme příklad.

Mějme 24bitový binární řetězec, který bude semínkem pro inicializaci nějakého PRBG. *Útočník 1* nemá žádné znalosti o tom, jak semínko vzniká. Jestliže bude chtít odhalit stav generátoru, nemá jinou možnost než vyzkoušet všechny možné 24bitové řetězce. Proto je z jeho pohledu entropie řetězce plných 24 bitů.

Útočník 2 má informaci o tom, že semínko je časový řetězec, formátovaný tak, že každý bajt obsahuje postupně hodinu, minutu a vteřinu. Touto znalostí je značně omezen počet různých hodnot, kterých může semínko nabývat. Oproti teoretickým 2^{24} možností, je to pouze $24 \cdot 60 \cdot 60 = 86\,400$, což (jsou-li všechny hodnoty stejně pravděpodobné) představuje přibližně $16{,}4$ bitů entropie. Tedy touto znalostí získal útočník téměř osm bitů informace.

Pokud si představíme reálnou a velmi častou situaci, že semínko je *aktuální* čas, pak útočník, který je schopen odhadnout čas generování, má situaci usnadněnou ještě více.

Z předchozích úvah je zřejmé, že při odhadech entropie bychom měli vždy uvažovat, vůči komu ji stanovujeme. Je jasné, že v průběhu vývoje generátoru nemáme nikdy možnost odhadnout, kolik toho útočník ví, a proto je v podstatě nemožné udělat rozumný odhad entropie. Dobré generátory se musí s tímto problémem vyrovnat. Vraťme se však nyní k definicím.

Definujme *generátor (pseudo)náhodných bitů (P)RBG*. Vzhledem k tomu, že je získání (pseudo)náhodných čísel z výstupu (P)RBG přímočaré, budeme pojmy *generátor (pseudo)náhodných bitů* a *generátor (pseudo)náhodných čísel* často zaměňovat.

Definice 1.2 Generátor náhodných bitů *je zařízení, jehož výstupem je posloupnost statisticky nezávislých a neovlivněných*[1] *binárních číslic ([MvOV96, str. 170, def. 5.1]).*

Definice 1.3 Generátor pseudonáhodných bitů *je deterministický algoritmus, jehož vstupem je náhodná binární sekvence délky k, tzv.* semínko *(z angl.* seed*), a výstupem je binární sekvence délky $l \gg k$, která „vypadá náhodně" ([MvOV96, str. 170, def. 5.1]).*

Tato definice není kvůli nedefinovanému pojmu „vypadat náhodně" zcela formální, ale pro tuto práci bude naprosto dostačující. Formální definici je možné najít např. v [Knu97, kap. 3.5].

Protože je proces generování deterministický, je zřejmé, že výstupem PRBG nemohou být libovolné binární sekvence délky l, ale nejvýše 2^k sekvencí. Jinak řečeno, entropie semínka se algoritmem PRBG nemůže zvýšit, tedy náhodnost výstupní sekvence závisí pouze na náhodnosti semínka.

[1] Pravděpodobnost bitu s hodnotou 1 je $1/2$.

Pokud budeme chtít generátor využít pro kryptografické účely, budeme na něj klást ještě další požadavky. Zřejmě budeme požadovat, aby velikost k semínka PRBG (resp. jeho entropie) byla dostatečně velká, abychom zabránili útoku hrubou silou. Dále požadujeme, aby generovaná sekvence nebyla výpočetně rozlišitelná od náhodné sekvence[2], a navíc nesmí být predikovatelná útočníkem.

Nyní vyslovíme dvě definice, na jejichž základě definujeme *kryptograficky bezpečný generátor pseudonáhodných bitů* (CSPRBG).

Definice 1.4 *Řekneme, že PRBG projde všemi statistickými testy s polynomiální časovou složitostí, jestliže neexistuje polynomiální algoritmus, který rozliší výstup PRBG od skutečně náhodné sekvence s pravděpodobností výrazně vyšší než* $1/2$ *([MvOV96, str. 171, def. 5.5]).*

Definice 1.5 *Řekneme, že PRBG projde* testem následujícího bitu*, jestliže neexistuje polynomiální algoritmus, který na základě prvních* i *bitů určí* $i+1$ *bit s pravděpodobností výrazně vyšší než* $1/2$ *([MvOV96, str. 171, def. 5.6]).*

Ačkoliv se zdá první definice silnější, dá se dokázat ([Yao82]), že jsou ve skutečnosti ekvivalentní, proto můžeme vyslovit definici CSPRBG.

Definice 1.6 Kryptograficky bezpečný generátor pseudonáhodných bitů *je generátor pseudonáhodných bitů, který projde testem následujícího bitu ([MvOV96, str. 171, def. 5.8]).*

Je nutné zmínit, že splnění této podmínky bývá dokazováno s pomocí aparátu vyčíslitelnosti a složitosti. Tedy nemáme formálně korektní matematický důkaz, protože je argumentováno „obtížností“ zvoleného výpočetního problému. Příkladem může být redukce na problém faktorizace, modulárního logaritmu či RSA.

V definici 1.6 předpokládáme, že útočník nemá žádnou znalost o vnitřním stavu generátoru. To je často příliš optimistický předpoklad, proto bývají požadovány ještě následující dvě komplementární vlastnosti:

- *Dopředná bezpečnost* – po kompromitaci části nebo celého vnitřního stavu PRBG nemá útočník možnost rekonstruovat dříve vygenerovaná data. U generátorů využívajících nějaký šifrovací algoritmus je této vlastnosti většinou dosaženo pravidelnou změnou klíče.

- *Zpětná bezpečnost* – po kompromitaci části nebo celého vnitřního stavu PRBG nesmí být útočník schopen predikovat další výstup generátoru. Toho je možné dosáhnout pravidelným přidáváním entropie k vnitřnímu stavu PRBG.

Otázkou je, jestli se generátor splňující tyto vlastnosti (resp. alespoň jednu z nich) dá ještě nazvat generátorem pseudonáhodných bitů podle vyslovené definice 1.3. V ní jsme řekli, že PRBG je *deterministickým* algoritmem s náhodným počátečním vstupem. Ovšem pokud dovolíme v průběhu generování změny vnitřního stavu (bez nichž není možné splňovat uvedené vlastnosti), generátor se již z globálního hlediska nechová deterministicky, pro pevné semínko neexistuje jediná výstupní sekvence.

Tyto námitky jsou samozřejmě čistě formální, proto budeme i nadále uváděné generátory označovat jako PRBG. Požadavek determinismu generátoru vznikl v oblasti

[2] Aby „vypadala náhodně“.

simulací, kde bylo nutné zajistit reprodukovatelnost výsledků experimentu. To je při kryptografickém využití PRBG většinou spíše nežádoucí, proto nebudeme od CSPRBG determinismus vyžadovat.

Posledním tématem, jímž se budeme v této části zabývat, je statistické testování náhodnosti výstupu generátorů (pseudo)náhodných sekvencí. Jak už jsme dříve zmínili, nemůže testování nikdy plně nahradit řádnou kryptoanalýzu. Testování statistických vlastností generátoru je však dobrým základem pro další analýzu a případné systematické špatné výsledky některého z testů by měly vést k zamítnutí generátoru.

Statistické testování se používá také všude tam, kde nemáme přístup k detailům návrhu a implementace (např. generátory na čipových kartách). Protože v případě naší práce k těmto detailům přístup máme, nebudeme statistické testování používat příliš intenzivně, a proto uvedeme jen základní typy používaných testů a princip jejich vyhodnocování.

Základem pro testování náhodnosti generovaných sekvencí je matematická statistika, konkrétně *teorie testování statistických hypotéz*. V našem případě bude testovanou *nulovou* hypotézou výrok: *„Vygenerovaná sekvence je náhodná“* a *alternativní* hypotézou: *„Vygenerovaná sekvence není náhodná“*.

Nejprve zvolíme vhodnou testovou statistiku, která odhaluje určitý druh nenáhodnosti. Ta má za předpokladu náhodnosti testované sekvence známé rozložení. Po vyhodnocení testové statistiky nulovou hypotézu *zamítáme* nebo *nezamítáme*. Vzhledem k pravděpodobnostní povaze výsledků neříkáme, že hypotézu přijímáme, teorie testování statistických hypotéz nám totiž nedává možnost žádnou hypotézu dokázat a přijmout, můžeme pouze s jistou pravděpodobností chyby zamítnout či nikoli.

Statistických defektů je mnoho druhů, proto existuje značné množství různých testů. V praxi se většinou používají sestavy těchto testů, nazývané *baterie*. Nejznámějšími bateriemi testů jsou DIEHARD [Mar] a NIST [FIP00], existuje i několik dalších (např. [Uni], [MT02], [CHMSS04]), ale jsou používány mnohem méně.

Kapitola 2

Symbian OS

V této kapitole uvedeme základní informace o operačním systému Symbian. Představíme nejdůležitější koncepty používání Symbian API a souvisejících knihoven. Zaměříme se zejména na funkcionalitu, která se využívá v našich aplikacích, a na prvky bezpečnosti implementované operačním systémem. Hlavním zdrojem informací pro nás byla kniha *Symbian OS C++ for Mobile Phones* [HS07].

Symbian je operační systém určený pro malá přenosná zařízení s relativně omezenými prostředky (ať už jde o zdroj energie, velikost paměti či výkon procesoru). Tato omezení od počátku určovala návrh a vývoj operačního systému.

Operační systém a vestavěné aplikace jsou uloženy v paměti ROM, ostatní aplikace a data jsou umístěna v RAM, která má jen omezenou velikost. Proto je na správu paměti v OS i jednotlivých aplikacích kladen velký důraz.

Zařízení s operačním systémem Symbian obsahuje mnoho vstupně/výstupních periferií (klávesnice, dotyková obrazovka, paměťové karty, USB, Wi-Fi, Bluetooth, ...), proto musí být OS poměrně komplexní, na druhou stranu ale nesmí zbytečně plýtvat omezenými prostředky. Z toho plyne důležitost mechanismů pro úsporu energie.

Protože jsou zařízení používající Symbian stále výkonnější, umožňují uživatelům zpracovávat více informací. Přitom některé jsou velmi citlivé (kontakty, kalendář, dokumenty). Proto je důraz na bezpečnost dalším důležitým aspektem při vývoji OS (zejména v řadě 9.x).

2.1 Základy Symbian OS

Jádro operačního systému Symbian se skládá ze dvou částí, nazývaných *nanojádro* a *mikrojádro*.

Nanojádro se zabývá operacemi vyžadujícími obsluhu v reálném čase, je zodpovědné za synchronizaci, přesné časování, obsluhu přerušení nebo plánování běhu vláken. Vytváří velmi malý RTOS (*Real Time Operating System*).

V mikrojádře se zpracovávají ostatní operace, které je potřeba provádět v privilegovaném režimu procesoru. Další jsou dostupné jako moduly běžící v uživatelském režimu.

MMU (*Memory Management Unit*) poskytuje lineární virtuální paměť. Z hlediska ochrany paměti je nejmenší jednotkou *proces*, pro procesor jsou to *vlákna*, která jsou plánována preemptivně. Paměťový prostor procesů je oddělen, zatímco vlákna jednoho procesu paměť sdílejí. Procesy pro komunikaci mezi sebou používají *meziprocesovou komunikaci* (*Inter-process communication*).

Velmi důležitou komponentou operačního systému Symbian je správa spotřeby. Problémy, kterými se musí zabývat, jsou velmi komplexní. Jak reagovat na nízký stav bate-

rie? Které periferie je možné přepnout do úsporného režimu nebo vypnout? Otázkou je i optimalizace běhu aplikací.

Operační systém Symbian je postaven na dialektu jazyka C++, většina principů je zachována. Navíc s rostoucím výkonem symbianovských zařízení přechází do Symbian C++ stále více prvků, které se dříve s ohledem na omezené prostředky řešily odlišně od standardu.

2.1.1 Cleanup Stack

Příkladem mohou být výjimky a jejich obsluha za běhu aplikace. Tak, jak ji definuje standard C++, je velmi náročná. Proto byl v Symbianu zaveden systém výjimek nazývaných *Leaves*.

Základním prvkem je *cleanup stack*, zásobník ukazatelů na objekty v paměti, které musí být smazány, pokud se objeví výjimka. Ukazatele se neukládají, resp. neodebírají automaticky, ale pomocí statických metod zásobníku (`CleanupStack::PushL()`, resp. `CleanupStack::Pop()`).

Výjimka se vyvolá metodou `User::Leave()`. Je možné ji odchytit pomocí makra `TRAP`. Konvencí je, že funkce, které mohu vyvolat výjimku, obsahují na konci svého názvu `L`.

2.1.2 Dvoufázová konstrukce objektů

Dalším rozdílem oproti standardizovanému C++ je způsob vytváření instancí tříd. Běžně by se nový objekt zkonstruoval tímto řádkem kódu:

```
Trida *objekt = new Trida();
```

Pokud konstrukce selže, vrátí se nulový ukazatel. Po každé konstrukci tak musíme testovat, není-li výsledný ukazatel `NULL`. Kód je pak nepřehledný, navíc je možné na testování úplně zapomenout. Proto se v Symbian C++ operátor `new` přetěžuje tak, aby v případě chyby vyvolal výjimku. Typicky kód vypadá následovně:

```
Trida *objekt = new (ELeave) Trida();
```

To ale samo o sobě nezabrání plýtvání pamětí v případě selhání konstrukce složitých objektů. Proto se zavádí dvoufázová konstrukce. Potenciálně nebezpečné operace (konstrukce atributů objektu, volání metod, které mohou vyvolat výjimku, apod.) jsou sdruženy do metody `ConstructL()`, která se volá až po uložení konstruovaného objektu na cleanup stack. Ten zajistí v případě chyby korektní destrukci objektu a uvolnění paměti. „Bezpečné“ operace se provádějí v běžném C++ konstruktoru.

Tato koncepce by byla náchylná k programátorským chybám (je nutné zavolat obě fáze konstrukce a ve správném pořadí), proto se většinou zavádí statická metoda `NewL()`, která zapouzdřuje obě fáze a vytváří tak bezpečný princip konstrukce.[1] Metodu `ConstructL()` je pak možno deklarovat jako privátní a máme zaručeno, že objekt bude zkonstruován korektně.

[1] Často se poskytuje i statická metoda `NewLC()` se stejnou funkcionalitou, rozdílem je, že tato ponechává konstruovaný objekt po vytvoření na cleanup stacku.

2.1.3 Aktivní objekty

Koncept aktivních objektů byl v Symbian C++ zaveden z důvodu potřeby řešit obsluhu asynchronních operací tak, aby byla jednoduchá a hlavně efektivní. To znamená žádné plýtvání omezenými zdroji aktivním čekáním ve smyčce, omezení potřeby synchronizace a podobně.

V operačním systému Symbian jsou aktivní objekty preferovanou cestou, jak řešit zpracování více úloh současně. Ve většině dnešních systémů jsou pro tuto funkci vyhrazena vlákna. Na rozdíl od vláken aktivní objekty nenabízejí preemptivní multitasking, proto je možné se vyhnout nutnosti synchronizace, která je velmi neefektivní.

Aktivní objekty se vytvářejí jako potomci třídy `CActive`. Jako atribut obsahují proměnnou `iStatus` uchovávající stav objektu a několik metod.

Popíšeme ty povinné (v `CActive` deklarované jako virtuální), jsou to `SetActive()`, `Cancel()`, `RunL()`, `RunError()` a `DoCancel()`. První dvě jsou většinou děděny beze změn, zbývající tři se musejí implementovat. Obvykle se definuje ještě další metoda `StartL()`, která požádá o asynchronní operaci, zavolá `SetActive()` a tím přidá tento objekt do prioritní fronty spravované plánovačem aktivních objektů (*Active Scheduler*), který je zodpovědný za obsluhu dokončených operací a volá metodu `RunL()` odpovídajícího aktivního objektu.

Protože metoda `RunL()` je provedena atomicky, měla by být co nejkratší, jinak může způsobit zdržení obsluhy jiných operací. Pokud tato metoda vyvolá výjimku, plánovač zavolá metodu `RunError()`.

Předčasně ukončit aktivní objekt je možné voláním metody `Cancel()`, která pouze zkontroluje, jestli je objekt právě nastaven jako aktivní a volá `DoCancel()`, jejíž implementace by měla zajistit korektní ukončení nedokončených operací.

I když je obsluha asynchronních operací poměrně komplexní, koncepce aktivních objektů zavedená v Symbianu ji maximálně zjednodušuje.

2.1.4 Bezpečnostní model

Bezpečnostní model operačního systému Symbian zahrnuje filosofii, architekturu a implementaci mechanismů, které mají omezit dopady špatně napsaného nebo škodlivého kódu. Potřeba těchto mechanismů stále roste, protože jak se funkcionalita mobilních zařízení zvyšuje, ukládáme do nich víc a víc citlivých informací. Nevýhodou zůstává, že veškerá bezpečnostní opatření musejí být dostatečně jednoduchá, aby co nejméně omezovala uživatele a ten byl ochoten je používat.

Bezpečnost přenosných zařízení je možné rozlišit do dvou kategorií, *fyzickou* a *aplikační*. Zařízení s operačním systémem Symbian typicky nebývají zabezpečena na úrovni hardwaru. Předpokládá se, že pokud útočník získá fyzický přístup k zařízení, má možnost získat veškerá uložená data. Proto je nutné, pokud to povaha uchovávaných dat vyžaduje, zajistit bezpečnost na aplikační úrovni.

Ve srovnání s předchozími verzemi je v Symbianu 9.x důraz na bezpečnostní aspekty celého systému podstatně větší. Kvůli zajištění integerity a původu bylo zavedeno povinné podepisování veškerých aplikací instalovaných do telefonů. Existuje několik způsobů, jak se s touto povinností vypořádat. Jedním z nich je použít tzv. *self-sign*, podepsání aplikace podpisem, který není certifikovaný, tudíž sice zajišťuje integritu, ale nemůže poskytnout důvěryhodnou záruku původu. Proto nemůže takto podepsaná aplikace

využívat některé funkce OS, které zajišťuje přístup k citlivým datům a nastavením přístroje.

V ostatních případech podepisujeme pomocí certifikovaného podpisu. Úroveň důvěry pak záleží na certifikačním procesu. Ještě do nedávné doby bylo možné získat tzv. *vývojářský certifikát* vázaný na konkrétní IMEI (nebo při splnění dodatečných podmínek množinu IMEI) a podepisovat vyvíjené aplikace offline. Bohužel tato možnost již neexistuje a byla nahrazena online formulářem[2], jehož vyplnění je obtěžující a zdlouhavé. V současné době je to však jediná možnost, jak zajistit své aplikaci přístup k chráněným API bez vlastnictví certifikovaného podpisu (např. VeriSign).

Pokud má být aplikace oficiálně uvedena na trh, je třeba, aby obdržela logo *Symbian Signed*, které dokazuje, že aplikace byla nezávisle testována a podepsána, tím je zajištěna její integrita i původ. Je zřejmé, že tato zcela oprávněná bezpečnostní opatření namířena proti potenciálním tvůrcům škodlivého kódu bohužel omezí i legitimní vývojáře a komplikují vývoj zejména freewarových aplikací. Takto restriktivní opatření jsou ale motivována velkými riziky plynoucími z používání těchto zařízení. Kromě zmiňovaného zneužití citlivých informací majitele je potřeba uvést i nebezpečí zneužití placených služeb (poplatky za telefonní spojení, mobilní internet). Příkladem by mohla být aplikace, která získá citlivá data uložená v mobilním telefonu a odešle je útočníkovi. Majitel kromě zneužití svých dat navíc zaplatí jejich přenos.

Bezpečnostní model operačního systému Symbian má několik komponent, uvedeme tři nejdůležitější, *systém oprávnění*, *ochrana dat* a *bezpečnostní instalátor softwaru*.

Systém oprávnění umožňuje jemné nastavení bezpečnostních politik pro jednotlivé aplikace. Při instalaci jsou aplikaci udělena požadovaná oprávnění (pokud to dovoluje úroveň důvěryhodnosti aplikace). Ve skutečnosti jde o oprávnění používat specifická API. Po instalaci již není možné je modifikovat. Kontrola přidělených oprávnění se řeší na úrovni jádra.

Důležitou poznámkou je, že knihovny běžící v rámci aplikace dědí oprávnění od své rodičovské aplikace, proto se požaduje, aby oprávnění poskytnutá knihovně byla při nejmenším stejná jako oprávnění jejich rodiče, jinak je aplikace ukončena s chybou `KErrPermissionDenied`.

Asi 60 % všech API nevyžaduje žádné oprávnění, zbývající část je možné použít pouze s nějakým oprávněním.[3] Ta se dělí do tří skupin. První skupinu tvoří oprávnění, která je možné udělit i aplikaci, která není podepsaná certifikovaným podpisem (self-sign). V anglické literatuře bývá tato skupina označována *user-grantable capabilities*, protože při instalaci je uživatel dotázán, jestli se má umožnit instalované aplikaci přístup k dané funkcionalitě.

Druhou skupinu tvoří oprávnění, která se udělují pouze aplikacím s důvěryhodným podpisem. Týkají se ochrany souborového systému, síťových nastavení a multimédií.

V poslední skupině jsou oprávnění, která je možné získat pouze se souhlasem výrobce. Jedná se o velmi citlivé oblasti (`DRM`, `AllFiles` a `TCB`), ke kterým většina aplikací nepotřebuje mít přístup.

Další komponentou bezpečnostního modelu je ochrana dat. Aplikace bez explicitního oprávnění (AllFiles) nemá přístup k některým částem souborového systému. To zajišťuje oddělení privátních dat jednotlivých aplikací i systému.

[2] `https://www.symbiansigned.com/app/page/public/openSignedOnline.do`

[3] Jsou uvedena v dokumentaci. Požadavky se mohou lišit i na úrovni jednotlivých metod třídy.

Složka `\sys\bin` obsahuje všechny spustitelné soubory, je jedinou složkou, odkud je možné aplikaci spustit. Pouze bezpečnostní instalátor a aplikace s oprávněním AllFiles do ní mohou zapisovat.

Složka `\private` obsahuje podsložky pro všechny instalované aplikace. Každá aplikace má svoje unikátní *SID* (*Secure ID*), které tvoří jméno podsložky a je podle něj za běhu kontrolován přístup k datům v nich obsažených. Aplikace má přístup pouze do složky označené svým SID.[4]

Složka `\resource\apps` je určena pro ukládání zdrojů jednotlivých aplikací. Jedná se o ikony, menu, bitmapy nebo lokalizační řetězce a další zdroje potřebné pro běh aplikace. Čtení je povoleno všem aplikacím, zápis pouze instalátoru (nebo aplikacím s oprávněním AllFiles).

Poslední komponentou bezpečnostního modelu operačního systému Symbian, kterou zde zmíníme, je instalátor softwaru. Ten nabízí bezpečný způsob instalace aplikací, na základě podpisu ověří důvěryhodnost aplikace a přidělí jí požadovaná oprávnění nebo její instalaci zamítne. Certifikáty, které slouží jako kořenové autority, jsou do zařízení instalovány výrobcem, uživateli se neumožňuje přidávat další.

Instalátor má oprávnění TCB (*Trusted Computing Base*) a má možnost modifikovat veškerá data a nastavení zařízení (adresář `sys`). Oprávnění, která instalátor aplikaci přidělí, jsou uložena přímo ve spustitelném souboru aplikace a díky ochraně dat není možné je později změnit.

Aplikace mohou být instalovány i na paměťové karty, pak se lokálně ukládá hash binárního souboru, aby se zabránilo následné modifikaci.

2.1.5 Kryptografická knihovna

Kryptografická knihovna není součástí standardního API operačního systému Symbian. Částečně byla zveřejněna až poměrně nedávno, bohužel dokumentace je velmi skromná, v podstatě jde pouze o hlavičkové soubory s velmi stručnými komentáři. Podrobnější návod je v dokumentaci pouze k hashovacím funkcím a PRNG. O použití „silné kryptografie“ dokumentace mlčí. Seznam zveřejněných funkcí je poměrně obsáhlý, hashovací funkce (MD5, HMAC, SHA1), z oblasti šifrování jsou to DES, 3DES, AES, RC2, RC4, RSA, Diffie-Hellman, šifrování na základě hesla, padding. Dále podpisové schéma DSA, modulární aritmetika s libovolně dlouhými čísly a generátor pseudonáhodných čísel.

Bohužel se ukazuje[5], že některé části knihovny nejsou zcela v souladu s definovanými standardy a jejich použití v případě požadavku na interoperabilitu je diskutabilní.

Dalším problémem je, že části knihovny nejsou na některých zařízeních vůbec implementovány. Jedná se např. o modulární aritmetiku, na mobilním telefonu Nokia N73 bez problémů funguje, na Nokii N82 lze třídu `RInteger` využívat pouze k ukládání čísel, aritmetické operace končí chybou.

Vzhledem k zaměření práce nás zajímal zejména generátor pseudonáhodných čísel. Ačkoli je dokumentace velmi stručná, vyplývá z ní, že použitý PRNG je kryptograficky bezpečný. Podle popisu má jít o *RANROT* algoritmus, který ale evidentně nemůže být považován za kryptograficky bezpečný (viz např. [Fog]), proto je na místě nedůvěra a

[4] Opět s výjimkou aplikací s oprávněním AllFiles.

[5] Například je popsán problém s RSA při použití PKCS#1 paddingu, který není zcela korektně implementován (nepřipojí se hlavička). Viz `http://jonmccune.wordpress.com/2008/06/05/symbian-os-v91-c-cryptography-apis-available/`.

pokud jde o kritickou aplikaci, zřejmě bychom jeho použití nedoporučili. V této práci popíšeme PRNG, který by náročné požadavky na kryptograficky bezpečný algoritmus splňovat měl.

Nyní ještě uvedeme testy rychlostí kryptografických funkcí použitých v našich implementacích (SHA1, SHA256, AES). V případě SHA1 a AES se jednalo o nativní implementaci z kryptografické knihovny operačního systému. SHA256 [Kaw] je referenční implementace upravená pro Symbian, autorem je Vinay Kawade.

V tabulce 2.1 uvádíme průměrný výkon těchto implementací. V případě hashovacích funkcí se pracovalo s 5MB souborem náhodných dat zpracovávaným po $0,5$MB částech. Pomocí AES byl zašifrován 1 MB dat. Výsledky byly zajímavé. Neočekávali jsme tak velký rozdíl ve výkonnosti hashovacích funkcí oproti AES.

Zařízení	**SHA1**	**SHA256**	**AES (256b klíč)**
Nokia N73	1799 kB/s	2667 kB/s	345 kB/s

Tabulka 2.1: Výsledky testů

Kapitola 3

Generátor pseudonáhodných čísel ANSI X9.31

Specifikace tohoto generátoru pseudonáhodných čísel se objevuje v příloze A.2.4 [Kel05] standardu ANSI X9.31-1998 [ANS93]. Protože generátor nemá definováno žádné jméno bývá označován podle dokumentu, ve kterém byl publikován, jako *X9.31 PRNG* nebo jednoduše *ANSI PRNG*.

ANSI generátor je velmi jednoduchý algoritmus, který užívá pouze jedné kryptografické funkce. Ve standardu [ANS93] je uvedena pouze možnost použití blokových šifer 3DES nebo AES, jinde se objevují i řešení postavená na hashovacích funkcích.

Nejpodstatnější rozdíl mezi dvěma standardizovanými variantami je ve velikosti parametrů algoritmu, který vzniká odlišnými délkami šifrovaných bloků. Zatímco 3DES pracuje se 64bitovými bloky, AES šifruje 128bitové bloky.

V naší implementaci jsme se rozhodli použít druhou variantu (s AES), proto popíšeme podrobněji tuto část, podrobnosti k algoritmu s 3DES je možné najít v [Kel05].

3.1 X9.31 PRNG s AES

Nechť K je 128bitový AES klíč, který se používá výlučně pro generování pseudonáhodných dat. Označme $E_K(X)$ aplikaci šifrovací funkce AES s klíčem K na 128bitový blok X. Nechť V je 128b semínko, DT 128b časová známka, I je mezivýsledek a XOR označíme operaci exkluzivní disjunkce. Pak je pseudonáhodné číslo R generováno takto:

$$I = E_K(DT)$$

$$R = E_K(I \text{ XOR } V)$$

Semínko V se aktualizuje:

$$V = E_K(R \text{ XOR } I)$$

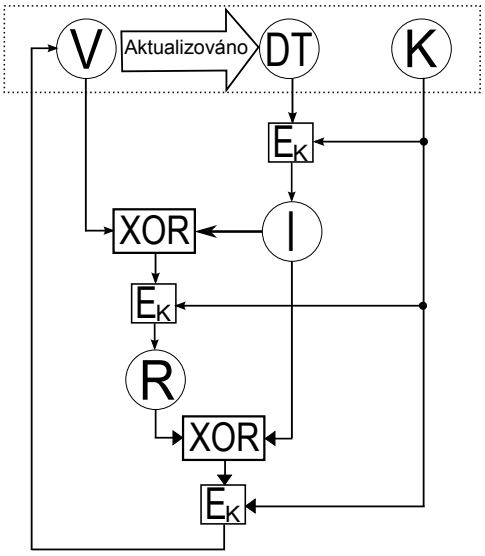
Obrázek 3.1 představuje fungování celého algoritmu.

3.2 Implementace v Symbian C++

Generátor pseudonáhodných čísel X9.31 byl implementován v jednoduché aplikaci s grafickým uživatelským rozhraním. Vzhledem k zaměření práce se strukturou aplikace nebudeme podrobněji zabývat. Zaměříme se na implementaci výkonné části aplikace, na generátor samotný.

Díky jednoduchosti a malému rozsahu je ANSI generátor implementován v jediné třídě, `CX931_AES`.

V implementaci se používají dvě kryptografické funkce, AES pro samotné generování a SHA1 pro operace se zásobníkem entropie. `CX931_AES` je potomkem třídy



Obrázek 3.1: Algoritmus X9.31 PRNG

`CCoeControl`, aby mohla zachytávat události generované klávesnicí. Navíc implementuje dvě rozhraní `MCameraObserver` a `MMdaAudioInputStreamCallback`, pro obsluhu kamery a mikrofonu.

V atributech třídy se ukládají informace o stavu generátoru, AES klíč, semínko a posledně použitá časová známka. Pro inicializaci generátoru se používá zásobník entropie, který sbírá náhodná data prostředí. Konkrétně se jedná o přesné časování stisku kláves před prvním požadavkem na generování, pět vzorků z hledáčku kamery a dvacet vzorků z mikrofonu. Zásobník má velikost 320 bitů a je implementován pomocí dvou 160bitových hashovacích kontextů [1] (SHA1).

Při prvním volání metod `GetRandom()` nebo `GetRandomFileL()` je generátor inicializován ze zásobníku entropie. Prvních 128 bitů se stane AES klíčem K, dalších 128 b naplní semínko V a zbývajích 64 bitů se použije pro doplnění časové známky DT. To je nutné udělat z toho důvodu, že čas je v prostředí operačního systému Symbian rozlišován pouze na 64 bitů. DT musí zachovat velikost šifrovacího bloku, proto je doplněn při počáteční inicializaci fixními 64 bity náhodných dat ze zásobníku entropie, druhá polovina se aktualizuje pokaždé.

Dále se postupuje přesně podle algoritmu na obrázku 3.1. DT se zašifruje a xoruje se semínkem V, vzniká mezivýsledek I, který je po zašifrování výstupem generátoru, tj. 128 bitů pseudonáhodných dat, a zároveň ovlivňuje novou hodnotu semínka.

$$V_{i+1} = E_K(\underbrace{E_K(V_i \text{ XOR } E_K(DT_i))}_{R_{i+1}} \text{ XOR } E_K(DT_i)) \tag{3.1}$$

[1] Přicházející data jsou do nich ukládána střídavě.

3.3 Bezpečnost

V prezentované podobě není možné generátor považovat za kryptograficky bezpečný. Není splněna ani podmínka dopředné, ani zpětné bezpečnosti.

Vzhledem k tomu, že šifrovací klíč určený pro generování se nikdy nemění, po kompromitaci vnitřního stavu generátoru je útočník schopen získat dříve vygenerovaná data. Z rovnice 3.1 vidíme, že pokud známe vnitřní stav generátoru (K, V_{i+1}, DT_{i+1}), prostými úpravami rovnice dostaneme R_{i+1} a V_i. Ke vzdálenějším hodnotám už není tak jednoduché se dostat, jejich bezpečnost závisí na bezpečnosti časové známky. Budeme ji diskutovat v následujícím odstavci, protože úzce souvisí i se zpětnou bezpečností.

Zpětná bezpečnost závisí pouze na časové známce, protože ta tvoří jediná data, která jsou přidávána do stavu generátoru a obsahují nějakou entropii. Bohužel jí není mnoho. Již jsme zmínili, že velikost DT je v podstatě redukována na 64 bitů (fixací první poloviny daty ze zásobníku entropie při počáteční inicializaci). Pokud předpokládáme, že útočník zná vnitřní stav generátoru, polovinu DT zná. Pro předpovídání budoucích výstupů generátoru mu proto stačí odhadovat časovou známku požadavků.

Při optimistickém předpokladu můžeme čekat, že útočník odhadne DT s přesností v řádu minut. Teoreticky je přesnost vnitřních hodin operačního systému v řádu mikrosekund, ale naše analýza ukázala, že časová známka je vždy dělitelná 125. To redukuje teoretický počet možností, které by musel útočník vyzkoušet na 480 000, což představuje přibližně 18,87 bitů entropie.

Pokud si představíme sofistikovanějšího útočníka, který odhadne DT s přesností na vteřinu, redukuje se entropie obsažená v 128bitové časové známce na 12,96 bitů (8000 možností).

Další špatnou vlastností je, že se generátor z kompromitovaného stavu nikdy nezotaví, protože není pravidelně reinicializován.

3.4 Výkon implementace

V případě ANSI generátoru byla hlavním výkonostním parametrem rychlost generování. Při testech jsme postupovali tak, že bylo generováno sto 1kB pseudonáhodných sekvencí. Testování proběhlo na dvou mobilních telefonech, Nokia N73 a N82. Výsledné rychlosti generování jsou v tabulce 3.1.

Zařízení	Min (kB/s)	Max (kB/s)	Průměr (kB/s)	Medián (kB/s)
Nokia N73	250	333	326	333
Nokia N82	250	333	264	250

Tabulka 3.1: Test rychlosti generování pro ANSI X9.31 PRNG

3.5 Shrnutí

Z výše uvedené neformální analýzy jasně vyplývá, že použití naší implementace ANSI X9.31 PRNG na místě, kde je vyžadován kryptograficky bezpečný generátor pseudonáhodných čísel, nemůžeme doporučit.

To ostatně ani nebylo původním úmyslem. Implementace tohoto jednoduchého generátoru měla sloužit pro seznámení se s programováním v Symbian C++ a prozkoumání možností prostředí s ohledem na nalezení potenciálních zdrojů náhodnosti. Některé části implementace jsou přímo použitelné v dalších aplikacích (aplikační rámec, obsluha kamery a mikrofonu, užití kryptografických funkcí). Zdrojový kód aplikace je na přiloženém CD a na internetových stránkách `http://sourceforge.net/projects/ansix931prng/`.

V průběhu implementace získali mnohé praktické zkušenosti z reálného prostředí vývoje aplikací pro mobilní telefony. Je třeba si uvědomit všechna omezení, kterých není na této platformě málo,např. omezené prostředky zařízení (CPU, paměť, baterie). Navíc primární určení mobilního telefonu je poněkud jiné než generování pseudonáhodných sekvencí, je tedy potřeba zajistit co nejmenší omezení uživatele.

Závěrem musíme zdůraznit, že implementace aplikace s přístupem k nižším úrovním operačního systému a zařízení je poměrně náročným úkolem, který je navíc komplikován nedostatkem dokumentace a omezeními výrobců zařízení.

V další kapitole se budeme zabývat pokročilejším generátorem, který oproti ANSI PRNG nabízí dostatečné záruky bezpečnosti, ať už ve fázi návrhu nebo implementace.

Kapitola 4

Fortuna PRNG

Zatímco dříve popsaný X9.31 PRNG nenabízí v prezentované verzi příliš velkou bezpečnost, generátor pseudonáhodných čísel Fortuna je po této stránce mnohem dále. Autory jsou známí kryptografové Bruce Schneier a Neils Ferguson. Fortuna navíc není jejich prvním generátorem. Ve skutečnosti je Fortuna PRNG modifikací jejich předchozího generátoru Yarrow [KSF99]. Hlavním vylepšením je práce se zdroji entropie a její ukládání do zásobníků entropie. Architekturu generátoru a doporučení k implementaci autoři shrnuli v knize *Practical Cryptography* [FS03]. Autoři bohužel v publikaci explicitně neuvádějí důkaz kryptografické bezpečnosti generátoru, není nám znám ani jiný zdroj, který by jej zmiňoval. Přesto je ve všech případech Fortuna PRNG uváděn jako příklad CSPRNG (viz např. [MCCM06]).

Nejdůležitějším problémem předchozího návrhu (Yarrow PRNG) byla manipulace se zdroji entropie. Bylo totiž nutné odhadovat (nejlépe za běhu) množství entropie dodané jednotlivými zdroji. To je samozřejmě už principiálně velmi obtížné a prakticky to znamenalo velké potíže s analýzou bezpečnosti výsledné implementace.

Proto autoři vylepšili svůj předchozí koncept a Fortuna už díky odlišné manipulaci se zásobníkem entropie nepotřebuje entropii zdrojů odhadovat (resp. není tolik závislá na přesnosti odhadů). Detaily popíšeme dále.

V době psaní této práce jsou nám známy dvě praktické implementace. První je použití Fortuny v linuxovém jádře (vyšlo jako patch jádra [TMC]) a druhou je aplikace pro Windows napsaná v C++ [Bon].

4.1 Architektura generátoru pseudonáhodných čísel Fortuna

Fortuna PRNG se skládá ze tří komponent. První komponentou je *generátor* samotný. Je to jednoduchý algoritmus, který ze semínka pevné délky vygeneruje libovolně dlouhou pseudonáhodnou sekvenci. Generátor je možné použít nezávisle na ostatních komponentách v případě, že nepotřebujeme kryptograficky bezpečná data (např. pro oblast simulací).

Druhou komponentou je *akumulátor*, který se stará o manipulaci se zdroji náhodnosti a ukládání dat do zásobníků entropie. Je také zodpovědný za pravidelnou reinicializaci (*reseeding*) generátoru.

Poslední komponentou je *inicializační soubor* (*seed file*), do kterého se ukládají data používaná pro počáteční inicializaci generátoru po restartu zařízení, kdy ještě není nasbíráno dostatek entropie pro korektní inicializaci.

Nyní popíšeme jednotlivé komponenty podrobněji.

4.1.1 Generátor

Generátor je velmi jednoduchou částí celého PRNG. V podstatě jde pouze o blokovou šifru v counter módu. Autoři doporučují použití například některého z kandidátů AES (Rijndael, Serpent, Twofish, ...) s délkou bloku 128 bitů a 256bitovým klíčem. Tajný vnitřní stav generátoru tvoří dvě položky, 256b klíč a 128b počítadlo.

Po každém požadavku na generování pseudonáhodných dat se vygenerují dva další bloky, které se použijí jako nový klíč. Tím se zajistí dopředná bezpečnost, protože po zahození předchozího klíče není možné získat dříve vygenerovaná data, i když by útočník znal vnitřní stav generátoru. Zpětná bezpečnost se řeší pomocí akumulátoru.

Jiným problémem týkajícím se statistických vlastností vygenerovaných bloků je, že výstup šifrovací funkce v counter módu je principiálně unikátní. To znamená, že při monotónně rostoucím počítadle a konstantním klíči se žádný blok nebude opakovat. To samozřejmě ze statistického pohledu není správně, protože v náhodné sekvenci tato situace nastat může. Autoři proto doporučují kompromisní řešení (viz [FS03, str. 162]), omezit množství vygenerovaných dat v rámci jednoho požadavku (tj. beze změny klíče) na 1 MB, kde se popsaný statistický defekt příliš neprojeví.

Dále autoři vysvětlují, proč není vhodné po každém požadavku spolu s klíčem resetovat i počítadlo. Toto opatření by mělo zabránit vzniku krátkých cyklů, které by se objevily při náhodném opakování klíče. Počítadlo o velikosti 128 bitů je dostatečně velké na to, aby nepřeteklo.[1]

Aplikační rozhraní generátoru je jednoduché a sestává pouze ze čtyř funkcí. Zmíníme je pouze krátce, podrobnější popis a pseudokód je možné najít v [FS03, str. 164–167].

První funkce nazvaná `InitializeGenerator` pouze vynuluje klíč a počítadlo, tím je generátor uveden do stavu, ve kterém není možné generovat data a čeká se na první volání funkce `Reseed`, která na základě svého argumentu (semínko – seed) nastaví klíč a inkrementuje počítadlo.

Od tohoto okamžiku je možné volat funkci `PseudoRandomData`, která voláním `GenerateBlocks` zajistí vygenerování požadovaného množství dat (až 1 MB). Každý blok je výsledkem operace šifrování inkrementovaného počítadla. Funkce `PseudoRandomData` nakonec vygeneruje dva bloky navíc a aktualizuje klíč.

4.1.2 Akumulátor

Druhou komponentou Fortuna PRNG je akumulátor. Jeho úkolem je sbírat data z různých zdrojů náhodnosti do zásobníků entropie a pravidelně reinicializovat generátor (volání `Reseed`).

Autoři doporučují, že by zdrojů náhodnosti mělo být co nejvíce. Není nutné zařadit pouze „dobré“. Mechanismus reinicializace počítá s tím, že jistá část zdrojů bude dodávat nenáhodná data (ať už mluvíme o statistických vlastnostech nebo o predikovatelnosti útočníkem). Pokud existuje alespoň jeden zdroj generující náhodné události, které útočník nemůže ovlivnit nebo předpovídat, generátor funguje správně. Tuto vlastnost umožňují splnit zásobníky entropie a reinicializační mechanismus.

Akumulátor obsahuje 32 zásobníků entropie, do kterých se rovnoměrně ukládají data dodaná zdroji náhodnosti. Obsluha jednotlivých zásobníků probíhá cyklicky. Pro každý

[1] Vygenerování 2^{128} bloků pseudonáhodných dat je daleko za výpočetními možnostmi a životností těchto zařízení.

zdroj existuje cyklický index určující, do kterého zásobníku mají data přijít. To zaručuje, že pokud existuje alespoň jeden nekompromitovaný zdroj, postupně „kontaminuje“ všechny zásobníky a útočníkova snaha vyjde na prázdno.

Reinicializační mechanismus je velmi komplexní. Jednotlivé reinicializace jsou očíslovány (1, 2, 3, . . .) a definuje se pravidlo, že zásobník P_i se použije právě tehdy, když 2^i dělí počítadlo reinicializací. To znamená, že zásobník P_0 se použije vždy, P_1 každou druhou reinicializaci, P_2 každou čtvrtou, atd. Výsledný řetězec (použitý v `Reseed`) vzniká zřetězením obsahu použitých zásobníků.

Díky rovnoměrnému rozložení dat ze zdrojů náhodnosti na zásobníky by obsahovaly jednotlivé zásobníky přibližně stejné množství entropie. Reinicializační mechanismus způsobuje, že např. P_1 (používaný dvakrát častěji než P_2) obsahuje přibližně dvakrát méně entropie než P_2. Pokud by tedy útočník získal dostatek informací, že by odhadl obsah P_0, měl by možnost (pokud se bude provádět `Reseed` pouze z něj) předvídat budoucí výstup generátoru. Samozřejmě pouze do další reinicializace, kdy se použije i P_1. V něm je dvakrát více entropie než v P_0 a to je pro útočníka buď nemožné uhádnout, nebo se mu to podaří a do další reinicializace zná generovaná data předem. Je však jen otázkou času, kdy mu dojdou prostředky a některý ze zásobníků bude obsahovat více entropie, než je útočník schopen překonat (množství entropie se zvyšuje exponenciálně). Za jak dlouho se PRNG zotaví z kompromitace, záleží na množství dat přitékajících do akumulátoru ze zdrojů náhodnosti (těch neovlivňovaných útočníkem).

Schneier a Ferguson v [FS03, str. 170] odhadují množství entropie potřebné pro zotavení z kompromitace na 2^{13} bitů v nejhorším případě. Nebezpečná situace by nastala, pokud by útočníkem nekontrolované zdroje nedodaly dostatek entropie v průběhu 2^{32} volání `Reseed`. Pak by dokonce i zásobník P_{31} neobsahoval dostatek entropie, aby znemožnil předvídat útočníkovi budoucí výstup. To by se mohlo stát tehdy, když by byl útočník schopen do zásobníků vložit obrovské množství vlastních „náhodných událostí“. Proto autoři navrhují omezit frekvenci reinicializací na 10 za vteřinu, pak by se P_{31} použil poprvé po 13 letech, což činí takový útok velmi nepraktickým.

Akumulátor přidává do aplikačního rozhraní Fortuna PRNG tři funkce. První z nich je `InitializePRNG`, která vynuluje počítadlo reinicializací a zavolá `InitializeGenerator`.

Druhou funkcí, z hlediska koncového uživatele nejdůležitější, je `RandomData`, která řídí proces generování pseudonáhodných dat a související reinicializace.

Na začátku se ověří, jestli není možné generátor reinicializovat. Podmínkami jsou čas od poslední reinicializace (min. 100 ms) a dostatek entropie v prvním zásobníku (tj. minimálně 256 bitů v P_0). Z toho vidíme, že alespoň částečně se musíme odhady entropie dat získaných ze zdrojů náhodnosti přece jen zabývat. Autoři Fortuny však uvádí, že odhady mohou být velmi optimistické, bez zásadního vlivu na bezpečnost implementace.

Eventuální reinicializace probíhá podle výše uvedeného pravidla.[2] Poté se volá `PseudoRandomData` generátoru, zajišťující vlastní generování.

Poslední funkcí, kterou akumulátor nabízí, je `AddRandomEvent`, kterou volají zdroje náhodnosti, aby přidali entropii do akumulátoru. Každý zdroj je identifikován unikátním číslem, podle něhož probíhá distribuce dat do zásobníků (každý zdroj je obsluhován cyklicky zvlášť).

[2] Pro připomenutí, zásobník P_i se použije právě tehdy, když 2^i dělí počítadlo reinicializací.

4.1.3 Inicializační soubor

Poslední komponentou je inicializační soubor, který se používá po restartu aplikace (resp. zařízení) ke korektní inicializaci generátoru. Ve skutečnosti jde o permanentní zásobník entropie nasbírané v předchozích bězích. Ačkoliv teoreticky jde o velmi jednoduchý koncept, praktická implementace není snadná a závisí na prostředí, ve kterém generátor poběží (operační systém, souborový systém).

Je zde několik striktních požadavků, při jejichž nesplnění má útočník poměrně dobré předpoklady získat alespoň část generovaných dat.

K inicializačnímu souboru by měla mít přístup pouze aplikace implementující generátor, jinak útočník zná počáteční stav generátoru a může předvídat výstup minimálně do první reinicializace, pokud navíc kontroluje některé zdroje náhodnosti, je předpoklad, že po jistou dobu bude schopen odhadovat obsah zásobníků entropie, které po restartu pravděpodobně nebudou obsahovat mnoho entropie.

Dále je třeba po použití obsahu reinicializačního souboru vynutit okamžitou aktualizaci obsahu reinicializačního souboru, ještě před tím, než může kdokoliv žádat generování dat. Problém vzniká v případě bufferovaných operací zápisu (na úrovni OS nebo disku), u kterých si nejsme nikdy jistí, jestli už zápis skutečně proběhl.

Ze stejného důvodu jako v předchozím případě (nedopustit, aby byl PRNG dvakrát ve stejném počátečním stavu), je třeba zamezit zálohování reinicializačního souboru. Navíc bývají zálohy méně chráněny než běžící souborový systém.

Obecně velmi těžko řešitelným problémem je situace, když se PRNG používá poprvé. To znamená, že neexistuje žádný reinicializační soubor, navíc je velmi pravděpodobné, že hned po instalaci uživatel bude generovat vysoce citlivá data, např. kryptografické klíče.

Z výše uvedeného je zřejmé, že bezpečná implementace reinicializačního souboru je náročným úkolem, na některých platformách je bohužel vždy do jisté míry kompromisem.

4.2 Implementace v OS Symbian

V této části popíšeme implementaci generátoru pseudonáhodných čísel Fortuna v prostředí operačního systému Symbian 9.x Series60 na mobilních telefonech. Hned na začátku je nutné upozornit, že vývoj probíhal na přístroji Nokia N73. Výsledná aplikace byla sice úspěšně testována i na dalších modelech firmy Nokia (N80, N82, E51), ale vzhledem k problémům s kompatibilitou z toho bohužel nelze usuzovat, že aplikace bude funkční i na jiných zařízeních.

Generátor Fortuna je navržen tak, aby běžel kontinuálně, sbíral entropii z prostředí a obsluhoval požadavky uživatelů na pseudonáhodná data. Proto byl použit *Klient-Server Framework* ze Symbian API, ten umožňuje komunikaci mezi procesy. PRNG tvoří serverovou část, ke které se klientské procesy mohou připojit a požadovat vygenerování dat. Server je implementován v aplikaci bez grafického rozhraní, klientské aplikace si přilinkováním dll knihovny zpřístupní exportované funkce.

Pro účely vývoje a testování byly vytvořeny dvě další grafické aplikace. První slouží ke spuštění, resp. zastavení Fortuna serveru. Měla by být používána pouze v testovacím prostředí, pro reálné nasazení je vhodnější spouštět server během startu zařízení a

zakázat předčasné ukončení.

Druhá aplikace slouží jako ukázka klientské aplikace, která se připojí k běžícímu Fortuna serveru a požádá o vygenerování zadaného množství pseudonáhodných dat, která uloží do souboru.

Tyto aplikace nebudeme dále práci popisovat, jsou dostatečně jednoduché, aby bylo jejich fungování pochopitelné z přiloženého zdrojového kódu.

4.2.1 Fortuna server

Jak už jsme uvedli dříve, Fortuna server je nejdůležitější částí celé aplikace, protože implementuje veškerou funkcionalitu. Server běží jako samostatný proces bez grafického rozhraní na pozadí, sbírá entropii do zásobníků a obsluhuje případné klientské požadavky na generování dat.

Implementace je poměrně rozsáhlá, protože pokrývá všechny tři komponenty Fortuna PRNG (generátor, akumulátor a inicializační soubor), navíc obsluhuje zdroje náhodnosti.

Každý server v Symbianu se skládá ze dvou základních tříd (v našem případě nazvané `CFServer` a `CFSession`), které jsou odvozeny od třídy `CServer2`, resp. `CSession2`. První zajišťuje fungování serveru vůči OS, druhá představuje serverovou stranu jednotlivých spojení (session) s klienty.

`CFSession`

Objekt této třídy je vytvářen operačním systémem v souvislosti se vznikem nové session, hlavním úkolem je parsování klientských požadavků zaslaných pomocí IPC (*Interprocess Communication*) a jejich předávání serveru.

Téměř veškerá funckcionalita se dědí z rodičovské třídy `CFSession`, pouze dvě metody jsou přepsány. Je to `ServiceL()`, která parsuje zprávu z argumentu. Pokud je požadavek korektní, předá jej serveru, jinak je klient násilně ukončen.

Pro obsluhu výjimek vzniklých v metodě `ServiceL` slouží `ServiceError()`.

`CFServer`

Tato třída je mnohem obsáhlejší, protože zahrnuje funkcionalitu celého PRNG, včetně aktivních objektů obsluhujících zdroje náhodnosti.

Vstupním bodem procesu je globální funkce `E32Main()`, kde se vytváří cleanup stack, plánovač aktivních objektů[3], konstruuje se objekt třídy `CFServer` a proces čeká na požadavky potenciálních klientů.

Třída obsahuje několik atributů, jeden představuje samotný PRNG, dále jsou zde čtyři implementované zdroje náhodnosti a obsluha inicializačního souboru.

`CFortunaPRNG` Třída `CFortunaPRNG` implementuje všechny tři komponenty Fortuna PRNG, generátor (`CFGenerator`), akumulátor (`TFPool`) a inicializační soubor (`RFile`).

[3] V běžných GUI aplikacích jsou tyto vytvářeny automaticky v aplikačním frameworku.

Třída `CFGenerator` je přímočarou implementací generátoru popsaného na straně 20. Obsahuje 256bitový AES klíč, 128bitové počítadlo pro generování a objekty představující šifrovací kontext.

První funkce popsaná v návrhu (`InitializeGenerator`) nebyla implementována jako jedna z metod, ale daná funkcionalita byla zahrnuta přímo v konstruktoru objektu. Ostatní funkce mají své protějšky ve stejnojmenných metodách.

Metoda `Reseed()` zřetězí svůj argument se stávajícím klíčem, výsledný řetězec se pak po průchodu funkcí SHA256 stane novým klíčem. Nakonec se inkrementuje počítadlo.[4]

Metoda `GenerateBlocks(TInt k, TDes8 &r)` nejdříve zkontroluje, jestli už je generátor inicializován, pak vygeneruje k bloků pseudonáhodných dat, které uloží do deskriptoru r. Každý blok je dlouhý 128 bitů a vznikne zašifrováním aktuálního stavu počítadla, které je následně inkrementováno. Tato metoda je privátní, klient ji volá prostřednictvím metody `PseudoRandomData(TInt n, TDes8 &r)`.

Ta navíc kontroluje, není-li požadované množství dat (n bajtů) větší než 1 MB. Pak volá `GenerateBlocks(`$\lceil n/16 \rceil$`, data)` a ořeže vrácená data na požadovanou velikost. Poté se vygenerují ještě další dva bloky sloužící pro aktualizaci generovacího klíče.

Dále jsou zde dvě pomocné metody. `IsInitialized()` kontroluje, je-li generátor inicializován (tj. má nenulové počítadlo). Protože je počítadlo implementováno pomocí 128bitového binárního řetězce, jde o porovnání s nulou po bajtech. Metoda `Inc()` inkrementuje počítadlo.

Akumulátor je implementován jako pole 32 objektů třídy `TFPool`. Každý objekt reprezentuje jeden zásobník entropie, který se plní daty ze zdrojů náhodnosti. Ačkoli se v návrhu objevují zásobníky jako neomezené řetězce, z praktických důvodů jsou implementovány pomocí hashovacích kontextů. Proto se musí pozměnit i způsob odhadu entropie obsažené v zásobníku. Autoři návrhu entropii odhadovali z velikosti zásobníku (tj. délky řetězce), ta je ale v našem případě stále stejná (256 b), proto se zavádí atribut zásobníku uchovávající jeho teoretickou délku (tedy jakýsi odhad množství entropie). Odhady množství entropie v datech z různých zdrojů náhodnosti jsou nastaveny po praktické analýze těchto zdrojů, v podstatě by však stačilo entropii odhadovat na základě délky dat, protože Fortuna není příliš citlivá na přesnost odhadů.

Data se do zásobníků přidávají pomocí metody `AddRandomEvent()`, kterou ale nevolají přímo zdroje náhodnosti. Důvodem je, aby nemohly ovlivnit distribuci dat do jednotlivých zásobníků entropie. Zdroje náhodnosti používají metodu `CFortunaPRNG::AddRandomEvent()`, která se postará o korektní rozdělení.

Metoda `Data()` se volá při reinicializaci generátoru metodou `Reseed()`, zásobník se při tom vyprázdní.

Komponenta inicializačního souboru není implementována v samostatné třídě, ale přímo v `CFortunaPRNG`. Ta část funkcionality, která se stará o čtení inicializačního souboru, je obsažena přímo v konstruktoru. Pokusí se otevřít soubor *seedFile.rnd* ve složce `\Private`, do které má přístup pouze tato aplikace[5]. Pokud soubor existuje a obsahuje alespoň 64 bajtů, generátor je inicializován, okamžitě vygeneruje 64 bajtů pseudonáhodných dat a aktualizuje inicializační soubor.

[4] To je nezbytně nutné udělat při prvním volání metody `Reseed()`, při dalších už ne. Ovšem pro zjednodušení inkrementujeme pokaždé.

[5] A aplikace s oprávněním AllFiles.

Pokud není soubor dostatečně dlouhý, data se nepoužijí a generátor zůstává v neinicializovaném stavu. Pokud soubor vůbec neexistuje (při prvním spuštění aplikace), vytvoří se prázdný a generátor zůstane neinicializovaný. V tomto stavu není možné obsloužit požadavky klientů, čeká se, až je možné inicializovat generátor ze zásobníků entropie.

Je dobré, aby byl inicializační soubor pravidelně aktualizován, proto se pomocí aktivního objektu `CFTimerSeedFile` každých 5 minut vygenerují čtyři bloky pseudonáhodných dat a obsah souboru se obnoví.

Nyní se vraťme k popisu třídy `CFortunaPRNG`, která propojuje všechny tři komponenty a vytváří tak kryptograficky bezpečný PRNG. Pro zajištění bezpečnosti je třeba generátor pravidelně reinicializovat. Dříve popsaný mechanismus využívá počítadla reinicializací `ReseedCnt` a výběru odpovídajících zásobníků entropie.

Z hlediska klientů Fortuna serveru je nejdůležitější metodou `RandomData()`, jejím úkolem je generování pseudonáhodných dat a zapojení reinicializačního mechanismu.

Reinicializace proběhne před začátkem generování, pokud jsou splněny následující dvě podmínky:

- V prvním zásobníku je dostatek entropie (alespoň 256 bitů).
- Od poslední reinicializace uběhlo minimálně 100 ms.

Samotná reinicializace je přesnou implementací dříve popsaného mechanismu. Zásobník P_i se použije právě tehdy, když 2^i dělí počítadlo reinicializací.

Zdroje náhodnosti V naší implementaci jsme použili čtyři zdroje náhodnosti, časovač (`CFTimerRandSrc`), časování stisku kláves (`CFKeyRandSrc`), mikrofon (`CFAudioRandSrc`) a kameru (`CFCamRandSrc`).

Zdroje náhodnosti jsou ze své podstaty asynchronní, proto jsou implementovány jako aktivní objekty. Každý zdoj má přiděleno jednoznačné ID, podle kterého jsou jeho data distribuována do zásobníků.

První zdroj náhodnosti sbírá data z časovače s mikrosekundovou přesností. Časovač generuje každou vteřinu událost a výsledná data přidávaná do akumulátoru jsou skutečným intervalem mezi zpracováními dvou po sobě jdoucích událostí.

Bohužel po analýze dat generovaných tímto zdrojem jsme se rozhodli jej nepoužít, protože data neobsahují téměř žádnou entropii. Je to dáno pravděpodobně nízkou latencí plánovače aktivních objektů a malou přesností časovače (viz diskuse u ANSI PRNG). Třída zůstala v projektu zachována (spíše pro zajímavost).

Druhý zdroj náhodnosti, který byl implementován, sbírá entropii z událostí generovaných stiskem tlačítek klávesnice, přesněji řečeno jejich časování. Komplikací z hlediska implementace je absence grafického rozhraní Fortuna serveru, což znemožňuje použití běžného mechanismu obsluhy událostí generovaných klávesnicí. Řešením je vytvoření „neviditelného“ okna (nemůže dostat fokus a není vidět mezi ostatními běžícími úlohami), které požádá *Window Server* o zasílání klávesových událostí. Poté se aktivní objekt uspí a čeká se na stisk klávesy, který se zpracuje pomocí metody `RWsSession::GetEvent()`. Ve skutečnosti nás zajímá pouze čas stisku klávesy, z něj můžeme podle odhadu (viz kap. 3.3) očekávat přibližně 13 bitů entropie.

Nyní popíšeme třetí zdroj náhodnosti sbírající entropii obsaženou v datech z mikrofonu. Je implementován v třídě `CFAudioRandSrc`.

Třída implementuje rozhraní `MMdaAudioInputStreamCallback`, to předepisuje metody, které musejí být definovány, aby mohla třída obsluhovat mikrofon. Průběh vypadá takto:

1. Nastaví se parametry zachytávání zvukových dat z mikrofonu, v našem případě 16bitové PCM se vzorkováním 8 kHz.
2. Voláním metody `CMdaAudioInputStream::Open()` se otevře spojení k mikrofonu.
3. Po dokončení této asynchronní operace se zpět volá `CFAudioRandSrc::MaiscOpenComplete()`.
4. Voláním metody `CMdaAudioInputStream::ReadL()` požádáme o data z mikrofonu.
5. O dokončení operace jsme informováni metodou `CFAudioRandSrc::MaiscBufferCopied()`, která ve svém argumentu vrátí požadované množství dat, v našem případě 1 kB.
6. Pokud požadujeme další data, pokračujeme bodem 4.

Data jsou pomocí `CFortunaPRNG::AddRandomEvent()` přidávána do akumulátoru. Podle odhadů (viz [KŠM07]) jeden bajt dat obsahuje asi 0,05 bitů entropie, tj. 51 bitů v každém bloku dat.

Množství dat, která požadujeme, a případný interval mezi požadavky je nutné velmi dobře nastavit. Vždy je to otázka kompromisu mezi bezpečností a použitelností aplikace. Musíme totiž uvážit, že nepřetržitý běh dosti zatěžuje telefon.

Posledním zdrojem náhodnosti je fotoaparát. Princip fungování třídy `CFCamRandSrc` je velmi podobný předchozímu. Třída implementuje rozhraní `MCameraObserver`, které předepisuje několik metod. Spolupráce s kamerou vypadá takto:

1. Voláme metodu `CCamera::Reserve()`.
2. Dokončení je signalizováno vyvoláním `CFCamRandSrc::ReserveComplete()`.
3. Voláme `CCamera::PowerOn()`.
4. Dokončení je signalizováno vyvoláním `CFCamRandSrc::PowerOnComplete()`.
5. Voláním `CCamera::StartViewFinderBitmapsL()` požadujeme data z hledáčku.
6. Data jsou nám kontinuálně doručována jako argument metody `CFCamRandSrc::ViewFinderFrameReady()`.

Data z hledáčku používáme proto, že jsou nejméně zatížena jakýmkoliv zpracováním (ať už hardwarovým nebo softwarovým). Bitmapa má rozměr 212×160 pixelů, každý je reprezentován třemi bajty. To představuje více než 100 kB dat, proto jsme se rozhodli data rozdělit na 32 částí a zapsat tak do každého zásobníku entropie 3180 bajtů, což podle [KŠM07, Tab. 1] znamená 11 080 bitů entropie pro každý zásobník. Je to efektivnější než uložit obrovské množství dat do jednoho zásobníku.

Opět musíme zmínit, že je potřeba pečlivě vyladit poměr mezi výkonem a bezpečností. U fotoaparátu navíc vzniká problém, který u mikrofonu nenastával. Zde je totiž přístup exkluzivní, pokud používáme kameru jako zdroj náhodnosti, není v tuto dobu dostupná uživateli telefonu. Navíc kontinuální běh velmi zatěžuje baterii a pravděpodobně i hardware fotoaparátu.

4.2.2 Fortuna klient

Klientská část Fortuna PRNG vytváří rozhraní pro používání generátoru. Je implementována v třídě `RFSession` v dll knihovně, exportují se čtyři funkce.

Dvě slouží k ovládání běhu Fortuna serveru (`Start()` a `StopServer()`). V reálném nasazení by asi neměly být tyto funkce exportovány. Server by se spouštěl při zapnutí telefonu a nemělo by být možné jej běžným způsobem zastavit.

Pomocí metody `Connect()` se klient připojí k Fortuna serveru a vytvoří novou session. Následně může požádat server o generování pseudonáhodných dat (metoda `GetRandomData()`).

4.2.3 Výkon implementace

Aby byl PRNG prakticky použitelný, musí být přiměřeně výkonný. V případě Fortuny nás zejména zajímalo, kolik entropie přitéká do akumulátoru a jaká je výstupní rychlost generování.

Výkon zdrojů náhodnosti závisí hlavně na nastavení objektů obsluhujících mikrofon a kameru. Třetí zdroj, klávesnice, je (jak bylo uvedeno dříve) poměrně pomalý. Pouze 13 bitů entropie na jeden stisk tlačítka spolu s malou frekvencí událostí činí tento zdroj pouze doplňkovým.

Mikrofon dodává nepoměrně více entropie (asi 100 B/s), navíc není k mikrofonu exkluzivní přístup, takže je možné jej používat kontinuálně.

Ovšem největší potenciál má jako zdroj náhodnosti kamera. Teoreticky je možné z dat kamery získat až 640 kB entropie za vteřinu. Nevýhodou je, že používáme-li kameru jako zdroj náhodnosti, žádná jiná aplikace nebo uživatel k ní nemá přístup. Kontinuální sběr dat z kamery by tedy měl být spíše vyjímečnou volbou, kromě obtěžování uživatele nedostupností kamery musíme uvažovat i pokles celkového výkonu telefonu, vybíjení baterie (u N73 tak vydrží přibližně 3 hodiny) a případné poškození hardwaru kamery.

Testy ukázaly, že při kontinuálním běhu všech zdrojů náhodnosti do zásobníků entropie přitéká asi 641 kB entropie za sekundu (hlavním faktorem je kamera).

Dále jsme testovali rychlost generování. Způsob testování byl stejný jako u ANSI generátoru, generovali jsme sto sekvencí o délce 1 kB. V tabulce 4.1 jsou výsledky testu.

Zařízení	Min (kB/s)	Max (kB/s)	Průměr (kB/s)	Medián (kB/s)
Nokia N73	400	1000	848	1000

Tabulka 4.1: Výsledky testu rychlosti Fortuna PRNG

Z výsledků je vidět, že implementace Fortuna PRNG je i při vyšší bezpečností přibližně třikrát rychlejší než ANSI PRNG (srov. 3.1).

4.3 Shrnutí

V této kapitole jsme prezentovali architekturu generátoru pseudonáhodných čísel Fortuna autorů Schneiera a Fergusona a jeho implementaci v prostředí mobilních telefonů s operačním systémem Symbian. Pro své bezpečné fungování používá Fortuna několik zdrojů náhodnosti, které umožňují zotavení z potenciální kompromitace útočníkem. Tyto zdroje (klávesnice, kamera, mikrofon) jsou hlavním faktorem ovlivňujícím výkon generátoru, jak po stránce rychlosti generování, tak celkové bezpečnosti. Vzhledem k primárnímu účelu zařízení, na kterém Fortuna běží, je třeba klást velký důraz na použitelnost aplikace a pokud možno co nejmenší zásah do fungování ostatních aplikací.

Zdrojový kód aplikace je dostupný na přiloženém CD nebo na internetových stránkách `http://sourceforge.net/projects/fortunaprng/`.

Poslední část této práce se bude věnovat využití Fortuna PRNG v protokolu pro bezpečné ustavení klíče mezi mobilními telefony s využitím Bluetooth.

Kapitola 5

Dohoda tajného klíče přes Bluetooth

Posledním úkolem práce byla implementace protokolu pro bezpečné ustavení tajného klíče mezi několika mobilními zařízeními. Jako komunikační kanál bylo zvoleno Bluetooth. V současné době je to zřejmě nejčastěji používaný způsob komunikace mezi mobilními zařízeními (samozřejmě s výjimkou běžné telefonní sítě). Výhodou je dostupnost této technologie na drtivé většině prodávaných telefonů, zabudovaná podpora operačního systému Symbian (správa zařízení, párování a autorizace) i to, že komunikace není zpoplatněna. Za nevýhodu je možné považovat omezený dosah (většinou 6–8 metrů).

Alternativou by bylo použití Wi-Fi, které však není na mobilních zařízeních zatím příliš rozšířené, nebo například SMS, které ale mají pouze omezenou kapacitu a jsou zpoplatněny podle tarifu operátorů.

Další volbou byl výběr kryptografického protokolu pro bezpečné ustavení sdíleného tajného klíče. Vybírali jsme z publikace *Protocols for Authentication and Key Establishment* a nakonec jsme zvolili protokol *Burmester-Desmedt Key Agreement* (dále jen BDP) [BM03, str. 214, Protocol 6.8]. Protokol je poměrně jednoduchý a snaží se optimalizovat poměr mezi počtem zasílaných zpráv a výpočetní složitostí jednotlivých fází. V podstatě je zobecněním známého Diffie-Hellman protokolu.

Protokol je nejvhodnější použít v komunikačních sítích kruhové topologie umožňujících broadcast. V takovýchto podmínkách je implementace protokolu velmi jednoduchá a přímočará. Protokol má dvě fáze, v první pošle každý uzel svým sousedům svoji část klíče (stejně jako v DH protokolu) a spočítá poměr mezi částečnými klíči obdrženými od sousedů. Ve druhé fázi probíhá broadcast těchto výsledků.

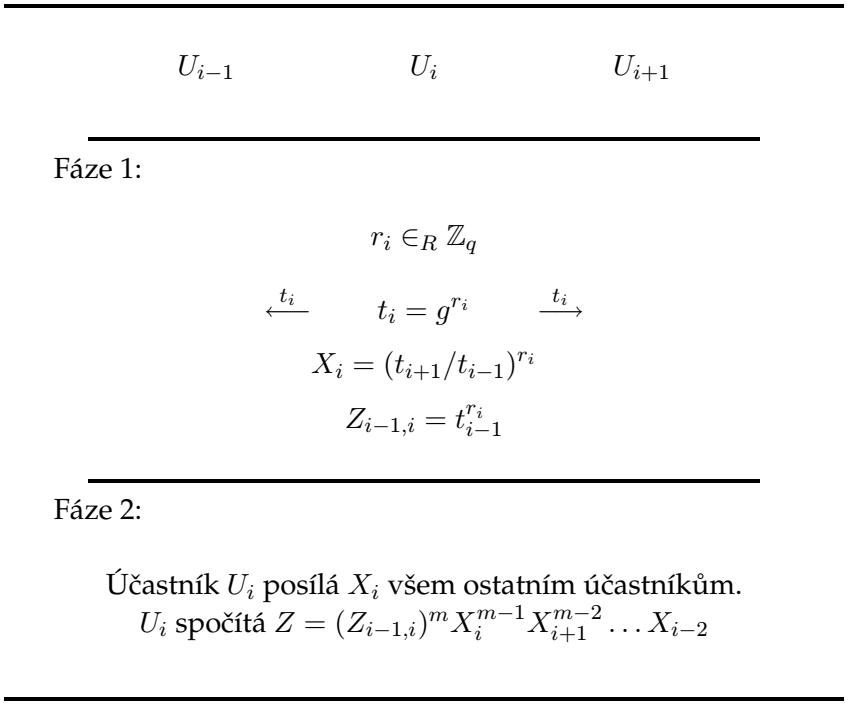
Jakmile má uzel zprávy od všech ostatních účastníků, spočítá klíč, který je tajný a sdílený mezi všemi zúčastněnými.

Samozřejmě všechny operace jsou realizovány v modulární aritmetice (modulo N).

Ze schématu na obrázku 5.1 jsou zřejmé dvě základní vlastnosti protokolu, tj. kruhová komunikace v první fázi a broadcastová ve fázi druhé. Ve své podstatě ani jednu z nich Bluetooth přímo nepodporuje a musí být nějakým způsobem emulovány. V našem případě, kde je počet současně komunikujících zařízení omezen na osm, to nečiní žádné zásadní problémy. Nyní popíšeme důležité části technologie Bluetooth, které budou podstatné pro porozumnění implementaci BDP.

5.1 Bluetooth

Technologie Bluetooth (dále jen BT) zajišťuje bezdrátový komunikační systém, který byl vytvořen jako náhrada „pevného“ připojení kabelem, proto se nejčastěji využívá u přenosných zařízeních (notebook, mobilní telefony, PDA, . . .) pro připojení různých



Obrázek 5.1: Burmester-Desmedt protokol

periferií (myš, klávesnice, headset, GPS, apod.).

BT pracuje v bezlicenčním pásmu 2,4 GHz a používá metodu *Frequency-hopping Spread Spectrum* (FHSS), jejíž princip spočívá v pseudonáhodném přeskakování mezi přenosovými frekvencemi. To zajišťuje odolnost proti rušení a také zvyšuje bezpečnost přenosu.

BT specifikace [SIG07] je vyvíjena a spravována asociací *Bluetooth Special Interest Group*. Je to skupina výrobců softwarových nebo hardwarových produktů, které používají BT (např. Nokia, Lenovo, Microsoft). Existuje několik verzí specifikace, první (kvůli mnoha chybám dnes už nepoužívaná) byla verze 1.0, po ní přišla verze 1.1, která byla standardizována jako IEEE 802.15.1-2002 [IEE02]. Opravila mnoho chyb předchozí verze a stala se základem pro dosud používanou verzi 1.2 (IEEE 802.15.1-2005 [IEE05]). Hlavním přínosem je efektivnější vyhledávání okolních zařízení, větší přenosová rychlost (až 1 Mb/s) a adaptivní FHSS, které se snaží preferovat méně používané frekvence.

Poté bylo organizacemi IEEE a BT SIG rozhodnuto, že další verze nebudou již procházet procesem standardizace, ale budou vycházet pouze jako specifikace. V roce 2004 to byla verze 2.0, hlavní vylepšení spočívá v zavedení EDR (*Enhanced Data Rate*), které zvětšuje přenosovou rychlost teoreticky až na 3 Mb/s a také snižuje spotřebu těchto zařízení.[1] Zajímavé je, že EDR není povinnou součástí, výrobci mohou uvést zařízení splňující specifikaci verze 2.0 aniž by implementovali EDR.

Aktuální verze specifikace s číslem 2.1 je implementována ve většině nových periferií, ale například v mobilech podpora zatím většinou chybí. V této verzi je hlavní důraz kladen na bezpečnost při zachování jednoduchosti, zajímavým konceptem je použití *Near*

[1] Všechna testovaná zařízení (N73, N80, N82) splňovala tuto specifikaci (včetně EDR).

Field Communication ke zjednodušení párování u blízkých zařízení (řádově centimetry).

Specifikace BT se stále vyvíjí, mluví se o verzi 3.0, vývoj směřuje ke stále větším přenosovým rychlostem, nižší spotřebě a lepšímu zabezpečení.

Architektura BT je poměrně složitá, ve své podstatě se řídí ISO/OSI modelem. Pro jednoduchost zmíníme jen protokoly, které přímo souvisejí s implementací BDP. Jsou to SDP (*Service Discovery Protocol*) a RFCOMM (*Radio Frequency Communications*). První zajišťuje vyhledávání zařízení, resp. služeb poskytovaných nalezeným zařízením a druhý je jednoduchý protokol pro sériovou komunikaci (v podstatě emulace RS-232[2]) zajišťující spolehlivý přenos dat. Podrobný popis těchto i dalších protokolů je možné najít v [SIG07].

BT umožňuje dva typy komunikace, dvoubodovou (*point-to-point*) a vícebodovou (*point-to-multipoint*). První, častěji používaný způsob, zajišťuje komunikaci mezi dvěmi zařízeními, v druhém případě se vytváří síť s hvězdicovou topologií nazývaná *piconet*. Toto uspořádání umožňuje komunikaci až osmi zařízení, jedno zařízení (označováno *master*) řídí veškerou komunikaci s ostatními (*slaves*). Z principu hvězdicové topologie vyplývá, že *master* je schopen komunikovat s ostatními přímo, zatímco *slave* může komunikovat s ostatními pouze prostřednictvím *master-zařízení*. Tedy je zřejmé, že implementace BDP bude vyžadovat, aby relativně značné množství zpráv přeposílal *master*, avšak vzhledem k striktnímu omezení počtu současně komunikujících zařízení to nebude činit potíže.

5.1.1 Bezpečnost BT

Nyní se budeme zabývat bezpečností a riziky spojenými s používáním BT. Jako každé přenosové médium i BT se může stát terčem různých útoků, ať už obecně rozšířených (*Man-in-the-Middle*, zachytávání paketů, apod.) nebo specifických (*Bluejacking*, *Bluesnarfing*, *Bluebugging*, atd.).

Oproti ostatním médiím však existuje jeden důležitý rozdíl. Ačkoli je to principiálně možné, většinou není BT komunikace trvalá. Jedná se spíše o jednotlivé (relativně krátkodobé) přenosy dat, proto je vystavení riziku útoku možné dobře omezit vypnutím BT nebo přepnutím do „neviditelného“ režimu.[3]

Pokud je BT aktivní a zařízení je nastaveno jako viditené (*discoverable*), přichází ke slovu několik bezpečnostních mechanismů definovaných BT specifikací.

Autentizace představuje ověření identity komunikujících zařízení (autentizace uživatelů se neprovádí).

Zajištění důvěrnosti zabraňuje odposlouchávání komunikace neautorizovaným zařízením.

Autorizace zajišťuje kontrolu přístupu k nabízeným službám.

Žádné z těchto zabezpečení není povinné, záleží pouze na poskytovateli konkrétní služby, zda umožní připojení neautentizovaného či neautorizovaného zařízení a jestli použije zabezpečený komunikační kanál.

[2] Bývá nazýván „cable replacement protocol“.

[3] Zařízení není možné vyhledat, ale principiálně je možné se k němu připojit, pokud známe jeho BT adresu (teoreticky je možné použít útoku hrubou silou).

Nyní popíšeme kompletní ustavení spojení z hlediska bezpečnosti. Prvním krokem je *párování zařízení*, principem je zadání identického PINu na obou zařízeních, které spolu zahajují komunikaci. Na základě párování se vytváří 128bitový tajný klíč (v angl. literatuře nazýván *link key*), ten je svázán s danou dvojicí komunikujících zařízení a používá se pro veškerá budoucí spojení této dvojice (s výjimkou explicitního požadavku na změnu). PIN může být teoreticky až 16 bajtů, pokud se použije kratší, je při generování klíče doplňován BT adresou zařízení.[4]

Popsaný způsob párování je relevantní pro zařízení se specifikací BT 2.0 a nižší, ve verzi 2.1 se přechází na *Secure Simple Pairing*, které využívá poněkud jiný mechanismus (*Elliptic Curve Diffie-Hellman*). Podrobnostmi se nebudeme zabývat, protože žádné z testovaných zařízení nesplňovalo tuto specifikaci.

Po ustavení klíče linky probíhá autentizace (pokud je vyžadována), jedná se o klasický protokol *výzva-odpověď*, konkrétně o důkaz znalosti tajného klíče (popis a schéma např. v [SP08, kap. 3.3]). Pokud autentizace selže, čeká se určitý časový interval (exponenciálně závisející na počtu chybných pokusů) s cílem zamezit útokům hrubou silou.

Pokud požadujeme zajištění důvěrnosti přenosu dat, použije se proudové šifrování s klíčem odvozeným z klíče linky. Pro generování proudu klíčů se využívá posuvných registrů (*LSFR*), délka klíče je variabilní, závisí na dohodě obou komunikujících stran a pohybuje se v intervalu 8 až 128 bitů. Je důležité upozornit na to, že šifrovací algoritmus (označovaný E_0) není standardizován a byl publikován útok [LMV05], který teoretickou obtížnost prolomení klíče snižuje z 2^{128} až na 2^{38}.

Třetím bezpečnostním mechanismem, který je implementován přímo na úrovni jednotlivých nativních protokolů, je *autorizace*. Základním principem je rozdělení spárovaných zařízení do dvou skupin, na *důvěryhodné* a *nedůvěryhodné*. Zařízení z první skupiny mají zajištěn přístup ke všem nabízeným službám, nedůvěryhodná mají automaticky přístup pouze ke službám, které nevyžadují autorizaci, k ostatním pouze po explicitním manuálním souhlasu poskytovatele.

Speciální publikace NIST SP800-121 [SP08, tab 4-1] uvádí 15 různých zranitelností, které existují ve verzích 1.2 až 2.0. Jako příklad uvedeme následující:

- možnost použití krátkého PINu a z toho plynoucí slabý linkový klíč;
- není známa kvalita generátoru pseudonáhodných čísel používaných při autentizaci a šifrování;
- algoritmus E_0 je slabý, navíc délka klíče je variabilní.

Těchto slabých míst využívá několik praktických útoků:

Bluejacking je ve většině případů neškodný (nedestruktivní) útok, jehož principem je zasílání nevyžádaného, obtěžujícího obsahu oběti[5].

Bluesnarfing je mnohem nebezpečnějším typem útoku, využívá různých chyb v implementaci firmwaru nebo protokolů k získání kontroly nad zařízením. Uvádí se, že je takto možné například získat IMEI mobilního telefonu a zneužít ho k odposlouchávání telefonních hovorů. U tohoto útoku oběť ještě stále musí do jisté míry „spolupracovat“ s útočníkem.

[4] Přehledné schéma ustavení klíče je možné najít v [SP08, obr. 3-2].

[5] Je zde zřejmá souvislost se spamem v případě elektronické pošty. Pokud útočník přesvědčí oběť k předem naplánovanému chování, může se i tento útok stát nebezpečný (např. poskytnutí citlivých údajů).

Bluebugging má podobné cíle jako bluesnarfing, hlavní rozdíl spočívá v tom, že oběť o tomto útoku neví, tzn. nemusí explicitně akceptovat příchozí spojení, apod. V případě úspěšného útoku získává útočník plnou kontrolu nad zařízením (u mobilního telefonu to může být například přístup k citlivým údajům jako jsou kontakty nebo kalendář, možnost využívat zpoplatněné služby, volání, SMS, internet, atd.).

Car Whisperer je nástroj vyvinutý skupinou *European Security Researchers*, který zneužívá BT handsfree sady instalované v automobilu k odposlouchávání hovorů nebo zasahování do nich. Útok využívá stálého, predikovatelného nastavení PINu v těchto zařízeních.

Existují další (viz výše citovaný dokument [SP08]), které už ale nejsou tak prakticky využitelné nebo ještě nejsou známy podrobné informace.

5.2 Implementace Burmester-Desmedt protokolu

Nyní se vraťme zpět k popisu BDP na začátku této kapitoly (obr. 5.1). Už tam jsme zmínili, že kvůli implementaci nad BT sítí bude potřebné poněkud pozměnit komunikační část protokolu. Zprávy definované BD protokolem nejsou modifikovány (stejně jako jejich pořadí), pouze postup posílání se mění. Zprávy od *slave-uzlů* jsou bez jakékoli modifikace přeposílány *masterem* původnímu adresátovi, navíc je do komunikace vloženo několik režijních zpráv.

V této části nejprve popíšeme implementaci ustavení komunikační sítě, tzn. aktivace BT, eventuální párování a autorizace a vytvoření hvězdicové topologie s jedním *masterem* a maximálně sedmi *slave-uzly*. Poté se budeme zabývat formátem přenášených zpráv a jejich souvislostí s popsaným BDP (obr. 5.1). Nakonec popíšeme kompletní běh protokolu a implementaci komunikační části v rámci symbianovského prostředí.

5.2.1 Ustavení komunikační sítě Bluetooth

Při implementaci této části výsledné aplikace byli nejdůležitějšími (a v podstatě jedinými dostupnými) zdroji informací dokument *S60 Platform: Bluetooth API Developer's Guide* [Nok08a] a zejména ukázková aplikace *S60 Platform: Bluetooth Point-to-Multipoint Example* [Nok08b]. Na jejich základě vznikla kostra aplikace, do které bylo ještě nutné doplnit definici posílaných zpráv a implementovat BD protokol.

Nyní popíšeme podstatné části Symbian Bluetooth API, konkrétně jeho druhou verzi, které je implementováno na zařízeních se Symbian OS od verze 9.1.

Bluetooth sokety

Základní koncepcí komunikace přes BT je použití soketů, implementace takovéto komunikace je z pohledu programátora stejná jako například implementace TCP/IP spojení. Socket API v Symbian OS definuje třídu `RSocket`, která zapouzdřuje veškerou funkcionalitu spojenou s vytvořením, spojením a komunikací přes soket.

Člověk seznámený s programováním soketů v běžných operačních systémech na PC by neměl mít se základním použitím třídy RSocket zásadní problémy. Poněkud matoucí může být pouze rozdělení rolí při komunikaci. Je zvykem označovat jedno zařízení

v síti jako *server* a ostatní, k němu se připojující zařízení, jako *klienty*. Tedy běžně *server* vytvoří soket, na kterém naslouchá[6] a čeká na příchozí spojení. Pokud se nějaký *klient* připojí[7], *server* přijímá spojení[8] a naváže jej na nově vytvořený soket, na původním soketu naslouchá a čeká na další pokusy o spojení. Tedy topologie této komunikace je hvězda, centrem je *server* přijímající spojení od klientských zařízení.

Intuitivní představa komunikace v BT síti piconet je taková, že máme hvězdicovitou topologii s *master-zařízením* v centru a *slave-zařízeními* okolo něho, potud je intuice správná. Ovšem pokud se budeme snažit použít terminologii *klient-server*, naše intuice zřejmě zklame, protože jako *klient* se (z hlediska soketů) chová právě jen *master-zařízení*, naopak *slave-zařízení* jsou v roli *serverů*.

Tedy *slave-zařízení* otevírají soket a čekají na příchozí spojení, zatímco *master* se připojuje k jednotlivým *slave-zařízením*. Z toho plyne i to, že *master* má přímé spojení se všemi *slave-zařízeními*, kdežto *slave-zařízení* jsou připojeny pouze k *master-zařízení* a komunikace s ostatními *slave-zařízeními* musí být přeposílána přes *master-zařízení*.

Komunikace po navázání všech spojení je již jednoduchá, spočívá v použití asynchronních metod zápisu do otevřeného soketu, resp. čtení z něj.[9]

Vzhledem k tomu, že topologie BT sítě není statická, ale velmi dynamická, musíme mít mechanismy, jež nám dovolí vyhledat zařízení, která jsou v našem okolí, a navíc potřebujeme zjistit, zda nalezená zařízení nabízejí námi požadované služby.

Vyhledávání okolních zařízení

Z pohledu budoucího *master-uzlu* nám tedy v první řadě půjde o vyhledání všech zařízení, která jsou v našem dosahu. Existují dvě možnosti, jak to udělat, použít *Bluetooth UI*, předpřipravený dialog využívaný ve většině aplikací používajících BT, který ale vyžaduje interakci s uživatelem, nebo hledat plně automatizovaně s využitím třídy `RHostResolver`. Pro účely naší aplikace je výhodnější použít druhý způsob, *Bluetooth UI* se hodí spíše pro komunikaci dvou zařízení. Funkcionalita je v aplikaci implementována třídou `CKeyAgreeDeviceDiscoverer`. Postup je tedy následovný:

1. Jak už jsme zmínili, veškerá komunikace se realizuje pomocí soketového API, ani vyhledávání není výjimkou. Proto se musíme nejdříve připojit k *soket-serveru* (`RSocketServ`) a zvolit odpovídající protokol (`BTLinkManager`).

2. Dalším krokem je vytvoření a inicializace objektu třídy `RHostResolver`. Musíme nastavit, jestli požadujeme pouze adresu nalezených zařízení nebo i jejich jméno (`KHostResInquiry` nebo `KHostResName`), navíc je možné určit mód vyhledávání (*Inquiry Access Code*).

   Existují dvě možnosti, *Generic Inquiry Access Code (GIAC)* a *Limited Inquiry Access Code (LIAC)*. Rozdíl mezi nimi je, jednoduše řečeno, v počtu vyhledaných zařízení, zatímco v módu *GIAC* odpoví na dotaz všechna dostupná zařízení v okolí, použijeme-li omezený dotaz, odpoví pouze zařízení, která jsou v režimu *Limited-Discoverable*. Výhodou této možnosti je zrychlení prohledávání, jestliže víme, že

[6] Na objektu soketu zavolá metodu `Listen()`.
[7] Zavolá metodu `Connect()`.
[8] Volá `Accept()`.
[9] Volání metod `Write()` a `Read()`.

zařízení, se kterými chceme komunikovat jsou v odpovídajícím režimu. Velkou nevýhodou použití *LIAC* módu je vynucení dvou oprávnění[10] (*Capability*), která není možné udělit uživatelem a jsou dostupné jen po testování v Symbian Signed. Považujeme toto omezení za natolik významné, že jsme se raději této možnosti vzdali. Zdržení 15–40 vteřin není pro naši aplikaci v této fázi kritické.

3. Po inicializaci a nastavení resolveru stačí jen zavolat asynchronní metodu `RHostResolver::GetByAddress()` a čekat na dokončení odpovídajícího aktivního objektu. Výsledkem je naplněná struktura `TNameEntry` obsahující adresu, jméno a další informace o nalezeném zařízení.

4. Po zpracování dodaných údajů je možné pokračovat v hledání voláním metody `RHostResolver::Next()`.

Tímto získáme postupně adresy všech okolních zařízení. Bohužel není (v současné verzi specifikace BT) možné přímo vyhledávat pouze zařízení splňující nějaké dodatečné podmínky (např. nabízející konkrétní službu), proto musíme z nalezených zařízení vybrat ty, na kterých běží požadovaná služba, pomocí *Service Discovery protokolu (SDP)*.

Poskytování a vyhledávání služeb

Dříve, než může být služba běžící na určitém zařízení využívána ostatními, musí být o její existenci vytvořen záznam v *SDP databázi*. Pro každou službu je zde několik atributů, které ji jednoznačně identifikují a slouží pro její vyhledání. Databáze funguje jako lokální server (třída `RSdp`), ke kterému se poskytovatel nové služby připojí a vytvoří relaci (klientskou část pokrývá třída `RSdpDatabase`). Voláním metody `RSdpDatabase::CreateServiceRecordL()` vytvoříme nový záznam a službu jednoznačně identifikujeme jejím *UUID (Universally Unique Identifier)*.

Vytvořený prázdný záznam je třeba naplnit. Pomocí metody `RSdpDatabase::UpdateAttributeL()` doplníme podporované protokoly (v našem případě se jedná o RFCOMM), jméno služby, případně její popis.

Nakonec se musí záznam zveřejnit (modifikací atributu `KSdpAttrIdServiceAvailability`). Od této chvíle je služba dostupná zvenčí.[11]

Tento popis je poměrně stručný, zájemce o podrobnosti (obzvláště k nastavení UUID, resp. množin UUID, a dalších atributů) odkážeme na [Nok08a, str. 32, kap. Advertising a service].

Jak už bylo uvedeno dříve, služby v síti piconet poskytují *slave-zařízení*. Veškeré manipulace s SDP databází jsou v implementaci zapouzdřeny v třídě `CKeyAgreeServiceAdvertiser`, podstatné jsou metody `StartAdvertiserL()`, vytvoří a naplní záznam, `UpdateAvailabilityL()`, zveřejní záznam, a konečně `StopAdvertiserL()`, odstraní záznam a provede úklid.

Nyní popíšeme, jak je implementováno vyhledávání požadovaných služeb pomocí *Service Discovery protokolu*. K veškeré funkcionalitě je přístup v třídě `CSdpAgent`. Vyhledávání je potenciálně časově náročné, proto jsou metody asynchronní a BT API poskytuje

[10] `NetworkControl` a `WriteDeviceData`

[11] Zveřejnění záznamu jistou chvíli trvá, testováním jsme zjistili, že pokud se nečeká 2–3 vteřiny, aplikace se chová nedeterministicky.

rozhraní `MSdpAgentNotifier`[12], tzv. *observer*, jehož implementováním vzniknou *callback funkce*, které jsou volány v reakci na různé události (nalezení služby, chyba).

V naší aplikaci je implementací rozhraní `MSdpAgentNotifier` třída `CKeyAgreeServiceDiscoverer`.

1. Nejdříve musíme vytvořit instanci objektu třídy `CSdpAgent`, ta je pomocí BT adresy pevně svázána s konkrétním zařízením, na kterém chceme služby hledat, a má referenci na objekt implementující rozhraní `MSdpAgentNotifier`[13].

2. Dále je nutné vytvořit vyhledávací filtr[14] a aplikovat jej na vyhledávacího agenta[15].

3. Samotné vyhledávání je iniciováno voláním asynchronní metody `CSdpAgent::NextRecordRequestL()`. Jakmile je hledání dokončeno, operační systém zavolá metodu `MSdpAgentNotifier::NextRecordRequestComplete()`[16], jejíž argumenty obsahují odkaz na první vyhovující službu a také celkový počet služeb vyhovujících zadanému vyhledávacímu vzorku. Odkazy na další vyhovující služby získáme opakovaným voláním metody `CSdpAgent::NextRecordRequestL()`.

4. Po vyhledání odpovídajících služeb ještě většinou potřebujeme získat další informace (např. jméno), v naší aplikaci komunikující pomocí RFCOMM půjde především o informaci, se kterým kanálem[17] je služba svázána. Opět se využije vyhledávacího agenta a jeho metody `AttributeRequestL()`. Po skončení operace se volá `MSdpAgentNotifier::AttributeRequestComplete()`.

5. V této chvíli již máme všechny potřebné informace a celé vyhledávání končí. Pokud chceme prohledávat i další nalezená zařízení, pak začínáme znovu od bodu 1.

Podrobnější popis vyhledávání služeb může čtenář najít v [Nok08a, str. 27, kap. Service discovery] nebo v dokumentaci.

Navázání spojení

V této chvíli se dostáváme do situace, kdy *master* našel všechna dostupná okolní zařízení a vybral ta, která nabízejí požadovanou službu (tzn. jsou připravena podílet se na běhu protokolu pro ustavení sdíleného klíče). Jak už bylo uvedeno u popisu soketového rozhraní na str. 33, každý *slave* má s touto službou asociován otevřený soket, na kterém poslouchá a čeká na příchozí spojení. *Master* má všechny potřebné informace (BT adresu a kanál), proto se může pokusit ke službě připojit. Na straně *master-zařízení* je spojení zapouzdřeno ve třídě `CKeyAgreeConnector`, pro *slave-zařízení* je to `CKeyAgreeListener`.

[12] Počáteční *M* ukazuje na třídu s čistě virtualními metodami (obdoba klíčového slova `interface` v Javě). Vícenásobná dědičnost je v Symbian C++ možná jen s použitím těchto tříd.
[13] V našem případě je to přímo `CKeyAgreeServiceDiscoverer`.
[14] Pomocí objektu třídy `CSdpSearchPattern`.
[15] Voláním metody `CSdpAgent::SetRecordFilterL()`.
[16] Implementace v `CKeyAgreeServiceDiscoverer::NextRecordRequestComplete()`.
[17] Obdoba portu u IP.

Třída `CKeyAgreeListener` Nejdříve popíšeme třídu `CKeyAgreeListener`, jejíž inicializace přichází ke slovu dříve. Objekty této třídy budou obsluhovat asynchronní události spojené se sokety (připojení, zápis a čtení dat), proto je třída implementována jako *aktivní objekt*, z čehož plyne přítomnost metod `RunL()`, resp. `RunError()`, které obsluhují korektní, resp. chybové ukončení asynchronního volání.

Konstrukce objektů je dvoufázová a podstatné na ní jsou dvě inicializace, jednak stavu objektu (třída implementuje poměrně komplexní stavový automat) a odkazu na hlavní třídu aplikace `CKeyAgreeEngine` implementující v podstatě veškerou funkcionalitu, protože ostatní objekty pouze využívají odkazu na ni k předání lokálních výsledků ke globálnímu zpracování.

Dalšími důležitými metodami jsou `StartListenerL()` a `StopListener()`. Pomocí první z nich se inicializuje spojení, tzn. na volném kanále se otevře nový soket pro protokol RFCOMM, na kterém se čeká na příchozí spojení[18]. Následně se nastaví bezpečnostní mechanismy používané během spojení (autentizace, autorizace, šifrování [19]).

Otevře se ještě jeden volný soket, který bude později použitý pro příchozí připojení a voláním `Accept()` na naslouchacím soketu (a aktivací objektu) začínáme čekat na příchozí spojení od *master-zařízení*. Po připojení *mastera* je další průběh řízen metodou `RunL()`, která pomocí změny stavu vnitřního automatu implementuje BD protokol. K této problematice se ještě vrátíme v samostatné části.

Ještě zmíníme dvě metody, `ReceiveData()` a `SendData()`. Z jejich názvů je zřejmý jejich účel, první volá metodu `RecvOneOrMore()` třídy `RSocket`, která asynchronně čeká na data zapsaná druhou stranou, a druhá naopak volá `RSocket::Write()` a zapíše data do soketu.

Na tomto místě je třeba upozornit na nepříjemný problém, který vzniká asynchronní povahou zápisu do soketu. Je totiž třeba rozlišit tři typy „dokončení“ této operace.

Za první „dokončení“ se dá považovat návrat z metody `RSocket::Write()`. Je zřejmé, že v tuto chvíli nemá protější strana komunikace zapsaná data, pouze se vydal pokyn k jejich zápisu. Poměrně překvapivé ale je, že návrat z této metody neznamená ve skutečnosti vůbec nic, dokonce ani to, že data předaná k zápisu v nějakém bufferu byla už zpracována (resp. převzata interními datovými strukturami operačního systému). To znamená, že s bufferem nesmí být manipulováno až do chvíle, kdy dochází k druhému „dokončení“ (tj. asynchronní návrat a vyvolání metody `RunL()` plánovačem aktivních objektů). Pak je garantováno, že data byla předána ke zpracování operačnímu systému a buffer je možno dále používat (i destruktivně). Předčasná destrukce bufferu má nedeterministické účinky na průběh komunikace, které se jen velmi těžko odhalují, opatrnosti je třeba zvláště při používání lokálních proměnných na zásobníku. Bohužel dokumentace o tomto problému mlčí (napoví až různé diskuzní příspěvky).

Třetí typ „dokončení“, který ale není nijak signalizován, je faktické doručení dat příjemcově aplikaci.

[18] Posloupností volání `Open()`, `Bind()`, `Listen()`.

[19] `TBTServiceSecurity::SetAuthentication()`, `TBTServiceSecurity::SetAuthorisation()`, `TBTServiceSecurity::SetEncryption()`

Třída `CKeyAgreeConnector` Nyní se budeme zabývat spojením na straně *master-zařízení*. Tato třída je velmi podobná svému protějšku `CKeyAgreeListener`, opět jde o aktivní objekt implementující stavový automat pro běh BD protokolu a obsluhující asynchronní operace čtení a zápisu. Zatímco každý *slave* má pouze jednu instanci objektu třídy `CKeyAgreeListener`, *master* má pro každé spojení (tzn. pro každé *slave-zařízení*) vlastní instanci `CKeyAgreeConnector`.

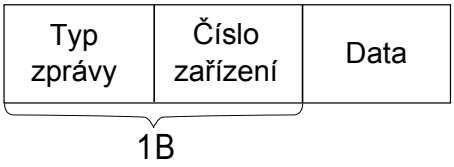
Popíšeme rozdílné části implementace. Jedná se hlavně o metodu `ConnectL()`, která se na základě informací získaných při vyhledávání zařízení (BT adresa) a služeb (kanál) pokusí připojit k zařízení. Nejdříve se požádá soket server o nový soket, nastaví se odpovídající bezpečnostní mechanismy a voláním metody `RSocket::Connect()` se provede připojení.

Toto se provede z hlavní aplikační třídy `CKeyAgreeEngine` pro všechna nalezená vyhovující zařízení a komunikační síť můžeme považovat za ustavenou. V této chvíli začíná vlastní běh BD protokolu pro ustavení sdíleného klíče.

5.2.2 Implementace protokolové části

Jak už jsme uvedli na začátku této kapitoly, originální *Burmester-Desmedt protokol* musí být pro běh v prostředí BT piconetu mírně modifikován. Jde zejména o přeposílání všech zpráv přes centrální *master* zařízení, což velmi mění „tvar" komunikace. Pro zjednodušení implementace byly přidány také další zprávy předávající doplňující řídící informace. Nejprve tedy uvedeme formát zpráv a jejich implementaci v třídě `CKeyAgreeProtMsg`.

Zprávy protokolu a třída `CKeyAgreeProtMsg` Formát zpráv protokolu je velmi jednoduchý, zprávy nemají pevnou délku. Obecně je formát zpráv naznačen na obr. 5.2.

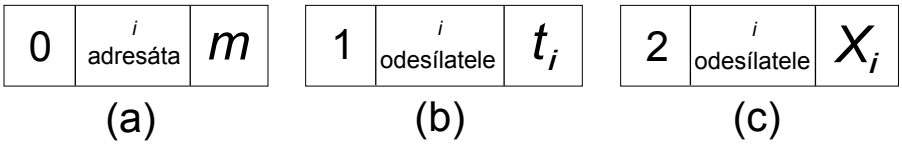


Obrázek 5.2: Obecný formát zprávy

V prvním bajtu jsou zakódovány dva údaje, první čtyři bity určují typ zprávy, zbývající bity značí v závislosti na typu zprávy buď číslo zařízení odesílatele, nebo příjemce. Protože máme pouze tři různé typy zpráv a maximálně osm zařízení v síti, údaje mohou být uloženy v jednom bajtu. Zbytek zprávy jsou data, jejichž sémantiku určuje první pole.

Definujeme tři typy zpráv, první typ (obr. 5.3a) definuje počáteční zprávu, posílá ji *master* postupně všem *slave-zařízením*. Zpráva obsahuje v datové části m (tj. počet účastníků protokolu), druhé pole v tomto případě znamená číslo příjemce přidělené *masterem* danému adresátovi.[20]

[20] V podstatě se jedná o číslo účastníka protokolu i uvedené v popisu protokolu 5.1.



Obrázek 5.3: Formát (a) prvního, (b) druhého a (c) třetího typu zpráv

Druhý typ zpráv (obr. 5.3b) slouží k přenosu t_i z první fáze protokolu 5.1. Druhé pole je číslo zařízení odesílatele (důležité kvůli přeposílání zpráv přes *master-zařízení*), v datové části se přenáší t_i samotné.

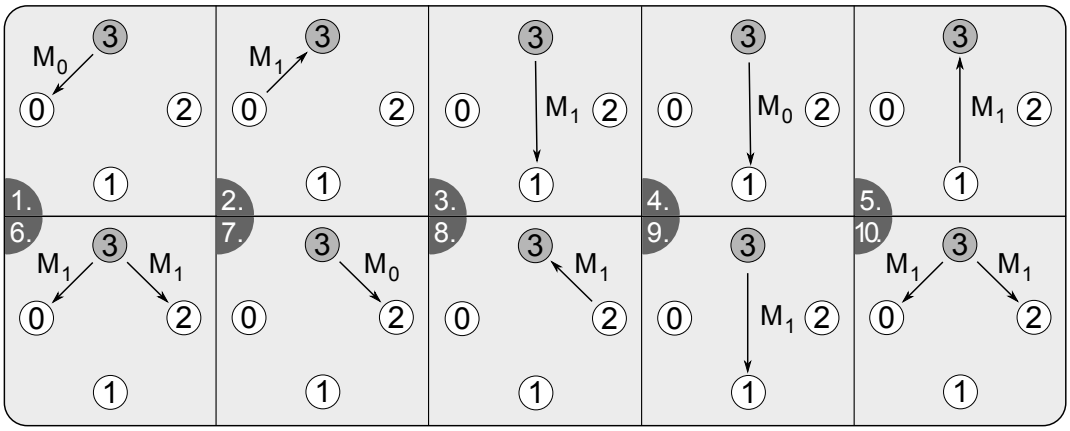
V třetím a posledním typu zpráv (obr. 5.3c) se přenáší data druhé fáze BD protokolu (X_i v popisu 5.1). Druhé pole opět obsahuje číslo odesílajícího zařízení a datové pole obsahuje přenášené X_i.

Zprávy jsou interně zpracovávány pomocí třídy `CKeyAgreeProtMsg`, ta obsahuje dvě datové položky pro uložení typu zprávy a čísla odesílatele (resp. příjemce) a ukazatel na data, získaný externalizací[21] obsahu třídy `RInteger`, která se používá kvůli uchovávání velkých číselných hodnot (t_i nebo X_i).

Zmíníme dvě důležité metody, `Externalize()`, která vytvoří ze zprávy uložené v datových položkách deskriptor (proud bajtů) uvedeného formátu (viz obr. 5.3). Tato data je pak možné přímo odeslat pomocí `RSocket::Write()`.

Na druhé straně je pak přístup k položkám zprávy možný po rekonstrukci zprávy z bajtového bufferu pomocí volání `Internalize()`.

Popis protokolu V této chvíli již máme popsány veškeré aplikační prostředky, které jsou obsaženy v implementaci a můžeme přejít k popisu protokolu samotného.



Obrázek 5.4: První fáze protokolu

Doplníme jej grafickým znázorněním probíhající komunikace na příkladě se čtyřmi komunikujícími zařízeními (obr. 5.4), což je reálná testovací konfigurace. Bohužel nebyla možnost vyzkoušet běh aplikace na více zařízeních, zejména proto, že implementace

[21] `RInteger::BufferLC()`

pro různé mobilní telefony se ukázala jako velmi složitá. Kompatibilita typově různých modelů (i stejného výrobce) je totiž velmi diskutabilní, zejména v oblasti implementace některých nepříliš často používaných API (např. Crypto API a implementace třídy `RInteger`).

Běh protokolu začíná po ustavení komunikační sítě. Z implementačních důvodů byl protokol modifikován do podoby na obrázcích 5.4 a 5.5, používá se princip *výzva-odpověď*, výzvu posílá *master* a *slave-zařízení* pouze odpovídají. Tímto získáváme striktní uspořádání všech zpráv protokolu. Dále se budeme držet rozdělení protokolu na dvě fáze (viz popis protokolu 5.1).

První fáze Pro připomenutí uveďme, že v první fázi si účastníci protokolu vymění se svými sousedy[22] „částečné klíče" $t_i = g^{r_i}$, kde r_i je vygenerované pseudonáhodné číslo (využívá se běžící Fortuna). Pak se spočítají hodnoty X_i a $Z_{i-1,i}$, které se využijí v druhé fázi.

Protokol zahajuje *master* zprávou M_0[23] zaslanou prvnímu *slave-zařízení*. Zpráva obsahuje jemu přidělené číslo zařízení i a v datové části celkový počet účastníků protokolu m.

Obecně to nemusí být první zpráva, kterou účastník dostává.[24] Důvodem je to, že odpověď M_1 od zařízení (i) je určena zařízením $(i-1)$ a $(i+1)$ a účastník $(i+1)$ ještě zprávu M_0 nedostal. Zřejmé je to z obr. 5.4 (1.–4. krok), *master* (3) posílá M_0 uzlu (0), ten odpovídá zprávou M_1, která je určena jeho sousedům (3) a (1). Proto ji v 3. kroku *master* přeposílá i uzlu (1) a teprve poté následuje výzva M_0 pro uzel (1).

Odpověď v 5. kroku sice vůbec nepatří *master-zařízení* (je adresována uzlu (0) a (2)), ale kvůli hvězdicové topologii musí být posílána přes *master-zařízení*, které ji nijak nemodifikuje ani nezpracovává, pouze přeposílá zamýšleným adresátům.[25]

Posledním krokem 1. fáze je odeslání vlastní zprávy M_1 *master-zařízením* sousedům (0) a $(m-2)$. Po tomto kroku mají všichni účastníci protokolu hodnoty t_i od svých sousedů a mohou spočítat X_i a $Z_{i-1,i}$ potřebné pro zbytek protokolu.

Druhá fáze Úkolem druhé fáze protokolu je rozšíření X_i mezi všechny ostatní účastníky. Po skončení této fáze tedy pro každý uzel platí, že zná všechna X_i, kde $0 \leq i < m$ a může tak spočítat sdílené tajemství $Z = (Z_{i-1,i})^m X_i^{m-1} X_{i+1}^{m-2} \dots X_{i-2}$.

Vzhledem k omezením daným prostředím, ve kterém protokol probíhá (v této fázi zejména absence všesměrového vysílání), musel být běh protokolu opět poněkud modifikován. Znovu jsme využili koncepce *výzva-odpověď* a jí daného uspořádání posílaných zpráv (obr. 5.5).

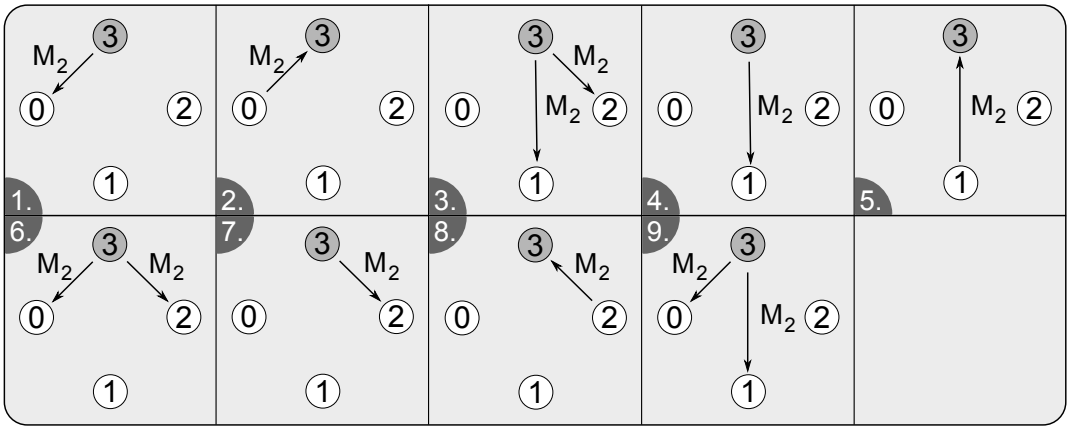
Druhou fázi protokolu zahajuje *master* posláním zprávy M_2 prvnímu zařízení (s číslem 0). Zpráva ve své datové části obsahuje X_i *mastera*. Odpověď zprávou M_2 obsahující X_i prvního zařízení je určena všem účastníkům, proto ji *master* ve třetím kroku přeposílá všem ostatním (uzlům (1) a (2)). Tento vzorec se opakuje i pro ostatní účastníky, pro uzel (1) v 4.–6. kroku a pro uzel (3) v 7.–9. kroku. Protože v 1., 4. a 7. kroku posílá postupně

[22] Ve smyslu kruhové topologie, v našem případě explicitně určené číslem zařízení přiděleným jim *master-zařízením*.

[23] M_j budeme označovat zprávy typu j.

[24] Ve skutečnosti je to úplně první zpráva pouze pro zařízení s číslem 0.

[25] Ti jsou jednoznačně určeni číslem odesílatele v těle zprávy.



Obrázek 5.5: Druhá fáze protokolu

vlastní X_i *master*, je zaručeno, že po posledním kroku protokolu mají všichni účastníci kompletní údaje pro výpočet sdíleného tajemství Z.

Analýza modifikací protokolu Porovnejme nyní náročnost provedení BD protokolu v jeho originální a v naší modifikované verzi. Veškeré výpočty jsou jak v původní, tak i v naší verzi distribuovány rovnoměrně mezi všechna zařízení. Role *master*, resp. *slave* nijak neovlivňuje výpočetní zatížení účastníků.

To nelze říci o komunikační náročnosti, která je v původním návrhu stejná u všech uzlů, ale v prostředí BT je kvůli hvězdicové topologii více zatížen centrální prvek, *master*. Ten musí přeposílat zprávy ostatních účastníků.

V originále pošle a přijme každý uzel v první fázi dvě zprávy, ve druhé fázi pak $m-1$ zpráv, kde m je počet účastníků protokolu. Celkový počet odeslaných i přijatých zpráv je pro libovolný uzel $m+1$.

Pokud se zaměříme na modifikovanou verzi, musíme rozlišit roli zařízení v BT komunikaci. *Master* v první fázi odešle $m-1$ zpráv M_0, v odpovědi na ně přijme $m-1$ zpráv M_1, které přepošle (tj. $2(m-3)+2$) a na závěr pošle vlastní M_1 svým sousedům (2 zprávy). Celkově odešle $3(m-1)$ a přijme $m-1$ zpráv. Ve druhé fázi musí *master* emulovat broadcast, proto je nárust komunikační složitosti větší. *Master* posílá $m-1$ „vlastních" zpráv M_2, v odpovědi na ně přijímá $m-1$ zpráv M_2, které rozesílá všem ostatním kromě odesílatele (tj. $(m-1)(m-2)$). Celkově odešle *master* ve druhé fázi $(m-1)^2$ zpráv a odešle $2(m-1)$.

Slave přijme v první fázi jednu zprávu M_0 a dvě M_1 a odešle jednu zprávu M_1. Celkově tedy odešle jednu zprávu a přijme tři. Ve druhé fázi odešle jednu zprávu M_2 a přijme $m-1$ zpráv M_2.

Pokud uvažujeme nejhorší případ, tedy osm komunikujících zařízení, pak každý *slave* odešle 2 zprávy a přijme 10, *master* musí přijmout 14 zpráv a 42 odeslat. To je sice oproti originální verzi rapidní nárust, ale z praktického hlediska je akceptovatelný. Musíme uvážit, že protokol pro ustavení sdíleného klíče neprobíhá kontinuálně, ale nejspíš pouze jednorázově na začátku sezení. Proto je časově nejnáročnější operací ustavení komunikační sítě (zejm. vyhledávání zařízení, které trvá řádově desítky vteřin), ne samotný

běh protokolu, jehož trvání je v řádu jednotek sekund. Pokud by se jednalo o jiný typ protokolu, např. distribuované generování dat, kde by komunikace byla dlouhodobější, bylo by lepší zvolit protokol, který by více respektoval specifika sítě Bluetooth.

Volba parametrů protokolu Z popisu Burmester-Desmedt protokolu (5.1) je vidět, že se zde vyskytují tři parametry, jejichž délku (resp. obsah) je třeba zvolit. Jedná se o generátor g, modulus N a náhodný exponent r_i. U prvních dvou jsme zvolili délku 1024 bitů, která je v dnešní době stále ještě akceptovaná jako bezpečná, a hodnoty byly zafixovány pomocí PC aplikace generující parametry pro Diffie-Hellman protokol. Na exponent nebývají kladeny žádné zvláštní požadavky, je proto generován pomocí Fortuna PRNG v délce 16 bitů.

Implementace v aplikaci Jak je naznačeno v popisu tříd `CKeyAgreeConnector` a `CKeyAgreeListener`, je běh protokolu řízen stavovými automaty implementovanými pomocí aktivních objektů v těchto třídách. Nebudeme zde uvádět úplný popis přechodových systémů těchto automatů, protože jsou poměrně komplexní. Zájemce odkazujeme na zdrojový kód aplikace dostupný na přiloženém CD nebo na `http://sourceforge.net/projects/securekeyagree/`.

Stejně jako se u *master-* a *slave-zařízení* liší přístup ke komunikační síti, tj. přes `CKeyAgreeConnector`, resp. `CKeyAgreeListener`, liší se i jejich stavové automaty. Mají disjunktní množiny stavů a *konektory* na straně *mastera* mají při jeho řídící funkci samozřejmě větší počet stavů než *listenery* na straně *slave-zařízení*. Dá se říci, že každý z $2m$ soketů má asociován nějaký stav svého stavového automatu.

Řízení automatu je implementováno v metodách `RunL()` odpovídajících tříd, ovšem veškerá funkcionalita je přesunuta do hlavní aplikační třídy `CKeyAgreeEngine`. Jedná se o metody `HandleListenerMsg0()`, `HandleConnectorMsg1()`, `SendMsg2()`, atd.

V těchto metodách se provádějí výpočty všech číselných parametrů protokolu. Protože jsou tyto hodnoty v praxi velké (např. 1024 bitů), nestačí běžná celočíselná aritmetika, ale musíme použít prostředků pro počítání s velkými hodnotami. V operačním systému Symbian existuje nativní knihovna kryptografických operací, jejíž třída `RInteger` tyto operace s jistými omezeními nabízí.[26]

Kromě uložení velkých celočíselných hodnot a základních aritmetických operací se v implementaci používá modulární aritmetika, konkrétně modulární umocňování.

GUI Uživatelské rozhraní je velmi jednoduché. Aplikace je tvořena jedním oknem vyplněným textovým polem (*Rich Text Editor*), jehož obsah je možné libovolně posouvat. V okně se vypisuje průběh komunikace (nalezená zařízení, zprávy o nalezení služby, připojení a nakonec o korektním průběhu celého protokolu).

Ovládacím prvkem je běžné menu dostupné na většině telefonů pod levým tlačítkem. Menu obsahuje čtyři položky.

Key Agreement Touto volbou se zařízení přidávají k protokolu (tj. zařízení je *slave*). Po implementační stránce je to vytvoření služby, otevření soketu a čekání na příchozí spojení.

[26] Nepříjemné je, že plné funkčnosti se nám podařilo docílit pouze s telefony Nokia N73 a N80.

Start Key Agreement Zařízení se stává *masterem*. Vyhledá okolní zařízení s odpovídající službou, připojí se k nim a zahájí běh protokolu.

Show connected Zobrazí připojená zařízení (vypovídací hodnotu má pouze u *mastera*).

Exit Poslední položka menu se zřejmým účelem.

Aplikace je plně funkční jen za dvou předpokladů. Prvním je dostupnost BT, to nemusí být aktivováno předem, aplikace si vyžádá povolení a aktivuje jej programově.

Druhou podmínkou je možnost generování pseudonáhodných čísel, v implementaci se spoléhá na již běžící generátor Fortuna, jehož dostupnost se při startu aplikace ověřuje, při negativním výsledku aplikace končí s chybou.

5.3 Shrnutí

V této kapitole jsme se zabývali výběrem a implementací vhodného protokolu pro bezpečné ustavení sdíleného klíče mezi několika mobilními telefony. Jako komunikační médium bylo zvoleno Bluetooth i přes některé z toho plynoucí nevýhody, jako je omezení počtu komunikujících zařízení, krátký dosah apod.

Implementován byl Burmester-Desmedt protokol, který je jednoduchým rozšířením Diffie-Hellman protokolu. Popsali jsme modifikace, které bylo nutné provést, abychom protokol přiblížili reálné topologii sítě a ostatním vlastnostem BT.

Aplikace byla testována na všech nám dostupných mobilních telefonech (Nokia N73, N80, N82, E51), bohužel plně funkční je pouze na prvních dvou. Modely Nokia N82 a E51 mají nesprávně implementované některé funkce z kryptografické knihovny, a proto není jejich použití možné. Řešením by bylo implementovat vlastní modulární aritmetiku na libovolně velkých číslech.

Zdrojový kód aplikace je dostupný na přiloženém CD nebo na internetových stránkách `http://sourceforge.net/projects/securekeyagree/`.

Závěr

Stálé zvyšování výkonu mobilních zařízení nyní umožňuje implementaci poměrně výpočetně náročných operací. V oblasti kryptografie jsou však zatím k dispozici pouze aplikace se základní funkcionalitou. Proto jsme se v této práci zabývali generováním pseudonáhodných sekvencí na mobilních zařízeních a jejich využitím v kryptografických protokolech. V současné době nemáme žádné informace o podobném výzkumu a zejména praktické implementaci podobných aplikací.

Na začátku práce jsme pro snazší uvedení čtenáře do problematiky shrnuli základní informace o generování náhodných, resp. pseudonáhodných sekvencí a zavedli pojmy důležité pro přesné pochopení dalších částí textu. Popsali jsme i základní principy operačního systému Symbian, které jsou nutné pro porozumění implementačním detailům v praktické části diplomové práce. Zejména jsme se zaměřili na popis bezpečnostních mechanismů zavedených v operačním systému Symbian řady 9.

Tři kapitoly jsou věnovány aplikacím, které jsou hlavním přínosem této diplomové práce. První z nich je aplikace implementující generátor pseudonáhodných čísel podle standardu ANSI X9.31. Generátor je poměrně jednoduchý a v této implementované verzi nezaručuje dopřednou, ani zpětnou bezpečnost. Hlavním přínosem aplikace implementující ANSI PRNG je použití zdrojů náhodnosti z prostředí mobilního telefonu, které se využívají pro jednorázovou počáteční inicializaci generátoru. Jedná se o uživatelský vstup z klávesnice, zvuková data z mikrofonu a obrazová data z hledáčku fotoaparátu. Jejich použití vyžaduje poměrně komplikovanou posloupnost kroků nutných k získání dat a programátor zde naráží na obecný problém této platformy, nedostatek podrobné dokumentace.

Druhou implementovanou aplikací je generátor pseudonáhodných sekvencí Fortuna podle návrhu autorů Schneiera a Fergusona. Tento PRNG má velmi propracovanou architekturu, navíc autoři v publikaci Practical Cryptography uvádějí podrobná implemetační doporučení. Proto bylo nejtěžší zvládnutí implementace kontinuálního běhu generátoru pomocí prostředků Symbian Klient-Server API a obsluha zdrojů náhodnosti dodávajících náhodné události.

Obzvláště důležité, zejména pro praktické nasazení, je nastavení zdrojů náhodnosti. Vždy se bude jednat o kompromis mezi mírou bezpečnosti a použitelnosti. Je totiž nutné si uvědomit, že použití mobilních zařízení pro kryptografické účely je pouze doplňkovou funkcí, která nesmí nijak zásadně omezovat ostatní funkce.

Příkladem je použití fotoaparátu jako zdroje náhodnosti. Protože je přístup k němu exkluzivní, pokud jej používáme kontinuálně pro sběr dat obsahujících entropii, uživateli nebude umožněno využití jeho primární funkcionality. Navíc se zdá, že ani hardwarově není stálý běh fotoaparátu bez problémů. Nepříjemná je i velká spotřeba omezené kapacity baterie.

Proto jsme v implementaci používali zdroje náhodnosti v pravidelných (nastavitelných) intervalech. Nastavení by mělo proběhnout po pečlivé úvaze o předpokládaném

množství generovaných dat a kritičnosti jejich použití. Obecně lze říci, že kontinuální běh zdrojů náhodnosti dodává značně větší množství entropie než je pro běžné použití třeba, proto je implicitně nastaven interval sběru dat na jednu minutu, což se jeví dobrým kompromisem.

V poslední části práce popisujeme Burmester-Desmedt protokol pro bezpečné ustavení tajného klíče a jeho implementaci v prostředí mobilních telefonů propojených pomocí Bluetooth. Kromě popisu protokolu se věnujeme i samotné technologii Bluetooth, zejména po stránce bezpečnosti.

V popisu implementace převažují detaily týkající se ustavení komunikační sítě, párování zařízení a autorizace. Implementace samotného protokolu je pak již poměrně jednoduchá. Důležité jsou zejména modifikace, které musely být provedeny kvůli některým vlastnostem Bluetooth, ty jsou podrobně popsány i graficky znázorněny spolu se zdůvodněním jejich účelu a korektnosti. Okrajově zmiňujeme i grafické rozhraní aplikace a volbu parametrů protokolu.

Celkově vznikaly při implementaci této aplikace největší potíže. Jednak je závislá na předchozí aplikaci (pro volbu náhodných exponentů se využívá Fortuna PRNG), navíc spolupráce několika mobilních telefonů různých typů se ukázala jako velmi komplikovaná. Při implementaci používané telefony Nokia N73 jsou s aplikací plně kompatibilní, to stejné se dá říci i o telefonu Nokia N80. Bohužel u ostatních testovaných zařízení (Nokia E51, Nokia N82) vznikaly komplikace, které zabraňují praktickému použití aplikace. Jedná se o neúplnou implementaci nativní kryptografické knihovny Symbianu, v našem případě se omezení týká modulární aritmetiky s libovolně velkými čísly.

Komunikační část protokolu je na všech testovaných modelech bez problémů, pro praktické nasazení aplikace by však bylo nutné poskytnout vlastní implementaci modulární aritmetiky.

Nyní shrneme výsledky práce, které jsou obsahem přiloženého CD. Jedná se o ANSI PRNG, s uvedenými výhradami požitelný generátor pseudonáhodných sekvencí. Dále Fortuna PRNG s propracovanou architekturou, která je výsledkem mnohaletých zkušeností známých autorů. Tento je v závislosti na nastavení zdrojů náhodnosti použitelný i pro kritické kryptografické aplikace. Pro budoucí práci zůstává zajímavou otázkou použití bezpečnějšího uložení inicializačního souboru, například na SIM kartě telefonu.

V neposlední řadě je třeba uvést aplikaci implementující Burmester-Desmedt protokol, která je logickým závěrem této práce, využívající předchozí aplikaci. Uvedeme dva návrhy pro vylepšení a rozšíření prezentované verze. V první řadě se jedná o typ komunikační sítě využívané pro výměnu zpráv protokolu. Zajímavou možností by mohlo být Wi-Fi nebo SMS. V případě použití Bluetooth by bylo vhodné navrhnout protokol, který by lépe využíval vlastností komunikační sítě.

Závěrem lze říci, že v této diplomové práci se i přes problematickou kompatibilitu různých modelů mobilních telefonů a nedostatek řádné dokumentace podařilo implementovat několik kryptografických aplikací pro operační systém Symbian. Věříme, že to přispěje k rozvoji této oblasti bezpečnosti.

Literatura

[ANS93] American National Standards Institute. *American National Standard X9.31-1992: Public Key Cryptography Using Reversible Algorithms for the Financial Services Industry*, Červen 1993.

[BM03] Colin Boyd and Anish Mathuria. *Protocols for Authentication and Key Establishment*. Springer, 2003.

[Bon] Ron Bondy. C++ Implementation of Fortuna PRNG. Dostupné na http://www.citadelsoftware.ca/Fortuna/Fortuna.htm (Květen 2009).

[CHMSS04] Julio César Hernández, José María Sierra, and André Seznec. The SAC Test: A New Randomness Test, with Some Applications to PRNG Analysis. In *Proceedings of the International Conference Computational Science and Its Applications – ICCSA 2004, LNCS vol. 3043*, pages 960–967, Duben 2004.

[FIP00] Federal Information Processing Standards Special Publication 800-22. A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Aplications, Říjen 2000. Dostupné na http://csrc.nist.gov/publications/nistpubs/800-22/sp-800-22-051501.pdf (Květen 2009).

[Fog] Agner Fog. Uniform Random Number Generators. Dostupné na http://www.agner.org/random/randoma.htm (Květen 2009).

[FS03] Niels Ferguson and Bruce Schneier. *Practical Cryptography*, chapter 10.3 Fortuna. John Wiley & Sons, Inc., New York, NY, USA, 2003.

[HS07] Richard Harrison and Mark Shackman. *Symbian OS C++ for Mobile Phones*, volume 3. John Wiley & Sons, Ltd, Symbian Press edition, 2007.

[IEE02] IEEE. *Standard 802.15.1-2002*, 2002. Dostupné na www.ieee802.org/15/pub/TG1.html (Květen 2009).

[IEE05] IEEE. *Standard 802.15.1-2005*, 2005.

[Kaw] Vinay Kawade. SHA-256 Hashing Algorithm. Dostupné na http://www.newlc.com/SHA-256-hashing-algorithm.html (Květen 2009).

[Kel05] Sharon S. Keller. *NIST-Recommended Random Number Generator Based on ANSI X9.31 Appendix A.2.4 Using the 3-Key Triple DES and AES Algorithms*. National Institute of Standards and Technology, Information Technology Laboratory, Computer Security Division, Leden 2005. Dostupné na http://csrc.nist.gov/groups/STM/cavp/documents/rng/931rngext.pdf (Květen 2009).

[Knu97] Donald E. Knuth. *The Art of Computer Programming, Seminumerical Algorithms*. Addison-Wesley, 3rd edition, 1997.

[KSF99] John Kelsey, Bruce Schneier, and Niels Ferguson. Yarrow-160: Notes on the Design and Analysis of the Yarrow Cryptographic Pseudorandom Number Generator. In *Sixth Annual Workshop on Selected Areas in Cryptography*. Springer Verlag, 1999. Dostupné na http://www.schneier.com/paper-yarrow.ps.gz (Květen 2009).

[KŠM07] Jan Krhovják, Petr Švenda, and Václav Matyáš. The sources of randomness in mobile devices. The 12th Nordic Workshop on Secure IT Systems, 2007.

[KUH04] Song-Ju Kim, Ken Umeno, and Akio Hasegawa. Corrections of the NIST Statistical Test Suite for Randomness. Cryptology ePrint Archive, Report 2004/018, 2004. Dostupné na http://eprint.iacr.org/2004/018.pdf (Květen 2009).

[LMV05] Yi Lu, Willi Meier, and Serge Vaudenay. The conditional correlation attack: a practical attack on Bluetooth encryption. In *Advances in Cryptology – CRYPTO 2005: 25th Annual International Cryptology Conference*, volume 3621 of *Lecture Notes in Computer Science*, pages 97–117, 2005. Dostupné na http://lasecwww.epfl.ch/pub/lasec/doc/LMV05.pdf (Květen 2009).

[Mar] George Marsaglia. Diehard Statistical Test. Dostupné na http://stat.fsu.edu/pub/diehard (Květen 2009).

[MCCM06] Robert McEvoy, James Curran, Paul Cotter, and Colin Murphy. Fortuna: Cryptographically Secure Pseudo-random Number Generation in Software and Hardware. *IET Conference Publications*, 2006(CP519):457–462, 2006.

[MT02] George Marsaglia and Wai Wan Tsang. Some Difficult-to-pass Tests of Randomness. *Journal of Statistical Software*, 7(3):1–8, 2002. Dostupné na http://www.jstatsoft.org/v07/i03/tuftests.pdf (Květen 2009).

[MvOV96] Alfred J. Menezes, Paul C. van Oorschot, and Scott A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1996.

[Nok08a] Forum Nokia. *S60 Platform: Bluetooth API Developer's Guide*, 2008. Dostupné na http://www.forum.nokia.com (Květen 2009).

[Nok08b] Forum Nokia. *S60 Platform: Bluetooth Point-to-Multipoint Example*, 2008. Dostupné na http://www.forum.nokia.com (Květen 2009).

[SIG07] Bluetooth SIG. *Specification of the Bluetooth System*, 2007. Dostupné na http://www.bluetooth.com/NR/rdonlyres/F8E8276A-3898-4EC6- B7DA-E5535258B056/6545/Core_V21__EDR.zip (Květen 2009).

[SP08] Karen Scarfone and John Padgette. Guide to Bluetooth Security. NIST Special Publication 800-121, 2008.

[TMC] Theodore Ts'o, Matt Mackall, and Jean-Luc Cooke. jlcooke's explanation of and improvements on /dev/random. Dostupné na http://jlcooke.ca/random (Květen 2009).

[Uni] University of Technology in Australia, Information Security Research Centre at Queensland. *Crypt-XS Test Suite*. Dostupné na http://www.isi.qut.edu.au/resources/cryptx (Květen 2009).

[Yao82] Andrew C. Yao. Theory and application of trapdoor functions. *Symposium on Foundations of Computer Science*, 0:80–91, 1982.