Masarykova univerzita
Fakulta informatiky

Implementace algoritmu Fortuna a ANSI X9.31 pro mobilní telefon s OS Android

Bakalářská práce

Martin Žák

Brno, jaro 2013

Masarykova univerzita
Fakulta informatiky

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

Student: Martin Žák

Program, obor (příp. specializace): Počítačové systémy a zpracování dat

Vedoucí práce: Ing. Mgr. Zdeněk Říha, Ph.D.

Katedra: KPSK

Název práce: Implementace algoritmu Fortuna a ANSI X9.31 pro mobilní telefon s OS Android

Zadání:

Věnujte se generátorům pseudonáhodných čísel Fortuna a ANSI X9.31. Popište jejich vlastnosti a shrňte jejich výhody a nevýhody. V praktické části práce implementujte zmíněné dva algoritmy na platformě mobilních telefonů založených na OS Android.

Základní literatura:

Niels Ferguson, Bruce Schneier. Practical Cryptography. John Wiley and Sons, 2003.

Souhlasím se zadáním (podpis, datum)

M. Žák
student(ka)

11.3.2013
vedoucí bakalářské práce

9.4.2013
vedoucí katedry

Prohlášení

Prohlašuji, že tato bakalářská 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.

Martin Žák

Vedoucí práce: Ing. Mgr. Zdeněk Říha, Ph.D.

Poděkování

Mé poděkování patří především vedoucímu práce Ing. Mgr. Zdeňkovi Říhovi, Ph.D. za skvělé vedení a cenné odborné rady. Rovněž bych tímto rád poděkoval svojí partnerce Heleně Grulichové za podporu, která taktéž umožnila vznik této práce. Nakonec děkuji firmě CUTTER Systems spol. s r. o. zejména pak Ing. Lukáši Greplovi a Ing. Martinu Řezáčovi, kterým vděčím za většinu svých zkušeností.

Shrnutí

Práce se zabývá generováním pseudonáhodných čísel na mobilních zařízeních s OS Android. V první části se čtenář seznámí se současnými poznatky z této oblasti. Druhá část se věnuje popisu a implementaci generátorů ANSI X9.31 a Fortuna.

Klíčová slova

Fortuna PRNG, ANSI X9.31, Android, pseudorandom number generator, PRNG, CSPRNG, CSPRBG

Obsah

Úvod

Bezpečnost informačních technologií považujeme za jedno z nejdůležitějších odvětví počítačové vědy. Neoddělitelnou součástí tohoto odvětví je bezpečné generování náhodných čísel (řetězců bitů). Kvalitní náhodná data jsou zapotřebí pro rozličné účely – typicky jako šifrovací klíče, inicializační vektory nebo kryptografické výzvy[1]. Prakticky ale neexistuje kryptografický protokol, který by se bez náhodných čísel obešel.

Pokud mluvíme o „bezpečném“ generování náhodných dat, požadujeme zde několik vlastností. Např. to, že není možné zpětně zjistit ani předvídat vygenerovaná data při kompromitaci vnitřního stavu generátoru, nebo že neexistuje algoritmus, který v polynomiálním čase dokáže předvídat další bit s pravděpodobností větší než 50 %. Běžné platformou nabízené generátory nemusí být pro kryptografické účely vhodné právě proto, že některé tyto vlastnosti nesplňují. Data, která generují, obvykle závisí pouze na prvotní inicializaci jejich stavu skutečně náhodnými daty. V kritických kryptografických aplikacích tyto jednoduché generátory nahrazujeme složitějšími, které používají silná kryptografická primitiva v kombinaci s různými na platformě závislými nebo hardwarovými zdroji entropie[2] pro průběžné změny vnitřního stavu.

S masivním rozvojem mobilních zařízení, kdy je stále více funkcionality běžné na osobních počítačích dostupné také na mobilních platformách, je potřeba i kvalitních generátorů náhodných čísel např. pro účely bankovnictví. V kontrastu s osobními počítači jsou ale pro implementaci generátorů na mobilních zařízeních stále kladena určitá omezení. Oproti klasickým mobilním zařízením již není tolik omezena okamžitá výpočetní síla, ale stále považujeme za vhodné omezit aktivitu procesů zejména kvůli výdrži baterie, která se neustále ukazuje jako největší slabina dnešních mobilních zařízení.

Cílem práce je především implementovat kryptograficky bezpečný generátor Fortuna PRNG[3] na mobilní platformě Android tak, jak je popsán v [1] s ohledem na výše zmíněná omezení. Práce se pak

1. Cryptographic nonce
2. Entropií v tomto případě nazýváme data s určitou mírou náhodnosti získaná např. z pohybu myší
3. PRNG – pseudorandom number generator

zaměřuje především na kvalitní implementaci a optimalizaci v rámci platformy Android. Důležitým cílem je rovněž vytvoření jednoduchého a dobře použitelného aplikačního rozhraní, a to jak pro účel generování náhodných dat, tak pro poskytování entropie generátoru. Práce si rozhodně neklade za cíl detailně prozkoumat zdroje entropie nebo provádět analýzu těchto zdrojů.

V první kapitole se čtenář seznámí se základy generování náhodných čísel a jejich praktické využití se zaměřením na kryptografii. Druhá kapitola pak poskytne stručný náhled na cílovou platformu Android a její kryptografickou knihovnu. V další kapitole je čtenáři představen standardizovaný generátor pseudonáhodných čísel ANSI X9.31, důležité poznatky z jeho implementace a krátký bezpečností rozbor. Poslední kapitola popisuje nejvýznačnější část práce – implementaci generátoru Fortuna autorů Bruce Schneiera a Neilse Fergusona. Nalezneme zde základní rozbor algoritmu s ohledem na předchozí práci autorů, postupy použité při integraci generátoru na cílovou platformu a řešení platformně specifických otázek, které nechali autoři otevřené pro konkrétní implementace. Poslední část kapitoly je věnována měřením, která mají potvrdit použitelnost generátoru.

1 Generování náhodných čísel

Tato práce se věnuje implementaci dvou generátorů náhodných čísel, proto na začátek vysvětlím pojem generátor náhodných čísel a další pojmy s tímto úzce související.

Mějme kostku o šesti stranách, kterou vrhneme. Výsledkem vrhu bude nějaké číslo nabývající hodnot od 1 do 6. Toto číslo můžeme označit za náhodné. Vrhněme kostkou opakovaně. Výsledkem bude sekvence čísel složená z hodnot 1 až 6. Pokud bychom provedli dostatek hodů, zjistíme že všechna čísla budou ve výsledné sekvenci zastoupena zhruba stejně (uvažujeme kvalitní kostku a dobrý vrh). Rozložení těchto hodnot v sekvenci je proto uniformní. Dále pokud vezmeme např. všechny možné dvojce čísel a zjistíme jejich četnost v sekvenci, opět se dostaneme k podobným číslům pro každou dvojici. Tento test mimo jiné demonstruje to, že žádné číslo není statisticky závislé na jeho předchůdci. Existuje celá řada dalších testů, jejichž účelem je odhalit nějakou statistickou závislost v sekvenci čísel. V každém případě jsou výsledky hodu kostkou vždy nezávislé na předešlých hodech, neboť se jedná o nezávislý jev. Definujme tedy generátor náhodných čísel (RNG[1]) jako zařízení, které produkuje čísla v pevně daném rozsahu, kdy každé číslo je statisticky nezávislé na všech ostatních. Zároveň pokud mluvíme o generátoru náhodných čísel, myslíme obvykle generátor s uniformním rozložením, neboť generování náhodnosti s uniformním rozložením se ukazuje jako nejpraktičtější. Typicky pokud chceme získat jiné než uniformní rozložení, nedocílíme toho vytvořením nového generátoru s tímto rozložením, ale aplikací nějaké metody, která nám toto rozložení vytvoří nad již existujícím generátorem s uniformním rozložením.

Z definice generátoru náhodných čísel lze již jednoduše odvodit definici pro informatiku vhodnějšího generátoru náhodných bitů. Řekněme, že máme kostku s osmi stěnami a na nich jsou čísla od 0 do 7. Zřetězením trojic bitů vzniklých převodem hodnot vrhů do binární soustavy, získáme souvislý proud bitů libovolné délky. Definujme tedy generátor náhodných bitů (RBG[2]) jako zařízení nebo algoritmus, který produkuje statisticky nezávislé bity [2].

---

1. Random Number Generator
2. Random Bit Generator

Než se podíváme na samotné metody generování náhodných čísel, je nutné definovat základní veličinu, kterou měříme náhodnost dat - – entropii. V kontextu této práce pak vždy myslíme pojem entropie z pohledu informační teorie, nikoliv z pohledu fyziky (termodynamiky), ačkoliv spolu oba tyto pojmy souvisí. Námi používaná entropie bývá někdy označována jako Shannonova podle Claude E. Shannona, který vyslovil jednu z jejích definicí: Entropie (neurčitost) zdroje je průměrné množství informace, kterou získá příjemce přijetím zprávy z tohoto zdroje. Veličinu můžeme vyjádřit v několika jednotkách, pro nás je však nejpřirozenější jednotkou bit. K pochopení napomůže srovnání dvou zdrojů. První zdroj produkuje coby zprávy znaky nějakého českého textu. Druhý zdroj pak produkuje znaky zcela náhodně. Intuitivně cítíme, že neurčitost prvního zdroje je poněkud nižší než druhého. Typicky pokud z prvního zdroje přijmeme písmeno t, lze jako další písmeno očekávat nějakou samohlásku – nejspíš písmeno e. Nebo pokud jsme již přijali celé slovo, pravděpodobně bude následovat mezera nebo interpunkční znaménko. Přijetím dalšího znaku tedy obvykle nezískáme příliš mnoho informace a na znacích (zprávách) je zřejmá vzájemná statistická závislost. Naopak přijetí znaku z druhého zdroje nám nedává jakýkoliv předpoklad o tom, co bude následovat. Informační hodnota zpráv je proto velmi vysoká, stejně jako entropie samotného zdroje. Entropii zdroje určuje následující vztah:

$$H(X) = -\sum_{i=1}^{n} p(x_i) \log_2 p(x_i)$$

kde $p(x_i)$ značí pravděpodobnost jevu, kdy zdroj vyšle znak x_i. Entropie rovněž značí minimální množství bitů, které jsou potřeba pro zakódování všech zpráv zdroje. Zdroj produkující hodnoty náhodných hodů kostkou má entropii $H(X) \approx 2.6$, což odpovídá minimu 3 bitů pro zakódování 6 hodnot. Na závěr této definice ještě zmíním důležitý fakt, že entropie je zcela subjektivní vzhledem k příjemci zpráv, neboť velmi záleží na informacích, které má příjemce dat k dispozici. Pokud by např. příjemce prvního zdroje z výše uvedeného příkladu měl k dispozici vysílaný text, entropie zdroje by v jeho případě byla nulová.

1.1 Metody generování náhodných sekvencí

Nyní popíši dvě základní metody, které používáme pro generování náhodných sekvencí. Mějme při tom na paměti, že podstatnými požadavky na proces generování jsou jeho plná automatizovanost, spolehlivost a vysoká kvalita samotného výstupu.

První metoda využívá pro generování dat měření našeho okolí, obvykle nějaký konkrétní fyzikální jev. Generátory založené na tomto principu se nazývají generátory skutečně náhodných čísel (TRNG[3]). Za příklad takovýchto generátorů může sloužit webová služba Random.org, která poskytuje napojení na generátor jehož výstup vzniká na základě měření odchylek v amplitudě radiového šumu v atmosféře [3]. Nebo podobná webová služba HotBits, jejíž výstup je založen na měření odchylky v rozpadu radioaktivních prvků [4]. Ovšem existují i hardwarové generátory skutečně náhodných čísel lokalizované přímo na zařízeních, která obvykle měří nějaké fyzikální vlastnosti součástek. Výstup z měření různých jevů nebývá zcela statisticky nezávislý, proto generátory implementují rozličné mechanismy, které entropii extrahují, nebo rozprostírají [5].

Velkou výhodou generátorů skutečně náhodných čísel je jejich stabilně kvalitní výstup s vysokou entropií, ovšem za cenu relativně nízkého výkonu, který nemusí vždy stačit pro požadovaný účel. Rovněž tento způsob generování náhodných dat obvykle vyžaduje kontinuální dodatečnou verifikaci kvality výstupu (obsažené entropie) kvůli případné poruše měřícího zařízení, nebo kvůli detekci pokusu o narušení funkčnosti zařízení. Bohužel tato verifikace je stále velmi náročným úkolem a v dnešní době bývá realizována již dříve zmíněnými statistickými testy, které ale zcela nedokazují skutečnou náhodnost dat. Jako příklad uvádím oblíbenou kolekci testů nazývanou DIEHARD od G. Marsaglia [6].

Především kvůli striktně omezenému výkonu generátorů skutečně náhodných čísel bývá první metoda kombinována s druhou z metod – generováním pseudonáhodných čísel. Generátor pseudonáhodných čísel (PRNG[4]) využívá pro generování náhodných dat algoritmus, který se opakovaně aplikuje na jeho vnitřní stav. Inicializační hodnotě to-

3. True Random Number Generator
4. Pseudorandom number generator

hoto stavu říkáme semínko (seed) a standardně ji získáme právě z generátoru skutečně náhodných dat. Jelikož výstup generátorů je obvykle zcela deterministický, bývají rovněž označovány jako deterministické generátory náhodných bitů (DRBG[5]). Jako typickou ukázkou tohoto principu uvádím již docela starý a prostý generátor LCG[6], který však stále nalezneme v některých programovacích jazycích[7]. Ten data generuje na základě tohoto vztahu:

$$X_{n+1} = aX_n + c \pmod{m}$$

kde a, c i m jsou vhodně zvolené konstanty. Vzhledem k jednoduchosti algoritmu byly v průběhu let ukázány různé statistické problémy výstupu generátoru LCG, avšak v dnešní době existují generátory, jejichž výstup nelze zatím prakticky rozeznat od skutečně náhodných dat. Této vlastnosti generátory dosahují, neboť jsou založeny na nějakém obtížně spočítatelném problému.

Obecně největší výhodou generátorů pseudonáhodných čísel je jejich rychlost, která závisí především na výpočetním výkonu stroje, což umožňuje dobré škálování. Další výhodou může být i samotný determinismus, kdy pro opakované spouštění programu můžeme dosáhnout stejných výsledků bez nutnosti uložit si celý vstup v podobě vygenerovaných dat, ale pouze semínko.

1.2 Aplikace náhodných čísel

Zabývejme se nyní využitím náhodných dat v různých oborech, a to i s ohledem na zmíněné metody. Typickou první aplikací, která člověka napadne, jsou rozličné loterie. Nebo např. webové služby zprostředkovávající karetní hry, pro které potřebujeme namíchat karty. Náhodná data mají své uplatnění i ve státní správě, kde slouží např. pro náhodný výběr vítězů veřejných zakázek. Všechny výše zmíněné aplikace typicky vyžadují vysokou kvalitu náhodných dat a nepříliš vysoký výkon, popř. lze generování dat lokalizovat na specializovaná zařízení. Oproti tomu existují různé aplikace náhodnostních algoritmů nebo simulací, které vyžadují velmi vysoký výkon, někdy i za cenu snížení kva-

5. Deterministic Random Bit Generator
6. Linear congruential generator
7. např. rand() v jazyce C

lity náhodných dat. U těchto aplikací může rovněž být žádanou vlastností i výše zmíněný determinismus. Za příklad nám může posloužit algoritmus RRTs[8] plánující trajektorii robota, který ověřuje bezkolizност náhodných poloh pro dosažení cíle v prostoru. Pro tyto a podobné aplikace jsou generátory pseudonáhodných čísel rozhodně vhodnější. Další velmi významnou a rozsáhlou oblastí použití náhodných čísel je oblast kryptografie.

Náhodná čísla jsou zcela přirozenou součástí současné kryptografie, neboť prakticky neexistuje kryptografický protokol, který by nějaká nepoužíval. Typické využití spočívá v generování kryptografických klíčů, a to jak dlouhodobých, které jsou posléze někde uloženy, tak i krátkodobých použitých ve fázi inicializace protokolů. V tomto směru si musíme uvědomit, že prakticky jakékoliv zařízení, které komunikuje přes otevřené prostředí, vyžaduje schopnost generovat si svá (bezpečná) náhodná data. Můžeme vzít v potaz např. čipovou platební kartu – už jen samotný platební terminál představuje potencionální riziko, protože může být pod vlivem útočníka. Stejné požadavky pak klademe na mobilní zařízení připojené k internetu, komunikující přes bluetooth apod. Je zřejmé, že generování dat musí být lokalizováno přímo na daném zařízení, protože už jen případné navázání komunikace se službou poskytující kvalitní náhodná data nějaká data vyžaduje.

Pro náš účel by nám nejlépe posloužil generátor skutečně náhodných čísel s dostatečným výkonem, aby dokázal v rozumném čase vygenerovat náhodná data o velikostech dnešních kryptografických klíčů. Bohužel těchto požadavků zatím není možné masově dosáhnout, ať z důvodu cílové ceny zařízení nebo technických limitací. Ačkoli na všech zařízeních jsou obvykle dostupná data s nějakou entropií (fluktuace diskové hlavy, šum z nějaké součástky apod.), není extrahované entropie dostatek. Z tohoto důvodu implementujeme do zařízení ještě generátory pseudonáhodných čísel, které pak pracují právě s výše zmíněnými daty. Na generátory pseudonáhodných čísel použitých v kryptografii však klademe další speciální požadavky, kterým věnuji další sekci.

8. Rapidly-exploring Random Trees

1.3 Generátory pseudonáhodných čísel v oblasti kryptografie

Aby mohl být generátor pseudonáhodných čísel použit pro kryptografické účely, musí splňovat několik požadavků. Po jejich splnění lze generátor označit za „kryptograficky bezpečný“ (CSPRBG, CSPRNG[9]). V tomto směru se však různé zdroje na definici (resp. požadavcích) kryptograficky bezpečného generátoru pseudonáhodných čísel zcela neshodují, proto se jeho přesným definováním nebudu přímo zabývat.

První požadavek přirozeně stanovuje požadovanou kvalitu výstupu generátoru. Ta je velmi důležitá, neboť můžeme předpokládat, že aktivní útočník má rovněž přístup ke službám generátoru. Ten by si mohl nechat vygenerovat nějaké větší množství dat a pokusit se z nich odhadnout třeba jen část vnitřního stavu, což by vedlo k zúžení stavového prostoru, který musí jinak útočník projít hrubou silou. Mějme přitom na paměti, že každý správně odhadnutý bit zmenší stavový prostor na polovinu. Požadovanou kvalitu výstupu kryptograficky bezpečných generátorů definujeme pomocí „testu dalšího bitu“[10] – Generátor obstojí v „testu dalšího bitu“, pokud neexistuje algoritmus, který by v polynomiálním časové složitosti dokázal na základě již vygenerované sekvence délky l, předpovědět hodnotu bitu $l + 1$ s pravděpodobností větší než $\frac{1}{2}$ [5]. Zjednodušeně řečeno nesmí být možné v dohledné době (mnoha let) odhadnout výstup generátoru s lepším výsledkem, než kdybychom jej hádali. Tato vlastnost lze pro praktické použití docela dobře prokázat vzhledem k použitým kryptografickým primitivům. U nich je buď dokázána neexistence algoritmů s polynomiální časovou složitostí, nebo takový algoritmus alespoň není znám. V roce 1982 bylo navíc dokázáno, že pokud generátor splní „test dalšího bitu“, projde i všemi statistickými testy[11]. Avšak statistické testy jsou pro ověření funkčnosti generátorů používány, neboť je vyžadují některé standardy, nebo protože mohou vést k odhalení nějaké implementační chyby.

Druhý požadavek směřuje spíš na praktickou stránku použití generátoru, kdy předpokládáme, že vnitřní stav byl odhalen – třeba kvůli nějaké implementační chybě v operačním systému apod. Po-

9. Cryptographically secure (strong) pseudorandom bit (number) generator
10. next-bit test
11. Andrew Chi-Chih Yao (1982)

kud už k tomuto dojde, považujeme za naprosto nutné, aby nebyla kompromitována i dříve vygenerovaná data. Jejich získání by mohlo mít skutečně fatální důsledky na dříve provedené bezpečnostní protokoly, neboť bychom získali např. použitý komunikační klíč. Naštěstí lze tohoto požadavku poměrně snadno dosáhnout pravidelnou aktualizací vnitřního stavu např. aplikováním jednosměrných funkcí. Tento požadavek někdy nazýváme dopředná bezpečnost.

Poslední požadavek je podle mého názoru nejobtížnější splnit. Uplatníme jej rovněž při kompromitaci vnitřního stavu a to tak, že pokud je již vnitřní stav kompromitován, nesmí nám jeho znalost umožnit získat jeho hodnoty v budoucnu. Útoku, kdy jednou získáme vnitřní stav a pak jej již odhalujeme znovu a znovu (např. hrubou silou), říkáme permanentní kompromitace [7]. Jedinou ochranou proti tomuto útoku je pravidelné obohacování vnitřního stavu entropií. Problém ale spočívá v tom, že množství entropie nelze přesně určit a záleží vždy na daných okolnostech a případné útočníkově znalosti zdrojů entropie. Z tohoto důvodu není nikdy zcela jasné, do jaké míry generátor požadavek splňuje. Tento požadavek někdy nazýváme zpětná bezpečnost.

V další kapitole se budu krátce věnovat cílové platformě pro implementace zadaných generátorů.

2 OS Android

Díky otevřenosti operačního systému Android a rovněž jeho velké popularitě jsou informace o architektuře a způsobu vývoje na této platformě velmi dobře dostupné. Proto se zaměřím pouze na nutné minimum přímo související s touto prací – kryptografické API[1].

Platforma Android staví rozhraní pro vývojáře na programovacím jazyku Java a spolu s ním rovněž převzala i značnou část specifikace základního API zvané Java SE[2]. Součástí specifikace je i balík *java.security*, který pomocí abstraktních tříd a rozhraní stanovuje unifikovaný způsob provádění různých kryptografických operací jako např. symetrické nebo asymetrické šifrování, práci s certifikáty apod. [8] Avšak jedná se pouze o abstraktní vrstvu, pro kterou musí každá platforma poskytnout vlastní implementaci. Pro zjištění konkrétní implementace platformy Android jsem musel nahlédnout do zdrojových kódů OS Android dostupných na [9]. Zjistil jsem, že OS Android používá pro implementaci různých tříd a rozhraní z balíku *java.security* knihovnu *Bouncy Castle*. Tato kryptografická knihovna se těší velké oblibě mezi vývojáři a je již léty ověřena, proto se na ni lze dobře spolehnout.

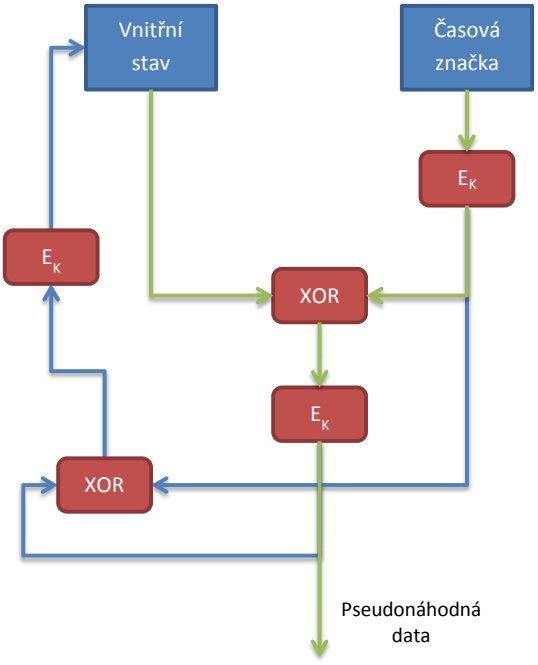
1. API – aplikační rozhraní (Application Programming Interface)
2. Java Standard Edition

3 ANSI X9.31

Tuto kapitol věnuji generátoru pseudonáhodných čísel popsaného ve Apendixu A.2.4 standardu ANSI X9.31, který udává postupy pro digitální podepisování dokumentů ve finančních institucích. Svou implementaci jsem však založil na standardu instituce NIST, který poskytuje přepis algoritmu zdarma [10].

3.1 Popis

Generátor ANSI X9.31 je formálně popsán ve výše zmíněném standardu, já jej popíši méně formálně s pomocí obrázku 3.1.



Obrázek 3.1: ANSI X9.31

Celý generátor se sestává z vnitřního stavu a šifrovacího klíče prvotně inicializovaného bezpečnými náhodnými daty (semínkem), obě tyto hodnoty musejí být využity pouze pro jednu instanci generátoru a

drženy v tajnosti. Poté během generování dat využívá generátor operace exkluzivního součtu (XOR) a šifrování symetrickou šifrou v režimu ECB (E_k), kde standard umožňuje použití šifry AES[1], nebo 3DES. Výběr algoritmu potom určuje bitové velikosti hodnot ve stavu generátoru a během jeho činnosti (hodnoty odpovídají velikosti bloku šifry). V případě použití šifry AES má vnitřní stav velikost 128 bitů a klíč jednu z doporučených hodnot pro šifru AES – tj. 128, 192 nebo 256 bitů. Pro šifru 3DES je vnitřní stav menší – pouze 64 bitů, přičemž klíč má velikost 168 bitů.

Proces generování jednoho bloku pseudonáhodných dat jsem rozdělil na dvě části: samotné generování dat (zelený tok) a následnou aktualizaci vnitřního stavu (modrý tok). Do procesu vstupuje z okolního prostředí pouze jediná hodnota – časová značka, jejíž velikost, resp. přesnost musí odpovídat velikosti bloků použité šifry. Výstup generátoru vzniká spojením a zašifrováním vnitřního stavu a již jednou zašifrované časové značky. Nový vnitřní stav pak vznikne opětovným spojením a zašifrováním výstupu generátoru a zašifrované časové značky. Tento proces se opakuje neustále dokola, dokud není vygenerováno dostatek pseudonáhodných dat.

3.2 Bezpečnost

Ačkoliv byl generátor ANSI X9.31 navrhován jako kryptograficky bezpečný, není za něj považován (alespoň ne podle všech definic), protože není dostatečně schopný zotavení z případné kompromitace stavu [7]. Navíc pokud skutečně nastane kompromitace stavu, jsou nepřímo ohrožena i samotná data, která již generátor vygeneroval. Problémem je, že během činnosti generátoru, není do jeho stavu dodán dostatek entropie. Jediná entropie dodaná během generování přichází v podobě časové značky. Bohužel množství této entropie je značně diskutabilní, neboť časovou značku lze do jisté míry odhadnout a zbylé malé množství entropie již otevírá možnost průchodu stavového prostoru hrubou silou [11][12]. Bezpečnost celého generátoru tedy závisí především na entropii dodané během jeho inicializace v podobě semínka a klíče pro symetrickou šifru.

1. Advanced Encryption Standard – šifra vybraná na základě soutěže vyhlášené standardizační institucí NIST

3.3 Implementace

Vzhledem k tomu, že generátor vyžaduje pouze interakci s běžným API poskytovaným OS Android a samotnou aplikací, implementoval jsem jej pouze jako knihovnu, kterou si každá aplikace ponese sama a v případě potřeby instancuje. Knihovna obsahuje jedinou třídu *Generator* a výjimku *GeneratorException* pro případ nevhodného zacházení nebo nějakých chyb.

Pro získání prvotního semínka a šifrovacího klíče implementace využívá standardního unixového mechanismu, kterým je speciální soubor /dev/urandom, jehož čtením opatříme semínko. Stejný mechanismus používá i např. již v OS Android přítomný generátor SHA1PRNG [13].

Další implementační záležitostí je získávání aktuální časové značky s vhodnou přesností, tj. v našem případě 128bitovou. Jak bylo již zmíněno, množství entropie obsažené v této značce ovlivňuje bezpečnost celého algoritmu. A zde platí, že čím přesnější časovou značku máme, tím více entropie obsahuje, neboť o to větší stavový prostor by musel útočník projít hrubou silou. Vzhledem k možnostem dnešních běžných počítačů, ale není možné měřit čas s takovouto granularitou, abychom získali 128bitovou časovou značku. To, co nám v tomto případě nabízí OS Android, jsou dva druhy časových značek. První z nich reprezentuje aktuální čas vyjádřený v milisekundách od 1.1.1970 00:00:00 UTC a uložený v 64 bitovém čísle. Druhá vyjadřuje čas od nějakého pevně zvoleného okamžiku s maximální přesností dostupnou na daném zařízení. Tento čas systém poskytuje v nanosekundách uložených v rámci 64 bitového čísla. Na přístroji ZTE Blade 2, který byl použit pro testování, byla tato přesnost experimentálně určena na $\frac{1}{6}$ mikrosekundy. Jelikož obě tyto značky mohou obsahovat jisté množství entropie, první pro svoje pevné ukotvení v rámci delšího časového úseku a druhá pro svoji přesnost, rozhodl jsem se tyto značky v implementaci spojit a to tím způsobem, že každá 64bitová značka tvoří jednu část 128bitové hodnoty. Abych navíc zajistil, že každá z těchto hodnot bude pro jednu iteraci generátoru odlišná, jak to uvádí standard, je výsledná hodnota zkombinována s 64bitovým počítadlem, které po každé iteraci zvýší svoji hodnotu. Kombinaci provádí operace XOR.

3.4 Shrnutí

Vzhledem k problémům s bezpečností generátoru ANSI X9.31 není příliš vhodné používat generátor jakkoliv dlouhodobě, nebo dokonce jako službu v OS, neboť tím můžeme zvýšit šanci na úspěšné prolomení a způsobené škody by mohly být značné. Avšak pokud získáme kvalitní bezpečné semínko z dobrého ale např. pomalého zdroje entropie, může být tento generátor použitelný pro okamžité vygenerování nepříliš velkého množství dat (v jednotkách Kb). Po ukončení práce s generátorem je však nutné důsledně zničit vnitřní stav a šifrovací klíč.

Tato implementace získala prvotní semínko ze služby poskytované OS přes speciální soubor /dev/urandom. Potom, co jsem se seznámil se základním nejen kryptografickým rozhraním nabízeným v rámci OS Android, se budu v další kapitole věnovat implementaci obdobné služby v podobě generátoru Fortuna.

4 Fortuna PRNG

V této kapitole se zabývám popisem a implementací generátoru pseudonáhodných čísel Fortuna popsaného v [1, kapitola 9]. Oproti generátoru ANSI X9.31 by měl přinést mnohem větší míru bezpečnosti, neboť je všeobecně považovaný za kryptograficky bezpečný. Pro své vlastnosti bývá generátor Fortuna obvykle použit jako dlouhodobě běžící poskytovatel náhodných dat.

4.1 Popis a bezpečnost

Generátor Fortuna je následovníkem úspěšného generátoru Yarrow od podobného kolektivu autorů. Z toho důvodu vysvětlím většinu návrhových principů na tomto předchůdci. Dalším důvodem je i to, že v případě popisu generátoru Yarrow se autoři více zaměřují na kryptografickou stránku věci, kdežto v případě generátoru Fortuna se již autoři soustředili spíše na stránku implementační.

Jedna z hlavních myšlenek obou generátorů je předpoklad, že pokud máme bezpečná kryptografická primitiva a dostatek počáteční entropie (což je základem všech generátorů pseudonáhodných čísel) jsme schopni generovat kvalitní náhodná data, která nelze se současnými výpočetními prostředky v prakticky blízké době rozeznat od skutečně náhodných dat. Bezpečnost generátoru tedy přímo závisí na bezpečnosti použitých kryptografických primitiv. V tomto směru se navíc autoři nevymezují na konkrétní algoritmy, naopak uvádějí koncepty generátorů obecně, a to i vzhledem k délce vstupů a výstupů použitých algoritmů. Autoři však uvádějí doporučení, kterým je v případě generátoru Yarrow kombinace jednosměrné funkce SHA1 a šifry 3DES (přičemž v budoucnu počítají s šifrou AES[1]) [7, str. 8] a pro generátor Fortuna jednosměrná funkce SHA-256 a v té době již vybraná šifra AES (případně šifry Serpent, nebo Twofish) [1, str. 143].

Motivací k vytvoření speciálního konceptu generátorů Yarrow a Fortuna byla pro autory skutečnost, že mnoho generátorů má tu vlastnost, kdy v případě kompromitace jejich vnitřního stavu již nejsou

1. Advanced Encryption Standard – bloková symetrická šifra doporučená standardizační institucí NIST, přičemž výběr probíhá na základě několik let trvající soutěže

nadále schopny generovat nepředvídatelná data, nebo alespoň ne dostatečně brzy. Za jednu z nejčastějších příčin získání vnitřního stavu autoři považují nedostatek entropie v počátečním stavu, což pak otevírá možnost vyčerpávajícímu prohledávání jeho stavového prostoru. Jako další významná rizika vedoucí ke kompromitaci nebo narušení vnitřních stavů autoři označují nevhodnou manipulaci se soubory obsahujícími semínko či klíče, nebo ovládnutí některých zdrojů entropie [7, str. 3-4].

Z tohoto důvodu autoři využívají entropii získanou z okolního prostředí ani ne tak jako náhodná data samotná (nebo pro jejich přímou tvorbu), ale pro obohacení vnitřního stavu generátorů. Tento koncept by tedy měl zajistit, že i v případě odhalení vnitřního stavu generátoru dojde díky příchozí entropii dříve či později k obnovení bezpečnosti generátoru. Čas, za který k tomu dojde, pak závisí na míře příchozí entropie. Na druhou stranu dojde-li k ovládnutí všech zdrojů entropie útočníkem a nedojde k získání vnitřního stavu generátoru, generátor je stále bezpečný – tedy alespoň do té míry, jako např. ANSI X9.31[7, str. 5]. Schopností zotavit se z kompromitace stavu generátor Fortuna splňuje jeden z požadavků na kryptograficky bezpečný PRNG.

Nyní se podívejme na komponenty obou generátorů. První komponenta se nazývá Generátor. Tuto komponentu lze považovat za jádro celku, neboť produkuje náhodná data. Sestává se z blokové šifry v módu ECB, jejího klíče a počítadla. Ve své podstatě se jedná o mód CTR blokové šifry, ale jsou zde drobné rozdíly. Klíč šifry je nejprve inicializován (tím se budu zabývat níže v sekci 4.1) a postupně obohacován nasbíranou entropií. Během generování náhodných dat jsou vždy generována nějaká data navíc, která jsou posléze použita jako nový klíč (starý se zahodí). Tímto krokem je zajištěna ochrana proti útoku Backtracking, kdy je útočník schopen při odhalení vnitřního stavu získat dříve vygenerovaná data [1, str. 143-147]. Rezistence vůči tomuto útoku je dalším z požadavků na kryptograficky bezpečný PRNG.

V tom, kdy má být vygenerovaný nový klíč, se již generátory liší. Zatímco generátor Yarrow toto prováděl po konstantním počtu vygenerovaných bloků (doporučeno bylo 10) [7, str. 9], generátor Fortuna toto provádí po každém požadavku na vygenerování náhodných dat. Tímto krokem autoři omezili počet dříve vygenerovaných bloků získaných útokem Backtracking z původních 10 na 0. I přesto však autoři v případě generátoru Fortuna omezili počet vygenerovaných bloků, než dojde ke změně klíče. Důvodem bylo zamezení projevu sta-

tistické vlastnosti, kdy při generování dat pomocí blokové šifry v režimu CTR data nikdy neobsahují stejné bloky. V případě generování většího množství dat by pak nenalezení stejných bloků ukázalo na to, že výstup není zcela náhodný. Autoři nastavili toto omezení na 2^{20}, přičemž v datech této délky je pravděpodobnost nalezení stejných bloků rovna 2^{-97}. Tak nízká pravděpodobnost již k odhalení vyžaduje provést velmi vysoký počet požadavků, což autoři považují za dobrý kompromis mezi výkonem vynaloženým na generování nových klíčů a statistickými vlastnostmi výstupu [1, str. 143].

Další z komponent je akumulátor entropie (Entropy accumulator), který koncentruje entropii a pravidelně obohacuje stav generátoru. Akumulátory obou generátorů se skládají ze samostatných zásobníků entropie. Zásobník představuje hashovací funkci, která kontinuálně zpracovává vzorky z různých zdrojů. Když dojde k použití zásobníku pro obohacení stavu generátoru, vezme se výsledek funkce a spojí se s vnitřním stavem funkcí XOR. Komponenta akumulátoru představuje největší rozdíl mezi oběma generátory. V případě generátoru Yarrow existují dva zásobníky – rychlý a pomalý. Rychlý zásobník má být pro obohacení stavu použit častěji, tak aby při kompromitaci stavu došlo k co nejrychlejšímu zotavení, zatímco pomalý zásobník je použit méně často, tak aby zaručil obohacení stavu o silné semínko obsahující hodně entropie. Vzorky jednotlivých zdrojů entropie jsou do zásobníků umísťovány střídavě [7, str. 6].

Nejpalčivější otázka celého generátoru Yarrow zní: kdy má ale vlastně dojít k obohacení stavu z jednotlivých zásobníků? Odpověď je vskutku jednoduchá: ve chvíli kdy je již míra entropie v zásobníku větší než nějaká prahová hodnota. V tom podle autorů spočívá největší slabina generátoru Yarrow – akumulátor se snaží množství entropie odhadovat [1, str. 150]. Problém nastane ve chvíli, kdy je vnitřní stav obohacován příliš často (např. kvůli zahlcení zásobníků falešnými vzorky), nebo naopak pokud jsme v odhadech příliš opatrní a k obohacení dochází zřídka. V prvním případě do generátoru nikdy nedodáme dostatečnou dávku entropie tak, aby se mohl zotavit z kompromitace stavu. V případě druhém zase nedojde k zotavení dostatečně rychle. Právě tuto slabinu generátoru Yarrow řeší generátor Fortuna. Akumulátor generátoru Fortuna již neobsahuje pouze dva zásobníky, ale hned několik (doporučení udává 32). Vnitřní stav pak obohacuje kombinací několika zásobníků tak, že první zásobník je použit pokaždé,

druhý po každé druhé, třetí po každé čtvrté[2] atd. Za povšimnutí stojí, že tím autoři vytvořili celou škálu zásobníků s rozdílnými „rychlostmi". K obohacení dojde vždy, když první zásobník nasbírá určité množství dat (nikoli entropie). Vzorky jsou do zásobníků distribuovány cyklicky. Tento koncept zaručí, že pokud systém poskytne dostatek útočníkovi neznámé entropie, k zotavení dojde vždy [1, str. 148-149].

Pro zotavení je potřeba určitá míra entropie (autoři uvádějí 128 bitů), pokud však zásobník P_x tuto entropii neobsahuje, pak možná zásobník P_{x+1} ano, neboť jí obsahuje dvakrát víc než P_x. Toto se opakuje stále dál, až po zásobník P_{31}. Avšak aby ani poslední zásobník neobsahoval dostatek entropie, musel by útočník donutit (požadavky na generování dat, vkládání falešných vzorků) generátor k provedení 2^{32} obohacení stavu. A to navíc dříve, než všechny zdroje, o kterých útočník nemá žádnou znalost, vygenerují 32krát 128 bitů entropie. Ačkoliv je toto nepravděpodobné, autoři přidali časové omezení na periodu obohacení stavu, které je nastaveno na 100ms. Toto drobné pozdržení obohacování zajistí, že poslední zásobník bude použit po 13 letech, což zcela vylučuje tento druh útoku [1, str. 150].

Autoři se v popisu ještě věnují tomu, kdy dojde k zotavení z předchozí kompromitace. Já se ale v této práci spokojím s tvrzením, že tato doba je závislá na míře entropie získané ze zdrojů neznámých útočníkovi.

Poslední komponenta obou generátorů se stará o soubor se semínkem (Seed File). Tato komponenta vznikla čistě kvůli tomu, že při startu počítače není okamžitě sesbíráno dostatek entropie, aby mohlo začít bezpečné generování náhodných dat. Z tohoto důvodu vygenerujeme malé množství dat navíc a následně je zapíšeme do souboru dostupného pouze generátoru. Při startu pak generátor obnoví svůj vnitřní stav z tohoto souboru. Okamžitě po vyčtení musí být soubor se semínkem vygenerován znova, a to ještě dříve než bude generátor dostupný komukoliv jinému. Toto se děje proto, aby si útočník hned po startu generátoru nevygeneroval nějaká data a nevypnul počítač ještě před zapsáním souboru se semínkem. Další obnovení stavu by pak totiž proběhlo ze stejného obsahu souboru a vygenerovaná data by byla stejná nebo velmi podobná jako útočníkova, ovšem v rukou nic netušícího uživatele [1, str. 155-158].

2. 2^x kde x je číslo zásobníku počínajíc 0

Na závěr této sekce ještě uvedu některé již existující implementace obou generátorů. Generátor Yarrow stojí za speciálním souborem /dev/random v případě unixových systémů FreeBSD [14] a Mac OS X [15]. Generátor Fortuna je implementován pro OS Windows [16], jako knihovna v jazyce JavaScript [17] a pro mobilní telefony Symbian jako součást jedné diplomové práce [12].

4.2 Implementace

4.2.1 Jádro

Jako první se budu zabývat jádrem samotného generátoru Fortuna. Celý generátor je jeho autory detailně popsán formou pseudokódu, a to i včetně ošetření argumentů jednotlivých funkcí. Při tvorbě jádra bylo proto mojí snahou co nejvíce se držet původního popisu autorů. Výsledkem je vytvoření tří základních tříd. Třída *Generator*, která přímo reflektuje komponentu generátor popsanou výše. Dále třída *FortunaPRNG*, která zaobaluje celé jádro a instancuje generátor. Tato třída plní částečně roli komponenty Accumulator – stará se o pravidelné obohacování stavu generátoru ze zásobníků entropie. Pro tyto zásobníky jsem však vytvořil samostatnou třídu *EntropyPool*. Ta zajišťuje kontinuální výpočet hashovací funkce a bufferování entropie tak, jak je to doporučeno autory [1, str. 151-152]. Roli správy souboru se semínkem (jeho přečtení a přepsání) jsem přenechal třídě *FortunaPRNG*.

Autoři v některých místech ukončují volání funkcí chybou. Pro tyto případy byla zděděním třídy *Exception* vytvořena výjimka *FortunaException*, kterou metody jádra vyhazují. Při implementaci jsem ale narazil na místa, kde může dojít rovněž ke vzniku chyby. Tímto místem je typicky zápis semínka do souboru nebo některé kryptografické operace. Přestože by k těmto situacím nemělo docházet (především pokud jde o kryptografické operace), stále by bylo vhodné na tyto situace reagovat odpovídajícím způsobem. Autoři v tomto směru neposkytují žádná doporučení. Vzhledem k povaze aplikace coby bezpečnostního modulu volím řešení v podobě implementace mechanizmu zneplatnění generátoru, který v případě takovéto neočekávané chyby znemožní generování dalších dat a zabrání tak případnému snížení bezpečnosti.

4.2.2 Integrace do OS Android

V této části vývoje jsem se zabýval začleněním jádra generátoru Fortuna do běhového prostředí poskytovaného OS Android. Dalším z cílů také bylo vytvoření knihovny, kterou lze jednoduše použít v libovolném projektu na platformě Android za účelem generování náhodných dat, popř. za účelem implementace dalších zdrojů entropie.

Oproti generátoru ANSI X9.31 generátor Fortuna nestačí pouze inicializovat ve chvíli, kdy potřebujeme generovat náhodná data. Generátor Fortuna vyžaduje dlouhodobý běh (ideálně jako služba systému) tak, aby byla postupně sbírána entropie z událostí v OS a obohacován vnitřní stav. Tato funkcionalita je na OS Android realizována pomocí služeb (Services).

Služba (Service) umožňuje nějakému kódu běžet dlouhodobě ve velké míře bez rizika ukončení kvůli nedostatku prostředků OS. Služba pak může běžet buď v dlouhodobém režimu, tj. dokud není explicitně zastavena (třeba i sama), nebo v režimu kdy jí OS sám zastaví ve chvíli, když již neexistují žádné úlohy, které by mohla vykonat [18]. V mém případě jsem využil první možnosti.

Komunikace aplikace v popředí se službou pak závisí do jisté míry na tom, o jaký typ služby se jedná. Služba bývá nejčastěji lokální (Local Service), potom totiž běží i ve stejném procesu jako zbytek aplikace a komunikace probíhá jednoduše voláním objektu služby resp. jejího rozhraní. Já ale chci zprostředkovat generování čísel i dalším aplikacím, zvolil jsem proto službu vzdálenou (Remote Service) [19].

V tomto případě pak komunikace probíhá pomocí IPC[3]. Zde lze použít dva koncepty nabízené OS Android. První z těchto konceptů je použití třídy *Messenger*. Tato třída umožňuje zasílání zpráv (Message). Ve zprávě jsou nám k dispozici dva argumenty typu integer a jeden argument typu objekt, který však musí implementovat rozhraní Parcelable, což umožní jeho binární serializaci. Další data mohou být doplněna pomocí třídy *Bundle*, která je používaná napříč celým OS Android pro serializaci dat. Tento koncept je výhodný v tom, že požadavky ve formě zpráv jsou doručovány postupně, a pokud nechceme jinak, i ve stejném vlákně, ve kterém běží celý zbytek aplikace – tedy není třeba řešit vláknovou bezpečnost [19]. Zpočátku se toto řešení pro generátor Fortuna může jevit jako vhodné, protože samotný generátor

3. Meziprocesová komunikace (Inter-process communication)

vyžaduje striktně sekvenční přístup. Pokud ale chceme maximalizovat výkon generátoru jako celku, je důležité, aby výpočty hashovacích funkcí v zásobnících entropie běžely vůči sobě paralelně, stejně jako to zmiňují autoři [1, str. 155]. Vzhledem k tomu, že většina dnešních mobilních zařízení mívá více než jedno jádro procesoru, jeví se jako vhodnější druhý koncept, kterým je použití rozhraní napsaného v jazyce AIDL[4].

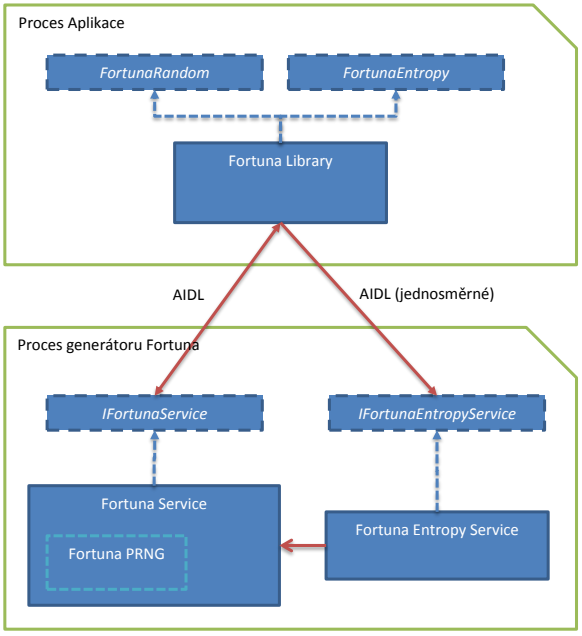
V případě specifikace rozhraní pomocí jazyka AIDL jsou implementace metod typicky volány ve vláknech threadpoolu, tedy zcela paralelně. To na jednu stranu přináší nutnost implementace generátoru jako vláknově bezpečného, na druhou stranu tím získáme možnost paralelního přístupu k jednotlivým zásobníkům entropie a threadpoolová vlákna použijeme pro výpočet hashovacích funkcí v zásobnících. Specifikace rozhraní v jazyce AIDL je poměrně snadná – rozhraní se implementuje velmi podobně jako v jazyce Java. Metody pak mohou mít argumenty a návratovou hodnotu pouze základních typů nebo argumenty, které implementují výše zmíněné rozhraní Parcelable [20].

Mimo základní definici rozhraní jazyk také umožňuje definovat rozhraní jako jednosměrné (oneway). Označením rozhraní jako jednosměrného se volání tohoto rozhraní z cizí aplikace stává neblokující, rozhraní pak ovšem může mít pouze metody bez návratového typu a pouze s vstupními parametry [20]. Takto specifikované rozhraní by bylo vhodné pro metodu přidávající data do zásobníků entropie, neboť má pouze vstupní parametry a výsledek volání není důležitý [1, str. 152]. Bohužel podle dokumentace API neumožňuje, aby jedna služba měla více rozhraní [21], i přestože některé zdroje uvádí, že to lze. Z tohoto důvodu jsem rozhodl do aplikace přidat ještě jednu službu, která poběží ve stejném procesu jako služba hlavní, za účelem poskytnutí jednosměrného rozhraní pro přidání entropie do zásobníků. Tato služba je potom s hlavní službou ve vztahu služby lokální, tudíž komunikace je přímá. V další fázi vývoje byla tato služba použita pro sběr entropie z některých zdrojů.

V tuto chvíli se stačí pouze připojit ke službám a využít jejich rozhraní, ale ve snaze ušetřit tuto práci vývojářům aplikací jsou tato rozhraní zabalena do knihovny, která vytváří jednoduché rozhraní pro práci s generátorem Fortuna. Po nainstalování hlavního balíku obsa-

4. Android Interface Definition Language

hujícího služby stačí jen referencovat knihovnu a instancovat jednu třídu. Celý návrh ilustruje obrázek 4.1.



Obrázek 4.1: Architektura aplikace

4.2.3 Soubor se semínkem

Na komponentu správy souboru se semínkem je autory kladeno několik požadavků. Nejdůležitějším z těchto požadavků je to, aby byl soubor přístupný výlučně generátoru a nikomu jinému [1, str. 155-157]. V tomto směru nám OS Android vyšel vstříc – vedle několika dalších řešení zaměřených na různé účely OS Android poskytuje možnost nazývanou Internal Storage. Data (resp. soubory) ukládaná pomocí Internal Storage jsou umístěna do vnitřní paměti mobilního zařízení a ve výchozím použití čitelná pouze aplikaci, která data zapsala. To vylučuje i přístup uživatele k těmto datům [22]. Toto zabezpečení lze překonat pouze tzv. „rootnutím“ mobilního zařízení, popř. zásahem do hardware, čímž se však v této práci nebudu zabývat. V případě odstranění aplikace jsou tato data rovněž smazána.

Dále autoři rozebírají atomicitu operace zápisu dat do souboru se semínkem. V tomto směru je velmi důležitým požadavkem to, aby byla data vygenerovaná v rámci obnovení stavu generátoru ze souboru skutečně zapsána na fyzické medium [1, str. 158]. Mé implementaci za účelem splnění tohoto požadavku nezbývá, než se spolehnout na bezchybnou implementaci operace *sync* třídy *FileDescriptor*, kterou získám ze třídy *FileOutputStream*.

4.2.4 První spuštění

V případě prvního spuštění generátoru Fortuna na mobilním zařízení není přítomen žádný soubor se semínkem, což autoři považují za velké bezpečnostní riziko. Důvodem je skutečnost, že by generátor mohl být použit pro generování např. kryptografických klíčů, ještě než byl do zásobníků sesbírán dostatek entropie. Za těchto okolností autoři doporučují poskytnutí souboru se semínkem vygenerovaným na jiném stroji, nebo vytvoření souboru interakcí s uživatelem během instalace. V každém případě ale musí být prvotní entropie poskytnuta [1, str. 159].

První z autory zmiňovaných možností není na mobilních zařízeních příliš realizovatelná, proto jsem se rozhodl odvodit řešení od možnosti druhé. Instalace aplikací na mobilních platformách je maximálně zjednodušená, a na spouštění nějakých procedur zde není prostor, proto bývá zvykem prvotní konfiguraci nechat na spuštění aplikace. V tomto případě se ale tato aplikace nachází mimo naše kompetence a i když by to bylo možné, nebylo by vhodné do jejího běhu jakkoliv zasahovat.

Pro tento účel byla do rozhraní knihovny přidána metoda, kterou aplikace používající generátor zavolá, když to její autor považuje za vhodné. Zavoláním metody dojde ke zkontrolování, zda-li byl generátorem správně načten soubor se semínkem, v opačném případě dojde k zobrazení aktivity, která vyzve uživatele k pohybu prstem po obrazovce dokud není nasbíráno dostatek událostí. Během pohybu prstu po obrazovce aktivita sbírá data ze senzorů osvětlení a gravitačního zrychlení. Data z událostí jsou posléze komprimována funkcí SHA-512 a předána generátoru coby semínko. Zavolání této funkce je povinné a pokud se tak nestane, generátor odmítne generovat data.

4.2.5 Zdroje Entropie

Tato práce si za cíl neklade hledání a implementaci vhodných zdrojů entropie – náročnost tohoto úkolu značně převyšuje její rozsah. Nicméně aplikační rozhraní knihovny je na implementaci zdrojů dobře připraveno. Pro účely testování však bylo nutné nějaké zdroje implementovat. Jako testovací zdroje jsem vybral zdroje, které jsou přítomny na drtivé většině zařízeních s OS Android a jejich implementace není nijak náročná. Těmito zdroji jsou senzor osvětlení a akcelerometry. Aby nebyla výdrž zařízení nikterak ovlivněna, služba data sbírá pouze pokud je obrazovka telefonu aktivní, což typicky znamená, že jsou senzory stejně zapnuty. Pokud obrazovka aktivní není, nelze stejně očekávat, že by tyto zdroje přinesly příliš mnoho entropie, neboť zařízení např. leží na stole a není používáno. Mimo jiné další testování ukázalo, že API přestane data ze senzorů samo poskytovat, pokud je obrazovka vypnuta.

4.2.6 Výkon

V případě implementace knihovny používané jinými aplikacemi považuji za vhodné provést testování výkonu celé implementace a detekovat potencionální místa, kde dochází k plýtvání výkonem. Navíc se zde nabízí srovnání s již zmíněnou implementací na telefonech s OS Symbian [12].

Výkon jsem měřil ve dvou situacích. Nejprve v rámci stejného procesu a poté v rámci jiného procesu. Rozdíl naměřených hodnot nám poskytne představu o zpomalení způsobeném meziprocesovou komunikací. Nejvíce vypovídající je hodnota naměřená z jiného procesu, neboť ostatní aplikace poběží právě v jiných procesech. Naměření hodnoty v podobě množství vygenerovaných dat ilustruje tabulka 4.1.

	Minimum	Maximum	Průměr	Střední hodnota
Jiný proces	40 kB/s	1080 kB/s	935 kB/s	980 kB/s
Stejný proces	40 kB/s	1435 kB/s	1350 kB/s	1400 kB/s

Tabulka 4.1: Výkon generátoru

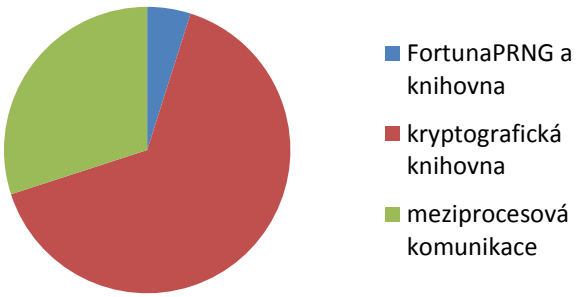
Výkonnost byla měřena na mobilním telefonu: ZTE Blade 2 OS Android v. 2.3.3, procesor: Qualcomm MSM7227A, ARM Cortex-A5,

Minimum	Maximum	Průměr	Střední hodnota
400 kB/s	1000 kB/s	848 kB/s	1000 kB/s

Tabulka 4.2: Výkon generátoru na OS Symbian [12]

1 GHz, 32 bit. Tento mobilní telefon lze v době psaní práce označit za průměrný. Oproti tomu výsledky implementace na OS Symbian byly získány na telefonu Nokia N73 s procesorem: Dual CPU ARM 9, 220 MHz, 32 bit. Velmi mě překvapilo, že výsledky na telefonech, které od sebe dělí téměř 6 let, jsou velmi srovnatelné.

Za účelem zjištění výkonu na jednotlivých vrstvách jsem provedl profilování metody generování náhodných dat. Výsledky této analýzy ukázaly, že 93 % procent výpočetního času aplikace stráví v metodě *doFinal* třídy *Cipher*, která tvoří API kryptografické knihovny. Tedy pouhých 7 % času zbývá na aplikaci a samotné jádro generátoru Fortuna. To je podle mého názoru dobrý výsledek z pohledu generátoru a aplikace, bohužel ale nedává prakticky žádný prostor pro jakékoliv optimalizace. Profilování je možné provést pouze v rámci jednoho procesu, proto musíme do celkového času započítat ještě zpomalení způsobené meziprocesovou komunikací.



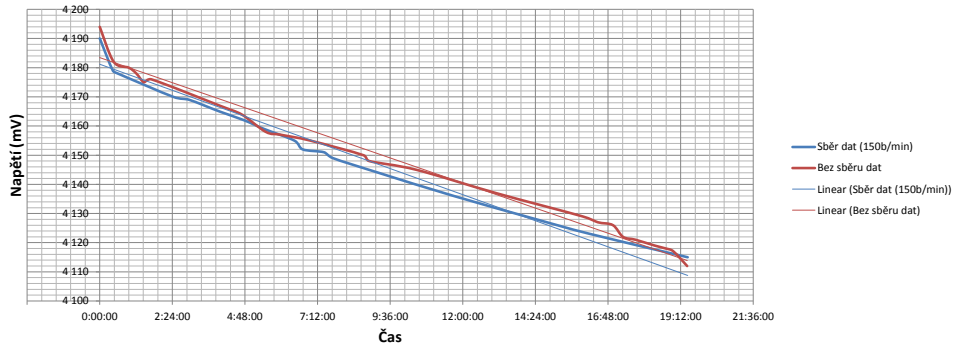
Obrázek 4.2: Poměr výpočetního času generování dat

Name	Incl Cpu Time %	Incl Cpu Time
0 (toplevel)	100,0%	30,661
1 cz/muni/fi/Fortuna/Context/FortunaService$1.GenerateRandomData (I)[B	99,3%	30,442
2 cz/muni/fi/Fortuna/Algorithm/FortunaPRNG.GenerateData (I)[B	99,2%	30,422
3 cz/muni/fi/Fortuna/Algorithm/FortunaPRNG.GenerateData ([BII)V	99,2%	30,402
4 cz/muni/fi/Fortuna/Algorithm/Generator.GenerateRandomData ([BII)[B	99,0%	30,352
5 cz/muni/fi/Fortuna/Algorithm/Generator.GenerateBlocks (I[BI)V	96,4%	29,543
6 javax/crypto/Cipher.doFinal ([BII[BI)I	93,4%	28,644
7 org/bouncycastle/jce/provider/JCEBlockCipher.engineDoFinal ([BII[BI)I	91,8%	28,135
8 org/bouncycastle/jce/provider/JCEBlockCipher$BufferedGenericBlockCipher.proc	77,9%	23,871
9 org/bouncycastle/crypto/BufferedBlockCipher.processBytes ([BII[BI)I	76,6%	23,476
10 org/bouncycastle/crypto/engines/AESFastEngine.processBlock ([BI[BI)I	69,5%	21,313
11 org/bouncycastle/crypto/engines/AESFastEngine.encryptBlock ([[I)V	58,2%	17,846
12 org/bouncycastle/jce/provider/JCEBlockCipher$BufferedGenericBlockCipher.dol	9,3%	2,843
13 org/bouncycastle/crypto/BufferedBlockCipher.doFinal ([BI)I	8,2%	2,518
14 org/bouncycastle/crypto/BufferedBlockCipher.reset ()V	6,8%	2,091

Obrázek 4.3: Profilování metody generování náhodných dat (stejný proces)

4.2.7 Výdrž baterie

Vzhledem k tomu, že implementace generátoru Fortuna je určena pro mobilní zařízení, považuji za vhodné diskutovat nároky na spotřebu baterie. Celkovou spotřebu rozdělím na dva aspekty. První z nich zastupuje aktivní spotřebu generátoru, tedy spotřebu v době generování náhodných dat. Tato spotřeba bude vždy závislá na využití generátoru ostatními aplikacemi, především pak na množství vygenerovaných dat. Proto lze tuto spotřebu přisoudit spíše aplikacím než generátoru samotnému. Druhým aspektem je spotřeba pasivní. Jelikož generátor běží neustále v pozadí a zpracovává události z různých zdrojů entropie, bylo by velmi nepraktické, pokud by toto zpracování nějak výrazně ovlivnilo životnost baterie zařízení. Naštěstí provedená měření ukazují, že při zpracování dat o objemu 150 b/min není spotřeba nijak negativně ovlivněna. Testovaná hodnota objemu byla zvolena na základě doporučení autorů, že by zdroje entropie měly poskytovat malé množství dat s vysokou entropií [1]. Ovšem je velmi důležité, aby se tímto doporučením řídily především jednotlivé implementace zdrojů entropie, neboť by jejich aktivita mohla způsobit skutečně rychlé vybití baterie. Provedená měření ilustruje obrázek 4.4.



Obrázek 4.4: Graf vybíjení baterie

4.3 Shrnutí

V této kapitole rozebírám generátor Fortuna vzhledem k jeho předchůdci generátoru Yarrow a prezentuji některé myšlenky a motivaci autorů vedoucí k jejich vytvoření. Rovněž jsem vysvětlil rozhodnutí vedoucí k návrhu a implementaci generátoru Fortuna vzhledem k cílové platformě OS Android. Ke generátoru jsou navíc implementovány dva zdroje entropie získávající data z akcelerometru a snímače okolního osvětlení, avšak spíš kvůli demonstraci a testovaní. Na konci také prezentuji výsledné měření dokazující použitelnost implementace vzhledem k nárokům na spotřebu baterie a výkonu generování dat.

5 Závěr

Tato práce se věnuje bezpečnému generování náhodných čísel především na mobilních zařízeních. Náhodná data jsou v oblasti kryptografie velmi důležitá, neboť na jejich bezpečnosti stojí většina kryptografických protokolů. V první části práce byly shrnuty poznatky ohledně generování náhodných čísel, speciálně pak na poli kryptografie. Tato část měla čtenáři přinést základ pro porozumění konceptů dvou implementovaných generátorů.

Prvním z generátorů je standardizovaný generátor ANSI X9.31, který jsem implementoval jako jednoduchou knihovnu a zároveň na něm demonstroval jeho zranitelnost vůči kompromitaci vnitřního stavu. Tato zranitelnost je způsobena nedostatkem entropie, která do vnitřního stavu přichází. Abych maximalizoval množství entropie, byla časová známka vstupující do algoritmu implementována složením známky z časovače s maximální přesností (experimentálně ověřenou na $\frac{1}{6}\mu s$) a běžného systémového času s rozlišením na ms.

Hlavním cílem práce však bylo implementovat kryptograficky bezpečný generátor pseudonáhodných čísel Fortuna, který má především odolat právě kompromitaci vnitřního stavu. V rámci plnění tohoto cíle vzniklo jádro algoritmu, jehož zdrojový kód ve velké míře odpovídá pseudokódu uváděného autory. Samotné jádro je navíc mimo metody zajišťující zápis a čtení souboru se semínkem závislé pouze na bezpečnostním API jazyka Javy, což jej činí velmi snadno přenositelným. Avšak hlavní přínos práce spočívá v návrhu a provedení integrace algoritmu na cílovou platformu. Jádro je začleněno tak, že běží v rámci samostatného procesu ve formě dvou služeb, přičemž první poskytuje komplexní rozhraní pro aplikace využívající generátor ke generování náhodných dat. Druhá služba pak poskytuje minimalistické ale nanejvýš výkonné rozhraní pro zdroje entropie. Nad těmito rozhraními jsem vytvořil ještě knihovnu, která zajišťuje meziprocesovou komunikaci a celkově zjednodušuje použití generátoru vývojářům koncových aplikací. Otázka dodání prvotní entropie, kterou nechali autoři otevřenou, je vyřešena pomocí aktivity vyžadující po uživateli přejíždění prsty po obrazovce, čímž entropii vytvoří. Použitelnost generátoru dokazují měření výkonnosti a dopadu běhu generátoru na životnost generátoru, kde získaná data ukazují, že implementace má do-

statečný výkon (střední hodnota odpovídá 980 kB/s) a zároveň netrpí žádným defektem omezujícím výdrž baterie. Jako pokračování vývoje se zcela jasně nabízí vypracování detailní analýzy a případné implementace zdrojů entropie pro generátor.

Literatura

[1] N. Ferguson, B. Schneier, and T. Kohno, *Cryptography engineering*. Indianapolis: Wiley Publishing, 2010.

[2] E. Barker and J. Kelsey, "DRAFT NIST Special Publication 800-90C, Recommendation for Random Bit Generator (RBG) Constructions," 2012. [Online]. Available: http://csrc.nist.gov/publications/drafts/800-90/draft-sp800-90c.pdf

[3] "RANDOM.ORG - True Random Number Service," 2013, [Navštíveno 30.04.2013]. [Online]. Available: http://www.random.org

[4] "HotBits: Genuine Random Numbers," 2013, [Navštíveno 30.04.2013]. [Online]. Available: http://www.fourmilab.ch/hotbits/

[5] A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone, *Handbook of Applied Cryptography*. CNC Press.

[6] G. Marsaglia, "The Marsaglia Random Number CDROM including the Diehard Battery of Tests of Randomness," 1995, [Navštíveno 12.11.2012]. [Online]. Available: http://www.stat.fsu.edu/pub/diehard/

[7] B. S. John Kelsey and N. Ferguson, "Yarrow-160: Notes on the design and analysis of the yarrow cryptographic pseudorandom number generator," 101 E Minnehaha Parkway, Minneapolis, MN 55419, USA, 1998. [Online]. Available: http://www.schneier.com/yarrow.html

[8] "Java SE Security," 2013, [Navštíveno 10.05.2013]. [Online]. Available: http://www.oracle.com/technetwork/java/javase/tech/index-jsp-136007.html

[9] "Android Developers," 2013, [Navštíveno 10.05.2013]. [Online]. Available: http://source.android.com/

[10] Sharon S. Keller, National Institute of Standards and Technology Information Technology Laboratory Computer Security Division, "NIST-Recommended Random Number Generator Based on ANSI X9.31 Appendix A.2.4 Using the 3-Key Triple DES and AES Algorithms," 2005, [Navštíveno 25.04.2013]. [Online]. Available: http://csrc.nist.gov/groups/STM/cavp/documents/rng/931rngext.pdf

[11] J. Krhovják, "Problematika náhodných a pseudonáhodných sekvencí v kryptografických eskalacních protokolech a implementacích na cipových kartách," Diplomová práce, Masarykova univerzita Fakulta Informatiky.

[12] J. Žižkovský, "Generování náhodných sekvencí v mobilních zařízeních," Diplomová práce, Masarykova univerzita Fakulta Informatiky.

[13] "SecureRandomSpi | Android Developers," 2013, [Navštíveno 01.04.2013]. [Online]. Available: http://developer.android.com/reference/java/security/SecureRandomSpi.html

[14] "RANDOM(4) FreeBSD Kernel Interfaces Manual," 2006, FreeBSD man command.

[15] "random(4) Mac OS X Manual Page," 2001, [Navštíveno 20.04.2013]. [Online]. Available: https://developer.apple.com/library/mac/documentation/Darwin/Reference/ManPages/man4/random.4.html

[16] "Fortuna PRNG from Practical Cryptography," 2013, [Navštíveno 20.04.2013]. [Online]. Available: http://www.citadelsoftware.ca/fortuna/fortuna.htm

[17] "Javascript Crypto Library," 2013, [Navštíveno 20.04.2013]. [Online]. Available: http://www.clipperz.com/open_source/javascript_crypto_library

[18] "Services | Android Developers," 2013, [Navštíveno 02.04.2013]. [Online]. Available: http://developer.android.com/guide/components/services.html

[19] "Service | Android Developers," 2013, [Navštíveno 02.04.2013]. [Online]. Available: http://developer.android.com/reference/android/app/Service.html

[20] "Android Interface Definition Language (AIDL) | Android Developers," 2013, [Navštíveno 02.04.2013]. [Online]. Available: http://developer.android.com/guide/components/aidl.html

[21] "Bound Service | Android Developers," 2013, [Navštíveno 02.04.2013]. [Online]. Available: http://developer.android.com/guide/components/bound-services.html

[22] "Storage Options | Android Developers," 2013, [Navštíveno 04.04.2013]. [Online]. Available: http://developer.android.com/guide/topics/data/data-storage.html