MASARYKOVA UNIVERZITA
FAKULTA INFORMATIKY

Generátory pseudonáhodných čísel v prostředí mobilních telefonů

BAKALÁŘSKÁ PRÁCE

Pavel Zářecký

Brno, jaro 2008

Prohlášení

Prohlašuji, že tato 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

Poděkování

Rád bych poděkoval vedoucímu své bakalářské práce panu Mgr. et Mgr. Janu Krhovjákovi za pomoc při shromažďování podkladů a cenné rady, které významně přispěly ke vzniku této práce.

Shrnutí

Tato práce se zabývá problematikou implementace generátorů pseudonáhodných čísel se zaměřením na prostředí mobilních telefonů. Účelem této práce je vytvoření a měření vybraných generátorů pro platformu S60, Symbian v9.1, 3rd Edition.

Klíčová slova

generátor pseudonáhodných čísel, PRNG, PRBG, pseudorandom, Symbian

Obsah

Kapitola 1

Úvod

Generátory pseudonáhodných čísel jsou využívány mnoha výpočetními systémy k rozmanitým účelům. V kryptografii se používají k vytváření šifrovacích klíčů a inicializačních vektorů, solení hesel a doplňování dat na předem stanovenou délku. Pseudonáhodná data a čísla se využívají v řadě dalších odvětví a aplikací, kde pomáhají simulovat náhodné chování.

Existují dva typy generátorů. Jedny jsou založeny na kryptografických funkcích, ostatní využívají nejrůznější matematické principy. Pro všechny však platí, že se snaží vytvářet bitové sekvence co nejvyšší délky z počátečních hodnot (semínka), získaných od uživatele nebo z prostředí.

Pojmy pseudonáhodná data nebo čísla potom vyjadřují výstupní hodnoty generátorů, které jsou přímo odvozeny ze semínek a nemají náhodný charakter. Ovšem při neznalosti vstupních hodnot není pozorovatel schopen určit ani vyloučit jednotlivé výstupní sekvence.

Bezpečnost pseudonáhodných výstupů lze v tomto kontextu chápat jako nepředvídatelnost nebo variabilitu, která plně závisí na dodávaných vstupních informacích generátoru. Je však těžké takovou bezpečnost měřit nebo počítat. Závisí totiž na komplexním zabezpečení celé implementace generování, ne pouze generátoru samotného.

V roce 2007 vydala americká organizace NIST (*National Institute of Standards and Technology*) upravenou publikaci SP800-90 (*Recommendation for Random Number Generation Using Deterministic Random Bit Generators*) týkající se doporučení pro generování pseudonáhodných dat. Dokument vnáší do problematiky kolem bezpečnosti generování nový pohled. Zabezpečení konkrétně vyčísluje jako práci potřebnou k předpovězení pseudonáhodných výstupů. Pomocí nového konceptu pak navrhuje i nové typy generátorů založené na už používaných principech, jako jsou šifrovací a hašovací algoritmy. Velká část práce je založena na tomto dokumentu a snaží se dodržet doporučení zde popsaná.

Součástí práce jsou implementace různých druhů generátorů. Na jejich základě je vytvořena aplikace testující vlastnosti těchto generátorů, především výkonnost a rychlost. Aplikace pracuje v prostředí S60 Platform, Symbian OS v9.1, 3rd Edition. Existují implementace nejrůznějších generátorů pro operační systémy běžných stolních počítačů. V případě mobilních zařízení ovšem řada chybí. Důvodem je především nižší výkon a špatná podpora kryptografických funkcí. S příchodem nových generací mobilních zařízení, především telefonů, ovšem rapidně narůstá výkonnost a podpora různých forem aplikačních rozhraní, které jsme dříve měli možnost používat pouze na PC.

Kapitola 2

Základní pojmy

Pro snazší porozumění následujícímu textu zadefinujeme nezbytnou terminologii týkající se daného tématu a používanou v práci.

- Entropie
  Míra neuspořádanosti, náhody nebo variability. Ukazuje také míru informace, kterou zpráva předává pozorovateli. Čím pravděpodobnější a očekávanější zpráva je, tím menší entropii vykazuje.

- RBG (*Random Bit Generator*)
  Generátor náhodných bitů, jehož výstupem jsou posloupnosti navzájem statisticky nezávislých bitů. Lze je rozdělit na dva typy:
  - NRBG (*Non-deterministic RBG*)
    Nedeterministický generátor náhodných bitů, někdy také označovaný jako TRBG (*True RBG*), používá nejrůznější vstupy, čímž dosahuje vysoký stupeň entropie. Můžou to být fyzikální hodnoty odvozené z okolí, jako například čas, teplota apod. nebo údaje ze systému, využití pamětí, disků, akce uživatele. Rozmanitost těchto vstupů ovlivňuje dosaženou entropii a tím i nepředvídatelnost výstupu.
  - DRBG (*Deterministic RBG*)
    Deterministický generátor náhodných bitů, také označovaný jako PRBG (*Pseudo RBG*), pracuje se vstupem (vstupy) jako tzv. semínkem (*seed*). Výstupem je pseudonáhodná sekvence bitů požadované délky. Při opakovaném spuštění se stejnými vstupy tvoří tento generátor shodné výstupní posloupnosti. Práce se zabývá právě tímto typem generátorů.

- RNG (*Random Number Generator*)
  Generátor náhodných čísel funguje stejně jako RBG, navíc převádí bitové sekvence na čísla.

- PRNG (*Pseudorandom Number Generator*)
  Generátor náhodných čísel, jehož bitovým generátorem je PRBG.

- Bezpečnostní síla (*Security Strength*)
  Hodnota určující množství práce (počet operací), které je nutné provést k prolomení daného kryptografického algoritmu. Udává se v bitech a obvyklé hodnoty jsou v rozmezí 128-256 pro běžné účely. Práce potřebná k prolomení je dána jako $2^{\text{bezpečnostní síla}}$ operací.

- Náhodná (bitová) sekvence
  Bitová sekvence složená z nul a jedniček, vyskytujících se s pravděpodobností právě $\frac{1}{2}$.

- Zpětná odolnost (*Backtracking Resistance*)
  Výstupní sekvence zůstane nerozlišitelná od náhodné i pro útočníka, který zkompromituje v budoucnu zabezpečení deterministického generátoru. To znamená, že útočník sice může odhalit sekvenci následující, ovšem nikoli předešlou. Zpětné odolnosti docílíme použitím jednosměrných vnitřních funkcí.

- Dopředná odolnost (*Prediction Resistance*)
  Kompromitace vnitřního stavu nemá vliv na bezpečí následujících výstupů. Útočník je schopen určit pouze část následující sekvence, další pokračování už není schopen rozlišit od náhodné sekvence. Dopředné odolnosti lze dosáhnout pravidelným obnovováním vnitřního stavu (*reseeding*).

  Existují různé typy PRNG algoritmů. Spousty z nich jsou založeny na jednoduchých matematických principech, čímž vykazují vysokou rychlost generování dat. Ovšem jejich

bezpečnost za tímto ukazatelem značně zaostává. Velmi často lze posloupnosti vytvořené danými algoritmy předvídat, nebo alespoň rozpoznat. Zbytek lze rozdělit do dvou významných skupin:

- PRNG využívající kryptografické funkce
  Využívají kryptografické mechanismy jako např. šifrovací nebo hašovací funkce. Tyto algoritmy umí pracovat jednosměrně, tedy mohou zaručit zpětnou odolnost generátoru. Byly navrženy pro vytváření dostatečně rozložených bitů, navíc jsou v mnoha případech standardizovány a hardwarově implementovány.
  Mezi zástupce této kategorie řadíme generátory ANSI X9.17 a X9.31 postavené na symetrických šifrách 3DES-2, resp. 3DES-3 a AES. Patří sem také FIPS 186-2, který používá DES nebo hašovací funkci SHA-1.

- Kryptograficky bezpečné PRNG
  Generátory tohoto typu využívají těžké problémy teorie čísel a složitosti k dosažení vysokého stupně bezpečnosti. Takto vytvořené PRNG většinou ovšem nevykazují vysokou rychlost generování, protože mnoho z nich používá náročné modulární umocňování.
  Jako příklad uveďme RSA PRNG založený na problému RSA. Existují také generátory stavící na problému faktorizace přirozených čísel, nebo eliptických křivkách.

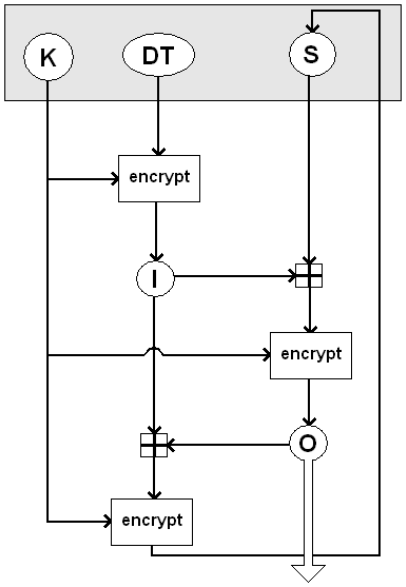
Kapitola 3

Vybrané generátory pseudonáhodných čísel

V této kapitole se seznámíme s nejpoužívanějšími generátory v oblasti kryptografie. Generátory norem ANSI X9.17 a X9.31 jsou založeny na symetrickém šifrování. Dále je zde popsán generátor podle normy FIPS 186-2 využívající jednosměrné funkce a z rodiny kryptograficky bezpečných generátorů je zde popsán RSA PRNG.

3.1 ANSI X9.17 a X9.31 PRNG

Generátor typu X9.17 pracuje se symetrickou šifrou 3DES-2 s klíčem o délce 128 bitů. Udržuje si semínko o délce 64 bitů a používá reprezentaci času a data taktéž o délce 64 bitů. Byl vyvinut pro generování pseudonáhodných klíčů algoritmu DES nebo vytváření inicializačních vektorů, lze ho však použít i pro jiné běžné účely [HoAC].



K – 128bitový klíč,

DT – 64bitová reprezentace času,

S – 64bitové semínko,

O – 64bitový výstup,

⊕ – XOR, nonekvivalence bit po bitu,

encrypt – symetrická šifra

Obr. 1: Schéma generátoru ANSI X9.17

Vstup generátoru:

- tajné 64bitové semínko (*S*)
- tajný 128bitový 3DES klíč (*K*)
- nezáporné celé číslo *n* (počet 64bitových bloků k vygenerování)

Výstupem generátoru je *n* 64bitových pseudonáhodných bloků $O_1, \ldots, O_n$.

Šifru 3DES-2 využívající klíče *K* označme E_K. Nonekvivalenci bit po bitu (XOR) označme symbolem ⊕. Pak generátor X9.17 podle normy ANSI pracuje následovně v *n* iteracích, přičemž v každé z těchto iterací generátor vrací na výstup 64bitovou hodnotu *O*:

1. Spočtení hodnoty *I*:

$$I = E_K(DT)$$

DT reprezentuje aktuální čas a datum na 64 bitech.

2. Odvození výstupu *O*:

$$O = E_K(I \oplus S)$$

3. Aktualizace semínka S:

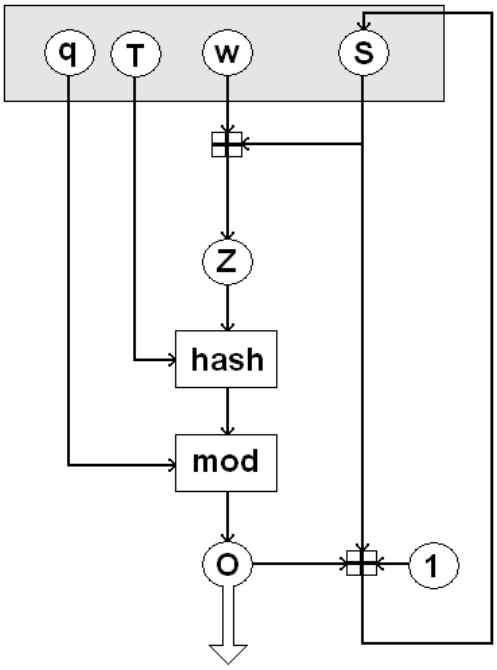
$$S = E_K(O \oplus I)$$

Generátor typu X9.31 se liší pouze použitou šifrou a souvisejícími parametry. 3DES-2 nahrazuje novějšími typy 3DES-3 nebo AES. 3DES-3 používá klíč o délce 192 bitů, ostatní parametry zůstávají na 64 bitech. U šifry AES lze použít klíče 128, 192 nebo 256 bitů dlouhé. Všechny ostatní parametry, ať už jde o semínko *S*, reprezentaci času *DT*, výstupní hodnoty *O* nebo odvozenou hodnotu *I*, šifra AES zpracovává na 128 bitech [NIST].

3.2 FIPS 186-2 PRNG

FIPS 186-2 je oficiální norma amerického standardizačního úřadu NIST zabývající se digitálním podpisováním. Její součástí se stal i generátor pseudonáhodných čísel, který se používá pro vytváření parametrů podpisového algoritmu DSA. Pro tuto oblast byl standardizován, lze ho ovšem bez problémů využít i pro jiné účely.

Generátor je postaven na jednosměrné funkci, kterou tvoří buď hašovací algoritmus SHA-1 nebo DES. V souvislosti s typem algoritmu se provádí početní operace ve zbytkové třídě 2^{160} (v případě DES) nebo 2^n, kde n je celé číslo z rozmezí <160; 512> (v případě SHA-1) [DSS].



q – 160bitové prvočíslo,

T – 160bitový řetězec,

w – uživatelský vstup,

S – semínko,

O – 160bitový výstup,

hash – jednosměrná funkce,

Obr. 2: Schéma generátoru FIPS 186-2

Vstup generátoru (n=160 pro DES, n je v rozsahu <160; 512> pro SHA-1):

- 160bitové prvočíslo q
- konstantní 160bitový řetězec T
- n-bitové semínko S
- nezáporné celé číslo m (počet 160bitových bloků k vygenerování)

Výstupem generátoru je m pseudonáhodných čísel z rozmezí <0; q-1>.

Generátor FIPS 186-2 předává na výstup 160bitové číslo O v m iteracích, každá iterace se skládá z následující procedury:

1. zpracování volitelného uživatelského vstupu (délka w je n bitů)
$$Z = S + w \pmod{2^n}$$
2. Odvození výstupu O:
$$O = hash(T, Z) \pmod{q}$$
3. Aktualizace semínka S:
$$S = O + S + 1 \pmod{2^n}$$

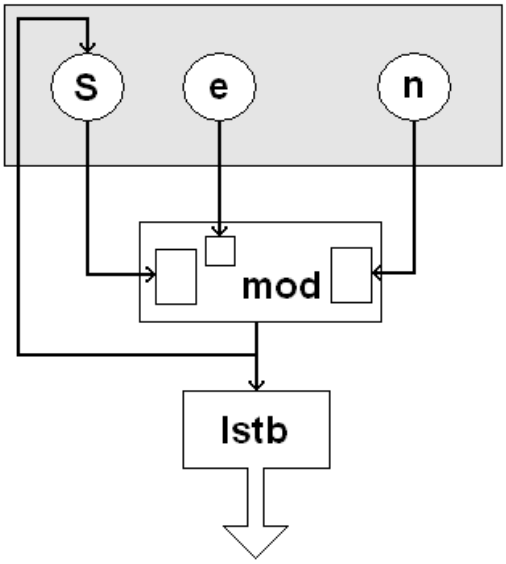
Uživatelský vstup je doporučován při generování tajných klíčů. Pro generování náhodných hodnot k běžným účelům není nutný.

3.3 RSA PRBG

Generátor pseudonáhodných bitů RSA patří, na rozdíl od předchozích dvou, mezi generátory kryptograficky bezpečné. Jak název napovídá, je založen na tzv. problému RSA. Využívá tedy složitosti tohoto problému k zabezpečení generování pseudonáhodných dat. Ovšem ani takový systém není naprosto bezpečný. Musí si uchovávat vnitřní stav a při jeho kompromitaci už nelze dosáhnout dopředné odolnosti bez jistých úprav algoritmu. Kompromitovat vnitřní stav pouze z výstupní sekvence je ovšem přinejmenším stejně složité jako řešit samotný problém RSA.

Při jednom průchodu generátor RSA vrací na výstup jenom jeden bit. To sice značně snižuje jeho výkonnost, ale na druhou stranu zvyšuje odolnost proti kompromitaci vnitřního stavu.

Na vstupu potřebuje algoritmus kromě semínka také dvě dostatečně veliká prvočísla p a q, z nichž vypočte hodnotu $n = p\ q$. Následně vygeneruje nebo jiným způsobem získá hodnotu e z rozmezí $(1; (p\text{-}1)\ (q\text{-}1))$, která je nesoudělná s $(p\text{-}1)\ (q\text{-}1)$ [HoAC].



S – semínko z intervalu $<1; n\text{-}1>$,

lstb – vrací poslední bit

Obr. 3: Schéma generátoru RSA

Vstup generátoru:

- semínko (S)
- počet generovaných bitů k
- hodnoty n a e, odvozené z dvou dostatečně velkých prvočísel

Výstupem algoritmu je k nezávislých pseudonáhodných bitů.

Generátor provede k-krát následující proceduru, kdy při každém průchodu předá na výstup jeden pseudonáhodný bit:

1. výpočet nového semínka S:

$$S = S^e \pmod{n}$$

2. Odvození výstupu O:

$$O = lstb(S)$$

Funkce *lstb* vrací poslední bit svého argumentu: $lstb(x) = x \bmod 2$

Kapitola 4

NIST SP 800-90

V březnu roku 2007 byla vydána upravená publikace [SP800-90] organizace NIST obsahující doporučení pro generování pseudonáhodných bitů a čísel, zejména pak na jeho bezpečnou implementaci. Následující kapitola popíše obecné požadavky na bezpečný a funkční generátor podle výše zmíněné publikace.

4.1 Základní popis

Text dělí generátor na tři základní stavební části: ošetření vstupů, vnitřní stav a funkční mechanismy. Na všechny tyto části klade nároky, které zajišťují zvolenou minimální míru zabezpečení a spolehlivosti.

Generátor zpracovává data obsahující jistou entropii jako vstupy z okolí. Data jsou poté použita buď jako semínko nebo jako pomocný náhodný vstup pro obnovení semínka. Vstupy by měly být utajeny, protože zajišťují bezpečnost generátoru. Entropie musí být dostatečná k zajištění příslušné bezpečnostní síly. K zisku takové entropie lze využít NRBG, nebo zdroj náhodných dat z fyzikálního prostředí.

Vedle vstupu entropie generátory také využívají informace o čase. Taková hodnota by měla mít přinejmenším poloviční bezpečnostní sílu než výstup generátoru.

K zabezpečení může pomoci také tzv. *personalizační řetězec* (*personalization string*). Ten reprezentuje informace o prostředí, ve kterém generátor pracuje. Doporučuje se požívat sériová čísla zařízení, tajné a veřejné klíče, identifikace aplikací, uživatelů a protokolů včetně jejich verzí, skryté hodnoty modulů a zařízení, náhodná čísla, časové známky, síťové adresy a v neposlední řadě také předvytvořené soubory s náhodnými daty (*seed file*).

Vnitřní stav generátoru je paměť skládající se ze všech parametrů, proměnných a dalších hodnot, se kterými dané DRBG pracuje. Uchovává jak administrativní data (bezpečnostní síla, požadavek na dopřednou odolnost a další nastavení), tak i data, která jsou využita pro generování pseudonáhodných bitů a během výpočtu se často mění (nazývají se *pracovní stav*). Pochopitelně je nutné vnitřní stav tajit, protože uchovává vše důležité pro generování dat.

Dále jsou popsány vnitřní mechanismy. Lze je rozdělit na následující:

- Ustavující funkce (*instantiate function*) provádí operace při zakládání generátoru. Zpravidla kombinuje vstupy entropie, času a personalizační řetězec, a vytváří tak semínko a pracovní stav.

- Generující funkce (*generate function*) produkují na žádost pseudonáhodný výstup a zároveň nový vnitřní stav pro další použití.

- Obnovující funkce (*reseed function*) mění vnitřní stav na základě dodatečného vstupu entropie.

- Rušící funkce (*uninstantiate function*) smaže vnitřní stav tak, aby nebyl rozpoznatelný v paměťovém prostoru ani po ukončení generování.

- Testovací funkce (*health test function*) ověřuje, zda za daného vnitřního stavu pracuje generátor korektně.

Ustavující funkce by měla používat k vytvoření semínka taková data, která vykazují dostatečnou bezpečnostní sílu. Při vytvoření instance generátoru je totiž nutné zvolit příslušnou bezpečnostní sílu celého generátoru. Shodnou sílu budou mít poté i jeho výstupy. Není vhodné jako vstup uvažovat data, která slouží k jinému účelu s vyšším zabezpečením. Prolomením zabezpečení generátoru by potom bylo možné takové informace odvodit s nižším než původně zamýšleným úsilím. Podobné pravidlo pochopitelně platí i pro opačný případ, kdy jsou k ustavení generátoru použita data s nižší bezpečnostní silou. Nedoporučuje se používat jednu instanci generátoru k dvěma rozdílným účelům, ani vytvářet dvě instance za pomoci stejných dat. Takové generátory by pak fungovaly do jisté míry stejně a vzájemně by tak snižovaly svou bezpečnostní sílu.

Obnovující funkci je třeba implementovat kvůli dopředné odolnosti. Při kompromitaci vnitřního stavu musí být v budoucnu možné navrátit generátor do bezpečného stavu, kdy útočník ztratí kontrolu nad vnitřním stavem. Takovou vlastnost lze zajistit pravidelným obnovováním semínka. Každému semínku přidružíme i jeho platnost a při jejím vypršení spustíme obnovující funkci. Útočník, který zná aktuální vnitřní stav, ale nemá k dispozici danou platnost zároveň s daty k vytvoření nového semínka, není schopen nový vnitřní stav odvodit. Platnost lze implementovat buď jako počet cyklů generování, dále časově nebo jako kombinaci těchto dvou variant.

Při použití platnosti semínka jako počtu cyklů generování lze zaútočit při nízkém využití generátoru. Obnovující funkce je potom spouštěna v dlouhých časových intervalech, mezi nimiž má útočník dost času k prolomení a předpovězení výstupní sekvence. Když je platnost semínka chápána jako časová známka, potom lze obnovení odvrátit při ovládnutí systémových hodin. Při každém průchodu dostane generátor stejný čas, tedy zjistí, že semínko ještě není třeba měnit, a útočník tak získá dostatek času k jeho prolomení. Kombinací platnosti časové a datové sice neodvrátíme hrozbu kompromitace úplně, nicméně útočníkova pozice se komplikuje, protože ke zjištění semínka musí využít obou útoků naráz.

Publikace dále obsahuje popis jednotlivých mechanismů. Každý generátor by měl ustavovat novou instanci pomocí obecně platné funkce, která zajistí danou bezpečnostní sílu. Ovšem jako první je třeba vyřešit přísun entropie, která je základem k dosažení náhodné či pseudonáhodné výstupní sekvence. K tomuto účelu je zavedena funkce *Get_entropy_input*. Dodejme ještě, že vstupní entropie by neměla být zajišťována pouze nadřazenou aplikací, která používá výstupy generátoru, ale měla by alespoň z části vycházet z okolních zdrojů. Volání výše zmíněné pseudofunkce vypadá následovně:

(*status*, *entropy_input*) = Get_entropy_input (*min_entropy*, *min_length*, *max_length*)

Výstupem je vstupní entropie (*entropy_input*) s alespoň *min_entropy* bity entropie (bezpečnostní síly) a délkou v rozmezí <*min_length*; *max_length*>. Často platí, že *min_length* = *min_entropy*, a tedy je možné druhý parametr vypustit. Hodnota *status* značí, jestli hledání entropie dopadlo úspěšně.

Publikace nicméně neuvádí, jak by měla být tato funkce implementována. Závisí tedy na daném prostředí a dostupných zdrojích, jaká strategie bude při sběru entropie použita.

4.1.1 Ustavující funkce

Po zajištění přísunu entropie je možné vytvářet instance. Ustavující funkce zajišťuje tyto operace:

1. kontrola vstupních parametrů
2. odvození bezpečnostní síly (popř. dalších parametrů)

3. získání vstupní entropie
4. získání časového údaje
5. odvození vnitřního stavu generátoru

Funkce může také vytvářet další vnitřní stavy, pokud to podporuje. V takovém případě vrací navíc ukazatele na tyto stavy. Tento postup se ale nedoporučuje, protože snižuje bezpečnostní sílu.

Proces ustavení nového generátoru je popsán funkcí *Insantiate_function*:

(*status*) = Instantiate_function (*requested_security_strength*, *prediction_resistance_flag*, *personalization_string*)

- *requested_security_strength*: požadovaná bezpečnostní síla. Některé implementace dovolují pouze jedinou hodnotu bezpečnostní síly, pak lze argument vypustit. Aplikace by ovšem měla mít přehled o zabezpečení generátoru.
- *predicition_resistance_flag*: požadavek na dopřednou odolnost. Generátory, které vždy nebo naopak nikdy nepodporují dopředně odolná generování, mohou tento argument vypustit.
- *personalization_string*: nepovinný avšak doporučený personalizační řetězec.

Dalšími argumenty, které vyžaduje ustavující funkce je vstup entropie a časový údaj. Výstupem je indikátor *status*, vyjadřující úspěch (SUCCESS), chybu (ERROR) nebo katastrofickou chybu (CATASTROPHIC_ERROR). Nadřazená aplikace by měla zkontrolovat správné vytvoření generátoru po volání ustavující funkce.

Pseudokód ustavující funkce:

1. Indikátor *status* je nastaven na úspěch.
2. Pokud je požadována vyšší bezpečnostní síla než maximální, kterou může hašovací funkce dosáhnout, pak *status* indikuje chybu.
3. Pokud je požadována dopředná odolnost a generátor ji nepodporuje, potom *status* indikuje chybu.
4. Pokud je personalizační řetězec delší než maximální přípustná hodnota, *status* hlásí chybu.
5. Jako bezpečnostní síla generátoru se zvolí nejbližší vyšší nebo stejná podporovaná hodnota oproti požadované.
6. Volání funkce *Get_entropy_input*, tedy získání vstupu entropie. Pokud je vrácen ERROR, *status* indikuje katastrofickou chybu.
7. Získá se časový údaj.
8. Spouští se funkce *Instantiate_algorithm*, která nastaví výchozí hodnoty vnitřního stavu. Každý typ generátoru si uchovává svůj specifický vnitřní stav, takže tato funkce je implementována každým různě.

4.1.2 Obnovující funkce

Obnovující funkce provádí následující kroky:

1. kontroluje vstupní argumenty
2. získá vstupní entropii
3. pomocí obnovujícího algoritmu kombinuje současný pracovní stav se získanou entropií a dodatečným vstupem a vytváří tak nový pracovní stav

Proces obnovení vnitřního stavu popisuje funkce:

(*status*) = Reseed_function (*additional_input*)

Řetězec *additional_input* není povinnou součástí obnovující funkce. Pouze pomáhá zvýšit bezpečnost generátoru za pomoci dat dodaných nadřazenou aplikací. Kontrolním výstupem je opět indikátor *status*.

Pseudokód obnovující funkce:

1. Indikátor *status* je nastaven na úspěch.
2. Jestliže délka pomocného řetězce *additional_input* je větší než povolená mez, *status* hlásí chybu.
3. Volá se funkce *Get_entropy_input*. Při chybě *status* indikuje katastrofickou chybu.
4. Je volán *Reseed_algorithm*. Podobně jako u ustavující funkce je tento algoritmus specificky implementován pro každý typ generátoru.
5. Pracovní stav je nahrazen nově vytvořeným.

4.1.3 Generující funkce

Generující funkce provádí následující kroky:

1. kontroluje vstupní argumenty
2. volá obnovující funkci, pokud je to vyžadováno
3. generuje daný počet pseudonáhodných dat
4. mění pracovní stav
5. vrací výstupní sekvenci bitů

Proces generace výstupní sekvence je popsán funkcí:

(*status*, *pseudorandom_bits*) = Generate_function (*requested_number_of_bits*, *requested_security_strength*, *prediction_resistance_request*, *additional_input*)

- *requested_number_of_bits*: Počet pseudonáhodných bitů, které má generátor vytvořit.
- *requested_security_strength*: Požadovaná bezpečnostní síla přidružená dané výstupní sekvenci. Nelze generovat bity s vyšší silou, než je hodnota uložená ve vnitřním stavu generátoru.
- *prediction_resistance_request*: Požadavek na dopřednou odolnost. Pokud je obnovovací funkce spouštěna při každém požadavku o pseudonáhodná data nebo není vůbec podporována, pak lze tento parametr vypustit. Jinak značí, že bude obnovení vnitřního stavu provedeno ještě před vytvořením výstupu.
- *additional_input*: Pomocný řetězec. Platí stejné podmínky jako u obnovující funkce.
- *pseudorandom_bits*: Výstupní sekvence pseudonáhodných bitů.

- *status*: Indikuje podobně jako v ostatních funkcích úspěch či chybu.

Pseudokód generující funkce:

1. *status* je nastaven na úspěch.
2. Jestliže požadovaný počet pseudonáhodných bitů překročí stanovenou mez, pak *status* indikuje chybu.
3. Pokud požadovaná bezpečnostní síla převyšuje hodnotu vnitřního stavu, potom *status* hlásí chybu.
4. Pokud je pomocný řetězec delší než stanovená mez, pak bude *status* nastaven na chybu.
5. Pokud je požadována dopředná odolnost a generátor ji nepodporuje, pak *status* indikuje chybu.

6. Jestliže je požadována dopředná odolnost, ať už z důvodu vnitřních či vnějších, spouští se funkce *Reseed_function*. Pokud skončí s chybou, *status* indikuje také chybu. Pokud je pomocný řetězec *additional_input* neprázdný, pak se využije pro obnovení. Dále už použit nebude (nastaví se na prázdný).
7. Volá se *Generate_algorithm*, specificky implementovaný pro každý typ generátoru:

   (*status*, *pseudorandom_bits*) = Generate_algorithm (*requested_number_of_bits*, *additional_input*)

   Pokud skončí chybou, *status* indikuje tuto chybu.
8. Pracovní stav je nahrazen nově vygenerovaným.
9. Funkce vrací řetězec *pseudorandom_bits* jako výstupní sekvenci.

4.1.4 Rušící funkce

Před zrušením dané instance generátoru je třeba smazat hodnoty vnitřního stavu tak, aby v paměti nezůstaly stopy po chráněných hodnotách. Funkce zkontroluje vnitřní stav, potom smaže všechny hodnoty a navrátí paměťový prostor.

4.1.5 Testovací funkce

Každá implementace generátoru by měla být testována jak zvenčí tak zevnitř samotným generátorem. V nejlepším případě je testována každá funkce zvlášť a pak jako celek i generátor. Implementace by měla umožnit aplikaci vyžádat si jakýkoli test kdykoli při běhu. Základní druhy testů jsou:

- Testy známých odpovědí (*known-answer testing*): Pro jisté typy generátorů existují známé hodnoty výstupů pro některé vstupní informace. Můžeme tedy vyzkoušet, zda generátor takový výsledek vyprodukuje. K takovým účelům se ovšem doporučuje vytvořit zcela novou instanci generátoru a známou hodnotu nepoužívat jako běžný výstup.
- Testy ustavující funkce (*Instantiation testing*): Zde se testuje hlavně zacházení se vstupními argumenty (bezpečnostní síla, požadavek dopředné odolnosti, zdroje entropie, časové údaje, chybová hlášení, personalizační řetězce).
- Testy generující funkce (*Generation testing*): Doporučuje se provádět testy známých odpovědí v přiměřených intervalech. Navíc testovat výstupy pro všechny podporované hodnoty bezpečnostní síly a reprezentativní konstantní parametry. Lze také testovat práci s chybovými stavy. Hodnoty, které generátor produkuje během testování, by neměli být použity jako pseudonáhodný výstup.
- Testy obnovující funkce (*Reseed testing*): Provádí se především testy vstupní entropie a chybových stavů. Testy by měly být prováděny při každém spouštění obnovující funkce.
- Testy rušící funkce (*Uninstantiation testing*): Rušení instance je součástí mnoha chybových hlášení. Je tedy vhodné provádět testování smazání hodnot v průběhu každého testu chybového stavu.

V případě, že je během testování objevena chyba, měla by být celá instance generátoru zrušena a před dalším generováním ustavena nová, stejně jako při vzniku katastrofické chyby.

Publikace se soustředí především na dva typy kryptografických primitiv – mechanismy založené na hašovacích funkcích a blokových šifrách.

Hašovací funkcí označujeme funkce jednosměrné nebo neinvertibilní, které z bitového vzorku libovolné délky vytváří bitový řetězec pevné, předem stanovené délky. Navíc výsledné

vzory jsou dobře statisticky rozložené po celém prostoru možných výsledků vzhledem ke vstupním datům.

Publikace uvádí dva typy generátorů založené na hašovací funkci s označeními:

- Hash_DRBG (používá bezklíčovou hašovací funkci)
- HMAC_DRBG (používá klíčovanou hašovací funkci)

Maximální bezpečnostní síla, kterou podporuje daný generátor je menší z hodnot bezpečnostní síly použité hašovací funkce a vstupní entropie. Jako hašovací funkci se doporučuje použít jednu z oficiálně schválených funkcí skupiny SHA. Následná tabulka ukazuje parametry generátoru založeného na dané hašovací funkci.

	SHA-1	**SHA-224**	**SHA-256**	**SHA-384**	**SHA-512**
podporovaná bezpečnostní síla [SP800-57]	80, 112, 128	80, 112, 128, 192	80, 112, 128, 192, 256	80, 112, 128, 192, 256	80, 112, 128, 192, 256
délka výstupu (*outlen*)	160	224	256	384	512
minimální entropie pro ustavení a obnovení	bezpečnostní síla				
minimální vstupní entropie	bezpečnostní síla				
maximální vstupní entropie	$\leq 2^{35}$ bitů				
velikost semínka Hash_DRBG (*seedlen*)	440	440	440	888	888
maximální délka personalizačního řetězce	$\leq 2^{35}$ bitů				
maximální délka přídavného vstupu	$\leq 2^{35}$ bitů				
maximální délka jednoho výstupu	$\leq 2^{19}$ bitů				
platnost semínka (*reseed_interval*)	$\leq 2^{48}$ výstupů				

Tab. 1: Parametry generátorů Hash_DRBG a HMAC_DRBG za použití daných hašovacích funkcí

Protože SHA-224 je založena na SHA-256 a SHA-384 na SHA-512, neexistuje žádný přínos ohledně produktivity ani bezpečnosti při použití funkcí SHA-224 a SHA-384 [SP800-90].

Před popisem generátoru Hash_DRBG je nutné zavést pomocnou odvozenou funkci *Hash_df*. Podobné odvozené funkce jsou vnitřní součástí také dalších generátorů a používají se během ustavení a obnovení vnitřního stavu. Hašovací funkci, kterou generátor používá, označme *Hash* a nechť délka výstupní sekvence je konstanta *outlen*. Potom odvozená funkce pracuje:

(*status*, *requested_bits*) = Hash_df (*input_string*, *no_of_bits_to_return*)

- *input_string*: řetězec určený k hašování
- *no_of_bits_to_return*: délka výstupního haše v bitech (hodnota by měla být shora omezena číslem 255×*outlen*)
- *status*: indikátor úspěchu nebo chyby
- *requested_bits*: výsledná hašovaná sekvence

Odvozená funkce Hash_df, potom pracuje následovně:

1. *temp* = prázdný řetězec
2. *len* = $\left\lceil \frac{no\_of\_bits\_to\_return}{outlen} \right\rceil$
3. *counter* = 8bitová binární hodnota reprezentující číslo 1
4. *len*-krát opakovat cyklus:
   a. *temp* = *temp* || Hash (*counter* || *no_of_bits_to_return* || *input_string*)
   Hodnota *no_of_bits_to_return* je zde použita jako 32bitový řetězec. Operace || značí konkatenaci (zřetězení).
   b. *counter* = *counter* + 1
5. *requested_bits* = Leftmost (*temp*, *no_of_bits_to_return*)
   Funkce *Leftmost* vrátí z řetězce *temp* pouze *no_of_bits_to_return* bitů zleva.
6. *status* = úspěch

4.2 Hash_DRBG

Generátor označený Hash_DRBG pracuje s bezklíčovou hašovací funkcí, která má shodnou nebo vyšší bezpečnostní sílu než výstupní sekvence bitů (Doporučeny jsou standardizované funkce řady SHA). Vnitřní stav se skládá z těchto proměnných a konstant:

- proměnná *V* velikosti *seedlen*, která se mění při každém volání generátoru
- konstanta *C* velikosti *seedlen*, která závisí na semínku
- čítač *reseed_counter* indikující počet výstupů od předešlé žádosti o ustavení nebo obnovení
- konstanta *security_strength* udávající bezpečnostní sílu generátoru
- konstanta *prediction_resistance_flag* indikující, zda je implementován mechanismus zajišťující dopřednou odolnost

Hodnoty *V* a *C* jsou kritické hodnoty, na kterých závisí zabezpečení celého generátoru a které je třeba chránit před útočníkem. Součástí ustavující funkce je algoritmus:

(*working_state*) = Hash_DRBG_Instantiate_algorithm (*entropy_input*, *nonce*, *personalization_string*)

1. *V* = Hash_df (*entropy_input* || *nonce* || *personalization_string*, *seedlen*)
2. *C* = Hash_df (0x00 || *V*, *seedlen*)
3. *reseed_counter* = 1

- *entropy_input*: vstup obsahující dostatečné množství entropie
- *nonce*: jednorázově použitá hodnota s možností opakování pouze po velmi dlouhých časových intervalech
- *personalization_string*: personalizační řetězec systému (nepovinná část)
- *working_state*: výsledný pracovní stav
- *seedlen*: velikost semínka (souvisí s danou hašovací funkcí)

Součástí obnovující funkce je algoritmus:

(*working_state*) = Hash_DRBG_Reseed_algorithm (*entropy_input*, *additional_input*)

1. *V* = Hash_df (0x01 || *V* || *entropy_input* || *additional_input*, *seedlen*)
2. *C* = Hash_df (0x00 || *V*, *seedlen*)

3. *reseed_counter* = 1

- *entropy_input*: vstup obsahující dostatečné množství entropie
- *additional_input*: pomocný vstup (nepovinný)
- *working_state*: nový pracovní stav

Generování probíhá za pomoci algoritmu:

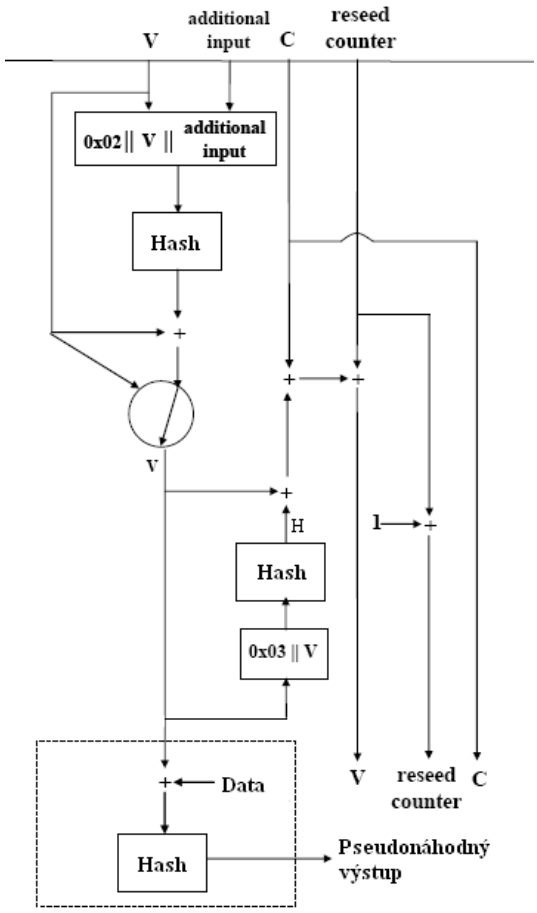
(*returned_bits*, *working_state*) = Hash_DRBG_Generate_algorithm (*requested_number_of_bits*, *additional_input*)

1. Jestliže *reseed_counter* > *reseed_interval*, provede se obnovení vnitřního stavu.
2. Jestliže *additional_input* je neprázdný, potom se provede:
   a. *w* = Hash (0x02 || *V* || *additional_input*)
   b. *V* = (*V* + *w*) mod $2^{seedlen}$
3. *returned_bits* = Hashgen (*requested_number_of_bits*, *V*)
4. *H* = Hash (0x03 || *V*)
5. *V* = (*V* + *H* + *C* + *reseed_counter*) mod $2^{seedlen}$
6. *reseed_counter* = *reseed_counter* + 1

- *requested_number_of_bits*: délka generované sekvence
- *additional_input*: pomocný vstup (nepovinný)
- *returned_bits*: výsledná sekvence
- *working_state*: nový pracovní stav
- *reseed_interval*: platnost semínka
- *Hashgen*: pomocná funkce, viz dále:

returned_bits = Hashgen (*requested_number_of_bits*, *V*)

1. $m = \left\lceil \frac{requested\_number\_of\_bits}{outlen} \right\rceil$
2. *data* = *V*
3. *W* = prázdný řetězec
4. *m*-krát provést cyklus:
   a. *W* = *W* || Hash (*data*)
   b. *data* = (*data* + 1) mod $2^{seedlen}$
5. *returned_bits* = Leftmost (*W*, *requested_number_of_bits*)



Obr. 4: Schéma generátoru Hash_DRBG

4.3 HMAC_DRBG

Generátor typu HMAC_DRBG používá klíčovanou hašovací funkci specifikovanou ve standardu FIPS 198 [FIPS198]. Vnitřní stav se skládá z hodnot:

- proměnná *V* o velikosti *outlen* bitů, která se mění při každém vygenerování *outlen* bitů
- klíč *K* o velikosti *outlen* bitů, který se mění přinejmenším po každém požadavku o vygenerování pseudonáhodné posloupnosti
- čítač *reseed_counter* indikující počet výstupů od předešlé žádosti o ustavení nebo obnovení
- konstanta *security_strength* udávající bezpečnostní sílu generátoru
- konstanta *prediction_resistance_flag* indikující, zda je implementován mechanismus zajišťující dopřednou odolnost

Tajné hodnoty, na kterých závisí bezpečnost generátoru, jsou zde semínko *V* a klíč *K*. Použitou hašovací funkci označme *HMAC*. K ustavení generátoru zde potom slouží algoritmus:

(*working_state*) = HMAC_DRBG_Instantiate_algorithm (*entropy_input*, *nonce*, *personalization_string*)

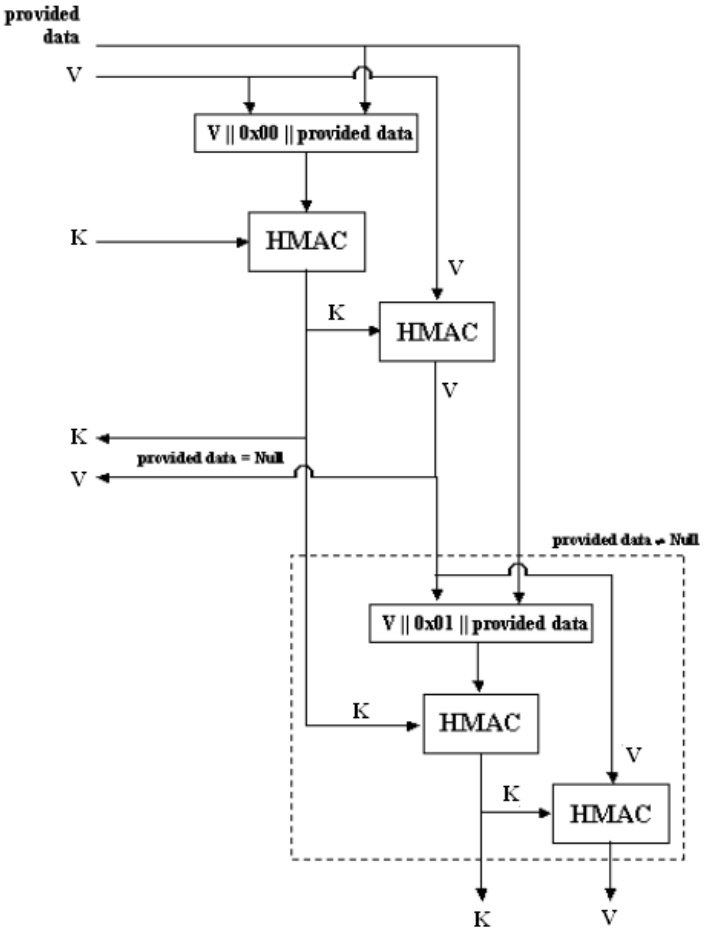
1. *K* = 0x00…00
2. *V* = 0x0101…01

3. (*K*, *V*) = Update (*entropy_input* || *nonce* || *personalization_string*, *K*, *V*)
4. *reseed_counter* = 1

Všechny hodnoty pracují podobně jako v generátoru Hash_DRBG. Funkce *Update* je součástí ustavující, generující a obnovující funkce, kde zastává roli odvozené funkce, podobně jako *Hash_df* v předešlém generátoru:

(*K*, *V*) = Update (*provided_data*, *K*, *V*)

1. *K* = HMAC (*K*, *V* || 0x00 || *provided_data*)
2. *V* = HMAC (*K*, *V*)
3. Jestliže *provided_data* je neprázdný řetězec, potom se provede ještě:
   a. *K* = HMAC (*K*, *V* || 0x01 || *provided_data*)
   b. *V* = HMAC (*K*, *V*)



Obr. 5: Schéma funkce Update generátoru HMAC_DRBG

Součástí obnovující funkce je algoritmus:

(*working_state*) = HMAC_DRBG_Reseed_algorithm (*entropy_input*, *additional_input*)

1. (*K*, *V*) = Update (*entropy_input* || *additional_input*)
2. *reseed_counter* = 1

Podobně jako v předešlých pseudokódech, *entropy_input* značí povinný entropický vstup a *additional_input* nepovinný, pomocný řetězec.

Ke generování slouží algoritmus:

(*returned_bits*, *working_state*) = HMAC_DRBG_Generate_algorithm (*requested_number_of_bits*, *additional_input*)

1. Jestliže *reseed_counter* > *reseed_interval*, pak je třeba provést obnovení vnitřního stavu.
2. Jestliže *additional_input* je neprázdný, potom se volá:
   (*K*, *V*) = Update (*additional_input*, *K*, *V*)
3. *temp* = prázdný řetězec
4. dokud délka *temp* < *requested_number_of_bits*:
   a. *V* = HMAC (*K*, *V*)
   b. *temp* = *temp* || *V*
5. *returned_bits* = Leftmost (*temp*, *requested_number_of_bits*)
6. (*K*, *V*) = Update (*additional_input*, *K*, *V*)
7. *reseed_counter* = *reseed_counter* + 1

- *requested_number_of_bits*: počet generovaných bitů
- *additional_input*: pomocný vstup (nepovinný)
- *returned_bits*: výsledná sekvence
- *working_state*: nový pracovní stav
- *reseed_interval*: platnost semínka

4.4 CTR_DRBG

Publikace zmiňuje pouze jeden typ generátoru založený na blokové šifře. Využívá standardizované šifry DEA nebo AES v módu CTR (*counter mode*). Bezpečnostní síla závisí nejen na použité šifře, ale i na klíči, který si generátor uchovává. K ustavení a obnovování vnitřního stavu se doporučuje používat odvozené funkce. Za splnění jistých podmínek (zajištění bezpečnostní síly entropického zdroje) lze však použít i běžnou šifrovací funkci. Technické parametry při použití jednotlivých šifer a odpovídajících klíčů zobrazuje tabulka 2.

Pokud generátor nepoužívá odvozené funkce ke zpracování vstupních informací, nedoporučuje se používat časové známky k sestavení semínka. Šifra 3-TDEA používá jako klíč tři 64bitové bloky obsahující 56 bitů klíče a 8 paritních bitů. Skutečná délka klíče je proto pouze 168 bitů.

Vnitřní stav sestává z datových položek:

- proměnná *V* délky *outlen*, která se mění při produkci každých *outlen* bitů
- klíč *K* délky *keylen*, který se mění při obnovení vnitřního stavu
- čítač *reseed_counter* indikující počet výstupů od posledního obnovení nebo ustavení
- konstanta *security_strength* udávající bezpečnostní sílu generátoru
- konstanta *prediction_resistance_flag* indikující, zda je implementován mechanismus zajišťující dopřednou odolnost

Tajné hodnoty, na kterých závisí ochrana generátoru, jsou klíč *K* a proměnná *V*.

	3-TDEA	AES-128	AES-192	AES-256
maximální bezpečnostní síla [SP800-57]	112	128	192	256
velikost výstupního bloku (*outlen*)	64	128	128	128
velikost klíče (*keylen*)	168	128	192	256
velikost semínka (*seedlen*)	232	256	320	384
maximální délka jednoho výstupu	$\leq 2^{13}$ bitů	$\leq 2^{19}$ bitů		
platnost semínka (*reseed_interval*)	$\leq 2^{32}$	$\leq 2^{48}$		
S použitím odvozené funkce:				
minimální délka vstupní entropie	bezpečnostní síla			
maximální délka vstupní entropie	$\leq 2^{35}$ bitů			
maximální délka personalizačního a pomocného řetězce	$\leq 2^{35}$ bitů			
Bez použití odvozené funkce:				
délka vstupní entropie	*seedlen*			
maximální délka personalizačního a pomocného řetězce	*seedlen*			

Tab. 2: Parametry generátoru CTR_DRBG za použití daných šifer

Použitou šifru označme *Block_Encrypt*, potom k obnovení vnitřního stavu slouží pomocná funkce:

(*K*, *V*) = Update (*provided_data*, *K*, *V*)

1. *temp* = prázdný řetězec
2. dokud délka *temp* < *seedlen*:
   a. $V = (V + 1) \bmod 2^{outlen}$
   b. *temp* = *temp* || Block_Encrypt (*K*, *V*)
3. *temp* = Leftmost (*temp*, *seedlen*)
4. *temp* = *temp* XOR *provided_data*
5. *K* = Leftmost (*temp*, *keylen*)
6. *V* = Rightmost (*temp*, *outlen*)

- *provided_data*: data s jejichž pomocí se obnoví vnitřní stav
- *XOR*: operace nonekvivalence bit po bitu
- *Leftmost*: vrací požadovaný počet bitů zleva
- *Rightmost*: vrací požadovaný počet bitů zprava

Pomocná odvozená funkce *Block_Cipher_df*:

(*status*, *requested_bits*) = Block_Cipher_df (*input_string*, *no_of_bits_to_return*)

1. Jestliže *no_of_bits_to_return* je větší než povolená mez, status hlásí chybu
2. *L* = (délka *input_string*) / 8
3. *N* = *number_of_bits_to_return* / 8
4. *S* = *L* || *N* || *input_string* || 0x80
5. dokud ((délka *S*) mod *outlen*) ≠ 0: *S* = *S* || 0x00
6. *temp* = prázdný řetězec
7. *k* = Leftmost (0x00010203…1D1E1F, *keylen*)
8. *i* = 0
9. dokud délka *temp* < *seedlen*:
   a. *temp* = *temp* || BCC (*k*, *i* || $0^{outlen\text{-délka}(i)}$ || *S*)
   b. *i* = *i* + 1
10. *k* = Leftmost (*temp*, *keylen*)
11. *X* = další *outlen* bity řetězce *temp*
12. *temp* = prázdný řetězec
13. dokud délka *temp* < *number_of_bits_to_return*:
    a. *X* = Block_Encrypt (*k*, *X*)
    b. *temp* = *temp* || *X*
14. *requested_bits* = Leftmost(*temp*, *number_of_bits_to_return*)

- *input_string*: vstupní řetězec
- *no_of_bits_to_return*: požadovaná délka výstupu
- *requested_bits*: výstupní sekvence
- *status*: indikátor úspěchu nebo chyby
- *BCC*: pomocná šifrovací funkce:

(*output_block*) = BCC (*k*, *data*)

1. *chaining_value* = 0^{outlen}
2. *n* = délka *data* / *outlen*
3. rozdělení *data* na *n* bloků délky *outlen* bitů označené $block_1$ až $block_n$
4. pro *i* = 0 až *n*:
   a. *input_block* = *chaining_value* XOR $block_i$
   b. *chaining_value* = Block_Encrypt (*k*, *input_block*)
5. *output_block* = *chaining_value*

- *k*: klíč k zašifrování
- *data*: vstupní řetězec
- *output_block*: výstupní sekvence

Ustavení nové instance generátoru CTR_DRBG při použití odvozené funkce probíhá za pomoci algoritmu:

(*working_state*) = CTR_DRBG_Instantiate_algorithm (*entropy_input*, *nonce*, *personalization_string*)

1. *seed* = Block_Cipher_df (*entropy_input* || *nonce* || *personalization_string*, *seedlen*)
2. *K* = 0^{keylen}

3. $V = 0^{outlen}$
4. (K, V) = Update (*seed*, K, V)
5. *reseed_counter* = 1

- *entropy_input*: vstup obsahující dostatečné množství entropie
- *nonce*: časový údaj nebo jiná náhodná hodnota
- *personalization_string*: personalizační řetězec
- *working_state*: inicializovaný pracovní stav

Obnovení vnitřního stavu vykonává algoritmus:

(*working_state*) = CTR_DRBG_Reseed_algorithm (*entropy_input*, *additional_input*)

1. *seed* = Block_Cipher_df (*entropy_input* || *additional_input*, *seedlen*)
2. (K, V) = Update (*seed*, K, V)
3. *reseed_counter* = 1

- *entropy_input*: vstup obsahující dostatečné množství entropie
- *additional_input*: pomocný vstup (nepovinný)
- *working_state*: nový pracovní stav

Generování za pomoci odvozené funkce provádí algoritmus:

(*returned_bits*, *working_state*) = CTR_DRBG_Generate_algorithm (*requested_number_of_bits*, *additional_input*)

1. Pokud *reseed_counter* < *reseed_interval*, pak je třeba obnovit vnitřní stav.
2. Jestliže *additional_input* je neprázdný:
   a. *additional_input* = Block_Cipher_df (*additonal_input*, *seedlen*)
   b. (K, V) = Update (*additional_input*, K, V)
   Jinak *additional_input* = $0^{seedlen}$
3. *temp* = prázdný řetězec
4. dokud délka *temp* < *requested_number_of_bits*:
   a. $V = (V + 1) \bmod 2^{outlen}$
   b. *temp* = *temp* || Block_Encrypt (K, V)
5. *returned_bits* = Leftmost (*temp*, *requested_number_of_bits*)
6. (K, V) = Update (*additional_input*, K, V)
7. *reseed_counter* = *reseed_counter* + 1

- *requested_number_of_bits*: délka výstupní sekvence
- *additional_input*: pomocný vstup (nepovinný)
- *returned_bits*: výstupní sekvence
- *working_state*: nový pracovní stav

4.5 Převod bitové sekvence na číslo

Generátory pseudonáhodných bitových sekvencí předávají na výstup pouze bitové řetězce předepsané délky. Často je ovšem třeba tyto sekvence převést na celá čísla z předem zadaného intervalu. Pokud je tento interval typu <*a*; *b*>, pak lze jednoduše převést na typ <0; *b*-*a*> a následně přičíst k výsledku hodnotu *a*.

Skutečný problém ovšem nastává, když cílový interval nemá rozměr mocniny dvou. Všechny možné výstupy by měly nastávat se stejnou pravděpodobností. Bitovou sekvenci však nelze převést na čísla z daného intervalu přímo. Některá čísla by potom vystupovala častěji, čímž dochází ke zkreslení. Proto se používají následující techniky [SP800-90]:

- Jednoduchá vylučovací metoda (*Simple Discard Method*)
  Pro interval výstupů <0; *r*-1> je vygenerována bitová sekvence délky $\lceil \log_2 r \rceil$. Následně je převedena na celé číslo, a pokud vyhovuje danému intervalu, je předána na výstup. Pokud leží mimo interval, provede se nové generování bitové sekvence.
  Tento způsob je efektivní v případě, kdy *r* není o mnoho menší než následující mocnina dvou. Tato metoda vykazuje efektivitu >50% a ≤100%.

- Komplexní vylučovací metoda (*Complex Discard Method*)
  Podobně jako v předešlé metodě se generuje sekvence bitové délky, která odpovídá příslušnému intervalu výstupních čísel a pokud příslušná bitová reprezentace neodpovídá požadovanému rozpětí, je zahozena a celý proces se opakuje. Zde se však negeneruje bitová sekvence pro každé jednotlivé číslo zvlášť, ale pro několik výstupů najednou. Je tedy výhodnější, když potřebujeme větší množství výstupů. Její efektivita je ovšem stejná jako u jednoduché metody.

- Jednoduchá modulární metoda (*Simple Modular Method*)
  Pro interval výstupů <0; *r*-1> a bezpečnostní parametr *s* je vygenerována bitová sekvence délky *m*+*s*, kde $m = \lceil \log_2 r \rceil$. Následně je převedena na celé číslo *c* a spočtena hodnota $a = c \bmod r$, která je předána na výstup.
  Nedochází zde k žádnému zahazování mezivýsledků. Výsledné hodnoty jsou ale lehce zkreslené, protože ne všechna čísla nastávají se stejnou pravděpodobností. Čím vyšší bezpečnostní parametr *s* použijeme, tím vyrovnanější rozložení pravděpodobnosti získáme. Doporučuje se používat hodnoty ≥64. S tím je spojená i jistá ztráta efektivity, protože je třeba vygenerovat větší množství bitů a následně provést modulární dělení.

- Komplexní modulární metoda (*Complex Modular Method*)
  Kombinuje výhody komplexní vylučovací a jednoduché modulární metody. Za pomoci bezpečnostního parametru se generuje bitová sekvence délky odpovídající několika výstupům najednou. Následně se rozdělí a modulárně vydělí jednotlivé výstupní hodnoty. Tento způsob je vhodný především pro situace, kdy je třeba hned několik výstupů naráz.

Kapitola 5

Testování generátorů

V této kapitole se zaměříme na testování vlastností vybraných generátorů. Především pak na rychlost, s jakou jednotlivé implementace vytvářejí pseudonáhodný výstup. Bude zde popsán také způsob generování ve vytvořené aplikaci.

Testování bylo provedeno na mobilním telefonu Nokia N73.

5.1 Prostředí mobilních telefonů

Před samotným zpracováním dat získaných při testování je třeba seznámit se s prostředím mobilních telefonů, potažmo otevřeného operačního systému Symbian v9.1, 3rd Edition [Symbian]. Zařízení využívající danou platformu pracují s procesory typu ARM (*Acorn RISC Machine*). Konkrétně mobilní telefon Nokia N73 používá procesor s pracovní frekvencí 220MHz [N73].

Ke generování je třeba podpora kryptografických funkcí v rámci knihoven. Výběr v rámci Symbian API je však značně omezený. V implementacích je použita knihovna OpenSSL, která zprostředkovává podporu pro nejrůznější druhy šifrovacích a hašovacích algoritmů [OpenSSL].

Důležitým faktorem při generování je také zdroj náhodných (entropických) dat. V prostředí mobilních zařízení je složité implementovat přenositelnou aplikaci sledující náhodné vnitřní systémové parametry. Ke sběru náhodných dat lze proto využívat například informace zprostředkované uživatelem (sledování stisku kláves), nebo informace přijaté z okolí (kamera mobilního telefonu) apod.

5.2 Implementace generátorů

Publikace popisující vybrané generátory neurčují přímo veškeré parametry pro jejich implementaci. V několika oblastech pouze vymezují možná řešení. Některé povinné vlastnosti bylo naopak nutné přepracovat z důvodů omezené podpory aplikačního rozhraní mobilních telefonů. Tato podkapitola popisuje specifika implementací jednotlivých generátorů.

Generátor FIPS 186-2 neprovádí modulární dělení pomocí prvočísla q, ale hodnotou 2^n. Není k dispozici knihovna pro generování velkých prvočísel podporovaná prostředím operačního systému Symbian.

Generátory Hash_DRBG a HMAC_DRBG nepodporují nastavení bezpečnostní síly vyšší než 128b. Chybí totiž knihovny algoritmů SHA-256 a SHA-512 a hašovací algoritmus SHA-1 je schopen zaručit bezpečnostní sílu pouze do výše 128b.

Generátor CTR_DRBG nepoužívá blokovou šifru v módu CTR. Tento mód totiž není podporován knihovnou OpenSSL, v implementaci ho proto nahrazuje mód ECB. Pro bezpečnostní sílu 112b je použita šifra 3DES, pro vyšší hodnoty potom AES.

Generátor RSA nebyl implementován z důvodů špatné podpory knihovních funkcí, není tedy zahrnut ani v testech.

Všechny generátory jsou implementovány jako třídy programovacího jazyka C++, přičemž první typy (podle norem ANSI a FIPS 186-2) jsou implementovány samostatně. Následující generátory vytvořené podle publikace [SP800-90] jsou zpracovány jako potomci abstraktní třídy nazvané *NIST_Generator*. Tento abstraktní předek vykonává práci všech obecných funkcí a odkazuje vždy na příslušný algoritmus, který je různě doimplementován až v třídách potomků jako chráněná metoda. Výsledné generátory se potom navenek chovají stejně, protože se ovládají metodami zděděnými z abstraktního předka. Parametr *reseed_counter* je nastaven na hodnotu 10, tedy po každých deseti zpracovaných požadavcích se provede obnovení vnitřního stavu.

Samotná aplikace pro mobilní prostředí operačního systému Symbian nabízí dva módy pojmenované PRBG a PRNG. V prvním módu je možno testovat rychlost generování pseudonáhodných dat, ve druhém potom generování pseudonáhodných čísel. V obou módech však musí uživatel nejprve vybrat příslušný generátor z nabídky. Pokud vybere generátor popsaný v publikaci [SP800-90], pak musí navíc zadat požadovanou bezpečnostní sílu výstupních dat. Vnitřní hodnoty jako semínko nebo šifrovací klíč jsou generovány náhodně, uživatel je nezadává.

Po nastavení generátoru lze opakovaně generovat data resp. čísla v závislosti na zvoleném módu. Před generováním je uživatel dotázán na délku generované sekvence (v Bytech), při generování čísel zadává uživatel rozsah intervalu možných výsledků. Ke generování čísel je použita jednoduchá vylučovací metoda.

Aplikace navíc měří čas potřebný k samotnému generování a vypisuje ho na obrazovku po dokončení generování. Do výsledného času není započítána režie spojená s inicializací a rušením instance generátoru.

Pokud při generování nahlásí příslušný generátor chybu, zobrazí se na obrazovce mobilního telefonu chybové hlášení.

5.3 Průběh generování

Tento test by měl být komplexním vyhodnocením vlastností daných implementací. Není tedy optimální testovat rychlost zpracování jednoho požadavku na sekvenci dat určité délky. Mezi jednotlivými požadavky totiž probíhají další procedury ovlivňující rychlost generování.

Vytvořená aplikace proto cyklicky zadává požadavek na vytvoření sekvence o délce 32B, dokud nebude vygenerována celá sekvence s uživatelem zadanou délkou. Během generování potom probíhají na generátorech procesy obnovení vnitřního stavu, které zpomalují celý výpočet.

Je nutné zmínit, že aplikace nevyužívá žádné informace z okolí ke sběru náhodných dat použitých jako semínko ke generování. Rychlost generátorů tedy není ovlivněna rychlostí přísunu náhodných dat s příslušnou entropií. Měřeno je tedy pouze samotné generování pseudonáhodných sekvencí bez sběru náhodných dat, inicializování a rušení instancí generátorů.

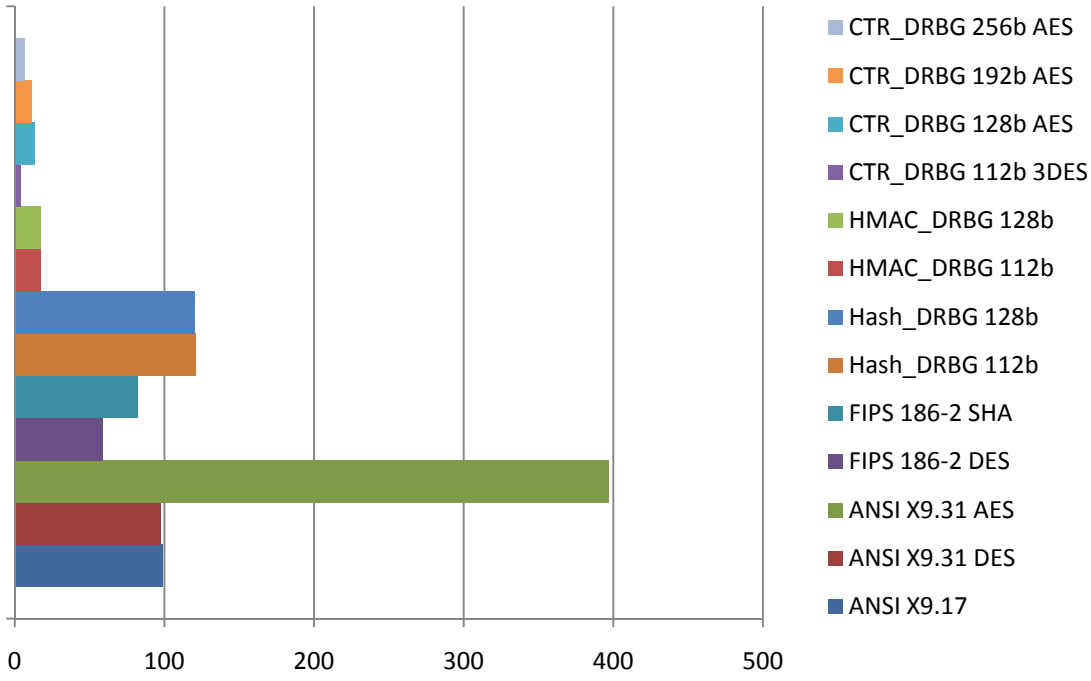
5.4 Výsledky testů

Všechny implementace ve všech dostupných módech (použitá vnitřní kryptografická funkce, různá bezpečnostní síla) byly testovány na rychlost generování pseudonáhodných sekvencí o délkách 100KB a 1MB. Měření každého nastavení bylo provedeno třikrát, abychom eliminovali vnější vlivy operačního systému. Můžeme však prohlásit, že k velkým rozdílům mezi jednotlivými pokusy nedocházelo. Zprávu o jednotlivých testech generátorů můžete najít na přiloženém CD.

Následující tabulka a graf ukazují průměrné časy, které byly třeba ke generování sekvencí příslušných délek na mobilním telefonu Nokia N73, a průměrné rychlosti generování:

Generátor		100KB	1MB	Rychlost generování
ANSI X9.17		1,02s	10,36s	98,84KB/s
ANSI X9.31	DES	1,02s	10,49s	97,62KB/s
	AES	0,25s	2,58s	396,90KB/s
FIPS 186-2	DES	1,70s	17,41s	58,82KB/s
	SHA	1,22s	12,44s	82,32KB/s
Hash_DRBG	112b	0,83s	8,48s	120,75KB/s
	128b	0,83s	8,51s	120,33KB/s
HMAC_DRBG	112b	5,67s	60,00s	17,07KB/s
	128b	5,67s	60,03s	17,06KB/s
CTR_DRBG	112b 3DES	23,16s	238,34s	4,27KB/s
	128b AES	7,31s	75,66s	13,53KB/s
	192b AES	8,60s	90,23s	11,35KB/s
	256b AES	15,56s	161,93s	6,32KB/s

Tab. 3: Časy potřebné ke generování pseudonáhodných sekvencí a průměrná rychlost



Graf 1: Průměrné rychlosti generování v KB/s

Z výsledků je patrné, že nejrychlejšími implementacemi jsou generátory rodiny ANSI. Typ X9.31 používající šifrovací algoritmus AES v rychlosti dokonce až čtyřikrát převýšil všechny ostatní implementace. Dalšími v pořadí jsou Hash_DRBG a FIPS 186-2. Nejpomalejší testovanou implementací je CTR_DRBG, navíc při použití šifry 3DES značně zaostává za verzí s AES.

Celkově pracují generátory s algoritmem AES rychleji než s verzemi DES. Navíc výsledky generátoru FIPS ukazují, že použití hašovacích funkcí namísto blokových šifer je výhodnější.

Testy nezahrnují další úskalí při generování pseudonáhodných sekvencí. Generátory rodiny ANSI nezpracovávají za běhu žádné další uživatelské vstupy na rozdíl od ostatních implementací. Rychlost přísunu takových dat potom značně determinuje rychlost celého generátoru.

Kapitola 7

Závěr

Téměř žádný aplikační nebo systémový návrhář nezačíná s primárním cílem generovat „kvalitní“ náhodná data. Místo toho se soustředí na cíl, kterého chce dosáhnout, a vybírá dostupné kryptografické funkce jako prostředky, které mu pomohou k danému cíli dospět. Bezpečností generovaných dat se začne zabývat až poté. Výběr správného generátoru pseudonáhodných čísel by měl obsahovat komplexní porovnání požadovaných parametrů (rychlost generování, nutná bezpečnostní síla, dostupnost jednotlivých kryptografických algoritmů). Práce ukazuje některé z možných voleb.

Pro účely simulování náhodných jevů v aplikacích dostačují generátory s nízkou mírou zabezpečení a vyšší rychlostí generování. Pro potřeby kryptografie tyto implementace ovšem často nestačí. Na zabezpečení generátorů staví další algoritmy. Pokud potřebujeme klíč pro šifrování, pak slabé zabezpečení generátoru velmi napomáhá útočníkovi k prolomení takové šifry.

Problémem generování, zvláště pak v prostředí mobilních telefonů, je sběr skutečně náhodných dat. Bez nich nelze tvořit bezpečné pseudonáhodné výstupy a při kompromitaci vnitřního stavu generátoru už nemůžeme dále generovat bezpečně. Nejbezpečnější cestou je využívat co největší množství zdrojů náhodných dat tak, abychom neumožnili útočníkovi ovládnout všechny.

Jednou z náplní této práce je důkladné prostudování publikace [SP800-90] a následná implementace a testování zde popsaných generátorů. Norma popisuje základní techniky generování pseudonáhodných čísel se zaměřením na bezpečnou implementaci. Nejdůležitější poznatky a doporučení jsou popsány v této práci a odpovídá jim i řešení implementací jednotlivých generátorů.

Součástí práce jsou implementace všech testovaných generátorů a jednoduchá aplikace, která pracuje v prostředí operačního systému Symbian S60 v9.1, 3rd Edition a ukazuje rychlost a kvalitu práce těchto generátorů. Zdrojové kódy implementací včetně příslušné aplikace jsou součástí přiloženého CD.

Literatura

[DSS] Federal Information Processing Standards Publication, National Institute of Standards and Technology.: *Digital Signature Standard (DSS)*, 19. května 1994. Dokument dostupný na URL http://www.itl.nist.gov/fipspubs/fip186.htm (2008)

[FIPS198] Federal Information Processing Standards Publication, National Institute of Standards and Technology.: *The Keyed-Hash Message Authentication Code (HMAC)*, 6. března 2002. Dokument dostupný na URL http://csrc.nist.gov/publications/fips/fips198/fips-198a.pdf (2008)

[HoAC] Menezes, Alfred J., van Oorschot, Paul C., Vanstone, Scott A.: *Handbook of Applied Cryptography*, CRC Press 1996, ISBN 0-8493-8523-7. Dokument dostupný na URL http://www.cacr.math.uwaterloo.ca/hac/ (2008)

[N73] *Nokia N73*. Parametry mobilního telefonu Nokia N73. Dokument dostupný na URL http://www.forum.nokia.com/devices/N73

[NIST] Keller, Sharon S.: *NIST-Recommended Random Number Generator Based on ANSI X9.31 Appendix A.2.4 Using the 3-Key Triple DES and AES Algorithms*, 31. ledna 2005. Dokument dostupný na URL http://csrc.nist.gov/groups/STM/cavp/documents/rng/931rngext.pdf (2008)

[OpenSSL] *OpenSSL*. Knihovna kryptografických funkcí OpenSSL. Dokument dostupný na URL http://www.openssl.org/

[SP800-57] Barker, E., Barker, W., Burr, W., Polk, W., Smid, M.: *Recommendation for Key Management*, 8. března 2007. Dokument dostupný na URL http://csrc.nist.gov/publications/nistpubs/800-57/sp800-57-Part1-revised2_Mar08-2007.pdf (2008)

[SP800-90] Barker, E., Kelsey, J.: *Recommendation for Random Number Generation Using Deterministic Random Bit Generators (Revised)*, březen 2007. Dokument dostupný na URL http://csrc.nist.gov/publications/nistpubs/800-90/SP800-90revised_March2007.pdf (2008)

[Symbian] *Symbian*. Oficiální stránky operačního systému Symbian. Dokumenty dostupné na URL http://www.symbian.com/