Masarykova universita
Fakulta informatiky

Kryptografické funkce v jazyce JAVA

Bakalářská práce

Ivan Fialík

2004

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.

Brno 22. 4. 2004

Děkuji Mgr. Zdeňku Říhovi, Ph.D. za odborné vedení bakalářské práce a poskytování cenných rad při jejím zpracování.

Shrnutí

Tato bakalářská práce má za cíl ukázat, jaké možnosti poskytuje programovací jazyk Java pro psaní programů, v nichž jsou prováděny nebo implementovány základní kryptografické operace.

Klíčová slova

Java, kryptografie, engine class, provider, balík java.security, balík javax.crypto, Rabinův kryptosystém, Ong-Shnor-Shamirův podepisovací systém, efektivita.

Obsah

1. Úvod

V posledních desetiletích došlo při uchovávání a předávání informací k přechodu od klasických postupů k postupům založeným na jejich elektronickém zpracování za použití výpočetní techniky. Aby tento přechod mohl úspěšně proběhnout a aby se mohly plně projevit výhody elektronického zpracování informací, bylo nutné vyřešit některé problémy týkající se zabezpečení digitálních dat, jež při použití klasických metod buď vůbec neexistovaly, nebo byly zanedbatelné. Na zvládnutí těchto problémů měla významný podíl kryptografie, a tak se z vědy využívané především vládci a vojevůdci stala věda, která ovlivňuje každodenní život většiny obyvatelstva.

Tato práce je základním přehledem kryptografických funkcí poskytovaných programovacím jazykem Java a ukazuje také, jak je možné v tomto jazyce implementovat některé kryptografické algoritmy. Java je objektově orientovaným jazykem, jenž za svou nedlouhou historii získal tolik příznivců, že z hlediska popularity mezi programátory v současnosti ohrožuje výsadní postavení jazyka C a z něj odvozených jazyků C++ a C#. Informace obsažené v dalším textu se vztahují k nejnovější verzi standardní javové platformy, jejíž celý název je Java 2 Platform, Standard Edition, v1.5.0, přičemž se budu snažit upozorňovat na rysy, jež se v této verzi Javy objevují nově. Práce je tvořena třemi hlavními částmi. Úvodní dvojice kapitol podává základní informace týkající se jazyka Java (především se zaměřením na principy objektově orientovaného programování) a vysvětluje základní kryptografické pojmy. Následující kapitola je popisem funkcionality kryptografických balíků v Java Core API. Tento popis zdaleka není úplný, soustředí se především na ty třídy, které jsou určeny pro realizaci základních kryptografických operací. V závěrečné části je ukázáno, jak lze v Javě implementovat Rabinův kryptosystém a Ong-Schnor-Shamirův podepisovací systém.

2. Programovací jazyk Java

2.1 Vlastnosti jazyka

Java patří mezi programovací jazyky třetí generace. Jedná se o objektově orientovaný jazyk vycházející z jazyků C a C++, který byl vyvinut firmou Sun Microsystems. V současnosti patří Java mezi nejpopulárnější programovací jazyky, což souvisí především s rozmachem internetových služeb v polovině devadesátých let minulého století. Nejčastěji se můžeme s programy v Javě setkat na stránkách WWW, kde jsou s její pomocí vytvářeny malé aplikace, tzv. applety.

Pravděpodobně největší předností Javy je platformová nezávislost programů. Zatímco pro většinu programovacích jazyků platí, že programy v nich napsané jsou buď kompilovány, nebo interpretovány, program v Javě podstupuje oba tyto procesy. Nejprve proběhne kompilace, během které program není přeložen do strojového kódu, nýbrž do jakéhosi mezikódu nazývaného bytecode, jenž je po spuštění programu interpretován tzv. javovým virtuálním počítačem (Java Virtual Machine, JVM). Přeložený program může běžet na libovolném počítači, na kterém je instalována javová platforma, která na softwarové úrovni zastřešuje rozdíly mezi operačními systémy nebo hardwarem. Vedle JVM javovou platformu dále tvoří vývojové nástroje (překladač, debugger, atd.) a Java Core API, což je aplikační programové rozhraní obsahující implementaci základních tříd. Další vlastnosti Javy, jež jsou pro programátora příjemné, uvedeme pouze stručně:

- robustnost – mechanismus výjimek umožňuje tvořit programy odolné vůči nejrůznějším chybovým stavům.
- podpora vytváření dynamických a distribuovaných aplikací.
- bezpečnost aplikací – např. programová kontrola přístupu k objektům či podpora šifrování.
- paralelismus – prostředky pro paralelní běh částí programu a jejich synchronizaci.
- odstranění řady problémů typických pro C++.

Z dosud uvedeného by se mohlo zdát, že Java je dokonalým jazykem, při jehož používání nevznikají žádné potíže, ale tak tomu není. Důsledkem platformové nezávislosti je poměrně malá rychlost programů. Především pro javová vývojová prostředí je charakteristická vysoká paměťová náročnost. Není-li k dispozici operační paměť odpovídající velikosti, je sice možné kompilovat a spouštět programy, nicméně dochází k dalšímu zpomalení práce počítače.

2.2 Základy programování v Javě

V této části budou vysvětleny základy objektově orientovaného programování. Důraz bude kladen především na jejich uplatnění v Javě.

Skutečné objekty vyskytující se kolem nás lze charakterizovat jejich vlastnostmi a činnostmi, které mohou provádět. Podobně též softwarové objekty mají určité vlastnosti a chování. Vlastnosti softwarového objektu jsou dány obsahem jeho proměnných, zatímco chování jeho metodami. Mezi nejznámější programovací jazyky patří Pascal, metody tedy můžeme přirovnat např. k pascalovským procedurám a funkcím. Softwarové objekty nám mohou reprezentovat nejen skutečné objekty, ale také objekty abstraktní. Příkladem takového abstraktního objektu může být třeba zlomek, který lze implementovat pomocí proměnných čitatel a jmenovatel a metod definujících, jak se tyto proměnné změní, aplikujeme-li na daný zlomek nějakou aritmetickou operaci.

Vytvoření softwarového objektu probíhá ve dvou fázích. Nejprve je nezbytné napsat a přeložit zdrojový text, popisující vlastnosti společné všem objektům daného druhu. Tuto definici společných vlastností nazýváme třídou (class). Až na malé výjimky je každá třída v Javě uložena do samostatného souboru takového, že jeho jméno se od jména dané třídy liší jen příponou .java. Vedle proměnných a metod náležejících nějakému konkrétnímu objektu je možné využít také proměnné a metody, jež jsou sdíleny všemi zástupci dané třídy. Nazýváme je statické a ve zdrojovém textu třídy je deklarujeme uvedením klíčového slova static. Vedle proměnných a metod se ve zdrojovém kódu třídy mohou vyskytovat též tzv. konstruktory. Konstruktor je speciální metoda volaná při vytváření objektu dané třídy, která definuje vlastnosti vytvářeného objektu. Oproti obvyklým metodám konstruktory nemají návratový typ a jejich název je stejný jako název příslušné třídy. Ve druhé fázi pomocí operátoru new vytváříme konkrétní objekty (instance) dané třídy. Předpokládejme, že jsme napsali a přeložili zdrojový text třídy Zlomek. Potom objekt této třídy reprezentující zlomek ½ vytvoříme následovně:

```
Zlomek z = new Zlomek(1,2);
```

Výsledkem této konstrukce bude vytvoření nového objektu třídy Zlomek a přiřazení odkazu na tento objekt do proměnné z.

Při práci se softwarovými objekty se často v rámci nějaké třídy vyskytují skupiny podobných objektů, které můžeme podrobněji definovat pomocí podtříd (subclasses, potomci). Každá podtřída vedle vlastních proměnných a metod přebírá (dědí) veškeré proměnné a metody nadřazené třídy (superclass, rodič), v jejichž deklaraci se objevuje modifikátor přístupu public nebo protected. Jestliže se potomek nachází ve stejném balíku (viz poslední odstavec této kapitoly) jako rodič, přebírá též všechny proměnné a metody, jež nemají určena přístupová práva. Nevyhovující metody můžeme v potomkovi předefinovat. Chceme-li vytvářenou třídu deklarovat jako podtřídu, uvedeme za jejím názvem klíčové slovo extends následované názvem rodičovské třídy.

Zvláštním případem třídy v Javě je tzv. abstraktní třída, kterou deklarujeme klíčovým slovem abstract. Jedná se o takovou třídu, která obsahuje alespoň jednu abstraktní metodu definovanou pouze pomocí její hlavičky a označenou rovněž klíčovým slovem abstract. Poněvadž abstraktní třída není úplně definovaná, není možné vytvořit její objekt. Hlavní význam abstraktních tříd spočívá v usnadnění práce při tvorbě rozsáhlejších programů.

Další strukturou, která se často vyskytuje především v rozsáhlejších javových programech, je rozhraní (interface). Jde o seznam metod reprezentovaných jejich hlavičkami. Rozhraní deklarujeme stejným způsobem jako třídu, pouze klíčové slovo class je nahrazeno klíčovým slovem interface a za klíčovým slovem extends může být uvedeno více rodičovských rozhraní. Pakliže třída definuje všechny metody nějakého rozhraní, zachytíme tento fakt tak, že za jejím názvem uvedeme klíčové slovo implements následované názvem daného rozhraní. Třída může implementovat více než jedno rozhraní. Pomocí rozhraní je možné dosáhnout větší přehlednosti rozsáhlých programů. Je také možné deklarovat proměnnou takto:

```
název rozhraní proměnná;
```

Touto konstrukcí obdržíme proměnnou schopnou nést odkaz na objekt libovolné třídy, jež implementuje uvedené rozhraní.

Javový program je tvořen alespoň jednou třídou a v případě potřeby také rozhraními. Programy složené z většího počtu tříd a rozhraní dále členíme na balíky (packages), což jsou skupiny souvisejících tříd a rozhraní. Každý balík je uložen ve zvláštním adresáři a pojmenován na základě cesty, jež k němu vede. Jméno balíku dostaneme tak, že v příslušné relativní cestě, která začíná v nějakém pevně určeném adresáři, nahradíme každý oddělovač položek tečkou. Ve zdrojovém textu třídy lze bez omezení pracovat s objekty a statickými metodami, které nejsou deklarovány

s modifikátorem přístupu private, tříd nacházejících se ve stejném balíku. Chceme-li využít třídu z jiného balíku, musíme tento úmysl deklarovat klíčovým slovem import následovaným jménem příslušného balíku, tečkou a názvem zpřístupňované třídy. Uvedeme-li za názvem balíku tečku a hvězdičku, zpřístupníme tímto všechny třídy z daného balíku. Pro zpřístupňování rozhraní platí stejná pravidla jako pro zpřístupňování tříd.

3.Kryptografie

3.1 Význam kryptografie

V každém historickém období existovaly informace, jež se určitá skupina lidí pokoušela utajit před ostatními. K tomuto účelu bylo využíváno nejrůznějších prostředků, z nichž jmenujme např. neviditelné inkousty či slavný německý šifrovací stroj Enigma, který sehrál významnou roli ve 2. světové válce. Tyto příklady současně ilustrují dva možné přístupy k utajování informací. Můžeme se snažit utajit samotnou existenci nějaké zprávy, nikoliv její obsah. Studiem těchto technik se zabývá steganografie. Druhou možností, jak lze k utajování přistoupit, je naopak vhodná modifikace obsahu zprávy.

Kryptografie je v současnosti základním nástrojem pro zabezpečení přenosu informací, jenž je stále častěji uskutečňován elektronickou formou. Velké rozšíření elektronického zpracování dat přineslo další požadavky na jejich bezpečnost. Elektronická data mohou být např. snadno neoprávněně změněna a problémem je také správná identifikace autora elektronické zprávy. Moderní kryptografie tyto problémy řeší pomocí protokolů založených na matematickém základě a na poznatcích z teorie složitosti. Bezpečnostních cílů, kterých lze pomocí různých kryptografických protokolů dosáhnout, bychom zde mohli uvést celou řadu, ale omezíme se pouze na následující čtyři, jež jsou považovány za základní:

- důvěrnost – pouze oprávněné subjekty mohou dešifrovat utajovanou zprávu.
- integrita dat – byl-li obsah zprávy změněn neoprávněným subjektem, její příjemce je schopen tuto skutečnost zjistit.
- autentizace – jestliže subjekt B dostal zprávu od subjektu A, potom B je schopen ověřit, zda A je autorem této zprávy (autentizace dat). Není možné, aby subjekt C vystupoval během komunikace se subjektem A pod identitou subjektu B, aniž by A tento fakt odhalil (autentizace uživatelů).
- nepopiratelnost – pokud subjekt A poslal zprávu subjektu B, pak A není schopen přesvědčit B, že tuto zprávu neposlal.

3.2 Symetrická kryptografie

V této a také v následující kapitole se budeme zabývat kryptografickými algoritmy, které jsou zaměřeny na dosažení důvěrnosti, nejprve ovšem definujeme několik základních pojmů. Kryptografickým systémem (kryptosystémem) nazýváme pětici (M, C, K, E, D). M označuje množinu všech možných zpráv a jejími prvky jsou řetězce nad nějakou konečnou abecedou Σ. Podobně C je množina možných šifer tvořená řetězci nad konečnou abecedou Δ. Písmenem K pak značíme množinu všech klíčů. E je množina prostých funkcí z M do C taková, že ke každému prvku množiny K existuje právě jedna funkce patřící do E. Analogicky definujeme D jako množinu funkcí z C do M takovou, že ke každému prvku množiny K existuje právě jedna funkce, jež patří do D. Dále požadujeme, aby pro každé $e \in E$ existovala funkce $d \in D$ taková, že pro každé $m \in M$ platí rovnost $d(e(m)) = m$. Pravíme, že došlo k rozlomení kryptosystému, pokud nějaký subjekt dokáže i bez znalosti použitého klíče k dané šifře určit odpovídající zprávu. Termínem kryptoanalýza bývá označováno studium metod vedoucích k rozlomení nejrůznějších kryptosystémů. Vědu vzniklou spojením kryptografie a kryptoanalýzy nazýváme kryptologie. Moderní kryptoanalýza je

založena na Kerckhoffově principu (1883), jenž říká, že spolehlivost kryptosystému nesmí záviset na utajení algoritmu, nýbrž pouze na utajení klíče.

Pro kryptosystémy splňující požadavek důvěrnosti platí, že příjemce zašifrované zprávy disponuje nějakou informací (klíčem), již použije k dešifrování zprávy. Klíč nesmí být znám žádnému neoprávněnému subjektu, poněvadž v opačném případě by určitý neoprávněný subjekt byl schopen rozluštit zprávu stejně snadno jako příjemce. Podobně jako příjemce také odesilatel pro zašifrování zprávy použije nějaký klíč. Řekneme, že kryptosystém je symetrický, pokud pro libovolnou dvojici klíčů (e, d) takovou, že d je odpovídající dešifrovací klíč k šifrovacímu klíči e, je výpočetně jednoduché určit d pouze na základě znalosti e a obráceně. Symetrické kryptosystémy bývají také často nazývány kryptosystémy s tajným klíčem. Poznamenejme, že ve většině v praxi používaných symetrických kryptosystémů jsou si šifrovací a odpovídající dešifrovací klíč rovny.

Rozlišujeme dvě základní skupiny symetrických kryptosystémů, blokové šifrovače a proudové šifrovače.

3.2.1 Blokové šifrovače

Blokový šifrovač je symetrický kryptosystém, v němž je šifrovaná zpráva nejprve rozdělena na části (bloky) stejné délky a ty jsou postupně šifrovány. Základními typy blokových šifrovačů jsou substituční šifrovače a transpoziční šifrovače.

Transpoziční šifrovač s délkou bloku t je takový blokový šifrovač, který permutuje symboly v každém bloku na základě nějaké pevně dané permutace na množině {1, 2, … , t}. Dešifrování zprávy zašifrované pomocí permutace e provádíme užitím permutace inverzní k e. Transpoziční šifrovače zachovávají pro každý symbol jeho počet ve zprávě, což umožňuje jejich snadnou kryptoanalýzu.

Substitučním šifrovačem rozumíme takový blokový šifrovač, který nahrazuje symbol (nebo skupinu symbolů) jiným symbolem (nebo skupinou symbolů). Substituční šifrovače nahrazující jediný symbol dále dělíme na monoalfabetické a polyalfabetické.

Jestliže substituční šifrovač nahrazuje daný symbol při každém jeho výskytu stejným symbolem, jenž je určen permutací na abecedě Σ, mluvíme o monoalfabetickém substitučním šifrovači. Existuje tedy právě $|\Sigma|!$ takových šifrovačů. Klíčem pro dešifrování zprávy zašifrované pomocí permutace e je permutace inverzní k e. Každé monoalfabetické šifrování, jež pracuje se zprávami v přirozeném jazyce, je možné poměrně snadno rozlomit. Je pouze třeba mít k dispozici dostatečně dlouhý vzorek šifrovaného textu a znát frekvenci výskytu jednotlivých symbolů abecedy a také jejich dvojic v daném jazyce. Z uvedeného důvodu se v současné době častěji používá buď polyalfabetické šifrování, nebo monoalfabetické šifrování nad umělým jazykem.

Polyalfabetický substituční šifrovač o délce bloku t je substituční šifrovač, jenž má následující vlastnosti. Množina klíčů je tvořena všemi množinami $\{p_1, p_2, \dots, p_t\}$ takovými, že pro každé i patřící do množiny {1, 2, … , t} p_i je permutace na množině Σ. Při šifrování bloku pomocí klíče $\{p_1, p_2, \dots, p_t\}$ nahrazujeme i-tý symbol v bloku symbolem určeným permutací p_i. K dešifrování zprávy zašifrované užitím množiny permutací E použijeme množinu permutací D, jejímiž prvky jsou právě ty permutace, které jsou inverzní k nějaké permutaci obsažené v E. Výhodou polyalfabetických šifrovačů je skutečnost, že nezachovávají frekvenci výskytu jednotlivých symbolů ve zprávě, přesto ovšem existují metody, které umožňují takový šifrovač rozlomit.

Nejznámějším symetrickým kryptosystémem je DES (Data Encryption Standard). Jedná se o substituční a transpoziční blokový šifrovač, který šifruje řetězce nad abecedou {0, 1}. Při šifrování zprávy v přirozeném jazyce musí být tato zpráva tedy nejdříve převedena na řetězec bitů. DES byl vyvinut firmou IBM a publikován v roce 1977. V současnosti je tento kryptosystém používán především v bankovnictví. DES pracuje s bloky délky 64 bitů, které šifruje na řetězce stejné délky za použití klíče o délce 64 bitů, z nichž 8 bitů zajišťuje paritní kontrolu. Skutečná

délka klíče je tedy 56 bitů. Pro DES byly stanoveny 4 základní režimy činnosti, jejichž princip může být použit i pro libovolný jiný blokový šifrovač:

- ECB (Electronic Code Book)
  Vstup – klíč K, zpráva tvořená bloky $m_1, \ldots, m_t$.
  Výstup – šifra tvořená bloky $c_1, \ldots, c_t$.
  Šifrování – $c_i = k(m_i)$, kde $1 \le i \le t$ a k je šifrovací algoritmus určený klíčem K.
  Dešifrování – $m_i = k^{-1}(c_i)$, kde $1 \le i \le t$.
- CBC (Cipher Block Chaining)
  Vstup – klíč K, zpráva tvořená bloky $m_1, \ldots, m_t$, 64-bitový řetězec c_0 (inicializační vektor).
  Výstup – šifra tvořená bloky $c_1, \ldots, c_t$.
  Šifrování – $c_i = k(c_{i-1} \oplus m_i)$, kde $1 \le i \le t$ a k je šifrovací algoritmus určený klíčem K.
  Dešifrování – $m_i = k^{-1}(c_i) \oplus c_{i-1}$, kde $1 \le i \le t$.
- CFB (Cipher FeedBack)
  Vstup – klíč K, zpráva tvořená r-bitovými bloky $m_1, \ldots, m_t$, $1 \le r \le 64$, 64-bitový řetězec I_1.
  Výstup – šifra tvořená r-bitovými bloky $c_1, \ldots, c_t$.
  Šifrování – $O_i = k(I_i)$, kde $1 \le i \le t$ a k je šifrovací algoritmus určený klíčem K.
  u_i = r nejlevějších bitů O_i.
  $c_i = m_i \oplus u_i$.
  $I_{i+1} = 2^r \cdot I_i + c_i \bmod 2^{64}$.
  Dešifrování – $m_i = c_i \oplus u_i$, kde $1 \le i \le t$, u_i, O_i a I_i vypočítáme stejně jako při šifrování.
- OFB (Output FeedBack)
  Vstup – klíč K, zpráva tvořená r-bitovými bloky $m_1, \ldots, m_t$, $1 \le r \le 64$, 64-bitový řetězec I_1.
  Výstup – šifra tvořená r-bitovými bloky $c_1, \ldots, c_t$.
  Šifrování – $O_i = k(I_i)$, kde $1 \le i \le t$ a k je šifrovací algoritmus určený klíčem K.
  u_i = r nejlevějších bitů O_i.
  $c_i = m_i \oplus u_i$.
  $I_{i+1} = 2^r \cdot I_i + u_i \bmod 2^{64}$.
  Dešifrování – $m_i = c_i \oplus u_i$, kde $1 \le i \le t$, u_i, O_i a I_i vypočítáme stejně jako při šifrování

3.2.2 Proudové šifrovače

Blokové šifrovače obvykle šifrují každý blok zprávy pomocí téhož klíče. Jednou z možností, jak zvýšit bezpečnost blokového šifrovače, je šifrovat různé bloky užitím různých klíčů. Proudový šifrovač můžeme definovat jako blokový šifrovač, jenž je navíc vybaven nějakým generátorem pseudonáhodných hodnot generujícím posloupnost klíčů. Zašifrování i-tého bloku zprávy je pak provedeno i-tým prvkem vygenerované posloupnosti a jeho dešifrování odpovídajícím dešifrovacím klíčem.

Proudové šifrovače poskytují vyšší úroveň bezpečnosti než šifrovače blokové. Jejich praktická použitelnost je ale omezena, neboť při větší délce zprávy vyžadují použití dlouhé posloupnosti klíčů a není jednoduché bezpečně přenést takovou posloupnost od odesilatele k příjemci. Pro dosažení vysoké úrovně bezpečnosti je nutné, aby se posloupnost klíčů svými vlastnostmi co nejvíce blížila náhodné posloupnosti, ovšem vygenerování takové posloupnosti je netriviálním problémem.

3.2.3 Protokoly dohody na klíči

Protokol dohody na klíči je takový kryptografický protokol, který umožňuje dvěma nebo více subjektům dohodnout se na tajné hodnotě (nejčastěji na klíči pro symetrický kryptosystém), přičemž zúčastněné subjekty komunikují přes nezabezpečené médium. Nejznámějším protokolem dohody je Diffie-Hellmanův protokol, který je popsán v následujícím odstavci.

Předpokládejme, že subjekty A, B chtějí společně vytvořit tajný klíč. A a B se nejprve nějakým postupem dohodnou na velkém prvočísle p a na přirozeném čísle q takovém, že $[q]_p$ je generátorem grupy $(\{[1]_p, [2]_p, \dots, [p-1]_p\}, \cdot)$ a $1 \leq q < p$. Dále si A zvolí přirozené číslo x z množiny $\{1, 2, \dots, p-2\}$ a vypočítá číslo $a = q^x \bmod p$. Podobně si B vybere přirozené číslo y, $1 \leq y < p-1$ a vypočítá číslo $b = q^y \bmod p$. Následně si A a B přes nezabezpečený kanál vymění hodnoty a, b. Nakonec A vypočítá $b^x \bmod p$ a B vypočítá $a^y \bmod p$, oba subjekty dostanou tutéž hodnotu $q^{xy} \bmod p$. Uvedený protokol může být snadno rozšířen tak, aby se jej mohlo účastnit tři nebo více subjektů. Jeho bezpečnost je založena na obtížnosti řešení problému diskrétního logaritmu.

3.3 Asymetrická kryptografie

Ve druhé polovině minulého století se v souvislosti s rozvojem informačních technologií ukázalo, že klasická kryptografie založená na symetrických kryptosystémech nedostačuje rostoucím nárokům na pružnost a bezpečnost přenosu informací. Za předpokladu, že není použit nějaký protokol dohody na klíči, je v symetrických kryptosystémech před vlastním odesláním zašifrované zprávy nutné, aby odesilatel poslal příjemci použitý klíč. K poslání klíče by měl být použit jiný (bezpečnější) kanál než pro poslání zprávy. Další nevýhoda symetrických kryptosystémů spočívá ve skutečnosti, že každá dvojice komunikujících partnerů musí mít vlastní klíč. Pokud uvážíme nějakou rozsáhlejší počítačovou síť, snadno zjistíme, že důsledkem jsou velmi vysoké nároky na správu klíčů. Uvedené problémy byly vyřešeny s využitím teorie složitosti tak, že byly vytvořeny kryptosystémy založené na výpočetně těžkých problémech.

Kryptosystém nazveme asymetrickým, jestliže pro libovolnou dvojici klíčů (e, d) takovou, že d je odpovídající dešifrovací klíč k šifrovacímu klíči e, nelze určit d pouze na základě znalosti e. Asymetrické kryptosystémy bývají také často označovány jako kryptosystémy s veřejným klíčem, šifrovací klíče označujeme jako veřejné klíče a dešifrovací klíče jako soukromé klíče.

Řekneme, že funkce f z množiny X do množiny Y je jednosměrná, pokud pro každé $x \in X$ je snadné vypočítat (v polynomiálním čase) hodnotu f(x) a pro dané $y \in Y$ je výpočetně obtížné najít takové $x \in X$, že f(x) = y. Jako příklady jednosměrných funkcí uvedeme následující funkce:

- násobení prvočísel – pro konečnou posloupnost prvočísel je lehké najít jejich součin, avšak pro velké přirozené číslo n je obtížné nalézt jeho rozklad na prvočísla (problém faktorizace přirozených čísel).
- modulární umocňování – pro prvočíslo p, přirozené číslo m a nezáporné celé číslo a lze jednoduše určit $0 \leq y < p$ tak, že $y \equiv m^a \pmod p$, ale pro prvočíslo p a přirozená čísla m, y je náročné nalézt nezáporné celé číslo a takové, že $y \equiv m^a \pmod p$ (problém diskrétního logaritmu).
- modulární výpočet druhé mocniny – pro přirozená čísla x, n lze jednoduše určit $0 \leq y < n$ tak, že $y \equiv x^2 \pmod n$, ale pro přirozená čísla y, n je obtížné vypočítat $1 \leq x < n$ takové, že $y \equiv x^2 \pmod n$ (problém diskrétní druhé odmocniny).

Ve třídě jednosměrných funkcí existují funkce $f : X \rightarrow Y$ takové, že na základě nějaké dodatečné informace je snadné k danému $y \in Y$ najít takové $x \in X$, že f(x) = y. Jednosměrnou funkci

s touto vlastností nazýváme jednosměrná funkce se zadními vrátky. Příkladem takové funkce je modulární výpočet druhé mocniny pro pevné n, jelikož problém diskrétní druhé odmocniny lze snadno vyřešit, známe-li rozklad modulu n na prvočísla. Význam jednosměrných funkcí se zadními vrátky spočívá v jejich vhodnosti pro tvorbu asymetrických kryptosystémů.

Při použití asymetrického kryptosystému k zabezpečení komunikace v rámci skupiny subjektů má každý subjekt vlastní veřejný klíč e a vlastní soukromý klíč d. Pro každou takovou dvojici klíčů platí, že pro libovolnou zprávu m je $D_d(E_e(m)) = m$, kde D_d je dešifrovací funkce určená klíčem d a E_e je šifrovací funkce daná klíčem e. Z vlastností asymetrického kryptosystému plyne, že veřejné klíče mohou být zveřejněny a utajovat je třeba pouze soukromé klíče. Komunikace mezi subjekty A, B, jež je zabezpečena použitím asymetrického kryptosystému, probíhá v následujících krocích. Subjekt A nejprve nalezne veřejný klíč náležející subjektu B, dále zašifruje odesílanou zprávu pomocí tohoto klíče a konečně odešle šifru subjektu B, který po přijetí šifry rekonstruuje zaslanou zprávu užitím svého soukromého klíče. Žádný jiný subjekt nemůže šifru dešifrovat, poněvadž podle předpokladu pouze ze znalosti šifry a veřejného klíče nelze určit soukromý klíč.

Pokud by veřejné klíče byly zveřejněny bez jakéhokoliv zabezpečení, asymetrický kryptosystém by se mohl stát snadným terčem následujícího útoku. Předpokládejme, že veřejné klíče jsou uloženy ve veřejně přístupném souboru. Potom útočník může v tomto souboru nahradit veřejný klíč subjektu A svým vlastním veřejným klíčem, čímž získá schopnost dešifrovat libovolnou zprávu určenou subjektu A. Po přečtení zprávy ji útočník zašifruje původním veřejným klíčem subjektu A, jemuž následně pošle šifru. Jak A tak i jednotliví odesilatelé zpráv v tomto scénáři útok nedetekují a budou předpokládat, že komunikují důvěrně. Aby byl tento typ útoku, znemožněn vstupují do hry subjekty označované jako důvěryhodné třetí strany (Trusted Third Parties, TTP). TTP je vybavena soukromým klíčem, který je utajován, a veřejným klíčem, jenž je znám všem subjektům, nějakého digitálního podepisovacího systému. Veřejný klíč subjektu A je potom do souboru uložen společně s řetězcem, který vznikne aplikací soukromého klíče TTP na zprávu tvořenou v nejjednodušším případě jménem subjektu A a jeho veřejným klíčem poté, co TTP ověřila identitu subjektu A. Tento řetězec nazýváme certifikát. Subjekt, který chce odeslat zprávu subjektu A, tak učiní pouze tehdy, pokud pomocí veřejného klíče TTP transformuje příslušný certifikát na řetězec tvořený jménem subjektu A a jeho veřejným klíčem. Problémem popsaného postupu je, že dojde-li k narušení důvěryhodnosti TTP, je narušena bezpečnost veškeré komunikace v rámci systému.

Na rozdíl od symetrických kryptosystémů asymetrické kryptosystémy umožňují bezprostřední komunikaci ve smyslu, že odeslání zprávy nemusí předcházet žádná dohoda na klíči mezi odesilatelem a příjemcem. Navíc při použití symetrického kryptosystému roste celkový počet klíčů kvadraticky vzhledem k počtu uživatelů, zatímco u asymetrického kryptosystému je tento nárůst lineární. Další výhodou asymetrických kryptosystémů je jejich snazší rozšiřitelnost o nové uživatele. Na druhé straně symetrické kryptosystémy jsou obvykle rychlejší a používají kratší klíče. V současnosti jsou často oba typy kryptosystémů používány společně tak, aby byly co nejvíce využity jejich výhody a eliminovány jejich nevýhody. V tomto případě typicky šifrování zpráv probíhá pomocí symetrického kryptosystému a dohoda na použitém klíči pomocí kryptosystému asymetrického.

3.3.1 Kryptosystém RSA

Kryptosystém RSA (Rivest, Shamir, Adleman), který byl vyvinut v roce 1978, je v současnosti nejpoužívanějším asymetrickým kryptosystémem. Jeho bezpečnost je založena na neexistenci efektivního algoritmu pro faktorizaci přirozených čísel.

Každý subjekt S si zvolí dvě velká prvočísla p, q přibližně stejné bitové délky a vypočítá jejich součin n. Následně si S zvolí přirozené číslo d takové, že $1 < d < \varphi(n) = (p - 1)(q - 1)$ a d je nesoudělné s $\varphi(n)$. Konečně S vypočítá přirozené číslo e tak, že $1 < e < \varphi(n)$ a $ed \equiv 1 \pmod{\varphi(n)}$. Veřejným klíčem subjektu S je dvojice (n, e), soukromým klíčem číslo d.

Předpokládejme, že subjekt A chce poslat zprávu m subjektu B. Označme veřejný klíč subjektu B (n, e) a jeho soukromý klíč d. A nejdříve převede zprávu m na celé číslo w, které patří do intervalu $[0, n - 1]$, posléze vypočítá šifru $c = w^e \bmod n$ a odešle c subjektu B. B po přijetí šifry c vypočítá $c^d \bmod n$ a získanou hodnotu převede zpět na původní zprávu m.

Dále dokážeme korektnost dešifrování. Ukážeme, že pro daný veřejný klíč (n = pq, e) a k němu náležející soukromý klíč d platí vztah $w^{ed} \equiv w \pmod{n}$, kde w je libovolné nezáporné celé číslo menší než n. Protože $ed \equiv 1 \pmod{\varphi(n)}$, existuje přirozené číslo k takové, že $1 + k\varphi(n) = ed$. Předpokládejme nejprve, že p ani q nedělí w. Pak nsd(n, w) = 1 a z Eulerovy věty (pro přirozené číslo n a celé číslo a nesoudělné s n platí vztah $a^{\varphi(n)} \equiv 1 \pmod{n}$, kde $\varphi(n)$ je počet všech přirozených čísel menších než n a nesoudělných s n) dostáváme $w^{ed} \equiv w \pmod{n}$. Dále může nastat situace, že právě jedno z dvojice prvočísel p, q dělí w. Předpokládejme tedy, že p dělí w. Potom máme $w^{ed} \equiv 0 \equiv w \pmod{p}$. Z Eulerovy věty obdržíme vztah $w^{ed} \equiv w \pmod{q}$. Dohromady dostáváme $w^{ed} \equiv w \pmod{n}$, čímž je důkaz ukončen, jelikož není možné, aby obě prvočísla dělila w.

3.3.2 Kryptosystém ElGamal

Kryptosystém, jehož bezpečnost je založena na neexistenci efektivního algoritmu pro řešení problému diskrétního logaritmu.

Každý subjekt S si zvolí velké prvočíslo p a přirozené číslo q takové, že $[q]_p$ je generátorem grupy $(\{[1]_p, [2]_p, \ldots, [p - 1]_p\}, \cdot)$ a $1 \le q < p$. Dále si S zvolí přirozené číslo x, $1 \le x < p$, a vypočítá přirozené číslo $y = q^x \bmod p$. Veřejným klíčem subjektu S je trojice (p, q, y), soukromým klíčem číslo x.

Opět předpokládáme, že subjekt A posílá zprávu m subjektu B. A nejprve vyhledá veřejný klíč subjektu B, označme jej (p, q, y), a převede zprávu m na celé číslo w takové, že $0 \le w < p$. Dále si A vybere přirozené číslo k takové, že $1 \le k \le p - 2$, vypočítá čísla $a = q^k \bmod p$ a $b = y^k w \bmod p$ a pošle šifru c = (a, b) subjektu B. B po přijetí šifry c s použitím svého soukromého klíče x vypočítá $ba^{-x} \bmod p$ a obdrženou hodnotu převede zpět na původní zprávu m.

Důkaz korektnosti dešifrování povedeme následujícím způsobem. Jelikož $y = q^x \bmod p$, platí vztah $ba^{-x} \equiv q^{kx} w q^{-kx} \equiv w \pmod{p}$.

3.3.3 Rabinův kryptosystém

Tento kryptosystém byl vytvořen na základě problému faktorizace přirozených čísel. Ve srovnání s výše uvedenou dvojicí kryptosystémů umožňuje výrazně rychlejší šifrování.

Každý subjekt S si zvolí dvě velká prvočísla p, q taková, že $p \equiv q \equiv 3 \pmod{4}$ a bitová délka p je přibližně rovna bitové délce q, a vypočítá jejich součin n. Veřejným klíčem subjektu S je číslo n, soukromým klíčem dvojice (p, q).

Nechť subjekt A posílá zprávu m subjektu B. Veřejný klíč subjektu B označíme n a jeho soukromý klíč (p, q). A si vyjádří zprávu m jako číslo w z množiny $\{0, 1, \ldots, n - 1\}$, vypočítá šifru $c = w^2 \bmod n$ a poté pošle c subjektu B. B po přijetí šifry c vypočítá čísla $r = c^{(p + 1)/4} \bmod p$ a $s = c^{(q + 1)/4} \bmod q$, následně najde celá čísla a, b splňující rovnost $ap + bq = 1$ a vypočítá hodnoty $x = (aps + bqr) \bmod n$ a $y = (aps - bqr) \bmod n$. Protože n je součinem dvou prvočísel, existují právě čtyři druhé odmocniny z c modulo n, jimiž jsou čísla x, y, –x mod n a –y mod n. Aby byl B

schopen z této čtveřice správně identifikovat číslo w, obsahuje obvykle toto číslo nějakou předem danou redundantní informaci (např. replikace určitého počtu posledních bitů), která jej s vysokou pravděpodobností jednoznačně určí. Je-li m smysluplná zpráva v přirozeném jazyce, lze ji snadno určit převedením všech čtyř odmocnin do přirozeného jazyka, aniž by bylo nutné používat redundantní informaci.

V důkazu korektnosti dešifrování ukážeme, že B správně vypočítá čtyři druhé odmocniny z c modulo n. Z rovnosti $c = w^2 \bmod n$ plynou vztahy $c \equiv w^2 \pmod p$ a $c \equiv w^2 \pmod q$. Dále umocníme číslo r modulo p a dostaneme $r^2 \equiv c^{(p+1)/2} \equiv w^{p+1} \pmod p$. Z Eulerovy věty potom získáme vztah $r^2 \equiv c \pmod p$. Obdobným postupem obdržíme vztah $s^2 \equiv c \pmod q$. Důsledkem jsou vztahy $r^2 \equiv c \pmod n$ a $s^2 \equiv c \pmod n$. Nyní umocníme x modulo n a dostaneme $x^2 \equiv a^2p^2s^2 + 2apsbqr + b^2q^2r^2 \equiv c(a^2p^2 + b^2q^2) \pmod n$. Dále $(ap + bq)^2 = a^2p^2 + 2apbq + b^2q^2 = 1 \equiv a^2p^2 + b^2q^2 \pmod n$, a proto platí $x^2 \equiv c \pmod n$. Podobně také $y^2 \equiv c \pmod n$. Poněvadž $-x \bmod n = n - x$, dostáváme $(-x)^2 \bmod n = (n - x)^2 \bmod n = n^2 - 2nx + x^2 \bmod n = x^2 \bmod n$. Ze stejného důvodu platí rovnost $(-y)^2 \bmod n = y^2 \bmod n$.

3.4 Digitální podpisy

Digitální podpisy jsou základním prostředkem pro zajištění autentizace dat a nepopiratelnosti. Umožňují digitálně podepsat digitální zprávu tak, že kdokoliv (případně pouze příjemce zprávy) může podpis ověřit. Navíc ověřením podpisu subjektu A subjekt nezíská žádnou informaci o tom, jak podepisovat zprávy podpisem subjektu A. Na rozdíl od ručně psaného podpisu je digitální podpis pevně spojen s podepsanou zprávou a pro dvě různé zprávy podepsané týmž subjektem jsou obvykle jejich podpisy různé. Podstata digitálního podepisování spočívá v tom, že k dané zprávě je připojen bitový řetězec, který tuto zprávu jednoznačně svazuje s nějakým subjektem.

Digitální podepisovací systém (DPS) je pětice (M, S, K, P, V), kde M je množina možných zpráv, S je množina možných podpisů a K množina možných klíčů. Dále P označuje množinu funkcí z M do S takovou, že pro libovolné $k \in K$ existuje právě jedna funkce $sig_k \in P$. Množina V obsahuje funkce z $M \times S$ do {true, false}. Pro každé $k \in K$ existuje jediná funkce $ver_k \in V$, jež je definována následovně. Pokud $s = sig_k(w)$, pak $ver_k(w, s) = true$, jinak $ver_k(w, s) = false$.

Podobně jako u asymetrických kryptosystémů má každý subjekt A svůj veřejný a soukromý klíč. Pomocí tajného soukromého klíče A generuje podpisy ke zprávám. Veřejný klíč je užíván jinými subjekty k ověření, zda daný podpis byl skutečně vygenerován subjektem A. Digitální podepisování a ověřování podpisu tedy probíhají podobně jako asymetrické šifrování, rozdíl je pouze v pořadí, v němž jsou aplikovány jednotlivé klíče daného subjektu. Důsledkem je, že na základě mnoha asymetrických kryptosystémů je možné vytvořit DPS. Přesněji formulováno pro asymetrický kryptosystém musí platit, že množina zpráv je rovna množině šifer. V tomto případě totiž pro každou dvojici klíčů (e, d) takovou, že d je odpovídající soukromý klíč k veřejnému klíči e, jsou odpovídající funkce E_e a D_d permutacemi na množině zpráv, z čehož dostáváme pro každou zprávu m platnost rovnosti $D_d(E_e(m)) = E_e(D_d(m)) = m$. Příkladem DPS vzniklého z asymetrického kryptosystému je podepisovací systém RSA, který se od kryptosystému RSA liší pouze tím, že subjekt A podepisuje zprávu m užitím svého soukromého klíče a ověření podpisu je prováděno aplikací veřejného klíče subjektu A a porovnáním získaného výsledku se zprávou m.

3.4.1 Podepisovací systém DSA

DSA (Digital Signature Algorithm) je první digitální podepisovací systém, který byl na vládní úrovni akceptován jako standard. Stalo se tak roku 1994 v USA. Bezpečnost algoritmu je založena na obtížnosti řešení problému diskrétního logaritmu.

Každý subjekt A si nejprve zvolí l-bitové prvočíslo p, kde $512 \leq l \leq 1024$ a $l = 64k$ pro nějaké přirozené číslo k. Dále si A zvolí 160-bitové prvočíslo q, které dělí $(p - 1)$, a přirozené číslo h takové, že $1 \leq h < p$. Posléze vypočítá číslo $r = h^{(p-1)/q} \bmod p$. Je-li $r = 1$, A opakuje volbu h a výpočet r. Nakonec si A vybere přirozené číslo x, $x < p$, a vypočítá $y = r^x \bmod p$. Veřejným klíčem subjektu A je čtveřice (p, q, r, y), soukromým klíčem číslo x.

Předpokládejme, že subjekt A posílá zprávu m subjektu B. Veřejný klíč subjektu A označíme (p, q, r, y) a jeho soukromý klíč x. Pomocí vhodné hašovací funkce h A převede zprávu m na 160-bitové přirozené číslo w. Následně si zvolí přirozené číslo k menší než q. Poté vypočítá přirozená čísla $a = (r^k \bmod p) \bmod q$ a $b = k^{-1}(w + ax) \bmod q$. Konečně A pošle subjektu B zprávu m opatřenou podpisem (a, b). Po přijetí zprávy (m, a, b) subjekt B pomocí funkce h vypočítá čísla $z = b^{-1} \bmod q$, $u = wz \bmod q$ a $v = az \bmod q$. B akceptuje podpis, právě když je splněna rovnost $a = (r^u y^v \bmod p) \bmod q$.

V důkazu korektnosti předpokládejme, že (a, b) je podpis, kterým subjekt A podepsal zprávu m. Potom s využitím vztahu $b = k^{-1}(w + ax) \bmod q$ dostáváme kongruenci $w \equiv kb - ax \pmod q$. Vynásobením obou stran kongruence číslem z obdržíme vztah $zw + azx \equiv k \pmod q$, který lze dále upravit na tvar $u + vx \equiv k \pmod q$. Protože $r = h^{(p-1)/q} \bmod p$, dostáváme $(r^u y^v \bmod p) \bmod q = (r^k \bmod p) \bmod q$, a tedy $a = (r^u y^v \bmod p) \bmod q$.

3.4.2 Ong-Schnor-Shamirův podepisovací systém

Podepisovací systém, který vedle vlastního podepisování umožňuje také přenos tajné informace mezi komunikujícími subjekty. Tato informace je použita k vygenerování podpisu, z něhož ji lze při znalosti soukromého klíče opět získat. Systém může najít uplatnění především v situacích, kdy z nějakého důvodu není možné použít šifrování, a přesto subjekty chtějí komunikovat důvěrně. Bezpečnost systému je založena na obtížnosti problému faktorizace přirozených čísel.

Každý subjekt A si zvolí velké přirozené číslo n takové, že n je liché a n není prvočíslo, a přirozené číslo k nesoudělné s n. Následně A vypočítá číslo $h = k^{-2} \bmod n$. Veřejným klíčem subjektu A je dvojice (n, h), soukromým klíčem číslo k.

Předpokládejme, že subjekt A chce poslat subjektu B tajnou zprávu m. Veřejný klíč subjektu A označíme (n, h) a jeho soukromý klíč k. A převede zprávu m na číslo w, které je prvkem množiny $\{0, 1, \ldots, n - 1\}$ a vybere si nedůležitou zprávu n, kterou převede na přirozené číslo v. Obě čísla v, w musí být nesoudělná s n. Posléze A vypočítá čísla $S_1 = 2^{-1}(vw^{-1} + w) \bmod n$ a $S_2 = k2^{-1}(vw^{-1} - w) \bmod n$ a pošle subjektu B zprávu v doplněnou podpisem (S_1, S_2). Jakmile B obdrží zprávu (v, S_1, S_2), ověří platnost podpisu zjištěním, zda je splněna kongruence $v \equiv (S_1)^2 - h(S_2)^2 \pmod n$, a vypočítá $v(S_1 + k^{-1}S_2)^{-1} \bmod n$. Nakonec B získanou hodnotu převede zpět na původní zprávu m.

Dále dokážeme korektnost systému. Platnost vztahu $v \equiv (S_1)^2 - h(S_2)^2 \pmod n$ plyne ze skutečnosti, že $h = k^{-2} \bmod n$. Korektnost dešifrování ukážeme následovně. Součet $S_1 + k^{-1}S_2 \bmod n$ je roven číslu $vw^{-1} \bmod n$, jehož inverzním prvkem modulo n je číslo $wv^{-1} \bmod n$. S využitím komutativity modulárního násobení dostáváme $vwv^{-1} \bmod n = vv^{-1}w \bmod n = w$.

3.5 Hašovací funkce

Hašovací funkce je funkce taková, že jejím definičním oborem je množina všech binárních řetězců a jejím oborem hodnot je množina binárních řetězců pevné délky n. Potom je možné, aby pomocí hašovací funkce byla informace libovolné délky reprezentována informací pevné délky, kterou nazýváme hašovací kód. Z důvodu kryptografické použitelnosti požadujeme, aby hašovací funkce h měla následující tři vlastnosti. Musí být výpočetně obtížné nalézt různé řetězce x, y

takové, že h(x) = h(y). Dále pro daný řetězec x musí být výpočetně obtížné najít odlišný řetězec y takový, že h(x) = h(y). Konečně musí být funkce h jednosměrná, tedy pro daný hašovací kód y musí být výpočetně obtížné najít řetězec x takový, že h(x) = y.

Kryptografie nejčastěji využívá hašovací funkce při digitálním podepisování a pro zajištění integrity dat. Hašovací funkce, při jejichž výpočtu se uplatňuje tajný klíč, jsou také použitelné pro dosažení autentizace dat. Funkcím tohoto typu říkáme klíčované hašovací funkce (Message Authentication Codes, MAC).

Při digitálním podepisování je obvykle na posílanou zprávu aplikována nějaká veřejně známá hašovací funkce a podepisován získaný hašovací kód. Příjemce zprávy nejprve rovněž vypočítá její hašovací kód, pomocí něhož následně ověří přijatý podpis. Toto řešení především pro dlouhé zprávy snižuje časovou a paměťovou náročnost procesů podepisování a ověřování, neboť dlouhá zpráva obvykle musí být rozdělena na bloky menší délky, z nichž každý je podepsán samostatně. Navíc je bezpečnější podepisovat hašovací kód, nikoliv jednotlivé části dlouhé zprávy.

Hašovací funkce použitelné pro zajištění integrity dat nazýváme Modification Detection Codes (MDC). Tyto funkce jsou obvykle veřejně známé a při výpočtu nevyžadují žádný klíč. S MDC pracujeme následujícím způsobem. V určitém čase vypočítáme hašovací kód k pro nějaká data a provedeme taková opatření, aby jej neautorizovaným způsobem nebylo možné změnit. Později pro ověření, zda data nebyla změněna, stačí opět vypočítat jejich hašovací kód a porovnat jej s hašovacím kódem k. Výsledkem je redukce problému zachování integrity potenciálně dlouhé zprávy na tentýž problém pro hašovací kód pevné délky.

Klíčovaná hašovací funkce je čtveřice (M, T, K, H), kde M je množina možných zpráv, T je množina možných hašovacích kódů a K množina možných klíčů. Písmenem H značíme množinu funkcí z M do T takovou, že pro každý klíč $k \in K$ existuje právě jedna funkce $h_k \in H$. Předpokládejme, že subjekt A chce poslat subjektu B zprávu m za použití nějaké klíčované hašovací funkce. Nejdříve se A a B dohodnou na tajném klíči k a potom A pošle B zprávu $(m, h_k(m))$. B po přijetí zprávy (m, t) vypočítá $s = h_k(m)$. Pokud s = t, B akceptuje zprávu m jako autentickou, v opačném případě ji odmítne. Uvedený postup vede k dosažení integrity, což plyne z vlastností hašovacích funkcí, a autentizace dat, neboť za předpokladu, že pouze A a B znají klíč k, žádný neoprávněný subjekt není schopen správně vypočítat hašovací kód pro svou zprávu. Naopak tento postup nezajišťuje důvěrnost, aby jí bylo dosaženo, musí být odesílána zašifrovaná zpráva.

3.6 Generátory náhodných čísel

Bezpečnost mnoha kryptosystémů je založena na výběru a následném použití nepředvídatelných hodnot. Příkladem je třeba klíč v kryptosystému DES nebo prvočísla p, q v kryptosystému RSA. V těchto případech vybraná hodnota musí být náhodná ve smyslu, že pravděpodobnost jejího výběru je dostatečně malá a pravděpodobnost výběru každé hodnoty je stejná..

Generátor náhodných bitů je zařízení nebo algoritmus, jehož výstupem je posloupnost hodnot z množiny {0, 1} taková, že znalost prvního až i-tého prvku této posloupnosti neposkytne žádnou informaci o hodnotě prvku číslo i + 1. O takové posloupnosti pak říkáme, že je náhodná. Generátor náhodných bitů může být také používán jako generátor náhodných čísel. Generátory tohoto typu obvykle pracují na fyzikálním principu a pro praktické uplatnění jsou většinou příliš pomalé. Navíc je často nutné vygenerované posloupnosti bezpečně uchovávat pro další použití a přenášet (např. jako klíč v DES), což může být při jejich větších délkách problematické. Z uvedených důvodů jsou místo generátorů náhodných bitů používány generátory pseudonáhodných bitů.

Generátor pseudonáhodných bitů je deterministický algoritmus, který na základě náhodné bitové posloupnosti délky k vygeneruje bitovou posloupnost délky $l \gg k$, která je výpočetně nerozlišitelná od náhodné posloupnosti. Výstup pak nazýváme pseudonáhodná posloupnost.

Algoritmus použitý pro generování bývá často všeobecně znám, utajovaná bývá pouze vstupní posloupnost.

4. Kryptografické funkce v Java Core API

4.1 Architektura kryptografických funkcí

Kryptografické třídy a rozhraní v Java Core API byly navrženy tak, aby bylo možné provádět nejrůznější kryptografické operace (např. digitální podepisování, generování klíčů, …) nezávisle na jejich implementaci a na použitém algoritmu. Algoritmové nezávislosti je dosaženo definováním tříd (engine classes), které poskytují funkcionalitu pro základní kryptografické operace. Jako příklady lze uvést třídy java.security.Signature (dále jen Signature) či javax.crypto.Cipher. Implementační nezávislosti je dosaženo pomocí třídy java.security.Provider, jejíž podtřídy (providery) fungují jako jakési seznamy dostupných kryptografických funkcí. Pokud se v programu pokusíme vytvořit např. objekt třídy Signature implementující algoritmus DSA, jsou prohledány přítomné providery a v případě nalezení třídy, jež implementuje tento algoritmus, je požadovaný objekt vytvořen. Je také možné uvést nejen název algoritmu, ale i jméno nějakého provideru, potom je prohledáván pouze uvedený provider. V dalším budeme pojmem provider označovat dohromady podtřídu třídy Provider a množinu všech tříd, které jsou v této podtřídě uvedeny.

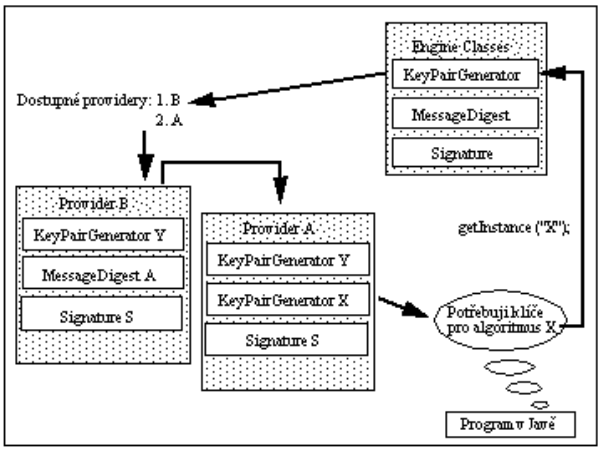
Engine classes definují základní kryptografické operace na abstraktní úrovni. Představují rozhraní, pomocí kterého lze přistupovat ke kryptografickému algoritmu určitého typu. Nabízí se otázka, jaké podmínky musí splňovat např. třída implementující digitální podpis DSA, abychom s ní mohli takto pracovat. Tato třída musí být deklarována jako potomek abstraktní třídy java.Security.SignatureSpi, musí tedy překrýt všechny její abstraktní metody. Uvedené tvrzení platí pro všechny engine classes, ke každé z nich existuje abstraktní třída, jež se nachází ve stejném balíku jako daná engine class a jejíž název se liší pouze příponou Spi, a třídy implementující konkrétní algoritmus je třeba deklarovat jako potomka příslušné abstraktní třídy. Podle konvence názvy všech metod v každé takové abstraktní třídě začínají slovem engine. Úspěšně vytvořená instance některé engine class pro nějaký algoritmus obsahuje jako soukromou proměnnou (proměnná deklarovaná s modifikátorem přístupu private) instanci potomka příslušné třídy typu Spi, jenž implementuje požadovaný algoritmus. Metody dané engine class potom volají odpovídající metody potomka třídy typu Spi.

Většina engine classes poskytuje pro vytváření instancí trojici metod, které v jednotlivých třídách pracují na naprosto stejném principu a liší se pouze návratovým typem. Jako reprezentanta opět zvolíme třídu Signature, pokud v kapitolách věnovaných jednotlivým třídám nebude řečeno jinak, k vytváření objektů dané třídy je určena až na návratový typ stejná trojice metod jako v případě třídy Signature:

- static Signature getInstance (String algorithm)
- static Signature getInstance (String algorithm, String provider)
- static Signature getInstance (String algorithm, Provider provider)

Unární metoda prohledává všechny dostupné providery v pořadí daném jejich prioritou, dokud nenajde třídu implementující požadovaný algoritmus. Obě binární metody prohledávají pouze uvedený provider. Pro všechny tři uvedené metody platí, že pokud při hledání neuspějí, jejich běh končí vytvořením výjimky java.security.NoSuchAlgorithmException. Jako parametr metod getInstance je nutné uvést standardní jméno algoritmu, tedy jméno, pod kterým se daný algoritmus

objevuje v providerech. Následující obrázek ilustruje zjednodušenou formou činnost unární metody getInstance:



Obrázek 4.1: Činnost metod pro vytváření instancí engine classes

Standardní distribuce Javy obsahuje několik providerů, z nichž nejvýznamnější jsou Sun a SunJCE. Součástí provideru Sun jsou třídy pro digitální podepisování a neklíčované hašování. Nástroje pro práci s těmito třídami najdeme v balíku java.security a jeho podbalících. SunJCE zahrnuje třídy umožňující šifrování a klíčované hašování, s nimiž můžeme pracovat pomocí tříd nacházejících se v balíku javax.crypto a jeho podbalících. Standardní jména algoritmů, které jsou pro vybrané engine classes dostupné ve standardní distribuci nejnovější verze Javy, uvádí tabulka 3.1. K jejímu obsahu uveďme, že jak DESede, tak TripleDES označuje algoritmus 3-DES.

Engine class	Standardní jména algoritmů
Cipher	AES, ARCFOUR, Blowfish, DES, DESede, PBEWithMD5AndDES, PBEWithMD5AndTripleDES, PBEWithSHA1AndDESede, PBEWithSHA1AndRC2_40, RC2, RSA
KeyAgreement	DiffieHellman
KeyFactory	DiffieHellman, DSA, RSA
KeyGenerator	AES, ARCFOUR, Blowfish, DES, DESede, HmacMD5, HmacSHA1, HmacSHA256, HmacSHA384, HmacSHA512, RC2
KeyPairGenerator	DiffieHellman, DSA, RSA
Mac	HmacMD5, HmacPBESHA1, HmacSHA1, HmacSHA256, HmacSHA384, HmacSHA512
MessageDigest	MD2, MD5, SHA, SHA-256, SHA-384, SHA-512
SecretKeyFactory	DES, DESede, PBE, PBEWithMD5AndDES, PBEWithMD5AndTripleDES, PBEWithSHA1AndDESede, PBEWithSHA1AndRC2_40
SecureRandom	SHA1PRNG
Signature	MD2WithRSA, MD5AndSHA1WithRSA, MD5WithRSA, NoneWithDSA, SHA1WithDSA, SHA1WithRSA, SHA256WithRSA, SHA384WithRSA, SHA512WithRSA

Tabulka 4.1: Standardní jména algoritmů poskytovaných standardní distribucí Javy 2, v1.5.0

4.2 Správa klíčů

Klíče jsou v Javě reprezentovány dvěma základními způsoby. První způsob se uplatňuje při provádění kryptografických operací, kdy jsou používány objekty tříd, jež implementují rozhraní java.security.Key. Tyto třídy obvykle neumožňují přímý přístup k jednotlivým složkám klíče, přístup uživatele bývá omezen na trojici metod definovaných rozhraním Key:

- String getAlgorithm ( ) – vrátí jméno algoritmu, pro který je daný klíč určen.
- byte [ ] getEncoded ( ) – vrátí daný klíč jako pole typu byte [ ].
- String getFormat ( ) – vrátí jméno formátu, jenž je pro převod klíče na pole využíván.

Rozhraní Key má několik potomků. Žádné z těchto rozhraní ovšem nepřidává žádnou metodu, jejich role spočívá v jednoznačné identifikaci různých druhů klíčů. Nejvýznamnější z nich jsou rozhraní java.security.PublicKey, java.security.PrivateKey a javax.crypto.SecretKey.

Druhý způsob reprezentace klíčů je uživatelsky přívětivější, neboť dovoluje uživatelům nejen přistupovat k jednotlivým částem klíče, ale i pomocí konstruktorů klíče samostatně vytvářet. Klíče tohoto typu se ale neuplatňují při provádění kryptografických operací, musí být převedeny na instanci třídy, jež implementuje rozhraní Key. Třídy umožňující pracovat s klíči uvedeným způsobem musí implementovat rozhraní java.security.spec.KeySpec, jež neobsahuje žádnou metodu. Ze tříd, které implementují rozhraní KeySpec zmíníme abstraktní třídu java.security.spec.EncodedKeySpec. Tato třída je velmi jednoduchá, obsahuje konstruktor EncodedKeySpec (byte [ ] encodedKey) a následující metody:

- byte [ ] getEncoded ( )
- abstract String getFormat ( )

Instance potomků třídy EncodedKeySpec se používají pro uchovávání klíčů převedených na pole hodnot typu byte.

Při vytváření klíčů pro asymetrické algoritmy jsou vždy veřejný a odpovídající soukromý klíč vytvořeny současně. Balík java.security obsahuje třídu KeyPair, jejíž instance jsou tvořeny veřejným a příslušným soukromým klíčem. K vytváření objektů třídy KeyPair slouží konstruktor KeyPair (PublicKey publicKey, PrivateKey privateKey). Třída KeyPair poskytuje dvě metody:

- PublicKey getPublic ( )
- PrivateKey getPrivate ( )

4.2.1 Třída java.security.KeyPairGenerator

Objekty této třídy generují dvojice klíčů pro asymetrické kryptosystémy a digitální podepisování a vrací je jako instance třídy KeyPair. K vytvoření objektu třídy KeyPairGenerator lze použít jednu z výše uvedených metod getInstance. Vytvořený objekt musí být před vlastním generováním klíčů inicializován. Uživatel může buď nechat inicializaci na daném objektu, který ji provede implicitně, nebo jej může inicializovat explicitně zavoláním některé z metod initialize.

Explicitně lze objekt třídy KeyParGenerator inicializovat dvěma způsoby. Podstatou prvního způsobu je, že uživatel určuje pouze požadovanou délku klíče a zbývající parametry potřebné pro generování si daný objekt vytvoří samostatně. Délka klíče je pro různé algoritmy interpretována různě, např. pro DSA jde o bitovou délku modulu p. Inicializaci tímto způsobem provedeme zavoláním jedné z následující dvojice metod:

- void initialize (int keysize) – v průběhu generování použije generátor pseudonáhodných hodnot z provideru s nejvyšší priorotou mezi všemi providery, které obsahují nějakou třídu sloužící ke generování pseudonáhodných čísel. Jestliže žádný provider takovou třídu neobsahuje, bude použit systémem poskytnutý generátor.
- void initialize (int keysize, SecureRandom random) – během generování použije generátor random.

Pro situace, v nichž je třeba specifikovat parametry použité při generování klíčů, je určen druhý způsob inicializace. Specifikaci parametrů provádíme pomocí objektů tříd, které implementují rozhraní java.security.spec.AlgorithmParameterSpec. K předání parametrů jsou určeny tyto metody:

- void initialize (AlgorithmParameterSpec params)
- void initialize (AlgorithmParameterSpec params, SecureRandom random)

Je-li objekt třídy KeyPairGenerator inicializován, je schopen generovat dvojice klíčů pomocí následující dvojice metod:

- final KeyPair genKeyPair ( )
- KeyPair generateKeyPair ( )

Tyto metody jsou funkčně ekvivalentní a jejich opakovaná volání dávají různé dvojice klíčů. Modifikátor final zde označuje metodu, která nesmí být potomky třídy KeyPairGenerator překryta.

4.2.2 Třída.javax.crypto.KeyGenerator

Třída KeyGenerator je určena ke generování klíčů pro symetrické kryptosystémy a klíčované hašovací funkce. Její instanci dostaneme pomocí výše uvedených metod getInstance. Podobně jako objekty třídy KeyPairGenerator také instance třídy KeyGenerator je třeba před vytvářením klíčů inicializovat, což může být provedeno buď implicitně, nebo explicitně.

K explicitní inicializaci lze přistoupit obdobnými dvěma způsoby jako v případě třídy KeyPairGenerator. Při inicializaci prvním způsobem ovšem oproti třídě KeyPairGenerator přibývá možnost předat danému objektu pouze generátor pseudonáhodných čísel:

- void init (SecureRandom random) – za použití generátoru random bude vytvořen klíč implicitní délky.
- void init (int keysize)
- void init (int keysize, SecureRandom random)

Předání parametrů objektu třídy KeyGenerator lze provést pomocí jedné z těchto metod:

- void init (AlgorithmParameterSpec params)
- void init (AlgorithmParameterSpec params, SecureRandom random)

Po inicializaci dojde k vytvoření nového klíče zavoláním metody SecretKey generateKey ( ). Opět platí, že daný inicializovaný objekt třídy KeyGenerator generuje při opakovaných voláních této metody různé klíče.

4.2.3 Třída javax.crypto.spec.SecretKeySpec

Při realizaci kryptografických operací je často nutné uchovávat klíče na disku nebo je nějakým způsobem přenášet od jednoho uživatele ke druhému. Pro zajištění existence objektů mimo JVM Java poskytuje techniku serializace. Tato technika spočívá v převodu objektu na posloupnost hodnot typu byte, která zachycuje obsah objektu, a uložení této posloupnosti do souboru na disk. Serializovat lze pouze objekty tříd, které implementují rozhraní java.io.Serializable, o takových objektech říkáme, že jsou serializovatelné. Serializovatelné objekty jsou trvalé ve smyslu, že mají schopnost přečkat ukončení běhu JVM a při dalším spuštění se znovu objevit v nezměněné podobě. Proces rekonstrukce serializovaného objektu nazýváme deserializace.

Častěji jsou ovšem klíče uchovávány a přenášeny ve formě polí typu byte [ ]. Nejjednodušší způsob, jak převést pole hodnot typu byte na objekt třídy, jež implementuje rozhraní SecretKey, nabízí třída SecretKeySpec, jejíž instanci obdržíme použitím jednoho z konstruktorů:

- SecretKeySpec (byte [ ] key, String algorithm) – vytvoří klíč pro uvedený algoritmus na základě pole key.
- SecretKeySpec (byte [ ] key, int offset, int len, String algorithm) – vytvoří klíč pro uvedený algoritmus na základě len hodnot z pole key, počínaje hodnotou na pozici offset.

Poznamenejme, že uvedené konstruktory je vhodné používat pouze pro takové algoritmy, jejichž klíče samy mají povahu pole hodnot typu byte. Příkladem takových algoritmů jsou klíčované hašovací funkce nebo kryptosystém DES.

4.2.4 Třída javax.crypto.SecretKeyFactory

K provedení složitějších převodů mezi klíči pro symetrické kryptosystémy a poli typu byte [ ] třída SecretKeySpec nestačí, je třeba použít třídu SecretKeyFactory. Tato třída poskytuje funkcionalitu pro převody mezi objekty tříd implementujících rozhraní SecretKey a objekty tříd, které implementují rozhraní KeySpec. Objekt třídy SecretKeyFactory obdržíme pomocí některé z výše uvedených metod getInstance.

Pro převod objektu třídy, jež implementuje rozhraní KeySpec, na objekt třídy implementující rozhraní SecretKey je určena metoda SecretKey generateSecret (KeySpec keySpec). Tato metoda na základě informace obsažené v objektu keySpec vytvoří instanci třídy implementující rozhraní SecretKey. Jako příklad použití metody generateSecret uvedeme převod pole hodnot typu byte na klíč pro kryptosystém DES:

```
SecretKeyFactory skf = SecretKeyFactory.getInstance ("DES");
KeySpec ks = new DESKeySpec (keyBytes);
SecretKey desKey = skf.generateSecret (ks);
```

Konstruktor DESKeySpec (byte [ ] key) vytvoří na základě prvních osmi hodnot pole key instanci třídy javax.crypto.spec.DESKeySpec. Objekty této třídy slouží k uchovávání klíčů pro DES převedených na pole hodnot typu byte.

Opačný převod lze realizovat pomocí metody KeySpec getKeySpec (SecretKey key, class keySpec), která vytvoří instanci třídy keySpec z klíče key. Např. klíč pro DES převedeme na pole hodnot typu bytes následujícím způsobem:

```
SecretKeyFactory skf = SecretKeyFactory.getInstance ("DES");
DESKeySpec dks = (DESKeySpec) skf.getKeySpec (desKey, DESKeySpec.class);
```

byte [ ] keyBytes = dks.getKey ( );

4.2.5 Třída java.security.KeyFactory

Třída KeyFactory je analogií třídy SecretKeyFactory pro asymetrické kryptosystémy a digitální podpisy. Její instanci dostaneme použitím jedné z obvyklé trojice metod getInstance.

Máme-li objekt třídy,která implementuje rozhraní KeySpec, převedeme jej na buď na soukromý, nebo na veřejný klíč zavoláním odpovídající metody z této dvojice metod:

- PrivateKey generatePrivate (KeySpec keySpec)
- PublicKey generatePublic (KeySpec keySpec)

Opačného převodu dosáhneme pomocí metody KeySpec getKeySpec (Key key, class keySpec), jež funguje stejně jako stejnojmenná metoda třídy SecretKeyFactory.

4.2.6 Třída javax.crypto.KeyAgreement

Třída KeyAgreement poskytuje metody pro realizaci protokolů dohody na klíči. S parametry, jež se v těchto protokolech uplatňují, a také s průběžnými výsledky je zacházeno stejně, jako by šlo o klíče asymetrických kryptosystémů, např. k vytvoření parametrů lze tedy použít objekt třídy KeyPairGenerator. Uvedený jev můžeme ilustrovat na příkladu implementace Diffie-Hellmanova protokolu obsažené v provideru SunJCE. V této implementaci je soukromým klíčem uživatele A jeho exponent x, veřejným klíčem hodnota q^x mod p, kde význam čísel p, q je stejný jako v části 3.2.3, a průběžné výsledky jsou považovány za veřejné klíče. Instanci třídy KeyAgreement je možné získat pomocí výše uvedených metod getInstance.

Po vytvoření musí být objekt třídy KeyAgreement inicializován s použitím soukromého klíče, který bude posléze sloužit pro výpočet výsledné hodnoty. Pro inicializaci jsou určeny tyto metody:

- void init (Key key) – inicializuje daný objekt pro provedení protokolu dohody na základě parametrů obsažených v objektu key. Případné použití generátoru pseudonáhodných hodnot je řízeno stejnými pravidly jako je tomu u metody void initialize (int keysize) třídy KeyPairGenerator.
- void init (Key key, SecureRandom random) – inicializuje daný objekt pro provedení protokolu dohody na základě parametrů obsažených v objektu key za použití generátoru random.
- void init (Key key, AlgorithmParameterSpec params) – inicializuje daný objekt pro provedení protokolu dohody na základě parametrů obsažených v objektech key a params. Případné použití generátoru pseudonáhodných čísel je řízeno pravidly uvedenými u unární metody init.
- void init (Key key, AlgorithmParameterSpec params, SecureRandom random) – inicializuje daný objekt pro provedení protokolu dohody na základě parametrů obsažených v objektech key a params za použití generátoru random.

Každý protokol dohody na klíči definuje určitý počet kroků, které musí být provedeny každým účastníkem protokolu. Provedení následujícího kroku dosáhneme zavoláním metody Key doPhase (Key key, boolean lastPhase). Parametr key označuje objekt třídy, jež implementuje rozhraní Key. Tento objekt je buď veřejným klíčem, nebo průběžným výsledkem některého z účastníků protokolu. Parametr lastPhase indikuje, jde-li o poslední krok protokolu. Po provedení posledního kroku protokolu dostane každý účastník sdílenou hodnotu použitím některé z následujících metod:

- byte [ ] generateSecret ( ) – vrátí sdílenou hodnotu jako pole hodnot typu byte. Toto pole je možné následně použít k vytvoření klíče požadovaného typu.
- int generateSecret (byte [ ] sharedSecret, int offset) – zapíše sdílenou hodnotu do pole sharedSecret, přičemž zápis probíhá od pozice offset. Vrátí počet takto zapsaných hodnot typu byte.
- SecretKey generateSecret (String algorithm) – vrátí klíč pro uvedený algoritmus.

4.3 Šifrování a dešifrování

4.3.1 Třída javax.crypto.Cipher

Tato třída je jádrem balíku javax.crypto. Poskytuje metody pro šifrování a dešifrování dat. Je použitelná jak nad symetrickými, tak nad asymetrickými kryptosystémy. Java standardně umožňuje používat kryptosystémy AES, DES, 3-DES, Blowfish a v nejnovější verzi také RSA, RC2 a ARCFOUR. Kryptosystémy DES, 3-DES a RC2 jsou poskytovány nejen ve standardní, ale také v modifikované podobě založené na principu PBE (Password-Based Encryption). Tento princip nevyžaduje uchovávání a přenášení klíčů, neboť při každém šifrování či dešifrování je klíč nově vytvořen na základě hašovacího kódu uživatelem zadaného hesla. Instance třídy Cipher jsou vytvářeny pomocí těchto metod:

- static Cipher getInstance (String transformation)
- static Cipher getInstance (String transformation, String provider)
- static Cipher getInstance (String transformation, Provider provider)

Uvedené metody se od standardních metod getInstance liší pouze tím, že umožňují uživateli podrobněji specifikovat, jakým způsobem mají být data transformována. Řetězec transformation může mít dvojí podobu:

- "algorithm/mode/padding" (příklad: "DES/CBC/PKCS5Padding")
- "algorithm" (příklad: "Blowfish")

V prvním případě uvádíme postupně algoritmus, režim činnosti šifrovače a způsob doplnění posledního bloku. Pokud chceme použít šifrovač v režimu CFB nebo OFB, bezprostředně za názvem režimu je třeba uvést požadovanou délku bloku, jinak bude použita implicitní hodnota. Vedle čtyř základních režimů standardní distribuce Javy nabízí i režim Propagating Cipher Block Chaining (PCBC). Jde o režim odvozený z režimu CBC, ve kterém šifra bloku m_i má podobu $c_i = k(m_{i-1} \oplus c_{i-1} \oplus m_i)$. Ve druhém případě uvádíme pouze název algoritmu, režim a způsob doplnění jsou určeny implicitně na základě definice příslušné třídy.

Při použití blokových šifrovačů se často stává, že délka šifrované zprávy není násobkem délky bloku. Potom je nutné poslední část zprávy vhodným způsobem doplnit tak, aby dosáhla délky bloku. Příkladem řešení problému posledního bloku je algoritmus PKCS#5, jenž nad řetězci hodnot typu byte pracuje tak, že poslední blok na kýženou délku doplní potřebným počtem čísel typu byte, z nichž hodnota každého je rovna počtu takto přidávaných čísel. Je-li délka zprávy rovna nějakému násobku délky bloku, potom je zpráva doplněna o celý blok takový, že hodnota všech čísel je rovna délce bloku.

Standardní název	Význam
NONE	Žádný režim
ECB	Režim Electronic Code Book
CBC	Režim Cipher Block Chaining
PCBC	Režim Propagating Cipher Block Chaining.
CFBn	Režim Cipher FeedBack o délce bloku n
OFBn	Režim Output FeedBack o délce bloku n
NoPadding	Poslední bloky nejsou doplňovány
PKCS5Padding	Algoritmus PKCS#5
ISO10126Padding	Algoritmus ISO10126

Tabulka 4.2: Standardní jména režimů činnosti blokových šifrovačů a algoritmů pro doplnění posledního bloku poskytovaných standardní distribucí Javy 2, v1.5.0

Abychom mohli objekt třídy Cipher použít pro šifrování nebo dešifrování musíme jej nejprve inicializovat pomocí jedné z dvojice konstant třídy Cipher:

- static final int ENCRYPT_MODE
- static final int DECRYPT_MODE

Inicializaci objektu třídy Cipher lze provést zavoláním jedné z následujících metod:

- void init (int opmode, Key key)
- void init (int opmode, Key key, SecureRandom random)
- void init (int opmode, Key key, AlgorithmParameterSpec params)
- void init (int opmode, Key key, AlgorithmParameterSpec params, SecureRandom random)
- void init (int opmode, Certificate certificate)
- void init (int opmode, Certificate certificate, SecureRandom random)
- void init (int opmode, Key key, AlgorithmParameters params)
- void init (int opmode, Key key, AlgorithmParameters params, SecureRandom random)

Na základě hodnoty parametru opmode bude objekt třídy Cipher inicializován buď pro šifrování, nebo pro dešifrování. V obou případech budou data transformována pomocí klíče, který je buď uveden přímo jako parametr key, nebo je obsažen v parametru certificate. V průběhu transformace je v případě potřeby používán buď generátor pseudonáhodných čísel, který je získán způsobem popsaným u metody void initialize (int keysize) třídy KeyPairGenerator, nebo generátor random. Zbývající argumenty představují parametry, jež jsou některými algoritmy vyžadovány pro jejich korektní funkci. Jako příklad uveďme algoritmus DES v režimu jiném než ECB, u něhož je v režimu dešifrování nezbytné uvést inicializační vektor, pomocí kterého byla data zašifrována. Pro úplnost dodejme, že v režimu šifrování není třeba inicializační vektor uvádět, v tomto případě je použita buď náhodně vygenerovaná, nebo implicitní hodnota. Použitý inicializační vektor vrací užitečná metoda byte [ ] getIV ( ). Jestliže zavoláme některou z metod init, dojde k resetování daného objektu, který přejde do stejného stavu, jako by byl nově vytvořen a inicializován.

Šifrování a dešifrování v Javě probíhá buď nad poli hodnot typu byte, nebo nad instancemi třídy java.nio.ByteBuffer, při šifrování dat jiného typu musí být tedy tato data převedena na objekt, který uvedenou podmínku splňuje. Transformace dat může probíhat dvěma základními způsoby, buď v jednom kroku, nebo ve více krocích. Druhou možnost využijeme např. tehdy, když předem

neznáme délku dat, s nimiž budeme pracovat. Data transformujeme v jednom kroku pomocí jedné z následujících metod:

- byte [ ] doFinal (byte [ ] input) – transformuje obsah pole input a všechna data uložená v rámci daného objektu třídy Cipher. Vrátí pole naplněné transformovanými daty.
- byte [ ] doFinal (byte [ ] input, int inputOffset, int inputLen) – transformuje všechna data uložená v rámci daného objektu a inputLen hodnot z pole input, počínaje hodnotou na pozici inputOffset. Vrátí pole naplněné transformovanými daty.
- int doFinal (byte [ ] input, int inputOffset, int inputLen, byte [ ] output) – transformuje všechna data uložená v rámci daného objektu a inputLen hodnot z pole input, počínaje hodnotou na pozici inputOffset. Transformovaná data zapíše do pole output a vrátí počet takto zapsaných dat.
- int doFinal (byte [ ] input, int inputOffset, int inputLen, byte [ ] output, int outputOffset) – transformuje všechna data uložená v rámci daného objektu a inputLen hodnot z pole input, počínaje hodnotou na pozici inputOffset. Transformovaná data zapíše do pole output, přičemž zápis proběhne od pozice outputOffset. Vrátí počet takto zapsaných hodnot.
- int doFinal (ByteBuffer input, ByteBuffer output) – transformuje všechna data uložená v rámci daného objektu a všechna data z objektu input nacházející se mezi pozicí, na kterou ukazuje ukazatel, a limitem. Transformovaná data zapíše do objektu output, přičemž zápis proběhne od pozice ukazatele. Vrátí počet takto zapsaných hodnot. Metodu lze použít pouze v nejnovější verzi Javy, což platí i pro všechny dále uvedené metody, jejichž parametry jsou objekty třídy ByteBuffer.

Pro transformaci ve více krocích lze použít tyto metody:

- byte [ ] update (byte [ ] input)
- byte [ ] update (byte [ ] input, int inputOffset, int inputLen)
- int update (byte [ ] input, int inputOffset, int inputLen, byte [ ] output)
- int update (byte [ ] input, int inputOffset, int inputLen, byte [ ] output, int outputOffset)
- int update (ByteBuffer input, ByteBuffer output)

Metody pro transformaci ve více krocích pracují analogicky jako výše uvedené metody doFinal, jediný rozdíl tkví ve skutečnosti, že při své činnosti nevyužívají algoritmu pro doplnění posledního bloku. Jestliže nějaká metoda update transformuje takový počet hodnot typu byte, který není roven násobku délky bloku, neúplný poslední blok je uložen v rámci objektu třídy Cipher a transformován, až když nastane jedna ze dvou možností. Buď bude následujícími voláními metod update nebo doFinal dodáno dostatečné množství hodnot, aby mohlo dojít k transformaci dat uložených v rámci objektu třídy Cipher, nebo tato data budou doplněna použitím algoritmu pro doplnění posledního bloku a poté transformována následkem volání některé metody doFinal. Vedle dosud uvedených metod lze transformaci dat ve více krocích ukončit také zavoláním jedné z následujících metod:

- byte [ ] doFinal ( ) – transformuje všechna data uložená v rámci daného objektu. Vrátí pole naplněné transformovanými daty.
- int doFinal (byte [ ] output, int outputOffset) – transformuje všechna data uložená v rámci daného objektu. Výsledek transformace zapíše do pole output, přičemž zápis proběhne od pozice outputOffset. Vrátí počet takto zapsaných hodnot.

Pro všechny metody doFinal platí, že po jejich zavolání daný objekt třídy Cipher přejde do stavu, v němž byl po posledním volání některé z metod init. Pokud transformujeme data pomocí některé z metod, jimž jako parametr předáváme výstupní objekt, je vhodné před vlastní transformací

zavolat metodu int getOutputSize (int inputLen), kde inputLen je počet hodnot ze vstupního objektu, které mají být transformovány, jež vrátí odhadovanou délku výstupu za předpokladu, že data budou transformována pomocí některé z metod doFinal.

4.3.2 Třída javax.crypto.CipherInputStream

Třída CipherInputStream umožňuje šifrování nebo dešifrování vstupního proudu binárních dat. Objekty této třídy se skládají z instance nějakého potomka abstraktní třídy java.io.InputStream a instance třídy Cipher. K jejich vytváření slouží tyto konstruktory:

- CipherInputStream (InputStream is) – vytvoří objekt třídy CipherInputStream na základě vstupního proudu is a instance třídy javax.crypto.NullCipher, jež implementuje šifrovač, který šifruje data aplikací funkce identita.
- CipherInputStream (InputStream is, Cipher c) – vytvoří objekt třídy CipherInputStream na základě vstupního proudu is a šifrovače c, který musí být inicializován.

K transformaci vstupního proudu dat je určena následující trojice metod, které nejprve načtou data z příslušného proudu a následně je předají objektu třídy Cipher:

- int read ( ) – vrátí následující byte z transformovaného proudu dat. Tato hodnota je vrácena jako číslo typu int náležející do intervalu [0, 255].
- int read (byte [ ] b) – naplní zadané pole hodnotami z transformovaného proudu a vrátí celkový počet do pole zapsaných hodnot. Je-li parametr b odkazem na neexistující pole, dojde k vymazání následujících b.length hodnot z transformovaného proudu (b.length je kapacita pole b).
- int read (byte [ ] b, int off, int len) – zapíše do pole b len hodnot z transformovaného proudu, přičemž zápis proběhne od pozice off. Vrátí celkový počet do pole zapsaných hodnot. Pokud je parametr b odkazem na neexistující pole, dojde k vymazání následujících len hodnot z transformovaného proudu.

Všechny tři uvedené metody vrací číslo –1, není-li možné vrátit následující hodnotu typu byte nebo zápsat data, protože bylo dosaženo konce proudu. Jako ukázku použití třídy CipherInputStream uvedeme zdrojový kód, který pomocí kryptosystému AES zašifruje obsah souboru plaintext.txt a šifru zapíše do souboru cryptotext.txt:

```
Cipher c = Cipher.getInstance ("AES");
c.init (Cipher.ENCRYPT_MODE, aesKey);
FileInputStream fis;
FileOutputStream fos;
CipherInputStream cis;
fis = new FileInputStream("C:/plaintext.txt");
cis = new CipherInputStream(fis, c);
fos = new FileOutputStream("C:/cryptotext.txt");
byte[ ] b = new byte[8];
int i = cis.read(b);
while (i != – 1) {
    fos.write(b, 0, i);
    i = cis.read(b);
}
```

4.3.3 Třída javax.crypto.CipherOutputStream

Vedle vstupních proudů existují v Javě pro binární data také výstupní proudy. Pro šifrování a dešifrování výstupního proudu binárních dat je určena třída CipherOutputStream. Objekty této třídy jsou tvořeny instancí potomka abstraktní třídy java.io.OutputStream a instancí třídy Cipher a obdržíme je pomocí následujících konstruktorů, jež pracují stejně jako konstruktory třídy CipherInputStream:

- CipherOutputStream (OutputStream os)
- CipherOutputStream (OutputStream os, Cipher c)

Metody transformující výstupní proud dat nejprve předají data ke zpracování objektu třídy Cipher a výstup zapíší do příslušného výstupního proudu:

- void write (int b) – předá k transformaci hodnotu b jako číslo typu byte.
- void write (byte [ ] b) – předá k transformaci veškerý obsah pole b.
- void write (byte [ ] b, int off, int len) – předá k transformaci len hodnot z daného pole, počínaje hodnotou na pozici off.

Z dalších metod třídy CipherOutputStream zmíníme metodu void flush ( ), jež vyprázdní daný výstupní proud. Tato metoda zapíše na místo určení všechny hodnoty typu byte, které byly zpracovány objektem třídy Cipher. Hodnoty typu byte uchovávané v objektu třídy Cipher a čekající na zpracování nebudou transformovány. Chceme-li dosáhnout transformace i těchto hodnot a zapsání výsledku, musíme buď dodat objektu třídy Cipher další data pomocí metod write, nebo použít metodu void close ( ), jež zavolá některou z metod doFinal objektu třídy Cipher, následně zavolá metodu flush a nakonec uzavře daný výstupní proud a uvolní všechny systémové zdroje s ním spojené. Pro srovnání uveďme, jak lze šifrovat obsah souboru pomocí třídy CipherOutputStream:

```
Cipher c = Cipher.getInstance ("AES");
c.init (Cipher.ENCRYPT_MODE, aesKey);
FileInputStream fis;
FileOutputStream fos;
CipherOutputStream cos;
fis = new FileInputStream("C:/plaintext.txt");
fos = new FileOutputStream("C:/cryptotext.txt");
cos = new CipherOutputStream(fos, c);
byte[ ] b = new byte[8];
int i = fis.read(b);
while (i != – 1) {
    cos.write(b, 0, i);
    i = fis.read(b);
}
cos.close( );
```

4.3.4 Třída javax.crypto.SealedObject

Pomocí třídy Cipher jsme schopni šifrovat javové objekty nepřímo, před vlastním šifrováním je musíme převést na typ byte [ ]. Přímočařejší cestu pro šifrování serializovatelných objektů

poskytuje třída SealedObject. Instance této třídy fungují jako jakési schránky pro jiné objekty. Uchovávané objekty jsou zašifrovány, aby byla zajištěna jejich důvěrnost..

Instanci třídy SealedObject vytvoříme zavoláním konstruktoru SealedObject (Serializable object, Cipher c). Tento konstruktor převede objekt object do serializované formy, která je následně zašifrována pomocí šifrovače c. Poznamenejme, že objekt c musí být inicializován pro šifrování. Všechny parametry, které byly použity při šifrování, jsou uloženy v rámci vzniklého objektu třídy SealedObject.

Z daného objektu třídy SealedObject získáme uchovávaný objekt pomocí jedné z trojice metod. Všechny tři metody dešifrují obsah daného objektu a vrátí deserializovaný výsledek tohoto procesu, jednotlivé metody se ovšem liší tím, jakým způsobem vytvářejí instanci třídy Cipher použitou při dešifrování:

- Object getObject (Cipher c) – objekt c je nutné korektně inicializovat pro dešifrování uchovávaného objektu.
- Object getObject (Key key) – parametrem je klíč určený pro dešifrování uchovávaného objektu. Na základě klíče key a informací uložených v daném objektu třídy SealedObject vytvoří instanci třídy Cipher a inicializuje ji pro dešifrování.
- Object getObject (Key key, String provider) – pracuje stejně jako předchozí metoda, ale při vytváření objektu třídy Cipher prohledává pouze uvedený provider.

4.4 Digitální podpisy

4.4.1 Třída java.security.Signature

Tato třída je určena pro elektronické podepisování a ověřování podpisů. Obvyklá distribuce Javy obsahuje třídy implementující podepisovací systémy DSA a RSA. Jak plyne z tabulky 3.1, pomocí DSA lze podepisovat buď samotnou zprávu, nebo její hašovací kód získaný aplikací některé neklíčované hašovací funkce, naopak v případě RSA je vždy podepisován hašovací kód. Je třeba upozornit, že uvedený výčet standardních jmen je použitelný v nejnovější verzi Javy, ale například v předcházející verzi 1.4.2 jsou dostupné pouze algoritmy označené standardními názvy MD2WithRSA, MD5WithRSA, SHA1WithDSA a SHA1WithRSA. Instanci třídy Signature obdržíme zavoláním jedné z trojice metod getInstance uvedených výše.

K rozlišení, je-li daný objekt třídy Signature používán pro generování nebo ověřování podpisu, slouží následující celočíselné konstanty:

- static final int UNITIALIZED
- static final int SIGN
- static final int VERIFY

Konstanta UNINITIALIZED představuje stav instance třídy Signature bezprostředně po vytvoření, význam zbývající dvojice konstant je zřejmý. Před vlastním podepisováním či ověřováním musí být objekt třídy Signature inicializován pomocí některé z těchto metod:

- void initSign (PrivateKey privateKey) – inicializuje daný objekt pro podepisování pomocí zadaného soukromého klíče.
- void initSign (PrivateKey privateKey, SecureRandom random) – inicializuje daný objekt pro podepisování pomocí daného soukromého klíče a generátoru random.
- void initVerify (PublicKey publicKey) – inicializuje daný objekt pro ověřování podpisu za použití daného veřejného klíče.

- void initVerify (Certificate certificate) – inicializuje daný objekt pro ověřování podpisu pomocí veřejného klíče obsaženého v objektu certificate.

Případné nastavení parametrů pro daný podepisovací systém je možné provést zavoláním metody void setParameter (AlgorithmParameterSpec params).

Jestliže jsme objekt třídy Signature inicializovali buď pro podepisování, nebo pro ověřování, musíme mu dále poskytnout data, nad kterými bude pracovat. Toho dosáhneme použitím následujících metod:

- void update (byte b) – přidá k aktuálním datům byte b.
- void update (byte [ ] data) – přidá k aktuálním datům obsah pole data.
- void update (byte [ ] data, int off, int len) – přidá k aktuálním datům len hodnot ze zadaného pole, počínaje hodnotou na pozici off.
- void update (ByteBuffer data) – přidá k aktuálním datům všechny hodnoty z objektu data, které se nacházejí se mezi pozicí, na níž ukazuje ukazatel, a limitem.

Předpokládejme, že jsme objekt třídy Signature inicializovali pro podepisování a dodali jsme mu všechna podepisovaná data. Podpis potom dostaneme, zavoláme-li jednu z následující dvojice metod:

- byte [ ] sign ( ) – vrátí výsledný podpis jako pole hodnot typu byte.
- int sign (byte [ ] outbuf, int offset, int len) – zapíše podpis do pole outbuf, přičemž zápis proběhne od pozice offset a pro podpis je vyhrazeno len pozic. Vrátí počet skutečně zapsaných hodnot

Obě uvedené metody resetují daný objekt třídy Signature a vrátí jej do stavu, ve kterém byl po posledním volání některé z metod initSign. V tomto stavu může být daný objekt znovu inicializován pomocí některé z inicializačních metod.

Před ověřením podpisu je třeba instanci třídy Signature patřičným způsobem inicializovat a předat jí pomocí metod update data, ke kterým podpis náleží. Podpis pak ověříme pomocí některé z těchto metod:

- boolean verify (byte [ ] signature)
- boolean verify (byte [ ] signature, int offset, int length)

Argumentem obou metod je pole signature obsahující ověřovaný podpis. V případě unární metody je podpisem veškerý obsah pole, naopak v případě ternární metody je za podpis považováno jen length hodnot z pole signature, počínaje hodnotou na pozici offset. Obě metody vrátí true, právě když byl podpis úspěšně ověřen. Jejich zavoláním dojde k resetování daného objektu, který přejde do stavu, v němž byl po posledním zavolání některé z metod initVerify. Opět platí, že v tomto stavu může být daný objekt znovu inicializován pomocí některé z inicializačních metod.

4.4.2 Třída java.security.SignedObject

SignedObject je třída použitelná pro dosažení integrity a autenticity javových objektů. Instance této třídy obsahuje nějaký serializovatelný objekt v serializované podobě a k němu náležející podpis. K vytváření objektů třídy SignedObject je určen konstruktor SignedObject (Serializable object, PrivateKey signingKey, Signature signingEngine), který provede serializaci daného objektu a takto získaný objekt podepíše pomocí klíče signingKey a objektu signingEngine.

Podpis uchovávaný v rámci instance třídy SignedObject lze ověřit zavoláním metody boolean verify (PublicKey verificationKey, Signature verificationEngine). Tato metoda vrátí true, právě když podpis obsažený v objektu třídy SignedObject je platným podpisem pro objekt uchovávaný tamtéž. Poznamenejme, že objekt třídy Signature nemusí být inicializován, což platí také pro výše uvedený konstruktor.

Logickým požadavkem je, aby objekt obsažený v instanci třídy SignedObject bylo možné získat v jeho původní podobě. Tento účel plní metoda Object getObject ( ), jež uchovávaný objekt deserializuje a vrátí jej.

4.5 Hašovací funkce

4.5.1 Třída java.security.MessageDigest

Třída MessageDigest umožňuje práci s neklíčovanými hašovacími funkcemi. Standardně Java nabízí několik hašovacích funkcí ze skupiny SHA, dále algoritmus MD5 a v nejnovější verzi také MD2. Instanci třídy MessageDigest vytvoříme zavoláním některé z výše uvedených metod getInstance.

Vytvořený objekt je třeba naplnit daty, jejichž hašovací kód má být počítán. To může být provedeno pomocí následujících metod:

- void update (byte input) – přidá k aktuálním datům hodnotu input.
- void update (byte [ ] input) – přidá k aktuálním datům obsah pole input.
- void update (byte [ ] input, int offset, int len) – přidá k aktuálním datům len hodnot ze zadaného pole, počínaje hodnotou na pozici offset.
- void update (ByteBuffer input) – přidá k aktuálním datům všechny hodnoty ze zadaného objektu, které se nacházejí mezi pozicí určenou ukazatelem a limitem.

K vlastnímu výpočtu hašovacího kódu jsou určeny tyto metody:

- byte [ ] digest ( ) – vrátí vypočítaný hašovací kód.
- byte [ ] digest (byte [ ] input) – přidá k aktuálním datům obsah pole input a vrátí vypočítaný hašovací kód.
- int digest (byte [ ] buf, int offset, int len) – zapíše vypočítaný hašovací kód do zadaného pole. Zápis proběhne od pozice offset a pro kód je vyhrazeno len pozic. Vrátí počet ve skutečnosti zapsaných hodnot.

Všechny tři metody resetují daný objekt třídy MessageDigest tak, že jej je možné následně použít k výpočtu hašovacího kódu pro nová vstupní data. Instance třídy MessageDigest mohou být resetovány kdykoliv v průběhu své existence, k tomuto účelu je určena metoda void reset ( ).

4.5.2 Třída java.security.DigestInputStream

Tato třída je obdobou třídy CipherInputStream, binární data procházející vstupním proudem ovšem nešifruje, nýbrž umožňuje vypočítat jejich hašovací kód. Objekty třídy DigestInputStream jsou tvořeny instancí nějakého potomka abstraktní třídy InputStream a instancí třídy MessageDigest. Vytváříme je pomocí konstruktoru DigestInputStream (InputStream stream, MessageDigest digest).

Objekty třídy DigestInputStream mohou pracovat ve dvou základních režimech, mezi nimiž lze přecházet pomocí metody void on (boolean on). Implicitně po každém načtení dat z proudu stream následuje jejich předání instanci třídy MessageDigest. Pokud ale zavoláme metodu on s argumentem false, načtená data nebudou předávána.

K načtení dat ze vstupního proudu stream a jejich následnému předání objektu třídy MessageDigest slouží tato dvojice metod:

- int read ( ) – načte následující byte ze vstupního proudu, předá jej případně objektu třídy MessageDigest a vrátí jako hodnotu typu int.
- int read (byte [ ] b, int off, int len) – zapíše len hodnot ze vstupního proudu do zadaného pole, přičemž zápis proběhne od pozice off. V závislosti na aktuálním režimu případně předá načtená data instanci třídy MessageDigest. Vrátí buď skutečný počet načtených hodnot, nebo číslo –1, jestliže v důsledku dosažení konce proudu nebyla načtena žádná data.

Poté, co ze vstupního proudu načteme data, dosáhneme vypočtení jejich hašovacího kódu zavoláním některé z metod digest objektu třídy MessageDigest.

4.5.3 Třída java.security.DigestOutputStream

Třída DigestOutputStream slouží pro výpočet hašovacího kódu binárních dat procházejících nějakým výstupním proudem. Její objekty se skládají z instance nějakého potomka třídy OutputStream a z instance třídy MessageDigest. Objekt třídy DigestOutputStream dostaneme použitím konstruktoru DigestOutputStream (OutputStream stream, MessageDigest digest).

Rozlišujeme dva základní režimy činnosti objektů třídy DigestOutputStream. Tyto režimy jsou analogické režimům popsaným v části věnované třídě DigestInputStream a k jejich přepínání opět slouží metoda void on (boolean on).

Zápis dat do příslušného výstupního proudu a jejich předání objektu třídy MessageDigest lze provést pomocí následujících metod:

- void write (int b) – zapíše zadané číslo jako hodnotu typu byte do výstupního proudu a předá jej případně objektu třídy MessageDigest.
- void write (byte [ ] b, int off, int len) – do výstupního proudu zapíše len hodnot z pole b, počína je hodnotou na pozici off. V závislosti na aktuálním režimu případně předá zapsaná data objektu třídy MessageDigest.

Stejně jako u předchozí třídy vypočteme hašovací kód zapisovaných dat, zavoláme-li některou z metod digest objektu třídy MessageDigest.

4.5.4 Třída javax.crypto.Mac

Mac je třída, která slouží k počítání klíčovaných hašovacích funkcí. Standardní distribuce Javy umožňuje používat klíčované funkce vytvořené na základě funkcí SHA-1, MD5 a v nejnovější verzi také SHA-256, SHA-384 a SHA-512. K vytváření objektů třídy Mac je určena obvyklá trojice metod getInstance.

Po vytvoření je třeba objekt třídy Mac inicializovat pomocí klíče, což lze provést zavoláním některé z následujících metod:

- void init (Key key)

- void init (Key key, AlgorithmParameterSpec params)

Argumentem obou metod je objekt třídy, jež implementuje rozhraní javax.crypto.SecretKey. V případě některých algoritmů (např. Hmac-MD5 a Hmac-SHA1) může být tento objekt naprosto libovolný a nezáleží na tom, pro který algoritmus byl původně určen. Binární metoda navíc umožňuje definovat další parametry, které se uplatní při výpočtu hašovacího kódu.

Data, nad kterými bude pracovat, jsou objektu třídy Mac poskytnuta použitím stejné čtveřice metod jako v případě třídy MessageDigest. K výpočtu jejich hašovacího kódu jsou potom určeny tyto metody:

- byte [ ] doFinal ( )
- byte [ ] doFinal (byte [ ] input)
- void doFinal (byte [ ] output, int outOffset)

První dvě metody fungují analogicky jako odpovídající metody digest třídy MessageDigest. Zbývající metoda vypočítá hašovací kód a zapíše jej do pole output, přičemž zápis probíhá od pozice outOffset. Všechny tři metody resetují daný objekt a převedou jej do stavu, v němž se nacházel po posledním volání některé z metod init. Podobně jako v případě třídy MessageDigest lze resetování též dosáhnout zavoláním metody void reset ( ).

4.6 Generátory náhodných čísel

Funkcionalitu pro generování náhodných a pseudonáhodných čísel poskytuje třída java.security.SecureRandom. Standardní distribuce Javy je vybavena implementací algoritmu SHA1PRNG, který je založen na postupném počítání hodnot h^i(seed), kde h představuje aplikaci hašovací funkce SHA-1, i je nezáporné celé číslo a seed je inicializační hodnota. Instanci třídy SecureRandom obdržíme zavoláním jedné z metod getInstance. Všechny tyto metody vrácí generátor, jenž není inicializován pomocí počáteční hodnoty. Alternativně lze objekt třídy SecureRandom vytvořit pomocí následujících konstruktorů:

- SecureRandom ( )
- SecureRandom (byte [ ] seed)

Oba konstruktory vytvoří objekt třídy SecureRandom pro algoritmus z instalovaného provideru s nejvyšší prioritou mezi všemi providery, jež obsahují nějakou třídu pro náhodné nebo pseudonáhodné generování. Zatímco nulární konstruktor se chová stejně jako výše uvedené metody getInstance, unární konstruktor inicializuje vytvořený generátor s použitím pole seed. K inicializaci instance třídy SecureRandom slouží také tyto metody:

- void setSeed(byte [ ] seed)
- void setSeed (long seed)

Uvedené metody lze volat kdykoliv v průběhu existence daného generátoru. Zadaná hodnota seed nenahradí aktuální inicializační hodnotu, nýbrž ji doplní, čímž je zajištěno, že nedojde ke snížení náhodnosti generovaných hodnot.

Pro vlastní generování náhodných hodnot je možné použít následující dvojici metod:

- int next (int numBits) – vrátí hodnotu typu int tvořenou uvedeným počtem náhodných bitů.
- void nextBytes (byte [ ] bytes) – naplní náhodnými hodnotami pole bytes.

Nakonec zmíníme dvojici metod, jež vrátí zadaný počet hodnot typu byte, pomocí nichž může být následně inicializován jiný generátor pseudonáhodných čísel:

- byte [ ] generateSeed (int numBytes)
- static byte [ ] getSeed (int numBytes)

5. Implementace kryptografických algoritmů

Součástí standardní distribuce Javy jsou třídy implementující několik základních kryptografických algoritmů, je ale zřejmé, že podstatně více algoritmů takto snadno dostupných není. Jestliže z nějakého důvodu potřebujeme použít algoritmus, který standardni distribucí Javy není podporován, máme dvě základní možnosti. Buď najdeme třídy, jež implementují požadovaný algoritmus, na webu, nebo je napíšeme vlastními silami. Základní informace o nejvýznamnějších providerech dostupných na webu nabízí následující tabulka:

Provider	Adresa	Některé algoritmy
Cryptix	http://www.cypherpunks.to/~cryptix	ElGamal (podpis), IDEA, MARS, RC6, SAFER, SKIPJACK, Square, Twofish,
IAIK	http://jce.iaik.tugraz.at/products/index.php	CAST128, GOST, MARS, Serpent, Twofish
Crypto-J	http://www.rsasecurity.com/products/bsafe/cryptoj.html	RC2, RC4, RC5, RSA
GNU Crypto	http://www.gnu.org/software/gnu-crypto	Anubis, Khazad, Serpent, Square, Twofish

Tabulka 5.1: Kryptografické providery

Pokud úspěšně vytvoříme nějaké kryptografické třídy a rozhodneme se k nim přistupovat pomocí engine classes prostřednictvím námi napsané podtřídy třídy Provider, dojde k tomu, že Java označí náš provider za nedůvěryhodný a odmítne jej používat. Toto je způsobeno faktem, že javové kryptografické providery musí být podepsány certifikační autoritou spojenou s firmou Sun. Získání požadovaného certifikátu není pro běžného uživatele zcela jednoduché, snazší cestou, jež vede k dosažení uvedeného cíle, je tedy pravděpodobně vhodná modifikace nějakého volně dostupného open-source provideru, jakým je např. Cryptix.

V této kapitole prezentujeme možnosti Javy při implementaci kryptografických algoritmů na příkladech Rabinova kryptosystému a Ong-Schnor-Shamirova podepisovacího systému. Často budeme používat operaci celočíselné dělení, budeme ji tedy začit symbolem /.

5.1 Rabinův kryptosystém

5.1.1 Třídy pro reprezentaci klíčů

Zopakujme si, že soukromým klíčem pro Rabinův kryptosystém je dvojice prvočísel p, q takových, že $p \equiv q \equiv 3 \pmod 4$, odpovídajícím veřejným klíčem je číslo $n = pq$. Tato čísla jsou obvykle příliš velká na to, abychom je v Javě mohli reprezentovat pomocí jednoduchých číselných typů. Pro práci s velkými čísly Java nabízí třídu java.math.BigInteger, která je pro naše potřeby vhodná také z toho důvodu, že obsahuje řadu metod pro modulární aritmetiku, ve zbytku této kapitoly tedy budeme pojmem číslo často rozumět objekt třídy BigInteger.

Klíče pro Rabinův kryptosystém budou představovat objekty tříd RabinPrivateKey a RabinPublicKey. Poněvadž obě uvedené třídy budeme deklarovat jako třídy implementující rozhraní Key (přesněji řečeno jeho potomky PrivateKey resp. PublicKey) a poněvadž s výjimkou metody getEncoded metody tohoto rozhraní budou v obou třídách definovány stejně, jejich společným předkem učiníme abstraktní třídu RabinKey. Navíc jelikož číslo n má význam nejen při šifrování, ale i při dešifrování, schránkou pro n budou právě objekty třídy RabinKey. Pro úplnost dodejme, že čísla p, q budeme uchovávat v objektech třídy RabinPrivateKey.

Z metod, které předepisuje rozhraní Key, stojí nejvíce za pozornost metoda getEncoded. Ve třídě RabinPublicKey tato metoda zavolá metodu byte [ ] getBytes (BigInteger b), které jako parametr předá číslo n, a vrátí takto získané pole. Metodě getBytes se budeme podrobněji věnovat později, prozatím pouze uveďme, že jde o metodu, která převádí objekty třídy java.math.BigInteger na pole hodnot typu byte. Zajímavějším úkolem je napsání metody getEncoded ve třídě RabinPrivateKey. V tomto případě převedeme na pole typu byte [ ] číslo p i číslo q a obě pole, jež dostaneme, zkopírujeme do výsledného pole:

```
public byte [ ] getEncoded( ) {
   byte [ ] a = RabinCipher.getBytes(p);
   byte [ ] b = RabinCipher.getBytes(q);
   byte [ ] c = new byte [a.length + b.length];
   System.arraycopy(a, 0, c, 0, a.length);
   System.arraycopy(b, 0, c, a.length, b.length);
   return c;
}
```

Pokud byl daný soukromý klíč vytvořen objektem třídy RabinKeyPairGenerator, bitová délka prvočísla q je buď rovná bitové délce prvočísla p, nebo je o jedničku větší. Důsledkem je, že uvedená implementace metody getEncoded umožňuje korektní rekonstrukci daného soukromého klíče.

V dosud uvedených třídách jsou všechny metody, které zpřístupňují čísla p, q nebo n, deklarovány s modifikátorem přístupu protected, což znamená, že mohou být volány pouze z dané třídy, ze všech potomků dané třídy a ze tříd nacházejících se ve stejném balíku jako daná třída. Abychom uživateli umožnili přímý přístup k jednotlivým složkám klíče, napíšeme třídy RabinPrivateKeySpec a RabinPublicKeySpec, které obě implementují rozhraní KeySpec. Třída RabinPrivateKeySpec má dvě proměnné, v nichž uchovává čísla p, q, v deklaraci třídy RabinPublicKeySpec se objevuje jediná proměnná, jež je určena pro číslo n. Pro obě třídy platí, že metody zpřístupňující jednotlivé proměnné jsou deklarovány s modifikátorem public, mohou tedy být volány odkudkoliv.

Dále budeme potřebovat objekty pro uchovávání klíčů převedených na pole. Příslušné třídy nazveme EncodedRabinPrivateKeySpec a EncodedRabinPrivateKeySpec. Obě třídy jsou potomky třídy EncodedKeySpec a obsahují pouze konstruktor a metodu String getFormat ( ).

5.1.2 Třída RabinKeyPairGenerator

Třídu RabinKeyPairGenerator stejně jako následující třídy RabinCipher a RabinKeyFactory budeme deklarovat jako potomka příslušné třídy typu Spi, konkrétně v tomto případě třídy KeyPairGeneratorSpi. První z dvojice metod třídy RabinKeyPairGenerator je metoda void initialize (int keysize, SecureRandom random), jež má totožnou funkci jako stejná metoda třídy KeyPairGenerator. Délkou klíče budeme rozumět bitovou délku modulu n, což nám mírně zkomplikuje

proces generování. Z důvodu, který vysvětlíme v následující kapitole, budeme generovat pouze takové klíče, jejichž délka patří do intervalu [21, 3092].

Generování klíčů je úkolem metody KeyPair generateKeyPair ( ), jež pracuje následovně. Nejprve s použitím metody třídy BigInteger static BigInteger probablePrime (int bitLength, Random rnd) vygenerujeme číslo p tak, že p je s velkou pravděpodobností prvočíslo, platí vztah $p \equiv 3 \pmod 4$ a jeho bitová délka je rovna číslu ks / 2, kde ks je požadovaná délka klíče. Proces generování prvočísla q je poněkud náročnější, neboť na základě požadované délky klíče a bitové délky prvočísla p nelze jednoznačně určit bitovou délku prvočísla q. Budeme tedy generovat čísla střídavě délky x a x + 1, kde x je bitová délka prvočísla p, tak dlouho, dokud nenarazíme na číslo q takové, že q je s velkou pravděpodobností prvočíslo, je kongruentní podle modulu 4 prvočíslu p a bitová délka součinu pq je rovna požadované délce klíče. Nakonec čísla p, q použijeme k vytvoření nových instancí tříd RabinPublicKey a RabinPrivateKey, ze kterých následně dostaneme objekt třídy KeyPair.

5.1.3 Třída RabinCipher

Tato třída představuje vlastní implementaci Rabinova kryptosystému. Aby mohla být potomkem třídy CipherSpi, musíme Rabinův kryptosystém implemenovat jako blokový šifrovač, který pracuje nad daty typu byte [ ]. Jak bylo zmíněno v části věnované kryptografii, aby bylo možné efektivně dešifrovat, je třeba do implementace Rabinova kryptosystému zabudovat určitou redundanci. Před šifrováním tedy provedeme replikaci l posledních hodnot typu byte, kde l = (délka bloku + 1) / 2. Naše implementace bude pracovat v režimu ECB, což znamená, že při různých výskytech bude daný blok šifrován stejně. K doplňování neúplných posledních bloků použijeme algoritmus PKCS#5.

Rabinův kryptosystém je postaven na modulární aritmetice, musíme tedy použít nějaké mechanismy pro převod polí hodnot typu byte na objekty třídy BigInteger a naopak. Pole na objekty třídy BigInteger budeme převádět pomocí konstruktoru BigInteger (int signum, byte [ ] magnitude). Parametr signum určuje znaménko vzniklého čísla tak, že –1 odpovídá zápornému číslu, 0 odpovídá nule a 1 kladnému číslu. Parametr magnitude reprezentuje absolutní hodnotu vytvořeného čísla zapsanou v soustavě o základu 256 (k záporným hodnotám typu byte je přičteno číslo 256), přičemž nejvýznamnější byte se nachází na pozici nula a nejméně významný na pozici magnitude.length – 1, kde length je proměnná udávající kapacitu daného pole.

Pro opačný převod poskytuje třída BigInteger metodu byte [ ] toByteArray ( ), jež vrací pole, které vznikne na základě bitové reprezentace daného čísla ve dvojkovém doplňkovém kódu. V modulární aritmetice pracujeme výhradně s nezápornými čísly, jejichž bitová reprezentace ve dvojkovém doplňkovém kódu je tvořena znaménkovým bitem 0 následovaným absolutní hodnotou. Z důvodu kompatibility s výše uvedeným konstruktorem potřebujeme převádět objekty třídy BigInteger na pole reprezentující jejich absolutní hodnotu. Tento požadavek metoda toByteArray splňuje, právě když bitová délka absolutní hodnoty daného čísla není násobkem osmi. V opačném případě tato metoda vrátí pole, které vedle vlastní absolutní hodnoty navíc obsahuje číslo 0 nacházející se na pozici nula. Aby nám převod čísel na pole fungoval korektně, použijeme pomocnou metodu byte [ ] getBytes (BigInteger b), která bude v případě potřeby nevhodné chování metody toByteArray korigovat.

Dále se budeme zabývat určením délky bloku zprávy a délky bloku šifry. Délku klíče opět označíme ks. Aby kryptosystém korektně fungoval, nesmí být blok zprávy po provedení replikace tvořen více než (ks – 1) / 8 hodnotami, jako délku bloku zprávy tedy zvolíme číslo (ks – 9) / 12. Při této volbě počet replikovaných hodnot nepřesáhne (ks + 15) / 24 a dohromady dostáváme, že délka bloku zprávy po provedení replikace bude nejvýše (3ks – 3) / 24. Z uvedených hodnot plyne omezení na délku klíče, aby šifrování mělo smysl, musí být ks > 20, a aby bylo možné

používat algoritmus PKCS#5, v důsledku malého rozsahu typu byte musí být ks < 12 · 257 + 9 = = 3093. Stejně jako délku bloku zprávy i délku bloku šifry definujeme v závislosti na délce klíče, hodnota délky bloku šifry je (ks + 7) / 8.

Jelikož ve zbytku této části budeme často používat názvy některých proměnných třídy RabinCipher, vysvětlíme nyní jejich význam. Pole data slouží k uchovávání dosud nezpracovaných hodnot typu byte, proměnná dataLength udává aktuální počet v poli data obsažených hodnot. Aktuální režim práce daného objektu je zachycen proměnnou state. Nakonec zmiňme proměnné pBlockSize a cBlockSize, jež po řadě obsahují aktuální délku bloku zprávy a aktuální délku bloku šifry.

K inicializaci objektu třídy RabinCipher je primárně určena metoda void engineInit (int opmode, Key key, SecureRandom random). Tato metoda nastaví proměnné daného objektu pro transformování dat v režimu opmode za použití klíče key a generátoru random. Dále na základě aktuální délky klíče vypočítá a nastaví hodnotu proměnných pBlockSize a cBlockSize. Objekt třídy RabinCipher je také možné inicializovat pomocí metod void engineInit (int opmode, Key key, AlgorithmParameterSpec params, SecureRandom random) a void engineInit (int opmode, Key key, AlgorithmParameters params, SecureRandom random), avšak protože naše implementace Rabinova kryptosystému nemá pro objekty params žádného využití, tyto metody pouze zavolají ternární metodu engineInit, které předají parametry opmode, key a random.

Důležitou metodou je metoda int engineGetOutputSize (int inputLen), která na základě délky vstupních dat, jež je dána hodnotou inputLen, a počtu hodnot obsažených v poli data vrací předpokládanou délku výstupu. Z důvodu větší jednoduchosti předpokládáme v režimu šifrování, že šifrovaná zpráva byla upravena algoritmem pro doplnění posledního bloku, šifra má pak délku cBlockSize((inputLen + dataLength + pBlockSize) / pBlockSize). V režimu dešifrování je situace jednodušší, délka šifry je vždy rovna násobku délky bloku a délku dešifrovaného textu, ve které je započítáno i doplnění posledního bloku, vypočteme vztahem pBlockSize((inputLen + dataLength) / cBlockSize).

Nyní přejdeme k metodám, jež zajišťují vlastní transformaci dat. Třída CipherSpi definuje dvojici metod engineUpdate, jež transformují pouze úplné bloky dat. Metoda byte [ ] engineUpdate (byte [ ] input, int inputOffset, int inputLen) zjistí předpokládanou délku výstupu, vytvoří pole out této délky a zavolá metodu int engineUpdate (byte [ ] input, int inputOffset, int inputLen, byte [ ] output, int outputOffset), jež do pole out zapíše výsledek transformace. Protože metody engineUpdate ani metody z nich volané žádným způsobem nepracují s algoritmem pro doplnění posledního bloku, v režimu šifrování je předpokládaná délka šifry vždy větší než její skutečná délka. Ternární metoda engineUpdate proto končí zkopírováním šifry do návratového pole, jehož kapacita je rovna skutečné délce šifry. Metoda engineUpdate arity pět nejprve vytvoří pole t a zkopíruje do něj jak obsah pole data, tak také obsah pole input. Dále vypočítá začátek a délku posledního neúplného bloku v poli t. Je-li délka pole t rovna násobku délky bloku, začátek neúplného posledního bloku je umístěn na pozici t.length – 1 a jeho délka je nastavena na nulu. Následně jsou všechny úplné bloky obsažené v poli t transformovány. Nakonec je neúplný poslední blok zkopírován do pole data.

Stejný vztah jako mezi metodami engineUpdate je také mezi dvojicí metod engineDoFinal, které transformují všechna dostupná data, čemuž v případě šifrování předchází aplikace algoritmu pro doplnění posledního bloku. V případě dešifrování metoda byte [ ] engineDoFinal(byte [ ] input, int inputOffset, int inputLen) vrátí pole upravené do podoby před použitím algoritmu pro doplnění posledního bloku. Metoda int engineDoFinal (byte [ ] input, int inputOffset, int inputLen, byte [ ] output, int outputOffset) nejprve vytvoří pole t a naplní jej stejným obsahem jako v případě metody engineUpdate. Poté v režimu šifrování vypočítá začátek a délku posledního neúplného bloku v t, zatímco v případě dešifrování zjistí začátek a délku posledního bloku. Posléze jsou transformovány všechny úplné bloky. Závěrem je v režimu šifrování patřičným způ-

sobem doplněn a transformován poslední blok, v režimu dešifrování je poslední blok zbaven následků použití algoritmu pro doplnění posledního bloku a zapsán do výstupního pole.

Poslední neúplný blok zprávy je šifrován až po doplnění na délku bloku, které je provedeno metodou int encryptBlockFinal (byte [ ] input, int inputOffset, int inputLen, byte [ ] output, int outputOffset). Návratovou hodnotou této metody je počet hodnot zapsaných do výstupního pole. Připomeňme, že pokud je délka zprávy násobkem délky bloku, je zpráva prodloužena o další blok, jehož všechny hodnoty se rovnají délce bloku. Typ byte zahrnuje celá čísla, která patří do intervalu [–128, 127], existuje tedy 256 hodnot typu byte, čímž je dána největší možná délka bloku. Protože číslo, jež potřebujeme při doplňování posledního bloku zapsat do pole hodnot typu byte, patří do množiny {1, … , 256}, musíme tuto množinu v metodě encryptBlockFinal vhodně zobrazit na množinu {–128, … , 127}:

```
protected int encryptBlockFinal (byte [ ] input, int inputOffset, int inputLen, byte [ ] output,
int outputOffset) {
   byte [ ] pom = new byte[pBlockSize];
   System.arraycopy(input, inputOffset, pom, 0, inputLen);
   int z = pBlockSize – inputLen;
   if (z > 127) {
         z = z – 256;
   }
   for (int i = inputLen; i < pom.length; i++) {
                        pom[i] = (byte) z;
   }
   int w =  encryptBlock(pom, 0, pBlockSize, output, outputOffset);
   return w;
}
```

Korektní dešifrování posledního bloku šifry zajišťuje metoda int decryptBlockFinal (byte [ ] input, int inputOffset, int inputLen, byte [ ] output, int outputOffset), která poslední blok dešifruje do zvláštního pole. Z tohoto pole jsou následně do výstupního pole output zkopírovány pouze ty hodnoty, jež do něj nebyly zapsány při použití algoritmu pro doplnění posledního bloku, a počet takto zkopírovaných hodnot je návratovou hodnotou metody.

```
protected int decryptBlockFinal (byte [ ] input, int inputOffset, int inputLen, byte [ ] output,
int outputOffset) {
   byte [ ] p = new byte[pBlockSize];
   int x = decryptBlock(input, inputOffset, inputLen, p, 0);
   x = p[p.length – 1];
   if (x < 0) {
        x = x + 256;
   }
   System.arraycopy(p, 0, output, outputOffset, pBlockSize – x);
   return pBlockSize – x;
}
```

Implementace Rabinova šifrovacího algoritmu je obsažena ve zdrojovém textu metody int encryptBlock (byte [ ] input, int inputOffset, int inputLen, byte [ ] output, int outputOffset). Nejprve do nově vytvořeného pole b zkopírujeme šifrovaný blok a zreplikujeme (pBlockSize + 1) / 2 posledních hodnot. Pak pole b převedeme na objekt třídy BigInteger a pomocí metod této třídy vypočítáme šifru. Získaný objekt nakonec převedeme na pole hodnot typu byte, jež zkopírujeme do výstupního pole. Metoda encryptBlock vrací počet hodnot zapsaných do výstupního pole (za

zapsání hodnoty považujeme i přeskočení pozice, která obsahuje hodnotu 0), jelikož šifruje pouze úplné bloky, návratová hodnota je vždy rovna délce bloku šifry.

```
protected int encryptBlock (byte [ ] input, int inputOffset, int inputLen, byte [ ] output, int outputOffset) {
   byte [ ] b = new byte[pBlockSize + (pBlockSize + 1) / 2];
   int v = (pBlockSize + 1) / 2;
   System.arraycopy(input, inputOffset, b, 0, pBlockSize);
   System.arraycopy(b, pBlockSize / 2, b, pBlockSize, v);
   BigInteger m = new BigInteger(1, b);
   RabinPublicKey rpk = (RabinPublicKey) key;
   BigInteger n = rpk.getN( );
   BigInteger two = BigInteger.valueOf(2);
   BigInteger c = m.modPow(two, n);
   byte [ ] cb = getBytes(c);
   System.arraycopy(cb, 0, output, outputOffset + cBlockSize – cb.length , cb.length);
   return cBlockSize;
}
```

Dešifrování probíhá podobně, je ovšem podstatně složitější než šifrování. Metoda int decryptBlock (byte [ ] input, int inputOffset, int inputLen, byte [ ] output, int outputOffset) vypočítá všechny čtyři odmocniny modulo n ze zpracovávaného bloku. Používaná redundance by měla být s vysokou pravděpodobností dostačující k jednoznačné identifikaci dešifrované zprávy, jestliže jednoznačná identifikace není možná, je na tuto skutečnost uživatel upozorněn výpisem na standardní výstup. Stejně jako předchozí metody také metoda decryptBlock vrací počet hodnot zapsaných do výstupního pole.

```
protected int decryptBlock (byte [ ] input, int inputOffset, int inputLen, byte [ ] output, int outputOffset) {
   RabinPrivateKey rpk = (RabinPrivateKey) key;
   BigInteger p = rpk.getP( );
   BigInteger q = rpk.getQ( );
   BigInteger n = rpk.getN( );
   if (p.compareTo(q) == 1) {
                  BigInteger h = p;
                  p = q;
                  q = h;
   }
   byte [ ] f = new byte [cBlockSize];
   System.arraycopy(input, inputOffset, f, 0, cBlockSize);
   BigInteger c = new BigInteger(1, f);
   BigInteger a = p.modInverse(q);
   BigInteger g = a.multiply(p);
   BigInteger one = BigInteger.valueOf(1);
   BigInteger b = g.subtract(one).divide(q);
   BigInteger four = BigInteger.valueOf(4);
   BigInteger ex = p.add(one).divide(four);
   BigInteger r = c.modPow(ex, p);
   BigInteger ex1 = q.add(one).divide(four);
   BigInteger s = c.modPow(ex1, q);
   BigInteger u = a.multiply(p).multiply(s);
```

```
        BigInteger v = b.multiply(q).multiply(r);
        BigInteger [ ] m = new BigInteger[4];
        m[0] = u.add(v).mod(n);
        m[1] = u.subtract(v).mod(n);
        m[2] = n.subtract(m[0]);
        m[3] = n.subtract(m[1]);
        byte [ ] mb = new byte[pBlockSize];
        boolean bl = false;
        boolean bol = false;
        byte [ ] ob = new byte[3*pBlockSize];
        for (int i = 0; i < 4; i++) {
                if (m[i].bitLength( ) < n.bitLength( )) {
                    byte [ ] pom = getBytes(m[i]);
                    if ((pom.length == pBlockSize + (pBlockSize + 1)/2) && isEqual(pom)) {
                                        if (!bl) {
                                            System.arraycopy(pom, 0, mb, 0, pBlockSize);
                                            bl = true;
                                        }
                                        else {
                                          System.arraycopy(pom, 0, ob, (i – 1)*pBlockSize, pBlock-
                                          Size);
                                          bol = true;
                                        }
                    }
                }
        }
        System.arraycopy(mb,0,output,outputOffset, pBlockSize);
        if (bol) {
            System.out.println("Desifrovani bloku cislo "+inputOffset/inputLen+" bylo nejednoznac-
            ne.");
            System.out.print("Dalsi varianty: ");
            Demo.print(ob);
            System.out.println();
        }
        return pBlockSize;
    }
```

Poznamenejme, že metoda static void print (byte [ ] p), jež náleží do třídy bc.crypto.Demo, provádí tisk pole hodnot typu byte na standardní výstup. Pomocná metoda static boolean isEqual (byte [ ] a) zjišťuje, zda-li dané pole obsahuje používanou redundantní informaci.

5.1.4 Třída RabinKeyFactory

Třída RabinKeyFactory provádí převody mezi jednotlivými druhy klíčů pro Rabinův kryptosystém. Tyto převody jsou relizovány následující trojicí metod:

- PrivateKey engineGeneratePrivate (KeySpec keySpec)
- PublicKey engineGeneratePublic(KeySpec keySpec)
- KeySpec engineGetKeySpec (Key key, Class keySpec)

Převody mezi objekty tříd RabinPublicKey a RabinPublicKeySpec a mezi objekty tříd RabinPrivateKey a RabinPrivateKeySpec jsou triviální, dále se tedy budeme zabývat pouze převody mezi instancemi tříd implementujících rozhraní Key a instancemi potomků třídy EncodedKeySpec. Máme-li převést objekt třídy EncodedRabinPrivateKeySpec na objekt třídy RabinPrivateKey, postupujeme následovně. Nejprve rozdělíme pole získané z objektu keySpec na dvě části délky l / 2 a (l + 1) / 2, kde l je délka daného pole. Dále každou z částí použijeme k vytvoření objektu třídy BigInteger a ze vzniklé dvojice objektů vytvoříme nový soukromý klíč. Zbývající převody jsou jednoduché. V metodě PublicKey engineGeneratePublic (KeySpec keySpec) převedeme pole obsažené v objektu třídy EncodedRabinPublicKeySpec na objekt třídy BigInteger, ze kterého následně dostaneme nový veřejný klíč. Metoda KeySpec engineGetKeySpec (Key key, Class keySpec) zavolá konstruktor třídy keySpec a předá mu pole vrácené metodou getEncoded klíče key.

5.2 Ong-Schnor-Shamirův podepisovací systém

5.2.1 Třídy pro reprezentaci klíčů

Veřejný a k němu náležející soukromý klíč pro Ong-Schnor-Shamirův podepisovací systém (OSS) jsou určeny přirozenými čísly n, h, k, kde n je liché přirozené číslo, které není prvočíslem, k je nesoudělné s n a $h = k^{-2} \bmod n$. Veřejným klíčem je dvojice (n, h), soukromým klíčem číslo k.

Třídy, jejichž objekty budou představovat klíče pro OSS, nazveme OSSPrivateKey a OSSPublicKey. Třída OSSPrivateKey implementuje rozhraní PrivateKey, třída OSSPublicKey rozhraní PublicKey. Stejně jako v případě Rabinova kryptosystému si usnadníme deklaraci uvedené dvojice tříd tím, že společné rysy obou tříd zachytíme v deklaraci jejich společného předka, abstraktní třídy OSSKey implementující rozhraní Key. Poněvadž při podepisování a také při extrakci do podpisu vložené zprávy potřebujeme znát číslo n, budeme jej uchovávat v objektech třídy OSSKey. Třídu OSSPublicKey potom deklarujeme jako schránku pro číslo h a podobně třídu OSSPrivateKey jako schránku pro číslo k.

Nejzajímavější metodou rozhraní Key je metoda getEncoded, proto se dále stručně zmíníme o implementaci této metody v jednotlivých třídách reprezentujících klíče. Metoda getEncoded ve třídě OSSKey je deklarována jako abstraktní, neboť ji na této úrovni není možné deklarovat tak, aby ji mohl zdědit některý z potomků třídy OSSKey. V případě třídy OSSPrivateKey uvažovaná metoda vrací pole, které dostaneme aplikací metody getBytes, již známe z části věnované Rabinovu kryptosystému, na čísla n, k a následným zkopírováním obsahu získaných polí do jediného pole c kapacity 2a.length, kde a je výsledek volání metody getBytes s argumentem n. Kopie pole a se v poli c nachází mezi pozicemi 0 a a.length – 1. Pole, jež reprezentuje číslo k, označíme b a jeho kopii umístíme mezi pozice c.length – b.length a c.length – 1. Z tímto způsobem naplněného pole jsme schopni hodnoty n, k, a tedy daný soukromý klíč, korektně získat. Naprosto stejně deklarujeme metodu getEncoded ve třídě OSSPublicKey, pouze místo čísla k tato metoda pracuje s číslem h.

Dále deklarujeme třídy OSSPrivateKeySpec, OSSPublicKeySpec, EncodedOSSPrivateKeySpec a EncodedOSSPublicKeySpec. Tyto třídy jsou velmi podobné odpovídajícím třídám pro Rabinův kryptosystém, proto se jimi nebudeme podrobněji zabývat.

5.2.2 Třída OSSKeyPairGenerator

Třídu OSSKeyPairGenerator deklarujeme jako potomka třídy KeyPairGeneratorSpi, musíme tedy překrýt abstraktní metody initialize a generateKeyPair. Metoda initialize stejně jako v případě třídy RabinKeyPairGenerator nastaví požadovanou délku klíče (značíme ji ks) a generátor pseudonáhodných čísel (sr), jenž bude při generování použit. Délkou klíče rozumíme bitovou délku čísla n.

Nyní popíšeme činnost metody generateKeyPair. Nejprve si náhodně zvolíme liché přirozené n, které není prvočíslem, požadované bitové délky. Třída BigInteger nabízí několik možností pro vygenerování náhodného prvočísla o dané bitové délce, pro složená čísla zde ovšem taková možnost chybí. Nejblíže našim požadavkům je konstruktor BigInteger (int numBits, Random rnd), jenž vytvoří náhodné celé číslo patřící do intervalu $[0, 2^{numBits} - 1]$. Tento konstruktor s parametry ks a sr budeme opakovaně volat tak dlouho, dokud neobdržíme číslo kýžených vlastností. Dále pomocí téhož konstruktoru vygenerujeme přirozené číslo k nesoudělné s n. Nakonec vypočítáme číslo k^{-1} mod n, umocníme jej modulo n, čímž dostaneme číslo h, a vytvoříme nové instance tříd OSSPublicKey a OSSPrivateKey.

5.2.3 Třída OSSSignature

OSS je běžně zařazován mezi podepisovací systémy, nicméně můžeme jej také považovat za asymetrický kryptosystém zvláštního typu, v němž je k šifrování i dešifrování používán soukromý klíč. Tuto dvojí povahu OSS je sice možné zachytit překrytím abstraktních metod třídy SignatureSpi, nicméně z důvodů, které vysvětlíme posléze, přidáme do zdrojového textu třídy OSSSignature metodu extractMessage sloužící k získání zprávy přenášené v rámci digitálního podpisu. Důsledkem je, že pokud zahrneme třídu OSSSignature v prezentované podobě do nějakého provideru, můžeme k ní sice přistupovat pomocí objektů třídy Signature, ale přístup tímto způsobem je omezen pouze na funkcionalitu digitálního podepisování. V případě, že chceme prostřednictvím objektů třídy OSSSignature přenášet tajné zprávy, je buď třeba, abychom k této třídě přistupovali přímo, nebo abychom jednoduchým způsobem upravili její zdrojový kód.

Způsob, jakým jsou deklarovány abstraktní metody engineUpdate třídy SignatureSpi, předpokládá, že v jejích potomcích budou tyto metody napsány tak, aby umožňovaly postupné doplňování dat, s nimiž má daný objekt pracovat. Data, jejichž podpis má být generován nebo ověřován, bychom v rámci objektů třídy OSSSignature mohli ukládat pomocí pole hodnot typu byte, problém je ovšem v tom, že existující pole v Javě nemůže měnit svou kapacitu. Museli bychom tedy buď před vlastním ukládáním dat vytvořit pole takové kapacity, o níž bychom věděli, že ji nepřekročíme, nebo při doplňování dat občas kopírovat aktuální data do nově vytvořeného pole s dostatečnou kapacitou. Java naštěstí nabízí elegantnější řešení tohoto problému. K ukládání dat použijeme třídu java.io.ByteArrayOutputStream, jejíž objekty představují výstupní tok, ve kterém jsou data zapisována do pole hodnot typu byte takového, že jeho kapacita dynamicky roste s počtem zapsaných dat. Tato třída poskytuje metodu byte [ ] toByteArray ( ), která vrací v daném objektu zapsané hodnoty. Podobným způsobem budeme také ukládat utajovanou zprávu v objektech třídy OSSMessage.

Přehled metod třídy OSSSignature začneme inicializačními metodami void engineInitSign (PrivateKey key) a void engineInitVerify (PublicKey key), které připraví daný objekt buď pro podepisování, nebo pro ověřování podpisu. Pro předání dat objektu třídy OSSSignature slouží metody engineUpdate (byte b) a engineUpdate (byte [ ] b, int off, int len). Pokud bychom při podepisování pomocí těchto metod předávali podepisovaná data i přenášenou zprávu, jen obtížně bychom předávané identifikovali. Metody engineUpdate proto budeme používat výhradně pro předávání podepisovaných dat nebo dat, jejichž podpis hodláme ověřovat, a přenášenou zprávu

poskytneme objektu třídy OSSSignature jiným způsobem. Ze zbývajících metod třídy SignatureSpi je pro předávání dat patrně nejvhodnější metoda void engineSetParameter (AlgorithmParameterSpec params). Tato metoda není deklarována jako abstraktní, což znamená, že lze vytvořit potomka třídy SignatureSpi, který není abstraktní, i bez překrytí této metody. Napíšeme tedy zdrojový text jednoduché třídy OSSMessage, která implementuje rozhraní AlgorithmParameterSpec, je potomkem abstraktní třídy java.security.AlgorithmParametersSpi a jejíž jedinou proměnnou je objekt třídy ByteArrayOutputStream. S přenášenou zprávu potom budeme pracovat následujícím způsobem. Budeme ji uchovávat v rámci objektu třídy OSSMessage a zavoláním metody engineSetParameter předáme tento objekt instanci třídy OSSSignature, až když bude zpráva kompletní.

Podepisování je realizováno metodou byte [ ] engineSign ( ). Tato metoda nejprve převede podepisovaná data a přenášenou zprávu na objekty třídy BigInteger. Následně ověří, je-li číslo získané převodem přenášené zprávy menší než modulus n (v opačném případě by nebylo možné zprávu z podpisu získat) a jsou-li obě čísla nesoudělná s n. Jestliže některá z uvedených podmínek není splněna, běh metody je ukončen vyvoláním výjimky. Dále metoda standardně pokračuje vypočtením digitálního podpisu za použití metod třídy BigInteger. Obě vypočtená čísla jsou převedena na pole hodnot typu byte a obě pole jsou zkopírována do výstupního pole délky 2l, kde l = (k + 7) / 8, k je bitová délka modulu n.

```
public byte [ ] engineSign( ) throws SignatureException {
   byte [ ] e = data.toByteArray( );
   data = new ByteArrayOutputStream ( );
   BigInteger w1 = new BigInteger(1, e);
   BigInteger w = new BigInteger(1, secret);
   secret = null;
   BigInteger n = ((OSSPrivateKey)key).getN();
   BigInteger k = ((OSSPrivateKey)key).getK();
   if  (!(w.compareTo(n) == p – 1)) {
            throw new SignatureException("Prenasena zprava neni mensi nez modulus n.");
   }
   BigInteger wInv = BigInteger.valueOf(0);
   w1 = w1.mod(n);
   try {
      wInv = w.modInverse(n);
   }
   catch(ArithmeticException ae) {
      throw new SignatureException(" w a n jsou soudelna.");
   }
   BigInteger one = BigInteger.valueOf(1);
   if (w1.gcd(n).compareTo(one) != 0) {
      throw new SignatureException(" w1 a n jsou soudelna.");
   }
   BigInteger two = BigInteger.valueOf(2);
   BigInteger twoInv = two.modInverse(n);
   BigInteger z = w1.multiply(wInv);
   BigInteger s1 = z.add(w).multiply(twoInv).mod(n);
   BigInteger s2 = z.subtract(w).multiply(k).multiply(twoInv).mod(n);
   int l = (n.bitLength( ) + 7) / 8;
   byte [ ] s = new byte[2 * l];
   byte [ ] b1 = getBytes(s1);
   byte [ ] b2 = getBytes(s2);
```

```
    System.arraycopy(b1, 0, s, l – b1.length, b1.length);
    System.arraycopy(b2, 0, s, 2 * l – b2.length, b2.length);
    return s;
}
```

K ověření podpisu je určena metoda boolean engineVerify (byte [ ] sigBytes), která na základě pole sigBytes zrekonstruuje daný podpis, dále převede příslušná data na objekt třídy BigInteger a konečně jednoduchým výpočtem podpis ověří. Metoda vrátí hodnotu true, právě když byl podpis úspěšně ověřen, v opačném případě vrátí hodnotu false.

```
public boolean engineVerify(byte [ ] sigBytes) throws SignatureException {
    BigInteger n = ((OSSPublicKey)key).getN( );
    BigInteger h = ((OSSPublicKey)key).getH( );
    int l = (n.bitLength( ) + 7) / 8;
    byte [ ] b1 = new byte[l];
    byte [ ] b2 = new byte[l];
    System.arraycopy(sigBytes, 0, b1, 0, l);
    System.arraycopy(sigBytes, l, b2, 0, l);
    BigInteger s1 = new BigInteger(1, b1);
    BigInteger s2 = new BigInteger(1, b2);
    BigInteger w1 = new BigInteger(1, data.toByteArray( ));
    data = new ByteArrayOutputStream ( );
    BigInteger wm = w1.mod(n);
    BigInteger two = BigInteger.valueOf(2);
    BigInteger t1 = s1.modPow(two, n);
    BigInteger t2 = s2.modPow(two, n).multiply(h);
    BigInteger w2 = t1.subtract(t2).mod(n);
    return wm.equals(w2);
}
```

Nakonec se budeme zabývat tím, jak z daného podpisu získat přenášenou zprávu. K tomuto účelu by se nabízelo vhodným způsobem překrýt metodu AlgorithmParameters engineGetParameters ( ), ale protože třída java.security.AlgorithmParameters patří mezi engine classes, její instanci je možné vytvořit pouze pro takový algoritmus, který se vyskytuje v některém z providerů. Abychom se vyhnuli potížím způsobeným neochotou Javy pracovat s uživatelem vytvořenými providery, k získání přenášené zprávy bude sloužit metoda byte [ ] extractMessage (byte [ ] sigBytes, PrivateKey key). Stejně jako metoda engineVerify i metoda extractMessage nejprve pole sigBytes převede na dvojici objektů třídy BigInteger, jež představují daný podpis. Poté pomocí soukromého klíče key vypočítá přenášenou zprávu a vrátí ji jako pole hodnot typu byte.

```
public  byte [ ]  extractMessage (byte [ ] sigBytes, PrivateKey key)  throws  InvalidKeyException {
    if (!(key instanceof OSSPrivateKey)) {
                throw new InvalidKeyException("Nezadan objekt tridy OSSPrivateKey");
    }
    BigInteger n = ((OSSPrivateKey)key).getN();
    BigInteger k = ((OSSPrivateKey)key).getK();
    int l = (n.bitLength( ) + 7) / 8;
    byte [ ] b1 = new byte[l];
    byte [ ] b2 = new byte[l];
    System.arraycopy(sigBytes, 0, b1, 0, l);
```

```
        System.arraycopy(sigBytes, l, b2, 0, l);
        BigInteger s1 = new BigInteger(1, b1);
        BigInteger s2 = new BigInteger(1, b2);
        BigInteger w1 = new BigInteger(1,data.toByteArray( ));
        data = new ByteArrayOutputStream ( );
        BigInteger kInv = k.modInverse(n);
        BigInteger j = kInv.multiply(s2).add(s1).mod(n);
        BigInteger jInv = j.modInverse(n);
        BigInteger w = w1.multiply(jInv).mod(n);
        byte [ ] m = getBytes(w);
        return m;
    }
```

5.2.4 Třída OSSKeyFactory

Tato třída je potomkem třídy KeyFactorySpi a pro převody mezi objekty tříd implementujících rozhraní Key a objekty tříd, jež implementují rozhraní KeySpec, poskytuje stejnou trojici metod jako třída RabinKeyFactory. Funkcionalitu těchto metod popíšeme opět pouze pro případ převodů mezi objekty tříd implementujících rozhraní Key a objekty potomků třídy EncodedKeySpec. Metoda PrivateKey engineGeneratePrivate (KeySpec keySpec) rozdělí pole hodnot typu byte, jež je proměnnou objektu keySpec, na dvě části stejné délky a z nich vytvoří objekty třídy BigInteger, jež použije jako parametry při volání konstruktoru třídy OSSPrivateKey. Stejným postupem vytvoří metoda PublicKey engineGeneratePublic (KeySpec keySpec) na základě objektu keySpec nový veřejný klíč pro OSS. Převody opačným směrem provádí metoda KeySpec engineGetKeySpec (Key key, Class keySpec). Tato metoda zavolá metodu getEncoded objektu key a pole, které takto získá, předá jako parametr konstruktoru třídy keySpec.

6. Efektivita kryptografických operací

Kryptografické algoritmy lze implementovat v mnoha programovacích jazycích, nabízí se tedy otázka, jak ve srovnání s ostatními jazyky obstojí Java. Situaci si zjednodušíme tím, že budeme uvažovat pouze jazyk C, který je společně s příbuznými jazyky C++ a C# v současnosti pravděpodobně nejrozšířenější.

Protože kryptografické algoritmy často pracují s velkými objemy dat, velmi důležitým srovnávacím parametrem je rychlost. Při porovnávání rychlosti kryptografických operací v Javě a v C se budeme soustředit na symetrické a asymetrické kryptosystémy. V obou jazycích napíšeme program šifrující stejná data za použití kryptosystému DES a režimu CBC. Oba programy dvacetkrát spustíme, změříme dobu trvání šifrování a vypočítáme aritmetický průměr z naměřených hodnot. Posléze totéž provedeme pro kryptosystém RSA Následují zdrojové texty jednotlivých programů, nejprve jsou uvedeny programy šifrující pomocí DES a dále programy šifrující pomocí RSA. Oba programy v C, byly vytvořeny za použití kryptografické knihovny cryptlib.

```
package bc.crypto;
import javax.crypto.*;
import java.security.*;

public class Test {
    public static  void main (String [ ] args)  throws  NoSuchAlgorithmException,  InvalidKey-
    Exception, IllegalBlockSizeException,  NoSuchPaddingException,  BadPaddingException,
    InvalidAlgorithmParameterException {
        byte [ ] p = {26, 72, 1, 52, 123, 24, 44, 68};
        long begin, end;

        byte [ ] pt = new byte[64];
        for (int i = 0; i < 8;i++) {
                System.arraycopy(p, 0, pt, i*p.length, p.length);
        }
        begin = System.currentTimeMillis( );
        Cipher c = Cipher.getInstance("DES/CBC/PKCS5Padding");
        KeyGenerator kg = KeyGenerator.getInstance("DES");
        SecretKey desKey = kg.generateKey( );
        c.init(Cipher.ENCRYPT_MODE, desKey);
        byte [ ] ct = c.doFinal(pt);
        AlgorithmParameters ap = c.getParameters( );
        end = System.currentTimeMillis( );
        long time = end – begin;
        System.out.println("Sifrovani trvalo "+time+" ms.");
    }
}

/*Test.c*/
#include "C:\CRYPTLIB\cryptlib.h"
#include "string.h"
#include "time.h"
#include "stdio.h"
```

```
   void main( ) {
      CRYPT_CONTEXT cryptContext;
      static unsigned char iv[CRYPT_MAX_IVSIZE];
      static int ivSize,i;
      clock_t begin, end;
      static char p[9]= {26, 72, 1, 52, 123, 24, 44, 68, 0};
      static char t[64];
      for(i = 0; i < 8;i++) {
            strcat(t, p);
      }
      begin = clock( );
      cryptInit( );
      cryptCreateContext(&cryptContext, CRYPT_UNUSED, CRYPT_ALGO_DES);
      cryptGenerateKey(cryptContext);
      cryptEncrypt(cryptContext,t,64);
      cryptGetAttributeString(cryptContext,CRYPT_CTXINFO_IV, iv, &ivSize);
      end = clock( );
      printf("Sifrovani trvalo %f ms.",(float)(end – begin)/CLOCKS_PER_SEC*1000);
      getch( );
      cryptDestroyContext(cryptContext);
      cryptEnd( );
   }
```

```
package bc.crypto;
import javax.crypto.*;
import java.security.*;

public class Test1 {
    public static  void  main (String [ ] args) throws  NoSuchAlgorithmException,  InvalidKey-
    Exception, IllegalBlockSizeException,  NoSuchPaddingException,  BadPaddingException,
    InvalidAlgorithmParameterException {
        byte [ ] p = {26, 72, 1, 52, 123, 24, 44, 68};
        long begin, end;

        byte [ ] pt = new byte[64];
        for (int i = 0; i < 8;i++) {
                System.arraycopy(p, 0, pt, i*p.length, p.length);
        }
        begin = System.currentTimeMillis( );
        Cipher c = Cipher.getInstance("RSA");
        KeyPairGenerator kg = KeyPairGenerator.getInstance("RSA");
        kg.initialize(594);
        KeyPair kp = kg.generateKeyPair( );
        PublicKey rsaKey = kp.getPublic( );
        c.init(Cipher.ENCRYPT_MODE, rsaKey);
        byte [ ] ct = c.doFinal(pt);
        end = System.currentTimeMillis( );
        long time = end – begin;
        System.out.println("Sifrovani trvalo "+time+" ms.");
    }
}
```

```
 /*Test1.c*/
#include "C:\CRYPTLIB\cryptlib.h"
#include "string.h"
#include "time.h"
#include "stdio.h"
   void main( ) {
      CRYPT_CONTEXT cryptContext;
      CRYPT_PKCINFO_RSA rsaKey;
      int i;
      clock_t begin, end;
      static char p[9]= {26, 72, 1, 52, 123, 24, 44, 68, 0};
      static char t[64];
      for(i = 0; i < 8; i++) {
         strcat(t, p);
      }
      begin = clock( );
      cryptInit( );
      cryptCreateContext(&cryptContext, CRYPT_UNUSED, CRYPT_ALGO_RSA);
      cryptSetAttributeString(cryptContext, CRYPT_CTXINFO_LABEL, p, 8);
      cryptSetAttribute(cryptContext, CRYPT_CTXINFO_KEYSIZE, 64);
      cryptGenerateKey(cryptContext);
      cryptEncrypt(cryptContext, t, 64);
      end = clock( );
      printf("Sifrovani trvalo %f ms.",(float)(end – begin)/CLOCKS_PER_SEC*1000);
      getch( );
      cryptDestroyContext(cryptContext);
      cryptEnd( );
}
```

Měření rychlosti všech programů probíhalo na počítači s procesorem Intel Pentium III 600E a operační pamětí o kapacitě 64 MB. Javový program šifrující kryptosystémem DES byl spouštěn ve vývojovém prostředí NetBeans 3.5.1, program v tomtéž jazyce šifrující pomocí RSA byl spouštěn ve vývojovém prostředí NetBeans 3.6 Beta. K měření rychlosti obou programů v jazyce C bylo použito vývojové prostředí Borland C++Builder 6. Průměrná doba šifrování v Javě činila 2368 ms pro DES a 5367 ms pro RSA, zatímco program v C byl s šifrováním hotov v průměru za 515 ms v případě DES a za 551 ms v případě RSA. Přestože tento jednoduchý pokus zdaleka nebyl úplným srovnáním rychlosti kryptografických operací v Javě a v C, je vidět, že Java za C v rychlosti zaostává. Pro menší objemy dat je rozdíl z hlediska uživatele zanedbatelný, ovšem při práci s rozsáhlejšími daty bude pravděbodně znatelný.

7. Závěr

Kryptografie v Javě je založena na principu engine classes, zvláštních tříd, které umožňují pracovat stejným způsobem se všemi třídami, jež poskytují určitou kryptografickou operaci, nezávisle na konkrétním algoritmu či na jeho implementaci. Aby bylo možné k dané třídě přistupovat pomocí engine class A, musí být tato třída deklarována jako potomek abstraktní třídy B, jež se nachází ve stejném balíku jako třída A a jejíž název se od názvu třídy A liší pouze příponou Spi. Zdárně vytvořená instance třídy A pro nějaký algoritmus obsahuje jako soukromou proměnnou instanci třídy, která implementuje tento algoritmus, a její metody volají v této třídě překryté metody třídy B. Zdrojem informací o dostupných třídách implementujících nějaký kryptografický algoritmus jsou podtřídy třídy java.security.Provider. Daného potomka této třídy spolu s množinou tříd v něm uvedených souhrnně označujeme pojmem provider.

Výše popsaný princip z hlediska běžného uživatele velmi dobře funguje při práci s kryptografickými třídami, které jsou buď součástí standardní distribuce Javy, nebo některého z providerů dostupných na webu. V tomto případě je předností Javy při realizaci kryptografických operací snadná manipulace s objekty tříd obsažených v Java Core API, což je do značné míry umožněno zevrubnou dokumentací dostupnou ze stránek http://java.sun.com. Důsledkem je, že uživatel se vůbec nemusí zajímat o podrobnosti týkající se jednotlivých algoritmů a jejich implementace, je například možné napsat program, který bude šifrovat data pomocí symetrického kryptosystému zadaného jako argument na příkazovém řádku. Problém ovšem nastává v okamžiku, kdy se rozhodneme používat vlastní provider. Samotné napsání podtřídy třídy Provider je jednoduché, poněvadž spočívá pouze v přiřazení typu a standardního názvu algoritmu jednotlivým kryptografickým třídám. Komplikací při tvorbě vlastních providerů je neochota Javy pracovat s nedůvěryhodnými providery. Právě tato skutečnost a také nepříliš vysoká rychlost kryptografických operací jsou asi největší nevýhody Javy při jejím použití pro kryptografické účely.

V této práci byla popsána implementace Rabinova kryptosystému a Ong-Schnor-Shamirova podepisovacího systému v Javě. Výsledkem práce jsou třídy určené pro použití přímým přístupem uživatele. Pro přístup pomocí engine classes je možné použít třídy implementující Rabinův kryptosystém beze změny, v případě Ong-Schnor-Shamirova podepisovacího systému je třeba změnit hlavičku metody byte [ ] extractMessage (byte [ ] sigBytes, PrivateKey key) ve třídě OSSSignature na AlgorithmParameters getParameters ( ) a odpovídajícím způsobem upravit její zdrojový kód. Prezentovaná implementace Rabinova kryptosystému vytvoří k dané zprávě šifru, jejíž délka je přibližně o polovinu delší než délka zprávy. Další vývoj této implementace by tedy mohl být veden snahou o zmenšení délky šifry, aniž by došlo k podstatnému zvýšení neurčitosti při jejím dešifrování. Aby obě popsané implementace mohly spolupracovat s jinými implementacemi téhož algoritmu, bylo by dále vhodné upravit definici metody byte [ ] getEncoded ( ) ve všech třídách implementujících rozhraní Key tak, aby tato metoda ve všech případech vracela klíč ve standardním formátu (např. X.509 pro veřejné klíče a PKCS#8 pro soukromé klíče).

Bakalářská práce svým rozsahem v žádném případě neumožňuje vyčerpávající popis kryptografických tříd a rozhraní, které se nacházejí v Java Core API. Pro podrobnější informace (například o třídách poskytujících funkcionalitu pro správu klíčů) odkazuji čtenáře na stránky http://java.sun.com. Stejně tak jsem se v části věnované kryptografii byl nucen z prostorových důvodů omezit pouze na základní informace o této zajímavé disciplíně. Tato práce se snažila ukázat, jak mnoho možností Java nabízí jak z hlediska používání hotových kryptografických tříd, tak také z hlediska implementace kryptografických algoritmů. Závěrem mohu jen vyjádřit přání, že se jí o tom podaří přesvědčit co nejvíce z jejích čtenářů.

Zdroje a použitá literatura

[1] Dokumentace ke standardní distribuci Javy 2, v 1.4.2
http://java.sun.com/j2se/1.4.2/docs/api

[2] Dokumentace ke standardní distribuci Javy 2, v 1.5.0 Beta
http://java.sun.com/j2se/1.5.0/docs/api

[3] Pitner T.: Java začínáme programovat, Grada 2002

[4] Materiály k předmětu PB162 Programování v jazyce Java, podzim 2002

[5] Kotala Z., Toman P.: Java sborník
http://dione.zcu.cz/java/sbornik.html

[6] Menezes A. J., van Oorschot P. C., Vanstone S. C.: Handbook of Applied Cryptography, CRC Press 2001
http://www.cacr.math.uwaterloo.ca/hac

[7] Učební text k předmětu MA024 Kryptografie
ftp://www.math.muni.cz/pub/math/people/Paseka/lectures/kryptografie.ps

[8] Materiály k předmětu IV054 Kódování, kryptografie a kryptografické protokoly, podzim 2002
http://fi.muni.cz/usr/gruska/crypto02

[9] Java Cryptography Architecture API Specification & Reference
http://java.sun.com/j2se/1.5.0/docs/guide/security/CryptoSpec.html

[10] Java Cryptography Extension (JCE) Reference Guide
http://java.sun.com/j2se/1.5.0/docs/guide/security/jce/JCERefGuide.html

[11] Knudsen J. B.: Java Cryptography, O'Reilly, 1998
http://www.coe.unic.edu/~vpitale/JavaCryptography.pdf

[12] Cryptography in Java
http://medialab.di.unipi.it/doc/JnetSec/jns_ch8.htm

[13] Why is developing JCE Providers so hard ?
http://www.pankaj-k.net/weblog-archive/2004_03.html

[14] cryptlib Security Toolkit
ftp://ftp.franken.de/pub/crypt/cryptlib/manual.pdf

Seznam příloh na CD

1. Balík bc.crypto.rabin
2. Balík bc.crypto.oss
3. Dokumentace k uvedeným balíkům