MAGAZÍN
KATEDRY INFORMATIKY

číslo 3 | červenec 2015 Univerzita Palackého v Olomouci

Úvodní slovo

Vážení čtenáři,

rádi bychom se s Vámi i v tuto prázdninovou dobu opět podělili o informace z Katedry informatiky a ze světa informatiky vůbec. Od posledního vydání našeho magazínu si naše katedra připsala na svůj účet dva významné úspěchy.

Článek členů naší katedry Radima Bělohlávka, Jana Outraty a Viléma Vychodila byl Radou pro výzkum, vývoj a inovace vybrán mezi 39 nejlepších publikací, které v České republice za posledních pět let vznikly v oboru Technické a informatické vědy.

Další úspěch si připsal sám profesor Bělohlávek, když se začátkem tohoto roku stal šéfredaktorem prestižního vědeckého časopisu International Journal of General Systems. Tento časopis je zaměřen na obecné aspekty různých oblastí informatiky a systémových věd jako jsou například neurčitost a složitost.

V tomto již třetím vydání našeho magazínu naleznete články o výuce a výzkumu programovacích jazycích na naší katedře. Seznámíte se také s Enigmou, šifrovacím zařízením používaným během 2. světové války. Chybět nebude ani hádanka, tentokrát programátorského rázu.

Za tým, který toto vydaní magazínu připravil, Vám příjemné čtení a prožití prázdninových měsíců přeje

Petr Krajča

redaktor Magazínu Katedry informatiky Univerzity Palackého v Olomouci

Obsah

Programovací jazyky

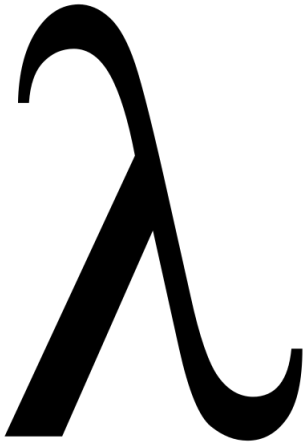
Petr Krajča

Informatika je sice mnohem víc než pouhé programování, ale programování a programovací jazyky k ní naprosto neodmyslitelně patří. V tomto článku bychom Vám rádi představili, jaké programovací jazyky na naší katedře vyučujeme.

Charakteristickým rysem výuky programovaní na naší katedře je, že se základními technikami programování jsou studenti seznamováni pomocí jazyků vycházejících z LISPu, konkrétně se jedná o jazyk Scheme, se kterým se setkávají studenti v prvních dvou semestrech v rámci předmětu Paradigmata programovaní. Při výuce objektově orientovaného programovaní se pak studenti setkají s dalším lispovým jazykem, tentokrát CommonLispem. Výhodou je, že oba jazyky mají jednoduchou syntaxi i sémantiku, se kterou se není nutné dlouze seznamovat, a lze se soustředit přímo na řešení konkrétních problémů, což je to hlavní nejen při výuce informatiky. Jen pro srovnání, jak náročné je pojmout celý jazyk se vším všudy: kompletní specifikace jazyka Scheme má asi 90 stran, jazyk Java je popsán na cca 800 stranách a specifikace C++ zabírá asi 1350 stran.

Jazyky vycházející z LISPu mají vedle své jednoduchosti i další praktické vlastnosti. I přesto, že vycházejí z funkcionálního paradigmatu, dá se v nich programovat imperativně a objektově. Není tedy nutné se učit více různých programovacích jazyků.

Dalším příjemným rysem je skutečnost, že v základu obsahují řadu prostředků a konstruktů, které nejsou v jiných „praktických" jazycích implementovány, nebo jsou implementovány s kompromisy – makra, líné vyhodnocování, streamy, vícenásobnou dědičnost, aktuální pokračovaní, atd. Mohli bychom se pousmát nad tím, že zatímco lambda výrazy a streamy jsou v Javě 8 žhavou novinkou, tak Scheme je má již několik desetiletí. Ale spíše než pošklebek to vnímejme jako hezký doklad toho, že Scheme/Lisp je z tohoto pohledu nadčasový jazyk a dá-



vá možnost seznámit se s různými programátorskými technikami předtím, než je autoři „praktických" programovacích jazyků zahrnou do svých specifikací.

Zkušenosti nabyté z jazyků jako jsou Scheme nebo CommonLisp lze přenést bez problémů do ostatních jazyků. Nejsou to proto jediné programovací jazyky, se kterými se naši studenti ve výuce setkávají. Úvodní kurzy programování jsou zaměřeny na jazyk C a v dalších semestrech si mohou studenti zvolit kurzy zaměřené na moderní programovací jazyky jako jsou C++, C#, či Java a profilovat tak své programátorské dovednosti. Do výuky je zařazen i assembler, který umožňuje pochopit fungovaní počítače na té nejnižší úrovni. Závěrem dodejme, že studenti nejsou nijak omezováni při vypracovávání závěrečných prací a úkolů, mají volnou ruku, co se týče volby programovacího jazyka, a mohou používat takový jazyk, který jim vyhovuje nejvíce.

Schemik

Petr Krajča

Na naší katedře probíhá výzkum programovacích jazyků. Potom, jak byly v úvodním článku vychváleny programovací jazyky vycházející z LISPu, nebude příliš překvapující, že při výzkumu používáme právě tento typ jazyků. Má to své nesporné výhody, zejména díky jednoduché syntaxi a sémantice není komplikované přidávat nové vlastnosti a experimentovat s nimi. Zaměřujeme se hlavně na to, jak usnadnit programátorům vývoj aplikací, které využívají více vláken, respektive procesorů.

Vícejádrové procesory jsou dnes standardem a paralelní programování je nedílnou součástí výuky informatiky již řadu let, přesto tvorba programů běžících ve více vláknech dokáže potrápit nejednoho zkušeného programátora, hlavně proto, že programovací jazyky neposkytují adekvátní prostředky, kterými by se dal paralelismus v programu vyjádřit. Ideální jazyk by z tohoto pohledu byl takový, který umožní bezpečně rozložit zátěž mezi více procesorů bez zásahu programátora. Proto v rámci výzkumu u nás vznikl minimalistický interpreter jazyka Scheme nazývaný Schemik [čti: Šemík], který právě toto umožňuje, tj. programátor napíše program ve Schemu a běhové prostředí se samo postará o rozdělení výpočtu mezi více vláken, potažmo procesorů. Omezený prostor našeho magazínu bohužel neumožňuje rozebrat, jak to celé funguje, ale rádi bychom Vám stručně nastínili, s jakými obecnými problémy se při návrhu Schemika setkáváme.

Kde jsou limity paralelizace?

Bylo by sice hezké napsat program a nechat jej, ať se sám přizpůsobí počtu procesorů, které má k dispozici, ale ne vždy je to proveditelné. Vezměte si například třídící algoritmus InsertSort. Není možné zařadit prvek do setříděné posloupnosti předtím, než je do ní zařazen prvek předchozí. Takový algoritmus opravdu více procesorů nevyužije, protože se jedná o sekvenční algoritmus. Na druhou stranu algoritmy pracující s velkými poli nebo algoritmy z rodiny „rozděl a panuj“ (např. QuickSort) jsou pro automatickou paralelizaci jak dělané.

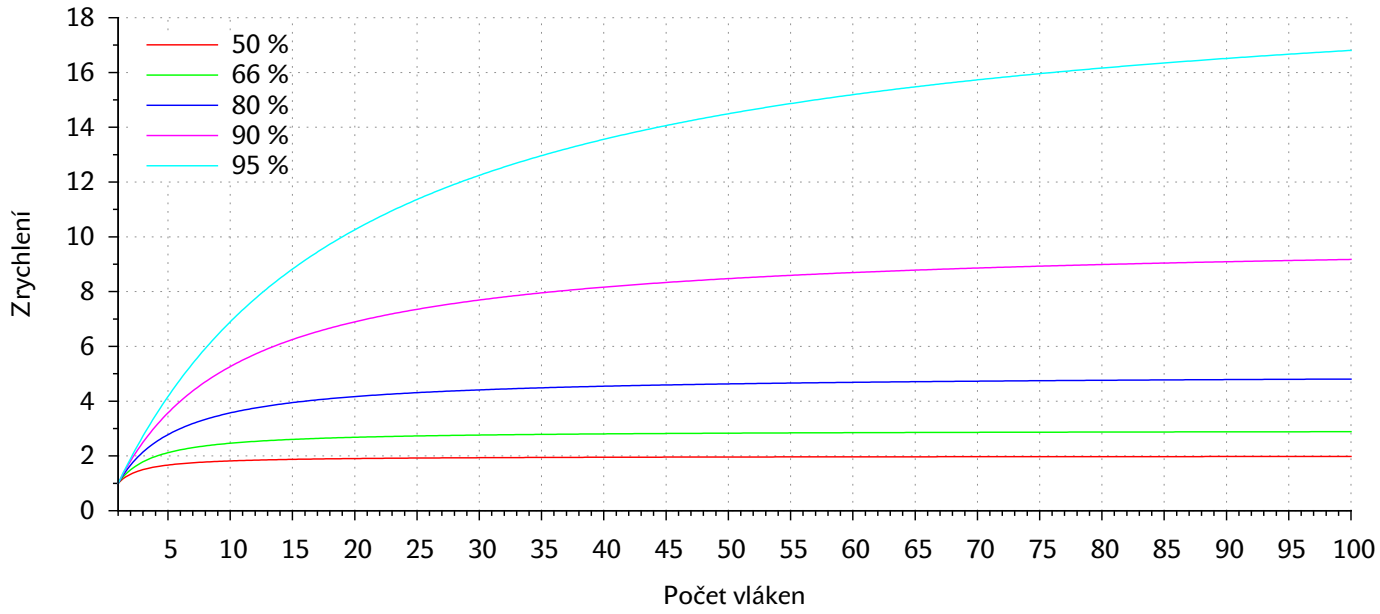
Kolik jader je potřeba?

Předchozí zjištění má ještě jeden zajímavý a poněkud nepříjemný důsledek – v téměř každém programu se najde část kódu, kterou nelze paralelizovat, např. inicilizaci. V roce 1967 formuloval Gene Amdahl (architekt legendárního mainframu IBM System/360), tzv. Amdahlův zákon, který udává, jak zrychlení části programu ovlivní výslednou rychlost programu. Předpokládejme, že část programu může běžet pouze sekvenčně, označme si to P, a zbývající část jde urychlit použitím N procesorů. Celkové dosažitelné zrychlení S se pak dá vyjádřit pomocí vzorce:

$$S(N) = \frac{1}{(1-P) + \frac{P}{N}}.$$

Jaké to má důsledky, můžete vidět v grafu na další straně.

Pokud program lze paralelizovat jen z poloviny, lepší než dvojnásobné zrychlení nedostaneme. Naštěstí obecně situace není tak zoufalá, jak by se mohlo zdát. To, jak je velké P ovlivňuje velikost zpracovávaných dat, takže pro velká data paralelizace smysl má. Navíc Amdahlův zákon předpokládá, že paralelní program provádí stejné kroky, jako by je prováděl program sekvenční, to nemusí být vždy pravda a existují programy, které tak dokáží Amdahlovu zákonu uniknout.



Paměť a více vláken

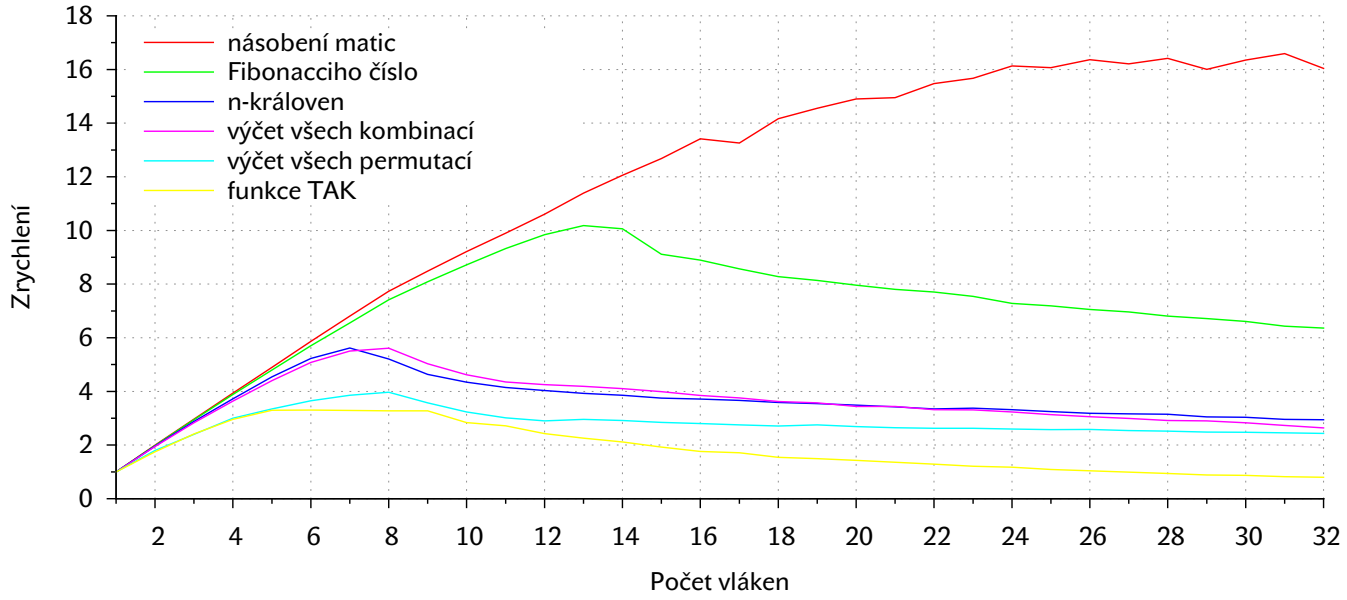
Z pohledu běhu programu ve více vláknech je jedním z největších úskalí práce s pamětí, která je sdílená mezi více vlákny. V případě imperativních nebo objektově orientovaných programovacích jazyků se sdílené paměti dá vyhnout jen velmi těžko. Obvykle se přístup ke sdílené paměti ošetřuje pomocí zámků, semaforů, monitorů a podobných jednoduchých synchronizačních prostředků. Toto řešení není příliš šťastné, protože může dojít (a často dochází) k nějakému přehlédnutí, které může skončit chybou souběhu (race-condition) nebo uváznutím (deadlockem), což jsou chyby, které jsou zákeřné tím, že vznikají často za velmi specifických podmínek, a proto se špatně odhalují a ladí. Problémem těchto synchronizačních nástrojů je jich jednoduchost, špatná komponovatelnost a nízká úroveň abstrakce. Používání zámků při tvorbě vícevláknových programů by se dalo přirovnat k programovaní v assembleru—programátor má sice plnou kontrolu nad tím, jak se program bude chovat, ale má i spoustu příležitostí udělat v programu chybu, o srozumitelnosti kódu nemluvě.

Jak z toho ven? Jako inspiraci si můžeme vzít databázové systémy, ty naprosto běžně zvládají zpracovávat velké množství požadavků prováděných souběžně a přitom to není pro programátora nějak extra náročné. Může za to systém transakcí, který zajišťuje to, že změny jsou prováděny v blocích (transakcích), které jsou od sebe izolovány, systém přechází z jednoho konzistentního stavu do druhého a změna je buď provedena jako celek nebo není provedena vůbec. Pokud dojde ke kolizi, např. ke dvěma konfliktním přístupům k jednomu záznamu v tabulce, systém to sám rozpozná a určí, která transakce se má provést, a která zopakovat.

Tento přístup lze aplikovat i na vícevláknové aplikace. Stačí mít prostředek, který určí, které části kódu se mají provádět jako ony zmíněné transakce a nechat provedení na běhovém prostředí, např. následovně:

```
atomic {
    account1.balance -= n;
    account2.balance += n;
}
```

Na rozdíl od běžných zámků nebo semaforů transakční zpracování poskytuje velmi dobrou míru abstrakce, transakce se dají do sebe komponovat a detekce



Vliv počtu jader na zlepšení výkonu na vybraných algoritmech implementových ve Schemiku.

konfliktů je přenechána běhovému prostředí. Není to však všespásné řešení. Potřebujeme, aby transakce šlo zopakovat, pokud je v kódu nějaký vedlejší efekt, vše se komplikuje, viz následující příklad.

```
atomic {
    account1.balance -= n;
    account2.balance += n;
    launchTheMissilles();
}
```

Pro některé programovací jazyky (Haskell, Java, Fortress) už podpora transakční paměti existuje a i náš Schemik ji má – každé vlákno má svůj izolovaný obraz paměti, a když dokončí svou činnost, běhové prostředí zkontroluje, zda došlo ke kolizi a pokud ne, ví, že výsledek je korektní a dá se použít. Pokud ke kolizi došlo je výpočet zopakován. Nepříjemnou vlastností je, že implementace transakční paměti vyžaduje kontrolu nad přístupy a změnami v paměti, což má svou režii. Proto Intel navrhl a implementoval rozšíření TSX pro instrukční sadu x86, avšak se v tomto rozšíření našla chyba, a proto toto rozšíření zablokoval a až nejnovější verze procesorů dostaly korektní hardwarovou podporu transakční paměti. Teď už jen zbývá čekat, jak se toho ujmou programátoři a jak se tento způsob programování uplatní.

- MyJIT – v rámci vývoje překladače Schemiku vznikla knihovna, které umožňuje za běhu (just-in-time) generovat strojový kód pro procesory rodiny x86, AMD64 a SPARC, a která byla vydána pod open-source licencí LGPL, s tím, že bude nápomocna při vývoji dalších k programovacích jazyků. K našemu milému překvapení si našla své uplatnění v dalších situacích, například při vývoji operačních systémů nebo jako součást systému Tigress vyvíjeného na University of Arizona, který slouží k dynamické obfuskaci prováděného kódu.
- Autor tohoto článku, Petr Krajča, získal grant Grantové Agentury ČR na výzkum modelů pro implicitně paralelní programování.

Enigma

Petr Osička

Enigma je elektromechanické zařízení pro šifrování a dešifrování zpráv. Byla využívána od dvacátých let minulého století, kdy ji navrhl německý inženýr Arthur Scherbius, nejdříve v komerční sféře a později v tajných službách a armádě. Využívala ji zejména nacistická vojska během 2. světové války. Příběh o Enigmě a jejím prolomení je součástí veřejného povědomí, stal se i námětem několika knih a filmů. Enigma je nicméně zajímavá i pro informatiky. Pokud porovnáme šifrování používané v současnosti s šifrováním Enigmou, uvědomíme si, jaký pokrok si v kryptografii vynutil rozvoj počítačů. Na druhé straně, snaha o prolomení Enigmy vedla k návrhům různých typů počítacích strojů, jež můžeme s trochou nadsázky označit za předchůdce počítačů. V článku se budeme zabývat tím, jaký typ šifry vlastně Enigma používala, popisem toho, jak Enigma vypadala a fungovala, a postupem, jaký používala nacistická vojska pro šifrování a dešifrování. V některém z budoucích čísel magazínu budeme pokračovat rozborem prolomení Enigmy.

Alfabetické šifry

Monoalfabetickou šifrou rozumíme jednoduchou substituční šifru, kdy každé písmeno abecedy nahradíme jiným písmenem podle jednoduchého pravidla. Jednoduchým příkladem je *Caesarova šifra*, ve které je pravidlo dáno posunem. Pro nenulové číslo n, které je seshora omezeno velikostí abecedy, zakódujeme znak právě tím znakem, který je v abecedě o n pozic dále. Pokud bychom se tak dostali za poslední znak abecedy, začneme zase od začátku (analogicky k modulo aritmetice). Pro pětiprvkovou abecedu *a,b,c,d,e* a $n = 2$, je Caesarova šifra dána tabulkou

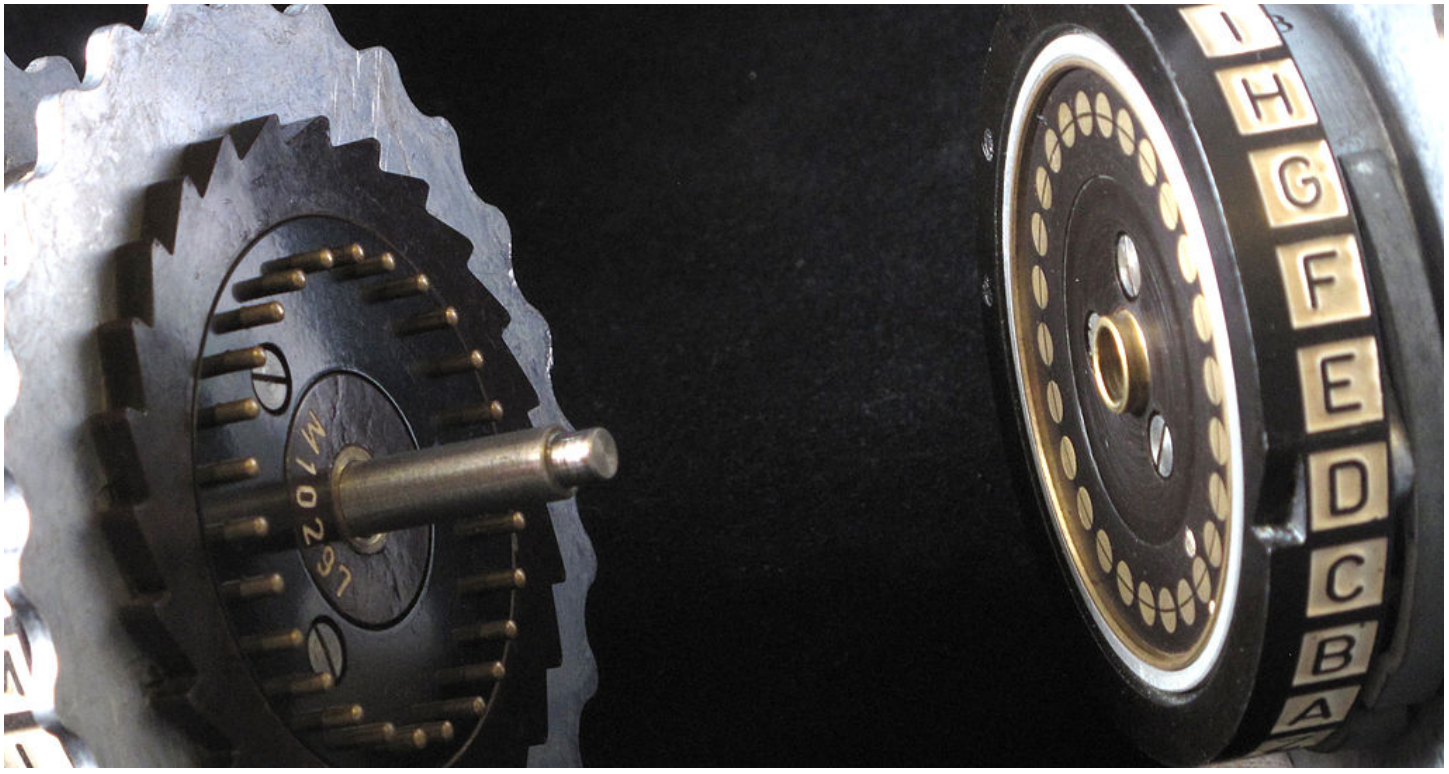
znak	šifra
a	c
b	d
c	e
d	a
e	b

a například slovo *bac* zašifrujeme jako *dce*. Za obecného předpokladu, používaného u vyhodnocení bezpečnosti všech kryptografických systémů, tedy že

> *vždy musíme předpokládat, že narušitel zná princip šifrovací metody, kterou používáme,*

lze Caeserovu šifru rozluštit za pár minut pouze s tužkou a papírem. Počet možných klíčů Caesarovi šifry je roven pouze počtu písmen v abecedě a je tedy snadné vyzkoušet všechny možnosti.

Sofistikovanější monoalfabetické šifry používají libovolné nahrazení jednotlivých písmen jinými písmeny takové, že různá písmena jsou nahrazena různě (a nemůže tedy dojít k situaci, kdy jsou dvě různá písmena nahrazena stejným písmenem). Klíčem nyní není velikost posunutí, ale celý sloupeček *šifra* z předchozí tabulky. Počet možných klíčů takové šifry pak velmi rychle roste, je roven faktoriálu z velikosti abecedy. Pro abecedu o 26 znacích je to přibližně $404 \cdot 10^{24}$ možných klíčů. I když je takovou šifru bez použití počítače mnohem těžší prolomit, úplně bezpečná také není. Její slabostí je možnost provedení statistické analýzy a použití poznatků, které náme o jazyce. O angličtině například víme, že *e* je nejčastější vyskytující se znak a *the* je nejčastěji se vyskytující slovo.



Detail rotorů. Vpravo je abecední kroužek.

Vylepšením monoalfabetických šifer, které vzniklo v 19. století, jsou *polyalfabetické* šifry. Princip vylepšení spočívá v tom, že místo jedné monoalfabetické šifry jich máme více a používáme je střídavě. Mimo jednotlivých monoalfabetických šifer je tedy součástí šifrování i pevně dané pravidlo, které určuje, jakým způsobem budeme jednotlivé šifry střídat. Enigma používá právě polyalfabetickou šifru, kdy je každý znak posílané zprávy zašifrován pomocí jiné monoalfabetické šifry (za předpokladu, že posílaná zpráva není příliš dlouhá, viz dále).

Enigma

Polyalfabetické šifry jsou náročné na ruční šifrování. Je potřeba velké množství šifrovacích tabulek (pro každou z použitých monoalfabetických šifer jedna) a navíc je nutné správně dodržovat pravidla pro střídání šifer. Z toho důvodu často lidé při šifrování nebo dešifrování zpráv dělají chyby. Enigma řeší oba dva problémy následovně:

- Elektrická část počítá jednu monoalfabetickou šifru. Funguje tak, že po zmáčnutí tlačítka na klávesnici se uzavře obvod a na výstupu se rozsvítí písmeno, které je zakódováním písmena odpovídajícího zmáčknuté klávese.
- Mechanická část se stará o výměnu jednotlivých monoalfabetických šifer. Po každém zmáčknutí tlačítka na klávesnici se aktuální monoalfabetická šifra, kterou Enigma šifruje, změní.

Obvod, který se uzavře při stistknutí tlačítka na klávesnici je veden přes následující komponenty Enigmy: rozvodnou desku, rotory (obvykle 3), reflektor, zpět přes rotory, rozvodnou desku a klávesnici (přes jiné tlačítko než bylo zmáčknuto na počátku, viz část o reflektoru dále) do žárovky a baterie.

Rotory jsou otočná kolečka (viz obrázek), která jsou v Enigmě postavena vedle sebe. Každý rotor má na obou svých stranách 26 kontaktů uspořádaných do kruhu. Každý kontakt odpovídá jednomu písmenu abecedy. Přitom kontakty z obou stran rotoru jsou propojeny tak, že jeden kontakt z levé strany je spojen právě s jedním kontaktem z pravé strany. Pokud si přestavíme proud plynoucí z jedné strany rotoru na druhou, můžeme říci, že rotor přestavuje jednu monoalfabetickou šifru. Kontakty rotorů postavených vedle sebe jsou propojeny. Proud tedy prochází přes všechny rotory a tvoří tak šifru složenou ze tří monoalfabetických šifer daných jednotli-

Enigma. V přední části je rozvodná deska.

vými rotory. Podotkněme ještě, že písmena abecedy nejsou na rotoru přiřazena kontaktům pevně, ale lze je nastavit pomocí *abecedních kroužků*.

Každý rotor je schopen se otáčet o 1/26. První rotor se otočí po každém zmáčnutí klávesy. Druhý rotor se poprvé otočí po dostatečném počtu otočení prvního rotoru (tento počet je dán umístěním *zarážky* na prvním rotoru) a poté vždy po 26 otočeních prvního rotoru. Otáčení dalších rotorů je dáno analogickým vztahem, tedy třetí rotor se otočí o 1/26 po úplném otočení druhého rotoru atd. Vynecháme-li drobný technický detail, můžeme říci, že rotory se dostanou otáčením z jakékoliv konfigurace zpět do stejné konfigurace po přibližně 26^3 zmáčknutích.

Reflektor přepojí výstup jednoho kontaktu na posledním rotoru do jiného kontaktu na posledním rotoru a proud pak projde zpět přes rotory. Důsledkem tohoto zapojení je, že Enigma nedokáže šifrovat znak sebou samým (tj. například písmeno *a* nelze zašifrovat jako *a*) a že postup šifrování a dešifrování zprávy je naprosto stejný, za předpokladu, že Enigmu na počátku nastavíme stejně u šifrování jako u dešifrování.

Rozvodná deska realizuje další monoalfabetickou šifru, která se skládá se šiframi odpovídajícím rotorům. Deska obsahuje 26 zástrček, každá z nich odpovídá jednomu písmenu. Zástrčky lze po dvojicích propojit kabelem a prohodit tak ve výsledné šifře odpovídající písmena. Standardem bylo propojení 10 dvojic písmen. Nutno podotknout, že rozvodná deska je německou inovací, která obrovským způsobem rozšířila počet možných konfigurací Enigmy. Šifry odpovídající jednotlivým rotorům jsou totiž stálé, kdežto šifru odpovídají rozvodné desce lze nastavit.

Nakonec dodejme, že Enigma existovala ve více typech, které se lišily v různých ohledech. Například námořnictvo mělo na výběr z 8 rotorů (namísto 5), používalo 4 a nikoliv 3 rotory, a později mělo i konfigurovatelný reflektor.

Šifrování a dešifrování

Předtím, než si řekneme postup pro šifrování a dešifrování zprávy, projděmě si, co vše je součástí nastavení Enigmy. Je to: (1) nastavení abecedních kroužků (viz část o rotorech nahoře), (2) výběr 3 rotorů z 5 možných, (3) nastavení rozvodové desky, (4) počáteční pozice rotorů. Počet všech možných počátečních konfigurací, za předpokladu, že jsou známa šifrování jednotlivých rotorů, je přibližně 10^{23}.

První tři součástí nastavení byly pro každých 24 hodin předepsány v šifrovací knize, která byla kurýrem rozdistribuována mezi německé jednotky. Počáteční pozice rotorů byla pro každou šifrovanou zprávu nastavena zvlášť a existovaly různé způsoby, jak ji poslat jako součást zprávy (poznamenejme, že počáteční pozici rotorů lze jednoznačně popsat pomocí tří písmen, která jsou „nahoře“ na již nastaveném abecední kroužku daných rotorů). Následující postup byl používán německou armádou před 2. světovou válkou. K odeslání zprávy, označme si ji z, šifrovací pracovník

1. Nastavil Enigmu do základní konfigurace podle šifrovací knihy.
2. Vybral a nastavil startovací pozice rotorů. Tyto pozice (tj. tři písmena jednoznačně ji reprezentující) byly součástí posílané zprávy.

3. Vybral cílené startovací pozice rotorů a pomocí Enigmy v současném nastavení je dvakrát zašifroval. Obržel tak šest písmen, která jsou zašifrováním cílené startovací pozice rotorů. Tato písmena byla součástí zprávy.

4. Nastavil rotory do cílené startovací pozice a zašifroval zprávu z.

Pro dešifrování zprávy poté provedl následující

1. Nastavil Enigmu do základní konfigurace podle šifrovací knihy.

2. Nastavil rotory podle počátečních tří písmen přijaté zprávy.

3. Dešifroval následujících šest písmen a obdržel tak cílené startovací pozice rotorů a Enigmu příslušně nastavil.

4. Dešifroval zbytek zprávy a obdržel tak z.

Všimněme si, že cílená startovací pozice byla posílána dvakrát. Němci to zřejmě prováděli jako jednoduchou formu samoopravného kódu, chtěli se ujistit, že tato část zprávy přijde správně. Byla to ovšem kryptografická chyba, která později přispěla k prolomení Enigmy polskou šifrovací agenturou. Na tomto prolomení se podílela trojice polských matematiků Marian Rejewski, Jerzy Różycki a Henryk Zygalski. O příspěvku Poláků k prolomení Enigmy se často nemluví a všechny zásluhy jsou neprávem dávány týmu z anglického Bletchley parku. I když se anglický tým potýkal se složitější verzí Enigmy (například byla přidána rozvodná deska a více rotorů), měl k dispozici poznatky svých polských kolegů. Více o prolomení Enigmy si ovšem ponecháme do některého z příštích čísel magazínu.

Hádanka

Pozorně si přečtěte následující kus kódu v jazyce C. Myslíte si, že lze tento kód přeložit? Podobný kus kódu by šel navrhnout i v jiných jazycích, např. v Javě, C++ nebo C#. Jako vždy najdete správné řešení na webových stránkách magazínu.

```
#include <stdio.h>

int main()
{
        http://www.inf.upol.cz/magazin
        printf("Pujde tento program prelozit a proc?\n");
        printf("Reseni se dozvite na vyse uvedene adrese.\n");
        return 0;
}
```

Redakce:
Petr Krajča, Petr Osička
Katedra informatiky PřF UP
17. listopadu 12
771 46 Olomouc

web: www.inf.upol.cz/magazin
email: magazin@inf.upol.cz
atom: http://www.inf.upol.cz/atom/magazin
telefon: 585634701
GPS: 49.591870, 17.262336