Zaciemnianie programów

Tomasz KAZANA

```
#define n ((e[++a]-42)/9-f[d+1])?
#define o printf("%c",c[" 01./::|\\"]);
char e[]="**3<HRZcir+3@OXakt;=GOXds*\?HRZcir*7HNZ'i19JS\\p*H[m1:CJSz*>H[`mr25\
\?Hx,P,B2Gs-KTfzRdvISeyCR-ISeu.<Ev+9+P,z,4PfzIdvO2*HRZcir6GPis=MU*3HRZcir*HZi\
1JS\\epy*>H[m1JSey*DH[m*3<HZiu-@P*3HZi<N]q1JS\\epy:[m1CJSeny06I[m*4\?HRZcir,\
\?*6HRZcir1J]q2K*H[m2K*H[m20K]qt0@M2DK]q,Jq1JS\\epy[m1:JSey+[m*3<HRZcir13Gt,\
=GVs*3<HRZcir1J]qz*HF*AH2;DK]quaO=G2:K]q]q1CJS\\pDVu1:JS*D!+3:BIOSY'egilqtxz\
\177.0249<==>EJMUY'ejov#$59@CJOXYZbfhlnrxy&+.57=@IMR[``bcfmnq!#),@",*f;

int main(int j,char *k[]) {
  int a,b,c,d,g,h,i=19;
  printf("      ");
  for(g=0[f=(char *)calloc(80+(h=atoi(1[k])),1)]=1; g<=h; g++) {
    if ((g>30)&&(f[i-2]+f[i-1]!=0)) i++;
    for(d=c=0; d<i; d++) {
      o f[d]=(e[b=c*9,b+=(c=d[f]),((a=e[345+b]+b/19*85-33-b/40*12-b/80*4)[e]
  -42)/9-f[d+1])?n n n n n n n n 0:a:a:a:a:a:a:a:a]-42)%9; }
      o if (0[f]-1) printf("\n%6i ",g); else printf("\n      "); } }
```

Każdy młody adept sztuki programowania pewnie nie raz słyszał od swych mentorów, że program nie tylko musi działać poprawnie i szybko, ale też musi być napisany w sposób czytelny. Studentów uczy się więc nie tylko języków programowania, algorytmów czy struktur danych, ale też próbuje się im przekazać prawidłowe nawyki dotyczące *stylu* programowania.

Buntownicy, jak zawsze, idą w drugą stronę. Od roku 1984 organizowany jest konkurs *International Obfuscated C Code Contest* na napisanie możliwie nieczytelnego programu w języku C. Na marginesie umieściliśmy zgłoszenie Nicolasa Ollingera, które w roku 2001 (ex aequo z Jasonem Orendorffem) zdobyło główną nagrodę. Czytelnik Chorobliwie Zawzięty może sprawdzić, że ten program realizuje działanie sita Eratostenesa.

Problem zaciemniania programów jednak wybiega daleko poza obszar sztubackich żartów.

Zaciemnianie (fachowo: obfuskacja) kodu źródłowego programu to jedno z ważnych wyzwań kryptologii. Motywacja do badania tego problemu jest chyba dość jasna. Dla zapewnienia bezpieczeństwa danych (albo z powodów czysto biznesowych) czasem autor programu chce tylko umożliwić jego wykonywanie komuś innemu, ale wcale nie chce ujawniać szczegółów jego działania. W dobie Internetu takie założenie trywialnie można zrealizować w modelu klient-serwer. Wystarczy założyć, że kod pewnego programu P znany jest tylko serwerowi, a jeśli użytkownik chce wykonać P na danych D, to powinien wysłać D do serwera, a ten obliczy i odeśle mu $P(D)$. Problem obfuskacji jest jednak trudniejszy. Chcemy mieć własność jak wyżej, ale w świecie *off-line*. Innymi słowy: zakładamy, że użytkownik przechowuje sam kod programu P, a i tak najlepsze, co może zrobić, by poznać jego działanie, to uruchamiać P na różnych danych i odczytywać wyniki. Chcielibyśmy, aby kod był tak zagmatwany, żeby każda próba jego obejrzenia i wyciągnięcia jakichś bardziej ogólnych wniosków musiała kończyć się niepowodzeniem.

Jak w całej kryptologii, początkowo do problemu zaciemniania kodu podchodzono w sposób nieformalny (czy raczej: niematematyczny). To znaczy definicje problemu były nieostre, a rozwiązania dyskutowane były wyłącznie na poziomie intuicyjno-inżynierskim. Sprawę zmieniła słynna prorocza praca *New Directions in Cryptography* Diffiego i Hellmana z roku 1976, w której (między innymi) podjęto próbę sformalizowania pojęcia obfuskacji programu. Konkretnie: niech $O(P)$ oznacza próbę zaciemnienia programu P. Powiemy, że $O(P)$ jest rzeczywiście obfuskacją P, gdy (1) dla każdego wejścia, P oraz $O(P)$ dają ten sam wynik oraz (2) dla każdego (wielomianowego) użytkownika A istnieje taki symulator S (czyli po prostu pewien wielomianowy program), że:

$$A\big(O(P)\big) \approx S^P,$$

gdzie: $A\big(O(P)\big)$ oznacza wynik działania A na danym kodzie $O(P)$, a S^P oznacza wynik działania S, który ma dostęp do serwera takiego, jak w akapicie wyżej (czyli obliczającego na żądanie wartości programu P). Wyniki programów nie muszą być deterministyczne, stąd znaczek „$\approx$” oznacza tutaj bliskość rozkładów możliwych wyników. Intuicyjnie ta definicja głosi, że użytkownik A, który ma pełny dostęp do kodu $O(P)$, nie ma specjalnej przewagi nad użytkownikiem dysponującym tylko dostępem do serwera, skoro ten drugi może symulować (w czasie wielomianowym i z dużą dokładnością) wynik obliczeń tego pierwszego.

Marzeniem kryptologów przez wiele lat pozostawał uniwersalny kompilator, który dla każdego programu potrafiłby wyznaczyć jego nową wersję, zaciemnioną według definicji wyżej. Niestety, w roku 2001 Barak et al. udowodnili, że taka uniwersalna konstrukcja nie może istnieć. Na długi czas ostudziło to zapał środowiska. Jednak przełom pojawił się w roku 2013, gdy kolejna grupa badaczy, pracując ze słabszą wersją definicji obfuskacji (*indistinguishability obfuscation*, w skrócie iO), wskazała (bez dowodu) obiecującego kandydata na uniwersalny kompilator. Od tamtej pory ruszyła lawina. Znamy już wiele uniwersalnych kompilatorów iO, dowodliwie poprawnych przy różnych, mniej lub bardziej egzotycznych założeniach. Znamy też wiele zastosowań, pokazujących, że sama definicja (nieprezentowana w tym tekście) iO nie jest za słaba w praktyce. Wciąż jednak wiele bardzo ważnych pytań w tej dziedzinie pozostaje bez odpowiedzi. Przede wszystkim:

Czy istnieje uniwersalny kompilator zaciemniający (w sensie iO), poprawny przy standardowych założeniach (a więc np. o istnieniu funkcji jednokierunkowych)?