2 Забазпечення комп'ютерної безпеки в державних, банківських та інших інформаційних системах

УДК 004.4

МЕТОДИ РЕАЛІЗАЦІЇ КРИПТОБІБЛІОТЕК

Олександр Голяка
НБУ

Анотація: **Розглядається проблема оптимальної за швидкістю виконання реалізації асиметричної криптобібліотеки. Показані підходи до реалізації, з яких обрано найбільш швидкий, а також основні методи криптобібліотеки.**
Summary: **This paper considers the execution speed problem of asymmetric cryptolibrary realization. Showed approaches to the implementation from which was chosen the most rapid, and also showed fundamental cryptolibraries methods.**
Ключові слова: **Криптобібліотека, мова «С», асемблер.**

Вступ

Існує два основних підходи до реалізації криптобібліотек – реалізація мовою високого рівня, такого, наприклад, як "С", і реалізація мовою низького рівня – асемблері (assembler). Перевагою реалізації програм мовою "С" є її портуємість, що означає запуск і компіляцію на будь-якій платформі з будь-якою архітектурою, для якої передбачений відповідний компілятор [1]. Однак реалізація на "С" є менш оптимальною за параметром швидкості обчислень (продуктивності), ніж реалізація на асемблері, тому що при програмуванні на "С" в значній мірі не беруться до уваги архітектурні особливості тієї чи іншої платформи. Це об'єктивний фактор і зумовлений він самою структурою і семантикою мов високого рівня. З іншого боку, розробка на асемблері передбачає досконале знання складної мікропроцесорної архітектури [2, 3]. Програміст повинен розуміти властивості асемблерних команд, правила їх обробки, структуру конвеєра та інших функціональних елементів процесора. У ряді випадків необхідна вузькоспеціалізована інформація про структуру кешу і пам'яті. Розробка на асемблері є скрупульозним, тривалим і дорогим завданням. Тому можна стверджувати, що професійний, сучасний компілятор, який може застосувати знання про архітектуру процесора, здатний в ряді випадків видати більш ефективний код, ніж недостатньо продумана або пряма реалізація на асемблері.

В даному контексті реалізація на "С" та асемблері має такі відмінності:

- реалізація на "С": портуємість, відносна дешевизна розробки, відносно невеликий час розробки, порівняно низька швидкість виконання;
- реалізація на асемблері: не портуєма, дорожнеча розробки, більший час розробки, висока швидкість виконання.

Крім того, є кілька моментів, які необхідно брати до уваги при виборі тієї або іншої технології, при реалізації програмних модулів.

Реалізація на "С" так чи інакше, залежно від методу реалізації, може вимагати використання так званих розширених типів даних. Розширені типи даних не є стандартними типами даних мови "С", але більшість компіляторів підтримують їх, як, наприклад, типи даних __int64 або long long. Використання цих типів даних обумовлено деяким збільшенням швидкості. В той же час передбачається їх підтримка з боку компіляторів. Це означає, що використання розширених типів даних призводить до незначної втрати портуємості на користь підвищення продуктивності.

При виборі асемблера як мови реалізації виникає проблема портуємості. Як тільки ця проблема вирішується, слід згадати про вартість реалізації на асемблері. Одним із способів є аналіз складності та вартості реалізації основних алгоритмів (наприклад, методу Монтгомері). Це потребує написання більшої кількості коду та ще більше часу на налагодження. Можна з впевненістю припустити, що в більшості випадків це не є доцільним.

Постановка задачі

Метою статті є розгляд ефективних підходів до реалізації криптографічних бібліотек/модулів, включаючи основні, а саме:

- реалізація на стандартному (ANSI) "C";
- реалізація на "C" з розширеними типами даних;
- повна реалізація на асемблері;
- реалізація на "C" з ядром на асемблері.

Підходи до реалізації криптографічних бібліотек

Реалізація на ANSI "C".

У мові "C" операнди арифметичного виразу перед обчисленням перетворюються до єдиного типу, який називається перетворений тип. Значення змінної може бути приведено до більшого або меншого значущого типу даних через усічення або доповнення, відповідно. При усіченні старші біти відкидаються, а при доповненні старші біти більш значущого типу даних заповнюються нулями. Результатом арифметичної операції також є перетворений тип даних. Усічення унеможливлює доступ до старших бітів результату деяких арифметичних операцій, що призводить до необхідності емуляції цих операцій. Наприклад, операція додавання. Нехай необхідно скласти два числа, розмірністю w біт, тоді розмір результату в гіршому випадку буде $w+1$ біт:

$$(2^w-1)+(2^w-1)=2^{w+1}-2,$$

$$(2^{w+1}-1)>(2^{w+1}-2)>(2^w-1).$$

Отже, в результаті додавання операндів виникає додатковий біт переносу, який відповідно до правил перетворення буде усічений. Процедура додавання, наведена нижче, складає числа розмірністю w біт, зберігаючи при цьому молодші w біт результату в S, а старший $(w+1)$-й біт переносу – в C:

```
#define WSIZE (8 * sizeof(word))
#define MSBMASK ((word)1 << (WSIZE - 1))
S = (a & ~ MSBMASK) + (b & ~ MSBMASK);
C = (a >> (WSIZE - 1)) + (b >> (WSIZE - 1)) + (S >> (WSIZE - 1));
S = a + b;
C >>= 1.
```

Операція множення в "C" зберігає лише молодшу частину результату, стосовно операндів максимальної розмірності. Нехай маємо $c=a\cdot b$, a і b є словами (word). Результатом операції буде величина, розмірність якої також буде word. Для отримання всього добутку необхідно кожен w-бітний операнд розбити на два числа по $w/2$ біт. Така процедура дозволяє отримати результат у вигляді пари слів (C,S).

```
#define WSIZE (8 * sizeof(word))
#define LOWBITS(x) ((x) & (~ ((word)0) >> (WSIZE/2)))
#define HIGHBITS(x) ((x) >> (WSIZE/2))
albl = LOWBITS (a) * LOWBITS (b);
ahbl = HIGHBITS(a) * LOWBITS (b);
albh = LOWBITS (a) * HIGHBITS(b);
ahbh = HIGHBITS(a) * HIGHBITS(b);
sum = LOWBITS(albh) + LOWBITS(ahbl) + HIGHBITS(albl);
S = (sum << (WSIZE/2)) + LOWBITS(albl);
C = ahbh + HIGHBITS(albh) + HIGHBITS(ahbl) + HIGHBITS(sum).
```

Реалізація на ANSI "C" з розширеними типами даних.

В основі цього підходу лежить використання нестандартного типу даних мови "С". Код, як і раніше, залишається портуємим, проте його використання обмежене кількістю платформ, які підтримують нестандартні розширення мови. Суть методу полягає в тому, що операнд, розмір якого вдвічі більший розміру регістра загального призначення, містить всі біти результату арифметичних операцій. Нехай розширений тип даних називається dword, тоді операції додавання та множення можна визначити таким чином:

```
# define WSIZE (8 * sizeof(word))
CS = (dword)a + (dword)b;
S = (word)CS;
C = (word)(CS >> WSIZE);
```

```
# define WSIZE (8 * sizeof(word))
CS = (dword)a * (dword)b;
S = (word)CS;
C = (word)(CS >> WSIZE).
```

Тут C і S мають тип word, а $C \cdot S$ - dword. Розширені типи можна використовувати і для інших операцій, наприклад, зсув і ділення.

Цей підхід не передбачає асемблерного програмування. В той же час компілятор повинен підтримувати розширені типи даних і вміти генерувати оптимальний код для фрагментів коду "С" з використанням цих типів даних. В даний час більшість компіляторів підтримують розширені типи даних. Наприклад: Microsoft Visual C++ має тип даних __int64, а компілятори для сімейства операційних систем Unix – long long. Розмір цього типу даних вдвічі більше розміру регістра процесора відповідної платформи.

Реалізація на асемблері.

Оптимальна реалізація криптографічних алгоритмів на асемблері передбачає детальне вивчення архітектури процесора. Необхідне глибоке розуміння моментів, пов'язаних з архітектурою набору команд, простору регістрів, різноманітних функціональних елементів, ієрархії пам'яті і т. д. Асемблерна реалізація дозволяє отримати швидкий код, жертвуючи при цьому портуємістю.

Реалізація на ANSI "C" з ядром на асемблері.

Збільшення швидкості, що досягається за рахунок застосування розширених типів даних, не є занадто значною в зв'язку з неефективним використанням процесорної архітектури. Продуктивність обмежена можливостями компілятора з оптимізації коду. Існує альтернативний підхід, який дозволяє поєднати гнучкість мови "С" і швидкість асемблера. Цього можна досягти, виділивши невеликий набір операцій, реалізація яких виконується на асемблері. Весь інший код виконується на "С". Можливий набір операцій наведено нижче.

Таблиця 1 – Набір операцій, реалізованих на асемблері

Операція	Визначення
ADD(C,S,a,b)	$(C,S) := a + b$
ADD2(C,S,a,b,c)	$(C,S) := a + b + c$
MUL(C,S,a,b)	$(C,S) := a \cdot b$
MULADD(C,S,a,b,c)	$(C,S) := a \cdot b + c$
MULADD2(C,S,a,b,c,d)	$(C,S) := a \cdot b + c + d$
SUB(C, S, a, b)	$(C,S) := a - b$
SUB2(C, S, a, b, c)	$(C,S) := a - b - c$
DIV(qh,ql,r,ah,al,b)	$(qh,ql) := [(ah,al)/b], r := (ah,al) \bmod b$

Ці операції необхідно реалізовувати у вигляді макросів або асемблерних вставок. Звичайно, можна реалізовувати їх і у вигляді функцій, але це однозначно призведе до втрати швидкості. Обсяг асемблерного коду мінімальний: кожна операція може бути реалізована 4-8 асемблерними інструкціями. В результаті грамотно спроектована реалізація, де поєднаний асемблерний код і код на "С", може з легкістю бути портуєма на іншу платформу. Для цього необхідно лише розробити для конкретної платформи машини асемблерний код, замінивши ним існуючі ділянки коду.

Основні арифметичні обчислення криптобібліотеки та їх реалізація через операції ядра

При визначенні набору арифметичних операцій ядра необхідно проаналізувати алгоритми та реалізацію криптосистем, в основі яких лежить операція модулярного зведення в ступінь. В той же час ядро повинно містити якомога менше операцій. Основними арифметичними операціями над числами великої розмірності, які використовують, наприклад, такі криптосистеми як RSA, DH, DSA, є додавання, множення, ділення, модулярне зведення в ступінь [4],[5].

Додавання.

Нехай a і b – числа, записані в системі числення за основою W :

$$a = (a_{k-1}, a_{k-2}, \ldots, a_0) = \sum_{i=0}^{k-1} a_i \cdot W^i$$

$$b = (b_{k-1}, b_{k-2}, \ldots, b_0) = \sum_{i=0}^{k-1} b_i \cdot W^i$$

$$a_i, b_i \in [0, W-1].$$

Іншими словами, якщо основа системи числення $W=2^w$, де w – розмір машинного слова (word) процесора, то числа a і b є масивами типу word, а k – розмір масиву (кількість слів). Тоді s – результат додавання чисел a і b :

$$s = a + b,$$

$$s = (s_k, s_{k-1}, s_{k-2}, \ldots, s_0) = \sum_{i=0}^{k} s_i \cdot W^i,$$

причому

$$\begin{aligned}
&(c_0, s_0) = a_0 + b_0,\\
&(c_1, s_1) = a_1 + b_1 + c_0,\\
&(c_2, s_2) = a_2 + b_2 + c_1,\\
&\vdots\\
&(c_k, s_{k-1}) = a_{k-1} + b_{k-1} + c_{k-2}.
\end{aligned}$$

Число s може бути отримано з використанням операції ядра ADD2. Відповідний фрагмент коду представлений нижче. Тут змінні a, b і c – масиви типу word, k – розмір масивів (кількість слів), C і S (Carry, Sum) також мають тип word:

```
C = 0
for i = 0 to k - 1 do
        ADD2 (C, S, a[i], b[i], C)
        s[i] = S
s[k] = C.
```

Множення.

Нехай a і b – числа, записані в системі числення за основою W :

$$a=(a_{k-1},a_{k-2},...,a_0)=\sum_{i=0}^{k-1}a_i\cdot W^i,$$

$$b=(b_{k-1},b_{k-2},...,b_0)=\sum_{i=0}^{k-1}b_i\cdot W^i,$$

$$a_i,b_i\in[0,W-1].$$

Класичний алгоритм множення (друга назва – метод олівця та паперу) використовує часткові добутки через множення компоненти множника b на все число a, з наступною сумою цих часткових добутків для отримання всього результату – числа s:

$$s=a\cdot b,$$

$$s=(s_{2k-1},s_{2k-2},s_{k-3},...,s_0)=\sum_{i=0}^{2k-1}s_i\cdot W^i.$$

Класичний алгоритм множення приведено нижче:

```
Ââ³ä: a,b
Âèâ³ä: s = a * b
0.    ï ðèñâî ¿òè  t_i:=0 äëÿ âñ³õ i=0,1,…,2k-1
1.    for i=0 to k-1
2.        C:=0
3.        for j=0 to k-1
4.            (C,S):=s_{i+j} + a_j · b_i + C
5.            s_{i+j}:=S
6.        s_{i+k}:=C
7.    return (S_{2k-1}, S_{2k-2},…, S_0).
```

Для реалізації алгоритму необхідно виконати на асемблері крок 4: $(C,S):=s_{i+j}+a_j\cdot b_i+C$, де змінні s_{i+j},a_j,b_i,C і S є словами (word) або w – бітними числами. В даному випадку можна використовувати операцію ядра MULADD2. Відповідний фрагмент коду приведено нижче. Змінні a,b,s – масиви типу word. Розмір масивів a і b – k слів. C і S (Carry, Sum) – змінні типу word.

```
for i = 0 to 2 * k - 1 do s[i] = 0
for i = 0 to k - 1 do
    C = 0
    for j = 0 to k - 1 do
        MULADD2 (C, S, a[i], b[j], s[i + j], C)
        s[i + j] = S
    s[i + k] = C.
```

Ділення.

Класичний алгоритм ділення з остачею.

Класичний алгоритм є формулюванням “шкільного” методу ділення. Зміст методу зводиться до такого: нехай ділене U складається з l розрядів, а дільник V – з n розрядів. Алгоритм за $l-n$ кроків обчислює значення Q і R такі, що виконується співвідношення $U=Q\cdot V+R$. Кожний крок алгоритму виконує ділення числа Z, в якому $(k+1)$ розрядів, на дільник V, розміром в k розрядів, отримуючи таким чином один розряд частки Q і k розрядів остачі R. Кожна остача менше дільника V, тому її можна сполучити з

наступним розрядом діленого, утворюючи число виду $R \cdot b +$ (наступний розряд діленого). Розмір отриманої таким чином остачі вже k+1 розрядів. Це число використовується як нове Z на наступному кроці алгоритму.

Формалізація цього алгоритму полягає в якомога більш точному наближенні до розряду частки Q. Ділення двох старших розрядів Z на старший V_{n-1} -й розряд числа V дає наближення $\hat{q}$, яке не менше істинного q, а при $V_{n-1} \geq [b/2]$ відрізняється від нього не більше ніж на 2. Більше того, якщо використовувати додаткові розряди чисел Z і V при оцінці частки (тобто три старших розряди Z і два старших розряди V), то ця оцінка (або наближене значення) буде практично завжди вірним або відрізнятись від істинного не більше ніж на 1. Причому ця ситуація можлива лише з вірогідністю $\approx 2/b$. Псевдокод цього алгоритму приведено нижче.

```
Ââ³ä: U- ä³ëåí å,  V- ä³ëüí èê
Âèâ³ä: Q - ÷àñòêà,  R - î ñòà÷à
0:      ï ðèñâî¿òè  Z:=U
1:      if  Z > V·b^(l-n)  then
2:              Z := Z-V·b^(l-n)
3:      for i=(l-1) down  to n do
4:              if  Z_i  = V_(n-1) then
5:                      q̂ := b-1
6:              else
7:                      q̂ := (Z_i ·b + Z_(i-1)) div V_(n-1)
8:              while  q̂·( V_(n-1)·b + V_(i-2)) > Z_i·b^2  + Z_(i-1)·b + Z_(i-2)  do
9:                      q̂ := q̂-1
10:             Q_(i-n)  := q̂
11:             Z := Z - q̂ ·V·b_(i-n)
12:             if Z<0 then
13:                     Z := Z + V·b_(i-n)
14:     ï ðèñâî èòü R := Z
15:     return (Q, R).
```

Ефективна робота цього алгоритму залежить від виконання циклу в рядочку 8. При використанні нормалізації рядок 9 буде виконаний з вірогідністю $\approx 2/b$.

У загальному випадку нормалізація полягає в домноженні U і V на нормалізатор d такий, що виконується нерівність $d \cdot V_{n-1} \geq [b/2]$. Звідси випливає, що $d = [b/V_{n-1}]$. На двійковому комп'ютері b буде степеню двійки і тому нормалізація може бути дуже ефективно виконана методом лінійного зсуву числа V вліво на стільки біт, скільки потрібно для установки самого старшого біта розряду V_{n-1} в 1. В результаті нормалізації довжина числа U збільшується на один розряд. В кінці алгоритму ділення істинне значення остачі R отримується за допомогою денормалізації, тобто ділення числа R на d або лінійного зсуву R вправо на стільки біт, на скільки, в свою чергу, були зсунуті числа U і V.

При реалізації алгоритму не зовсім зручною є реалізація перевірки в рядочку 8. А саме, нерівності $\hat{q} \cdot (V_{n-1} \cdot b + V_{i-2}) > Z_i \cdot b^2 + Z_{i-1} \cdot b + Z_{i-2}$. Переформулювати цю перевірку можна таким чином: нехай

$$\hat{q} := (Z_i \cdot b + Z_{i-1}) \text{ div } V_{n-1}$$
$$\hat{r} := (Z_i \cdot b + Z_{i-1}) \text{ mod } V_{n-1}$$

де $\hat{q}$ і $\hat{r}$ відповідно частка і остача. Тоді:

$$(Z_i \cdot b + Z_{i-1}) = \hat{q} \cdot V_{n-1} + \hat{r} \Leftrightarrow$$
$$b \cdot (Z_i \cdot b + Z_{i-1}) = b \cdot (\hat{q} \cdot V_{n-1} + \hat{r}) \Leftrightarrow$$
$$b \cdot (Z_i \cdot b + Z_{i-1}) + Z_{i-2} = b \cdot (\hat{q} \cdot V_{n-1} + \hat{r}) + Z_{i-2} \Leftrightarrow$$
$$Z_i \cdot b^2 + Z_{i-1} \cdot b + Z_{i-2} = \hat{q} \cdot V_{n-1} \cdot b + \hat{r} \cdot b + Z_{i-2}$$

нерівність

$$\hat{q} \cdot (V_{n-1} \cdot b + V_{i-2}) > Z_i \cdot b^2 + Z_{i-1} \cdot b + Z_{i-2}$$

зводиться до перевірки нерівності

$$\hat{q} \cdot V_{n-2} > \hat{r} b + Z_{i-2}$$

Тому рядочки 8, 9 алгоритму ділення можуть бути замінені наступними:

```
r := (Z_i · b + Z_{i-1}) mod V_{n-1}
do {
        if q = b or q · V_{n-2} > r · b + Z_{i-2} then
                q := q - 1
                r := r + V_{n-1}
} while r < b
```

Ця нотація зручна для реалізації мовами як низького, так і високого рівня.

Зведення в ступінь.

Основною операцією криптосистем відкритого ключа, таких як RSA, DH, DSA є модулярне зведення в ступінь. Суть цієї операції – серія множень за модулем з наступним застосуванням бінарних (дихотомічних) або m-арних алгоритмів. Одним з найбільш ефективних методів реалізації даної операції є застосування методу множення Монтгомері. Його використовують для прискорення множень за модулем, необхідних в процесі зведення в ступінь.

Метод Монтгомері обчислює:

$\text{MontPro}(a,b) = a \cdot b \cdot r^{-1} \bmod n$,

де $a,b < n$ і r такі, що НСД$(n,r)=1$. Найбільш ефективний даний метод тоді, коли r є степенем двійки. Це обумовлено тим, що в алгоритмі використовується ділення на степінь двійки, що для ряду мікропроцесорів є дуже швидкою операцією.

Нині існує декілька підходів до реалізації методу Монтгомері, в тому числі метод CIOS, який є найбільш оптимальним для ряду поширених процесорів.

Розглянемо множення Монтгомері.

Нехай модуль n – число, розміром k біт: $2^{k-1} n < 2^k$ і нехай $r = 2^k$. Алгоритм множення Монтгомері припускає взаємну простоту r і n: НСД(n,r)=НСД$(2^k, n) = 1$. Ця вимога задовольняється при непарному n.

Визначимо лишок числа a за модулем n: $a<n$, як $\overline{a} = a \cdot r (\bmod r)$. Легко довести, що множина $\{a \cdot r \bmod n \mid 0 \le a \ge n-1\}$ є приведеною системою лишків, тобто містить всі числа від 0 до $n-1$. Також існує взаємнооднозначне співвідношення між числами з інтервалу від 0 до $n-1$ і елементами цієї множини. Арифметика Монтгомері використовує цю властивість через більш швидку процедуру обчислення лишку за модулем n добутку двох чисел, також представлених у вигляді лишків. Нехай дані лишки за модулем n є $\overline{a}$ і $\overline{b}$. Добуток Монтгомері обчислює лишки за модулем n, або n-лишки: $\overline{c} = \overline{a} \cdot \overline{b} \cdot r^{-1} (\bmod n)$, де r^{-1}, число, зворотне до r за модулем n: $r^{-1} \cdot r = 1 (\bmod n)$. Число $\overline{c}$ є n-лишком добутку $c = a \cdot b (\bmod n)$

$$\overline{c} = \overline{a} \cdot \overline{b} \cdot r^{-1} (\bmod n),$$
$$= a \cdot r \cdot b \cdot r \cdot r^{-1} (\bmod n),$$
$$= c \cdot r (\bmod n).$$

Для визначення алгоритму Монтгомері необхідна додаткова величина n', яка є цілим числом, з такою властивістю: $r \cdot r^{-1} - n \cdot n' = 1$ (співвідношення Безу). Числа r^{-1} і n' – цілі і можуть бути обчислені за допомогою розширеного алгоритму Евкліда. Обчислення MontPro $(\overline{a},\overline{b})$ відбувається таким чином:

function MontPro $(\overline{a}, \overline{b})$

1: $t := \overline{a} \cdot \overline{b}$

2: $u := (t + (t + n' \bmod r) \cdot n)/r$

3: if $u \geq n$ then return u - n else return u.

Множення за модулем r, також як і ділення, є виключно швидкими операціями, бо r – ступінь двійки. Тому добуток Монтгомері потенційно швидший і простіший, ніж звичайне (класичне) обчислення $a \cdot b \bmod n$, яке припускає ділення на n. Однак, в силу того, що перетворення в n-лишок (обчислення n'), а також зворотне перетворення потребують певних, відносно великих затрат часу, то метод Монтгомері не використовують для одиничних обчислень модулярних добутків. Цей метод застосовують у випадку декількох добутків за одним і тим самим модулем. Це випадок, коли необхідно обчислити модулярне зведення в ступінь. Використовуючи бінарний (або дихотомічний) метод для обчислення степенів, можна замінити операцію зведення в ступінь серією множень і зведення в квадрат за модулем n.

Зведення в ступінь по Монтгомері.

Нехай j кількість біт експоненти e. Алгоритм зведення в ступінь обчислює $x:=a^e \bmod n$, використовуючи $O(j)$ викликів процедури множення Монтгомері. Крок 3 алгоритму обчислює x, використовуючи $\overline{x}$ через властивості алгоритму Монтгомері, MontPro $(\overline{x},1)=\overline{x} \cdot 1 \cdot r^{-1} = x \cdot r \cdot r^{-1} = x \bmod n$.

function MontExp (a, e, n)

1: $\overline{a} = a \cdot r \bmod n$

2: $\overline{x} = 1 \cdot r \bmod n$

3: for $i = j - 1$ down to 0

$\overline{x} := \text{MontPro}(\overline{x}, \overline{x})$

if $e_i = 1$ then $\overline{x} := \text{MontPro}(\overline{x}, \overline{a})$

4: return $x := \text{MontPro}(\overline{x}, 1)$

або інший варіант:

function MontExp (a, e, n)

1: $\overline{m} = a \cdot r \bmod n$

2: $\overline{x} = 1 \cdot r \bmod n$

3: for $i = 0$ to $j - 1$

if $e_i = 1$ then $\overline{x} := \text{MontPro}(\overline{x}, \overline{m})$

$\overline{m} := \text{MontPro}(\overline{m}, \overline{m})$

4: return $x := \text{MontPro}(\overline{x}, 1)$

Для реалізації операції на великих числах, необхідно розбити числа на слова. Якщо w комп'ютерне слово, тоді числа можуть бути представлені у вигляді послідовності цілих чисел, кожне з яких представлено в системі числення за основою $W = 2^w$. Тоді, якщо велике число розбито на s слів, то r береться, як $r = 2^{sw}$.

Існують такі реалізації множення Монтгомері: SOS, CIOS, FIOS, FIPS, CIHS. Найцікавішими є SOS і CIOS. Обидва методи розрізняються як за технологією, так і за швидкістю виконання, але в той же час зберігають можливість простоти реалізації. Метод CIOS (Coarcely Integrated Operand Scanning) покращує метод SOS через поєднання процедур множення і зведення. Замість того, щоб обчислювати всі добутки $a \cdot b$, а потім зводити, можна переміщуватись між ітераціями зовнішніх циклів процедур множення і зведення. Це можливо внаслідок того, що значення m на i-той ітерації зовнішнього циклу процедури зведення залежить лише від значення $t[i]$, що в свою чергу повністю обчислюється на i ітерації зовнішнього циклу процедури множення. Це приводить до такого алгоритму:

```
for i = 0 to s - 1
        C := 0
        for j = 0 to s - 1
                (C, S) := t[j] + a[j] * b[i] + C
                t[j] := S
        (C, S) := t[s] + C
        t[s] := S
        t[s + 1] := C
        C := 0
        m := t[0] * n′[0] mod W
        for j = 0 to s - 1
                (C, S) := t[j] + m * n[j] + C
                t[j] := S
        (C, S) := t[s] + C
        t[s] := S
        t[s + 1] := t[s + 1] + C
        for j = 0 to s
                t[j] := t[j + 1],
```

де масив t первісно дорівнює нулю. Останній j-цикл використовується для зсуву результату на одне слово вправо (тобто ділення на 2^w). Невелика оптимізація і сполучення зсуву з процедурою зведення:

```
m := t[0] * n′[0] mod W
(C, S) := t[0] + m * n[0]
for j = 1 to s - 1
        (C, S) := t[j] + m * n[j] + C
        t[j - 1] := S
(C, S) := t[s] + C
t[s - 1] := S
t[s] := t[s + 1] + C.
```

Масив t для проміжних даних використовує тільки $s + 2$ слів. Метод CIOS з невеликою оптимізацією вимагає $2s^2 + s$ множень, $4s^2 + 4s + 2$ додавань, $6s^2 + 7s + 2$ читань, $2s^2 + 5s + 1$ записів і використовує пам'ять в розмірі $s + 3$ слів.

Висновки

В результаті практичної реалізації запропонованих в даній статті підходів виявилося, що найбільш швидкою є реалізація криптобібліотеки, отриманої при використанні асемблерного ядра і алгоритму модулярного множення Монтгомері. Тому сучасним розробникам криптосистем, які вбудовуються в застосування реального часу, необхідно звертатись до асемблерної реалізації ресурсоємних обчислень, і переформулювати криптографічні алгоритми, беручи до уваги архітектурні особливості конкретної платформи. Крім того, на підтримку з'явились потужні сучасні інструментальні засоби розробки програм на асемблері, такі як MASM32, AsmStudio, NASM, які за своїми можливостями наближуються до високорівневих засобів проектування застосувань.

Подальші дослідження можуть бути направлені на продовження оптимізації швидкості алгоритмів, які є основою створення та модернізації криптобібліотек, що використовуються в програмних системах реального часу з великою кількістю криптоопераций.

Література: ***1.*** *Б. Страуструп. Язык программирования Си++. — 3-е изд. — спб., М.: «Невский діалект», издательство «Бином», 1999. — 991 с.* ***2.*** *Б. Э. Смит, М. Т. Джонсон. Архитектура и программирование микропроцессора INTEL 80386. — М.: «Конкорд», 1992. — 334 с.* ***3.*** *Сингер М. Мини-ЭВМ PDP-11: Программирование на языке ассемблера и организация машины. — М.: «Мир», 1984. — 272 с.* ***4.*** *Дональд Кнут. Искусство программирования, том 1. Основные алгоритмы = The Art of Computer Programming, vol. 1. Fundamental Algorithms. — 3-е изд. — М.: «Вильямс», 2006. — С. 720.* ***5.*** *A. Menezes, P. van Oorschot, S. Vanstone. Handbook of Applied Cryptography. — CRC-Press, 1996. — 816 p. — (Discrete Mathematics and Its Applications).* ***6.*** *Венбо Мао. Современная криптография: теория и практика = Modern Cryptography: Theory and Practice. — М.: «Вильямс», 2005. — 768 с.* ***7.*** *Нильс Фергюсон, Брюс Шнайер. Практическая криптография = Practical Cryptography: Designing and Implementing Secure Cryptographic Systems. — М.: «Диалектика», 2004. — 432 с.* ***8.*** *Шнайер Б. Прикладная криптография. Протоколы, алгоритмы, исходные тексты на языке Си = Applied Cryptography. Protocols, Algorithms and Source Code in C. — М.: Триумф, 2002. — 816 с.*

УДК 681.325.59:519.6

АВТОКОРИГУЮЧІ ВЛАСТИВОСТІ ЕЛЕМЕНТАРНИХ АВТОМАТІВ

Ярослав Клятченко, Оксана Тарасенко-Клятченко
НТУУ «КПІ»

Анотація: **Розроблена методика і запропонована система показників для дослідження та оцінки ефекту від врахування автокоригуючих властивостей функцій переходів елементарних автоматів.**
Summary: **Developed methods and proposed a system of indexes for research and evaluation the self-correction effect for transition function of elementary automaton.**
Ключові слова: **Автокорекція, спотворення сигналу.**

І Постановка задачі

Латентні автокоригуючі властивості довільних перемикальних функцій, що полягають у формуванні правильних їх значень за наявності спотворень (техногенних або умисних) їх аргументів, досить повно досліджені в роботах [1, 2]. Врахування цих властивостей забезпечує більш високу ймовірність отримання правильного результату деякого логічного перетворення, що описується заданими перемикальними функціями, ніж це могло бути обумовлено тільки ймовірностями появи неспотворених значень аргументів. Відомо також [3], що дуже багато структур цифрових пристроїв можна подати як композицію логічного перетворювача та технічної пам'яті на основі елементарних автоматів (ЕА) – тригерів різних типів. При цьому ЕА є автоматами другого роду (автоматами Мура) з двома стійкими станами [4], для яких функція виходів однозначно визначається станом автомата. Тому в умовах потенційної загрози вхідних спотворень стосовно ЕА можуть виникати наступні ситуації:

- спотворення вхідних сигналів нема і ЕА перемикається правильно (тобто, відповідно до його функції переходів (ФП));
- спотворення вхідних сигналів є і ЕА перемикається неправильно (переходить не в той стан, який задається його ФП);
- спотворення вхідних сигналів є, але ЕА до нього нечутливий (наприклад, ЕА типу *RS* не перемикається і зберігає свій попередній стан, коли на його входах два нулі);
- спотворення вхідних сигналів є, але ЕА переходить в той же стан, який задається неспотвореними вхідними сигналами (наприклад, ЕА типу *R* переходить в нуль під дією вхідних сигналів як *R=1, S=0*, так і *R=1, S=1*).

Всі відомі методи оцінки достовірності (вірогідності) функціонування цифрових пристроїв в умовах потенційної загрози вхідних спотворень [5] орієнтовані тільки на ситуації 1) і 2). Це було спричинене тим, що безумовно приймалася гіпотеза про дуже малу ймовірність техногенного спотворення вхідних сигналів, а можливість їх умисного спотворення взагалі не допускалась. Однак відомо [6], що нормативні