Математички факултет,
универзитет у Београду

Мастер рад

Разбијање алгоритма *DES* грубом силом коришћењем акцелератора *Maxeler*

Јован Радосављевић

ментор
проф. др. Миодраг Живковић

2. октобар 2016.

Садржај

1 Увод

DES је један од шифарских система који је имао највише утицаја на развој модерне криптографије. Широко је коришћен поред осталог за невојну комуникацију владиних органа у САД, као и у комерцијалне сврхе. Замењен је новим стандардом AES, јер је због дужине кључа од свега 56 бита и због напретка технологије цена напада грубом силом (испробавање свих могућих кључева) постала мала. Прецизније, алгоритам *DES* се и даље користи, али у варијанти са удвострученим кључем.

DES Cracker је машина којом је применом грубе силе алгоритам *DES* разбијен за 22 сата и 15 минута. Машина је конструисана као кандидат за такмичење "*DES Challenge III*" на коме је освојила прво место [5, 6].

Maxeler Technologies је једна од водећих компанија која пружа решења за програмирање на нивоу хардвера (*FPGA*). Такав начин програмирања омогућује паралелизацију, односно велико убрзавање израчунавања, уз битно смањен утрошак електричне енергије. *Maxeler* је развио разноврсне архитектуре које се користе у зависности од потреба корисника.

Циљ рада је практична реализација напада са познатим паром (отворени текст, шифрат) на систем *DES* алгоритмом грубе силе, причему се за паралелизацију користи персонални рачунар са картицом Maxeler. За развој програма коришћена окружење MaxIDE са преводиоцем *MaxCompiler*, који на основу кода написаног на високом језику генерише хардверску имплементацију.

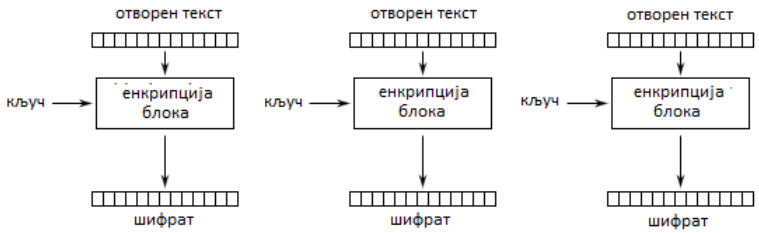
2 Алгоритам *DES*

DES (Data Encryption Standard) је алгоритам за шифровање који је настао седамдесетих година двадесетог века у *IBM*-у када је институт *NIST* (енг. National Institute of Standards and Technology) отворио конкурс са циљем да добију алгоритам за шифровање који ће бити сигуран, доступан свима, једноставан и јевтин за имплементацију. Почетна верзија алоритма коју је *IBM* предложио звала се Луцифер (енг. Lucifer) и имала је кључ величине 112 битова. После детаљне анализе које су спровели у унституту *NIST* одлучили су да направе две измене алгоритма у договору са *IBM*-ом. Прва је била да се промене субституционих табеле које се користе за пермутацију битова у току алкоритма, а друга је била да се кључ смањи са 112 на 56 битова. Иако су у институту NIST тражили да алгоритам буде што сигурнији да би могао да се користи чак и у влади Сједињених држава одлучили су се за овај корак из страха да би остале владе могле да искористе овај алгоритам против њих. *DES* је постао стандард 23. новембра 1976. године. Иако стар, овај алгоритам успешно одолева криптоанализи до данас [1].

2.1 Подела отвореног текста на блокове

Приликом шифровања великог отвореног текста (поруке са пуно карктера), потребно је поделити отворен текст на више блокова. У случају алгоритма *DES* отворени текст се дели у блокови величине од по 64 бита. Постоји више начина шифровања (енг. mode of operation) отвореног текста користећи подељене блокове: *ECB* (Electronic Code Book), *CBC* (Cipher Block Chaining), *PCBC* (Propagating Cipher Block Chaining), *CFB* (Cipher Feedback), *OFB* (Output Feedback) і *CTR* (Counter). Сваки од наведених начина за шифровање блокова отвореног текста је симетричан, што значи да се кључеви који служе за шифровање отвореног текста користе и за дешифровање шифрата [2].

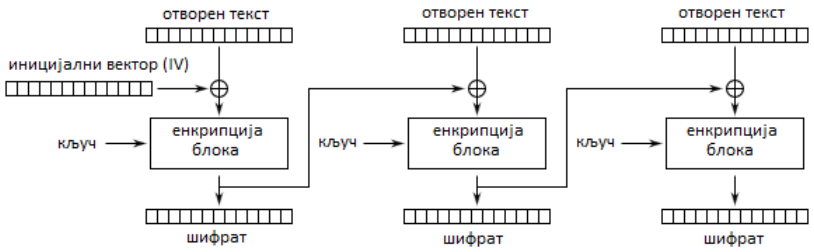
ECB (Electronic Code Book) (слика 1) је један од начина шифровања подељеног отвореног текста на блокове тако што се сваки блок шифрује засебно. Овај начин шифровања блокова отвореног текста се хардверски лако имплементира због тога што се сваки блок отвореног текста шифрује засебно. Међутим због особине да се сваки блок отвореног текста шифрује засебно није препоручљиво користити овај начин јер постоји могућност да се грубом силом паралелно нападну подељени блокови [2].



Слика 1: Шифровање подељених блокова коришћењем *ECB*

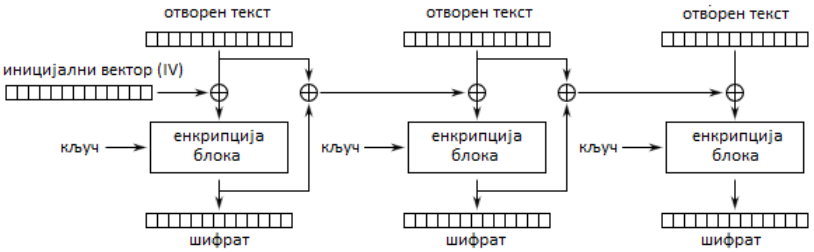
CBC (Cipher Block Chaining) (слика 2) је сигурнији начин шифровања подељених блокова отвореног текста за разлику од претходно наведеног начина за шифровање

блокова отвореног текста *ECB*. *CBC* за шифровање блока отвореног текста користи шифрат добијен од претходног блока отвореног текста, док се први блок отвореног текста шифрује коришћењем иницијалног вектора (блока) јер претходни блок не постоји. Хардверска имплементација оваквог начина шифровања блокова отвореног текста је тежа у односу на претходно описани начин *ECB* јер се шифрат блока не може израчунати док није познат шифрат претходног блока [2, 3].



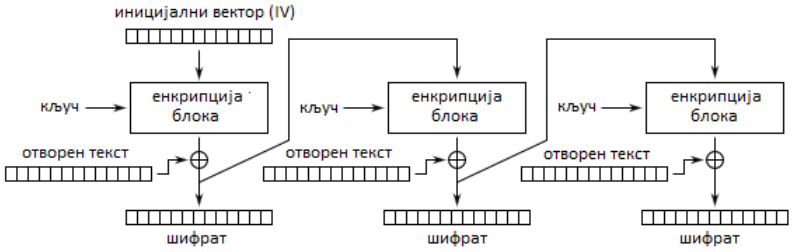
Слика 2: Шифровање подељених блокова коришћењем *CBC*

PCBC (Propagating Cipher Block Chaining) (слика 3) је начин шифровања подељених блокова отвореног текста у основи сличан са претходно наведеним начином *CBC*. Једина разлика између *PCBC* и *CBC* је то што се за шифровање блока отвореног текста у *PCBC* поред шифрата добијеног од претходног блока отвореног текста користи и сам блок претходног отвореног текста. *PCBC* још увек није стандардизован иако је коришћен у алгоритму Керберос 4 (енг. Kerberos 4) [2].



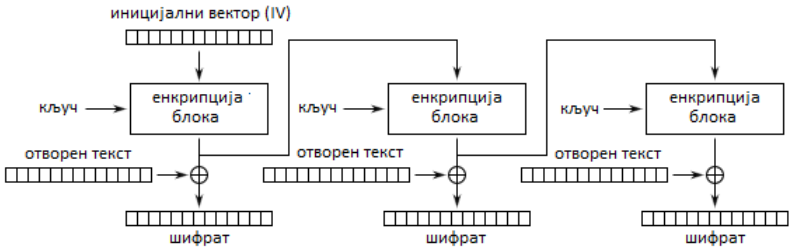
Слика 3: Шифровање подељених блокова коришћењем *PCBC*

CFB (Cipher Feedback) (слика 4) је начин за шифровање подељених блокова отвореног текста који користи псеудо-случајни генератор приликом шифровања. Почетни корак у начину *CFB* је да се шифрује псеудо-случајни вектор на који се потом, да би се добио шифрат, користи оператор ексклузивно или са првим блоком отвореног текста. остали шифрати блокова отвореног текста се добијају коришћењем шифрата претходног блока отвореног текста [2].



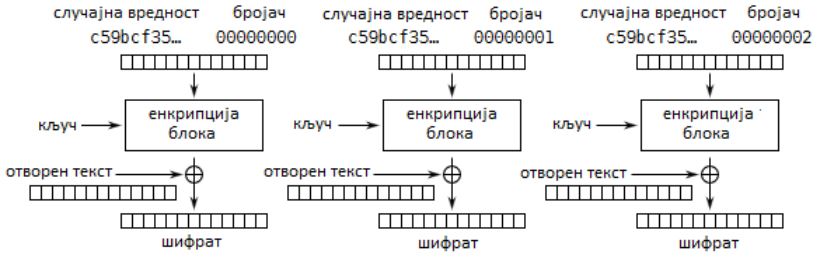
Слика 4: Шифровање подељених блокова коришћењем *CFB*

OFB (Output Feedback) (слика 5) је начин шифровања блокова отвореног текста који је сличан претходно наведеном начину *CFB*. Разлика је у томе што се за шифровање блока не користи шифрат претходног блока, већ вредност која је била пре извршавања оператора ексклузивно или са блоком отвореног текста. Што значи да зависиност шифровања блока отвореног ткста и претходног блока отвореног текста не постоји. Због претходно наведене особине *OFB* је рањив на нападе [2].



Слика 5: Шифровање подељених блокова коришћењем *OFB*

CTR (Counter) (слика 6) је начин шифровања блокова отвореног текста који је врло сличан начину за шифровање блокова *CFB*. За разлику од *CFB*, *CTR* не користи шифрат претходног блока отвореног текста за рачунање наредног већ користи бројач блокова [2].



Слика 6: Шифровање подељених блокова коришћењем *CTR*

2.2 Опис алгоритма *DES*

DES је симетрични блоковски алгоритам који шифрује блокове величине 64 бита помоћу кључа величине 64 бита (56 бита која се користе као кључ и 8 бита која се користе за проверу). Састоји се од 16 рунди (циклуса) које користе комбинације аритметичких и логичких операција [1].

2.2.1 Генерисање кључева

Алгоритам *DES* користи симетрични кључ K од 64 бита од кога се генеришу 16 кључева од по 48 бита који се користе за шифровањње и дешифровање. Сваки од 16 кључева се користи у посебној рунди [11].

$$K = (e_1, e_2, ..., e_{64})$$

Кључ K_0 се добија тако што се на кључ K примени пермутација $PC-1$

$PC-1$														
i	1	2	3	4	5	6	7	8	9	10	11	12	13	14
$PC1(i)$	57	49	41	33	25	17	9	1	58	50	42	34	26	18

$PC-1$														
i	15	16	17	18	19	20	21	22	23	24	25	26	27	28
$PC1(i)$	10	2	59	51	43	35	27	19	11	3	60	52	44	36

$PC-1$														
i	29	30	31	32	33	34	35	36	37	38	39	40	41	42
$PC1(i)$	63	55	47	39	31	23	15	7	62	54	46	38	30	22

$PC-1$														
i	43	44	45	46	47	48	49	50	51	52	53	54	55	56
$PC1(i)$	14	6	61	53	45	37	29	21	13	5	28	20	12	4

$$K_0 = PC1(K) = (e_{57}, e_{49}, ..., e_4)$$

Запажа се да су из пермутације $PC-1$ изостављени битови 8, 16, ..., 64, то је зато што се ови битови користе само као контрола парности. Због тога што су изостављени ови битови кључ K_0 садржи 56 бита.

Кључ K_0 се дели на леву и десну половину од по 28 бита C_0 и D_0.

$$C_0 = (e_{57}, e_{49}, ..., e_{36})$$

$$D_0 = (e_{63}, e_{55}, ..., e_4)$$

На основу добијених C_0 и D_0 рачуна се C_n и D_n за $1 \leq n \leq 16$ тако што се сваки пар C_n и D_n рачуна користећи померањем у лево претходног пара (C_{n-1} и D_{n-1}) за одређени број битова дат у табели BI.

BI								
итерација i	1	2	3	4	5	6	7	8
број помераја у лево $BI(i)$	1	1	2	2	2	2	2	2

BI								
итерација i	9	10	11	12	13	14	15	16
број помераја у лево $BI(i)$	1	2	2	2	2	2	2	1

$$C_1 = C_0 \ll BI(1) = (e_{57}, e_{49}, ..., e_{36}) \ll 1 = (e_{49}, e_{41}, ..., e_{57})$$
$$...$$
$$C_{16} = C_{15} \ll BI(16)$$

$$D_1 = D_0 \ll BI(1) = (e_{63}, e_{55}, ..., e_4) \ll 1 = (e_{55}, e_{47}, ..., e_{63})$$
$$...$$
$$D_{16} = D_{15} \ll BI(16)$$

Потом се од сваког пара C_n и D_n за $1 \leq n \leq 16$ пермутацијом $PC-2$ добија кључуч K_n

$PC-2$												
i	1	2	3	4	5	6	7	8	9	10	11	12
$PC2(i)$	14	17	11	24	1	5	3	28	15	6	21	10

$PC-2$												
i	13	14	15	16	17	18	19	20	21	22	23	24
$PC2(i)$	23	19	12	4	26	8	16	7	27	20	13	2

$PC-2$												
i	25	26	27	28	29	30	31	32	33	34	35	36
$PC2(i)$	41	52	31	37	47	55	30	40	51	45	33	48

$PC-2$												
i	37	38	39	40	41	42	43	44	45	46	47	48
$PC2(i)$	44	49	39	56	34	53	46	42	50	36	29	32

$$K_1 = PC2(C_1D_1) = PC2((e_{49}, e_{41}, ..., e_{57})(e_{55}, e_{47}, ..., e_{63})) =$$
$$PC2((e_{16}, e_{24}, ..., e_8, e_{12}, e_{20}, ..., e_4)) = (e10, e51, ..., e31)$$
$$...$$
$$K_{16} = PC2(C_{16} + D_{16})$$

Пермутација $PC-2$ прави селекцију 48 бита од почетних 56.

2.2.2 Шифровање поруке

Нека је M порука (отворен текст који треба шифровати) од које пропуштањем кроз алгоритам *DES* треба да се добије шифрат.

$$M = (e_1, e_2, ..., e_{64})$$

На поруку M се примењује иницијална пермутација IP.

IP																
i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
$IP(i)$	58	50	42	34	26	18	10	2	60	52	44	36	28	20	12	4

IP																
i	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32
$IP(i)$	62	54	46	38	30	22	14	6	64	56	48	40	32	24	16	8

IP																
i	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48
$IP(i)$	57	49	41	33	25	17	9	1	59	51	43	35	27	19	11	3

IP																
i	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	64
$IP(i)$	61	53	45	37	29	21	13	5	63	55	47	39	31	23	15	7

$$IP(M) = (m_{58}, m_{50}, ..., m_7)$$

Блок битова IP се дели на леву и десну половинну од по 32 бита (L_0 и R_0). За сваку од 16 руди се добија по један пар L_n, R_n.

$$L_0 = (m_{58}, m_{50}, ..., m_8)$$

$$R_0 = (m_{57}, m_{49}, ..., m_7)$$

$$L_n = R_{n-1}, 1 \leq n \leq 16$$

$$R_n = L_{n-1} + f(R_{n-1}, K_n), 1 \leq n \leq 16$$

где је + сабирање битова по модулу 2 блокова једнаких дужина (битовски оператор ексклузивно или), K_n представља кључ за рунду n чије је детаљно израчунавање објашњено у подпоглављу 2.1. Израчунаванје функције f биће детаљно представљено следеће.

Израчунавање функције *f*: Улазни параметри функције f су R_{n-1} дужине 32 бита и K_n дужине 48 бита, а излаз је блок од 32 бита. R_{n-1} проширујемо пермутацием E на блок од 48 бита.

E												
i	1	2	3	4	5	6	7	8	9	10	11	12
$E(i)$	32	1	2	3	4	5	4	5	6	7	8	9

E												
i	13	14	15	16	17	18	19	20	21	22	23	24
$E(i)$	8	9	10	11	12	13	12	13	14	15	16	17

E												
i	25	26	27	28	29	30	31	32	33	34	35	36
$E(i)$	16	17	18	19	20	21	20	21	22	23	24	25

E												
i	37	38	39	40	41	42	43	44	45	46	47	48
$E(i)$	24	25	26	27	28	29	28	29	30	31	32	1

$$E_0 = E(R_0)$$
$$...$$
$$E_{16} = E(R_{16})$$

На E_{n-1} и K_n се примењује сабирање битова по модулу 2. Резултат се дели на 8 дела по 6 бита B_1, B_2, ... B_8.

$$E_{n-1} + K_n = (B_{n,1}B_{n,2}B_{n,3}B_{n,4}B_{n,5}B_{n,6}B_{n,7}B_{n,8}), 1 \leq n \leq 16$$

Сваки $B_{n,i}$, $1 \leq i \leq 16$ се мења вредношћу из табеле Si тако да први и задњи бит одређују врсту, а четири унутрашњих битова одређују колону.

$S1$		$\_xxxx\_$															
		0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
$x\_\_\_\_x$	0	14	4	13	1	2	15	11	8	3	10	6	12	5	9	0	7
	1	0	15	7	4	14	2	13	1	10	6	12	11	9	5	3	8
	2	4	1	14	8	13	6	2	11	15	12	9	7	3	10	5	0
	3	15	12	8	2	4	9	1	7	5	11	3	14	10	0	6	13

$S2$		$\_xxxx\_$															
		0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
$x\_\_\_\_x$	0	15	1	8	14	6	11	3	4	9	7	2	13	12	0	5	10
	1	3	13	4	7	15	2	8	14	12	0	1	10	6	9	11	5
	2	0	14	7	11	10	4	13	1	5	8	12	6	9	3	2	15
	3	13	8	10	1	3	15	4	2	11	6	7	12	0	5	14	9

$S3$		$\_xxxx\_$															
		0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
$x\_\_\_\_x$	0	10	0	9	14	6	3	15	5	1	13	12	7	11	4	2	8
	1	13	7	0	9	3	4	6	10	2	8	5	14	12	11	15	1
	2	13	6	4	9	8	15	3	0	11	1	2	12	5	10	14	7
	3	1	10	13	0	6	9	8	7	4	15	14	3	11	5	2	12

$S4$		$\_xxxx\_$															
		0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
$x\_\_\_\_x$	0	7	13	14	3	0	6	9	10	1	2	8	5	11	12	4	15
	1	13	8	11	5	6	15	0	3	4	7	2	12	1	10	14	9
	2	10	6	9	0	12	11	7	13	15	1	3	14	5	2	8	4
	3	3	15	0	6	10	1	13	8	9	4	5	11	12	7	2	14

$S5$		$\_xxxx\_$															
		0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
$x\_\_\_\_x$	0	2	12	4	1	7	10	11	6	8	5	3	15	13	0	14	9
	1	14	11	2	12	4	7	13	1	5	0	15	10	3	9	8	6
	2	4	2	1	11	10	13	7	8	15	9	12	5	6	3	0	14
	3	11	8	12	7	1	14	2	13	6	15	0	9	10	4	5	3

$S6$		$\_xxxx\_$															
		0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
$x\_\_\_\_x$	0	12	1	10	15	9	2	6	8	0	13	3	4	14	7	5	11
	1	10	15	4	2	7	12	9	5	6	1	13	14	0	11	3	8
	2	9	14	15	5	2	8	12	3	7	0	4	10	1	13	11	6
	3	4	3	2	12	9	5	15	10	11	14	1	7	6	0	8	13

$S7$		$\_xxxx\_$															
		0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
$x\_\_\_\_x$	0	4	11	2	14	15	0	8	13	3	12	9	7	5	10	6	1
	1	13	0	11	7	4	9	1	10	14	3	5	12	2	15	8	6
	2	1	4	11	13	12	3	7	14	10	15	6	8	0	5	9	2
	3	6	11	13	8	1	4	10	7	9	5	0	15	14	2	3	12

$S8$		$\_xxxx\_$															
		0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
$x\_\_\_\_x$	0	13	2	8	4	6	15	11	1	10	9	3	14	5	0	12	7
	1	1	15	13	8	10	3	7	4	12	5	6	11	0	14	9	2
	2	7	11	4	1	9	12	14	2	0	6	10	13	15	3	5	8
	3	2	1	14	7	4	10	8	13	15	12	9	0	3	5	6	11

$$S_n = (S(B_{n,1})S(B_{n,2})S(B_{n,3})S(B_{n,4})S(B_{n,5})S(B_{n,6})S(B_{n,7})S(B_{n,8})), 1 \leq n \leq 16$$

На S_n се примењује пермутација P.

P								
i	1	2	3	4	5	6	7	8
$P(i)$	16	7	20	21	29	12	28	17

P								
i	9	10	11	12	13	14	15	16
$P(i)$	16	7	20	21	29	12	28	17

P								
i	17	18	19	20	21	22	23	24
$P(i)$	2	8	24	14	32	27	3	9

P								
i	25	26	27	28	29	30	31	32
$P(i)$	19	13	30	6	22	11	4	25

$$f(R_{n-1}, K_n) = P_n(S_n), 1 \leq n \leq 16$$

Да би се добио шифрат C на конкатенацију битова R_{16} и L_{16} се примењује инверзна пермутација IP^{-1}.

IP^{-1}																
i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
$IP^{-1}(i)$	40	8	48	16	56	24	64	32	39	7	47	15	55	23	63	31

IP^{-1}																
i	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32
$IP^{-1}(i)$	38	6	46	14	54	22	62	30	37	5	45	13	53	21	61	29

IP^{-1}																
i	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48
$IP^{-1}(i)$	36	4	44	12	52	20	60	28	35	3	43	11	51	19	59	27

IP^{-1}																
i	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	64
$IP^{-1}(i)$	34	2	42	10	50	18	58	26	33	1	41	9	49	17	57	25

$$C = IP^{-1}(R_n L_n), 1 \leq n \leq 16$$

2.2.3 Псеудо код алгоритма *DES*

Алгоритам DES:
Улаз: порука m, кључ k
Излаз: шифрат c
slice(i,j) враћа j битова почевши од i-тог бита

```
for i:=0 to 55 do
  _k[0][i] := k[PC1[i]];
_c[0] := _k0.slice(0,28);
_d[0] := _k0.slice(38,58);
for i:=1 to 16 do
  _c[i] := _c[i-1]<<BI[i];
  _d[i] := _d[i-1];
for i:=1 to 16 do
  for j:=0 to 47 do
    _k[i][j] = (_c[i]+_d[i])[PC2[j]];

for i:=1 to 64 do
  _ip[i] := m[IP[i]];
_l[0] := _ip.slice(0,32);
_r[0] := _ip.slice(32,32);
for i := 1 to 16 do
  _l[i] := r[i-1];
  for j:=0 to 47 do
    _e[i][j] := _r[n-1][E[j]];
  _f[i] := _k[i] ^ _e[i];
  for j:=0 to 7 do
    _b[i][j] := _f[i].slice(i*8,6);
  for j:=0 to 7 do
    _s := _s +S[j+1][_b[i][j].slice(0,1)
      +_b[i][j].slice(6,1)][_b[i][j].slice(1,4)];
  for j:=0 to 32 do
    _p := _s[i][P[j]]
  _r[i] := _l[i-1] ^ _p;
_rl := _r[16] + _l[16];
for i:=0 to 63 do
  c[i] := _rl[IP1[i]];
return c;
```

3 Напади грубе силе на *DES*

Једини познати начин да се угрози интегритет *DES*-а јесте применом алгоритма грубе силе. Данашњи рачунари су вишеструко бржи од оних који су постојали деведесетих година прошлог века, што значи да у јединици времена може да се шифрује знатно већи број блокова отвореног текста. То отвара могућност да уколико нам је познат пар отворени текст и шифрат, испроабамо све могуће комбинације кључева на одређени блок отвореног текста и тако добијемо кључ који слика дати отворен текст у дати шифрат [4].

Следећа табела приказује број шифрованих блокова разних процесора у једној секунди.

Процесор	Брзина (у MHz)	Број шифрованих блокова у секунди
8088	4,7	370
68000	7,6	900
80286	6	1100
68020	16	3500
68030	16	3900
80386	25	5000
68030	50	10000
68040	25	16000
68040	40	23000
80486	66	43000
Sun ELC		26000
HyperSparc		32000
RS6000-350		53000
parc 10/52		84000
DEC Alpha 4000/610		154000
HP 9000/887	125	196000

3.1 *DES Cracker*

DES Cracker је машина која за познат пар отворен текст и шифрат коришћењем алгоритма грубе силе, испробавањем сваког кључа, налази кључ који је коришћен за шифровање датог отвореног текста тако да се добије дати шифрат. Машину *DES Cracker* је направила организација *EFF* (Electronic Frontier Foundation) направљена 1998. година као кандидат за такмичење "*DES Challenge II*" које је организовала *RSA* лабораторија (RSA Laboratory). Ова машина је лако освојила прву награду јер је успела да за мање од 3 дана пронађе одговарајући кључ за дати пар отворен текст и шифрат (претходни рекорд је био 39 дана који је постигнут користећи мрежу од десет хиљада рачунара). Дизајн машине *DES Cracker* је концептуално једноставан. Представља рачунар са великим бројем специјализованих чипова (1500 чипова) и софтвером који даје инструкције

чиповима и служи као интерфејс ка кориснику. Цео пројекат изградње машине *DES Cracker* је коштао око 210 000 долара од тога 80 000 су коштали дизајн, интеграција и тестирање, а преосталих 130 000 је коштао материјал за изградњу рачунара. Студије које су спровели криптографи 1996. године су као резултат дале процену да би дужина кључа алгоритма *DES* требало да буде бар 90 бита да би алгоритам био сигуран од оваквих типова напада [5, 6].

Побољшана верзија машине *DES Cracker* је конструисана 1999. године као кандидат за конкурс "*DES Challenge III*". Унапређена верзија машине *DES Cracker* је за 22 сата и 15 минута успела да пронађе кључ који одговара познатом датом пару отворен текст и шифрат [5, 6].

3.2 Напад на *DES* сусрет на пола пута

Криптографи су крајем седамдесетих година двадесетог века сматрали да сигурност алгоритма за шифровање *DES* може да се унапреди двоструким извршавањем алгоритма *DES* (двоструки *DES*) са два различита кључа. То би значило да је укупна дужина кључа 2×56 бита тј. 112 бита. За ову дужину кључа постоји $2^{112} \approx 10^{33}$ различитих комбинација кључева. Када би се за сваки покушај кључа утрошила по једна микросекунда за све комбинације кључева било би потребно 10^{22} дана, тако да је покушај да се грубом силом од пара отвореног текста и шифрата добије кључ у потпуности бесмислен [2].

Дипломирани студент електротехничког факултета на Станфорд Универзитету (енг. Electrical Engineering Department of Stanford University) Вајтфилд Дифи (енг. Whitfield Diffie) и асистент електротехничког факултета на Станфорд Универзитету Мартин Хелман (енг. Martin Hellman) су 1977. године објавили рад [10] у коме су описали напад на двоструки *DES* коме је дат назив сусрет на пола пута (енг. *Meet-in-the-Middle*). У овом раду су показали да коришћење двоструког *DES* алгоритма нема никакве предности у односу на обичан *DES* [2, 10].

Сусрет на пола пута је напад који може да се примени на било који блоковски алгоритам за шифровање који користи више кључева за шифровање истим алгоритмом. Напад подразумева да су познати пар отворен текст и шифрат.

Нека су:

$$C = E_b(k_b, E_a(k_a, P)), \tag{1}$$

$$P = D_a(k_a, D_b(k_b, C)), \tag{2}$$

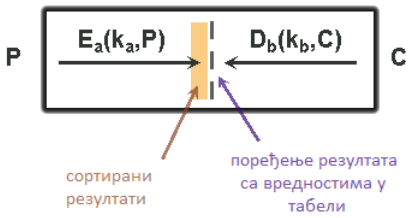
где је C шифрат, P отворен текст, E алгоритам за шифровање и D алгоритам за дешифровање [2].

Из израза (1) и (2) може се закључити да је:

$$D_b(k_b, C) = E_a(k_a, P) \tag{3}$$

Пошто су по претпоставци пар шифрат C и отворен текст P познати, прави се табела са резултатима алгоритма за шифровање за сваку комбинацију кључева за једну

страну једнакости (3). Нека то буде на пример $E_a(k_a, P)$. У табелу се уписују резултати за сваку могућу комбинацију кључева примењеног алгоритма за шифровање на отворен текст P. Табелу треба сортирати по добијеним резултатима да би се касније смањило време претраживања појединачног резултата у табели. Затим се рачунају резултати алгоритма за дешифровање друге стране једнакости (3) тј. $D_b(k_b, C)$ за сваку могућу комбинацију кључева на шифрат C. Резултати добијени на основу леве стране једнакости (3) $D_b(k_b, C)$ се траже у табели са резултатима десне стране једнакости (3) $E_a(k_a, P)$. Тражени кључеви алгоритма који су били непознати су пар кључева који се поклапају по горе наведеном алгоритму [2]. Илустрација напада сусрет на пола пута на блоковске алгоритме је приказана на слици 7.



Слика 7: Илустрација напада сусрет на пола пута

4 Програмирање вођено током података. Макселер акцелератор и развојно окружење

Програмирање вођено током података (енг. Data flow programming) је програмска парадигма која је настала средином двадесетог века. Међутим, рачунари који подржавају ову парадигму су заживели у продаји тек недавно. Програми који су писани коришћењем ове парадигме не користе хардвер да би контролисали ток података већ модификују хардвер тако да улазни подаци само прођу кроз тај хардвер правећи резултат. Време потребно за извршавање програма писаних коришћењем парадигме вођене током података је вишеструко смањено јер омогућава вишеструку паралелизацију [7].

Рачунари који подржавају програмирање вођено током података се обично користе као акцелератори за програме који су писани коришћењем парадигме контролисаног тока података (енг. Control flow) кад је потребно да се паралелизује извршавање унутар петље [7].

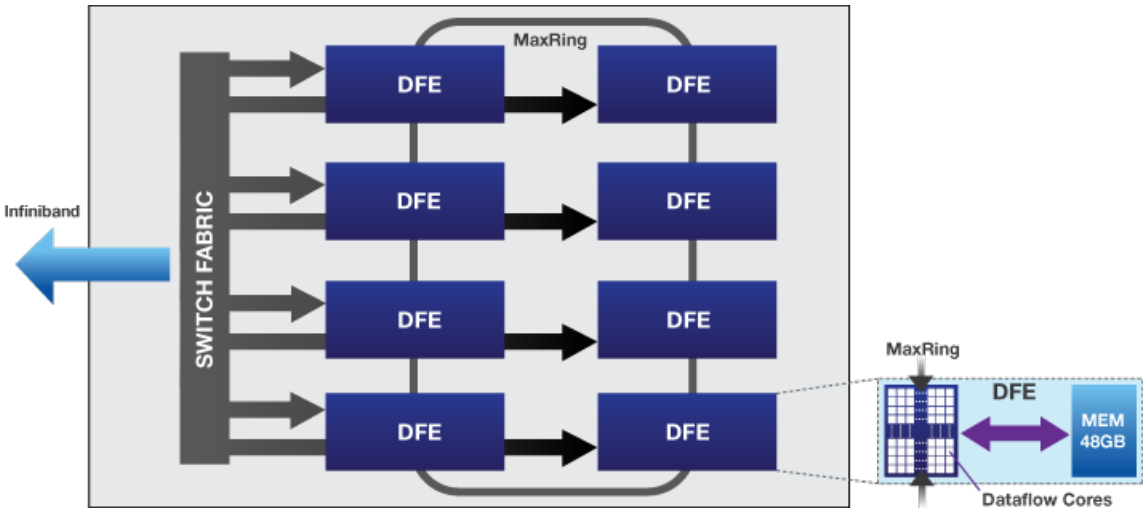
Макселер технологије (енг. Maxeler Tehnologies) је једна од водећих компанија која пружа хардверска и софтверска решења за програмирање вођено током података. Основана је 2003. године и од тада успешно влада тржиштем. За писање програма вођеним током подата на акцелераторима које производи компанија макселер користи се језик који је зове *MaxJ* (Maxeler Java). *MaxJ* је надоградња језика јава (енг. Java) и садржи две врсте променљивих, софтверске и хардверске. Софтверске променљиве су преузете из језика јава и користе се током компилације и након тога нестају, а хардверске променљиве су заправо оне које одређују ток података [9].

4.1 Архитектура Макселер акцелератора

Макселеров акцелератор се састоји из више јединица *DFE* (Dataflow engines). Једна *DFE* јединица саджи своју локалну меморију која повезана са процесором. Постоје две врсте меморија у *DFE*: брза меморија (енг. Fast Memory) која има капацитет неколико мегабајта, али има велику брзину протока података, чак и до неколико терабајта у секунди и велика меморија (енг. Large Memory) која има капацитет величине неколико гигабајта [7, 9].

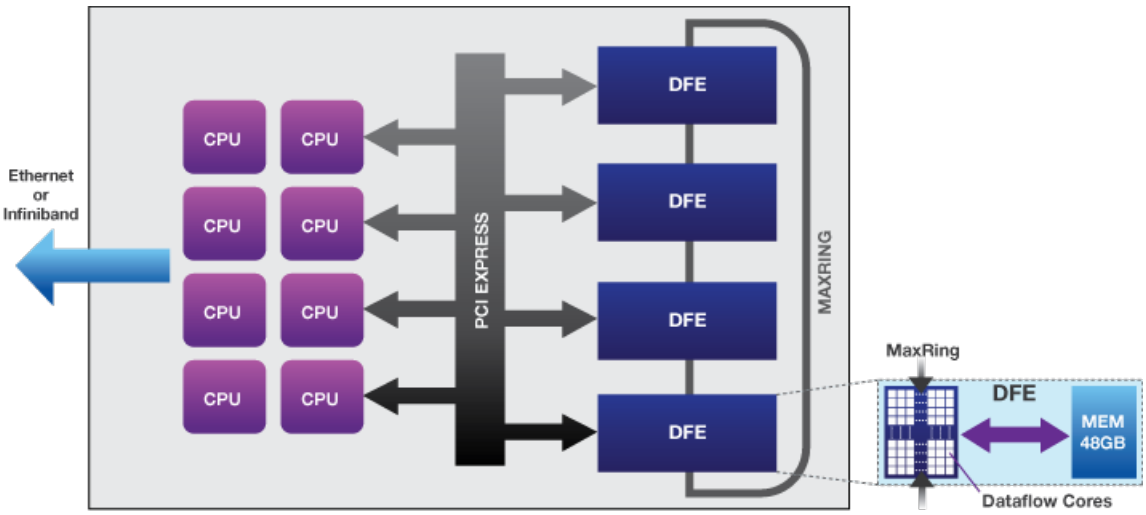
Постоји више типова архитектура Макселер акцелератора које су подељене у серије *MPC-X*, *MPC-C* и *MPC-N* [9].

Серија *MPC-X* (слика 8) представља архитектуру која садржи више *DFE* јединица као дељени ресурс на магистрали. Ова архитектура се због велике брзине и великог капацитета меморије (до 768 гигабајта) користи као сервер за које се планира велики број корисника, сервисе на облаку (енг. Cloud service), *HPC* (High Performance Computing) кластере [9].



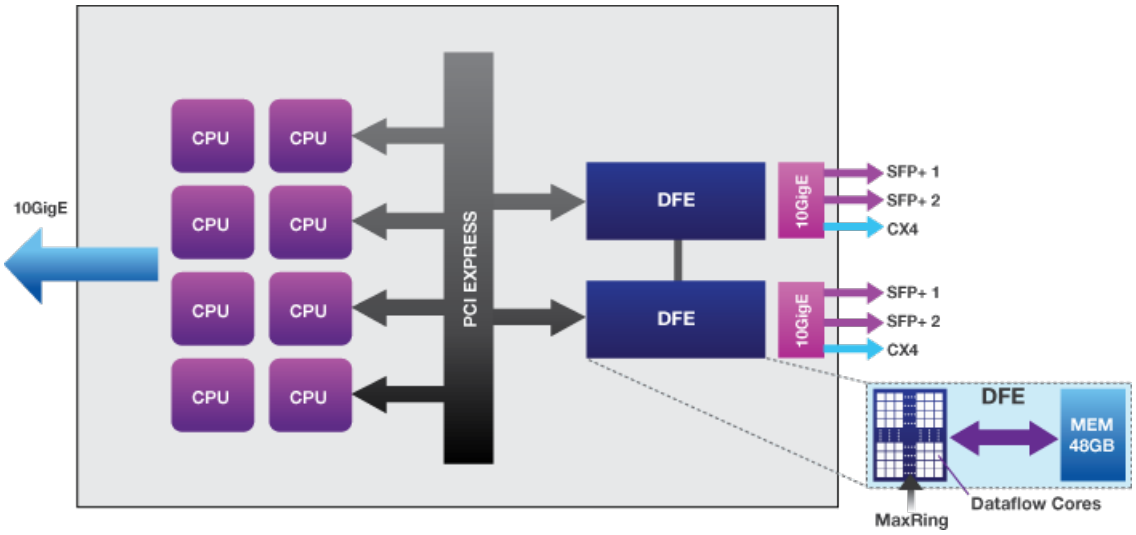
Слика 8: Архитектура Макселер акцелератора серије *MPC-X*

Серија *MPC-C* (слика 9) представља архитектуру која омогућава велику брзину комуникације између *DFE* јединица и процесора. Ова архитектура се користи за развијање сложених алгоритама за које би требало до 50 сервера са просечном јачином процесора уз огромну уштеду електричне енергије и материјалних средстава [9].



Слика 9: Архитектура Макселер акцелератора серије *MPC-C*

Серија *MPC-N* (слика 10) представља архитектуру која има скоро неприметно кашњење у комуникацији *DFE* јединица и процесора. Оваква архитектура омогућава директну везу са јединицама *DFE* кроз коју може да прође велика количина података за кратко време [9].



Слика 10: Архитектура Макселер акцелератора серије *MPC-N*

4.2 Програмабилна дигитално иинтегрисана кола *FPGA*

FPGA (Field programmable gate arrays) (слика 11) је дигитално интегрисано коло које има могућност програмирања након завршене производње, међутим може да се програмира само једанпут (енг. one-time programmable). У односу на остала дигитална кола са могућношћу програмирања кола *FPGA* имају предности зато што се лако програмирају, нису скупа и лако се набављају. Кола *FPGA* су настала 80-их година двадесетог века. Нагли пораст коришћења *FPGA* кола почео је 90-их година са растом употребе комуникационих технологија и интернета [8].

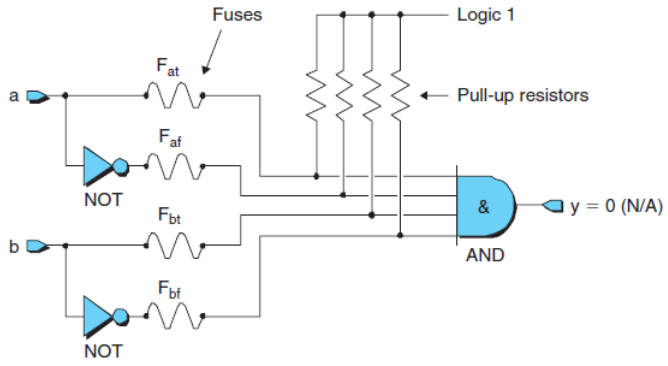
4.2.1 Програмирање *FPGA* чипова

Технологије које се најчешће користе за програмирање *FPGA* чипова су: *antifuse*, *SRAM* и *FLASH EPROM*. Ове технологије могу се разумети уколико се разуме технологија *fusible-link* [8].

Технологија *fusible-link*

FPGA чипови купљени за програмирање у технологији *fusible-link* (слика 11) садрже осигураче на свим вредностима улазних променљивих и њиховим комплементима. Комбинације улазних вредности и комплемената променљивих представљају улаз у конјукције које формирају излаз. Програмирање *FPGA* чипова коришћењем технологије *fusible-link* се постиже сагоревањем осигурача на вредностима улазних променљивих или вредностима њихових комплемената. Сагоревањем осигурача прекида се веза одређене вредности улазних променљивих и улаза у конјукције. Осигурачи се сагоревају пропуштањем већег напона на улазе одређених променљивих. *FPGA* чипови у *fusible-link* технологији се могу програмирати само једанпут јер се осигурачи који су прегорели не могу заменити или регенерисати [8].

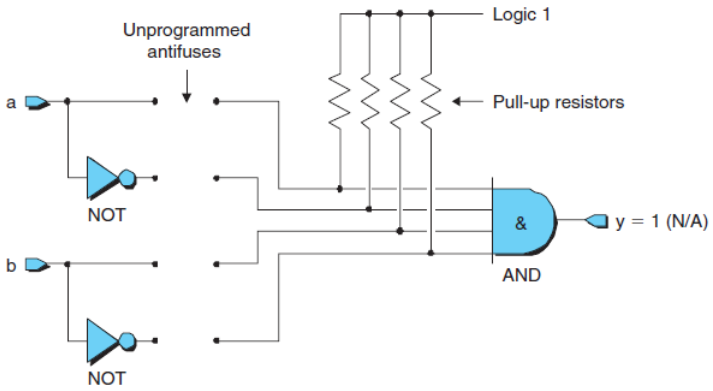
Иако се техноологија *fusible-link* за програмирање *FPGA* чипова више не користи представља добар пример за разумевање осталих технологија за програмирање *FPGA* чипова [8].



Слика 11: Пример иницијалног *FPGA* чипа за програмирање технологијом *fusible-link*

Технологија *antifuse*

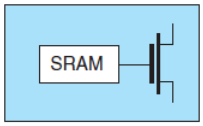
FPGA чипови који користе технологију *antifuse* (слика 12) за програмирање функционишу на супротан начин од *FPGA* чиповима који користе технологију *fusible-link*. Купљени чипови за програмирање у технологији *antifuse* не садрже ни једну везу вредности улазних променљивих нити њихових комплемената са конјукцијама које представљају излаз. Програмирање *FPGA* чипови са технологијом *antifuse* се врши додавањем одређених веза на улазне променљиве или њихове комплементе. *FPGA* чипови у технологији *antifuse* се такође могу програмирати само једанпут јер се додате везе се не могу уклонити. Овако програмирани *FPGA* чипови имају велику брзину и троше мало струје [8].



Слика 12: Пример иницијалног *FPGA* чипа за програмирање технологијом *antifuse*

Технологија програмирања чипа на основу *SRAM* меморије

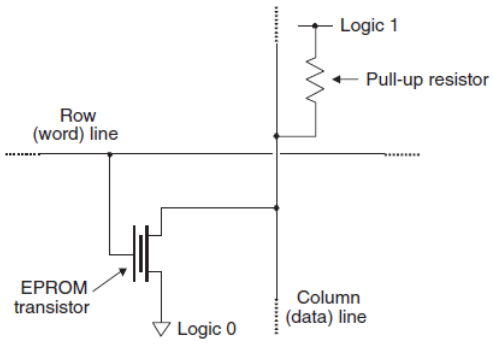
Технологија за програмирање *FPGA* чипова на основу *SRAM* (Static Random Access Memory) је најчешће коришћена технологија. *FPGA* чипови који користе *SRAM* технологију садрже велики број *SRAM* ћелија (слика 13). *SRAM* ћелија има особину да чува прослеђену вредност све док чип има електриичног напајања. Мана *SRAM* технологије за програмирање *FPGA* чипова је то што се за израду ћелије користи шест транзистора и што се након нестанка електричног напајања губе све програмиране вредности *SRAM* ћелија. Међутим битна предност је у томе што се *FPGA* чипови програмирани у технологији *SRAM* могу изнова програмирати [8].



Слика 13: Пример *SRAM* ћелије *FPGA* чипа

Технологија програмирања чипа на основу *FLASH* меморије

Технологија програмирања *FPGA* чипova на основу *FLASH* меморије је новија технологија која је настала од технологије за програмирање чипова *EPROM* (erasable programmable read-only memory). У иницијалном непрограмираном стању *EPROM FPGA* чипови садрже ненапуњене контакте који немају утицаја на остатак чипа. Програмирање *FPGA* чипова са *EPROM* технологијом се врши пропуштањем великог напона између контаката и уземљења. Промене су доста стабилне и под нормалним условима рада могу да остану не промењене чак и до десет година. Овако програмирани контакти у *FPGA* чиповима са *EPROM* технологијом могу да служе као меморијске ћелије (слика 14) [8].



Слика 14: Пример *EPROM* ћелије *FPGA* чипа

Напреднија верзија технике *EPROM* за програмирање *FPGA* чипова је *EEPROM* (electrically erasable programmable read-only memories) чија је величина ћелије два ипо

пута већа од еквивалентне *EPROM* ћелије јер садржи два транзистора и довољан размак између тих транзистора [8].

FLASH технологија за програмирање *FPGA* чипова може да има сличну архитектуру као *EPROM* или *EEPROM*. Постоји могућност да се цео *FPGA FLASH* чип врати на иницијално стање. *FPGA FLASH* чипови троше мало електричне енергије, брзи су и могу се изнова програмирати [8].

4.3 Превођење програма коришћењем Макселеровог компајлера

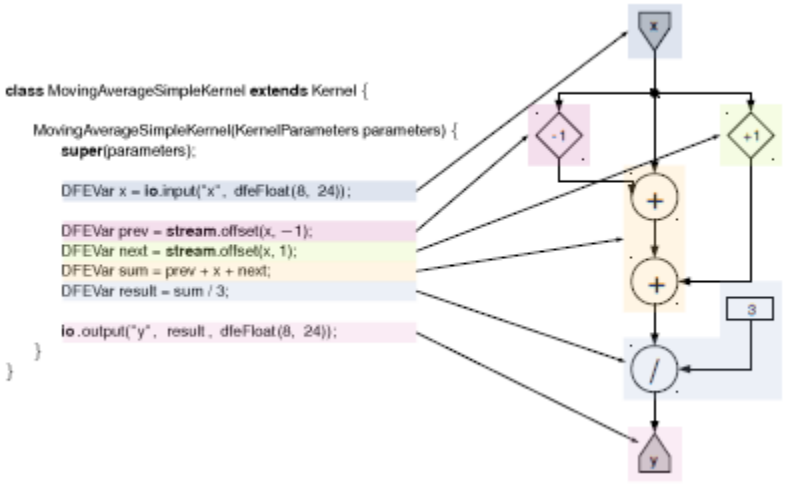
Програм писан за коришћење Макселеровог компајлера (енг. MaxCompiler) се састоји од више датотека које могу да се сврстају у три групе: кернел (енг. kernel), менаџер (енг. manager) и програм за процесор (енг. CPU) [7, 9].

Кернел датотека је датотека која се пише на језику *MaxJ* и представља део програма који садржи израчунавања која ће бити имплементирана на хардверу. Програм може да садржи више кернел датотека [7, 9].

Менаџер датотека се такође пише на језику *MaxJ* и чини везу између кернела, процесора и радне меморије (енг. RAM). Може да постоји само једна менаџер датотека [7, 9].

Програм за процесор представлља део koda који је могуће писати на језицима *C* и *C++* и представља део који ће програм користити за читање и писање података у кернел или радну меморију [7, 9].

Компилација Макселеровим компајлером може да траје дуже време (до неколико сати зависи од сложености програма), али је зато покретање програма веома брзо (неколико милисекунди). Макселеров компајлер преводи наведене датотеке писане на језику *MaxJ* у датокеке које представљају дизајн *FPGA* са екстензијом *.max* и везује их са компајлираном *C* или *C++* датотеком [7, 9]. Пример превођења *MaxJ* кода на *FPGA* (слика 15)



Слика 15: Пример компилације кернел датотеке на *FPGA* дизајн

5 Реализација напада на *DES* и резултати

Коришћењем акцелератора *Maxeler* добија се могућност паралелизације процецеса шифровања отвореног текста са смањеном потрошњом електричне енергије. Акцелератор паралелизацијом процеса шифровања отвара могућности реализације напада са познатим паром отворени текст, шифрат на систем *DES* алгоритмом грубе силе.

За развој програма коришћено је окружење *MaxIDE* на дистрибуцији линукса *CentOS*. Окружење *MaxIDE* садржи преводиоц *MaxCompiler* који на основу кода написаног на високом језику генерише хардверску имплементацију.

Програм се састоји од дела писаног за процесор који генерише све могуће комбинације битова кључа дужине 56 битова, припрема меморију за чување резултата који се добијају по извршавању паралелизованог дела кода. Овај део кода је писан на језику *C* и при извршавању програма извршава се на главном процесору.

Други део програма је кернел датотека која је писана на *MaxJ*. Овај део кода чита битове кључа прослеђене од кернел датотеке преко менаџер датотеке и врши пермутације над њим генеришући 16 кључева за примену у рундама. Преко менаџер датотеке прослеђује се још и порука која би требало да се шифрује. Порука пролази кроз 16 рунди уз помоћ 16 генерисаних кључева и трансформише се у шифрат који се прослеђује (преко менаџер датотеке) кернел датотеци који га уписује у предвиђени део меморије. Код кернел датотеке се преводи на дигитални дизајн *FPGA* кола. Овај део кода је могуће извршавати паралелно на *Maxeler* акцелератору.

5.1 Опис дела кода који се извршава на процесору

У делу кода који је писан за извршавање на процесору неопходно је обезбедити одређени меморијски простор који ће се користити за чување података које треба проследити кернел датотеци и података које треба преузети од кернел датотеке. За алоцирење меморије користи се тип *uint64_t* јер одговара величини блока (кључа или поруке) који се користи за шифровање алгоритмом *DES*.

```
1  uint64_t size = 0x100000000000000;
2  uint64_t part = 0x4000000000;
3  uint64_t sizeBytes = part * sizeof(uint64_t);
4  uint64_t *keys = malloc(sizeBytes);
5  uint64_t *result = malloc(sizeBytes);
6  uint64_t message = 0x0123456789ABCDEF;
7
8  for(uint64_t i=0; i<size; i++){
9    keys[i%part] = i;
10   if(size % part == part-1)
11     DES(message, size, keys, result);
12 }
```

Наведени део кода алоцира меморију за одређени број комбинација (величина се чува у променљивој *part*) кључева (*keys*) и меморију за уписивање резултата који ће се користити у једној размени информација са кернел датотеком (*result*). Промењива *size* представља укупан број комбинација кључева, а променљива *message* представља поруку коју треба шифровати. Петља наведеног кода позива функцију *DES* када се

попуни величина низа *keys*. Вредност која представља број кључева који се шаљу кернел датодеци у једној итерацији (*part*) зависи од меморије коју акцелератор поседује.

Функција *DES(message, size, keys, result)* је дефинисана у кернел датотеци и не извршава се на процесору.

5.2 Опис кернел датотеке

Крнел датотека садржи имплементацију функције *DES*. Да би се промењиве које су дефинисане у делу кода који се извршава на процесору и пренете помоћу менаџер датотеке могле користити, потребно је дефинисати *DFEVar* тип променљивих којима ће се доделити пренете вредности. Код кернел датотеке се извршава паралелно за сваку појединачну вредност низа.

```
1 DFEVar key = io.input("keys", bit64_t);
2 DFEVar message = io.scalarInput("message", bit64_t);
```

Свако језгро *Maxeler* акцелератора узима посебну вредност из низа *keys* који садржи елементе типа *bit64_t* и уписује његову вредност у променљиву *key* типа *DFEVar*. За читање поруке прослеђене преко менаџер датотеке је ствар мало другачија. Пошто је променљива која се прослеђује типа *bit64_t* иста вредност се чита за свако језгро и уписује у променљиву *message* типа *DFEVar*.

За рачунање пермутације битова користи се метода *slice* (*slice(int pos, int len = 1)* враћа *len* битова почевши од бита на позицији *pos*. Ова метода је метода променљиве типа *DFEVar*. Ефикасна је јер се извршава хардврески) и оператор надовезивања #. На пример за рачунање *PC-1* пермутације кључа може се искористити следећи код

```
1 DFEVar k =
2         key.slice(6)# key.slice(13)# key.slice(20)# key.slice(27)# key.slice
    (34)# key.slice(41)# key.slice(48)
3       # key.slice(55)# key.slice(5)# key.slice(12)# key.slice(19)# key.slice
    (26)# key.slice(33)# key.slice(40)
4       # key.slice(47)# key.slice(54)# key.slice(4)# key.slice(11)# key.slice
    (18)# key.slice(25)# key.slice(32)
5       # key.slice(39)# key.slice(46)# key.slice(53)# key.slice(3)# key.slice
    (10)# key.slice(17)# key.slice(24)
6       # key.slice(0)# key.slice(7)# key.slice(14)# key.slice(21)# key.slice
    (28)# key.slice(35)# key.slice(42)
7       # key.slice(49)# key.slice(1)# key.slice(8)# key.slice(15)# key.slice
    (22)# key.slice(29)# key.slice(36)
8       # key.slice(43)# key.slice(50)# key.slice(2)# key.slice(9)# key.slice
    (16)# key.slice(23)# key.slice(30)
9       # key.slice(37)# key.slice(44)# key.slice(51)# key.slice(31)# key.slice
    (38)# key.slice(45)# key.slice(52);
```

Наведени код врши пермутацију над променљивом *key* тако што помоћу методе *slice* извлачи битове на датим позицијама и надовезује их помоћу оператора #. Резултат се уписује у променљиву *k* типа *DFEVar*

Такође метода *slice* се може користити за рачунање леве и десне половине пермутованог кључа издвајањем узастопних битова.

```
1 DFEVar c0 = k.slice(28,28);
```

```
2
3 DFEVar d0 = k.slice(0,28);
```

Оператор ексклузивног или ^ може да се користи и за промењиве типа *DFEVar*.

```
1 DFEVar ke = k ^ e;
```

Један од начина да се приступи вредностима табела S које су константне је да их чувамо припремљене у меморији.

```
1 private static final double S_val[] =
2     {
3         S1,
4         S2,
5         S3,
6         S4,
7         S5,
8         S6,
9         S7,
10        S8
11              };
12
13 private final Memory<DFEVar> S = mem.alloc(bit4_t, 512);
```

Наведени код уписује у меморију акцелератора садржај табела S.

Читање вредности за дефинисане меморијске локације S могу се прочитати методом *read*. Пример ове методе приликом читања из табела S дат је у коду:

```
1 DFEVar s =
2   S.read(constant.var(bit3_t, s_table[0])#ke.slice(47)#ke.slice(42)#ke.slice
      (43, 4))
3   # S.read(constant.var(bit3_t, s_table[1])#ke.slice(41)#ke.slice(36)#ke.slice
      (37, 4))
4   # S.read(constant.var(bit3_t, s_table[2])#ke.slice(35)#ke.slice(30)#ke.slice
      (31, 4))
5   # S.read(constant.var(bit3_t, s_table[3])#ke.slice(29)#ke.slice(24)#ke.slice
      (25, 4))
6   # S.read(constant.var(bit3_t, s_table[4])#ke.slice(23)#ke.slice(18)#ke.slice
      (19, 4))
7   # S.read(constant.var(bit3_t, s_table[5])#ke.slice(17)#ke.slice(12)#ke.slice
      (13, 4))
8   # S.read(constant.var(bit3_t, s_table[6])#ke.slice(11)#ke.slice(6)#ke.slice
      (7, 4))
9   # S.read(constant.var(bit3_t, s_table[7])#ke.slice(5)#ke.slice(0)#ke.slice
      (1, 4));
```

5.3 Превођење, покретање и резултати програма

За превођење и покретање програма коришћена је *MAX3 Vectis Maxeler* картица која има 24*GB DDR* радне меморије. У картици налази *Xilinx*-ов *FPGA* чип *Virtex 6 SXT475*. Превођење програма на овој картици трајало је 17 минута и 4 секунде. На основу мерења перформанси програма за кључеве дужине: 20, 25, 30, 35 битова утврђено да би на овој картици напад са познатим паром отворен текст и шифрат могао да се изврши за 23 дана.

6 Закључак

Помоћу акцелератора *Maxeler* (картицом *MAX3 Vectis Maxeler*) напад грубе силе на *DES* са познатим паром отворен текст и шифрат могуће је успешно извршити, користећи развијени програм, у временском ринтервалу од 23 дана. У раду је детаљно описан поступак шифровања алгоритмом *DES*, као и начин на који би се алгоритам могао имплементирати користећи окружење *MaxIDE* са преводиоцем *MaxCompiler*. Рад саджи и опис начина рада и структуре програма за покретање на акцелератору *Maxeler*.

Перформансе програма преведеног *MaxCompiler*-ом зависе од перформанси *Maxeler*-овог акцелератора. Временски период од 23 дана за успешно извршавање напада грубе силе би могло да се смањи коришћењем акцелератора са већом меморијом, која омогућује више паралелних обрада.

Верзија алгоритма *DES* која је успела да се одржи до данас је троструки *DES* (Triple-DES). Троструки *DES* је алгоритам који три пута шифрује блок отвореног текста примењујући алгоритам *DES* са два различита кључа (први кључ је једнак трећем). Ова верзија алгоритма *DES* је сигурнија од основне верзије алгоритма.

Напад грубом силом на алгоритам *DES* са познатим паром отворен текст и шифрат је успешно извршен за мање од дан програмибилним чипом *Spartan-6 LX* [12]. Напад грубом силома на алгоритам *DES* није више питање могућности, већ питање постигнутог времена.

Литература

[1] Bruce Schneier Applied Cryptography, Second Edition: Protocols, Algorthms, and Source Code in C *Wiley Computer Publishing, John Wiley & Sons*, 1996.

[2] Adam J. Elbirt Understanding and Applying Cryptography and Data Security *CRC Press, Taylor & Francis Group*, 2009.

[3] Jonathan S. Greenfield Distributed Programming Paradigms with Cryptography Applications *Springer*, 1994.

[4] Matt Curtin Brute Force: Cracking the Data Encryption Standard *Springer*, 2005.

[5] `https://w2.eff.org`

[6] `http://emea.emc.com`

[7] Veljko Milutinović, Jakob Salom, Nemanja Trifunović, Roberto Giorgi Guide to DataFlow Supercomputing Basic Concepts, Case Studies, and a Detailed Example *Springer*, 2015.

[8] Clive Maxfield FPGAs: Instant Access *Newnes*, 2008.

[9] `http://www.maxeler.com`

[10] Whitfield Diffie, Martin Hellman Exhaustive cryptanalysis of the NBS Data Encryption Standard *Stanford University*, 1977.

[11] Kenneth H. Rosen Taylor & Francis Group *Newnes*, 2007.

[12] `http://www.sciengines.com/company/news-a-events/74-des-in-1-day.html`