• Appendix A - Hardware Performance Figures for the AES Finalists.

The following is a reproduction of some of the more telling results for the Hardware Performance Simulations of Round 2 Advanced Encryption Standard Algorithms by the National Security Agency [29]. Please see Chapter three for more details;

Parameter	Iterative 3in1		Pipelined 3in1	
	Min.	Max.	Min.	Max.
Area (um2)	52,717,560	127,432,766	670,181,619	1,333,099,6
Transistor Count	734,751	1,950,277	9,340,818	20,667,3
Input/Outputs Required	520	520	520	5
Throughput (Mbps)	29.06	56.71	1123.89	2189.
Key Setup Time Encrypt (ns)	9553	27470	3718	10206.
Key Setup Time Decrypt (ns)	9553	27470	3718	10206.
Algorithm Setup Time (ns)	0	0	0	
Time to Encrypt One Block (ns)	2256.92	8600.1928	1987.98	3872.
Time to Decrypt One Block(ns)	2256.92	8600.1928	1987.98	3872.

Figure 50 Mars 3-in-1 result summary

Parameter	Iterative 128-Bit		Pipelined 128-Bit	
	Min.	Max.	Min.	Max
Area (um2)	51,986,292	126,827,662	669,735,987	1,332,(
Transistor Count	724,403	1,941,371	9,334,605	20,(
Input/Outputs Required	520	520	520	
Throughput (Mbps)	29.06	56.71	1123.89	
Key Setup Time Encrypt (ns)	9553	27429	3712	
Key Setup Time Decrypt (ns)	9553	27429	3712	
Algorithm Setup Time (ns)	0	0	0	
Time to Encrypt One Block (ns)	1987.98	3872.26	1987.98	
Time to Decrypt One Block(ns)	1987.98	3872.26	1987.98	

Figure 51 Mars 128Bit result summary

Parameter	Iterative 3in1		Pipelined 3in1	
	Min.	Max.	Min.	Max.
Area (um2)	16,027,545	21,660,006	318,931,799	592,088,5
Transistor Count	248,686	430,436	7,304,500	16,055,2
Input/Outputs Required	520	520	520	5
Throughput (Mbps)	54.94	103.98	1192.47	2170.
Key Setup Time Encrypt (ns)	8,139.00	15,348.96	3,659.51	6,922.
Key Setup Time Decrypt (ns)	8,139.00	15,348.96	3,659.51	6,922.
Algorithm Setup Time (ns)	0	0	0	
Time to Encrypt One Block (ns)	1233.2	2329	1179.2	214
Time to Decrypt One Block(ns)	1233.2	2329	1179.2	214

Figure 52 RC6 3-in-1 result summary

Parameter	Iterative 128-Bit		Pipelined 128-Bit	
	Min.	Max.	Min.	Max.
Area (um2)	13,966,637	19,248,830	285,334,063	453,248,2
Transistor Count	217,008	307,247	6,853,619	11,611,3
Input/Outputs Required	520	520	520	5
Throughput (Mbps)	59.68	102.83	1216.38	2196
Key Setup Time Encrypt (ns)	5,742.00	13,776.64	3,728.50	6,897
Key Setup Time Decrypt (ns)	5,742.00	13,776.64	3,728.50	6,897
Algorithm Setup Time (ns)	0	0	0	
Time to Encrypt One Block (ns)	1,244.80	6,674.62	1,165.40	2,104
Time to Decrypt One Block(ns)	1,244.80	6,674.62	1,165.40	2,104

Figure 53 RC6 128Bit-in-1 result summary

Parameter	Key Size Selected	Iterative 3in1		Pipelined 3in1	
		Min.	Max.	Min.	Max.
Area (um2)	-	24,158,661	46,362,850	307,129,513	471,996,329
Transistor Count	-	481,002	1,029,054	4,190,611	7,130,697
Input/Outputs Required	-	520	520	520	520
Throughput (Mbps)	128 Bit	190.05	447.55	4026.42	5337.78
	192 Bit	158.38	372.96	4026.42	5337.78
	256 Bit	135.75	319.68	4026.42	5337.78
Key Setup Time Encrypt (ns)	128 Bit	0	0	0/29.28*	0/44.76*
	192 Bit	0	0	0/29.28*	0/44.76*
	256 Bit	0	0	0/29.28*	0/44.76*
Key Setup Time Decrypt (ns)	128 Bit	286	673.5	233.99	318.63
	192 Bit	343.2	808.2	262.83	374.33
	256 Bit	400.4	942.9	277.25	402.18
Algorithm Setup Time (ns)	-	0	0	0	0
Time to Encrypt One Block (ns)	128 Bit	286	673.5	239.8	317.9
	192 Bit	343.2	808.2	287.76	381.48
	256 Bit	400.4	942.9	335.72	445.06
Time to Decrypt One Block (ns)	128 Bit	286	673.5	239.8	317.9
	192 Bit	343.2	808.2	287.76	381.48
	256 Bit	400.4	942.9	335.72	445.06

* Note: Both Key Setup/Key Agility times provided, as they are different for this algorithm

Figure 54 Rijndael 3-in-1 result summary

Parameter	Iterative 128-Bit		Pipelined 128-Bit	
	Min.	Max.	Min.	Max.
Area (um2)	20,739,239	33,851,050	262,721,033	419,88
Transistor Count	275,485	641,681	3,660,325	4,29
Input/Outputs Required	520	520	520	
Throughput (Mbps)	271.13	605.77	4200.85	57
Key Setup Time Encrypt (ns)	0.00	0.00	0/26.6*	0/3
Key Setup Time Decrypt (ns)	211.30	472.10	226.87	3
Algorithm Setup Time (ns)	0	0	0	
Time to Encrypt One Block (ns)	211.3	472.1	222.8	
Time to Decrypt One Block(ns)	211.3	472.1	222.8	

* Note: Both Key Setup/Key Agility times provided, as they are different for this algorithm

Figure 55 Rijndael 128Bit result summary

Note; Both Key Setup and Key Agility times for Twofish are different.

Parameter	Iterative 3in1		Pipelined 3in1	
	Min.	Max.	Min.	Max.
Area (um2)	13,779,834	23,274,086	210,124,852	438,561
Transistor Count	204,617	345,483	3,298,104	5,741
Input/Outputs Required	520	520	520	
Throughput (Mbps)	102.00	202.30	5298.01	803
Key Setup Time Encrypt (ns)	19.77	39.21	18.98	2
Key Setup Time Decrypt (ns)	672.18	1333.14	212.55	36
Algorithm Setup Time (ns)	0	0	0	
Time to Encrypt One Block (ns)	510.08	775.18	510.08	77
Time to Decrypt One Block(ns)	510.08	775.18	510.08	77

Figure 56 Serpent 3-in-1 result summary

Within the implementation of the SERPENT algorithm, the key size is padded to 256 bits internally, so the effects of varying key sizes have negligible impact on the performance metrics of the design. Consequently, only the 3-in-1cases is provided.

Parameter	Key Size Selected	Iterative 3in1		Pipelined 3in1	
		Min.	Max.	Min.	Max.
Area (um2)	-	12,700,019	23,044,514	187,058,851	343,109,723
Transistor Count	-	182,533	377,599	2,666,940	5,750,915
Input/Outputs Required	-	520	520	520	520
Throughput (Mbps)	128 Bit	59.50	104.60	1429.00	2278.00
	192 Bit	50.00	90.96	1164.00	1957.00
	256 Bit	42.95	80.51	977.20	1718.00
Key Setup Time Encrypt (ns)	128 Bit	42.48	65.36	0/65.63*	0/127.42*
	192 Bit	61.28	106.78	0/65.63*	0/127.42*
	256 Bit	79.49	149	0/65.63*	0/127.42*
Key Setup Time Decrypt (ns)	128 Bit	42.48	65.36	0/65.63*	0/127.42*
	192 Bit	61.28	106.78	0/65.63*	0/127.42*
	256 Bit	79.49	149	0/65.63*	0/127.42*
Algorithm Setup Time (ns)	-	0	0	0	0
Time to Encrypt One Block (ns)	128 Bit	1223.2	2151.2	1123.8	1791
	192 Bit	1407.2	2559.6	1307.8	2199.4
	256 Bit	1589.8	2980	1490.4	2619.8
Time to Decrypt One Block (ns)	128 Bit	1223.2	1223.2	1123.8	1791
	192 Bit	1407.2	1407.2	1307.8	2199.4
	256 Bit	1589.8	1589.8	1490.4	2619.8

* Note: Both Key Setup/Key Agility times provided, as they are different for this algorithm

Figure 57 Twofish 3-in-1 result summary

Parameter	Iterative 128-Bit		Pipelined 128-Bit	
	Min.	Max.	Min.	Max.
Area (um2)	9,158,239	16,110,756	121,705,687	225,298,323
Transistor Count	134,997	264,058	1,785,286	3,783,973
Input/Outputs Required	520	520	520	520
Throughput (Mbps)	60.84	105.14	1341.44	2273.53
Key Setup Time Encrypt (ns)	60.87	105.2	0/47.29*	0/91.86*
Key Setup Time Decrypt (ns)	60.87	105.2	0/47.29*	0/91.86*
Algorithm Setup Time (ns)	0	0	0	0
Time to Encrypt One Block (ns)	1217.4	2104.00	1126	1908.4
Time to Decrypt One Block(ns)	1217.4	2104.00	1126	1908.4

* Note: Both Key Setup/Key Agility times provided, as they are different for this algorithm

Figure 58 Twofish 128Bit result summary

Note; Both Key Setup and Key Agility times for Twofish are different.

• Appendix B - A implemantation of A5 in C.

```
/*
* A pedagogical implementation of A5/1.
*
* Copyright (C) 1998-1999: Marc Briceno, Ian Goldberg, and David Wagner
* see www.scard.org for full version
*/
#include <stdio.h>
/* Masks for the three shift registers */
#define R1MASK 0x07FFFF /* 19 bits, numbered 0..18 */
#define R2MASK 0x3FFFFF /* 22 bits, numbered 0..21 */
#define R3MASK 0x7FFFFF /* 23 bits, numbered 0..22 */
/* Middle bit of each of the three shift registers, for clock control */
#define R1MID 0x000100 /* bit 8 */
#define R2MID 0x000400 /* bit 10 */
#define R3MID 0x000400 /* bit 10 */
/* Feedback taps, for clocking the shift registers.
* These correspond to the primitive polynomials
```

```
* x^19 + x^5 + x^2 + x + 1, x^22 + x + 1,

* and x^23 + x^15 + x^2 + x + 1. */

#define R1TAPS 0x072000 /* bits 18,17,16,13 */

#define R2TAPS 0x300000 /* bits 21,20 */

#define R3TAPS 0x700080 /* bits 22,21,20,7 */


/* Output taps, for output generation */

#define R1OUT 0x040000 /* bit 18 (the high bit) */

#define R2OUT 0x200000 /* bit 21 (the high bit) */

#define R3OUT 0x400000 /* bit 22 (the high bit) */


typedef unsigned char byte;

typedef unsigned long word;

typedef word bit;


/* Calculate the parity of a 32-bit word, i.e. the sum of its bits modulo 2 */

bit parity(word x) {

x ^= x>>16;

x ^= x>>8;

x ^= x>>4;

x ^= x>>2;

x ^= x>>1;

return x&1;

}


/* Clock one shift register */

word clockone(word reg, word mask, word taps) {

word t = reg & taps;

reg = (reg << 1) & mask;
```

```
reg |= parity(t);

return reg;

}


/* The three shift registers. They're in global variables to make the code easier to *understand. A better implementation would not use global variables. */

word R1, R2, R3;


/* Look at the middle bits of R1,R2,R3, take a vote, and return the majority value of *those 3 bits. */

bit majority() {

int sum;

sum = parity(R1&R1MID) + parity(R2&R2MID) + parity(R3&R3MID);

if (sum >= 2)

return 1;

else

return 0;

}


/* Clock two or three of R1,R2,R3, with clock control according to their middle bits.

* Specifically, we clock Ri whenever Ri's middle bit agrees with the majority value of *the three middle bits.*/

void clock() {

bit maj = majority();

if (((R1&R1MID)!=0) == maj)

R1 = clockone(R1, R1MASK, R1TAPS);

if (((R2&R2MID)!=0) == maj)

R2 = clockone(R2, R2MASK, R2TAPS);

if (((R3&R3MID)!=0) == maj)

R3 = clockone(R3, R3MASK, R3TAPS);

}
```

```
/* Clock all three of R1,R2,R3, ignoring their middle bits. This is only used for key *setup. */

void clockallthree() {

R1 = clockone(R1, R1MASK, R1TAPS);

R2 = clockone(R2, R2MASK, R2TAPS);

R3 = clockone(R3, R3MASK, R3TAPS);

}


/* Generate an output bit from the current state. You grab a bit from each register via *the output generation taps; then you XOR the resulting three bits. */

bit getbit() {

return parity(R1&R1OUT)^parity(R2&R2OUT)^parity(R3&R3OUT);

}


/* Do the A5/1 key setup. This routine accepts a 64-bit key and a 22-bit frame *number. */

void keysetup(byte key[8], word frame) {

int i;

bit keybit, framebit;

/* Zero out the shift registers. */

R1 = R2 = R3 = 0;


/* Load the key into the shift registers, LSB of first byte of key array first, clocking *each register once for every key bit loaded. (The usual clock control rule is *temporarily disabled.) */

for (i=0; i<64; i++) {

clockallthree(); /* always clock */

keybit = (key[i/8] >> (i&7)) & 1; /* The i-th bit of the

key */

R1 ^= keybit; R2 ^= keybit; R3 ^= keybit;

}


/* Load the frame number into the shift registers, LSB first, clocking each register *once for every key bit loaded. (The usual clock control rule is still disabled.) */
```

```
for (i=0; i<22; i++) {
clockallthree(); /* always clock */
framebit = (frame >> i) & 1; /* The i-th bit of the frame #*/
R1 ^= framebit; R2 ^= framebit; R3 ^= framebit;
}
/* Run the shift registers for 100 clocks to mix the keying material and frame number
* together with output generation disabled, so that there is sufficient avalanche.
*We re-enable the majority-based clock control rule from now on. */
for (i=0; i<100; i++) {
clock();
}

/* Now the key is properly set up. */
}
/* Generate output. We generate 228 bits of
* keystream output. The first 114 bits is for
* the A->B frame; the next 114 bits is for the
* B->A frame. You allocate a 15-byte buffer
* for each direction, and this function fills
* it in. */
void run(byte AtoBkeystream[], byte BtoAkeystream[]) {
int i;

/* Zero out the output buffers. */
for (i=0; i<=113/8; i++)
AtoBkeystream[i] = BtoAkeystream[i] = 0;
/* Generate 114 bits of keystream for the
* A->B direction. Store it, MSB first. */
for (i=0; i<114; i++) {
```

```
clock();
AtoBkeystream[i/8] |= getbit() << (7-(i&7));
}

/* Generate 114 bits of keystream for the
* B->A direction. Store it, MSB first. */
for (i=0; i<114; i++) {
clock();
BtoAkeystream[i/8] |= getbit() << (7-(i&7));}}
```

• Appendix C - A implementation of COMP128 in C.

```
/* An implementation of the GSM A3A8 algorithm. (Specifically, COMP128.)
* Copyright 1998, Marc Briceno, Ian Goldberg, and David Wagner.
* see www.scard.org for full version
* All rights reserved.
*/
typedef unsigned char Byte;
#include <stdio.h>
/* #define TEST */

/*
* rand[0..15]: the challenge from the base station
* key[0..15]: the SIM's A3/A8 long-term key Ki
* simoutput[0..11]: what you'd get back if you fed rand and key to a real
* SIM.
*
* The GSM spec states that simoutput[0..3] is SRES,
* and simoutput[4..11] is Kc (the A5 session key).
* (See GSM 11.11, Section 8.16. See also the leaked document
* referenced below.)
```

```
* Note that Kc is bits 74..127 of the COMP128 output, followed by 10
* zeros.
* In other words, A5 is keyed with only 54 bits of entropy. This
* represents a deliberate weakening of the key used for voice privacy
* by a factor of over 1000.
*
* Verified with a Pacific Bell Schlumberger SIM. Your mileage may vary.
*
* Marc Briceno <marc@scard.org>, Ian Goldberg <iang@cs.berkeley.edu>,
* and David Wagner <daw@cs.berkeley.edu>
*/

void A3A8(/* in */ Byte rand[16], /* in */ Byte key[16],
/* out */ Byte simoutput[12]);

/* The compression tables. */
static const Byte table_0[512] = {
102,177,186,162, 2,156,112, 75, 55, 25, 8, 12,251,193,246,188,
109,213,151, 53, 42, 79,191,115,233,242,164,223,209,148,108,161,
252, 37,244, 47, 64,211, 6,237,185,160,139,113, 76,138, 59, 70,
67, 26, 13,157, 63,179,221, 30,214, 36,166, 69,152,124,207,116,
247,194, 41, 84, 71, 1, 49, 14, 95, 35,169, 21, 96, 78,215,225,
182,243, 28, 92,201,118, 4, 74,248,128, 17, 11,146,132,245, 48,
149, 90,120, 39, 87,230,106,232,175, 19,126,190,202,141,137,176,
250, 27,101, 40,219,227, 58, 20, 51,178, 98,216,140, 22, 32,121,
61,103,203, 72, 29,110, 85,212,180,204,150,183, 15, 66,172,196,
56,197,158, 0,100, 45,153, 7,144,222,163,167, 60,135,210,231,
174,165, 38,249,224, 34,220,229,217,208,241, 68,206,189,125,255,
239, 54,168, 89,123,122, 73,145,117,234,143, 99,129,200,192, 82,
```

104,170,136,235, 93, 81,205,173,236, 94,105, 52, 46,228,198, 5,

57,254, 97,155,142,133,199,171,187, 50, 65,181,127,107,147,226,

184,218,131, 33, 77, 86, 31, 44, 88, 62,238, 18, 24, 43,154, 23,

80,159,134,111, 9,114, 3, 91, 16,130, 83, 10,195,240,253,119,

177,102,162,186,156, 2, 75,112, 25, 55, 12, 8,193,251,188,246,

213,109, 53,151, 79, 42,115,191,242,233,223,164,148,209,161,108,

37,252, 47,244,211, 64,237, 6,160,185,113,139,138, 76, 70, 59,

26, 67,157, 13,179, 63, 30,221, 36,214, 69,166,124,152,116,207,

194,247, 84, 41, 1, 71, 14, 49, 35, 95, 21,169, 78, 96,225,215,

243,182, 92, 28,118,201, 74, 4,128,248, 11, 17,132,146, 48,245,

90,149, 39,120,230, 87,232,106, 19,175,190,126,141,202,176,137,

27,250, 40,101,227,219, 20, 58,178, 51,216, 98, 22,140,121, 32,

103, 61, 72,203,110, 29,212, 85,204,180,183,150, 66, 15,196,172,

197, 56, 0,158, 45,100, 7,153,222,144,167,163,135, 60,231,210,

165,174,249, 38, 34,224,229,220,208,217, 68,241,189,206,255,125,

54,239, 89,168,122,123,145, 73,234,117, 99,143,200,129, 82,192,

170,104,235,136, 81, 93,173,205, 94,236, 52,105,228, 46, 5,198,

254, 57,155, 97,133,142,171,199, 50,187,181, 65,107,127,226,147,

218,184, 33,131, 86, 77, 44, 31, 62, 88, 18,238, 43, 24, 23,154,

159, 80,111,134,114, 9, 91, 3,130, 16, 10, 83,240,195,119,253

},

table_1[256] = {

19, 11, 80,114, 43, 1, 69, 94, 39, 18,127,117, 97, 3, 85, 43,

27,124, 70, 83, 47, 71, 63, 10, 47, 89, 79, 4, 14, 59, 11, 5,

35,107,103, 68, 21, 86, 36, 91, 85,126, 32, 50,109, 94,120, 6,

53, 79, 28, 45, 99, 95, 41, 34, 88, 68, 93, 55,110,125,105, 20,

90, 80, 76, 96, 23, 60, 89, 64,121, 56, 14, 74,101, 8, 19, 78,

76, 66,104, 46,111, 50, 32, 3, 39, 0, 58, 25, 92, 22, 18, 51,

57, 65,119,116, 22,109, 7, 86, 59, 93, 62,110, 78, 99, 77, 67,

```
12,113, 87, 98,102, 5, 88, 33, 38, 56, 23, 8, 75, 45, 13, 75,
95, 63, 28, 49,123,120, 20,112, 44, 30, 15, 98,106, 2,103, 29,
82,107, 42,124, 24, 30, 41, 16,108,100,117, 40, 73, 40, 7,114,
82,115, 36,112, 12,102,100, 84, 92, 48, 72, 97, 9, 54, 55, 74,
113,123, 17, 26, 53, 58, 4, 9, 69,122, 21,118, 42, 60, 27, 73,
118,125, 34, 15, 65,115, 84, 64, 62, 81, 70, 1, 24,111,121, 83,
104, 81, 49,127, 48,105, 31, 10, 6, 91, 87, 37, 16, 54,116,126,
31, 38, 13, 0, 72,106, 77, 61, 26, 67, 46, 29, 96, 37, 61, 52,
101, 17, 44,108, 71, 52, 66, 57, 33, 51, 25, 90, 2,119,122, 35
}, table_2[128] = {
52, 50, 44, 6, 21, 49, 41, 59, 39, 51, 25, 32, 51, 47, 52, 43,
37, 4, 40, 34, 61, 12, 28, 4, 58, 23, 8, 15, 12, 22, 9, 18,
55, 10, 33, 35, 50, 1, 43, 3, 57, 13, 62, 14, 7, 42, 44, 59,
62, 57, 27, 6, 8, 31, 26, 54, 41, 22, 45, 20, 39, 3, 16, 56,
48, 2, 21, 28, 36, 42, 60, 33, 34, 18, 0, 11, 24, 10, 17, 61,
29, 14, 45, 26, 55, 46, 11, 17, 54, 46, 9, 24, 30, 60, 32, 0,
20, 38, 2, 30, 58, 35, 1, 16, 56, 40, 23, 48, 13, 19, 19, 27,
31, 53, 47, 38, 63, 15, 49, 5, 37, 53, 25, 36, 63, 29, 5, 7
}, table_3[64] = {
1, 5, 29, 6, 25, 1, 18, 23, 17, 19, 0, 9, 24, 25, 6, 31,
28, 20, 24, 30, 4, 27, 3, 13, 15, 16, 14, 18, 4, 3, 8, 9,
20, 0, 12, 26, 21, 8, 28, 2, 29, 2, 15, 7, 11, 22, 14, 10,
17, 21, 12, 30, 26, 27, 16, 31, 11, 7, 13, 23, 10, 5, 22, 19
}, table_4[32] = {
15, 12, 10, 4, 1, 14, 11, 7, 5, 0, 14, 7, 1, 2, 13, 8,
10, 3, 4, 9, 6, 0, 3, 2, 5, 6, 8, 9, 11, 13, 15, 12
}, *table[5] = { table_0, table_1, table_2, table_3, table_4 };

void A3A8(/* in */ Byte rand[16], /* in */ Byte key[16],
```

```
/* out */ Byte simoutput[12])

{

Byte x[32], bit[128];

int i, j, k, l, m, n, y, z, next_bit;

/* ( Load RAND into last 16 bytes of input ) */

for (i=16; i<32; i++)

x[i] = rand[i-16];


/* ( Loop eight times ) */

for (i=1; i<9; i++) {

/* ( Load key into first 16 bytes of input ) */

for (j=0; j<16; j++)

x[j] = key[j];

/* ( Perform substitutions ) */

for (j=0; j<5; j++)

for (k=0; k<(1<<j); k++)

for (l=0; l<(1<<(4-j)); l++) {

m = l + k*(1<<(5-j));

n = m + (1<<(4-j));

y = (x[m]+2*x[n]) % (1<<(9-j));

z = (2*x[m]+x[n]) % (1<<(9-j));

x[m] = table[j][y];

x[n] = table[j][z];

}

/* ( Form bits to bytes ) */

for (j=0; j<32; j++)

for (k=0; k<4; k++)

bit[4*j+k] = (x[j]>>(3-k)) & 1;

/* ( Permutation but not on the last loop ) */
```

```
if (i < 8)
for (j=0; j<16; j++) {
x[j+16] = 0;
for (k=0; k<8; k++) {
next_bit = ((8*j + k)*17) % 128;
x[j+16] |= bit[next_bit] << (7-k);
}
}
}

/ * ( At this stage the vector x[] consists of 32 nibbles.
* The first 8 of these are taken as the output SRES. */

/* The remainder of the code is not given explicitly in the
* standard, but was derived by reverse-engineering.*/

for (i=0; i<4; i++)
simoutput[i] = (x[2*i]<<4) | x[2*i+1];
for (i=0; i<6; i++)
simoutput[4+i] = (x[2*i+18]<<6) | (x[2*i+18+1]<<2)
| (x[2*i+18+2]>>2);
simoutput[4+6] = (x[2*6+18]<<6) | (x[2*6+18+1]<<2);
simoutput[4+7] = 0;
}
```

• Appendix D – A Verilog implementation of A5

A5/1 Circuit, expressed in Verilog. The following is equivalent to the purported A5/1

pedagogical C in Appendix A.

```
/*
a5.v
Purported A5/1 circuit, expressed in Verilog.
13 May 99
David Honig
honig@sprynet.com
Derived from Briceno, Goldberg, Wagner's Pedagogical C Code
of May 99.
To load key: assert Startloading, load data starting on next clock,
bitwise (1 delay + 64 key + 22 frame clocks).
Then wait for Doneloading to be asserted (100 more clocks). Then
harvest your bits.
A testbench and sample output is appended as comments.
This synthesizes to about 150 LCs and runs at 80 Mhz on
the smaller Altera CPLDs e.g., 10K30.
*/
module a5(Clk, Reset_n,
Bitout,
Keybit,
Startloading,
Doneloading);

input Clk, Reset_n;
output Bitout; // output keystream
reg Bitout;
input Keybit; // input keybits 64 + 22
input Startloading; // initial keyload
```

```
output Doneloading; // signal done of keyloading

reg Doneloading;


// internal state; lsb is leftmost

reg [18:0] lfsr_1;

reg [21:0] lfsr_2;

reg [22:0] lfsr_3;


reg [1:0] State; // FSM control

reg [6:0] Counter; // for counting steps

reg [2:0] Phase; // for sequencing phases


wire hi_1, hi_2, hi_3;

assign hi_1 = lfsr_1[18];

assign hi_2 = lfsr_2[21];

assign hi_3 = lfsr_3[22];


wire mid1, mid2, mid3;

assign mid1=lfsr_1[8];

assign mid2=lfsr_2[10];

assign mid3=lfsr_3[10];


wire maj;

assign maj=majority(mid1, mid2, mid3);


wire newbit1, newbit2, newbit3;

assign newbit1= ( lfsr_1[13] ^ lfsr_1[16] ^ lfsr_1[17] ^ lfsr_1[18] );

assign newbit2= ( lfsr_2[20] ^ lfsr_2[21] ) ;

assign newbit3= ( lfsr_3[7] ^ lfsr_3[20] ^ lfsr_3[21] ^ lfsr_3[22] );
```

```
parameter IDLE=0;

parameter KEYING=1;

parameter RUNNING=2;


always @(posedge Clk or negedge Reset_n) begin

if (!Reset_n) begin: resetting


$display("a5 reset");

Doneloading <=0;

Bitout <=0;

{lfsr_1, lfsr_2, lfsr_3} <= 64'h 0;

{State, Counter, Phase} <=0;


end // reset

else begin

case (State)


IDLE: begin: reset_but_no_key


if (Startloading) begin: startloadingkey

// $display("Loading key starts at %0d ", $time);

State <= KEYING;

{lfsr_1, lfsr_2, lfsr_3} <= 64'h 0;

Phase <=0; Counter<=0;

end // if

end // idle


KEYING: begin
```

```
case (Phase)

0: begin: load64andclock
clockallwithkey;

// $display("Loading key bit %0b %0d at %0d %0x", Keybit, Counter, $time, lfsr_1);
if (Counter==63) begin
Counter <=0;
Phase <= Phase +1;
$display(" ");
end
else Counter <=Counter+1;
end

1: begin: load22andclock

// $display("Loading frame bit %0b at %0d %0d %0x", Keybit, Counter, $time, lfsr_1);
clockallwithkey;
if (Counter==21) begin
Counter <=0;
Phase <= Phase +1;
end
else Counter <=Counter+1;
end

2: begin: clock100

majclock;
```

```
if (Counter ==100) begin

$display("Done keying, now running %0d\n", $time);

State <= RUNNING;

end

else Counter <= Counter+1;

end

endcase // Phase

end // keying


RUNNING: begin


Doneloading <=1; // when done loading

Bitout <= hi_1 ^ hi_2 ^ hi_3;

majclock;


end // running

endcase // State

end // else not resetting

end // always


function majority;

input a,b,c;


begin

case({a,b,c}) // synopsys parallel_case

3'b 000: majority=0;

3'b 001: majority=0;

3'b 010: majority=0;
```

```
3'b 011: majority=1;


3'b 100: majority=0;
3'b 101: majority=1;
3'b 110: majority=1;
3'b 111: majority=1;
endcase
end
endfunction

task clock1;
begin
lfsr_1 <= ( lfsr_1 << 1 ) | newbit1;
end
endtask

task clock2;
begin
lfsr_2 <= (lfsr_2 << 1) | newbit2;
end
endtask
task clock3;
begin
lfsr_3 <= (lfsr_3 << 1) | newbit3;
end
endtask
task clockall;
begin
clock1;
```

```
clock2;

clock3;

end

endtask

task clockallwithkey;

begin

lfsr_1 <= ( lfsr_1 << 1 ) | newbit1 ^ Keybit;

lfsr_2 <= ( lfsr_2 << 1 ) | newbit2 ^ Keybit;

lfsr_3 <= ( lfsr_3 << 1 ) | newbit3 ^ Keybit;

end

endtask

task majclock;

begin

if (mid1 == maj) clock1;

if (mid2 == maj) clock2;

if (mid3 == maj) clock3;

end

endtask

endmodule


/***************** CUT HERE FOR TESTBENCH test_a5.v ***************************

module test_a5;


reg Clk, Reset_n;

wire Bitout; // output keystream

reg Keybit; // input keybits 64 + 22

reg Startloading; // initial keyload

wire Doneloading; // signal done of keyloading
```

```
reg [0:7] key [7:0];

reg [22:0] frame;


a5 dut(Clk, Reset_n,

Bitout,

Keybit,

Startloading,

Doneloading);


always @(Clk) #5 Clk <= ~Clk;


integer i,j;


initial begin

#5

key[0]= 8'h 12;

key[1]= 8'h 23;

key[2]= 8'h 45;

key[3]= 8'h 67;

key[4]= 8'h 89;

key[5]= 8'h AB;

key[6]= 8'h CD;

key[7]= 8'h EF;


frame <= 22'h 134;

Clk <=0;

Reset_n <=1;

Startloading <=0;

Keybit <=0;
```

```
#10 Reset_n <=0;

#10 Reset_n <=1;


// key setup

#100

Startloading <=1; $display("Starting to key %0d", $time);

for (i=0; i<8; i=i+1) begin

for (j=0; j<8; j=j+1) begin

#10 Startloading <=0;

Keybit <= key[i] >> j;

end // j

end // i


for (i=0; i<22; i=i+1) begin

#10 Keybit <= frame[i];

end


wait(Doneloading); $display("Done keying %0d", $time);

$write("\nBits out: \n");

repeat (32) #10 $write("%b", Bitout);

$display("\nknown good=\n%b", 32'h 534EAA58);

#1000 $display("\nSim done."); $finish;

end // init

endmodule
```

• Appendix E – C code for Brute force attack on A5

```
/* Brute force attack on A5, by Eoin Ward eoinward@yahoo.com
```

```
*A5 implementation from Briceno, Goldberg, Wagner's Pedagogical C Code

*Optimizations

*Generates one byte first and compares before generating rest of output.

*only generates 64 bits of output instead of the full 228 bits.

*/

#include <stdio.h>

#include <stdlib.h>

#include <conio.h>

#include <time.h>


/* Masks for the three shift registers */

#define R1MASK 0x07FFFF /* 19 bits, numbered 0..18 are all set to 1 */

#define R2MASK 0x3FFFFF /* 22 bits, numbered 0..21 are all set to 1*/

#define R3MASK 0x7FFFFF /* 23 bits, numbered 0..22 are all set to 1*/


/* Middle bit of each of the three shift registers, for clock control */

#define R1MID 0x000100 /* bit 8 --the 8th bit is 1*/

#define R2MID 0x000400 /* bit 10 --the 10th bit is 1*/

#define R3MID 0x000400 /* bit 10 --the 10th bit is 1*/


/* Feedback taps, for clocking the shift registers. These correspond to the primitive

* polynomials x^19 + x^5 + x^2 + x + 1, x^22 + x + 1,

* and x^23 + x^15 + x^2 + x + 1. */


#define R1TAPS 0x072000 /* bits 18,17,16,13 --are one */

#define R2TAPS 0x300000 /* bits 21,20 --are one */

#define R3TAPS 0x700080 /* bits 22,21,20,7 --are one*/


/* Output taps, for output generation */
```

```
#define R1OUT 0x040000 /* bit 18 (the high bit) */

#define R2OUT 0x200000 /* bit 21 (the high bit) */

#define R3OUT 0x400000 /* bit 22 (the high bit) */


typedef unsigned char byte;

typedef unsigned long word;

typedef word bit;

/*prototypes missing from orignail code causing warnings*/

bit majority(void);

void clockallthree(void);

void Clock(void);

bit getbit(void);

void test(void);

bit parity(word x);

word clockone(word reg, word mask, word taps);


/* Calculate the parity of a 32-bit word, i.e. the sum of its bits modulo 2 */

bit parity(word x) {

x ^= x>>16; /* ^= Assign bitwise XOR >> shift right */

x ^= x>>8;

x ^= x>>4;

x ^= x>>2;

x ^= x>>1;

return x&1;}


/* Clock one shift register */

word clockone(word reg, word mask, word taps) {

word t = reg & taps;

reg = (reg << 1) & mask;
```

```
reg |= parity(t);
return reg;}

/* The three shift registers. */
word R1, R2, R3;

/* Look at the middle bits of R1,R2,R3, take a vote, and
* return the majority value of those 3 bits. */
bit majority() {
int sum;
sum = parity(R1&R1MID) + parity(R2&R2MID) + parity(R3&R3MID);
if (sum >= 2) /* the sum of the 8th bit of */
return 1; /* R1 the 10th bit of r2 and the 10th bit of R3*/
else
return 0;
}

/* Clock two or three of R1,R2,R3, according to their middle bits.*/
void Clock() {
bit maj = majority();
if (((R1&R1MID)!=0) == (int)maj)
R1 = clockone(R1, R1MASK, R1TAPS);
if (((R2&R2MID)!=0) == (int)maj)
R2 = clockone(R2, R2MASK, R2TAPS);
if (((R3&R3MID)!=0) == (int)maj)
R3 = clockone(R3, R3MASK, R3TAPS);
}

/* Clock all three of R1,R2,R3, ignoring their middle bits.
```

```
* This is only used for key setup. */
void clockallthree() {
R1 = clockone(R1, R1MASK, R1TAPS);
R2 = clockone(R2, R2MASK, R2TAPS);
R3 = clockone(R3, R3MASK, R3TAPS);
}

/* Generate an output bit from the current state.
* You grab a bit from each register via the output generation taps;
* then you XOR the resulting three bits. */
bit getbit() {
return parity(R1&R1OUT)^parity(R2&R2OUT)^parity(R3&R3OUT);
}

/* Do the A5/1 key setup. This routine accepts a 64-bit key and
* a 22-bit frame number. */
void keysetup(byte key[8], word frame) {
int i;
bit keybit, framebit;

/* Zero out the shift registers. */
R1 = R2 = R3 = 0;

/* Load the key into the shift registers,
* LSB of first byte of key array first,
* clocking each register once for every
* key bit loaded. (The usual clock control rule is temporarily disabled.) */
for (i=0; i<64; i++) {
clockallthree(); /* always clock */
```

```
keybit = (key[i/8] >> (i&7)) & 1; /* The i-th bit of the key */

R1 ^= keybit; R2 ^= keybit; R3 ^= keybit;

}


/* Load the frame number into the shift registers, LSB first,

* clocking each register once for every key bit loaded. (The usual clock

* control rule is still disabled.) */

for (i=0; i<22; i++) {

clockallthree(); /* always clock */

framebit = (frame >> i) & 1; /* The i-th bit of the frame #*/

R1 ^= framebit; R2 ^= framebit; R3 ^= framebit;

}


        /* Run the shift registers for 100 clocks to mix the keying material and frame *number together with
        output generation disabled, so that there is sufficient *avalanche. We re-enable the majority-based clock
        control rule from now on. */

for (i=0; i<100; i++) {

Clock();

}


/* Now the key is properly set up. */

}


/* This generates the first byte in the AtoB stream to for comparision, if it's the same *the rest of the bits will be
generated*/

void runfirstbyte(byte AtoBkeystream[])

{

int i;

for (i=0; i<=63/8; i++) /*zero register first 14 bytes*/

AtoBkeystream[i] = 0;

for (i=0; i<8; i++) {
```

```
Clock();
AtoBkeystream[i/8] |= (byte)(getbit() << (7-(i&7)));
                    }

}
/*only runs if first byte compares as the same as the know good output
* This only produces the remaining 56 bits not the full 228 bits. */

void run(byte AtoBkeystream[]) {
int i;
for (i=8; i<64; i++) {
Clock();
AtoBkeystream[i/8] |= (byte)(getbit() << (7-(i&7)));
}
}

/*Code that implements the attack. */
void test() {
word frame = 0x134; /* known frame number*/
byte key[8] = {0x00, 0x00, 0x40, 0x66, 0x88, 0xAA, 0xCC, 0xEE};
byte goodAtoB[8] = { 0x53, 0x4E, 0xAA, 0x58, 0x2F, 0xE8, 0x15,
0x1A}; /* first 64bits of the know good output */
byte AtoB[8];
int i,b1,b2,b3,b4,b5,b6,b7,b8,count=0;

/* big loop to increment each bit in the key*/
for (b8=0;b8<256;b8++)
{
key[7]= (byte)(key[7] + 1);
```

```
for (b7=0;b7<256;b7++)
{
key[6]= (byte)(key[6] + 1);
for (b6=0;b6<256;b6++)
{
key[5]= (byte)(key[5] + 1);
                    for (b5=0;b5<256;b5++)
{
key[4]= (byte)(key[4] + 1);
                                  for (b4=0;b4<256;b4++)
{
key[3]= (byte)(key[3] + 1);
                                                for (b3=0;b3<256;b3++)
{
key[2]= (byte)(key[2] + 1);
printf(" %d tests so far\n" ,(65536*count));
count=count+1; /*prints number of keys tested*/
                                                                  for( b2=0;b2<256;b2++)
{
key[1]= (byte)(key[1] + 1);
                                                                                  for( b1=0;b1<256;b1++)
{
int failed=0;
key[0] = (byte)(key[0] + 1);
keysetup(key, frame);
runfirstbyte(AtoB); /*generate first byte and test*/
if (AtoB[0] != goodAtoB[0])
failed = 1;
if(failed){}else{
```

```
run(AtoB); /* if same generate the rest*/

for (i=1; i<8; i++)

if (AtoB[i] != goodAtoB[i])

failed = 1;

if (failed) {} else{ /*if = 1 try next key else print out key*/

printf("key found.\n");

printf("key: 0x");

for (i=0; i<8; i++)

printf("%02X", key[i]);

printf("\n frame number: 0x%06X\n", (unsigned int)frame);

printf("known good output(64bits only): \n");

printf(" A->B: 0x");

for (i=0; i<4; i++)

printf("%02X", goodAtoB[i]);

printf("\n observed output:\n");

printf(" A->B: 0x");

for (i=0; i<4; i++)

printf("%02X", AtoB[i]);

printf("\n");

return;

}}

}

}

}

}

}

}

}
```

```
}


int main(void) {

long starttime;

long endtime;

time( (time_t*) &starttime ); /*sets a start time*/

test(); /*Runs Attack*/

time( (time_t*) &endtime);

printf("Elapsed: %d", (int)endtime - starttime);

/* output the time taken for the attack in seconds very little over head as it sets the sarttime at the beign and just takes it
away form the end time no counting involved*/

getch();

return 0;

}
```