Samira C. Oliva Madrigal, CmpE 130

BLOCK CIPHERS: CRYPTO WORK HORSE

***Abstract*— A high-level introduction to the structure of ciphers, block ciphers, and attacks such as Side Channel Attacks. Two canonical examples are presented with emphasis on the AES-128 block cipher with special focus on the core algorithms used and applications in Cryptography to help maintain data integrity and solve contemporary security problems.**

I. INTRODUCTION

Cryptography is everywhere, from secure communication (HTTPs for web traffic, IEEE 802.11i WPA2 for wireless traffic), encrypting files, user authentication, and in general, content protection. It is an incredible tool that forms the base layer for a number of security mechanisms. Information protection is always a concern especially in consideration of computationally secure ciphers and the advancement towards Quantum Computers and Quantum Cryptography.

There are several examples of badly broken ad-hoc ciphers. The paper presents, ciphers, pseudo random generators, information theoretical security, semantic security, and block ciphers. After an overview of the building blocks, follows background on DES—how it came to surface and security flaws. The last part concentrates on AES-128, provides a high-level of how it works, and explores the overall algorithm, core algorithms, rationale behind the three different keys available, current use and support as well as possible attacks known.

II. BACKGROUND

First, a cipher is defined over a triple (K, M, C) denoting the key space (set of all possible keys), message space (set of all possible messages), and cipher text space (set of all possible cipher texts). It is pair of efficient algorithms E and D where E: K x M → C and D: K x C → M such that these algorithms are in accord with the consistency equation (1) which every cipher must satisfy, otherwise, it cannot decrypt. E is always a randomized algorithm and D is always deterministic and has no dependency on any randomness used by the algorithm.

The One Time Pad (OTP) is a cipher with perfect secrecy (see equation 2) not prone to cipher text (C) only attacks; it is a secure stream cipher. OTP is defined as $M = C = \{0,1\}^n$ and $K = \{1,0\}^n$ where $\{0,1\}^n$ denotes the set all n-bit binary strings. The cipher text $C := E(k, m) = k \oplus m$ and the plaintext, $P := D(k, C) = k \oplus c$. The problem is that the random bit string key it uses must have length as long as the plain text (P) to be encrypted—impractical. Though computing XORs is rather fast, truly random key generation is difficult.

There are two types of random key generators, truly random (TRG) and pseudo random (PRG). The former type are difficult because of their sensitivity to the changes in the environment. [2] Therefore generating truly random keys in practice, is not something easy. Thos leads into what is known as a stream cipher (crypto building block) which replaces a truly random key with a pseudorandom key. In this case, the key stream from the pseudorandom key is XORed with the text stream of the message to produce the cipher text stream. The PRG is defined as a function G, that takes in an s-bit seed in the seed space $\{0,1\}^s$ and maps it to a larger n-bit string in the space $\{0,1\}^n$. It has the property that $n >> s$. The goal of the generator G is that it is efficiently computed by a deterministic algorithm which implies that there is no randomness in it. However, the seed given as input is random. The second property for the generator is that the output should look random (more on this on section IV).

A.Original Equations

The consistency equation:

$$\forall m \in M, \forall k \in K: \\ D(k, E(k,m)\} = m \qquad (1)$$

Shannon's Definition of Perfect-Secrecy:
A cipher E, D over (K, M, C) has perfect secrecy:

$\leftrightarrow \forall m_0, m_1 \in M$, the length (m_0) = length(m_1)

$\leftrightarrow \forall c \in C: c$

$$P [E(k, m_0) = c] = P[E(k,m_1) = c] \qquad (2)$$

where k is uniform in K, uniformly sampled from the key space K. It is also implied that $|K| \geq |M|$.

Stream Cipher:

$$C := E(k, m) := m \oplus G(k) \qquad (3)$$

$$P := E(k, c) := c \oplus G(k) \qquad (4)$$

Advantage of adversary over G:

$$Adv[A, G] := |P(A(G(k) = 1] – P[A(r) = 1] | \in [1,0] \qquad (5)$$

Samira C. Oliva, samira.c.oliva@acm.org

Semantic Security Advantage of Adversary A a over scheme E:

$$Adv_{SS}[A,E] := | P(W_0) - P(W_1) | \in [1,0] \quad (6)$$

Feistel Network mapping 2n-bit input to 2n-bit L_i output:

$$L_i = R_{i-1} \text{ for } i=1,\ldots,d \quad (7)$$

Feistel Network mapping 2n-bit input to 2n-bit R_i output:

$$R_i = f_i(R_{i-1}) \oplus L_i \quad \text{for } i=1,\ldots,d \quad (8)$$

Inverse of Feistel Network for L_i output:

$$L_i = f_{i+1}(L_{i+1}) \oplus R_{i+1} \quad \text{for } i=1,\ldots,d \quad (9)$$

Inverse of Feistel Network R_i output:

$$R_i = L_{i+1} \quad \text{for } i=1,\ldots,d \quad (10)$$

Linear attack relationship:

$$P(\{m[i_1] \oplus \ldots \oplus m[i_r]\} \oplus \{c[j_1] \oplus \ldots \oplus c[j_k]\} = \{k[l_1] \oplus \ldots \oplus c[l_u]\}) = \tfrac{1}{2} + \varepsilon \quad (11)$$

B.Theorems

Yao's Theorem: (1)

If $\forall i \in \{0,\ldots, n-1\}$ PRG, G is unpredictable at position i, then G is a secure PRG.

Theorem: (2)

G: K → M $\{0,1\}^n$ is a secure PRG → stream cipher E derived from G is semantically secure.

C.Schematics

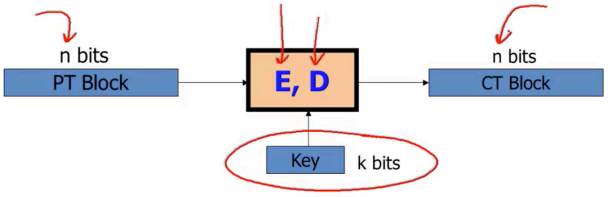


Fig. 1 Block Cipher General Description

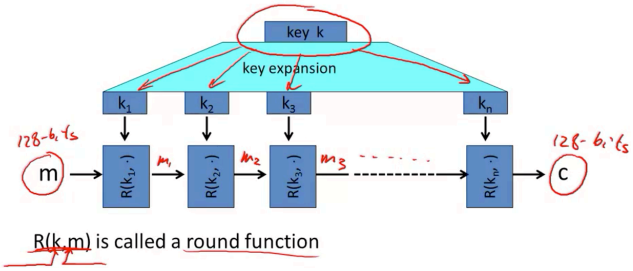


Fig. 2 High-level overview of a Block Cipher

Given functions $f_1, \ldots, f_d$: $\{0,1\}^n \rightarrow \{0,1\}^n$

Goal: build invertible function $F: \{0,1\}^{2n} \rightarrow \{0,1\}^{2n}$

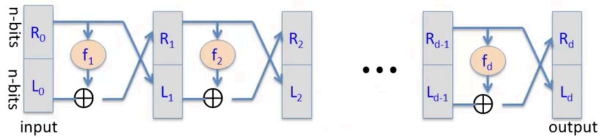
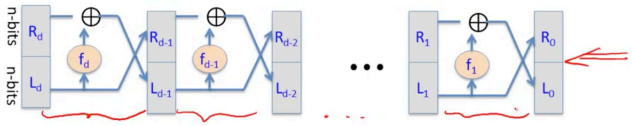


Fig. 3 DES cipher



- Inversion is basically the same circuit, with $f_1, \ldots, f_d$ applied in reverse order

Fig. 4 DES decryption circuit

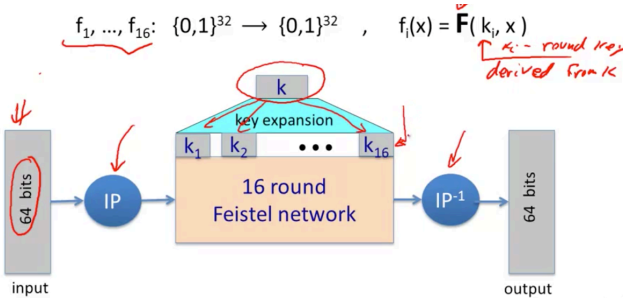


Fig. 5 16 round Feistel Network: DES

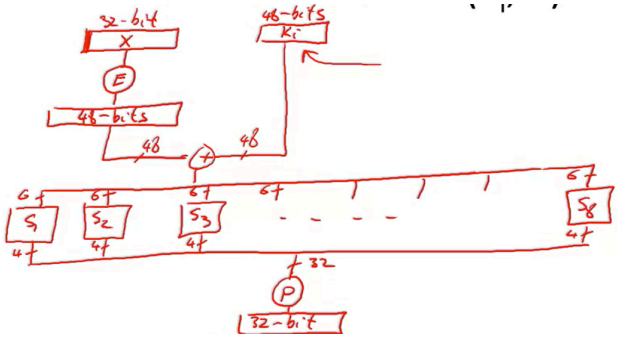


Fig. 6 Function $F(k_i, x)$ from DES

S_5	Middle 4 bits of input															
	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111
Outer bits 00	0010	1100	0100	0001	0111	1010	1011	0110	1000	0101	0011	1111	1101	0000	1110	1001
Outer bits 01	1110	1011	0010	1100	0100	0111	1101	0001	0101	0000	1111	1010	0011	1001	1000	0110
Outer bits 10	0100	0010	0001	1011	1010	1101	0111	1000	1111	1001	1100	0101	0110	0011	0000	1110
Outer bits 11	1011	1000	1100	0111	0001	1110	0010	1101	0110	1111	0000	1001	1010	0100	0101	0011

4

16

Fig. 7 S-box

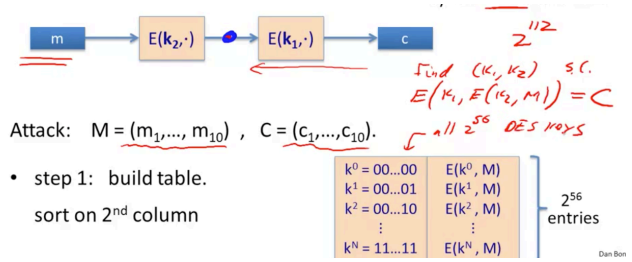


Fig. 8 Meet-in-the-middle attack

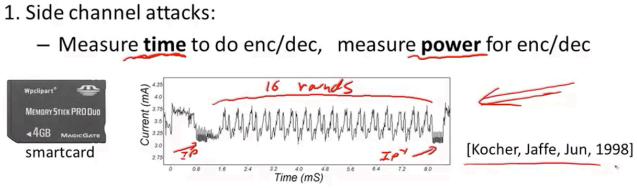


Fig. 9 Timing attacks

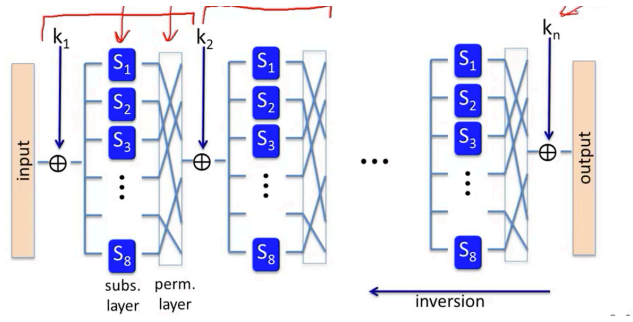


Fig. 10 AES Substitution Network

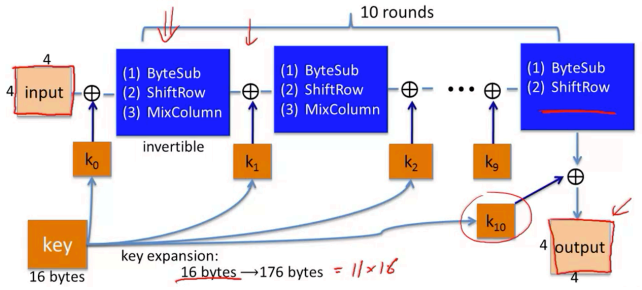


Fig. 11 AES Schematic

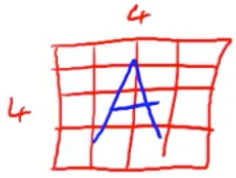


Fig. 12 AES Internals: ByteSub function

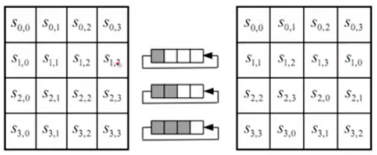


Fig. 13 AES Internals: ShiftRows function

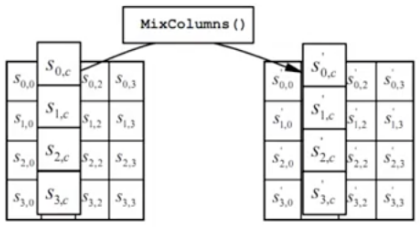


Fig. 14 AES Internals: MixColumns function

	Code size	Performance
Pre-compute round functions (24KB or 4KB)	largest	fastest: table lookups and xors
Pre-compute S-box only (256 bytes)	smaller	slower
No pre-computation	smallest	slowest

Fig. 15 AES code size and performance trade-off

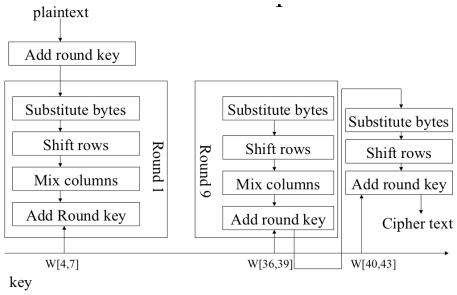


Fig. 16 Encryption flow-chart of AES Cipher

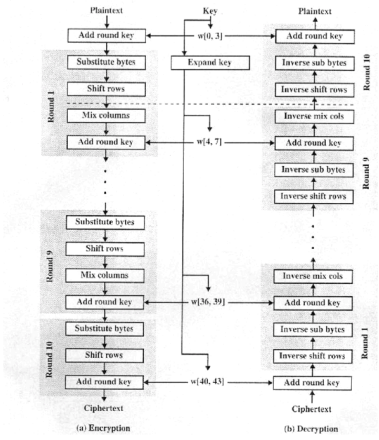


Fig. 17 Flow-char for AES algorithm scheme

Key size (words/bytes/bits)	4/16/128	6/24/192	8/32/256
Number of rounds	10	12	14
Expanded key size (words/byte)	44/176	52/208	60/240

Fig. 18 AES specifics

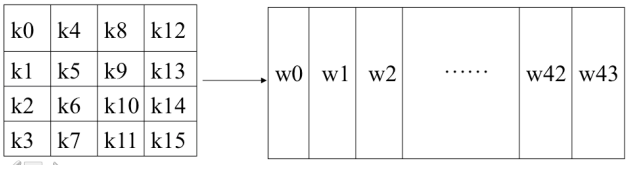


Fig. 19 Input Block and Key Expansion

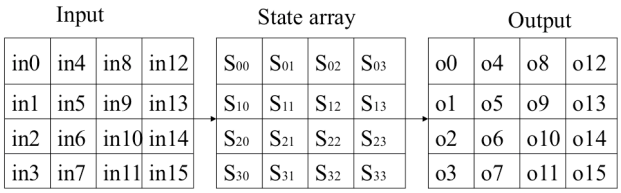


Fig. 20 Input Block to State Array

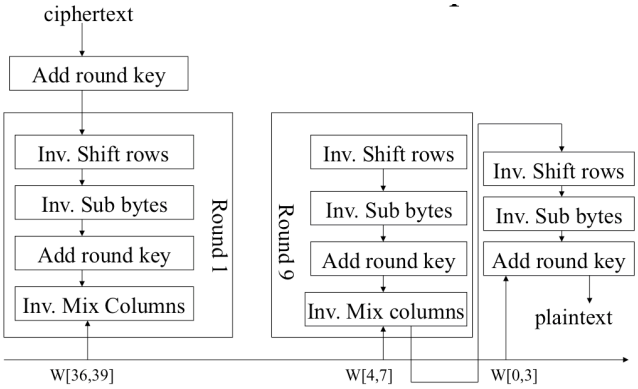


Fig. 21 Decryption flow-chart of AES Cipher

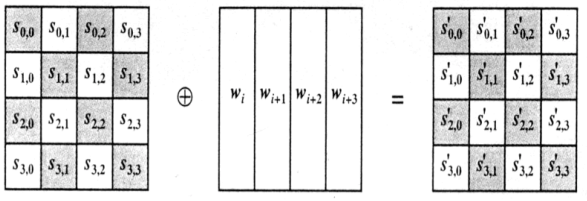


Fig. 22 AddRoundKey function

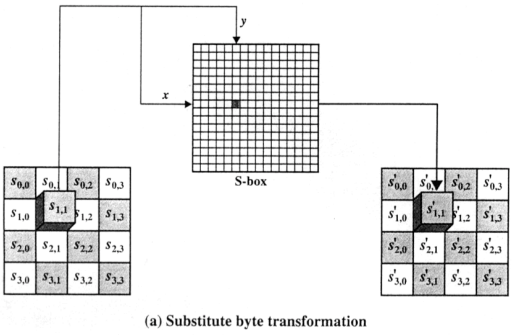


Fig. 23 ByteSub function

x \ y	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
0	63	7c	77	7b	f2	6b	6f	c5	30	01	67	2b	fe	d7	ab	76
1	ca	82	c9	7d	fa	59	47	f0	ad	d4	a2	af	9c	a4	72	c0
2	b7	fd	93	26	36	3f	f7	cc	34	a5	e5	f1	71	d8	31	15
3	04	c7	23	c3	18	96	05	9a	07	12	80	e2	eb	27	b2	75
4	09	83	2c	1a	1b	6e	5a	a0	52	3b	d6	b3	29	e3	2f	84
5	53	d1	00	ed	20	fc	b1	5b	6a	cb	be	39	4a	4c	58	cf
6	d0	ef	aa	fb	43	4d	33	85	45	f9	02	7f	50	3c	9f	a8
7	51	a3	40	8f	92	9d	38	f5	bc	b6	da	21	10	ff	f3	d2
8	cd	0c	13	ec	5f	97	44	17	c4	a7	7e	3d	64	5d	19	73
9	60	81	4f	dc	22	2a	90	88	46	ee	b8	14	de	5e	0b	db
a	e0	32	3a	0a	49	06	24	5c	c2	d3	ac	62	91	95	e4	79
b	e7	c8	37	6d	8d	d5	4e	a9	6c	56	f4	ea	65	7a	ae	08
c	ba	78	25	2e	1c	a6	b4	c6	e8	dd	74	1f	4b	bd	8b	8a
d	70	3e	b5	66	48	03	f6	0e	61	35	57	b9	86	c1	1d	9e
e	e1	f8	98	11	69	d9	8e	94	9b	1e	87	e9	ce	55	28	df
f	8c	a1	89	0d	bf	e6	42	68	41	99	2d	0f	b0	54	bb	16

Figure 7. S-box: substitution values for the byte xy (in hexadecimal format).

Fig. 24 S-box

$$\begin{bmatrix} b'_0 \\ b'_1 \\ b'_2 \\ b'_3 \\ b'_4 \\ b'_5 \\ b'_6 \\ b'_7 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \\ b_6 \\ b_7 \end{bmatrix} + \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}$$

Fig. 25 S-box Computation

x \ y	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
0	52	09	6a	d5	30	36	a5	38	bf	40	a3	9e	81	f3	d7	fb
1	7c	e3	39	82	9b	2f	ff	87	34	8e	43	44	c4	de	e9	cb
2	54	7b	94	32	a6	c2	23	3d	ee	4c	95	0b	42	fa	c3	4e
3	08	2e	a1	66	28	d9	24	b2	76	5b	a2	49	6d	8b	d1	25
4	72	f8	f6	64	86	68	98	16	d4	a4	5c	cc	5d	65	b6	92
5	6c	70	48	50	fd	ed	b9	da	5e	15	46	57	a7	8d	9d	84
6	90	d8	ab	00	8c	bc	d3	0a	f7	e4	58	05	b8	b3	45	06
7	d0	2c	1e	8f	ca	3f	0f	02	c1	af	bd	03	01	13	8a	6b
8	3a	91	11	41	4f	67	dc	ea	97	f2	cf	ce	f0	b4	e6	73
9	96	ac	74	22	e7	ad	35	85	e2	f9	37	e8	1c	75	df	6e
a	47	f1	1a	71	1d	29	c5	89	6f	b7	62	0e	aa	18	be	1b
b	fc	56	3e	4b	c6	d2	79	20	9a	db	c0	fe	78	cd	5a	f4
c	1f	dd	a8	33	88	07	c7	31	b1	12	10	59	27	80	ec	5f
d	60	51	7f	a9	19	b5	4a	0d	2d	e5	7a	9f	93	c9	9c	ef
e	a0	e0	3b	4d	ae	2a	f5	b0	c8	eb	bb	3c	83	53	99	61
f	17	2b	04	7e	ba	77	d6	26	e1	69	14	63	55	21	0c	7d

Figure 14. Inverse S-box: substitution values for the byte xy (in hexadecimal format).

Fig. 26 Inverse S-box

$$\begin{bmatrix} b'_0 \\ b'_1 \\ b'_2 \\ b'_3 \\ b'_4 \\ b'_5 \\ b'_6 \\ b'_7 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \\ b_6 \\ b_7 \end{bmatrix} + \begin{bmatrix} 1 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

Fig. 27 Inverse S-box Computation

$$\begin{bmatrix} s'_{0,c} \\ s'_{1,c} \\ s'_{2,c} \\ s'_{3,c} \end{bmatrix} = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \begin{bmatrix} s_{0,c} \\ s_{1,c} \\ s_{2,c} \\ s_{3,c} \end{bmatrix} \quad \text{for } 0 \le c < Nb. \qquad (5.6)$$

As a result of this multiplication, the four bytes in a column are replaced by the following:

$$s'_{0,c} = (\{02\} \bullet s_{0,c}) \oplus (\{03\} \bullet s_{1,c}) \oplus s_{2,c} \oplus s_{3,c}$$

$$s'_{1,c} = s_{0,c} \oplus (\{02\} \bullet s_{1,c}) \oplus (\{03\} \bullet s_{2,c}) \oplus s_{3,c}$$

$$s'_{2,c} = s_{0,c} \oplus s_{1,c} \oplus (\{02\} \bullet s_{2,c}) \oplus (\{03\} \bullet s_{3,c})$$

$$s'_{3,c} = (\{03\} \bullet s_{0,c}) \oplus s_{1,c} \oplus s_{2,c} \oplus (\{02\} \bullet s_{3,c}).$$

Fig. 28 Mix-Columns Computations

$s'(x) = a^{-1}(x) \otimes s(x)$:

$$\begin{bmatrix} s'_{0,c} \\ s'_{1,c} \\ s'_{2,c} \\ s'_{3,c} \end{bmatrix} = \begin{bmatrix} 0e & 0b & 0d & 09 \\ 09 & 0e & 0b & 0d \\ 0d & 09 & 0e & 0b \\ 0b & 0d & 09 & 0e \end{bmatrix} \begin{bmatrix} s_{0,c} \\ s_{1,c} \\ s_{2,c} \\ s_{3,c} \end{bmatrix} \quad \text{for } 0 \le c < Nb. \qquad (5.10)$$

As a result of this multiplication, the four bytes in a column are replaced by the following:

$$s'_{0,c} = (\{0e\} \bullet s_{0,c}) \oplus (\{0b\} \bullet s_{1,c}) \oplus (\{0d\} \bullet s_{2,c}) \oplus (\{09\} \bullet s_{3,c})$$

$$s'_{1,c} = (\{09\} \bullet s_{0,c}) \oplus (\{0e\} \bullet s_{1,c}) \oplus (\{0b\} \bullet s_{2,c}) \oplus (\{0d\} \bullet s_{3,c})$$

$$s'_{2,c} = (\{0d\} \bullet s_{0,c}) \oplus (\{09\} \bullet s_{1,c}) \oplus (\{0e\} \bullet s_{2,c}) \oplus (\{0b\} \bullet s_{3,c})$$

$$s'_{3,c} = (\{0b\} \bullet s_{0,c}) \oplus (\{0d\} \bullet s_{1,c}) \oplus (\{09\} \bullet s_{2,c}) \oplus (\{0e\} \bullet s_{3,c})$$

Fig. 29 Inverse Mix-Columns Computations

Values of RC[j] in hexadecimal are:

j	1	2	3	4	5	6
Rc[j]	01	02	04	08	10	20

Fig. 30 Example of round constant values

Cipher key = 2b7e151628aed2a6abf7158809cf4f3c
w0=2b7e1516 w1=28aed2a6 w2=abf71588 w3=09cf4f3c

i	temp	RotWord	SubWord	Rcon[i/4]	XOR	w[i-4]	result
4	09cf4f3c	cf4f3c09	8a84eb01	01000000	8b84eb01	2b7e1516	a0fafe17
5	A0fafe17					28aed2a6	88542cb1
6	88542cb1					Abf71588	23a33939
7	23a33939					09cf4f3c	2a6c7605

Fig. 31 Round constant example for AES-128

Cipher	Block/key size	Speed (MB/sec)
3DES	64/168	13
AES	128/128	109

Fig. 32 Performance comparison

	Encryption	Decryption
Clock cycles needed	~6600	~8400
Flash usage (bytes)	1839	2423
RAM usage (bytes)	80	80

Fig. 33 AES-128 Performance for high speed setting in C-compiler in IAR

III. PRGs

Because the key length is less than the message length, the stream cipher does not meet perfect secrecy. In order for PRGs to be secure, they must be unpredictable. Otherwise, given even only the first i bits (1,…,i) of the output, there exists an algorithm that can compute the remainder of the (i+1, …, n) bits (the rest of the stream). A PRG is secure ↔ ∀i there is no efficient algorithm that can predict (i + 1) for a non-negligible ε. In practice, ε is a scalar and it is considered non-negligible for $\varepsilon \geq 1/(2^{30})$ (means the event is likely to happen over 1GB of data) and negligible for $\varepsilon \geq 1/(2^{80})$ (means the event will not happen over the life of the key). In essence, it is the PRG that defines the security of the stream cipher. Boneh warns that in general, built in functions like random() from the glibc are never to be used. One example of this is Kerberos Version 4.0 (a network authentication protocol), which used random() and got burned for it.

Boneh notes that "in more rigorous crypto, the definitions are a bit different" and it's more about the probabilities of events taken as functions of a parameter of security. So, in theory, ε is a function $\varepsilon : Z^{\geq 0}$ (functions that act as non-negative integers) that maps to output $R^{\geq 0}$ (non-negative probabilities). [1] Therefore, ε is non-negligible ↔ ∃d: $\varepsilon(\lambda) \geq 1/\lambda^d$ (if ε is ≥ to some polynomial λ of degree d, infintely often for infinitely many times, so ε ≥ 1/polynomial for many λ). [1] And, ε is negligible ↔ ∀d, $\lambda \geq \lambda_d : \varepsilon(\lambda) \leq 1/\lambda^d$, meaning that for any degree polynomial d, ∃ some lowerbound λ_d such that ∀ $\lambda \geq \lambda_d$; the function $\varepsilon(\lambda) \leq 1/\lambda^d$. [1] The function is negligible if it is less than all the polynomial functions ($1/\lambda^d$) for sufficiently large λ. For example, $\varepsilon(\lambda) = 1/2^{\lambda}$ is negligible—the function drops exponentially in λ for any constant d and there's a sufficiently large λ so that $(1/2)^{\lambda} < (1/\lambda)^d$. And, $\varepsilon(\lambda) = 1/\lambda^{1000}$ would be non-negligible since it rarely moves and for say d = 20,000, the function is > than $1/\lambda^{20,000}$ and so for a polynomial fraction, the function is greater.

As an example, if $\varepsilon(\lambda) = \{1/2^{\lambda}$ for odd λ, and $1/\lambda^{1000}$ for even λ}, being exponentially small for the odd λ and polynomially small for the even λ, then ε(λ) would be non-negligible because is a function is polynomially small many times, the event it happens with probabibility ε(λ), it is already too large for practical use in cryptography.[1]

IV. PRG SECURITY AND COMPUTATIONAL SECURITY

A more concise definition of a PRG entails defining what the second property means—what it means for the output of the PRG to look random. So, let G: K → $\{0,1\}^n$ be a PRG; the output of the generator must be indistinguishable from random. Consider a distribution defined by choosing uniformly a random key k in the key space K (K →k, output G(k)) and choosing a truly random key ($\{0,1\}^n \rightarrow$ r, output r). These two distribution must be indistinguishable from each other in order for the PRG to be secure, so an

attacker should not be able to distinguish between the output of G over a tiny set to that of the uniform distribution over the entire set. Statistical tests help to define the concept since such a test could take in an n-bit string and decide if it is looks random or not, yet a good number of statistical tests is by no means a good way to evaluate security.

A.Advantage

The concept of Advantage is what helps to better define security. Measuring the advantage of an adversary (algorithm A) relative to G as seen in equation 5, which basically expresses the likelihood of A output is 1 given a pseudorandom string G(k) minus the probability of A output a 1 given a truly random string r. Since it is a difference of probabilities, the advantage will always be a number in the interval [0,1]. If Adv is close to 1, this means the algorithm can distinguish pseudorandom from truly random and otherwise if Adv is close to 0, so advantage tells whether the adversary was able to break the PRG or not.

A PRG is defined to be secure if ∀ efficient statistical tests A, the advantage $Adv_{PRG}(A,G)$ is "negligible"—algorithm A was very close to 0 and not able to distinguish the PRG from random. Note that this considers only "efficient tests". However, there is no known way of proving a PRG is secure, otherwise if there was such proof, it would implies $P \neq NP$ (can be verified but very hard to compute) and if $P = NP$ (the problem can be verified "NP" but may not be quickly solve in polynomial time "P") would show there exist no secure PRGs but there are no such proves known to date. [1] A very famous therorem by Yao in 1982 shows that an unpredictable PRG is necessarily secure (see theorem 1).

B.Computational Security

In general, the concept used throughout has to do with the indistinguishability from uniform. Two distributions p_1 and p_2 are computationally indistinguishable from random (denoted by $p_1 \approx_p p_1$). This means that p_1 cannot be distinguished from p_2 in polynomial time which means that a statistical test would behave the same for both distributions giving an output less than negligible. So, more concisely, a PRG is secure $\leftrightarrow$ ($K \rightarrow k$, output $G(k)$) $\approx_p$ uniform ($\{0,1\}^n$).

V. SEMANTIC SECURITY

The previous section clearly shows that a stream cipher is secure if its PRG is secure. "We always define security as: what can the attacker do and what is the attacker trying to do?" [1] Well, Shannon's definition of perfect secrecy was found to be too strong and the keys required too long. This paved the way for semantic security—instead of requiring two distributions to be exactly the same, only requires that they be computationally indistinguishable ($\approx_p$). So, though the distribution might be very different, the attacker cannot efficiently distinguish them. A cipher (E, D) is then said to have perfect secrecy $\leftrightarrow \forall m_0,m_1 \in M$ where ($|m_0| = |m_1|$), $\{E(k, m_0)\} \approx_p \{E(k, m_1)\}$ where ($K \rightarrow k$). However, because such definition is still too strong so that it cannot be satisfied. The additional constraint is then that instead of having the definition hold for all message pairs, it holds only for $\forall m_0,m_1$ that can be exhibited by the attacker.

This is more clearly seen and defined by two experiments (for a one time pad) EXP(0) and EXP(1) or EXP(b) where b is 0 or 1. An adversary A is trying to break the system and a challenger comprised of two challengers (in one case it takes a bit b set to 0 and in another b set to 1). The challenger selects a random key, then the adversary outputs to two messages m_0 and m_1 of equal length. The challenger outputs the encryption of m_1 or m_0. In EXP(0), the output is the encryption $c = E(k, m_0)$ and in EXP(1), the output is $c = E(k, m_1)$. Then, the adversary A will try to figure out whether he received the encryption of m_1 or m_0. The following defines the experiment b output is 1, $W_b :=$ [EXP(b) =1], so in EXP(1), $W_1 :=$ [EXP(1) =1] means the adversary output 1 and EXP(0), it means $W_0 :=$ [EXP(0) =1] the adversary output 1.

The advantage of the adversary A over encryption scheme E given in equation 6. Basically, it the difference of two probabilities, the point of interest is then whether A output a 1 in these experiments. If A output 1 in both experiments with the same probability, this means it was not able to distinguish between the two experiments. However, if A outputs 1 in one experiment and 1 with a significant probability in the other, then A was able to distinguish between the experiments. If the advantage is close to 0, it means A was not able to distinguish between the two experiments. Then, a semantically secure encryption scheme E is semantically secure if $\leftrightarrow$ ∀efficient algorithms/adversaries A, $Adv_{SS}[A,E]$ is negligible. This means that there does not exist an efficient algorithm that can distinguish $E(k, m_0)$ from $E(k, m_1)$, so ∀ explicit message pairs $m_0, m_1 \in M$ that the adversary was able to exhibit, he was not able to distinguish between the two distributions $\{E(k, m_0) \approx_p E(k, m_1)\}$.

Some of the implications of this definition mentioned by Boneh, are that "if a cipher is semantically secure, then NO bit of information is revealed to an efficient adversary; this is basically the perfect secrecy concept applied to only efficient adversaries, rather than to all adversaries".[1] Therefore, a secure PRG implies a semantically secure stream cipher (see theorem 2). Basically, a stream cipher that is derived from a secure generator is going to be semantically secure.

VI. WHAT ARE BLOCK CIPHERS

Now, block ciphers are a more powerful primitive. As seen in figure 1, a block cipher is made up of an encryption and decryption algorithm E and D where both take a key k as input. "The point of a block cipher is that it takes in an n bit input and outputs exactly an n bit output". [1] They operate on large blocks of bits of fixed length. [3] Two canonical examples of block ciphers are the Data Encryption Standard

(DES) and the Advanced Encryption Standard AES (discussed later in subsequent sections). In general, the larger the key, the slower the block cipher but the more secure and harder to break.

A.High-level Overview

Block ciphers are typically built by iteration (see figure 2).

i. First input, a key k, is expanded into a sequence of keys $k_1, \ldots, k_n$, called "round keys".
ii. The cipher uses these round keys by encrypting the messages iteratively (over and over) using a round function R(k, m) where k := round key, and m := the current state of the message. So the message m and key k_1, are initially input to R and $R(k_1, m)$ outputs m_1, then the second iteration $R(k_2, m_1) \rightarrow m_2$, the third $R(k_2, m_2) \rightarrow m_3$, all the way to $R(k_n, m_{n-1}) \rightarrow C$—until a cipher text C is obtained.

In general, different block ciphers have different numbers of rounds and different round functions. Although block ciphers are slower than stream ciphers, in some respects they are more efficient.

B.Abstract concept of How Block Ciphers Work

The objective of this subsection is to help show how to use block ciphers in a correct manner by providing an abstraction of a block cipher as a Pseudo Random Permutation (PRP) and a Pseudo Random Function (PRF) which capture more accurately how block ciphers work and help better determine what constructions are correct. So, a PRF is defined over a triple (K, X, Y) where K is the key space, X is the input space, and Y is the output space. It is a function F that takes as input a key K and input X and outputs some Y element of the output space, F: KxX→Y. The only requirements are that the F is efficient to compute and "evaluatable" given x and k. [1]

A PRP however, more accurately describes a block cipher and it is defined over (K, X) key space K and set X. The function E: KxX→X takes as input an element in X and an element in K and outputs an element in X. As usual, the requirements are that there exist an efficient deterministic algorithm to evaluate E(K,X). It is also required that the function E(k,.) is one-to-one once that the key k is fixed which means that it is also invertible. The third requirement is that there exist an efficient inversion of the algorithm D(k,y). "Given some output, there's only one input that maps to that output, and given a particular output, will output the original preimage that mapped to that output" [1] Because a PRP describes "very accurately and syntactically what a block cipher is, the terms of PRP and block cipher are used interchangeably depending on the context being discussed—this way when thinking PRP, concretely, think AES/3DES". [1]

Function wise, a PRP is a PRF with some structure where X = Y and with the secret key k, it is invertible. In a way, a PRP can be considered as a specialized version of a PRF though it is not completely precise to make such considerations.

C.Security of PRF

In explaining what it means for a PRF or PRP to be secure, this will capture what it means for a block cipher to be secure. This is necessary so that when dealing with an actual construction of a block cipher, it is understood what exactly is being constructed. This is illustrated as follows: first, let F: KxX → Y denote a secure PRF. Then, a set of functions is defined, namely, Funs[X,Y] := the set of all functions from set X to set Y and let set $S_F := F(k, .)$, so $S_F = \{F(k, .) : k \in K\} \leq$ Funs[X,Y] [1]. The set S_F "denotes the set of all functions from X to Y that are specified by the PRF (F(k, .)) as soon a key k is fixed, then the second argument floats which defines a function from X to Y; We look at the set of all such functions for all possible keys $k \in K$" [1].

For AES-128 (key length 128 bits), Funs[X,Y] has size $|Y|^{|X|}$, since X and Y are both 2^{128}, F[X,Y] has size $(2^{128})^{2\wedge 128}$, this is "more particles than there are in the universe"[1] And, S_F has size |K|, again for 128-bit keys in AES, that means $|S_{AES}| = 2^{128}$. Clearly, $S_F << F[X,Y]$. This leads to the definition of security for a PRF. A PRF is secure ↔ an arbitrarily chosen function from the set F[X,Y] is indistinguishable from a pseudorandom function from X to Y from the set S_F. This means that "the uniform distribution of the set of pseudorandom functions S_F cannot be distinguished from the uniform distribution on the set of all functions F[X,Y]" [1].

One way to test this is to consider: if an attacker is trying to determine whether he is interacting with a truly random or pseudorandom function, it will submit points in X over and over (x_1, x_2, etc.). Then for each such query, the returned value is either the value of the pseudo random function or a truly random function at the point x. For all queries the attacker will either receive the value from the truly random function or the value from the pseudorandom function. If the attacker cannot differentiate whether he is interacting with a pseudorandom or truly random function, then the PRF is said to be secure.

For a PRP, it is analogous. Attacker will query set X on a pseudorandom permutation or a truly random permutation. Again, a PRP is secure if the attacker cannot tell the difference. So the objective is that the random function and permutations must appear like if they were truly random because even if on a single known input, the output is distinguishable from random, the PRF is basically broken.

A.Obtaining a PRG from a PRF

By assuming a PRF of the form $F: K \times \{0,1\}^n \rightarrow \{0,1\}^n$. Then, a generator is defined as $G: K \rightarrow \{0,1\}^{nt}$ with seed space K for the PRF, and an output space of t blocks each

comprised of n bits, for a total output of nt bits for some given t value. Then a rather simple construction can be made by taking the PRF and evaluating it at 0,…, t. So, G(k) = F(k,0) || F(k,1) || F(k,2) || … || F(k,t), where k is the key of the PRF and || denotes concatenation. That is a generator which takes the key of the PRF and it expands it into nt bits. Boneh donotes that this generator has a key property of being "parallelizable", meaning that given two processors, one can compute the odd entries and the other the even entries, thereby doubling the speed of the generator. [1] This is a very important property since several stream ciphers are "inherently sequential" meaning they run at the same speed despite the number of cores. Since PRGs yield stream ciphers, this basically gives a parallelizable stream cipher.

Defining security is directly obtained from the PRF property where F(k,.) is indistinguishable from a truly random function f(.). The principle lies in the fact that the output of G(k) where G(k) = f(0) || f(1) || … || f(t), using a truly random function f(.), that the output using a pseudorandom function is indistinguishable. This concludes what a block cipher is and the security property for a block cipher.

VII. DES EXAMPLE

Again, as explained previously, block ciphers are built by iteration where a key k is expanded into round keys, then a round function R(k,m) "is applied to an input message over and over again and after they are applied, a cipher text C is produced as the final output". [1] This section shows how DES uses this block cipher format. In order to specify how any block cipher works, the key expansion mechanism and the round function employed are key. However, this section focuses on the round function.
First, Boneh provides some interesting history on how the development of DES came to be. It started in the 1970s when IBM came up with a cipher designed by Horst Feistel (head of the crypto group at that time) due to customer demands for some form of encryption. Later, in 1973, the government became aware it needed some form of encryption and so the National Bureau of Standards (NBS) made a request "for proposals for a block cipher that will become a federal standard". [1] In 1976, the NBS adopted DES a federal standard but the cipher used a key length of 56 bits and a block length of 64 bits. Boneh noted this as the Achilles heel of DES. [1] Though DES was a very successful block cipher, it was not long before it was broken by exhaustive search, "meaning a machine was able to search all the 2^{56} possible keys and recover a particular challenge key". [1] Later in 2000, the National Institute of Standards (NIST) put out a request regarding proposals for a new block cipher. In 2000, Rijndael was adopted as AES (later section).

A.Feistel Network

As figure 3 shows, DES is built based on a Feistel Network. From a set of d arbitrary functions $f_1, \ldots, f_d$: $\{0,1\}^{2n} \rightarrow \{0,1\}^{2n}$ an invertible function F: $\{0,1\}^{2n} \rightarrow \{0,1\}^{2n}$ is built. The input is 2 blocks of n-bits R (right) and L (left) since a Feistel Network is in general defined from top to bottom. R_0 input gets copied into the L_1 output. But for R_1, the output $f_1(R_0)$ gets XORed with L_0. The property the Feistel Network has for this function is that for all functions $f_1, \ldots, f_d$, F is invertible which is necessary for decryption. This is proven by computing the rounds in reverse order (see figure 4). To encrypt, start from $f_1,\ldots,f_d$, and to decrypt, start from $f_d,\ldots,f_1$.

This is specially useful for hardware implementations where the (E, D) are essentially the same except one has the functions applied in reverse order. The Feistel Network is a "general mechanism for building invertible functions (block ciphers) from arbitrary functions, it's used in many block ciphers but not AES". [1] A theorem by Loby and Rackoff show that shows if you start with a secure PRF, the result is a secure PRP [1]. As an example, a secure PRF used for 3 rounds in a Feistel Netowork would result in a secure PRP (F: $K^3 \times \{0,1\}^{2n} \rightarrow \{0,1\}^{2n}$) where at $f_1(.)$ the PRF function is computed as $F(K_0,R_0)$, at $f_2(.)$ as $F(K_1,R_1)$, and at $f_3(.)$ as $F(K_2,R_2)$ where the keys are all independent. A Feistel Network used a PRF in every round.

B.DES is a 16 round Feistel Network

DES is defined as a 16 round Feistel Network with a set of 16 functions $f_1,\ldots,f_{16}: \{0,1\}^{32} \rightarrow \{0,1\}^{32}$ where $f_i(x) = F(k_i,x)$ and k_i denotes the different round keys and $f_1,\ldots,f_{16}$ are derived from the PRF function F using different round keys. DES therefore acts on blocks of 64 bits (2 x 32). Each k_i is derived from the 56-bit key k, and using these 16 different round keys results in 16 different round functions which give the Feistel Network. To decrypt, the 16 round keys are used in reverse order.

The function $F(k_i,x)$ is described as follows (please refer to figure 6):

i. The two inputs are: x: 32-bit values and k_i: 48-bit key
ii. The input x goes through an expansion "E box". All E does is to move some bits around and replicate others.
iii. The 48-bit output of E is XORed with the round key k_i
iv. The 48 bits are broken into 8 groups of 6 bits.
v. These bits go into "S-boxes"
vi. Each S-box maps 6 bits to 4 bits
vii. The total 32-bit output of the S-boxes is then fed into a last permutation P (has nothing to do with security just required).
viii. Finally, the 32-bit output of the F function is observed.

This illustrates that by using different round keys, different round functions are generated which is how the DES functions are formed.

C.S-boxes

S-boxes are functions that map 6-bits to 4-bits s-box: $\{0,1\}^6 \rightarrow \{0,1\}^4$ implemented as look-up tables. This means the output is on all $2^6 = 64$ possible inputs so that the table contains 64 values each at 4-bits. Figure 7 shows an example with a 4 by 16 tables (64 values). For instance, say the input is 6-bits (011011), the s-box might take the two outer bits (01) and the four inner bits (1101) then do a lookup and output 1001 which corresponds to row 01 and column 1101. In order to defeat attacks on DES, some of the rules for S-boxes require that "no output bit is a linear function of the input bits" and that "S-boxes are 4-to-1 maps (every output has exactly 4 preimages)" since given enough input output pairs means the entire key can be recovered [1]. S-boxes are considered the "smarts" of DES and in general, for block ciphers, they serve the purpose of obscuring the relationship between the cipher text and the key.

D.Exhaustive Search Attacks

The attack's goal is that, given a few input output pairs, $(m_i,c_i = E(k,m_i)$ for i=1,…,n), what is the key k that maps all the message $m_1,\ldots,m_n$ to the cipher texts $c_1,\ldots,c_i$? For three input output pairs, assuming DES to be ideal, since DES has 2^{56} keys, pretending it's a set of 2^{56} invertible, random functions denoted by $\pi_1,\ldots, \pi_{(2^{56})}$: $\{0,1\}^{64} \rightarrow \{0,1\}^{64}$, Boneh shows that for all message cipher text pairs m,c, at most one key maps m to c, c = DES(k,m) with probability $1 - 1/2^{56} = 99.5\%$, so for one pair m,c a request for key k has a unique solution with that probability and this is true for all message cipher text pairs. [1] Boneh proves that the probability

$P(\exists\ k` \neq k: c = DES(k,m) = DES(k`,m)\} \leq$

By union bound:

$$\Sigma_{(k` \in \{0,1\}^{56})} P(DES(k,m0 = DES(k`,m)) \leq [1/ (2^{64})](2^{56}) = 1 / (2^8) = 1/256$$

where $1/(2^{64})$ is for one key k. So the probability that a random permutation of k` collides with k at a given message m, is P(that key k is not unique) = 1/256, so the probability that it is unique is then, $1 - 1/256 = 99.5\%$. This means that just one message cipher text pair (m,c) alone completely determines the key k. For 2 DES pairs $(m_1 , c_1 = DES(k,m_1))$ and $(m_2 , c_2 = DES(k,m_2))$ Boneh explains that the unicity probability is $\approx 1 - 1/(2^{71}) \approx 1$ which says that for two pairs, the probability that there exists only 1 such key is ≈ 1. For AES-128 this probability is $\approx 1 - 1/(2^{128})$. [1]

Exhaustive search tries all possible keys one by one until the correct one is found. In 1997 a company named RSA Security, published their first challenge to break DES. [4] The company published three cipher texts (8 bytes each) for which the plain texts was known and a number of other cipher texts—all encrypted with the same key. The challenge was to decrypt the rest of the cipher texts using the key found from an exhaustive search over the first three pairs. More precisely, the goal was to find $k \in \{0,1\}^{56}$ such that $DES(k,m_i) = c_i$ for i=1,2,3. In 1997 this was done in about 3 months using an internet search—on average, though the key space is 2^{56}, finding the key only takes searching through about half the key space. In 1988 the Electronic Frontier Foundation (EFF) contacted cryptography expert Paul Kocher; he built special hardware, a machine named "Deep Crack" that was able to break DES in only 3 days (cost of $250K vs. the prize of $10K). [1],[4]-[7].

Later in 1999 when RSA published a new challenge; through combined search and deep crack, DES was broken in 22 hours. The point is that DES is completely dead and no longer used. Basically, anyone can recover a 56-bit DES key. The time for cracking goes down further with advancements in hard ware and technology. For instance, in 2006, a project named COCOCABANA used 120 FPGAs and was able to break DES in 7 days. The main lesson Boneh points out in his lectures is that 56-bit key ciphers and completely insecure [1].

E.3-DES

Because DES was used extensively, one method to try to strengthen it was by artificially expanding the key size. So 3DES used three independent keys (key size is 3x56 =168 bits long) and was defined as $K^3 xM \rightarrow M$ as $3E((k_1, k_2, k_3, m)$. But if all the keys are the same, it equals DES. An exhaustive search would take 2^{168} which is "more than all the machines on earth working together for ten years would be able to do". [1] The problem with this construction is speed—just too slow.

F.Meet-in-the-middle attack

In general, anything that is greater than 2^{90} is considered to be sufficiently secure against exhaustive search. [8] The problem with 3DES is that it was too slow but 2DES using 2 keys was insecure even though 2^{112} is large. The problem is that this type of construction is prone to meet-in-the-middle attacks. For say, 10 input output pairs, $M = (m_1, \ldots,m1_0)$ and $C = (c_1,\ldots,c_{10})$, the goal is to find two keys k_1 and k_2 such that $E(k_1, E(k_2, M)) = C$ (the cipher text vector C). The equation can be rewritten as $E(k_2, M) = D(k_1, C)$ which applies the decryption algorithm to both sides using key k_1. This is very important as noted in Boneh's lectures, every time an equation can be written this way with variables on two independent sides, this implies that there is an attack faster than exhaustive search referred to as meet-in-the-middle attack. First, a key must be found that maps M to a specific value * and which also maps C to the same value.

The attack builds a table for all possible k_2 values and encrypts M (see figure 8). The table is then sorted based on the encryption column. The time to build the table is 2^{56} times the sorting time $\log(2^{56})$. This table is then used to construct all possible values "in the forward direction for point *". [1] This gives a total run time of 2^{63} that is well within the reach of exhaustive search and therefore 2DES was not used. The same attack can be used on 3DES using all possible 2^{118} des keys with another attack that only explores 2^{112} possible keys but the run time is $\approx 2^{112}$ is still

too large. Because 3DES is a NIST standard it is used quite a bit but DES, never. [1] Different constructions of DES such as DESX protect against exhaustive search without considering performance penalties. DESX has key size (64 + 56 + 64) with one key as long as they key size and two as long as the block size; it is defined as $EX(k_1, k_2, k_3, m) = k_1 \oplus E(k_2, m \oplus k_3)$ that computes $m \oplus k_3$, then $E(k_2, m \oplus k_3)$, and finally $k_1 \oplus E(k_2, m \oplus k_3)$. The XORs are rather fast but the only problem is that it is susceptible to subtle attacks with best known attack to take time $2^{(64+56)} = 2^{120}$. In general, attacks on DESX take time of block size + key size, but "analysis shows that there is no known exhaustive search attack for this type of construction" [1].

VIII. More sophisticated attacks

The purpose of this section is to give a high-level description of some attacks and emphasize why it is always advised to not design your own block cipher but rather use standards like 3DES or AES.

A.Side-Channel Attacks (Timing Attacks)

These type of attacks measure time to encrypt and decrypt. If you consider something like smart card, say a memory stick, that uses a secret key for some form of authentication and implements a block cipher. An attacker can take the card into a laboratory and then measure very precisely the length of time the card took to encrypt and decrypt. In 1998, Jaffe, Kocher, and Jun found that "if encryption depends on the bits of the secret key, then by measuring the time, the attacker can learn something about the key and be able to completely extract it" [1], [9]. There are a number of examples with timing attacks where simply by measuring very precisely several operations with the encryption algorithm, the key can be completely extracted.

B.Power Attacks

Other type of attacks entail measuring the consumption of power during encryption and encryption of the card (see figure 9). A device that is able to measure the current drawn by card and very precisely graph the current. Because in general smart cards are not too fast, the exact power consumed can be measured "at every clock cycle as the clock executes" [1]. The graph in figure 9 shows the current in the y-axis and the time in the x-axis, where the graph clearly shows the DES is operating. In the start of the graph, the initial permutation is seen, then the exact 16 hills and troughs of the graph reveal the 16 round functions of DES. For such a graph, Boneh notes that "you can read the key bits off one by one just by seeing how much power the card consumed as it was doing the different operations" [1].

In differential power analysis, measurements of the power consumed over iterative runs of the encryption algorithm will reveal any dependency of the bits in the key and the amount of current that is consumed. If such a dependency is found, this means that the key can be completely extracted.

The attacks mentioned in this section were discovered by cryptographer Paul Kocher and his colleagues. These type of attacks encompass per se multicore processors and not just smart cards. CACHE misses can reveal the secret key too. If one core computes (E,D) and there's malicious code on another core, both sharing the same CACHE, then simply by timing the CACHE incurred by the cipher (E,D), the key can be extracted.

C.Fault Attacks

Any errors in any output is enough to extract the key. An attack could be something like overclocking a card to malfunction and get it to output erroneous data. But, "any errors in the last round of the encryption process are enough to produce the secret key" [1] Again, Boneh emphasizes the use of standard libraries like open SSL and others because even implementing crypto primitives means one must ensure against these type of attacks in sub-sections A-C.

D.Linear and Differential Attacks

Any attack that recovers a key in time less than exhaustive search time (2^{56}) is counted as an attack on a block cipher. Linear cryptanalysis can be something like the following. Given a cipher text c = DES(k,m), it looks for a linear relationship between the m, k, and c. In particular, for completely independent m and c, $P(\{m[i_1] \oplus \ldots \oplus m[i_r]\} \oplus \{c[j_1] \oplus \ldots \oplus c[j_k]\} = \{k[l_1] \oplus \ldots \oplus c[l_u]\}) = ½$ exactly. So the XORing of a subset of the message bits XORed with the XOR of a subset of the cipher text bits, the equality of a subset of the XOR of the key bits is exactly ½. However, if there is any bias, the probability is ½ + ε for some small ε. So if there's a linear relationship, the probability becomes , $P(\{m[i_1] \oplus \ldots \oplus m[i_r]\} \oplus \{c[j_1] \oplus \ldots \oplus c[j_k]\} = \{k[l_1] \oplus \ldots \oplus c[l_u]\}) = ½ + \varepsilon$.

The 5ht s-box in DES has this type of relationship because it turns out to be too close to a liner function. "As the 5th s-box propagates through the entire DES circuit it generates a relation of this type with $\varepsilon = (1/2)^{21}$ " [1]. The relation in equation 11 and the following theorem can be used to determine key bits. First, for a given number of $1/\varepsilon^2$ random (m_i, c_i) pairs, (m, DES(k,m)); independent message and the corresponding cipher texts, the equation is used to capture a relationship between the messages and all cipher texts for all the pairs given. If this $m[i_1] \oplus \ldots \oplus m[i_r]\} \oplus \{c[j_1] \oplus \ldots \oplus c[j_k]$ is correct more than half of the time, then the majority of the values for $\{k[l_1] \oplus \ldots \oplus c[l_u]\}$ will be correct, and more than half of the time, the key bits will be correct with probability greater than 99.7% due to the bias. This means that within time $(1/\varepsilon^2)$ the correct XOR of a lot of key bits can be obtained—with $(1/\varepsilon^2)$ input/output pairs, the key bits $\{k[l_1] \oplus \ldots \oplus c[l_u]\}$ can be found in time $\approx 1/\varepsilon^2$. [1]

For a linear attack on DES, $\varepsilon = 1/(2^{21})$ so for $\varepsilon = 1/(2^{42})$ input/output pairs, the key bits $\{k[l_1] \oplus \ldots \oplus c[l_u]\}$ can be

found in time 2^{42} (about 14 key bits). This mean a brute-force attack can recover the reaming 42 bits of the key in 2^{42} for a total time of $\approx 2^{43}$ which is better than 2^{56} exhaustive search but this requires at least three pairs. [1]

IX. Generic attacks on all block ciphers

A.Quantum attacks

All block ciphers are susceptible to this type of attack. To better understand, first, consider a generic search problem where a function f is defined over a large domain x, f: $x \rightarrow \{0,1\}$, that outputs either a 1 or a 0. Say the function outputs a 0 but perhaps for one input, it will output a 1. The problem is then defined as finding the particular input in the $x \in X$ such that f(x) = 1. [1] A classical computer will try all possible inputs that will be linear in the size of the domain so O(|X|), for some generic algorithm. However a computer based on quantum physics can solve this problem faster.

Physicist Richard Feynman is accredited as a pioneer in quantum computing and received a Nobel Prize in 1982 for his theory on Quantum electrodynamics [10][11]. Because Feynman noted the difficulty of simulating quantum experiment on classical computers and concluded that the experiments were computing things too fast for classical computers. Boneh notes that a quantum computer can solve the problem in time $O(\sqrt{X}) = O(|X|^{1/2})$, so that "though the computer knows nothing about the function and is treating it like a black box, it is able to find a point where f(x) = 1 in time $O(|X|^{1/2})$" [1]. And in fact, there seems to be a lot of work and advancement in developing a quantum computer at the Centre for Quantum Computation & Communication Technology in Australia and at the Centre for Quantum Computation in Oxford University with Physicist David Deutsch[12]-[16]

B. Quantum Exhaustive Search

For a given pair (m_i,c_i), where E(k,m) = c, let f be function on k where $k \in K$, then f: ={ 1 ↔ E(k,m) = c and 0 otherwise}. The function is 1 at only one point in the key space K. Grover (quantum algorithm) ran on a quantum circuit can find the key k in time $O(\sqrt{X})$ which completely destroys DES in time 2^{28}, ran on a modern computer, the key for des takes 200 million step and Grover can find it about ≈ 1 ms.

For AES-128 this implies the key can be found in $\approx 2^{64}$. "These days, 2^{64} is insecure since it's well within the reach of exhaustive search." Boneh notes that if anyone where able to built a quantum computer, AES-128 would no longer be secure, in fact, the advice would then be to immediately switch to AES-256 which uses a 256-bit key, since then Grover would run in time 2^{128}. This is one of the reasons why AES was designed with a 256-bit key in mind.

X. AES high-level description with Internals and Crypto properties for aes-128

DES and 3DES have resulted to be too slow for advances in modern hardware. As mentioned earlier the NIST started a new project for a new block cipher and in 2000 (published in 2001) it adopted Rijndael as "AES". [21] Rijndael is the original name given to AES by the Belgiam cryptographers that developed it, Joan Daemen and Vincent Rijmen. [1] [17][18]. The original Rijndael algorithm allowed for flexibility in block size and key length, but these were not adopted as part of the standard. [22] AES has a block size of 128-bits and three possible key sizes: 128, 192, and 256-bit, with corresponding number of rounds is 10, 12, and 14 respectively (see figure 18). [21] The longer key size makes it more secure but slower since this entails more rounds to compute. Therefore AES-128 is the fastest and AES-256 is the slowest.
AES is a completely different construction from a Feistel Network—a Substitution Permutation Network. In contrast to a Feistel Network where only half the bits are changed from round to round, in a substitution permutation network, all the bits must be changed in every round (see figure 10 for more details). [1] The following is a generic construction of AES where the current input is a array composed of 4 rows and "Nk" columns and depending on the key size, Nk = 4 (for 128-bit), 6 (for 192-bit), or 8 (for 256-bit). Then, "the input key is expanded into an array of 44, 52, or 60 words of 32-bits each" (see figure 19) [21] The input is block of 128-bits is then copied to state array with Nb number of columns. The number of rounds depends on the key size and each round is a "repetition of functions that perform a transformation over a state array / matrix" [21] The round function is composed of four main functions which are a permutation function AddRoundKey and three subfunctions (ByteSub, ShiftRow, and MixColumn). The overall process is as follows:

i. The current state of the input is XORed with first round key k_1 (AddRoundKey)
ii. Substitution Layer: the output from step i. goes through substitution layer where blocks of state are replaced with other blocks of state based on a substitution table (the round function includes permutations other operations along with the subfunctions: ByteSub, ShiftRow, MixColumn, and AddRoundKey)
iii. Permutation Layer: the output then goes through this layer where bits are shifted around and permuted.
iv. Repeat (i-iii) in a loop and XOR with next round key k_2 until the final round is reached and XORed with the last round key, then the final output is the cipher text.

Every single step in this network is reversible which is necessary for decryption. The way decryption works is that the output is taken and then every step in the network is applied in reverse order.

A. AES-128 Schematic

Some of the specifics of AES (see figure 11):

i. It operates on 128-bit block size (16 bytes) and computes a total of 10 rounds
ii. These 16 bytes are written as a 4 x 4 matrix where each cell in the matrix contains 1 byte
iii. Then starting with the first round, this matrix is XORed with the first round key k_1 (using AddRoundKey function)
iv. Apply the round function F comprised of one permutation function and three substitution functions on the state: (1) ByteSub, (2) ShiftRow, and (3)MixColumn, (4)AddRoundKey
v. Loop: repeat now with XOR of next round key, apply F, and so forth. This is done ten times.
vi. In the last round (number ten) only (1)ByteSub, (2)ShiftRow are applied.
vii. Finally, this output is XORed with the last round key AddRoundKey and the output is produced.

In every step, the 4 x 4 matrix is kept so that the final output is also 4 x 4 (16 bytes = 128-bits). The round keys ($k_0,\ldots,k_{10}$) of course, come from a key expansion mechanism that maps 16-bytes to 176 bytes (11 keys at 16 bytes each). These keys are also 4 x 4 matrices that get XORed with the current state of the input. This concludes a high-level description of how AES-128 works and the following subsections give a high level description as well as internal specifics of the three functions that comprise the round function F.

B.ByteSub

Byte substitution is "a 1 byte s-box that contains a 256-byte table that applies the s-box to every byte in the current state" [1] The current state being the 4 x 4 table (say the "A" table) and the s-box is applied to each element in the table, $\forall$ i,j: s[A[i,j]] $\rightarrow$ A[i,j]. The current cell is used as an index into the lookup table and the value from the lookup table (the next state) is what is outputed. It performs a 'byte-by-byte' substitution of the state" [21]. Here "each byte is considered an element in GF(2^8); a 16 x 16 table, called S-box contains all the possible 256 elements; a byte's four leftmost bits serve as a row index, and rightmost ones as a column index; together, the S-Box and above mapping define a 1-to-1 function f : GF(2^8); and each byte B in the state matrix is substituted with f(B)" [24]

A field is defined as an algebraic object with * and + operations. Using * and + all elements in the field must form commutative groups with specific identity (0 or 1) and inverse (-a or a^{-1}) for each operator. Also, for all elements a, b, c, the distributive identity a*(b + c) = (a*b) + (a*c) needs to hold. "It turns out that for any prime integer p and any integer n greater than or equal to 1, there is a unique field with p^n elements in it, denoted GF(p^n)" [24] Difficulty arises in finding the inverse of an element. Neal Wagner's book on the Laws of Cryptography describes the following regarding AES and GF(2^8) :

"AES works primarily with *bytes* (8 bits), represented from the right as":

$$b_7b_6b_5b_4b_3b_2b_1b_0.$$

"The 8-bit elements of the field are regarded as *polynomials* with coefficients in the field $\mathbf{Z}_2$:"

$$b_7x^7 + b_6x^6 + b_5x^5 + b_4x^4 + b_3x^3 + b_2x^2 + b_1x^1 + b_0.$$

"The field elements will be denoted by their sequence of bits, using two hex digits." For further details please see reference listed [24].

Computing the cells of the S-box takes three steps. First, the cells are numbered row by row in ascending order. Second, "each cell's number is substituted with its multiplicative inverse over GF(2^8), and lastly, the cell's bits go through a transformation: bi' = b[(i+4)mod8] + b[(i+5)mod8] + b[(i+6)mod8] + b[(i+7)mod8] + ci, then bi' = new bit value, ci = the I'th bit of {11000110}." [24] The key properties of the S-box that helps protect against attacks are that there cannot be fixed points (nothing of S(a) = a), and no complements of a fixed points, the S-box cannot be self invertible meaning nothing of the form S(a) =InverseS(a).

Decryption circuit performs same computations as ByteSub except that the inverse S-box is used. This is "computed by applying the inverse affine transformation.. then substituting with the multiplicative inverse, of the cell's value in the S-Box. The inverse transformation is: bi' = b[(i+2)mod8] + b[(i+5)mod8] + b[(i+7)mod8] + di then, bi' = new bit value, di = the I'th bit of {00000101}." [24] Please see figures 24 – 27.

C.ShiftRows

This function performs a cyclic shift on each one of the rows—basically a permutation. It shifts the last three rows of the state by different offsets. [21] In figure 13, it is seen that the second row (from top to bottom) has a left shift by one position, the third row has left shift by 2 positions, and the last row has a left shift by 3 positions. It is presumably easy to code. The main effect is diffusion. The decryption circuit uses ShiftRows except that it performs right shift.

D.Mix-Columns

This function applies a linear transformation to each column (32 bits). A specified matrix used in the function multiplies each column that then becomes the next column. This is illustrated in figure 14. More specifically, all columns of the state have their data mixed, independent of each other by applying arithmetic over a Galois Field of GF(2^8) where "each column is treated as degree 3 polynomial over the GF, multiplied by the fixed polynomial $a(x)=\{03\}x^3+\{01\}x^2+\{01\}x+\{02\}$ modulo x^4+1 which was selected so that the multiplication/transformation is

invertible (coefficients multiplication is the GF(2^8) multiplication" . [21][24]

Key properties of this function are that the "transformation is a linear code with a maximal distance between code words and combined with ShiftRows, after several rounds all output bits depend on all input bits." [24] The decryption circuit uses Mix-Colums with the inverse of a(x) as " $a^{-1}(x)=\{0b\}x^3+\{0d\}x^3+\{09\}x+\{0e\}$" [24] See figures 28 and 29.

E.AES in Hardware

A generic fact of AES is that it permits for no-pre-computation for a small implementation on restricted environment, however this would be the slowest because all the functions, everything would have to be computed on the fly. On the other hand, given a lot of space to store a lot of code, functions (1) to (3) can be pre-computed and yield very good performance because the only computations would then be reduced to table lookups and XORs. So there is a trade-off with code size and performance (see figure 15).

Boneh mentions one example of AES in Javascript. Basically, an AES library (6.4 KB) is sent to a browser; it has no pre-computed tables and the code is fairly small. The way to get the best performance is that the browser pre-computes the tables but they are stored on the client machine (with presumably a lot of memory). Then the machine can encrypt. This gives best performance when sending an "implementation of AES over a network" [1]

Moreover, AES has become such a widely used block cipher that it is expected to be used when companies develop products. Intel started putting AES into the processor since Westmere. [1][19][20] Intel added special instructions that help to make AES faster. More specifically, the two instructions in Westmere are:

i. **aesenc**: implements one round function of AES (F with k_0), it applies the three noted functions (ByteSub, etc.) and XORs them with the round key.

ii. **aesenclast**: implements the last round key of AES.

These instructions are called using 128-bit registers that correspond to the state of AES. For 128-bit registers xmm1=state, xmm2=round key, the call would look something like:

aesenc xmm1, xmm2;

This runs one round of AES and places the result in state register xmm1. To implement the entire AES, you would call aesenc nine times and aesenclast one time—these ten instructions are the entire implementation of AES. The claim is that AES can actually run 14 times faster (in contrast to something like Open SSL on hardware without these instructions) because the implementation is done inside the processor. However, other hardware products like those from AMD also use these instructions. [1]

F.The Security of AES

Throughout the entire time that AES has been studied, there are only two attacks known to date. The first is on AES-128, "Best key recovery attack" which is four times faster than exhaustive search and that treats AES as if it had a 126-bit key. However, because 2^{126} is too large to compute on, it does not affect the security of AES. The second attack, considered to be more significant is "Related key attack" on AES-256. This is because a weakness in the key expansion mechanism of AES was uncovered which gives way to related key type of attacks.

The attack basically considers that $\sim 2^{99}$ input/output pairs are obtained from four related keys. The keys are very closely related keys such that key1 ≈ key2 except for some bits and key3 ≈ key1 except for bits and the same for key4 so that their hamming distances (the number of positions at which the corresponding bits are different) is quite short. This second is not really a problem because 2^{99} is still to large to compute on and though it is much better than exhaustive search of 2^{256}, in practice, keys are chosen at random so that there are no related keys. The only way this attack applies is if there are related keys.

XI. AES PSEUDO-CODE

Figure 16 shows a flow chart for the AES and the following is the pseudo-code taken from the Hebrew Institute of Computer Science [21]. Figure 16 shows the execution and function calls for one round, the 9th round, and the last round. In the codes, Nb denotes block size, Nr number of rounds and Nk the key length.

A. Pseudo-code: Encryption and Decryption

```
Cipher(byte in[4*Nb], byte out[4*Nb], word w[Nb*(Nr+1))
Begin
    by state[4,Nb]
    state = in
    AddRoundKey(state, w[0,Nb-1])

    for round=1 to Nr-1
        SubBytes(state)
        ShiftRows(state)
        MixColums(state)
        AddRoundKey(state, w[round*Nb, round+1)*Nb-1])
    end for
SubBytes(state)
ShiftRows(state)
AddRoundKey(state, w[Nr*Nb,(Nr+1)*Nb-1])

Out = state
end
```

As it was explained earlier, Nb:= number of columns depends on the key size, for AES-128, Nb is 4. Only the function AddRoundKey uses the key. Figure 21 shows the decryption flow chart (Inverse Cipher). In this case, as illustrated below, the expanded key is applied in reverse order. Because the functions are reversible, their inverse is

used in decryption—the output is taken and applied in reverse order. Figure 17 gives a description of the cipher.

The AddRound key function "uses four different words from the expanded key array, each column in the state matrix is XORed with a different word, the function is basically the heart of the encryption, the properties of other functions are permanent and known to all" [21] See figure 22 for an illustration.

```
InvCipher(byte in[4*Nb], byte out[4*Nb],wordw[Nb *(Nr+1))
Begin
    by state[4,Nb]
    state = in
    AddRoundKey(state, w[Nr*Nb, (Nr+1)*Nb-1])

    for round=1 to Nr-1
         InvShiftRows(state)
         InvSubBytes(state)
         AddRoundKey(state, w[round*Nb, round+1)*Nb-1])
         InvMixColums(state)
    end for

InvShiftRows(state)
InvSubBytes(state)
AddRoundKey(state, w[0,Nb-1])
Out = state
end
```

In decryption, the inverse of the function AddRoudKey is the same except the key is used in reverse order and computation is: $(A \oplus B) \oplus B = A$.

B.Key expansion

The key expansion function for AES, takes an Nb word as the input key and expands it to Nb * (Nr + 1) words in the form of a linear array. An Nb word round key is used in the first call to AddRoundKey and for the subsequent Nr rounds of AES. " The key is first copied into the first Nb words, the remainder of the expanded key is filled Nb words at a time." [24]

```
KeyExpansion(byte key[16], word w[44])
{
    word temp;
    for( i = 0; i < 4; i++)
         w[i] = (key[4*i], key[4*i + 1], key[4*i + 3],
         key[4*i + 3])
    for(i = 4; i < 44; i++)
    {
         temp = w[i-1];
         if(i mod 4 == 0)
              temp = SubWord(RotWord(temp) XOR Rcon[i/4];
         w[i] = w[i – 4] XOR temp;
    }

}
```

In the code above, RotWord does a "one byte circular left shift on a word. For instance, consider RotWord[b0,b1,b2,b3] = [b1,b2,b3,b0]; SubWord performs a byte substitution on each byte of input word using the S-box; and SubWord(RotWord(temp)) is XORed with Rcon[j], the round constant." [24] The round constant is defined as word where the three least significant bytes are zero. The value in each round is different and it is defined as "Rcon[j] = (RC[j],0,0,0) where where RC[1] = 1, RC[j] = 2 * RC[j-1] and multiplication is defined over $GF(2^8)$" [24] Figure 30 and 31 give an example of the RC[j] hex values and round constant for AES-128.

The requirements behind the key expansion are that it had to satisfy the property of non-linearity which "prohibits the full determination of round key differences from cipher key differences", diffusion so that "each cipher key bit affects many round key bits", and the round constant which " eliminates symmetry or similarity between the way round keys are generated." [24] This makes sense because knowing any number less than Nk consecutive bits of a round key or cipher makes it very hard to recover the remainder of the unknown bits.

C.Mathematical Review

The following specifics are also taken from the Hebrew Institute of Computer Science:

- Arithmetic operations on bytes must to work in a finite field and treat each byte as an element.
- $GF(2^8)$ - Finite field containing 256 elements.
- Each element is a polynomial of degree 7 over Z2, so that an element is defined by 8 binary values – a byte.
- The order is such that MSB is the highest degree's coefficient.
- Addition – polynomial addition, over Z_2. May be implemented using XOR.
- Multiplication – polynomial multiplication, over Z_2, modulo irreducible polynomial:

$$X^8 + X^4 + X^3 + X + 1$$

May be implemented using repeated use of left shift and XOR.

D.Core algorithms applied separately in applications

At a more intricate level of the AES internal algorithms, there is a lot of shifting, XORing, and a lot of complex arithmetic operations over $GF(2^8)$ as well as transformations and diffusion. However, no specific core sorting algorithm was identified. In addition, a simple C code implementation of AES-128 was analyzed, confirming no sorting algorithm. However, in the midst of this, it was noted that AES is used in conjunction with a core algorithm like merge-sort for sorting encrypted databases where and with more advanced algorithms for hashing like AES-hash. [28]-[31]

XII. CONCLUSION

At the moment, AES appears to be the block cipher of choice with a lot of support. Despite the threat of Quantum Cryptanalysis, the 256-bit key version projects a good long term success of AES, at least until quantum computers are able to break it, or any number of vulnerabilities come to surface. All in all, details of the algorithmic construction are

very complex, and the NIST publication notes that there are no known semi-weak/weak keys identified, which implies any of the key sizes may be used freely without restriction.

There are also implementation options for developers which may offer performance advantages. For further reference, please see the NIST AES publication which provides a wonderful and very detailed description of AES including the mathematical preliminaries, algorithmic specifications, arithmetic computations over GF(2^8), and the decryption circuit. [22] "Now-a-days AES is being used for almost all encryption applications all around the world." [31] The last two figures show performance numbers.

XIII. APPENDICES AND REFERENCES

[1] https://class.coursera.org/crypto-011

[2] R. Kadir et al. (2011) *Randomness Analysis of Pseudorandom Bit Sequences* [online]. Available PDF: http://www.ipcsit.com/vol2/73-B254.pdf

[3] http://en.wikipedia.org/wiki/Block_cipher

[4] http://en.wikipedia.org/wiki/DES_Challenges

[5] http://en.wikipedia.org/wiki/EFF_DES_cracker

[6] https://w2.eff.org/Privacy/Crypto/Crypto_misc/DESCracker/HTML/19980716_eff_des_faq.html

[7] http://www.cryptography.com/technology/applied-research/research-efforts/des-key-search/des-key-search-photos.html

[8] http://mathworld.wolfram.com/PolynomialTime.html

[9] http://www.cryptography.com/public/pdf/DPA.pdf

[10] http://www.xootic.nl/magazine/jul-2003/west.pdf

[11] R. Bone et al. (n.d.). *A Brief History of Quantum Computing* [online]. Available PDF: http://www.doc.ic.ac.uk/~nd/surprise_97/journal/vol4/spb3/

[12] https://www.youtube.com/watch?v=cugu4iW4W54

[13] http://en.wikipedia.org/wiki/Grover%27s_algorithm

[14] http://en.wikipedia.org/wiki/Quantum_algorithm

[15] http://en.wikipedia.org/wiki/Quantum_circuit

[16] http://demonstrations.wolfram.com/QuantumCircuitImplementingGroversSearchAlgorithm/

[17] http://en.wikipedia.org/wiki/Advanced_Encryption_Standard

[18] http://embeddedsw.net/Cipher_Reference_Home.html

[19] http://en.wikipedia.org/wiki/Westmere_%28microarchitecture%29

[20] S. Gueron. (2010) *Advanced Encryption Standard Instruction Set* [online]. Available PDF: http://www.intel.com/content/dam/www/public/us/en/documents/white-papers/aes-instructions-set-white-paper.pdf

[21] S. Kipnis (2003). *System and Networking Security Course Homepage* [online]. Available: http://www.cs.huji.ac.il/~sans/

[22] National Institutre of Standars (2001) . *FIPS 197, Advanced Encryption Standard (AES)* [online]. Available PDF: http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf

[23] http://www.cs.ucf.edu/~wocjan/Teaching/COT5937-Fall2013/stream_ciphers.pdf

[24] N.R. Wagner. (2001). *The Laws of Cryptography* [online]. Available PDF: http://www.cs.utsa.edu/~wagner/laws/FFM.html

[25] https://www.math.washington.edu/~morrow/336_12/papers/juan.pdf

[26] http://en.wikipedia.org/wiki/P_versus_NP_problem

[27] T.H. Coremn et al. (2009). *Introduction to Algorithms, Third Edition* [online]. Available PDF: http://mitpress.mit.edu/sites/default/files/titles/content/9780262033848_pre_0001.pdf

[28] http://csrc.nist.gov/groups/ST/toolkit/BCM/documents/

proposedmodes/aes-hash/aeshash.pdf

[29] Texas Instruments. (2009). *AES -128 – A C implementation for Encryption and Decryption* [online]. Available PDF: http://www.ti.com/lit/an/slaa397a/slaa397a.pdf

[30] K. Fleming et al. (2009). *High-throughput Pipelined Mergesort* [online]. Available PDF: http://people.csail.mit.edu/kfleming/papers/mergesort_08.pdf

[31] N. PK (n.d.). *Advanced Encryption Standard implementation in C* [online]. Available PDF: http://comp.ist.utl.pt/ec csc/Code/Ciphers/AES_Encrypt.cpp