Report Master Colloquium A

Giorgio Sidari

Uncertainty of Online Transaction Systems

Sommario

Uncertainty of Online Transaction Systems

Introduction

Online payment systems are at the core of digital commerce, facilitating the seamless exchange of money between buyers and sellers. While these systems have evolved to become more efficient, their vulnerabilities remain a critical concern, as attackers continually exploit weaknesses in the ecosystem.

Common attacks include SQL injection, cross-site scripting, distributed guessing attacks, man-in-the-middle attacks, and vulnerabilities from weak data storage mechanisms. These threats exploit flaws in databases, insecure channels, and outdated encryption techniques.

To counter these vulnerabilities, modern systems employ advanced security measures:

1. **Cryptographic Algorithms**: AES for strong symmetric encryption, RSA for secure key exchanges, and SHA-512 for data integrity.
2. **EMV Technology**: Chip-based cards with tamper-resistant encryption.
3. **3D Secure Protocols**: Additional authentication layers like Verified by Visa and SecureCode.
4. **Steganography and Visual Cryptography**: Techniques to hide and distribute sensitive data securely.

Core research problem

Online payment systems are increasingly targeted by sophisticated cyberattacks, such as distributed guessing attacks and data breaches, due to their reliance on inconsistent security implementations and outdated cryptographic methods. Despite advances in encryption algorithms, challenges remain in balancing security, computational efficiency, and adaptability to evolving threats. These gaps create a pressing need for innovations that ensure robust security while maintaining system efficiency.

Overarching goal

This research aims to enhance the security and efficiency of online payment systems by optimizing the AES encryption algorithm using an alternative S-box (Anubis) and integrating a unified cryptographic architecture. These innovations seek to mitigate vulnerabilities, such as distributed guessing attacks, while reducing hardware utilization, power consumption, and execution time. The ultimate objective is to develop a practical and adaptable framework for securing sensitive financial transactions.

AES encryption

AES is a ***symmetric block cipher*** designed for electronic data security, operating iteratively on 128-bit data blocks. Its simplicity stems from its byte-based structure, which makes implementation straightforward. The encryption process involves multiple transformation rounds, with the number of rounds (10, 12, or 14) determined by the key length (128, 192, or 256 bits). The figure below illustrates both data block encryption process and key expansion process. Each round includes four main data transformations: SubBytes, ShiftRows, MixColumns, and AddRoundKey. Simultaneously, the key expansion process is carried out in three steps: KeySubWord, KeyRotWord, and KeyXOR.

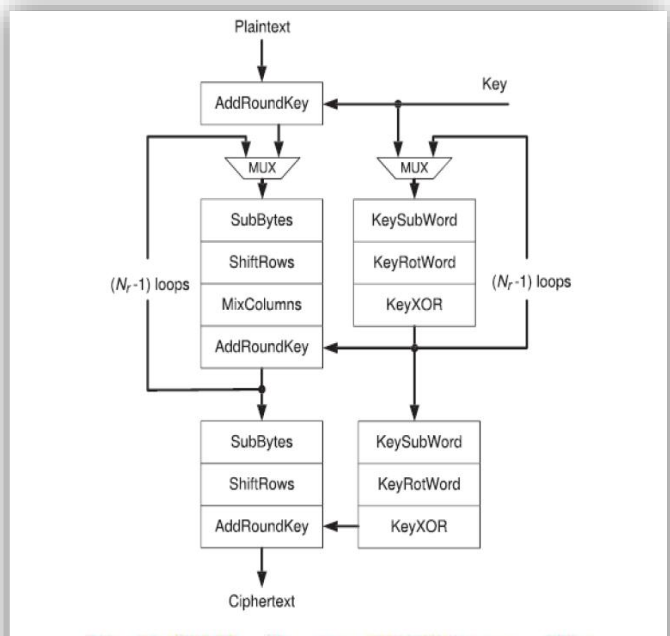


Fig. 1. Block diagram of AES encryption.

The four AES steps are listed below:

- **Sub bytes**: It is a non-linear substitution of bytes that operates independently on each byte of the State using a substitution table (S box). This S-box which is invertible and is constructed by first taking the multiplicative inverse in the *Galois Field* (2^8) with irreducible polynomial **m(x) = x^8 + x^4 + x^3 + x + 1**. The element {00} is mapped to itself. Then affine transformation is applied (over GF (2)).

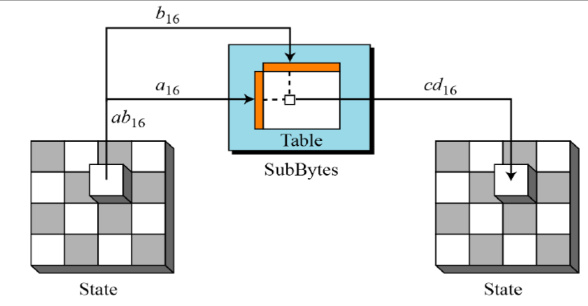


Fig 1.1: Substitution

- **Shift Rows**: In this transformation, the first row of the state array kept as it is. But the bytes in the second, third, and forth rows are cyclically shifted by one, two, and three bytes to the left, respectively.

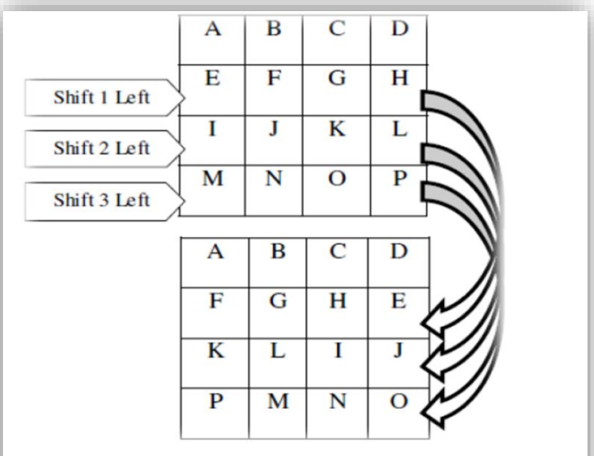


Fig 1.2: Shifting Rows

- **Mix Columns**: here columns of the state are considered as polynomials over GF (2^8) and multiplied by modulo x^4 + 1 with a fixed polynomial c(x), given by:

  c(x)={03}x^3 + {01}x^2 + {01}x + {02}.

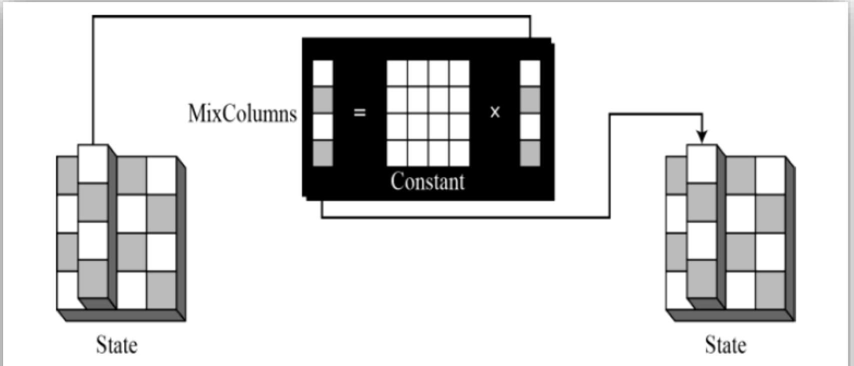


Fig 1.3: Mixing

- **Add Roundkey**: Using a bitwise exclusive-or (*XOR*) operation, a round key is added to the state array. These round keys are computed in the key expansion process.

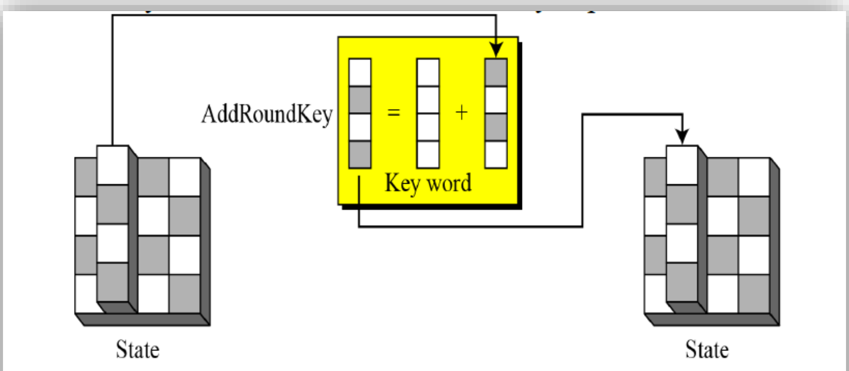


Fig 1.4: Adding Key

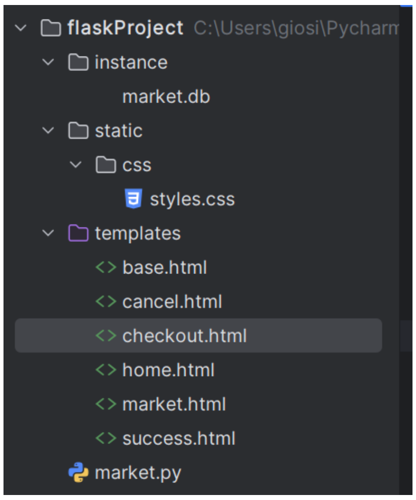
Similarly, there are three steps in each key expansion round:

- **KeySubWord**: it takes a four-byte input word and output word is produced by substituting each byte in the input to another byte according to the S-box.
- **KeyRotWord**: represents the second step of expansion. It takes a word [a3, a2, a1, a0], performs a cyclic permutation, and returns the word [a2, a1, a0, a3] as output.
- **KeyXOR**: In key XOR function of key expansion, Every word w[i] is equal to the XOR of the previous word, w [i-1], and the word Nk positions earlier, w [i –Nk]. Nk equals 4, 6 or 8 for the key lengths of 128, 192 or 256 bits, respectively.

Prototype with Stripe

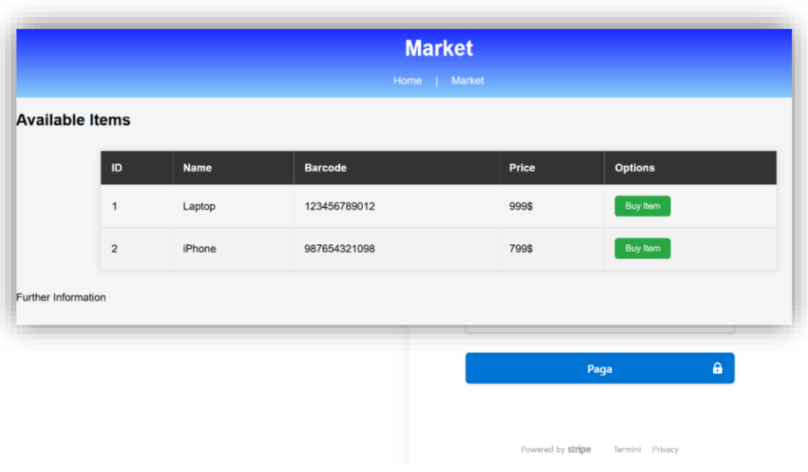
A prototype website was developed using REST (***RE****presentational* ***S****tate* ***T****ransfer*) APIs and the Stripe library for test online transactions. Stripe, a widely used payment platform, incorporates AES encryption.

- **Frontend**: Scripts (e.g., styles.css, base.html) ensure consistent page layouts.
- **Backend**: Implemented in **Python** with **Flask** (micro Web framework). Scripts like *market.py* manage the database, market page, and checkout session. Transactions redirect users to test checkout pages, confirming successful or failed payments via predefined responses.



```
flaskProject C:\Users\giosi\Pycharm
    instance
        market.db
    static
        css
            styles.css
    templates
        base.html
        cancel.html
        checkout.html
        home.html
        market.html
        success.html
    market.py
```

After pressing "buy item" button, the page will redirect to the checkout page.



Market

Home | Market

Available Items

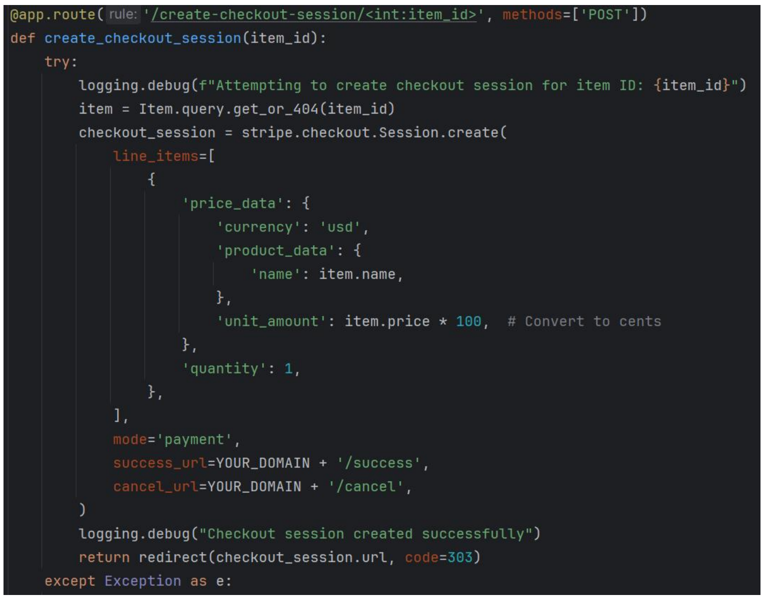
ID	Name	Barcode	Price	Options
1	Laptop	123456789012	999$	Buy Item
2	iPhone	987654321098	799$	Buy Item

Further Information

Here it is the checkout in test mode, it is possible to fill it with card prototypes available in Stripe documentation. After clicking on Buy ("Paga" in Italian) the session will move to page *success.html*, if credentials are correct, to *cancel.html* if the payment was not completed.

Encryption process inside the prototype

The **Stripe Python library** does not directly implement AES encryption algorithm in the code exposed to the user. Instead, Stripe relies on secure HTTPS communication (TLS) to encrypt data between your application and Stripe servers. AES encryption is used internally by Stripe for storing sensitive information, but these details are abstracted away and not part of the Python library's source code.

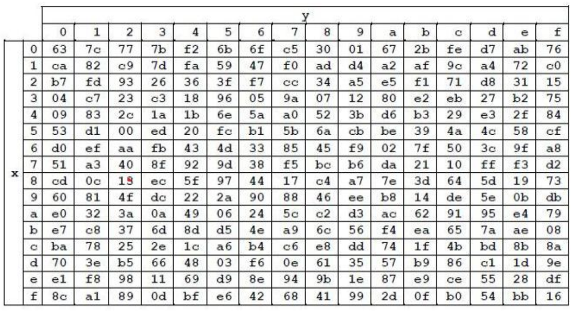


```python
@app.route( rule: '/create-checkout-session/<int:item_id>', methods=['POST'])
def create_checkout_session(item_id):
    try:
        logging.debug(f"Attempting to create checkout session for item ID: {item_id}")
        item = Item.query.get_or_404(item_id)
        checkout_session = stripe.checkout.Session.create(
            line_items=[
                {
                    'price_data': {
                        'currency': 'usd',
                        'product_data': {
                            'name': item.name,
                        },
                        'unit_amount': item.price * 100,  # Convert to cents
                    },
                    'quantity': 1,
                },
            ],
            mode='payment',
            success_url=YOUR_DOMAIN + '/success',
            cancel_url=YOUR_DOMAIN + '/cancel',
        )
        logging.debug("Checkout session created successfully")
        return redirect(checkout_session.url, code=303)
    except Exception as e:
```

When you create a checkout session using the Stripe Python SDK, it leverages HTTPS to ensure secure communication.

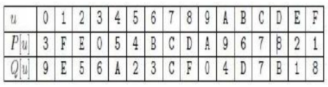
AES OPTIMIZATION

The proposed system compares AES encryption with two different S-Boxes. S-box with 256 byte lookup table (**Rijndael S-Box**) and AES with 16 byte S-Box (**Anubis S-Box**). Anubis encrypts blocks of 128 bits, which are internally represented as 16 bytes arranged in a 4-by-4 matrix.



		y															
		0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
x	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

Fig. 2. Rijndael S-box(256 byte)



u	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
$P[u]$	3	F	E	0	5	4	B	C	D	A	9	6	7	B	2	1
$Q[u]$	9	E	5	6	A	2	3	C	F	0	4	D	7	B	1	8

Fig. 3. AnuBis S-box(16 byte)

The most important differences between Anubis and Rijndael are the following:

- **Involutional structure**: In Anubis, all steps are involutions. This implies that the implementation of the decryption operation can be the same as the encryption operation, except for a change in the key schedule (while in Rijndael encrypting and decrypting shift rows step can bring to different values). This should in principle reduce the code size or area in software, respectively hardware applications that implement both encryption and decryption.
- **Different S-box**: Anubis S-box is generated in a pseudo-random way. In fact, the polynomial expansion of the S-box is certainly more involved. The disadvantage is a more complex hardware implementation.
- **more complex key scheduling**: From Anubis lookup table, element u represent input byte. In Anubis S-box, during SubByte transformation, the first four bytes of input (represented by **u**) are substituted by corresponding **p[u]** and last four bytes(represented by **u**) are substituted by corresponding **Q[u].**

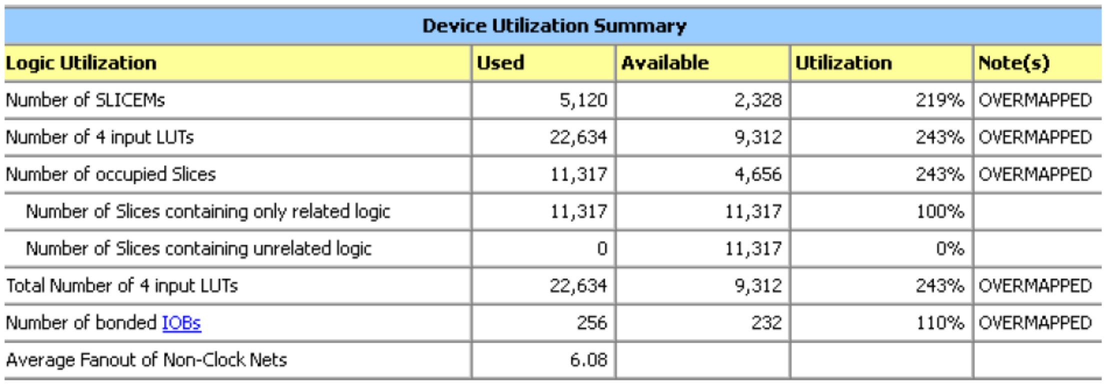
Measurements and results

An efficient implementation of the S-box is the main challenge for compact or highspeed hardware implementations of Rijndael. A XILINX Spartan-3E XC3S500E-4FG320 FPGA is used to compare AES with Rijndael S-box and AES with Anubis S-box. The result conveys that the performance of the proposed work has been improved. Optimized and synthesized VHDL code is used for AES encryption.

Xilinx ISE 13.2i software is used for VHDL code implementation and Modelsim simulator is used for the simulation. In this, AES encryption using both lookup tables implemented and it takes as input 128-bit data block and 128-bit key.

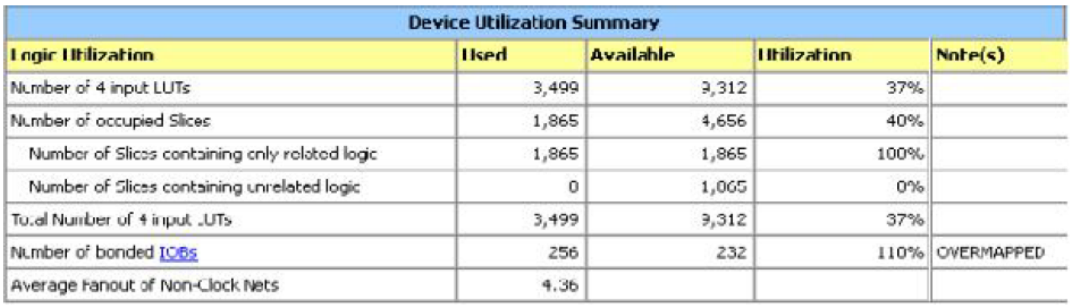
Device utilization summary of both S-boxes are shown below and from that, several facts are inferred which reveals the benefits of Anubis S-box. Here “overmapped” represents extra hardware that is needed for the implementation of AES algorithm.

Table 1. Device utilization summary of AES with 256 byte S-Box



Device Utilization Summary				
Logic Utilization	**Used**	**Available**	**Utilization**	**Note(s)**
Number of SLICEMs	5,120	2,328	219%	OVERMAPPED
Number of 4 input LUTs	22,634	9,312	243%	OVERMAPPED
Number of occupied Slices	11,317	4,656	243%	OVERMAPPED
Number of Slices containing only related logic	11,317	11,317	100%	
Number of Slices containing unrelated logic	0	11,317	0%	
Total Number of 4 input LUTs	22,634	9,312	243%	OVERMAPPED
Number of bonded IOBs	256	232	110%	OVERMAPPED
Average Fanout of Non-Clock Nets	6.08			

Table 2. Device utilization summary of AES with 16 byte S-Box



Device Utilization Summary				
Logic Utilization	**Used**	**Available**	**Utilization**	**Note(s)**
Number of 4 input LUTs	3,499	9,312	37%	
Number of occupied Slices	1,865	4,656	40%	
Number of Slices containing only related logic	1,865	1,865	100%	
Number of Slices containing unrelated logic	0	1,065	0%	
Total Number of 4 input LUTs	3,499	9,312	37%	
Number of bonded IOBs	256	232	110%	OVERMAPPED
Average Fanout of Non-Clock Nets	4.36			

As shown, the performances of both the systems are compared. From the observations, we can infer that using Anubis S-box results in usage of less memory space, reduced hardware utilization, power consumption and execution time for encryption process.

Introduction to RSA

RSA is an asymmetric algorithm used in cryptography (*Asymmetric encryption is the process of using a public key from a public/private key pair to encrypt plaintext, and then using the corresponding private key to decrypt the ciphertext*), mainly based on the definition of totient and on modulus operations.

R.S.A. is named after its three inventors: Ronald Rivest, Adi Shamir e Leonard Adleman

Euler's Totient is a term that denotes all the integers smaller than **n** that have no common factors with **n**. Another way of stating this is how many numbers smaller than **n** are co-prime with **n** (*note: two numbers are co-prime if they have no common factors*). For example, if **n = 8** , the answer would be 3, 5, and 7. 1 is also included as a special case, thus the totient of 8 would be 4. For a prime number instead, all smaller positive integers are co-prime with it. Put more generally, the totient of any prime number **p** is **p-1**. Moreover, if **k=m*n** (both **m** and **n** prime numbers), the totient of that **k** is **(m-1)(n-1)**.

In addition, the modulus operation is a key mathematical component of RSA algorithm. To use it, given a number **A** to which we would like to calculate the modulus **N**, $A \bmod (N)$, simply divide **A** by **N** and return the remainder.

RSA algorithm

Step 1: Generate two large random primes, **p** and **q**, of approximately equal size such that their product $n = p*q$ is of the required bit length (such as 2048 bits, 4096 bits, etc.)

Step 2: Let $n = p*q$ and let $m = (p\text{-}1)*(q\text{-}1)$

Step 3: Choose a small number **e**, co-prime to **m**.

Step 4: Find **d**, such that $d*e \bmod m \equiv 1$. And that is it, now just publish **e** and **n** as the public key. Keep **d** and **n** as the *secret key*.

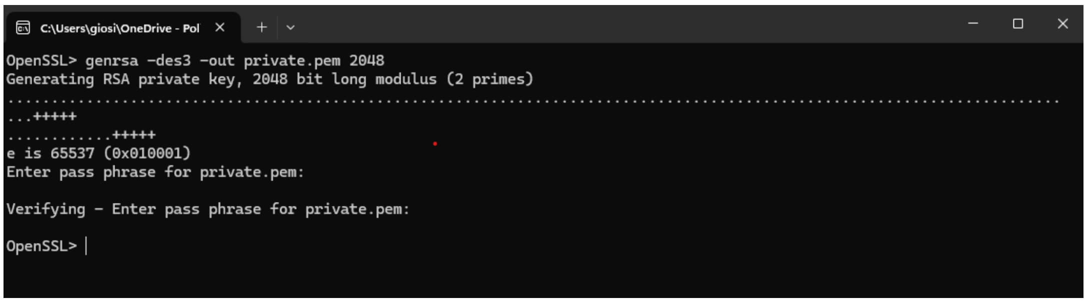
How can I generate a public and private key pair using RSA?

One of the most popular commands in SSL to create, convert, manage the SSL Certificates is OpenSSL.

You can generate a public and private RSA key pair like this:

1) start OpenSSL and write this line:

   genrsa -des3 -out privateKeyTest.pem 2048

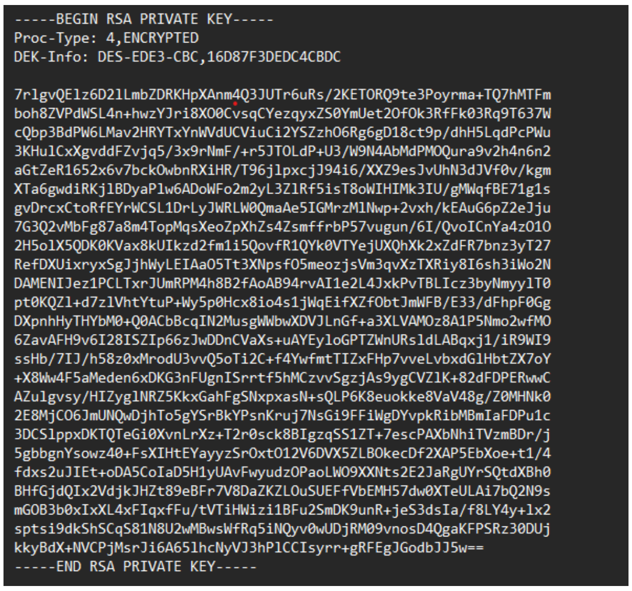


```
OpenSSL> genrsa -des3 -out private.pem 2048
Generating RSA private key, 2048 bit long modulus (2 primes)
..........................................................................................................
...+++++
...........+++++
e is 65537 (0x010001)
Enter pass phrase for private.pem:

Verifying - Enter pass phrase for private.pem:

OpenSSL> |
```

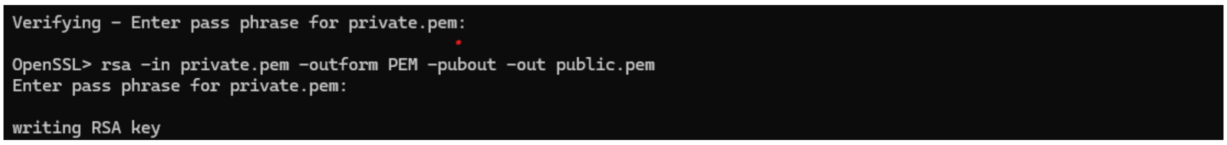
Now, opening the file private.pem, the output should be similar to the following:



```
-----BEGIN RSA PRIVATE KEY-----
Proc-Type: 4,ENCRYPTED
DEK-Info: DES-EDE3-CBC,16D87F3DEDC4CBDC

7rlgvQElz6D2lLmbZDRKHpXAnm4Q3JUTr6uRs/2KETORQ9te3Poyrma+TQ7hMTFm
boh8ZVPdWSL4n+hwzYJri8XO0CvsqCYezqyxZS0YmUet2OfOk3RfFk03Rq9T637W
cQbp3BdPW6LMav2HRYTxYnWVdUCViuCi2YSZzhO6Rg6gD18ct9p/dhH5LqdPcPWu
3KHulCxXgvddFZvjq5/3x9rNmF/+r5JTOLdP+U3/W9N4AbMdPMOQura9v2h4n6n2
aGtZeR1652x6v7bckOwbnRXiHR/T96jlpxcjJ94i6/XXZ9esJvUhN3dJVf0v/kgm
XTa6gwdiRKjlBDyaPlw6ADoWFo2m2yL3ZlRf5isT8oWIHIMk3IU/gMWqfBE71g1s
gvDrcxCtoRfEYrWCSL1DrLyJWRLW0QmaAe5IGMrzMlNwp+2vxh/kEAuG6pZ2eJju
7G3Q2vMbFg87a8m4TopMqsXeoZpXhZs4ZsmffrbP57vugun/6I/QvoICnYa4zO1O
2H5olX5QDK0KVax8kUIkzd2fm1i5QovfR1QYk0VTYejUXQhXk2xZdFR7bnz3yT27
RefDXUixryxSgJjhWyLEIAaO5Tt3XNpsfO5meozjsVm3qvXzTXRiy8I6sh3iWo2N
DAMENIJez1PCLTxrJUmRPM4h8B2fAoAB94rvAI1e2L4JxkPvTBLIcz3byNmyylT0
pt0KQZl+d7zlVhtYtuP+Wy5p0Hcx8io4s1jWqEifXZfObtJmWFB/E33/dFhpF0Gg
DXpnhHyTHYbM0+Q0ACbBcqIN2MusgWWbwXDVJLnGf+a3XLVAMOz8A1P5Nmo2wfMO
6ZavAFH9v6I28ISZIp66zJwDDnCVaXs+uAYEyloGPTZWnURsldLABqxj1/iR9WI9
ssHb/7IJ/h58z0xMrodU3vvQ5oTi2C+f4YwfmtTIZxFHp7vveLvbxdGlHbtZX7oY
+X8Ww4F5aMeden6xDKG3nFUgnISrrtf5hMCzvvSgzjAs9ygCVZlK+82dFDPERwwC
AZulgvsy/HIZyglNRZ5KkxGahFgSNxpxasN+sQLP6K8euokke8VaV48g/Z0MHNk0
2E8MjCO6JmUNQwDjhTo5gYSrBkYPsnKruj7NsGi9FFiWgDYvpkRibMBmIaFDPu1c
3DCSlppxDKTQTeGi0XvnLrXz+T2r0sck8BIgzqSS1ZT+7escPAXbNhiTVzmBDr/j
5gbbgnYsowz40+FsXIHtEYayyzSrOxtO12V6DVX5ZLBOkecDf2XAP5EbXoe+t1/4
fdxs2uJIEt+oDA5CoIaD5H1yUAvFwyudzOPaoLWO9XXNts2E2JaRgUYrSQtdXBh0
BHfGjdQIx2VdjkJHZt89eBFr7V8DaZKZLOuSUEFfVbEMH57dw0XTeULAi7bQ2N9s
mGOB3b0xIxXL4xFIqxfFu/tVTiHWizi1BFu2SmDK9unR+jeS3dsIa/f8LY4y+lx2
sptsi9dkShSCqS81N8U2wMBwsWfRq5iNQyv0wUDjRM09vnosD4QgaKFPSRz30DUj
kkyBdX+NVCPjMsrJi6A65lhcNyVJ3hPlCCIsyrr+gRFEgJGodbJJ5w==
-----END RSA PRIVATE KEY-----
```

2) The previous command generates a 2048-bit RSA key pair, encrypts them with a password you provide, and writes them to a file. You need to next extract the public key file. You will use this, for instance, on your web server to encrypt content so that it can only be read with the private key.
3) Export the RSA Public Key to a File
   a. type this command:
      ➢ *rsa -in private.pem -outform PEM -pubout -out public.pem*

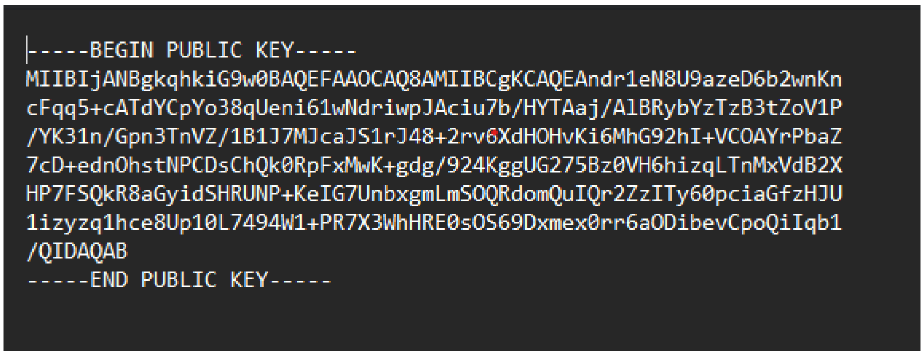


```
Verifying - Enter pass phrase for private.pem:

OpenSSL> rsa -in private.pem -outform PEM -pubout -out public.pem
Enter pass phrase for private.pem:

writing RSA key
```

4) Next open the public.pem and ensure that it starts with a: -----BEGIN PUBLIC KEY-----

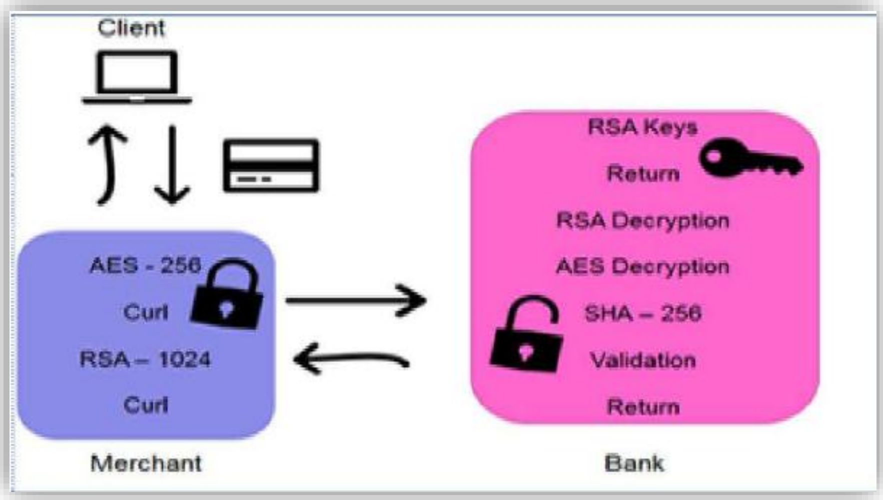


```
-----BEGIN PUBLIC KEY-----
MIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEAndr1eN8U9azeD6b2wnKn
cFqq5+cATdYCpYo38qUeni61wNdriwpJAciu7b/HYTAaj/AlBRybYzTzB3tZoV1P
/YK31n/Gpn3TnVZ/1B1J7MJcaJS1rJ48+2rv6XdHOHvKi6MhG92hI+VCOAYrPbaZ
7cD+ednOhstNPCDsChQk0RpFxMwK+gdg/924KggUG275Bz0VH6hizqLTnMxVdB2X
HP7FSQkR8aGyidSHRUNP+KeIG7UnbxgmLmSOQRdomQuIQr2ZzITy60pciaGfzHJU
1izyzq1hce8Up10L7494W1+PR7X3WhHRE0sOS69Dxmex0rr6aODibevCpoQiIqb1
/QIDAQAB
-----END PUBLIC KEY-----
```

Optimal method to enhance security

Nowadays online sites have different input fields in their payment processing page like some have Card no., Exp, CVV, some others have 3D secure pin instead. These differences create a variety of distinct vulnerabilities that allow attackers to extrapolate part of sensitive data of the same person from multiple sites. Therefore, it is recommended for all the merchant sites to use the same architecture for the payment process. We will do three important modifications to systems that exist already:

- Eradicate distributed guessing attack to most extent;
- Be implemented with the latest security transactions;
- Be easily updated later in the future.



Steps

1. User enters his/her credit card details on the merchant site. The data is sent to merchant server;
2. AES key and AES initialization vector for 256 bits are generated;
3. Generated keys are used to encrypt data acquired from the merchant site with AES and Chain Block Cyphering (CBC);
4. After the encryption, merchant server sends a request to bank server for an RSA 1024 bits public key;
5. AES key and AES vector are encrypted using RSA;
6. The encrypted message is sent to the bank server using **cURL** (*Client for URL: a library (**libcurl**) and command-line tool (**curl**) for transferring data using various network protocols*);
7. RSA decryption;

8. AES decryption;
9. Except the card number, remaining card details are encrypted using sha512, because in the bank database are encrypted using it
10. Encrypted data is compared with the one present in the database for a particular card number
11. If the card is verified a Boolean "*true*" is returned to merchant server and the transaction is successful
12. In order to avoid distributed guessing attacks, there is a limit imposed on invalid transactions.

Future improvements

Future work involves optimizing AES encryption using the Anubis S-box. While direct access to the AES implementation within the Stripe library is not possible, other libraries, such as PyCrypto, provide opportunities to explore and experiment with AES encryption. Additionally, creating a unified payment architecture could enhance security and streamline updates.

Conclusion

Online payment systems combine technology, policy, and human factors. While vulnerabilities remain, advances in cryptography, authentication, and data storage have improved resilience. Adopting layered security, standardizing architectures, and leveraging technologies like AI and blockchain can mitigate risks. As attackers evolve, proactive measures and innovation are essential to secure digital transactions.

Concise summary of results

The study demonstrated that replacing the Rijndael S-box in AES encryption with the Anubis S-box significantly improved performance metrics, including reduced memory usage, power consumption, and encryption time. These results were achieved using optimized and synthesized VHDL code for hardware implementation, validated through simulations on FPGA platforms. Additionally, a proposed prototype employing the optimized AES and RSA encryption within a unified payment architecture effectively addressed distributed guessing attacks and ensured secure data transmission and storage. By integrating these cryptographic enhancements, the system achieved would improve resilience against cyber threats, proving the feasibility of combining robust encryption with practical implementation to advance the security of online payment systems.

References

Abdul Jaleel, J., & Assis, A. (2013). Optimization of AES Encryption Algorithm with S-Box. In *International Journal of Engineering Research and Technology* (Vol. 6, Issue 2). http://www.irphouse.com

Bajaj, R. D., & Gokhale, U. M. (2016). AES ALGORITHM FOR ENCRYPTION. *International Journal of Latest Research in Engineering and Technology (IJLRET) Www.Ijlret.Com* ||, *02*, 63–68. www.ijlret.com

Bobba, S. (2020). Enhancing the Security of Online Card Payment System. *International Journal of Advanced Trends in Computer Science and Engineering*, *9*(2), 2055–2059. https://doi.org/10.30534/ijatcse/2020/178922020

Easttom, C. (2017). The RSA Algorithm Explored. In *International Journal of Innovative Research in Information Security (IJIRIS)* (Vol. 01). www.ijiris.com