# Aristotle University of Thessaloniki
# MSc: Interactive Software and Hardware Technologies
# Internet-of-Things Security Project

*Student*: Prodromos Polychroniadis
*AEM:* 9

Based on the description of the project given to us, the security requirements to be followed for the Smart City IoT system are:

1. The things need to be **authenticated** in order to **establish a secure channel** to the cloud server. Otherwise the provided data **cannot be trusted**.
2. The cloud server must be **authenticated** to the things for **preventing** unauthorized users from observing sensitive information.

I will study and compare three different public-key crypto systems, in order to determine which one is more suitable for the IoT system described, along with its security requirements, while also respecting its resource-usage limitations. Those three crypto systems are:

- The RSA crypto system
- The elliptic-curve cryptography
- The ElGamal encryption system for public-key cryptography

## Specifics

In this section of the project, I will analyze and describe the specifics of each of the crypto systems mentioned above and examine their security benefits, while also comparing them with each other. The conclusions section will follow after that.

## RSA (Rivest–Shamir–Adleman) crypto system

The RSA algorithm is an asymmetric cryptography algorithm. Therefore, it is, in general, much more expensive to compute, both in resources and time, than symmetric encryption algorithms. However, RSA being asymmetric, it uses longer keys than symmetric encryption and so it provides better security. While the longer key length in itself is not so much a disadvantage, it contributes to slower encryption speed.

The idea of RSA is based on the fact that it is extremely difficult for someone to factorize a large integer. The public key consists of two numbers where one number is the multiplication of two large prime numbers. Also, the private key is derived from the same two prime numbers. So if somebody can factorize the large number, the private key is compromised. Therefore encryption strength totally lies on the key size and if we double or triple the key size, the strength of encryption increases exponentially. RSA keys can be typically 1024 or 2048 bits long, but experts believe that 1024 bit keys could be broken in the future. However, until now it seems to be an infeasible task.

The RSA crypto system consists of four steps. These are the **Key Generation**, **Key Distribution**, **Encryption** and **Decryption**. Let's take a better look at how these steps work in order to achieve RSA encryption.

1. **Key Gen:** Firstly, we must choose two distinct and random prime numbers (p and q). We must keep those numbers secret at all times. Moreover, to make factoring harder for the attacker, those two numbers can be similar in magnitude but different in length by a few digits. Next, we compute the number n which is equal to p*q. The number n is what we called above as key length and so it expressed in bits. It is released as part of the public key and is used as the modulus for both keys (public and private). After we compute n, we must compute λ(n), where λ is Carmichael's totient function and must be kept in secret. Next, we choose an integer $e$ so that $1 < e < \lambda(n)$ and GreatestCommonDivisor($e$, $\lambda(n)$) = 1. If both those relations are true, then e and $\lambda(n)$ are co-prime. The e number is then released as another part of the public key. Finally, we determine d as $d = e^{-1} \pmod{\lambda(n)}$ and we keep it secret as the private key exponent. After we have completed all those computations, we can make the following claims. The public key consists of the modulus n and the public exponent e. The private key consists of the private exponent d, which must be kept secret. The

p, q, and $\lambda(n)$ must also be kept secret, as stated above, because they can be used to calculate d from the start. In fact, they could and should all be discarded after d has been computed.

2. **Key distribution:** In order to use RSA for the communication between the things and the cloud server, the things must know the server's public key to encrypt their messages. The cloud server will use its own private key to decrypt them. So, the cloud server can transmit its public key (that consists of n and e calculated above) to the things. However, it must always keep its private key secret. The same can happen when the server tries to communicate messages (like firmware updates or other messages) to the things, so the reverse distribution can occur.
3. **Encryption:** After the things have the server's public-key, they are able to send messages to that server. In order to do this, the things must first turn their message (the plaintext) into an integer m, so that $0 \leq m < n$ is true, by using an agreed-upon reversible protocol. Then, they compute the ciphertext c, using the server's public key e, corresponding to $c = m^e \bmod n$. When this is successful, the things transmit c to the server. This of course can occur the other way around.
4. **Decryption:** The server can recover the message from c by using its private key exponent d, that we calculated in the first step. The computation of $c^d = m^{e*d} = m \bmod n$, gives us the message integer m. Given that integer, the server can recover the original message by using the agreed reversible protocol.

This description is generally the way the RSA algorithm works. However, there is one more security measure the algorithm can take into account. RSA can also be used to sign messages, in order to verify the origin of the messages to the receiver. Suppose the server wishes to send a signed message to a thing. It can use its own private key to do so. It produces a hash value of the message, raises it to the power of d mod n, and attaches it as a "signature" to the message. When the thing receives the signed message, it uses the same hash algorithm in conjunction with the server's public key. It raises the signature to the power of e mod n, and compares the resulting hash value with the message's actual hash value. If the two agree, he knows that the author of the message was in possession of the server's private key, and that the message has not been tampered with since.

## Elliptic-curve cryptography

Elliptical curve cryptography is a method of encoding data files so that only specific individuals can decode them. ECC is based on the mathematics of elliptic curves and uses the location of points on an elliptic curve to encrypt and decrypt information. ECC affords efficient implementation of wireless security features but has some disadvantages when compared with other cryptography techniques.

It employs a relatively short encryption key, a value that must be fed into the encryption algorithm to decode an encrypted message. This short key is faster and requires less computing power than other first-generation encryption public key algorithms. For example, a 160-bit ECC encryption key provides the same security as a 1024-bit RSA encryption key and can be up to 15 times faster, depending on the platform on which it is implemented. The advantages of ECC over RSA are particularly important in wireless devices, where computing power, memory and battery life are limited. Of course, this is useful knowledge to know as in our Smart City IoT system, the things are these kind of devices.

One of the main disadvantages of ECC is that it increases the size of the encrypted message significantly more than RSA encryption. Furthermore, the ECC algorithm is more complex and more difficult to implement than RSA, which increases the likelihood of implementation errors, thereby reducing the security of the algorithm.

Elliptic curves are applicable for key agreement, digital signatures, pseudo-random generators and other tasks. Indirectly, they can be used for encryption by combining the key agreement with a symmetric encryption scheme. Several discrete logarithm-based protocols have been adapted to elliptic curves. One of the most well-established elliptic-curve algorithms is the Elliptic Curve Diffie–Hellman (ECDH) key agreement scheme, which is based on the Diffie–Hellman scheme. Many other ECC algorithms are based on this one, so I will continue with further analysis of the ECDH.

In general, the Elliptic-curve Diffie–Hellman is an anonymous key agreement protocol that allows two parties, each having an elliptic-curve public–private key pair, to establish a shared secret over an insecure channel and, thus, be able to communicate. Let’s see how to key establishment of this algorithm works.

We use ECDH when the things wants to establish a shared key with the server (or vice-versa), but the only channel available for them may be eavesdropped by a third party. Initially, the domain parameters (**Setup**), indicated by the elliptic curve mathematics, must be agreed upon. Also, each party must have a key pair (**Key Gen**) suitable for elliptic curve cryptography, consisting of a private key d (a randomly selected integer in the interval [1, n-1] and a public key represented by a point Q (where Q = d*G, that is, the result of adding G to itself d times). Let the server's key pair be (ds, Qs) and the things' key pair be (dt, Qt). Each party must know the other party's public key prior to execution of the protocol.

The things compute point (xk, yk) = dt*Qt. The server computes point (xk, yk) = ds*Qs. The shared secret is xk (the x coordinate of the point). Most standardized protocols based on ECDH derive a symmetric key from xk using some hash-based key derivation function. The shared secret calculated by both parties is equal.

The only information about their private key that the things initially expose is their public key. So, no party other than the things can determine their private key, unless that party can solve the elliptic curve discrete logarithm problem, which is extremely difficult if the implementation is correct. The server's private key is similarly secure. No party other than the things or the server can compute the shared secret, unless that party can solve the elliptic curve Diffie–Hellman problem.

The public keys are either static (and trusted, say via a certificate) or ephemeral. Ephemeral keys are temporary and not necessarily authenticated, so if authentication is desired, authenticity assurances must be obtained by other means. Authentication is necessary to avoid man-in-the-middle attacks. If one of either the server's or the things' public keys is static, then man-in-the-middle attacks are thwarted. Static public keys provide neither forward secrecy nor key-compromise impersonation resilience, among other advanced security properties. Holders of static private keys should validate the other public key, and should apply a secure key derivation function to the raw Diffie–Hellman shared secret to avoid leaking information about the static private key. We can authenticate Diffie Hellman keys either:

- by including the static Diffie Hellman parameters of one party (the server in TLS) into a certificate which is signed by a trusted authority (where the static parameters stay the same for all key exchanges) or

- by requiring the server to sign the ephemeral Diffie Hellman key. Thereby, the public verification key corresponding to the signing key is put into a certificate which is signed by a trusted authority.

If the correct authentication doesn’t happen then the messages exchanged are susceptible to person-in-the-middle attacks. While the shared secret may be used directly as a key, it is often desirable to hash the secret to remove weak bits due to the Diffie–Hellman exchange.

One feature that can be achieved for confidential communication when exchanging keys using a key exchange protocol such as ECDH is forward secrecy, which you will not have when using asymmetric encryption and sending encrypted messages under a fixed ElGamal (next crypto system to be analyzed) public key to a receiver. As said before, ECC algorithms like the  Elliptic-curve Diffie–Hellman crypto system can also be used to implement and acquire digital signatures, in order to achieve identity authentication.

## ElGamal encryption system - public-key cryptography

In cryptography, the ElGamal encryption system is an asymmetric key encryption algorithm for public-key cryptography which is based on the Diffie–Hellman key exchange, that we mentioned above in the ECC section.  ElGamal allows Elliptic Curves and so has some of the advantages of the ECC.  In contrast to the ECDH, ElGamal encryption is a public key encryption scheme and may be seen as a non-interactive ECDH key exchange.

The ElGamal encryption system has benefits, which however, are more interesting when using ElGamal encryption as a building block for "larger" cryptographic protocols. For instance:

- It is an encryption scheme which allows multiplying plaintext hidden inside of ciphertexts and when using the homomorphic property with an encryption of the identity element 1 of the group allows to **publicly re-randomize ElGamal**

**ciphertexts** and obtain new ciphertexts for the same message which are unlinkable to the original ciphertexts.

- There are efficient honest-verifier zero-knowledge proofs of knowledge to prove properties of ElGamal ciphertexts without revealing the plaintext, like equality of plaintexts.
- ElGamal can be used to construct a threshold cryptosystem. As an example, there may be n parties holding shares of the secret decryption key and a ciphertext can only be decrypted if at least *k* of these *n* parties are involved in the decryption process but fewer then *t* parties will fail in decrypting.

In comparison to the RSA algorithm, ElGamal is better suited to resource-constrained environments like the IoT system we want to secure, as it happens with the ECC algorithms. Also, the encrypted message in RSA is always longer than the ElGamal one.

The algorithm used consists of three components. These are the **Key Generation**, **Encryption** and **Decryption**. Let's take a better look at how these steps work in order to achieve the ElGamal cryptography.

1. **Key Generation:** The things (or the server) generates a key pair by generating an efficient description of a cyclic group G of order q with generator g. Let e represent the unit element of the group G. We choose an integer x randomly from the interval [1, q-1] and we compute $h = g^x$. The public key then consists of the values (G,q,g,h). The things publish this public key and retain x as their private key, which must be kept secret.
2. **Encryption:** The server encrypts a message M to the things under their public key we stated above by mapping the message to an element m of G using a reversible mapping function. Then the server chooses an integer y randomly from the interval [1, q-1] and compute $s = h^y$. This is called the shared secret. In addition to that, the server computes $c1 = g^y$ and c2 = m*s. Finally, the server sends the ciphertext (c1,c2) to the things.
3. **Decryption:** After the encryption, the things decrypt the ciphertext (c1,c2) with their private key x by computing $s = c1^x$. Since $c1 = g^y$, $c1^x=g^{xy}=h^y$ which is the same shared secret that was used by the server in encryption. Then, we compute $s^{-1}$. If G is

a subgroup of a multiplicative group of integers mod n, the modular multiplicative inverse can be computed using the Extended Euclidean Algorithm. Then, we compute $m = c2*s^{-1}$. This calculation produces the original message m, because c2=m*s. Finally, we map m back to the plaintext message M.

## Conclusions

In the previous sections I examined the advantages and disadvantages between the RSA algorithm, the ECC and the ElGamal encryption system, while also explaining their algorithmic steps in detail.

As stated by the project and explained before, we need to construct a secure IoT Smart City system. Besides the security aspect, though, we must also take into consideration the limitations that a system like this will have in regards to resources like battery like, execution times, memory shortage and others.

As a result I must exclude RSA from the possible solutions as even if it provides security advantages, it will be very costly to be used by the things of the Smart City leading to malfunctions or unwanted behaviors. In general, RSA is never recommended for devices with such limitations as the ones described.

So, at this point, we have to decide between ECC and the ElGamal encryption system. My preference is leaning towards the ECC and ECDH protocol. Its main and only disadvantage in my opinion is the complexity of its correct implementation, but considering that we can have expertise in this aspect, I believe that this obstacle can be easily surpassed. Both ECC and ElGamal respect the device limitations to be used in the Smart City. The advantages that give ECC the edge against ElGamal are the security properties of forward secrecy, identity-based encryption and digital signatures. As I believe and have understood, all the other advantages of ECC are shared with the ElGamal encryption system as the ElGamal algorithm is based of ECC and DH as stated in its description. So just that three

security properties made me decide that ECC is the better option, even if there are not many other significant differences between ECC-ECDH and ElGamal.