Security Library For A Smart Home Network

Cody Ebert

August 14, 2017

1 Abstract

With the addition of embedded technologies into everyday electronic items, many homes have become “smart” homes. However, these items are highly susceptible to exploitation due to their lack of security features. In order to help protect their sensitive data, either in transit or stored on the network, we created a customizable security library that can be utilized by these devices. We implemented our security library on the Java ME embedded platform and utilized Bouncy Castle’s open source cryptography distribution in order to create a security library that is flexible, portable, and resource efficient. Our experimental results suggest that our security library provides functionality for key exchange, encryption, authentication, and data integrity for these underpowered devices.

2 Introduction

The emergence of the Internet of Things (IoT) has led to many innovations in Smart Home technology. For example, many home owners now manage appliances throughout their home over a network such as the Internet. The increased popularity of such devices has created a market boom for embedded technologies, but unfortunately, due to the limited nature of IoT devices (e.g. low cost, development, and functionality), the security of IoT is also limited. On the other hand, many IoT devices still require similar security guarantees to that of regular end hosts.

A customizable security library implemented in Java could provide needed security functionalities such as key exchange, authentication, encryption, and integrity to a multitude of embedded devices. Java has been a preferred language for many developers around the world since the 1990’s due to the platforms robustness, ease of understanding, security, and platform independence. The Java platform is also widely distributed which allows for ease of integration. However, Java has occasionally been criticized as being too computationally heavy, but with recent developments in the Java Micro Edition (Java ME), embedded devices can now utilize Java’s distribution. Java ME has low system requirements but still supports major components of the more popular Java Standard Edition (Java SE) platform. The Java ME platform runs on embedded devices with a memory footprint as low as 128 KB RAM and 1 MB ROM making it viable for many IoT devices present in a Smart Home. While SSL is available in Java ME by default, we implemented a new security library that utilizes Bouncy Castle’s distributed security library for Java ME. Bouncy Castle’s unique implementation allows for future customization.

When two parties wish to communicate over a network securely, they often expect confidentiality, authentication, and integrity to be provided by the system. Public-key cryptography, also known as asymmetric cryptography, uses a mathematically related key pair that consists of a public key used for encryption and a private key used for decryption. Along with the ability to encrypt and decrypt data between two parties, public-key cryptography can be used as a form of authentication and integrity. The ability to reveal the public portion of the key pair while keeping security intact is a major benefit to public key cryptography, but there are also drawbacks. Asymmetric encryption is expensive and slow, whereas it generally proves to be much more efficient to use symmetric cryptography (a single shared secret key) for standard data encryption operations. Symmetric keys can also provide authentication and integrity, similar to asymmetric cryptography. However, symmetric key encryption has its own drawbacks. Most importantly, it is required that two parties share the same symmetric key. This poses the problem of securely transferring a shared key across a network where it may become exposed to eavesdropping. In order to combat the weaknesses of both forms of cryptography, it's typical to take advantage of both asymmetric and symmetric cryptography. Typically, asymmetric cryptography is usually used to security send a pre-generated symmetric key through a key exchange protocol such as Diffie-Hellman. Subsequently, this shared secret encrypts/decrypts data exchanged between the two parties through symmetric key cryptography. Using a combination of asymmetric and symmetric cryptography negates the shortcomings when either scheme is used alone.

This project focuses on keeping overhead to a minimum while providing confidentiality, authentication, and integrity. Therefore, we utilized a combination of symmetric and asymmetric cryptography to create a customizable security library implemented in Java. Since Elliptic Curve Cryptography (ECC) currently maintains the best resource usage for asymmetric cryptography, particularly key strength to size ratio, our system utilizes ECC. Though ECC has a wide variety of applications, this project only utilizes the generation of EC key-pairs and a version of the Diffie-Hellman protocol that uses ECC (known as ECDH) to exchange symmetric keys. The EC key-pairs and ECDH are meant to establish a symmetric key in order to leverage the speed advantage of symmetric cryptographyn and provide encryption, authentication, and integrity. A default key size of 128 bits would be used to take advantage of a smaller key size while still maintaining security. The encryption scheme used is AES-128, and authentication and integrity are provided by using either the AES Galois Counter Mode or by simply attaching a Message Authentication Code to a message.

When developing in an IoT environment, security is crucial. Having the ability to access easy to use, efficient functions that provide key exchange, encryption, authentication, and integrity is vital in an environment that is so vulnerable to attack. In the next section, we further discuss and describe ECDH, AES encryption, Message Authentication Codes, and Galois Counter mode, as well as discuss their importance to our security library.

3 Implementation

3.1 Key Exchange

While cryptography is an ever evolving field that seeks a balance between cryptographic integrity and system efficiency, the core of modern cryptography utilizes highly difficult or resource intensive mathematical problems. Historically, the backbone of public-key systems relies on the difficulty of prime factorization; there is simply no efficient algorithm to compute the factors of large numbers containing large prime numbers (though, this may change in the future with advances in mathematics or quantum computing). That is, it is quite easy to multiply two large numbers but very difficult to factor one large number. Thus, the difficulty of breaking such a cryptographic scheme correlates to the size of these numbers.

In 1978, Ronald Rivest, Adi Shamir, and Leonard Adleman developed the first public-key based encryption algorithm known as RSA. The security of RSA relies on the Discrete Log Problem (DLP). The discrete logarithm problem is defined as given a group G, a generator g of the group and an

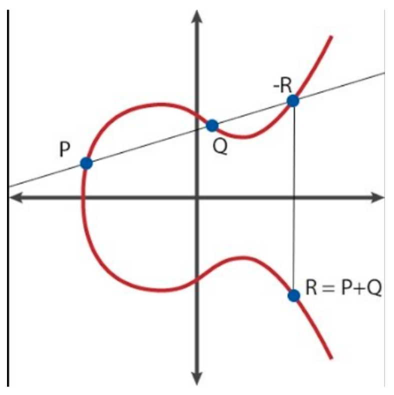
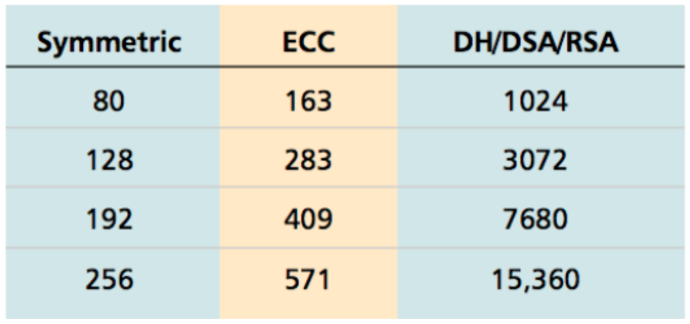


Figure 1: Example of addition group law on integer solutions on an elliptic curve



Symmetric	ECC	DH/DSA/RSA
80	163	1024
128	283	3072
192	409	7680
256	571	15,360

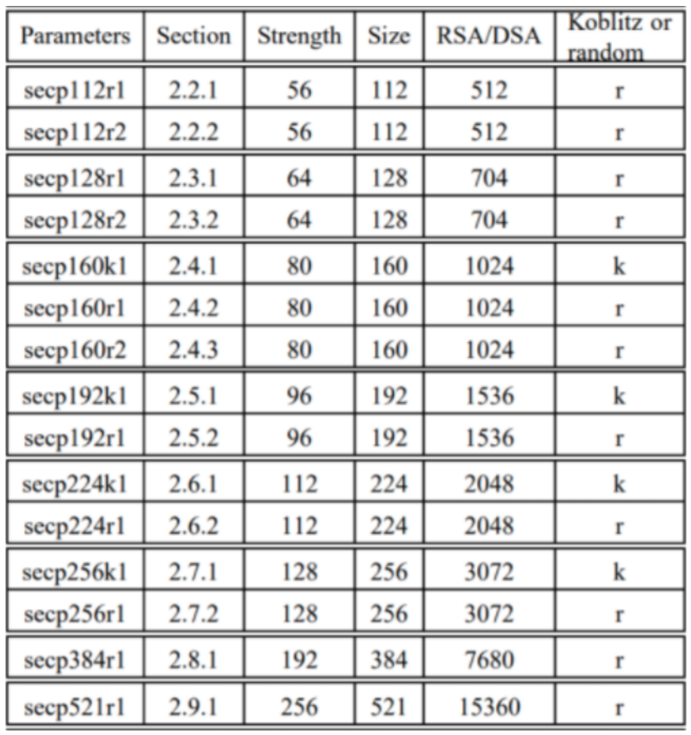
Table 1: Key Sizes for equivalent Security Levels

element h of G, find the discrete logarithm to the base g of h in the group G. The problem's difficulty depends on the group used, therefore certain groups are preferred when used for cryptography. For example, RSA utilizes the group of integers modulo a large number (100 digits long) n, under integer multiplication.

However, in 1985 two mathematicians independently proposed the use of elliptic curves in cryptography (ECC). An elliptic curve E_k is defined over a field K such that its characteristic doesn't equal 2 or 3. For $a, b \in K$, an elliptic curve is the solution set of (x, y) pairs in K^2 that satisfy the following equation: $y^2 = x^3 + ax + b$. After much mathematical research, points on elliptic curves can surprisingly comprise a group, and therefore, ECC can utilize the DLP similar to RSA, only with a different group. Namely, one can create an addition group law under the integer points of an elliptic curve. With this group in mind, one can integrate elliptic curves into other cryptographic schemes that utilize other forms of the DLP (e.g. Diffie-Hellman key exchange or digital signatures).

ECC has a variety of applications in cryptography including elliptic curve Diffie-Hellman, digital signatures, and encryption. The preeminent advantage to using ECC is the ratio of key size to security strength. In Table 1 we can see a comparison of security strength and key size between symmetric keys, ECC keys, and standard public-key schemes like RSA. The size of keys grows dramatically for equivalent strength in standard RSA keys while ECC key strength to key size ratio grows slowly. While ECC doesn't have the same strength to size ratio as typical symmetric cryptography this is a remarkable improvement over previous public-key crypto-systems such as RSA.

A study at Sun Microsystems Laboratory found that ECC-160 point multiplication outperforms the RSA-1024 private-key operation by an order of magnitude and is within a factor of 2 of the



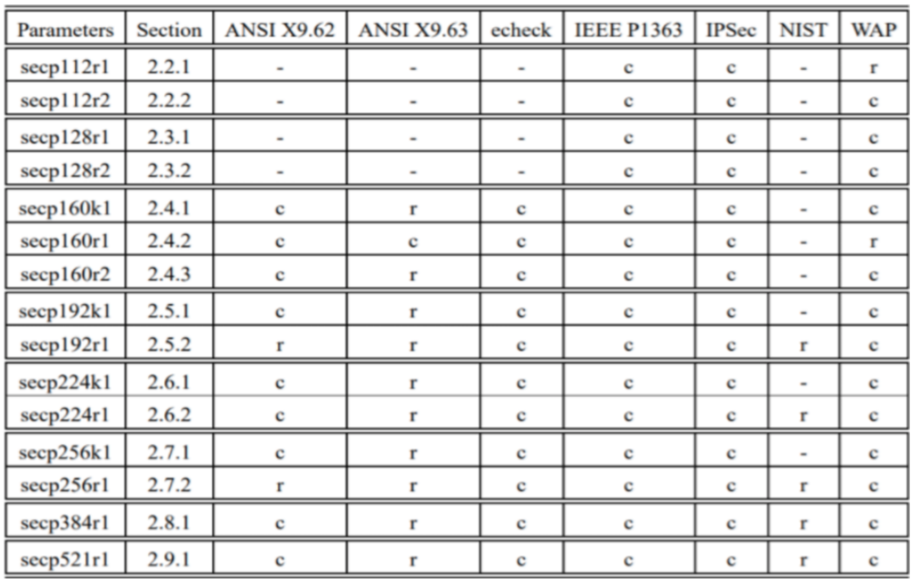
Parameters	Section	Strength	Size	RSA/DSA	Koblitz or random
secp112r1	2.2.1	56	112	512	r
secp112r2	2.2.2	56	112	512	r
secp128r1	2.3.1	64	128	704	r
secp128r2	2.3.2	64	128	704	r
secp160k1	2.4.1	80	160	1024	k
secp160r1	2.4.2	80	160	1024	r
secp160r2	2.4.3	80	160	1024	r
secp192k1	2.5.1	96	192	1536	k
secp192r1	2.5.2	96	192	1536	r
secp224k1	2.6.1	112	224	2048	k
secp224r1	2.6.2	112	224	2048	r
secp256k1	2.7.1	128	256	3072	k
secp256r1	2.7.2	128	256	3072	r
secp384r1	2.8.1	192	384	7680	r
secp521r1	2.9.1	256	521	15360	r

Table 2: Curve Properties

RSA-1024 public-key operation. This key size to strength ratio is beneficial to environments that have limited resource, such as IoT environments, since smaller keys allows for less memory usage, less data to transmit, and faster operations. There are a variety of curves to choose from when running or implementing ECC. The particular curve and naming convention denotes the size of the field over which the curve is defined. The curve name corresponds to the size of the key pair as well as the secret generated (through a Diffie-Hellman key exchange). Table 2 indicates a number of approved named curves as well as their properties and strength compared to size and standard RSA/DSA keys.

All ECC functionalities in this project were implemented using open source code created by Bouncy Castle. As previously mentioned, Bouncy Castle has a distribution for the Java ME platform and proved to be a good candidate for implementation since the Bouncy Castle distribution provided much of the functionality (particularly in the way of ECC) that was heavily lacking the standard Java ME security library. As discussed earlier, choosing the right elliptic curve is important. In this project the default chosen was secp128r1. Smaller keys are better in restricted environments but there must also not be a major loss to security. Table 3 indicates the achievability of certain government standards by each curve. A *c* indicates the curve complies with the standard, while an *r* indicates it is explicitly recommended.

While the secp128r1 curve is not always recommended, it does comply with several standards, and furthermore, keys generated with this curve produce a 128 bit key when used with ECDH for use as a symmetric key. It is recommended that symmetric key sizes do not fall below 80 bits, but 128 bit symmetric keys provide more security while remaining relatively small. Additionally, 128 bit keys allow us to use AES-128 rather than an encryption algorithm like TDEA. Although, in some circumstances it is viable to use secp128r1, it may be wise to use a curve with a larger field such as secp160r1 to generate the initial EC key-pair, then simply use the secret generated via ECDH to



Parameters	Section	ANSI X9.62	ANSI X9.63	echeck	IEEE P1363	IPSec	NIST	WAP
secp112r1	2.2.1	-	-	-	c	c	-	r
secp112r2	2.2.2	-	-	-	c	c	-	c
secp128r1	2.3.1	-	-	-	c	c	-	c
secp128r2	2.3.2	-	-	-	c	c	-	c
secp160k1	2.4.1	c	r	c	c	c	-	c
secp160r1	2.4.2	c	c	c	c	c	-	r
secp160r2	2.4.3	c	r	c	c	c	-	c
secp192k1	2.5.1	c	r	c	c	c	-	c
secp192r1	2.5.2	r	r	c	c	c	r	c
secp224k1	2.6.1	c	r	c	c	c	-	c
secp224r1	2.6.2	c	r	c	c	c	r	c
secp256k1	2.7.1	c	r	c	c	c	-	c
secp256r1	2.7.2	r	r	c	c	c	r	c
secp384r1	2.8.1	c	r	c	c	c	r	c
secp521r1	2.9.1	c	r	c	c	c	r	c

Table 3: Curves recommended by Trusted Standards

derive a smaller key (128 bits). Using a curve such as secp160r1 provides larger key-pairs, and in turn more security, which may be more beneficial for other forms of ECC, like the Elliptic Curve Digital Signature Algorithm (ECDSA). However, since the main goal of our implementation is to exchange a symmetric key to be used for encryption and authentication, it is more vital to focus on the strengths of this key. Therefore, as mentioned, the secp128r1 curve was used because it was the simplest way to produce a 128 bit secret.

Functionality related to key generation, key pair generation, and key derivation, is provided in our KeyGen class. The implementation for EC key-pair generation in this project is straightforward to use. Most parameters are taken care of without the need for users input, especially if the user simply uses the default parameters (i.e. secp128r1 curve, 128 bit key). There is room for users to choose a different curve and key size if desired. Bouncy Castle's implementation of ECC offers a number of different curves to be used, including all those in Table 2 and 3, as well as others not mentioned in the recommended curve list. It should be noted that the key size in this library refers to the secret size generated from the ECDH exchange. If a curve other than secp128r1 is being used the user must input the secret size(in bytes, not bits) that corresponds to the selected curve, otherwise an error will be thrown when attempting to generate a secret. If a distinct curve is desired, the proper secret size can be calculated by converting the field size to bytes. For example, the secp160r1 curve would generate a 20 byte secret, but if a smaller symmetric key is desired, it is possible to take this 20 byte secret generated and derive a smaller key using the key derivation function available in the class.

Key exchange was implemented using a version of Diffie-Hellman using ECC in the form of ECDH. The Diffie-Hellman key exchange protocol is popular in most crypto-systems. It's a rather clever protocol in which each party chooses a private key, say G or K, calculates the corresponding public key G_x or K_y , and sends this public key to the other party. Once received each party can then calculate a secret number by combining the other party's public key with their own private key such as $(Gx)y$ or (Ky)x. This provides both parties with a secret key shared between the two that, while revealing the public keys, keeps the private keys secret, rendering the shared secret difficult to compute by malicious parties listening in. This is described by in Figure 2.

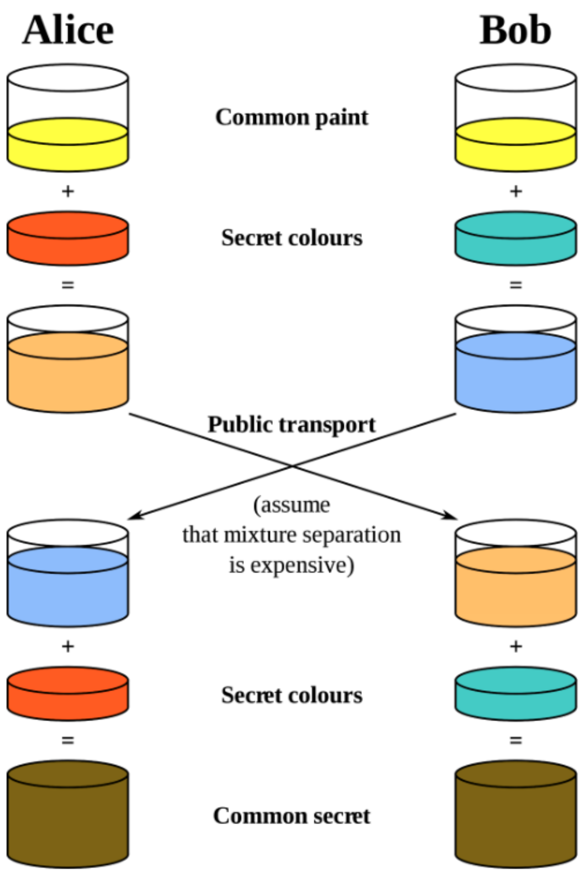


Figure 2: Diffie-Hellman Key Exchange

Due to the nature of the algorithm, it can be easily converted to use ECC, as mentioned previously. In this project, Diffie-Hellman key exchange was implemented using ECC with Bouncy Castles open source deployment. The ECDH implemented in this project is easy to use, straightforward, and can be used in conjunction with the key generation functionality. Users simply need to provide two AsymmetricKeyParameters (a class provided by Bouncy Castle). The parameters provided should correspond to the server's private key and the client's public key (as is typical in a Diffie-Hellman exchange). If the user has their key in bytes, functionality is provided to convert a byte array into the proper parameter. Executing ECDH will return a shared secret to the user in the form of a byte array.

It's natural to touch on ephemeral versus static key pairs as this has an effect on performance and security. An ephemeral key pair is generated for one time use. The key pair will only be used during one connection and discarded afterward, this means that the identity of the sender is not necessarily verified and should be verified in another way. Static key pairs are kept for a longer period of time and can be used to verify the identity of the sender. In this library using ephemeral or static key pairs is left to the user, although, the functionality directly provides the ability for a ephemeral scheme and therefore using a static scheme requires extra help for initial authentication (this can be accomplished with a certificate chain but this is not in the scope of this project). Ephemeral keys offer the benefit of forward security, which is the idea that (since keys are only used once per connection) sensitive data from previous connections cannot be compromised. However, using ephemeral keys has its drawbacks as well, including more overhead from key generation, and the lack of authentication. Since the ephemeral keys are only used once, they must be accompanied by another form of authentication during the initial connection. In this project, this is left up to the user. Static keys on the other hand allow malicious persons to compromise all past data that was passed using that given key pair. Though this lowers the security of the scheme, it also means that the same key is used multiple times which relieves some overhead from key generation, as well as providing the authentication needed at the initial connection. Either scheme has benefits and drawbacks but still perform the function of providing a shared secret between two parties.

3.2 Encryption

While it is possible to use a shared secret directly as a key, it is recommended to use this shared secret to derive a new key. However, this requires both parties to use the same key derivation formula. Key derivation is the act of taking a secret value, such as a password, shared secret, or passphrase and creating one or more secret keys. The act of deriving a key allows a party to take one key that may not meet proper format and converting it to the correct size. Most key derivations use hash functions to produce a given output size. Occasionally, they add additional non-secret parameters to add entropy to the derived key; this is known as key diversification. In this project, a simple key derivation formula was devised and is relatively easy to use. The MD5 hash algorithm is used by default due to its relative security and an output of 128 bits. This function takes both the parties public keys used in the exchange, and hashes them with the shared secret. It's important to note that the public keys should be updated at the same time for both parties. This means that parties on both sides of the communication must put the public keys in the hash algorithm in the same order. Failure to do this will result in a different derived key and any attempted secure communication will fail. Luckily, our function rearranges the public keys in the correct order for the users. The key derivation function will produce any size key as specified by the user. This means that a variable length secret can be converted into a specified length simply by specifying the desired length in the parameters and given the proper digest.

Once a symmetric key has been exchanged and/or derived users may begin exchanging encrypted data. Perhaps the most common symmetric ciphers used for encryption and decryption in modern systems is AES, also known as the Advanced Encryption Standard. It was first published by Vincent Rijmen and Joan Daemen in 1998, and it superseded the Data Encryption Standard (DES). Originally adopted by the U.S. government, AES is now widely spread as the standard cipher for symmetric encryption. In this project, it sufficed to simply use AES since our default key size matched the block size for AES at 128 bits. Additionally, Java ME's standard library also provides the ability to implement AES encryption. The implementation in this project provides users with a relatively simple interface for encrypting data with AES given plaintext and a symmetric key. Given a plaintext and key in the form of byte arrays, the function will automatically pad the plaintext to fit the proper multiples of the block size of 128 bits (16 bytes) when needed. Once padded, the plaintext is encrypted using the provided key and returns ciphertext in the form of a byte array. Similarly, the decryption function takes ciphertext and the corresponding key and returns the plaintext.

3.3 Authentication and Integrity

Finally, our security library provides functionality for authentication and data integrity. Many systems provide authentication and integrity through the form of digital signatures using algorithms such as Digital Signature Algorithm (DSA) or its elliptic curve counterpart (ECDSA). Digital signatures use asymmetric key-pairs to "sign" messages and offer three applications: authentication,

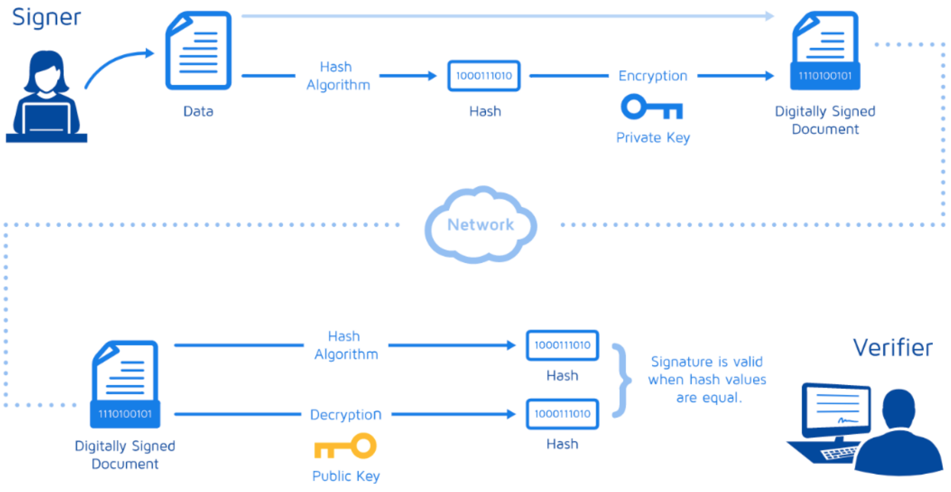


Figure 3: Digital Signature Process

$$HMAC(K,m) = H\Big((K' \oplus opad) \mid\mid H\big((K' \oplus ipad) \mid\mid m\big)\Big)$$

Figure 4: HMAC Algorithm

integrity, and nonrepudiation. To achieve authentication a private key is used by the sender to encrypt a given message, and receivers can use the sender's public key to verify this signature by decrypting the message. Typically, the message is first hashed to compress the data and add the ability to check data integrity. Once the message is hashed, it is then encrypted using the sender's private key. The sender sends both the original data (possibly encrypted using a symmetric key), along with the signed hash. The receiver then decrypts the hashed message, and hashes the original message. If the hash sent by the sender and the hash computed by the receiver match, then the data has remained intact and has not be tampered with by malicious forces. Digital signatures also offer the benefit of non-repudiation. This is the idea that a sender cannot later deny sending any given message since, by design, the sender must use their private key. Additionally, those who have possession of the sender's public key cannot maliciously fake a digital signature from the sender.

Although digital signatures are a powerful scheme for providing authenticity and integrity, they also add quite a bit of overhead. As previously discussed, asymmetric encryption and decryption take an enormous amount of time and resources compared to symmetric cryptography. Fortunately, we can achieve much of the same functionality with symmetric cryptography in the form of a Message Authentication Code (MAC). MACs can provide authentication and integrity similar to digital signatures but do not provide non-repudiation. However, message authentication codes are based on symmetric cryptography rather than asymmetric cryptography and are therefore much faster. Most digital signatures are used prior to the establishment of a shared secret or when non repudiation is critical. In an IoT environment, speed and resource efficiency are critical, so it's natural to implement MACs over digital signatures, even given the lack of non-repudiation.

Message authentication codes offer authentication and integrity in much the same way as digital signatures but rather than using a key pair, they use a shared secret or symmetric key. Typically a message is generated and encrypted using AES, this encrypted message is then pushed through the given MAC function. There are a number of ways to implement MAC functions; typically imple-

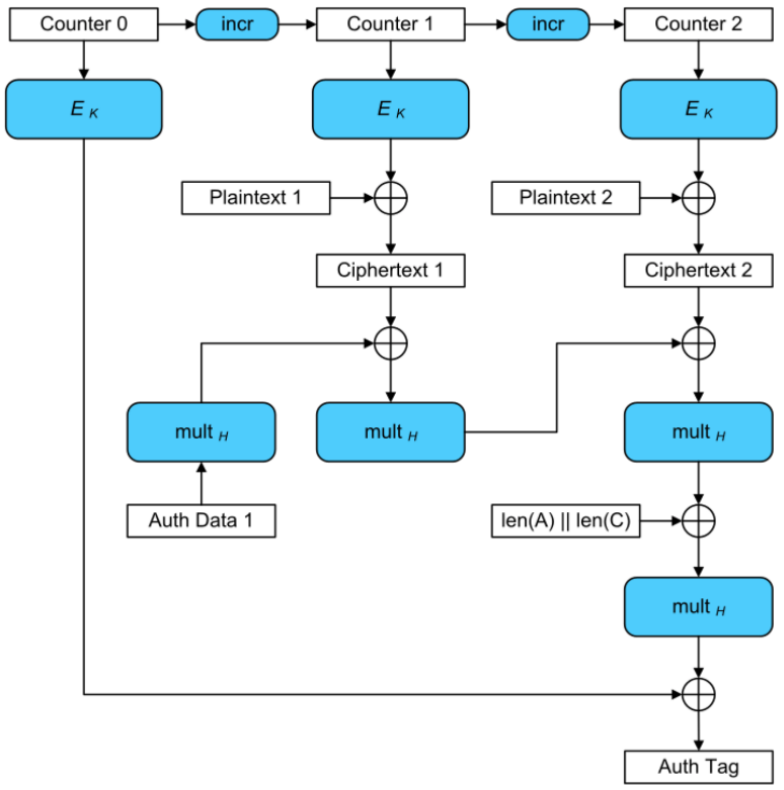


Figure 5: Galois Counter Mode Operations

mented using hash based algorithms, known as HMACs, or cipher block algorithms, CMACs. Our security library implements the MAC in the form of a HMAC. While some hardware optimizations may make CMACs faster than HMACs, hashing algorithms typically add less overhead than cipher block algorithms. Therefore, HMACs are more suitable for an embedded environment.

A user inputs a symmetric key and message into the HMAC, and the key is truncated or padded to the proper block size. This key is then XORed with the value $0x5c$. The non-XORed key is then also XORed with the value $0x36$, and the result is concatenated with the input message. This concatenation is then hashed using the specified hash algorithm. The resulting hash is then concatenated with the initial XORed key value, and once again hashed using the specified algorithm. The final result produces the HMAC. To check the message's authenticity and integrity, the end user must recompute the HMAC using the shared symmetric key along with the received message. If the result is the same as the HMAC received, then the end user knows the message is valid.

It's important to note that the order in which a message is encrypted and the MAC is formed makes a difference in security. By encrypting a message and passing the cipher text into the MAC algorithm rather than passing the plaintext means that integrity is provided not only for the plaintext but also for the ciphertext. Additionally, if the encryption is malleable then the MAC will indicate any tampering with the ciphertext. If plaintext is passed rather than ciphertext, the MAC does not provide integrity for the ciphertext and it is theoretically possible to alter a message to appear valid when it is not if the given the cipher scheme is malleable. Though the implementation in this library does not explicitly execute this way, users who access the library should remember this fact when integrating these functions in their programs.

Our implementation is relatively simple to use since only the key and data are required as input for authentication. Similar to most functionalities of this project, the HMAC uses the open source Bouncy Castle distribution. This data should be encrypted prior to being MACed. The default hash function used is MD5 since its output is 128 bits and we wish to have smaller MACs while operating in embedded environments. While MD5 is the default, a different digest may be used at the user's discretion. All HMACs are returned as a byte array and can be easily verified by supplying the MAC, shared key, and original message. The verification function uses the HMAC function to recompute MAC and compare it to the received MAC. When they are the same, the function returns true, otherwise it returns false.

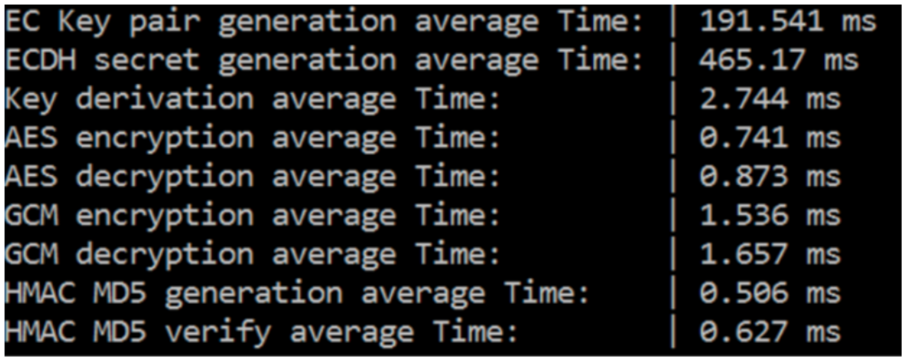
Another methodology to provide authentication and integrity is to utilize the Galois Counter Mode (GCM) mode of operation for a symmetric block cipher. This mode of operation can be used in symmetric block ciphers to add authentication and integrity while it encrypts the data. This differs from simply using a MAC function because it is a mode of operation used in *conjunction* with a cipher algorithm. The block size for GCM is 128 bits which is perfect for this library as it matches the default key size used. GCM requires users to supply: a secret key, a nonce, plaintext, and the additional data to be authenticated. The secret key is typically the symmetric shared secret or derived key between two parties. A nonce must be randomized for each message and is used as a counter. The nonce does not necessarily need to remain secret and can therefore be passed as plain text. However, it is crucial to make sure the nonce is not reused as this will depreciate the level of security. The plaintext required is simply any data that requires encryption, authentication, and integrity checks. Additional data is data that does not need to be encrypted but should be checked for authenticity and integrity. Running GCM will return both encrypted data along with an authentication tag (MAC).

Galois counter mode works like many other counter modes associated with encryption. However, GCM provides a MAC as well. The operation of GCM is as follows. First a nonce is supplied in conjunction with a counter. This value is encrypted using the supplied key and XORed with the first block of the plaintext to produce a block ciphertext. This ciphertext will be used in our output but is also used to compute a MAC during operation. To compute the MAC, this block of ciphertext is multiplied with the key in what is known as a Galois field. The result is XORed with the next cipher block that is computed in the same way as the first. Often, the additional (non-encrypted) data is multiplied and XORed with the ciphertext before the next operation. This continues until all plaintext is encrypted and both ciphertext and a MAC are produced. Decryption works similarly, the user must provide the ciphertext, secret key, along with the authentication tag to decrypt and authenticate the data transmitted. Below is a diagram outlining the operation of GCM.

In this project GCM was implemented using Bouncy Castle's open source distribution. The functionality implemented in this library allows users to provide a key, plaintext, nonce and associated data. Although a nonce is required, functionality for generating a nonce is provided in the class. The encryption function will return a single byte array containing ciphertext and the authentication tag. Decryption is simple to use as well, end users must provide the proper secret key, ciphertext, associated data, and nonce. If the MAC does not match the MAC provided, an error is thrown. The GCM encryption can be used in place of simply using AES and generating a separate MAC to provide integrity and authentication.

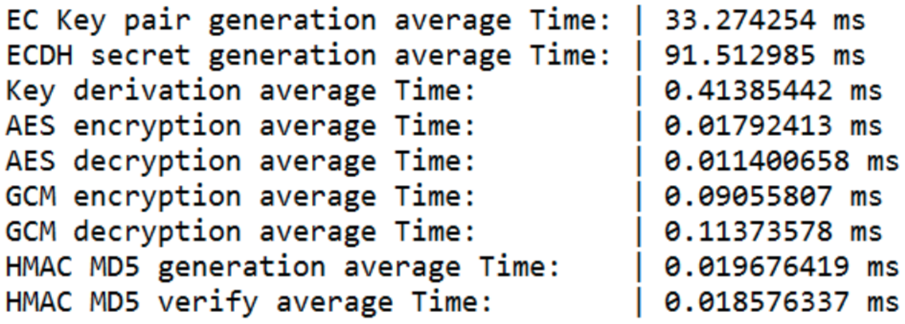
3.4 Results And Analysis

We ran tests using the functions developed for our library on both an Intel Galileo Gen 2 and a Macbook pro to see the difference in runtime and determine if the speeds on the Galileo are feasible for real world applications. The results of our tests are as follows in Figure 7 and Figure 8



```
EC Key pair generation average Time:   | 191.541 ms
ECDH secret generation average Time:   | 465.17 ms
Key derivation average Time:           | 2.744 ms
AES encryption average Time:           | 0.741 ms
AES decryption average Time:           | 0.873 ms
GCM encryption average Time:           | 1.536 ms
GCM decryption average Time:           | 1.657 ms
HMAC MD5 generation average Time:      | 0.506 ms
HMAC MD5 verify average Time:          | 0.627 ms
```

Figure 6: Average runtime for functions from our library run on the Galileo Gen 2



```
EC Key pair generation average Time:   | 33.274254 ms
ECDH secret generation average Time:   | 91.512985 ms
Key derivation average Time:           | 0.41385442 ms
AES encryption average Time:           | 0.01792413 ms
AES decryption average Time:           | 0.011400658 ms
GCM encryption average Time:           | 0.09055807 ms
GCM decryption average Time:           | 0.11373578 ms
HMAC MD5 generation average Time:      | 0.019676419 ms
HMAC MD5 verify average Time:          | 0.018576337 ms
```

Figure 7: Average runtime for functions from our library run on the Macbook Pro

respectively.

As expected the Galileo Gen 2 did not perform as well as the Macbook Pro. However, the numbers for the Galileo are not overly burdensome. The ECDH secret generation took the longest amount of time by far but was still a reasonable speed. If secrets are reused for a period of time the overhead of secret generation will be marginal. EC Key pair generation had the second longest runtime and surprisingly was less than the secret generation. Similar to the usage of a shared secret (symmetric key), prolonged usage of a key pair may help cut down overhead but depending on the application using ephemeral key pairs may be viable. Another surprising result is the runtime of AES in GCM mode versus the combination of AES and HMAC. Since GCM is renowned for its speed and efficiency it is surprising that the average run time of GCM is still greater than the runtime for both AES encryption and the generation of an HMAC combined.

To conclude, while run times on the Galileo were significantly worse than those on the Macbook, this was expected. It was surprising to find that many of these algorithms ran at the speeds measured considering the Galileo's low resources. Although the Galileo is more powerful than many IoT devices, it is not unreasonable to assume Smart Home appliances could use this library as an addition to security.

4 Case Study: Fire Detection System

Often, a large variety of embedded devices will comprise a smart home. For example, a smart home may contain smart light bulbs, a smart refrigerator, or even a smart oven. In order to concretely show the efficacy of our security library, we introduced it to a common household IoT device: a Fire Detection System. This was implemented using an Intel Galileo Gen 2 board with a Grove flame

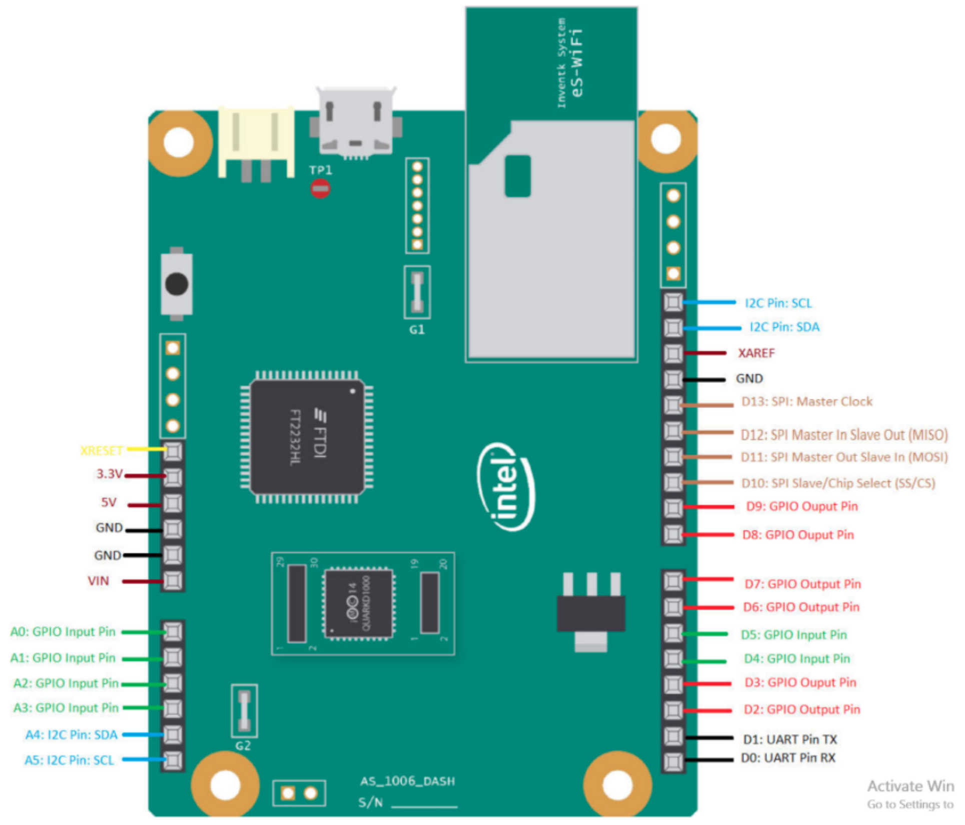


Figure 8: Board Layout of Galileo Gen2 5

detection sensor and a Grove temperature sensor. Our Fire Detection System utilizes sensors and our security library to monitor a home environment and securely transmit data to a host server to alert the homeowner of a possible fire. Using this application, we were able to compare the efficiency of the different encryption schemes in our security library.

A case study like the Fire Detection System requires a number of different I/O operations including sensor reading and socket connections. All I/O operations required specific permissions to be added to the Application Descriptor file in the JAD in order to be executed. Each permission requires specific resources to be added as well as the particular action to be made on a given resource. Resources and actions range widely but add an additional layer of security for the application.

Implementing a case study that required sensors meant that we would need to access and interpret data picked up by the sensors. Typically, this is done using Java classes made available by the manufacturer. However, this is not the case for Java ME; many of the sensor classes provided have dependencies to the Java SE library and therefore cannot be used with the Java ME platform. Fortunately, Java ME does support classes for reading data from most pins. The Galileo Gen 2 has a variety of pins that can be used for sensors or data output(such as LED or sound). There are a number of different types of I/O depending on the type of data that needs to be received or transmitted.

The Galileo Gen 2 distribution of Java ME supports the GPIOPin classes, which allows for general purpose pin configuration and I/O, as well as I2C and others. Since the flame sensor uses digital signal (on or off) to alert of any flames detected simply using the GPIOPin class to read the data sufficed. It's important to note that while the standard distribution for Java ME (used

	MacBook Pro	Galileo Gen 2
Message: GCM	0.61 ms	305 ms
Message: AES/HMAC	1.8 ms	219 ms
Message: No encryption	0.359	3 ms
Key Exchange	0.63 ms	443 ms

Table 4: Case Study Results

in IDE's and emulators during development) does support the ADC(analog to digital conversion) classes, the distribution for Galileo Gen 2 does not. This meant reading from the pin would be significantly harder to do with Java. By nature, the ADC pins, as well as others, output data into a file that can be accessed through the command line, for example the output of the A0 pin could be read by looking at: /sys/bus/iio/devices/iio:device0/in_voltage0_raw. Although Java has the ability to read from files, Java ME's JRE has different root directories making it more difficult to track down the proper file paths. The easiest way was to set up a symlink from the output file of the sensor to a file in Java ME JRE's root directory. Once the symlink was set up it was easy to set the correct file read permissions in the Application descriptor to allow for reading the data, rather than adding another permission related to ADC or GPIO pins.

Once the ability to gather information from the temperature and flame sensors had been established the data needed to be transmitted to the user. To accomplish this, client/server and message classes were created to facilitate data transfer. A mock up protocol was also created in order to transfer vital data that wasn't required to be encrypted, such as message lengths, key lengths, types of messages, and encryption schemes being used. Without a mockup protocol it would be difficult for server/client to agree on encryption schemes(i.e. GCM vs AES). One should note, this protocol should not be used for real world applications and was simply created for testing the case study. Much of the case study was designed to be run using 128 bit symmetric keys and the curve secp128r1 for key generation and may work differently if other curves or key sizes are used.

We created server, client, and message classes to better organize the cast study and send data between a central node and the Fire Detection System. The central node (in this case a Macbook Pro was used) would run the server class awaiting connection from the client (Fire Detection System). The classes support four types of messages. First, if a key exchange has not taken place and a symmetric key is not yet established between the two parties the client will send it's public key to the server. The server will respond by sending its public key and both parties will compute a shared secret which will then be derived into a symmetric key.

Additional communication can take place using either no encryption, AES encryption with an HMAC for authentication and integrity, or AES/GCM which will also provided encryption, authentication and message integrity. Below are the findings.

Comparing these numbers to the results found in the previous section Results and Analysis, it is clear that there is a significant amount of overhead added with the protocol and network usage. In both cases for the Mac and Galileo the key exchange did not include key generation as it is assumed a key has already been generated. It is particularly strange that the Galileo took significantly longer to send a key but this may be due simply to the payload size. The biggest surprise is how fast using no encryption is compared to using either GCM or AES/HMAC. While encryption adds overhead, looking at the previously stated results made it seem like the overheard wouldn't be this significant.

In an application like this it may be viable to deal with the overhead from encryption, authentication, and integrity. However, more vital, lower resource devices may suffer greatly from the implementation of these functions if the device is required to talk over a network frequently.

4.1 Implementation Issues

There were a number of unanticipated issues that delayed the final product by up to a month. Many of these issues related back directly to Oracle's distribution of the Java ME platform where fixes or workarounds were few and far between. The first, and perhaps greatest, issue was related to Oracle's Java ME SDK distribution. In order to develop Java applications for embedded devices through Java ME, you are required to download and install a number of tools, namely the Java ME SDK. This SDK contains a number of useful and required tools for development, including a device manager and embedded emulators. The SDK is only distributed for Windows and has plugins for only two IDEs: NetBeans and Eclipse.

After installing both the SDK and the plugins for Eclipse you can create a Java ME project and are required to choose a embedded emulator configuration profile to test your application. A number of emulators are provided by the SDK and are managed by the device manager. However, contrary to the functionally outlined in the documentation, the IDE didn't recognize any of the emulator profiles and therefore would not allow for the creation of a Java ME project. After a number of days searching for a fix turned up nothing and trying different versions of both IDEs and the SDK with no success, an alternative workaround was sought. According to Oracle's documentation Java ME supported a subset of the libraries found in Java SE so it seemed logical to develop using only this subset of libraries in the Java SE environment. This turned out to be an unwise decision because, in fact, the libraries themselves contained a subset of the required classes and many of the classes had been reworked for Java ME.

In short, a project developed in Java SE would not run in the Java ME environment. Looking back this seemed obvious, but at the time the list of options seemed short. Much of the developed code would have to be reworked and rewritten to run on Java ME which presented a set back, not only because much of the code was useless but also because Java ME's subset of the Java's Security library was missing a number of classes that were vital to the project. Luckily, Bouncy Castle, a well known distributor of open source security libraries, had a deployment of their security API specifically for Java ME.

At this point, since the Java ME wasn't working with Eclipse or Netbeans, the project was developed in Intellij. However, this proved to be a source of its own issues since, while Intellij could run Java using the Java ME runtime environment, the IDE failed to build JAR files properly and consistently produced corrupt JAR files. Building the JAR files from the command line also failed to produce working outputs. At this point it seemed wise to return to trying to get the Java ME SDK to work properly. After reinstalling all the proper tools and scouring for answers, a fix was found. There was a recent post indicating a problem with the device manager provided by the Java ME SDK. During deployment a configuration file buried deep in the Java directory path was missing a number of lines that were vital to the operation of the device manager. Additionally the device manager was required for Java ME SDK to work at all, since almost the entire project, from the configuration files to the libraries used in the distribution, were all based on the configuration of the emulators. As expected, appending the missing lines to the configuration file caused the device manager to run properly. After more research, it was apparent that this bug had been cataloged but no official fix had been released and would not be released until Java ME 9.

Documentation for the Intel Galileo Gen 2 and Java ME were also lacking in general, and documentation specific to Java ME was very hard to find or decipher. For example, figuring out how to set permissions for a Java ME project, or what Java libraries are actually supported in this particular distribution were not made obvious. In particular the documentation that related the

Galileo to the Java ME distribution was difficult to find. While in part this is partially the fault of Oracle, there is almost no documented cases of other projects developed on the Galileo using Java ME. Clearly, Oracle has intended Java ME's use on the Galileo since they have a specific distribution for it, but either few people have used it or they ran into far fewer issues than we did on this project.

5 Conclusion

The introduction of smart devices into our homes may increase convenience, but security and privacy may also be effected. Smaller devices with fewer resources may not be capable of certain security mitigations, and therefore, finding a balance between security and efficiency is a difficult task. In order to bring additional security to IoT environments, we utilized Java Micro Edition and Bouncy Castle to create a customizable security library for key exchange, key generation, encryption, authentication, and integrity. Elliptic curve cryptography offers a faster, more efficient way to generate symmetric keys, so we utilized ECDH to generate a shared secret and create a shared symmetric key even for devices that have few resources. Further, AES encryption with a relatively small key of 128 bits helps IoT environments encrypt quickly and communicate securely. Authentication and integrity are also important for well rounded security in an IoT environment, but schemes like digital signatures which use asymmetric cryptography may be unsuitable for a low resource environment, so we utilized MACs and authentication tags in our security library. Our symmetric key MACs offer both the security and speed needed for authentication and integrity in an IoT environment. In our case study, we found that not only do these security functions run on IoT devices, but the overhead is even less than expected. While these functionalities add overhead to common operations, this library may be a possible solution for certain IoT devices. Usage of this library may bring the currently lacking required security to many smart home environments while offering a customizable and easy to use interface.

References

[1] C. Dong. Math in network security: A crash course. [Online]. Available: https://www.doc.ic.ac.uk/~mrh/330tutor/

[2] N. Gura, A. Patel, A. Wander, H. Eberle, and S. C. Shantz, "Comparing elliptic curve cryptography and rsa on 8-bit cpus." Springer.

[3] S. E. C. Certicom, "Sec 2: Recommended elliptic curve domain parameters," *Proceeding of Standards for Efficient Cryptography, Version*, vol. 1, 2000.

[4] E. Rescorla, "Diffie-hellman key agreement method," 1999.

[5] R. L. Rivest, A. Shamir, and L. Adleman, "A method for obtaining digital signatures and public-key cryptosystems," *Communications of the ACM*, vol. 21, no. 2, pp. 120–126, 1978.

[6] Understanding digital signatures. [Online]. Available: https://www.docusign.com/how-it-works/electronic-signature/digital-signature/digital-signature-faq

[7] M. Bellare and C. Namprempre, "Authenticated encryption: Relations among notions and analysis of the generic composition paradigm," *Advances in CryptologyASIACRYPT 2000*, pp. 531–545, 2000.

[8] M. Bellare, "Keying hash functions for message authentication." Springer.

[9] D. McGrew and J. Viega, "The galois/counter mode of operation (gcm)."

[10] Overview of oracle java me embedded permissions. [Online]. Available: http://docs.oracle.com/javame/8.1/me-dev-guide/security.htm

[11] (2016) Intel galileo gen2device i/o preconfigured list. [Online]. Available: https://docs.oracle.com/javame/8.3/get-started-galileo/device-i-o-preconfigured-list.htm